Aspektorientierte Software-Entwicklung

unter besonderer Berücksichtigung der begrifflichen Zusammenhänge und der Einbettung in den Entwicklungsprozess

Diplomarbeit im Fach Informatik
vorgelegt von
Irene Bonomo-Kappeler
CH-5430 Wettingen
Matrikelnummer 99-706-806

Angefertigt am
Institut für Informatik der
Universität Zürich
Prof. Dr. M. Glinz

Betreuer: S. Meier
Abgabe der Arbeit: 1. August 2004

Zusammenfassung

In konventionellen Software-Systemen können die *Core Concerns* (dt. Kernfunktionalität) für sich allein sauber modularisiert werden. In jedem System gibt es jedoch auch *Crosscutting Concerns* (z.B. Autorisierung, Logging etc.), welche die Core Concerns quer schneiden (engl. *crosscut*). Sie lassen sich nicht eindeutig einem Modul zuordnen, sondern sind über den ganzen Code verstreut. Dadurch verhindern sie in konventionellen Software-Systemen eine wirklich saubere Modularisierung und beeinträchtigen die Verständlichkeit, Pflege, Wiederverwendbarkeit und (Rück-) Verfolgbarkeit. Die *Aspektorientierung* ist ein relativ junges Forschungsgebiet, das zum Ziel hat, Methoden und Sprachen zu entwickeln, mit denen auch Crosscutting Concerns sauber modularisiert werden können. Das Ziel dieser Arbeit ist es, den Stand der Forschung auf dem Gebiet der Aspektorientierung aufzuzeigen, eine einheitliche Begriffswelt zu schaffen und die Aspektorientierung in den Entwicklungsprozess einzubetten.

Abstract

In traditional software systems, the core concerns alone can be cleanly modularized. However, in every system there are crosscutting concerns (e.g. authorization, logging etc.) which crosscut the core concerns. They cannot be assigned to a single module, but are scattered throughout the code. By that, they prevent the traditional software systems from being cleanly modularized and affect their comprehensibility, evolution, reusability and traceability. Aspect-orientation is a relatively new research area which aims at the development of methods and languages which allow to cleanly modularize crosscutting concerns as well. The goal of this paper is to give an overview of the research results, contribute to a clear understanding of the most important concepts and embed aspect-orientation into the software development process.

Dank

Meinem Betreuer, Silvio Meier, danke ich herzlich für die mustergültige Betreuung, die vielen Denkanstösse, die kritischen Fragen und das Durchlesen der Entwürfe dieser Arbeit. Was das wissenschaftliche Arbeiten betrifft, habe ich viel von ihm gelernt. Ich wünsche ihm im Hinblick auf seine Dissertation alles Gute.

Meinem Mann, Alessandro Bonomo, danke ich von ganzem Herzen für die liebevolle und aufbauende Begleitung während der gesamten Studienzeit. Ohne ihn hätte ich das Abenteuer Studium niemals gewagt.

Meiner langjährigen Arbeitgeberin, der UBS AG, danke ich für die finanzielle Unterstützung und für die Flexibilität in Bezug auf meine Arbeitszeiten während der letzten knapp fünf Jahre. Ich war froh, dass ich während der vier Monate, in denen diese Arbeit entstand, zu Hause bleiben konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Einleitung

Der Einleitungsteil beschreibt die Motivation, die Aufgabenstellung und die Gliederung dieser Arbeit. Ausserdem wird der Umgang mit den englischen Fachbegriffen erklärt.

1. Motivation

Hauptsächliches Ziel des Software Engineering ist die termingerechte und kostengünstige Entwicklung und Pflege qualitativ hochwertiger Software. Die Erreichung dieser Ziele erfordert möglichst gut modularisierte Software mit einer möglichst geringen Komplexität.

In einem konventionellen[1] Software-System können die *Core Concerns*[2] (dt. Kernfunktionalität) für sich allein betrachtet nach den Regeln der Kunst (Schach, 1999)[3] sauber in Module gegliedert werden. In jedem System gibt es jedoch auch Concerns (z.B. Sicherheit, Performance, Fehlerbehandlung), welche die Kernfunktionalität quer schneiden (engl. *crosscut*) und sich folglich nicht eindeutig einem Modul zuordnen lassen. Dies führt dazu, dass Fragmente solcher *Crosscutting Concerns* unkorreliert (fehlende Kohäsion) und ungekapselt über den ganzen Code verstreut sind. Diese Crosscutting Concerns sind es, die in konventionellen Software-Systemen eine wirklich saubere Modularisierung verhindern und die Verständlichkeit, Pflege, Wiederverwendbarkeit und (Rück-)Verfolgbarkeit beeinträchtigen. Ursache für dieses Problem ist die Tatsache, dass konventionelle Programmiersprachen die Systemdekomposition nur in einer Dimension zulassen. Man spricht in diesem Zusammenhang von der *dominanten Dekomposition*. Mit anderen Worten: ein natürlicherweise mehrdimensionales Problem muss eindimensional gelöst werden. Die Ansätze der *Aspektorientierung* sind ein Versuch, mehrdimensionale Lösungen für dieses Problem anzubieten.

Jeder aspektorientierte Entwicklungsprozess besteht aus den drei grundlegenden Phasen *Identifikation*, *Separation* und *Integration*:

Identifikation. In der Identifikationsphase werden die relevanten Core und Crosscutting Concerns mit Hilfe von verschiedenen Verfahren identifiziert.

Separation. In der Separationsphase geht es darum, alle Concerns unabhängig voneinander zu spezifizieren, bei Bedarf zu operationalisieren, modular zu entwerfen und zu implementieren. Das Stichwort hierzu heisst *Separation of Concerns* und wird auf Dijkstra (1976) zurückgeführt. Solcherart modular implementierte Core Concerns werden *Komponenten* genannt; modular implementierte Crosscutting Concerns werden als *Aspekte* bezeichnet. Auch in konventionellen Ansätzen sind durchaus Möglichkeiten zur Separation of Concerns vorhanden (z.B. im Bereich von nichtfunktionalen Anforderungen), jedoch beschränken sie sich dort auf die Identifikation, Spezifikation und Operationalisierung.

Daneben müssen in der Separationsphase auch die *Integrationsregeln* spezifiziert werden, nach denen die Concerns später zum Gesamtsystem zusammengefügt werden.

Integration. In der Integrationsphase werden die fertig implementierten Komponenten und Aspekte anhand der Integrationsregeln zum Gesamtsystem zusammengefügt. Speziell in der aspektorientierten Programmierung wird die Integration mit *Weaving* (dt. Weben) bezeichnet. In einigen Ansätzen wird die Integration auch *Komposition* (engl. composition) genannt, was keinesfalls mit der gleichnamigen Assoziation im UML-Klassendiagramm verwechselt werden darf.

[1] Unter „konventionell“ wird hier „nicht-aspektorientiert“ verstanden. Aus der Sicht der Aspektorientierung gehört auch ein objektorientiertes System zu den konventionellen Systemen!

[2] Concern kann auf Deutsch mit „Belang, Anliegen, Interesse“ übersetzt werden (Link Everything Online (LEO), 2004).

[3] Die Form der Literaturverweise entspricht dem Vorschlag von Deininger et al. (1992). Das Literaturverzeichnis ist im Anhang B zu finden. Verweise auf Internet-Seiten erfolgen mittels Fussnoten.

Ziel der Aspektorientierung ist es, die Integration im Entwicklungsprozess möglichst lange hinauszuzögern. Dadurch unterscheiden sich im Wesentlichen auch die konventionellen von den aspektorientierten Ansätzen. Bei den konventionellen Ansätzen erfolgt die Integration notgedrungen bereits in der Entwurfsphase, weil die entsprechenden Programmiersprachen keine Möglichkeiten besitzen, überlappende Concerns in mehreren Dimensionen zu implementieren. Bei den aspektorientierten Ansätzen bleiben die Concerns auch während der Entwurfs- und Implementierungsphase noch getrennt und werden – je nach Ansatz – zur Übersetzungs- oder zur Laufzeit integriert. Dies hat den Vorteil, dass die einzelnen Aspekte im Quellcode immer noch sauber modularisiert sind, was die Verständlichkeit, die Wiederverwendung und die (Rück-)Verfolgbarkeit erleichtert.

2. Aufgabenstellung

Die Aufgabenstellung für diese Diplomarbeit lautet wie folgt (Zitat):

2.1 Thema

„Der Stand der Forschung auf dem Gebiet der aspektorientierten Software-Entwicklung mit einem Fokus auf die Identifikation, Darstellung und Verwendung von Aspekten in den frühen Phasen des Software Engineering.“

2.2 Inhalt

„Aspektorientierte Software-Entwicklung (AOSD) ist ein sehr junges Gebiet des Software Engineering. Aspektorientierung ist definiert als die Berücksichtigung von Entwurfsentscheidungen, welche quer durch das ganze System getroffen werden müssen (crosscutting concerns).

Aspektorientierung als Paradigma im Software Engineering steht noch am Anfang der Entwicklung, wie etwa die objektorientierte Software-Entwicklung vor ca. 20 Jahren. Die meisten Forschungsaktivitäten fanden bisher auf dem Gebiet der aspektorientierten Programmierung (AOP) statt. Es hat sich gezeigt, dass bisher neue Paradigmen in der Software-Entwicklung meist ausgehend von der Programmierung in anderen Phasen adoptiert wurden. Dies dürfte auch bei der aspektorientierten Software-Entwicklung so sein. So ist in neuerer Zeit zu beobachten, dass z.B. auf dem Gebiet von aspektorientierten Architekturen (AOA) geforscht wird und auch auf dem Gebiet von aspektorientiertem Requirements Engineering (AORE).

Das Ziel dieser Arbeit ist es, den aktuellen Stand der Forschung auf dem Gebiet der Aspektorientierung zu untersuchen. Dazu sollen aus der Fülle der bisherigen Forschungsresultate die bedeutsamsten herausgegriffen und diskutiert werden. Ein besonderes Augenmerk soll dabei denjenigen Arbeiten gelten, welche sich in Richtung aspektorientiertem Requirements Engineering bzw. aspektorientiertem Architekturentwurf (werden im Forschungsgebiet *Early Aspects* genannt) bewegen.

In einem weiteren Schritt sollen aktuelle Trends und ungeklärte Fragen in diesem Forschungsgebiet herausgearbeitet werden. Eine Bewertung der Trends, Probleme und ungeklärten Fragen, sowie das Skizzieren von möglichen Lösungen bzw. Antworten runden diese Arbeit ab.“

Die Liste der Fragen ist im Anhang A zu finden.

3. Umgang mit englischen Fachbegriffen

Das Dilemma zwischen der Lesbarkeit eines deutschen Texts und der Verständlichkeit der Begriffe in einem englisch geprägten Fachgebiet wurde wie folgt zu lösen versucht:

Wo sich ein guter (einfacher, passender, verständlicher) deutscher Begriff anbietet, wird der englische Begriff auf Deutsch übersetzt, besonders dann, wenn sich der deutsche Begriff bereits etabliert hat. In solchen Fällen wird bei der erstmaligen Verwendung des Begriffs das englische Original in Klammern beigefügt. Beispiel: Anwendungsfall (engl. Use Case). In einigen Fällen werden der deutsche *und* der englische Begriff benutzt. Beispiel: von *Schnittstelle* ist die Rede, wenn es allgemein um Schnittstellen zwischen zwei Systemen geht, von *Interface* ist die Rede, wenn es sich konkret um Interfaces von objektorientierten Programmiersprachen (z.B. Java) handelt.

Wo sich kein guter deutscher Begriff anbietet, wird der englische Begriff beibehalten und bei der erstmaligen Verwendung kursiv gesetzt. Beispiele: *Crosscutting Concern*, *Scattering*. In die deutsche Fachsprache eingebürgerte englische Begriffe werden ebenfalls nicht übersetzt und wie deutsche Wörter behandelt. Beispiele: Debugging, Pooling. Auch englische Namen von Konzepten, Sprachen und Produkten werden nicht auf Deutsch übersetzt. Beispiele: Hyperspace, Aspect Moderator Framework, AspectJ. Nicht deklinierbare englische Adjektive werden in Anführungs- und Schlusszeichen gesetzt. Beispiel: „crosscutting“ Verhalten. Englische Begriffe im Genitiv Singular erhalten kein -s, da dies auf Englisch den Plural und nicht den Genitiv anzeigt. Beispiel: des Framework.

4. Gliederung

Der Hauptteil der Arbeit ist in vier Teile gegliedert: Zu Beginn werden im Teil I die wichtigsten Begriffe im Zusammenhang mit der Aspektorientierung geklärt. Der Teil II gibt anschliessend einen Überblick über den Stand der Forschung anhand einer Auswahl von aspektorientierten Ansätzen. Im Teil III wird die Einbettung der Aspektorientierung ins Software Engineering untersucht, indem Antworten auf die im Anhang A aufgeführten Fragen gesucht werden. Die skizzierten Lösungsansätze und die gewonnenen Erkenntnisse werden im Teil IV kritisch gewürdigt.

Teil I:
Die wichtigsten Begriffe der Aspekt-orientierung

Auch wenn (oder gerade weil) es sich bei der Aspektorientierung um ein junges Forschungsgebiet handelt, sollten zumindest die wichtigsten Begriffe sauber definiert sein und einheitlich verwendet werden. Dies ist in der Literatur keineswegs der Fall. Die Begriffe werden zwar – mit Ausnahmen – einigermassen einheitlich verwendet, aber auffallend selten überhaupt definiert. In der Literatur zur Aspektorientierung kommen die Begriffe *Concern*, *Anforderung* und *Aspekt* besonders häufig vor. Sie werden in den Kapiteln 5 bis 7 einzeln diskutiert, bevor das Kapitel 8 ihre Zusammenhänge aufzeigt. Für Concern und Aspekt wird ausserdem eine Begriffsdefinition vorgeschlagen.

5. Concerns

Das Wort *Concern* lässt sich auf Deutsch mit „Anliegen, Belang, Interesse“ übersetzen (Link Everything Online (LEO), 2004). Es geht nach Duden (1997) auf lat. concernere „beachten, berücksichtigen; betreffen, sich beziehen auf“, eigentlich „Unterschiedliches zusammenmischen“ (lat. con… „zusammen, mit“ und lat. cernere „unterscheiden“) zurück. Der Concern passt also auch aus etymologischer Sicht ausgezeichnet zur Aspektorientierung.

5.1 Begriffsdefinition

Obwohl der Concern-Begriff intuitiv verständlich ist, ist es schwierig, in der Literatur eine gute Definition zu finden. Häufig wird der Begriff nämlich nicht definiert, sondern lediglich anhand von Aufzählungen eingeführt, so zum Beispiel in Bergmans, Aksit (2001a): „… concerns, such as access control, synchronization …“. Folgende Definitionen sind zu finden:

> **Definition a)** *Concern*. Gegenstand der Betrachtung. Starkes Interesse oder Beachtung (engl. regard), üblicherweise aufgrund einer persönlichen Bindung oder Beziehung (Merriam-Webster, 2004).
>
> **Definition b)** *Concern*. Massgebliches Systemziel auf oberster Ebene, das üblicherweise aus den kritischen Geschäftszielen des Auftraggebers abgeleitet wird (Sommerville, Sawyer, 1997).
>
> **Definition c)** *Concern*. Irgendeine Sache, die an einem Software-System interessiert (Sutton, Rouvellou, 2002).
>
> **Definition d)** *Concern*. Belang (engl. interest), welcher die Entwicklung oder den Betrieb eines Systems betrifft, oder irgendein anderer Aspekt, der für einen oder mehrere Beteiligte (engl. stakeholder) kritisch oder sonst wichtig ist (IEEE, 2000).
>
> **Definition e)** *Concern*. Spezifische Anforderung oder Gesichtspunkt, welche(r) in einem Software-System behandelt werden muss, um die übergreifenden Systemziele zu erreichen (Laddad, 2003).
>
> **Definition f)** *Concern*. Irgendeine Sache, die an einem System interessiert, d.h. ein Ziel oder eine Menge von Eigenschaften, die das System erfüllen muss (Brito, Moreira, 2004).

Es ist positiv zu vermerken, dass sich die Definitionen nirgends widersprechen, sondern lediglich andere Schwerpunkte setzen. Die Beurteilung, welche Definition sich am besten eignet, erfolgt anhand der folgenden beiden Kriterien:

- **Herkunft**. Die Bedeutung des Worts Concern deutet darauf hin, dass Concerns etwas sind, wofür sich die an der Entstehung eines Systems Beteiligten im Allgemeinen und der Auftraggeber[4] im Besonderen interessieren, die sie persönlich betreffen und daher nicht unberührt lassen. Die Definition des Concern-Begriffs soll folglich eine Aussage darüber machen, *woher Concerns kommen*, nämlich dass Concerns aus Anliegen oder Anforderungen von Beteiligten resultieren.

[4] Mit Auftraggeber ist der Auftraggeber persönlich und/oder dessen Vertreter im Projekt gemeint.

- **Bezugspunkt**. Concerns werden nicht zum Selbstzweck definiert, sondern sind verbindliche Vorgaben für die Gestaltung eines Systems. Also soll die Definition des Concern-Begriffs auch eine Aussage darüber machen, *worauf sich Concerns beziehen*, in unserem Fall auf Software-Systeme. Im Sinne von Laddad (2003) ist ein Software-System nichts anderes als die Realisierung einer Menge von Concerns. Vorgaben, die sich nicht auf das System, sondern auf den Entwicklungsprozess beziehen (z.B. Kosten- und Zeitvorgaben) sind in diesem Sinne *keine* Concerns.

Alle der oben zitierten Definitionen erfüllen mindestens eines der beiden Kriterien. Favorisiert wird die Definition e), weil sie sowohl die Herkunft („Anforderung …, um die … Systemziele zu erreichen“) als auch den Bezugspunkt („Software-System“) umfasst. Den Definitionen a) und b) fehlt der Bezugspunkt, der Definition c) fehlt die Herkunft. Die Definition d) gibt den Bezugspunkt ungenau wieder („Entwicklung oder … Betrieb eines Systems“). Definition f) erfüllt wie e) auch beide Kriterien, ist aber weniger prägnant.

5.2 Kategorisierung von Concerns

Es ist schwierig, den Concern-Begriff sinnvoll zu kategorisieren, da er nicht ganz präzis definiert ist und es vom Auftraggeber abhängt, was er als Concern betrachtet und was nicht. Dies hat zur Folge, dass instabile Kategorien entstehen, die sich fallweise ändern. Aus diesem Grund wird von einer Kategorisierung mit Ausnahme der Aufteilung in *Core Concerns* und *Crosscutting Concerns* (siehe im Abschnitt 5.3) abgesehen.

Die folgende Kategorisierung illustriert, weshalb es problematisch ist, Concerns zu kategorisieren: Nach Araujo et al. (2002) gibt es Crosscutting Concerns auf der *Anforderungsebene* (z.B. Antwortzeit, Sicherheit) und Crosscutting Concerns auf der *Entwurfs-/Implementierungsebene* aufgrund von technologischen Einschränkungen (z.B. Fehlerbehandlung, Synchronisation). Die Grenze zwischen Anforderungen und Entwurfsentscheidungen ist fliessend und hängt vom Auftraggeber ab. Je nachdem, wie diese Grenze im Einzelfall gezogen wird, gehören die Concerns zur einen oder zur anderen Kategorie.

5.3 Core Concerns und Crosscutting Concerns

Laddad (2003) unterscheidet zwischen Core Concerns und Crosscutting Concerns.

> **Definition** *Core Concerns*. Realisieren die Kernfunktionalität eines Systems.
>
> **Definition a)** *Crosscutting Concerns*. Realisieren diejenigen Funktionen eines Systems, welche die Core Concerns oder andere Crosscutting Concerns quer schneiden, beispielsweise Authentisierung oder Logging.
>
> **Definition b)** *Crosscutting Concerns*. Zwei Concerns sind „crosscutting“, wenn sich Methoden dieser Concerns schneiden (Elrad et al., 2001).

Nach Elrad et al. (2001) ist es wichtig zu beachten, dass Concerns immer nur relativ zu einer bestimmten Dekomposition „crosscutting“ sind.

Die Unterscheidung zwischen Core und Crosscutting Concerns ist nicht in allen Ansätzen gleichermassen relevant. So werden zum Beispiel in der MDSOC (Multi-Dimensional Separation of Concerns) und Hyper/J (siehe im Abschnitt 9.6) alle Concerns ausdrücklich gleich behandelt, und es wird nicht zwischen Core und Crosscutting Concerns unterschieden. Solche Ansätze werden auch *symmetrisch* genannt, während Ansätze, die zwischen Core und Crosscutting Concerns unterscheiden, *asymmetrisch* genannt werden (siehe im Abschnitt 13.2).

In einem der Ansätze der aspektorientierten Anforderungstechnik, AORE (Aspect-oriented Requirements Engineering, siehe im Abschnitt 12.2), werden Crosscutting Concerns als *Aspektkandidaten* bezeichnet, ein Begriff, der zwar andernorts nicht üblich ist, aber durchaus als Synonym für Crosscutting Concerns verwendet werden kann, zumal er unverbraucht und mit keinerlei Bedeutung belegt ist. Bisweilen ist auch von *nicht-funktionalen Concerns* als Synonym für Crosscutting Concerns die Rede (z.B. in Brito, Moreira, 2004). Dieser Begriff ist irreführend und sollte vermieden werden, da Crosscutting Concerns oft funktionalen Charakter haben. Andernfalls wären sie gar nicht implementierbar. In Elrad, Filman, Bader (2001) werden die Crosscutting Concerns *aspektbezogene Anforderungen* genannt, was zwar gut klingt, aber aufgrund des im Kapitel 6 beschriebenen Unterschieds zwischen Anforderungen und Concerns ebenfalls zu vermeiden ist. Der Begriff *Kernfunktionalität* kann als Synonym bzw. deutsche Übersetzung von Core Concern verwendet werden.

5.4 Separation of Concerns

Die *Separation of Concerns* ist ein grundlegendes Software Engineering-Prinzip und ein Schlüssel zum Verständnis der Aspektorientierung. Dijkstra (1976) erwähnt den Begriff zum ersten Mal:

> „To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused. This is what I mean by 'focussing one's attention upon a certain aspect'; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of. I usually refer to it as 'a separation of concerns' …"

Eine sinnvolle Definition findet sich in Ossher, Tarr (2001):

> **Definition** *Separation of Concerns:* Fähigkeit, nur diejenigen Teile der Software zu identifizieren, kapseln und manipulieren, die für ein bestimmtes Konzept, ein bestimmtes Ziel oder einen bestimmten Zweck relevant sind.

In der Literatur wird die Erfindung des Begriffs *Separation of Concerns* oft Parnas (1972) zugeschrieben, obwohl er ihn in seinem Essay „On the Criteria To Be Used in Decomposing Systems into Modules" nicht explizit erwähnt. Dennoch lässt sich zweifellos ein Zusammenhang zwischen Parnas (1972) und der Separation of Concerns herstellen: Hauptthema des Essays ist die Modularisierung im Allgemeinen und das *Information Hiding* im Besonderen. Concerns sind im Prinzip nichts anderes als Module auf einer höheren Abstraktionsebene. Sie gehorchen den gleichen Gesetzen wie Module (siehe im Abschnitt 7.3): „Gute" Concerns haben wie „gute" Module einen möglichst starken inneren Zusammenhang (starke Kohäsion), sind möglichst unabhängig von anderen Concerns (schwache Kopplung), verknüpfen ihre Daten und die auf diesen Daten operierenden Funktionen (Kapselung) und verbergen ihre „Implementierungsdetails"[5] gegenüber anderen Concerns (Information Hiding). Da eine solcherart saubere Modularisierung eine wichtige Voraussetzung für die Umsetzung der Separation of Concerns ist, ist die Behauptung, Parnas habe die Separation of Concerns erfunden (nicht wörtlich, aber sinngemäss), nicht ganz falsch.

Die Separation of Concerns hat nicht nur Vorteile, sondern auch einen Nachteil: Die Gefahr besteht, dass dabei der Überblick über das Gesamtsystem verloren geht.

[5] Concerns werden nicht direkt implementiert. Dennoch ist Information Hiding auch bei Concerns sinnvoll: Von Interesse ist die Leistung, die ein Concern erbringt, und nicht die Art und Weise, wie er diese Leistung erbringt.

5.5 Scattering und Tangling

Um die Separation of Concerns in Software-Systemen zu erreichen, wäre es wünschenswert, wenn man die Concerns unabhängig voneinander spezifizieren, implementieren und erst möglichst spät zum Gesamtsystem zusammenfügen könnte. Betrachtet man die Core Concerns für sich allein, können sie dank der Objektorientierung sauber modularisiert werden. Kommen hingegen Cross-cutting Concerns hinzu – und dies ist in realen Software-Systemen kaum zu vermeiden – ist die Separation of Concerns auch mit den Mitteln der Objektorientierung nicht zu erreichen. Wenn Ossher, Tarr (2001) im Zusammenhang mit den Core Concerns von der *Tyrannei der dominanten Dekomposition* sprechen, meinen sie damit, dass die dominante (z.B. objektorientierte) Art der Systemdekomposition der Software eine Struktur aufzwingt, die es unmöglich macht, andere Arten von Concerns sauber zu modularisieren. Typischerweise entstehen dabei die folgenden beiden Probleme, *Scattering* und *Tangling*, welche die Separation of Concerns verletzen.

Scattering (dt. Verstreutheit) bezieht sich auf *einen einzelnen Concern* (z.B. Logging) und entsteht, wenn die Implementierung dieses Concern auf mehrere Module verteilt ist. Der Scattering-Begriff wird, wie so oft im Bereich der Aspektorientierung, in der Literatur häufig verwendet, jedoch selten definiert. Tarr et al. (1999) definieren ihn sinngemäss wie folgt:

> **Definition** *Scattering*. Ein einzelner Concern wirkt sich auf mehrere Entwurfs- und Code-Module aus.

Die Abbildung 5-1 illustriert das Scattering anhand eines Beispiels: Hier ist der Logging-Concern „scattered“, d.h. er ist in sämtlichen Modulen implementiert, die den Systemzustand verändern.

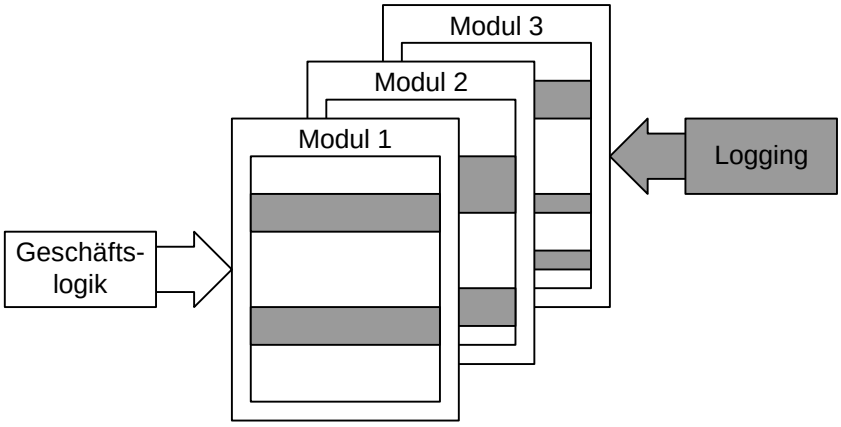


Abbildung 5-1: Scattering von Concerns

Scattering führt einerseits zu redundanten oder zumindest ähnlichen Code-Blöcken in allen betroffenen Modulen und damit zu einer schlechten Pflegbarkeit sowie zu einer hohen Fehleranfälligkeit bei Änderungen. Anderseits ist es kaum möglich zu verfolgen, in welchen Modulen eine bestimmte Anforderung implementiert ist.

Tangling (dt. Durcheinander) bezieht sich auf *ein einzelnes Modul* und entsteht, wenn dieses Modul die Implementierungen mehrerer (Fragmente von) Concerns enthält. Auch der Tangling-Begriff wird in der Literatur häufig verwendet, jedoch selten definiert. Die folgende Definition stammt wiederum von Tarr et al. (1999):

> **Definition** *Tangling*. Code-Fragmente, die mehrere Concerns betreffen, sind in einem einzelnen Modul vermischt.

Die Abbildung 5-2 zeigt ein Beispiel für Tangling: In diesem Modul sind neben der Geschäftslogik auch noch (Fragmente der) Persistenz- und Logging-Concerns implementiert.

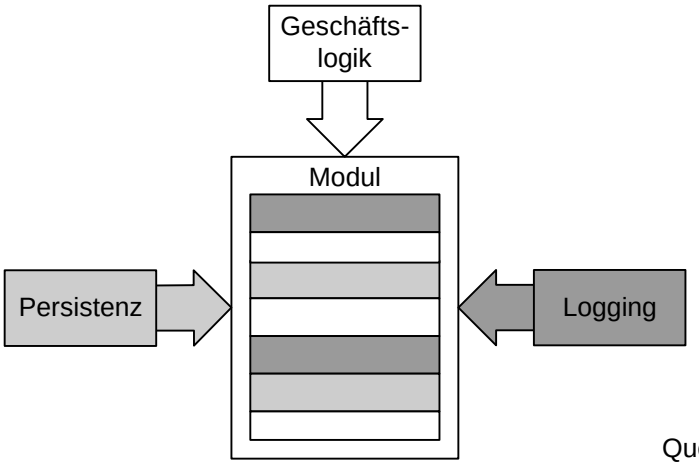


Abbildung 5-2: Tangling von Concerns

Tangling führt erstens zu schlecht lesbarem Code, was negative Auswirkungen auf die Pflegbarkeit und damit indirekt auch auf die Lebensdauer und die Wirtschaftlichkeit der Software hat. Zweitens sind von Tangling betroffene Module schlecht wiederverwendbar, da sie mehrere (Fragmente von) Concerns enthalten. Und drittens ist es schwierig bis unmöglich zurückzuverfolgen, welche Anforderungen ein bestimmtes Modul implementiert.

6. Anforderungen

Da *Crosscutting* Concerns und *nicht-funktionale* Anforderungen in der Literatur (z.B. Brito, Moreira, Araujo, 2002) bisweilen als Synonyme verwendet werden, ist es erforderlich, sich auch mit dem Anforderungsbegriff zu beschäftigen. Über die Zusammenhänge zwischen Concerns und Anforderungen gibt das Kapitel 8 Auskunft.

6.1 Begriffsdefinition

Nach Glinz (2003a) wird der Anforderungsbegriff wie folgt definiert:

> **Definition** *Anforderung*. 1. Eine Bedingung oder Fähigkeit, die eine Sache oder eine Person erfüllen oder besitzen muss, um (von Dritten) an sie gestellte Wünsche und Erwartungen zu erfüllen. 2. Eine Bedingung oder Fähigkeit, die eine Software erfüllen oder besitzen muss, um einen Vertrag, eine Norm oder ein anderes, formell bestimmtes Dokument zu erfüllen.

In diesem Sinne sind die Anforderungen als *detaillierte Spezifikationen von Concerns* zu verstehen. Die Anforderungen können in funktionale, nicht-funktionale und implizite Anforderungen unterteilt werden.

6.2 Funktionale Anforderungen

Funktionale Anforderungen werden überall ähnlich definiert. Die folgende Definition stammt von Sommerville, Sawyer (1997):

> **Definition** *funktionale Anforderung*. Beschreibung, was ein Software-System tun sollte.

Diese Definition ist am einfachsten zu verstehen, wenn man sie der Definition der nicht-funktionalen Anforderung gegenüberstellt.

6.3 Nicht-funktionale Anforderungen

Auch nicht-funktionale Anforderungen werden überall ähnlich definiert.

> **Definition a)** *nicht-funktionale Anforderung*. Nebenbedingung für die Implementierung von funktionalen Anforderungen (Sommerville, Sawyer, 1997).
>
> **Definition b)** *nicht-funktionale Anforderung*. Globale Eigenschaft eines Systems, welche die funktionalen Anforderungen einschränkt (Araujo et al. 2002).

Nicht-funktionale Anforderungen werden oft auch als *Qualitätsattribute* (siehe Brito, Moreira, Araujo, 2002) bezeichnet, d.h. als Systemeigenschaften, die zur übergreifenden Produktqualität beitragen.

6.4 Funktionale und nicht-funktionale Anforderungen

Beim Versuch, funktionale und nicht-funktionale Anforderungen zu unterscheiden, haben sich die Konzepte von *Essenz* und *Inkarnation* sowie das Gedankenexperiment der *perfekten Technologie* (McMenamin, Palmer, 1988) als hilfreich erwiesen:

- Die **Essenz** eines Systems besteht aus seinen wahren Anforderungen und umfasst die Eigenschaften eines Systems, die auch dann vorhanden wären, wenn es mit perfekter Technologie implementiert wäre. Ein System mit perfekter Technologie hat eine unendlich grosse Verarbeitungsleistung und unbeschränkte Leistungsfähigkeit, kann unendlich viele Informationen speichern, verursacht keine Kosten, benötigt weder Energie noch Raum und macht niemals Fehler.
- Die **Inkarnation** eines Systems ist die Gesamtheit aller Personen, Geräte und Maschinen, die dazu benutzt werden, die Essenz eines Systems zu implementieren.

In diesem Sinn repräsentieren die funktionalen Anforderungen die Essenz eines Systems, während die nicht-funktionalen Anforderungen aus der Tatsache resultieren, dass es keine perfekte Technologie gibt.

6.5 Nicht-funktionale Anforderungen und Crosscutting Concerns

Nach Burge, Brown (2002) werden in der Literatur häufig Zuverlässigkeit, Verfügbarkeit, Performance, Sicherheit, Skalierbarkeit, Erweiterbarkeit, Handhabbarkeit, Pflegbarkeit, Interoperabilität, Kosten etc. als typische Beispiele für nicht-funktionale Anorderungen genannt. Aufgrund dieser Beispiele lässt sich eine begriffliche Verwandtschaft zwischen nicht-funktionalen Anforderungen und Crosscutting Concerns vermuten.

Nicht-funktionale Anforderungen lassen sich insofern von funktionalen Anforderungen abgrenzen, dass sie

- sich tendenziell nicht auf bestimmte Funktionen, sondern auf das System als Ganzes beziehen,
- häufig universell sind und auf irgendein System angewendet werden könnten und
- typischerweise die Funktionen quer schneiden (engl. crosscut).

Auch darin kommen die nicht-funktionalen Anforderungen den Crosscutting Concerns begrifflich nahe.

Diese auf den ersten Blick intuitiv festgestellte Übereinstimmung zwischen nicht-funktionalen Anforderungen und Crosscutting Concerns hält jedoch einer näheren Betrachtung nicht stand. Wie die Abbildung 6-1 zeigt, überschneiden sich die beiden Begriffe zwar in b), aber nach Burge, Brown (2002) gibt es zwei deutliche Unterschiede zwischen nicht-funktionalen Anforderungen und Crosscutting Concerns, die es nahe legen, die beiden Begriffe keinesfalls als Synonyme zu verwenden.

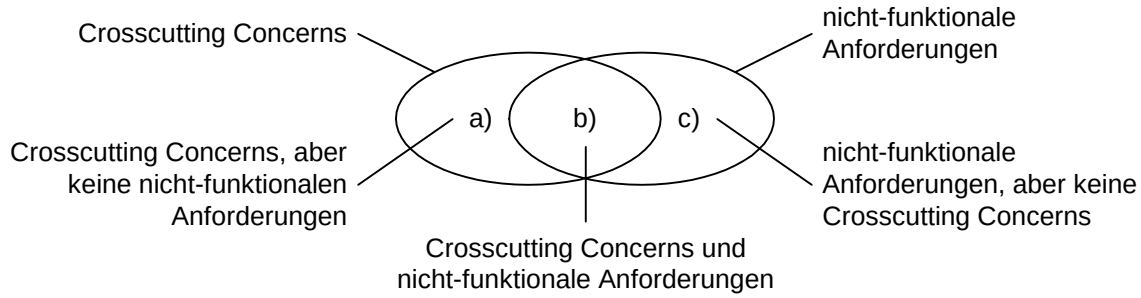


Abbildung 6-1: Crosscutting Concerns und nicht-funktionale Anforderungen

a) Crosscutting Concerns, die keine (nicht-funktionalen) Anforderungen sind. Der Sicherheits-Concern wird in der Literatur (z.B. Brito, Moreira, 2004) oft als typisches Beispiel für einen Crosscutting Concern genannt. Ist der Sicherheits-Concern auch eine nicht-funktionale Anforderung? Die Frage lässt sich mit einem klaren Nein beantworten. Anforderungen haben laut ihrer Definition Vertrags-Charakter. Die *Prüfbarkeit* ist folglich ein grundlegendes Merkmal jeder Anforderung. Das heisst, es muss möglich sein nachzuweisen, ob eine Anforderung erfüllt wurde oder nicht. Der Sicherheits-Concern für sich allein ist noch viel zu wenig konkret, um implementier- und prüfbar zu sein. Es muss zuerst definiert werden, was unter dem abstrakten Sicherheitsbegriff überhaupt zu verstehen ist, was der Auftraggeber bezüglich Sicherheit genau vom künftigen Software-System erwartet. Mit anderen Worten: Der Sicherheits-Concern muss zuerst durch Operationalisierung in prüfbare funktionale Anforderungen zerlegt werden. In Sousa et al. (2004), Abbildung 6, findet sich ein illustratives Beispiel für eine mögliche Operationalisierung des Sicherheits-Concern mit Hilfe eines Softgoal Interdependency Graph (SIG). Bei derartigen Crosscutting Concerns handelt es sich also keinesfalls um nicht-funktionale Anforderungen, sondern um „crosscutting“ *Abstraktionen von funktionalen Anforderungen*. Das mag der Grund sein, weshalb Rashid et al. (2002) die Crosscutting Concerns mit „high-level non-functional requirements“ umschreiben.

b) Crosscutting Concerns, die gleichzeitig nicht-funktionale Anforderungen sind. Ist ein Crosscutting Concern so präzis spezifiziert, dass er implementier- und prüfbar ist, handelt es sich gleichzeitig um eine nicht-funktionale Anforderung.

c) Nicht-funktionale Anforderungen, die keine Crosscutting Concerns sind. Bei einigen zu Beginn dieses Abschnitts erwähnten nicht-funktionalen Anforderungen ist eine Zerlegung in funktionale Anforderungen und damit eine Implementierung nicht möglich:

- Anforderungen, die sich nicht auf das Betriebsumfeld, sondern auf das Entwicklungsumfeld eines Software-Systems beziehen, z.B. Erweiterbarkeit, Pflegbarkeit.
- Anforderungen, die Entscheidungskriterien sind, um eine Auswahl zwischen verschiedenen Lösungsvarianten zu treffen, z.B. Kosten.

Bei solchen Anforderungen handelt es sich zwar um typische nicht-funktionale Anforderungen, jedoch nicht um Crosscutting Concerns, da sie nicht durch Software-Systeme, sondern durch Entwicklungsvorgaben umgesetzt werden.

6.6 Implizite Anforderungen

In jeder Software gibt es Module, die sich nicht auf Anforderungen aus der Anforderungsspezifikation zurückführen lassen. Darunter könnte beispielsweise ein Modul zur Implementierung des MVC-Modells (siehe im Abschnitt 14.2) fallen. Auch solche Module dienen nicht dem Selbstzweck, sondern lassen sich durchaus auf Anforderungen zurückführen, nämlich auf *implizite Anforderungen*. Dabei handelt es sich um Selbstverständlichkeiten, die von jedem Software-System erwartet werden, unter Umständen ohne dass sich der Auftraggeber dessen bewusst ist.

> **Definition** *Implizite Anforderung*. Erwartungen, die verschiedene Beteiligte an ein System oder an den Entwicklungsprozess haben, ohne dass sie dies explizit äussern.

Implizite Anforderungen können *funktional* sein. Beispielsweise erwartet jeder Auftraggeber heutzutage selbstredend, dass die Darstellung eines Sachverhalts am Bildschirm automatisch aktualisiert wird, wenn sich der Sachverhalt ändert. Häufig sind implizite Anforderungen aber auch *nicht-funktional* wie z.B. Pflegbarkeit, Bedienbarkeit, Korrektheit etc. Wie diese Beispiele zeigen, haben implizite Anforderungen oft *Querschnittscharakter*. Aus diesem Grund sind sie für die aspektorientierte Entwicklung von besonderem Interesse. Typischerweise sind implizite Anforderungen nicht anwendungsspezifisch. Gerade Systemfunktionen oder -eigenschaften, die der Auftraggeber aus seiner Erfahrung mit anderen Systemen für selbstverständlich hält, wird er nicht explizit anfordern.

Die Abgrenzung zwischen explizit spezifizierten und impliziten Anforderung ist fliessend und variiert von System zu System und von Unternehmen zu Unternehmen. Je nach Persönlichkeit und Erfahrung des Auftraggebers und der Entwickler werden mehr oder weniger Anforderungen explizit spezifiziert. Ausserdem spielen auch die Gepflogenheiten innerhalb der jeweiligen Informatik-Organisation eine Rolle: Was in Entwicklungsrichtlinien geregelt ist, muss nicht mehr anwendungsspezifisch festgehalten werden.

Implizite Anforderungen erscheinen in keinem Dokument und keinem Modell der Anforderungsphase, werden weder validiert noch durch den Auftraggeber abgenommen, und doch können sie kritisch sein für den Erfolg eines Systems, wenn sie fehlen. In der Motivationstheorie von Herzberg (Scholz, 2000) gibt es dazu eine Analogie, die *Hygienefaktoren*: Das sind – im Gegensatz zu den *Motivatoren* – Bedürfnisse, deren Befriedigung jeder Mitarbeiter selbstverständlich erwartet und die daher keinerlei motivierenden Einfluss haben. Wenn sie befriedigt werden, können sie allenfalls Unzufriedenheit abbauen, jedoch nicht die Zufriedenheit erhöhen. Sie wirken jedoch äusserst demotivierend, wenn sie nicht befriedigt werden. Typische Hygienefaktoren sind beispielsweise die Arbeitsbedingungen und das Gehalt.

Implizite Anforderungen haben durchaus ihre Existenzberechtigung, da es nicht wirtschaftlich ist, sämtliche möglichen Anforderungen an ein System zu erheben. Die Kunst der Entwicklung besteht darin, die impliziten Anforderungen zu erraten, ggf. zu erfragen und umzusetzen.

7. Aspekte

Aspekt bedeutet „Betrachtungsweise, Gesichtspunkt; Aussicht“ und ist von lat. aspicere „hinsehen“ abgeleitet (Duden, 1997). Aspekte und *Crosscutting Concerns* haben nicht die gleiche Bedeutung, obwohl sie in der Literatur (z.B. Baniassad, Clarke, 2004) bisweilen als Synonyme verwendet werden. Wie Mili, Elkharraz, Mcheick (2004) schreiben, gehört die Semantik von Concerns zur Problemdomäne, während die Semantik von Aspekten zur Lösungsdomäne gehört. Diese Unterscheidung ergibt Sinn, da Crosscutting Concerns als Aspekte implementiert werden *können*, aber nicht *müssen*. Crosscutting Concerns gibt es – als Problem –, seit es (Anforderungen an) Software-Systeme gibt, während Aspekte – als Lösungsmöglichkeit – erst seit wenigen Jahren ein Thema sind.

7.1 Begriffsdefinition

Das Wort Aspekt wurde durch Kiczales et al. (1997) zum ersten Mal in seiner heutigen Bedeutung verwendet. Folgende Begriffsdefinitionen sind zu finden:

> **Definition a)** *Aspekt*. Gut modularisierter Crosscutting Concern (Kiczales et al., 2001a).
>
> **Definition b)** *Aspekt*. Modulare Implementierung eines Crosscutting Concern (Lieberherr, Orleans, Ovlinger, 2001).
>
> **Definition c)** *Aspekt*. Einheit modularer Definitionen von Crosscutting Concerns (Sakurai et al., 2004).
>
> **Definition d)** *Aspekt*. Modulare Einheit einer „crosscutting" Implementierung (Jacobson, 2003).
>
> **Definition e)** *Aspekt*. In einem separaten Modul spezifizierter Crosscutting Concern (Rashid et al., 2002).
>
> **Definition f)** *Aspekt*. Funktionale Einheit, die durch das ganze übrige Systemverhalten gewoben ist (Baniassad, Clarke, 2004).
>
> **Definition g)** *Aspekt*. Eigenschaft eines Systems, die sich nicht unbedingt nach den funktionalen Komponenten des Systems ausrichtet (engl. align), sondern dazu neigt, die funktionalen Komponenten quer zu schneiden (Constantinides et al., 2000).
>
> **Definition h)** *Aspekt*. Software-Produkt (engl. artefact), das einen *Concern* behandelt und das (im Idealfall mittels eines automatisierten Prozesses, z.B. *Weaving*) mit den Basis-Modulen und anderen Aspekten verknüpft werden muss, um eine vollständige Applikation zu erhalten (Atkinson, Kühne, 2003).

Auch hier ist wie bei den Concerns positiv zu vermerken, dass sich die Definitionen nirgends widersprechen, sondern lediglich mehr oder weniger ausführlich, mehr oder weniger präzis sind. Favorisiert wird die Definition b). Sie ist kurz, prägnant und betont das Wesentliche, nämlich dass Aspekte einerseits *Module* und anderseits *implementierte Crosscutting Concerns* sind. Dies ist jedoch nur eine vorläufige Definition des Aspektbegriffs. Die Definition erfährt im Abschnitt 8.3 noch zwei Präzisierungen und liegt erst dort ihrer endgültigen Fassung vor.

7.2 Aspekte und Komponenten

Aspekte schneiden Komponenten (oder ggf. andere Aspekte) quer. Das Ziel der aspektorientierten Programmierung ist es, Komponenten und Aspekte sauber voneinander zu trennen. Begrifflich lassen sie sich nach Kiczales et al. (1997) wie folgt unterscheiden:

> **Definition** *Komponente*. Eine Systemeigenschaft ist eine Komponente, wenn sie sauber (d.h. lokal, verwendbar und zusammensetzbar) in einer *verallgemeinerten Prozedur* eingekapselt werden kann. Komponenten resultieren üblicherweise aus der Dekomposition der Kernfunktionalität eines Software-Systems.
>
> **Definition** *Aspekt*. Eine Systemeigenschaft ist ein Aspekt, wenn sie *nicht* sauber in einer verallgemeinerten Prozedur eingekapselt werden kann. Aspekte resultieren üblicherweise *nicht* aus der Dekomposition der Kernfunktionalität eines Software-Systems.

Unter einer *verallgemeinerten Prozedur* (engl. generalized procedure, GP) wird der gemeinsame Abstraktions- und Dekompositionsmechanismus verstanden, der den meisten existierenden (unter anderem auch den objektorientierten und den prozeduralen) Programmiersprachen zu Grunde liegt. Diese Programmiersprachen werden auch GP-Sprachen genannt. Verallgemeinerte Prozeduren können Objekte, Methoden, Prozeduren etc. sein. Eine Gemeinsamkeit der GP-Sprachen ist, dass jeweils nur ein einziger *Dekompositions-Mechanismus* zur Verfügung steht, der zur Dekomposition der Kernfunktionalität[6] eingesetzt werden kann. Der entsprechende einzige *Kompositions-Mechanismus* sind Prozedur-Aufrufe. Da es nur diesen einen Kompositions-Mechanismus gibt, müssen „crosscutting" Systemeigenschaften in GP-Sprachen manuell ins System eingefügt werden, was zum bekannten Tangling-Problem führt.

Die MDSOC und Hyper/J pflegen auch hier – wie schon bei den Concerns (siehe im Abschnitt 5.3) – eine andere Betrachtungsweise. So wie sie nicht zwischen Core und Crosscutting Concerns unterscheiden, unterscheiden sie auch nicht zwischen Komponenten und Aspekten. Sie kennen nur die einander völlig gleichgestellten so genannten *Hyperslices*.

Wo nichts anderes erwähnt ist, wird in dieser Arbeit davon ausgegangen, dass die Komponenten objektorientiert implementiert werden.

7.3 Aspekte, Komponenten und Module

Aspekte und Komponenten haben nebst den erwähnten Unterschieden auch Gemeinsamkeiten. Die wichtigste ist, dass sowohl Aspekte als auch Komponenten möglichst sauber modularisiert sein sollten.

> **Definition** *Modul*. Ein Modul ist eine lexikalisch zusammenhängende Folge von abgegrenzten Programm-Anweisungen mit einem gemeinsamen Namen, unter dem es von anderen Systemteilen aufgerufen werden kann (Schach, 1999).

Für Komponenten gibt es eine Reihe allgemein anerkannter Kriterien, die für eine saubere Modularisierung zu berücksichtigen sind. Nun stellt sich natürlich die Frage, ob die gleichen Modularisierungskriterien auch für Aspekte gelten. Dies wird anhand der Modularisierungskriterien *Kohäsion*, *Kopplung*, *Datenkapselung* und *Information Hiding* von Schach (1999) untersucht. Dabei wird von Aspekten im Sinn von AspectJ (siehe im Abschnitt 9.3) ausgegangen.

Die **Kohäsion** ist nach Glinz (2003b) ein Mass für die Stärke des inneren Zusammenhangs eines Moduls. Die folgende Skala der Ausprägungen der Kohäsion reicht nach Schach (1999) von 1 (schwach) bis 6 bzw. 7 (stark). Anzustreben ist eine starke Kohäsion, d.h. die Ausprägungen 6 (bei objektorientierten Systemen) bzw. 7 (bei Systemen mit funktionaler Dekomposition). Eine starke Kohäsion erleichtert die Pflege und Wiederverwendung von Modulen.

1. Ein Modul hat **zufällige Kohäsion**, wenn es Aktivitäten durchführt, die nichts miteinander zu tun haben. Zufällige Kohäsion heisst also keine Kohäsion.
2. Ein Modul hat **logische Kohäsion**, wenn es miteinander in Beziehung stehende Aktivitäten durchführt, die jeweils vom aufrufenden Modul (z.B. mit einem Funktionscode als Argument) selektiert werden.
3. Ein Modul hat **zeitliche Kohäsion**, wenn es Aktivitäten durchführt, die in zeitlicher Beziehung zueinander stehen.

[6] Die *Dekomposition der Kernfunktionalität* ist nicht zu verwechseln mit der *funktionalen Dekomposition*. Letzteres bezeichnet die hierarchische Zerlegung eines Software-Systems in seine Haupt- und Teil-Funktionen, während sich ersteres auf eine beliebige Gliederung der Kernfunktionalität eines Software-Systems bezieht, sei dies eine Gliederung nach Funktionen, Daten, Klassen oder Prozessen.

4. Ein Modul hat **prozedurale Kohäsion**, wenn es Aktivitäten durchführt, die sich auf eine vorgegebene Reihenfolge von Schritten beziehen.
5. Ein Modul hat **kommunikative Kohäsion**, wenn es prozedurale Kohäsion hat und alle Aktivitäten auf den gleichen Daten operieren.
6. Ein Modul hat **objektbezogene Kohäsion** (engl. informational cohesion), wenn es Aktivitäten mit jeweils eigenem Ein- und Ausgang sowie unabhängiger Implementierung durchführt, die auf den gleichen Daten operieren. Ein Beispiel für ein Modul mit Informations-Kohäsion ist die Implementierung eines abstrakten Datentyps, oder, da Objekte häufig abstrakte Datentypen repräsentieren, eines Objekts.
7. Ein Modul hat **funktionale Kohäsion**, wenn es genau eine Aktivität durchführt oder genau ein Ziel erreicht.

Für Aspekte ist *funktionale Kohäsion* anzustreben, weil ein Aspekt genau eine Aktivität durchführen soll, nämlich sein „crosscutting" Verhalten. Damit kann ein Aspekt am leichtesten gepflegt und wiederverwendet werden. Logische Kohäsion kommt für Aspekte nicht in Frage, da sie nicht direkt aufgerufen werden. Zeitliche, prozedurale und kommunikative Kohäsion sind für Aspekte widernatürlich, da dem „crosscutting" Verhalten weder eine zeitliche noch eine logische Reihenfolge innewohnt. Gegen die objektbezogene Kohäsion schliesslich spricht die Tatsache, dass Aspekte nicht primär daten- sondern verhaltensorientiert sind. Wenn man alle „crosscutting" Verhaltensweisen, die auf den gleichen Daten operieren, im Sinne eines abstrakten Datentyps in einen Aspekt packte, würde damit die Wiederverwendbarkeit dieses Aspekts kompromittiert.

Die **Kopplung** ist nach Glinz (2003b) ein Mass für die Abhängigkeit zwischen zwei Modulen. Die folgende Skala der Ausprägungen der Kopplung reicht nach Schach (1999) von 1 (stark) bis 5 (schwach). Anzustreben ist eine schwache Kopplung, also die Ausprägung 5. Eine schwache Kopplung führt zur grösstmöglichen Unabhängigkeit zwischen den Modulen und erleichtert damit ihre Wiederverwendung und Pflege.

1. Zwei Module haben **inhaltliche Kopplung**, wenn das eine direkt auf die Daten des anderen zugreifen kann.
2. Zwei Module haben **globale Kopplung** (engl. common coupling), wenn beide auf die gleichen globalen Daten zugreifen können.
3. Zwei Module haben **Steuerungskopplung**, wenn das aufrufende Modul durch Übergabe von Steuerdaten (z.B. einem Funktionscode) explizit die Logik des aufgerufenen Moduls steuert.
4. Zwei Module haben **Datenbereichskopplung** (engl. stamp coupling), wenn das aufrufende Modul dem aufgerufenen Modul eine ganze Datenstruktur übergibt, von der dieses nur einen Teil braucht.
5. Zwei Module haben **Datenkopplung**, wenn das aufrufende Modul dem aufgerufenen genau die Daten übergibt, die dieses braucht.

Für die Interaktion zwischen Komponenten und Aspekten ist eine Art *Datenkopplung* anzustreben. Sie entspricht im Prinzip, nicht aber in der Umsetzung der Datenkopplung zwischen zwei Komponenten. Der Unterschied besteht darin, dass das Laufzeitsystem und nicht die Komponente den Aspekt aufruft. Aus Sicht des Aspekts ist dieser Unterschied aber nicht von Bedeutung. Alle anderen Kopplungsarten erhöhen die gegenseitige Abhängigkeit und erschweren dadurch die Pflege und Wiederverwendung der Aspekte.

Datenkapselung bedeutet nach Schach (1999), dass eine Datenstruktur zusammen mit den Aktivitäten, die auf dieser Datenstruktur operieren, in ein Modul gepackt wird. Die Datenkapselung geht einher mit objektbezogener Kohäsion bzw. abstrakten Datentypen. Da für Aspekte wie erwähnt die funktionale Kohäsion anzustreben ist, ist das Modularisierungskriterium der Datenkapselung nur

bedingt auf Aspekte anwendbar. Hingegen gilt der allgemeinere Kapselungsbegriff durchaus auch für Aspekte. Er bedeutet nach Schach (1999), dass alle Aspekte[7] eines Gegenstands der realen Welt in einer Einheit (z.B. einem Modul) zusammengefasst werden.

Information Hiding bedeutet nach Schach (1999), Module so zu entwerfen, dass die Implementierungsdetails vor den anderen Modulen verborgen bleiben. Information Hiding bewirkt, dass die Implementierung eines Moduls geändert werden kann, ohne dass andere Module davon betroffen sind. Dies vermindert die Abhängigkeit zwischen den Modulen und erleichtert damit ihre Pflege. Dieses Modularisierungskriterium gilt uneingeschränkt auch für Aspekte.

Aus den Ausführungen dieses Abschnitts lässt sich schliessen, dass für Komponenten und Aspekte im Wesentlichen die gleichen Modularisierungskriterien gelten. Damit können sowohl Aspekte als auch Komponenten als Unterbegriffe des Modulbegriffs betrachtet werden.

7.4 Eigenschaften von Aspekten

Aspekte besitzen im Idealfall die folgenden Eigenschaften. Die ersten fünf können durch die Entwickler direkt beeinflusst werden, während die letzen vier auch vom gewählten Ansatz abhängen.

- Aspekte sind definitionsgemäss **„crosscutting“**, d.h. sie schneiden Komponenten und andere Aspekte quer. Damit lassen sich die Aspekte von den Komponenten, die naturgemäss nicht „crosscutting“ sind, abgrenzen.
- Aspekte sind **modular**. Diese Eigenschaft teilen sie wie im Abschnitt 7.3 gezeigt mit den Komponenten. Während diese die Core Concerns implementieren, implementieren jene die Crosscutting Concerns. Dadurch wird die *Separation of Concerns* erreicht. Die Kehrseite modularer Aspekte ist, dass es schwieriger wird, das System als Ganzes zu erfassen.
- Aspekte sind **homogen**. Sie führen an den Punkten (Join Points), an denen sich Komponenten und Aspekte schneiden, immer die gleichen Operationen aus. Ein Beispiel aus dem Datenbank-Bereich: Ein Aspekt, der für verschiedene Komponenten die Verbindungen zur Datenbank auf- und abbaut, ist *homogen*, weil in allen Komponenten die genau gleichen Operationen ausgeführt werden. Demgegenüber ist ein Aspekt, der Daten aus verschiedenen Komponenten in der Datenbank speichert, *nicht homogen*, weil je nach Datenstruktur (z.B. Kunden-, Produkt-, Bestelldaten etc.) andere Operationen ausgeführt werden. Daraus resultiert eine Art von *logischer Kohäsion*, die aufgrund der Modularisierungskriterien aus dem Abschnitt 7.3 nicht erwünscht ist. Probleme dieser Art eignen sich im Übrigen generell nicht als Aspekte: Auf der beschriebenen Abstraktionsebene (Daten in der Datenbank speichern) sind sie zwar „crosscutting“, aber nicht homogen. Auf einer tieferen Abstraktionsebene (z.B. Kunden-, Produkt-, Bestelldaten etc. in der Datenbank speichern) sind sie dann zwar homogen, aber mit grosser Wahrscheinlichkeit nicht mehr „crosscutting“.
- Aspekte sind **nebenwirkungsfrei**, d.h. sie sollen den Systemzustand mit Ausnahme ihrer lokalen Variablen möglichst nicht verändern. Dies wird im Abschnitt 14.4 genauer erläutert.
- Aspekte sind **wiederverwendbar**. Aspekte realisieren oft Crosscutting Concerns, die nicht anwendungsspezifisch, sondern in allen Software-Systemen ähnlich bis gleich sind. Aspekte eignen sich daher besonders gut zur Wiederverwendung. Im Extremfall sollen Aspekte – zur Manifestation ihrer Qualität – sogar **zertifizierbar** sein (Elrad et al., 2001).
- Aspekte sind **separierbar** und **zusammensetzbar**. In Bezug auf ihre Entwicklung, Pflege und Wiederverwendung müssen die Aspekte voneinander und von den Komponenten *separierbar* sein. In Bezug auf ihre Nutzung müssen die Aspekte *zusammensetzbar* sein,

[7] hier ist Aspekt nicht im Sinn der Definition im Abschnitt 7.1 gemeint, sondern im umgangssprachlichen Sinn.

d.h. mit Komponenten und anderen Aspekten zu einem Gesamtsystem zusammengefügt werden können (Mili, Elkharraz, Mcheick, 2004, und Elrad et al., 2001).

- Aspekte sind **additiv** (Laddad, 2003). Diese Eigenschaft bezieht sich auf das Hinzufügen/ entfernen von Aspekten zum/vom Gesamtsystem, d.h. zu/von Komponenten und anderen Aspekten. Ein *additiver* Aspekt kann zu/von einem bestehenden System hinzugefügt/ entfernt werden, ohne dass der Quellcode dieses Systems modifiziert werden muss. Ein *invasiver* Aspekt ist genau das Gegenteil: Damit er zu/von einem bestehenden System hinzugefügt/entfernt werden kann, muss der Quellcode modifiziert werden.
- Aspekte sind **transparent**. Die Komponenten sind sich der Aspekte, von denen sie quer geschnitten werden, nicht bewusst (engl. oblivious).
- Aspekte sind **aktiv** in dem Sinne, dass das „crosscutting" Verhalten unter ihrer Kontrolle ausgeführt wird. Im Gegensatz dazu müsste das „crosscutting" Verhalten von *passiven* Aspekten durch die Komponenten aufgerufen werden. Transparente Aspekte sind immer aktiv, da die Komponenten von den Aspekten nichts wissen.

Die vier „S" von H. Ossher im Interview von Elrad et al. (2001) drücken weitere wünschbare Eigenschaften von Aspekten aus:

- Aspekte sind **simultan**. Verschiedene Dimensionen der Systemdekomposition existieren nebeneinander.
- Aspekte sind **unabhängig** (engl. self-contained). Jeder Aspekt muss deklarieren, wovon er abhängig ist, so dass er für sich allein und unabhängig von anderen Aspekten verstanden, modifiziert sowie ein- und ausgeschaltet werden kann. In Hyper/J geschieht dies beispielsweise mittels der so genannten *deklarativen Vollständigkeit.*
- Aspekte sind **symmetrisch**. Es gibt keinen Unterschied zwischen Aspekten und Komponenten, so dass sie flexibel zusammengesetzt werden können. Alle Concerns sind gleichberechtigt, egal, ob sie durch Aspekte oder Komponenten realisiert sind. Diese Eigenschaft ist kennzeichnend für die Subjektorientierung (siehe im Abschnitt 10.1), die MDSOC und Hyper/J. In allen anderen Ansätzen, die Aspekte und Komponenten voneinander trennen (und das sind die weitaus meisten), sind höchstens die Aspekte untereinander, jedoch nicht die Aspekte und Komponenten symmetrisch.
- Aspekte sind **spontan**. Das Identifizieren und Kapseln neuer Concerns ist im Laufe des Software-Lebenszyklus jederzeit möglich.

7.5 Kategorisierung von Aspekten

Es gibt verschiedene Dimensionen, nach denen Aspekte kategorisiert werden können. Die verschiedenen Dimensionen werden jeweils mit typischen Aspekt-Beispielen illustriert. Zuerst werden die verwendeten Beispiele in alphabetischer Reihenfolge kurz erklärt:

- **Fehlerbehandlungsaspekte** ersetzen z.B. geprüfte durch ungeprüfte Ausnahmen, um Übersetzungsfehler zu vermeiden.
- **Geschäftsaspekte** stellen die systemweite Einhaltung bestimmter Geschäftsregeln sicher.
- **Logging-Aspekte** halten bestimmte durchgeführte Operationen fest, sei dies durch Anzeige am Bildschirm, Ausdruck auf Papier oder Speicherung in einer Datei. Für Logging-Aspekte gibt es viele verschiedene Anwendungsmöglichkeiten: Auditing, Tracing, Debugging, Fehler-Logging, Profiling, Testing etc.
- **Nebenläufigkeitsaspekte** helfen, nebenläufige Prozesse sicher zu steuern. Beispiele sind Aspekte zur Erzwingung der Ein-Thread-Regel von Swing (Details siehe im Abschnitt 9.3) sowie Synchronisationsaspekte (Lese/Schreib-Sperren).

- **Optimierungsaspekte**. Pooling- und Caching-Aspekte ermöglichen die Mehrfachverwendung von Ressourcen (z.B. Connections, Threads) und erhöhen damit die Systemleistung. Anstatt dass jeder Prozess jede benötigte Ressource vor ihrer Verwendung erzeugen und anschliessend wieder zerstören muss (was viel *Overhead* verursacht), werden die Ressourcen in einem *Pool* bzw. *Cache* verwaltet, und die Prozesse können sie bei Bedarf von dort beziehen. Der wesentliche Unterschied zwischen Pooling und Caching besteht nach Laddad (2003) darin, dass Pooling eine exklusive Nutzung durch jeweils einen Prozess erfordert, während Caching eine gleichzeitige Nutzung durch verschiedene Prozesse erlaubt.
- **Programmierrichtlinienaspekte** (engl. policy enforcement). Mit Hilfe von Aspekten können in AspectJ gewisse Entwurfsgrundsätze (z.B. Variablenzugriffe nur über `get-` und `set`-Methoden, verbotene Aufrufe von Methoden aus bestimmten Paketen etc.) durchgesetzt werden.
- **Sicherheitsaspekte** umfassen in erster Linie die Authentisierung (Sicherstellung der Identität eines Benutzers oder Systems) und die Autorisierung (Sicherstellung der Zugriffsberechtigung).

7.5.1 Dimension „Herkunft“

Aspekte können aus *funktionalen*, *nicht-funktionalen* oder *impliziten Anforderungen* stammen:

Aspekte aus funktionalen Anforderungen. Funktionale Anforderungen werden durch den Auftraggeber explizit gestellt und beziehen sich auf die Funktionen des zu entwickelnden Systems. Typische aus funktionalen Anforderungen stammende Aspekte sind Fehlerbehandlungsaspekte, Geschäftsaspekte und Logging-Aspekte zwecks Auditing. Der Auftraggeber kann Aspekte aus funktionalen Anforderungen beim Abnahmetest direkt prüfen.

Aspekte aus nicht-funktionalen Anforderungen. Nicht-funktionale Anforderungen werden ebenfalls durch den Auftraggeber explizit gestellt und sind wie im Abschnitt 6.3 erwähnt Nebenbedingungen für die Implementierung der funktionalen Anforderungen. Diese Kategorie könnte nach Glinz (2003a) weiter unterteilt werden in Aspekte aus *Leistungsanforderungen* (z.B. Optimierungsaspekte), Aspekte aus *besonderen Qualitäten* und Aspekte aus *Randbedingungen*. Nicht-funktionale Anforderungen lassen sich, wie schon das Wort sagt, nicht direkt als Funktionen implementieren. Es ist Sache der Entwickler, darüber zu entscheiden, mit welchen technischen Mitteln eine nicht-funktionale Anforderung am besten implementiert wird. Diese Entwurfsentscheidungen (und die daraus resultierenden Aspekte) bleiben dem Auftraggeber gegenüber verborgen. Zu dieser Kategorie gehören typischerweise Optimierungsaspekte, die aus Leistungsanforderungen abgeleitet werden können. Der Auftraggeber kann Aspekte aus nicht-funktionalen Anforderungen beim Abnahmetest nur indirekt prüfen: Er prüft nicht den Aspekt selbst (z.B. den Optimierungsaspekt), sondern die zugrunde liegende nicht-funktionale Anforderung (z.B. die Leistungsanforderung).

Aspekte aus impliziten Anforderungen. Implizite Anforderungen werden durch den Auftraggeber nicht explizit gestellt, er geht aber implizit davon aus, dass sie erfüllt sind. Ein typisches Beispiel für aus impliziten Anforderungen stammende Aspekte sind Nebenläufigkeitsaspekte, die unter anderem der Korrektheit und Bedienbarkeit dienen. Der Auftraggeber wird Aspekte aus impliziten Anforderungen beim Abnahmetest nicht explizit prüfen.

7.5.2 Dimension „Bezug“

Nicht alle Aspekte realisieren Anforderungen an das zu entwickelnde System. Es gibt auch Aspekte, die den Entwicklungsprozess unterstützen. Erstere werden *Produktionsaspekte* (engl. production oder deployment aspects) genannt, Letztere *Entwicklungsaspekte* (engl. developmental

aspects). Diese Kategorisierung stammt von Laddad (2003) und ist AspectJ-spezifisch, könnte aber in anderen Ansätzen analog übernommen werden.

Produktionsaspekte. Produktionsaspekte sind der Normalfall. Sie implementieren die Crosscutting Concerns und sind während der gesamten Implementierungs-, Test- und Nutzungsphase eines Systems relevant. Typische Beispiele für Produktionsaspekte sind Fehlerbehandlungsaspekte, Geschäftsaspekte, Logging-Aspekte zwecks Auditing, Nebenläufigkeitsaspekte, Optimierungsaspekte und Sicherheitsaspekte.

Entwicklungsaspekte. Entwicklungsaspekte helfen, die Qualität (z.B. Effizienz und Effektivität) des Entwicklungsprozesses zu verbessern. Im Unterschied zu den Produktionsaspekten sind sie ausschliesslich während der Implementierungs- und Testphase im Gebrauch und könnten theoretisch kurz vor der Inbetriebnahme des Systems entfernt werden. Meistens will man sie aber in der Implementierungs- und Testphase der nächsten System-Releases wieder verwenden, so dass sie im System belassen, jedoch *deaktiviert* werden. Für das Deaktivieren gibt es ein Idiom, das im Abschnitt 9.3 erklärt wird. Typische Beispiele für Entwicklungsaspekte sind Logging-Aspekte zur Unterstützung des Testens (siehe im Abschnitt 14.5) und Programmierrichtlinienaspekte.

7.5.3 Dimension „Wirkung“

Aspekte können sich im Zeitpunkt ihrer Wirkung unterscheiden. *Statische Aspekte* wirken nach Laddad (2003) zur Übersetzungszeit, *dynamische Aspekte* zur Laufzeit. Diese Unterscheidung ist AspectJ-spezifisch.

Statische Aspekte. Statische Aspekte können als Erweiterungen des Übersetzers betrachtet werden, indem sie die statische Struktur des Codes prüfen. Typische Beispiele sind Programmierrichtlinienaspekte.

Dynamische Aspekte. Dynamische Aspekte wirken zur Laufzeit und repräsentieren den Normalfall. Zu dieser Kategorie gehören alle anderen Aspekte, also Fehlerbehandlungsaspekte, Geschäftsaspekte, Logging-Aspekte, Nebenläufigkeitsaspekte, Optimierungsaspekte und Sicherheitsaspekte.

7.5.4 Dimension „Bedeutung“

Die Bedeutung eines Aspekts kann anhand der Konsequenzen, die seine Entfernung bzw. Deaktivierung für das Funktionieren des Systems hat, gemessen werden:

Übersetzungsfehler. Die Entfernung von Fehlerbehandlungsaspekten führt zu Übersetzungsfehlern.

Fehlverhalten. Die Entfernung von Geschäftsaspekten (führt zu verletzten Geschäftsregeln), Logging-Aspekten zwecks Auditing (fehlendes Logging), Nebenläufigkeitsaspekten (Inkonsistenzen) und Sicherheitsaspekten (fehlende Authentisierung und Autorisierung) kann zu Fehlverhalten führen.

Performanceeinbusse. Die Entfernung von Optimierungsaspekten kann schlimmstenfalls zu Performance-Einbussen führen.

Keine Beeinträchtigung. Die Entfernung von Entwicklungsaspekten führt zu keinerlei Beeinträchtigung.

7.6 Aspekte in der Modellierungssprache ADORA

Der Aspektbegriff in der Modellierungssprache ADORA (siehe Joos, 1999) hat *inhaltlich* mit dem Aspektbegriff der Aspektorientierung nicht viel gemeinsam. Er bezieht sich auf Systemmodelle, beispielsweise Funktions-, Struktur- und Verhaltensmodelle, welche die Funktions-, Struktur- und Verhaltens*aspekte* eines Systems modellieren. Sommerville (2001) nennt diese *Blickwinkel* oder *Perspektiven*, Oestereich (2001) nennt sie *Sichten*. Der Aspektbegriff in ADORA ist also in einem viel umgangssprachlicheren Sinn zu verstehen als der Aspektbegriff der Aspektorientierung. Das ist wohl auch der Grund, dass Joos (1999) den Aspektbegriff weder erklärt noch definiert.

Hingegen besteht zwischen dem Aspektbegriff von ADORA und dem Aspektbegriff der Aspektorientierung in *struktureller* Hinsicht durchaus eine Verwandtschaft: Das Ziel von ADORA ist die Integration einer Menge nebeneinander stehender Aspektmodelle in ein zentrales Gesamtmodell. In diesem Sinn hat die Aspektorientierung ein strukturell ähnliches Problem zu bewältigen wie ADORA, nämlich die Integration von Aspekten und Komponenten, das *Weaving*.

Der Aspektbegriff von ADORA hat ausserdem eine gewisse Ähnlichkeit mit den Dimensionen der MDSOC (siehe im Abschnitt 9.6), die unter anderem eine Funktions-Dimension und eine Daten- bzw. Klassen-Dimension umfassen.

8. Begriffliche Zusammenhänge

Nachdem die Begriffe *Concern*, *Anforderung*, *Aspekt* und *Komponente* isoliert voneinander betrachtet wurden, wird nun versucht, ein Zusammenhang zwischen ihnen herzustellen. Unter den vielen Möglichkeiten wurde eine ausgewählt, die dem gängigen Sprachgebrauch am ehesten entspricht.

8.1 Grobe Betrachtung

Um die groben Zusammenhänge zu verdeutlichen, werden die Begriffe in der Abbildung 8-1 einander gegenübergestellt. Ein Pfeil von Kästchen A zu Kästchen B bedeutet „aus A entsteht B“. Auf dieser Abstraktionsebene sieht alles noch relativ einfach aus.

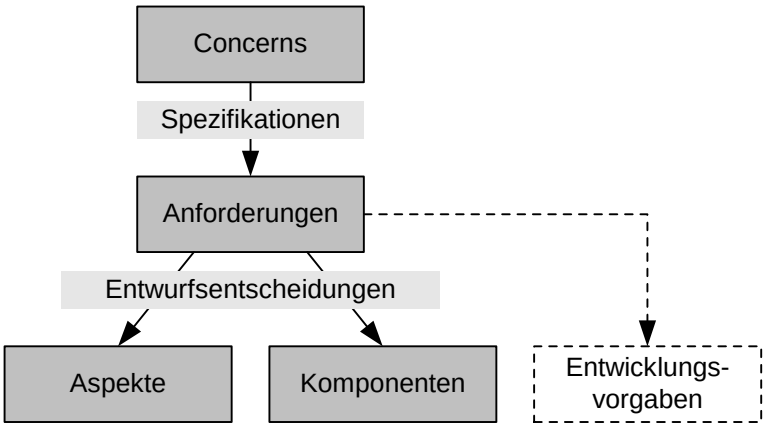


Abbildung 8-1: Grobe begriffliche Zusammenhänge

Concerns und Anforderungen. Durch Spezifikationen entstehen aus Concerns Anforderungen Die Concerns werden gemäss ihrer Definition direkt aus den Systemzielen abgeleitet und sind demzufolge eher allgemein formuliert. Damit sie überhaupt implementiert und validiert werden können, müssen sie in Form von detaillierten, prüfbaren Anforderungen spezifiziert werden.

Anforderungen, Aspekte und Komponenten. Durch Entwurfsentscheidungen entstehen aus Anforderungen Aspekte und Komponenten. Um die erwähnten Scattering- und Tangling-Probleme zu vermeiden, ist es ein erklärtes Ziel der aspektorientierten Entwicklung, sämtliche Anforderungen, welche Crosscutting Concerns spezifizieren, als Aspekte zu entwerfen und zu implementieren, während die Anforderungen, welche Core Concerns spezifizieren, als Komponenten entworfen und implementiert werden.

Anforderungen und Entwicklungsvorgaben. Der Vollständigkeit halber werden hier auch die Entwicklungsvorgaben aufgeführt, die aus nicht-funktionalen Anforderungen (z.B. Kosten) resultieren (siehe auch im Abschnitt 6.5). Sie werden in dieser Arbeit nicht weiter behandelt.

8.2 Detaillierte Betrachtung

Der Teufel liegt im Detail! Geht man eine Abstraktionsebene tiefer, stellen sich die Zusammenhänge schon recht kompliziert dar, wie die Abbildung 8-2 zeigt.

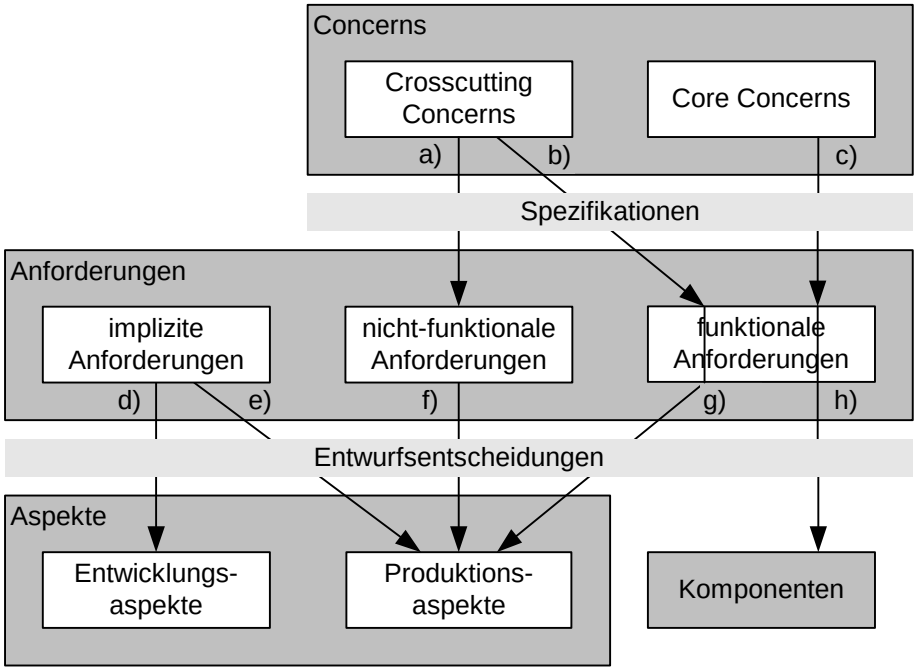


Abbildung 8-2: Detaillierte begriffliche Zusammenhänge

a) Crosscutting Concerns → nicht-funktionale Anforderungen. Crosscutting Concerns mit nicht-funktionalem Charakter werden durch nicht-funktionale Anforderungen spezifiziert. Als typische Beispiele dafür können genannt werden:

- **Antwortzeit-Concerns** werden durch (nicht-funktionale) Leistungsanforderungen spezifiziert, z.B. „Die durchschnittliche Antwortzeit darf 2 s nicht überschreiten".
- **Concerns zur Einhaltung bestimmter Normen oder Gesetze** werden durch (nicht-funktionale) Randbedingungen spezifiziert, z.B. ein Buchhaltungssystem muss bestimmten Rechnungslegungsvorschriften genügen.

b) Crosscutting Concerns → funktionale Anforderungen. Crosscutting Concerns mit funktionalem Charakter werden durch funktionale Anforderungen spezifiziert. Typische Beispiele dafür sind:

- **Logging-, Anzeige- oder Druck-Concerns**. Die Anforderung, bestimmte Operationen in einem Logbuch festzuhalten, ist von Natur aus funktional. Das gleiche gilt für die Anforderung, bestimmte Informationen übergreifend anzuzeigen oder auszudrucken.
- **Geschäftsregel-Concerns** stellen die Einhaltung übergreifender Geschäftsregeln sicher, z.B. „Der Saldo eines Sparkontos in einer Bankanwendung darf nie negativ werden".
- **Sicherheits-Concerns**. Sie sind ursprünglich oft vage formuliert und haben eher nicht-funktionalen Charakter, z.B. „Das System muss den unautorisierten Zugriff auf die Kundenstammdaten verhindern, soweit dies technisch möglich ist" (Glinz, 2003a). Solche Anforderungen bedürfen einer ausführlichen und sorgfältigen Analyse und Spezifikation und müssen – noch auf der Anforderungsebene – stufenweise konkretisiert (Fachbegriff: *operationalisiert*) werden, damit ihre Auswirkungen auf den weiteren Entwicklungsprozess (z.B. Machbarkeit, Zeit und Kosten für Entwurf und Implementierung) beurteilt werden können. Als Darstellungsmittel für die Operationalisierung eignet sich beispielsweise ein Softgoal Interdependency Graph (SIG, siehe im Abschnitt 12.6).

Ganz nebenbei: Aus der Tatsache, dass es viele Crosscutting Concerns gibt, die durch funktionale Anforderungen spezifiziert werden, lässt sich erkennen, dass Crosscutting Concerns und nicht-funktionale Anforderungen mit Bestimmtheit keine Synonyme sind.

c) Core Concerns → funktionale Anforderungen. Die Definition des Begriffs Core Concern („Core Concerns realisieren die Kernfunktionalität eines Systems") aus dem Kapitel 5 sagt es bereits aus: Core Concerns werden typischerweise als funktionale Anforderungen spezifiziert. Dies ist auch der Grund, weshalb es in der Abbildung 8-2 keinen Pfeil von den Core Concerns zu den nicht-funktionalen Anforderungen gibt.

d) Implizite Anforderungen → Entwicklungsaspekte. Implizite Anforderungen sind naturgemäss nicht anwendungsspezifisch, betreffen das Software-System als Ganzes und können daher, falls sie funktionalen Charakter haben, als Aspekte implementiert werden. Bezieht sich eine implizite Anforderung auf den *Entwicklungsprozess*, kann sie als Entwicklungsaspekt implementiert werden, z.B. kann ein Debugging-Aspekt zur Unterstützung der Fehlersuche die Produktivität erhöhen. Die meisten Entwicklungsaspekte dürften aus impliziten Anforderungen stammen.

e) Implizite Anforderungen → Produktionsaspekte. Hat eine implizite Anforderung „crosscutting" Charakter, was der Normalfall sein dürfte, kann sie als Produktionsaspekt implementiert werden, z.B. können Nebenläufigkeitsaspekte (siehe im Abschnitt 7.5) zur Korrektheit und Bedienbarkeit der Software beitragen. Die Implementierung einer impliziten Anforderung als Komponente ist selten, weshalb in der Abbildung 8-2 kein Pfeil von den impliziten Anforderungen zu den Komponenten eingezeichnet ist.

f) Nicht-funktionale Anforderungen → Produktionsaspekte. Implementierungen sind naturgemäss *funktional*. Nicht-funktionale Anforderungen können also nicht direkt entworfen und implementiert werden, sondern müssen zuerst – auf der Entwurfsebene – stufenweise konkretisiert (quasi *funktionalisiert*) werden. Das heisst, es ist zu überlegen, mit welchen Massnahmen eine bestimmte Anforderung am besten erfüllt werden kann. Auch auf dieser Ebene eignet sich der Softgoal Interdependency Graph (SIG, siehe im Abschnitt 12.6) als Darstellungsmittel. Ein schönes Beispiel dafür zeigt die Abbildung 6 in Sousa et al. (2004). Haben solcherart konkretisierte nicht-funktionale Anforderungen „crosscutting" Charakter (was der Normalfall sein dürfte), werden sie als Produktionsaspekte (oder in Hyper/J als separate Concerns, so genannte Hyperslices) entworfen und implementiert, andernfalls als Komponenten.

Die Abgrenzung zwischen der Konkretisierung (Operationalisierung) auf der Anforderungsebene und der Konkretisierung (Funktionalisierung) auf der Entwurfsebene ist nicht scharf. Im Fall des

Sicherheits-Concern kann die Grenze beispielsweise quer durch einen Software Dependency Graph verlaufen. Als Hilfsmittel zur Abgrenzung wird empfohlen, sich bei jedem Konkretisierungsschritt zu fragen, ob er für den Auftraggeber relevant ist. *Für den Auftraggeber relevant* bedeutet: erfordert die Zustimmung des Auftraggebers (Glinz, 2003a) und wird durch den Auftraggeber beim Abnahmetest geprüft. Ist dies der Fall, gehört die Konkretisierung zur Anforderungsebene, sonst gehört sie zur Entwurfs- und Implementierungsebene. Im Zweifelsfall ist es besser, den Auftraggeber einmal zu viel als einmal zu wenig zu involvieren.

g) Funktionale Anforderungen → Produktionsaspekte. Hier geht es um funktionale Anforderungen, die über den Pfeil b) aus Crosscutting Concerns entstanden sind. Sie können als Produktionsaspekte implementiert werden. Die funktionalen Anforderungen, die über den Pfeil c) aus Core Concerns entstanden sind, werden im nächsten Abschnitt h) behandelt.

h) Funktionale Anforderungen → Komponenten. Es sind drei Fälle zu unterscheiden, in denen funktionale Anforderungen als Komponenten implementiert werden:

Der **erste Fall** betrifft funktionale Anforderungen, die über den Pfeil c) aus Core Concerns entstanden sind. Sie werden typischerweise und seit jeher als Komponenten implementiert.

Im **zweiten Fall** geht es um funktionale Anforderungen, die über den Pfeil b) aus Crosscutting Concerns entstanden sind. In einer konventionellen Programmierumgebung müssen sämtliche Crosscutting Concerns mangels Alternativen als Komponenten implementiert werden. Das sind genau die Fälle, welche die erwähnten Scattering- und Tangling-Probleme verursachen. Natürlich steht es auch in einer aspektorientierten Programmierumgebung jedem Entwickler frei, Crosscutting Concerns als Komponenten zu implementieren.

Der **dritte Fall** ist etwas weniger offensichtlich und nicht ganz einfach zu verstehen. Betroffen sind wie im zweiten Fall funktionale Anforderungen, die über den Pfeil b) aus Crosscutting Concerns entstanden sind. Es kommt häufig vor, dass ein Crosscutting Concern teils als Aspekt, teils als Komponente implementiert wird. Dies rührt nach Laddad (2003) daher, dass viele Crosscutting Concerns aus zwei Teilen bestehen, einem *Server*-Teil und einem *Client*-Teil. (Die Begriffe *Server* und *Client* sind dabei im klassischen Sinn der objektorientierten Programmierung zu verstehen, nämlich als Leistungserbringer und Leistungsempfänger, und nicht als Hardware-Einheiten.)

- **Server-Teil**. Der Server-Teil (z.B. ein Logger) ist für sich allein betrachtet nicht „crosscutting“ und beispielsweise mittels Klassen und Interfaces sauber modularisierbar. Der Server-Teil wird daher typischerweise als Komponente implementiert.
- **Client-Teil**. Demgegenüber hat der Client-Teil (z.B. ein Logging-Auftrag an den Logger) in jedem Fall „crosscutting“ Charakter, sobald das Logging Operationen aus mehreren Komponenten betrifft. Der Client-Teil wird daher typischerweise als Aspekt implementiert.

Die Abbildung 8-3 hilft, diese Unterscheidung besser zu verstehen. Auf der linken Seite sind die Quellcode-Fragmente der konventionellen Implementierung eines Crosscutting Concern, hier zum Beispiel eines Logging-Concern, dargestellt. Die Scattering- und Tangling-Probleme beziehen sich offensichtlich nur auf den Client-Teil des Logging-Concern, nämlich den Logging-Auftrag an den Logger. Der Logger ist schon in der konventionellen Implementierung eine sauber modularisierte Komponente, z.B. eine Klasse. Die rechte Seite zeigt die aspektorientierte Implementierung des gleichen Logging-Concern. Hier ist nun auch der Client-Teil mit Hilfe eines Aspekts sauber modularisiert. Der Server-Teil bleibt gegenüber der konventionellen Implementierung unverändert.

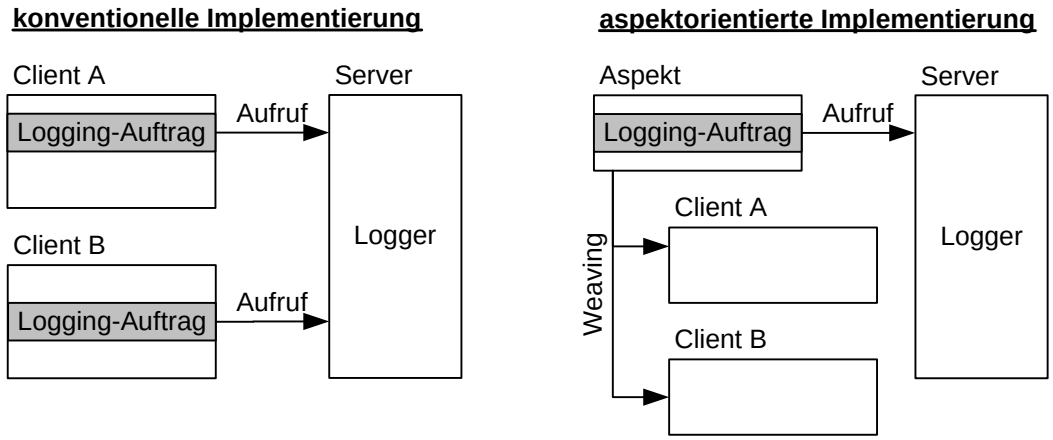


Abbildung 8-3: Der Client- und Server-Teil von Crosscutting Concerns

Es bleibt zu ergänzen, dass der Server-Teil selbstverständlich nicht in jedem Fall als Komponente implementiert werden *muss*. Er *kann* und soll als Komponente implementiert werden, wenn dies zu einer besseren Modularisierung führt, er kann aber auch in den Aspekt integriert werden. Auch hier bildet Hyper/J eine Ausnahme, indem der Server- und der Client-Teil gemeinsam in einem Hyperslice zusammengefasst werden.

Schlussfolgerung. Der Entscheid, ob eine Anforderung als Aspekt oder Komponente entworfen und implementiert werden soll, hängt nicht so sehr davon ab, ob sie funktional oder nicht-funktional ist, sondern vielmehr davon, ob sie „crosscutting“ ist bzw. ob sie sich auf einen Crosscutting Concern oder auf einen Core Concern zurückführen lässt. Aus diesem Grund ist die Unterscheidung funktional/nicht-funktional in den folgenden Kapiteln von untergeordneter Bedeutung, auch wenn in einigen der im Kapitel 12 beschriebenen Ansätze der aspektorientierten Anforderungstechnik von nicht-funktionalen Anforderungen bzw. nicht-funktionalen Concerns die Rede ist.

8.3 Revidierter Aspektbegriff

Zur Erinnerung: Im Abschnitt 7.1 wurde der Aspektbegriff wie folgt definiert:

> **Definition** *Aspekt*. Modulare Implementierung eines Crosscutting Concern (Lieberherr, Orleans, Ovlinger, 2001).

Die Erkenntnisse aus dem Abschnitt 8.2 über den Zusammenhang zwischen Crosscutting Concerns und Aspekten sind in der Abbildung 8-4 dargestellt. Einerseits werden nicht immer die gesamten Crosscutting Concerns als Aspekte (grau schattiert) implementiert, anderseits gibt es Aspekte, die nicht aus Crosscutting Concerns abgeleitet werden können.

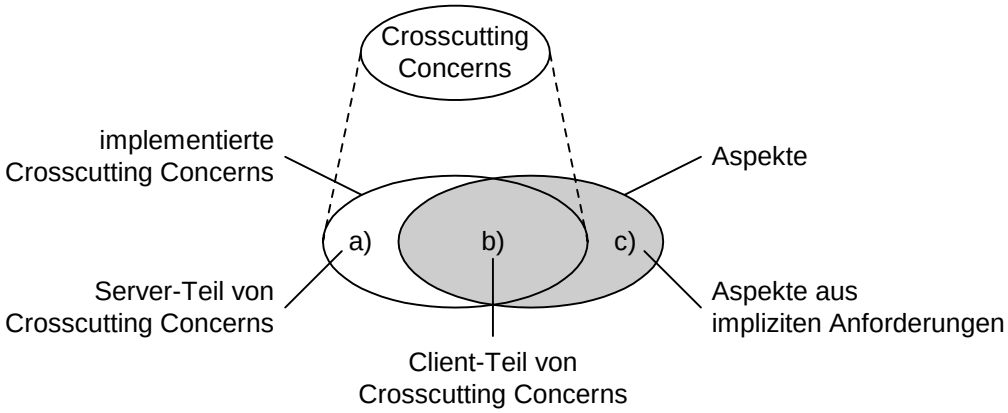


Abbildung 8-4: Crosscutting Concerns und Aspekte

Aufgrund dieser Erkenntnisse ist die obige Definition in zweierlei Hinsicht zu präzisieren:

- Oft wird nicht der gesamte Crosscutting Concern als Aspekt implementiert, sondern nur der *Client-Teil* davon. Nur er ist tatsächlich „crosscutting“ und gehört in der Abbildung 8-4 zur Schnittmenge b) von implementierten Crosscutting Concerns und Aspekten. Der *Server-Teil* hingegen, zu finden in der Menge a), hat nicht „crosscutting“ Charakter. Er wird daher konventionell (z.B. durch eine objektorientierte Klasse) und nicht als Aspekt implementiert. Demzufolge muss er aus der obigen Definition ausgeschlossen werden.
- Einige Aspekte, zusammengefasst durch die Menge c) in der Abbildung 8-4, können nicht auf Crosscutting Concerns zurückgeführt werden, sondern sie implementieren implizite Anforderungen. Sie fehlen in der obigen Definition.

Die diesbezüglich überarbeitete Definition des Aspektbegriffs lautet:

> **Definition** *Aspekt.* Modulare Implementierung einer *impliziten Anforderung* oder des *Client-Teils* eines Crosscutting Concern.

Geht es nicht um eine möglichst präzise, sondern in erster Linie um eine möglichst einfache, selbsterklärende Definition für den allgemeinen Sprachgebrauch, wird empfohlen, nach wie vor die ursprüngliche Definition zu verwenden.

Teil II:
Die aspektorientierten Ansätze im Überblick

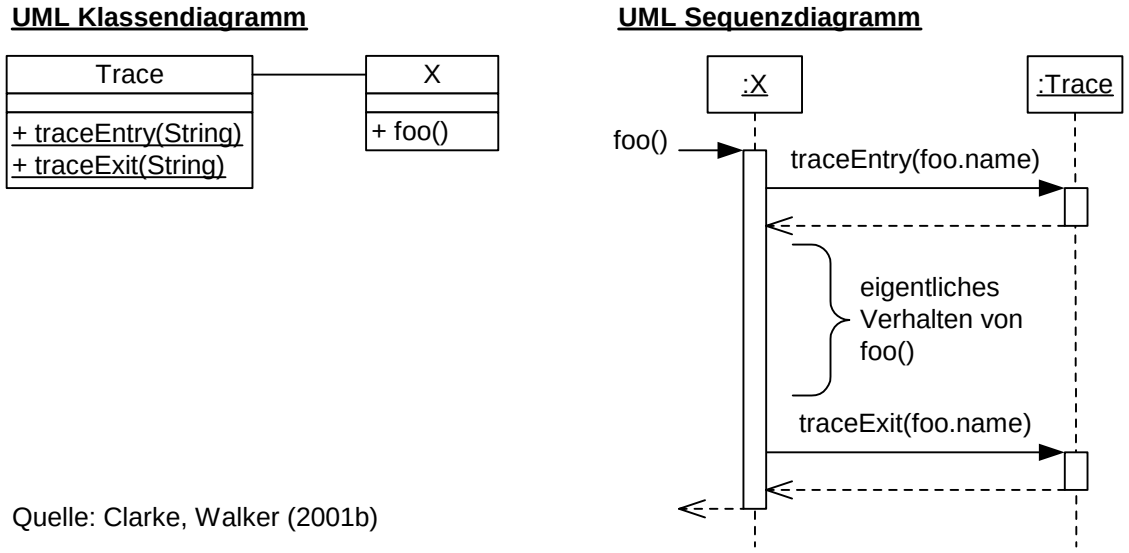
Der Teil II gibt einen Überblick über das Forschungsgebiet der Aspektorientierung anhand von ausgewählten aspektorientierten Ansätzen aus der Literatur. Grundsätzlich werden nur Ansätze berücksichtigt, die eine gewisse Allgemeingültigkeit und/oder eine gewisse Verbreitung in der Literatur gefunden haben. Die Terminologie der Ansätze wird beibehalten und nicht durch die im Teil I definierten Begriffe ersetzt. Auf wesentliche Unterschiede in der Begriffswahl wird hingewiesen.

Wie die meisten Neuerungen in der Informatik hat sich auch die Aspektorientierung aus der Programmierung heraus entwickelt. Es wird allgemein Kiczales et al. (1997) zugeschrieben, den Begriff *Aspekt* zum ersten Mal erwähnt zu haben. Kiczales war massgeblich an der Entwicklung von AspectJ beteiligt. Erst einige Jahre nach dem Aufkommen der aspektorientierten Programmierung wurden erste aspektorientierte Ansätze für frühere Phasen des Entwicklungsprozesses veröffentlicht. Nicht zuletzt aus diesem Grund sind die Ansätze nicht in der Reihenfolge der Phasen, sondern genau umgekehrt beschrieben: zuerst im Kapitel 9 die Ansätze der aspektorientierten Programmierung (engl. Aspect-oriented Programming, AOP), dann im Kapitel 10 die Ansätze des aspektorientierten Entwurfs (engl. Aspect-oriented Design, AOD), im Kapitel 11 die Ansätze der aspektorientierten Architektur (engl. Aspect-oriented Architecture, AOA) und schliesslich im Kapitel 12 die Ansätze der aspektorientierten Anforderungstechnik (engl. Aspect-oriented Requirements Engineering, AORE). Überdies ist es nützlich, mit den wichtigsten Konzepten der aspektorientierten Programmierung vertraut zu sein, bevor man sich mit den Ansätzen der früheren Phasen des Entwicklungsprozesses befasst.

9. Ansätze der aspektorientierten Programmierung

9.1 Einführendes Beispiel einer konventionellen Implementierung

Anhand eines einfachen mit UML[8] modellierten Beispiels soll nun illustriert werden, worin die Probleme liegen, wenn *Crosscutting Concerns* (hier: Tracing) mit konventionellen Mitteln realisiert werden. Tracing ist nach Laddad (2003) eine spezielle Form von Logging, bei welchem bestimmte Informationen zu Beginn und am Ende der Ausführung ausgewählter Methoden festgehalten werden. Die Klasse `Trace` in der Abbildung 9-1 realisiert das gewünschte Tracing-Verhalten.



Quelle: Clarke, Walker (2001b)

Abbildung 9-1: Beispiel für die konventionelle Programmierung eines Crosscutting Concern

`foo()` repräsentiert eine Methode, für die ein Tracing gewünscht wird. Vor dem eigentlichen Verhalten von `foo()` wird die `traceEntry()`-Methode der `Trace`-Klasse aufgerufen, unmittelbar danach die `traceExit()`-Methode. Diese Art von Entwurf bringt die folgenden für Crosscutting Concerns typischen Probleme mit sich, die im Abschnitt 5.5 detailliert erläutert werden:

- **Tangling**. Die `foo()`-Methode enthält neben ihrem eigentlichen Verhalten zusätzlich auch Fragmente (`traceEntry()`, `traceExit()`) des Tracing-Verhaltens. Die Lesbarkeit und Wiederverwendung des Codes wird dadurch erschwert.
- **Scattering**. Das Tracing-Verhalten ist über mehrere Klassen (`Trace`, `X`) verstreut. Jede Methode, für die ein Tracing gewünscht wird, muss die entsprechenden Interaktionen mit der `Trace`-Klasse implementieren. Änderungen des Tracing-Verhaltens erfordern unter Umständen Änderungen in allen vom Tracing betroffenen Methoden.

Mit anderen Worten: In diesem Beispiel ist die *Separation of Concerns* verletzt.

[8] www.uml.org (25.07.2004). Die deutschen Bezeichnungen und die Notation der UML stammen aus Oestereich (2001).

9.2 Grundlagen und Übersicht

9.2.1 Separation und Integration

Grundidee der aspektorientierten Programmierung ist es, die *Separation of Concerns* von Komponenten und „crosscutting“ Aspekten zu meistern.

Wie bereits im Kapitel 1 erwähnt, erfolgt dies bei allen Ansätzen in zwei Schritten:

1. **Separation**. Im ersten Schritt werden, wie in der Abbildung 9-2 dargestellt, die Komponenten und Aspekte möglichst getrennt voneinander entworfen und in ihren jeweiligen Sprachen (z.B. Java[9] als Komponentensprache, AspectJ als Aspektsprache) implementiert. Dazu gehört auch die Implementierung der Integrationsregeln.

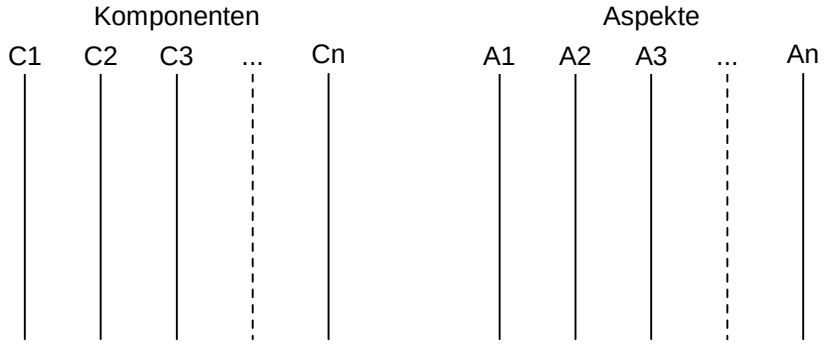


Abbildung 9-2: Separation von Komponenten und Aspekten

2. **Integration**. Im zweiten Schritt werden die fertig implementierten Komponenten und Aspekte anhand der Integrationsregeln zusammengefügt (engl. compose), um damit das Gesamtsystem zu erzeugen. Dies illustriert die Abbildung 9-3. Dabei kann der gleiche Aspekt mit allen oder nur mit einigen Komponenten interagieren und umgekehrt.

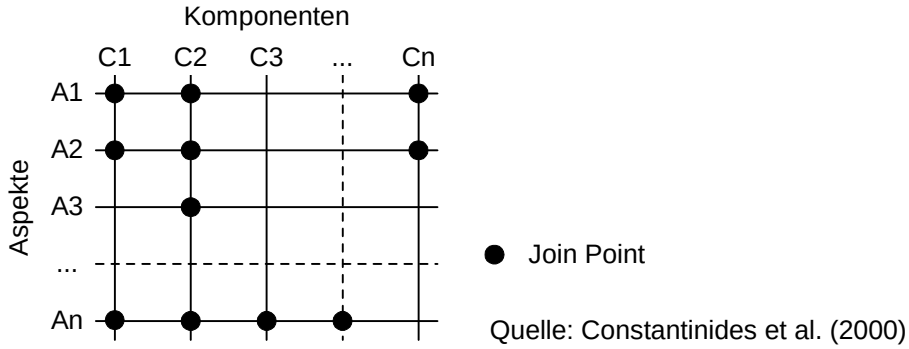


Abbildung 9-3: Integration von Komponenten und Aspekten

[9] www.java.sun.com (22.07.2004)

9.2.2 Integration von Komponenten und Aspekten

Die Integration wird auch mit *Weaving* (engl. Weben) bezeichnet. Sie funktioniert im Prinzip wie in der Abbildung 9-4 dargestellt, indem die Aspekte und Komponenten einem *Weaver* (engl. Weber) übergeben werden, der daraus das lauffähige Gesamtsystem erzeugt. Das Weaving erfolgt nach Constantinides et al. (2000) entweder *statisch* zur Übersetzungszeit durch einen Precompiler, der aus den Sprachkonstrukten der Aspektsprache Sprachkonstrukte der Komponentensprache erzeugt, oder *dynamisch* zur Laufzeit durch das Laufzeitsystem.

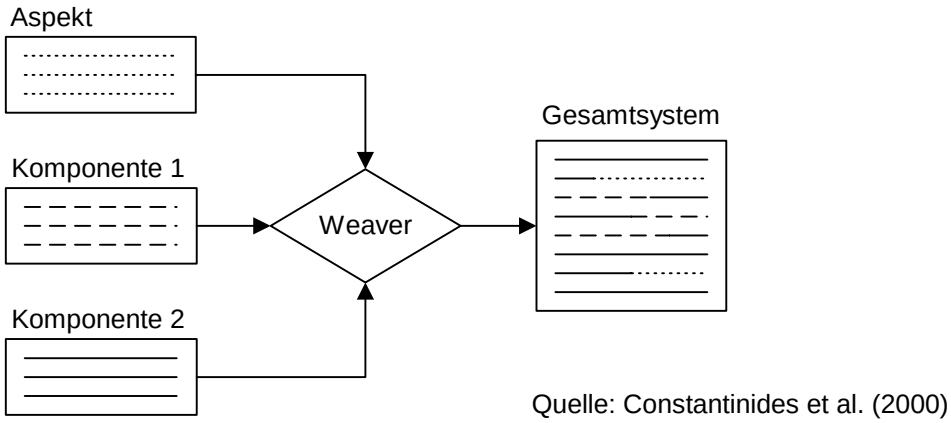


Abbildung 9-4: Weaving von Komponenten und Aspekten

Art und Zeitpunkt des *Weaving* begründen – neben den Sprachen zur Implementierung der Aspekte – die hauptsächlichen Unterscheidungsmerkmale zwischen den verschiedenen Ansätzen. Constantinides et al. (2000) wenden die folgenden Kriterien zur Unterscheidung und Klassifizierung der Ansätze der aspektorientierten Programmierung an:

- **Sprache**. Werden die Aspekte in einer *Anwendungsbereich-spezifischen* oder einer *universellen* Aspektsprache implementiert, oder können *bestehende* (z.B. objektorientierte) Sprachen verwendet werden? In der Anfangszeit der aspektorientierten Programmierung dominierten Anwendungsbereich-spezifische Sprachen (siehe Kiczales et al., 1997), während die neueren Sprachen universell sind. Anwendungsbereich-spezifische Sprachen werden hier nicht weiter besprochen.
- **Art des Weaving**. Erfolgt das Weaving *statisch* (wie z.B. in AspectJ) oder *dynamisch* (wie z.B. in reflexiven Ansätzen)? Ersteres hat den Vorteil besserer Performance, während Letzteres den Vorteil grösserer Flexibilität hat.
- **Quellcode-Transformation**. Ist eine Transformation des Quellcodes notwendig (wie z.B. in AspectJ) oder nicht (wie z.B. in reflexiven Ansätzen)? Quellcode-Transformation bedeutet, dass der (getrennte) Quellcode von Komponenten und Aspekten vor der eigentlichen Übersetzung in einen aus Komponenten und Aspekten bestehenden (vermischten) Quellcode transformiert wird.
- **Zeitpunkt des Weaving**. Erfolgt das Weaving *vor der Übersetzung* durch einen Precompiler (wie z.B. in AspectJ), zur *Übersetzungszeit* (wie z.B. im Aspect Moderator Framework) oder zur *Laufzeit* (wie z.B. in reflexiven Ansätzen)? Je nachdem wird die Separation of Concerns länger oder weniger lange bewahrt.

Auf diese Kriterien wird bei der Behandlung der einzelnen Ansätze hingewiesen.

Es gibt in der Literatur einige Autoren, die unter Weaving ausschliesslich das statische, nicht aber das dynamische Weaving verstehen (z.B. Diaz Pace, Campo, 2001). Dies entspricht nicht dem Begriffsverständnis in dieser Arbeit.

9.2.3 Linguistische und objektorientierte Ansätze

Diaz Pace, Campo (2001) teilen die Ansätze in *linguistische* und *objektorientierte* Ansätze ein. Alle in diesem Abschnitt erwähnten Ansätze werden in diesem Kapitel – in der gleichen Reihenfolge – detailliert beschrieben und im Abschnitt 9.12 miteinander verglichen.

Bei den **linguistischen Ansätzen** wird eine Menge neuer Sprachkonstrukte definiert, mit deren Hilfe die Aspekte implementiert werden können. Ferner muss ein Weaver zur Verfügung gestellt werden, der die Aspekte in die Komponenten einwebt und damit das Gesamtsystem erzeugt. Wegen der zusätzlichen Sprachkonstrukte erfolgt das Weaving in allen Ansätzen statisch durch einen Precompiler vor der eigentlichen Übersetzung. Damit ist eine Quellcode-Transformation notwendig. Die Tabelle 9-1 listet die in diesem Kapitel besprochenen linguistischen Ansätze auf, die alle eine gewisse Verbreitung in der Literatur gefunden haben. Der bedeutendste unter ihnen, die aspektorientierte Programmierung, gibt gleich dem ganzen Gebiet seinen Namen.

linguistischer Ansatz, Sprache	einführende Literatur
Aspektorientierte Programmierung, AspectJ	Kiczales et al. (2001a)
Adaptive Programmierung, DemeterJ	Orleans, Lieberherr (2001)
Multi-Dimensional Separation of Concerns (MDSOC), Hyper/J	Ossher, Tarr (2001)
Composition Filters (CF)	Bergmans, Aksit (2001a)

Tabelle 9-1: Linguistische Ansätze der aspektorientierten Programmierung

Bei den **objektorientierten Ansätzen** wird auf die Definition neuer Sprachkonstrukte verzichtet. Die Aspekte werden mit bewährten Mitteln der Objektorientierung implementiert, z.B. als (Meta-) Klassen. Das Weaving erfolgt statisch zur Übersetzungszeit (z.B. im Aspect Moderator Framework) oder dynamisch zur Laufzeit (z.B. bei den Metaobjektprotokollen). Eine Quellcode-Transformation ist in beiden Fällen nicht notwendig. Im Unterschied zu den linguistischen Ansätzen existiert bei den objektorientierten Ansätzen keine definierte Aspektsprache, die in diesem Kapitel besprochen werden könnte, ganz im Gegenteil: Es sind unzählige Varianten von objektorientierten Ansätzen denkbar. Es werden daher nicht einzelne Ansätze besprochen, sondern Gruppen von Ansätzen, aus denen zur Illustration jeweils ein Beispiel (siehe in der Tabelle 9-2) ausgewählt wird.

objektorientierter Ansatz	einführende Literatur
Frameworks am Beispiel des Aspect Moderator Framework (AMF)	Constantinides et al. (2000)
Reflexive Ansätze am Beispiel der Anwendung eines Metaobjektprotokolls mit MetaJava	Völter (2001)

Tabelle 9-2: Objektorientierte Ansätze der aspektorientierten Programmierung

Ein **Vergleich zwischen linguistischen und objektorientierten Ansätzen** zeigt, dass die Ansätze in Bezug auf die Erreichung der im Kapitel 1 erwähnten Ziele (Verständlichkeit, Pflegbarkeit, Wiederverwendbarkeit, (Rück-) Verfolgbarkeit) praktisch gleichwertig sind. Die wesentlichen Unterschiede liegen in anderen Bereichen. Die Tabelle 9-3 stellt die Vor- und Nachteile von linguistischen Ansätzen und Frameworks zusammen, wie sie teilweise auch in Constantinides et al. (2000) und Diaz Pace, Campo (2001) erwähnt sind. Zu bemerken ist, dass sich diese Vor- und Nachteile auf die heutige Situation beziehen. In ein paar Jahren kann sich die Situation ganz anders präsentieren; vielleicht werden einst aspektorientierte Konzepte vollständig in objektorientierten Sprachen integriert sein.

	linguistische Ansätze	Frameworks
Vorteile	• sofort einsetzbar • leicht erlernbar • sukzessive einführbar • bessere Performance durch statisches Weaving (*Inlining*)	• flexible Konstrukte • Offenheit • Erweiterbarkeit • Sprachneutralität • Verwendung bestehender Sprachen und Werkzeuge • Langlebigkeit
Nachteile	• eingeschränkter Sprachumfang • Integration der Sprachen in Entwicklungswerkzeuge erforderlich • Kurzlebigkeit	• schlechtere Performance bei dynamischem Weaving • beträchtliche Einarbeitungszeit • Umgehbarkeit der Separation of Concerns

Tabelle 9-3: Vergleich von linguistischen und objektorientierten Ansätzen

Die reflexiven Ansätze lassen sich weder eindeutig auf die Seite der linguistischen Ansätze noch eindeutig auf die Seite der Frameworks schlagen. Sie haben ähnliche Vor- und Nachteile wie die Frameworks, jedoch zusätzlich den Nachteil eines eingeschränkten Sprachumfangs, was die reflexiven Sprachkonstrukte (z.B. Java Reflection) betrifft, die nicht in jeder objektorientierten Programmiersprache gleich mächtig sind.

9.2.4 Beurteilungskriterien

Zur Beurteilung der Ansätze werden die folgenden Kriterien angewendet:

- Ist der Ansatz gut verständlich und einfach zu erlernen?
- Wie lässt sich der Ansatz mit anderen Ansätzen vergleichen?
- Weisen die resultierenden Aspekte die im Abschnitt 7.4 erwähnten Eigenschaften (separierbar und zusammensetzbar, additiv, transparent, aktiv) auf, die vom gewählten Ansatz abhängen? (Wird eine Eigenschaft in der Beurteilung nicht erwähnt ist, gilt sie als erfüllt.)

9.3 Aspektorientierte Programmierung (AOP) und AspectJ

Während in der Anfangszeit der aspektorientierten Programmierung Anwendungsbereich-spezifische Sprachen dominierten (siehe Kiczales et al., 1997), ist AspectJ universell einsetzbar. AspectJ ist eine Erweiterung von Java und damit ein linguistischer Ansatz. Das Weaving erfolgt statisch durch einen Aspektübersetzer, der Java-Quellcode erzeugt, bevor der eigentliche Java-Übersetzer zum Zug kommt. Zu beachten ist, dass Aspekte – im Gegensatz zu Klassen – nicht für sich allein übersetzt werden können, sondern nur gemeinsam mit den Komponenten, die sie quer schneiden.

Unter den Ansätzen der aspektorientierten Programmierung ist AspectJ der bedeutendste. Dies lässt sich daran erkennen, dass viele andere Ansätze mit AspectJ verknüpft werden. So wird AspectJ um Konstrukte für die adaptive Programmierung erweitert (siehe im Abschnitt 9.4), und die Composition Patterns werden auf AspectJ abgebildet (siehe im Abschnitt 10.2). Ausserdem finden die anderen Ansätze mit Ausnahme von Hyper/J in der neueren Literatur kaum mehr Erwähnung. Aus diesem Grund basiert insbesondere der Teil III dieser Arbeit auf der Terminologie von AspectJ.

Die folgende Einführung in AspectJ stammt grösstenteils aus dem gut aufgebauten und mit vielen Beispielen illustrierten Buch „AspectJ in Action" von Laddad (2003). Ausnahmen sind vermerkt. Daneben sind auch „An Overview of AspectJ" von Kiczales et al. (2001a) und "Getting Started with AspectJ" von Kiczales et al. (2001b) als Einstieg zu empfehlen. AspectJ ist eine mächtige

Sprache; die nachfolgende Beschreibung kann bestenfalls die wichtigsten Konzepte von AspectJ kurz erläutern und damit einen ersten Eindruck vermitteln.

9.3.1 Warum es die Aspektorientierung braucht

Die Objektorientierung hat sich für die Modularisierung der Core Concerns[10] bestens bewährt, versagt aber, wenn es um die Modularisierung von Crosscutting Concerns geht. Warum? Wie im Abschnitt 8.2 erwähnt, sind Crosscutting Concerns häufig in zwei Teilen implementiert, einem *Server*-Teil und einem *Client*-Teil.

Der **Server-Teil** (z.B. ein Autorisierungs-Modul) ist in der objektorientierten Programmierung mit Klassen und Interfaces sauber modularisierbar. Die Verwendung von Interfaces führt zu einer losen Kopplung zwischen den Client-Klassen und den Server-Klassen, die das Interface implementieren. Den Client-Klassen ist nicht bewusst, welche Implementierung sie genau benutzen, und damit können die Implementierungen problemlos verändert oder sogar ausgetauscht werden.

Demgegenüber muss der **Client-Teil** (z.B. eine Anfrage an das Autorisierungs-Modul) in jeder einzelnen Komponente, die das „crosscutting" Verhalten benötigt, implementiert werden. Damit ist die Implementierung des Client-Teils über mehrere Komponenten verstreut (engl. scattered) und mit den Komponenten vermischt (engl. tangled). Genau hier setzt die Aspektorientierung ein. Dank der aspektorientierten Programmierung benötigen die Komponenten keinerlei Wissen über die Existenz von Crosscutting Concerns, und keine Komponente muss den Client-Teil eines Crosscutting Concern implementieren. Sämtliche Änderungen und sogar die Ein- und Ausschaltung von Aspekten können lokal im Server-Teil oder im Aspekt selbst abgehandelt werden.

9.3.2 Vorgehen

Eine aspektorientierte Implementierung erfolgt in drei Schritten:

1. **Aspektbezogene Separation**[11]. In diesem Schritt werden die Core Concerns (z.B. Geschäftslogik) und die Crosscutting Concerns (z.B. Logging, Autorisierung etc.) identifiziert und voneinander getrennt.
2. **Implementierung der Concerns**. Jeder Concern wird unabhängig vom anderen mit konventionellen Mitteln implementiert. Es entstehen also ein Geschäftslogik-Modul, ein Logging-Modul, ein Autorisierungs-Modul etc. Das Autorisierungs-Modul umfasst vielleicht ein Interface, einige konkrete Implementierungen und eine Klasse, welche die Auswahl der Implementierung verbirgt. Beachte: Viele Crosscutting Concerns haben einen konventionell implementierten Kern!
3. **Aspektbezogene Integration**[12]. In diesem Schritt werden die Integrationsregeln in Form von Aspekten definiert. Die eigentliche Integration (das *Weaving*) erfolgt später beim Übersetzen, und der *Aspect Weaver* benutzt diese Regeln, um das Gesamtsystem zu erzeugen.

In den nächsten Abschnitten werden nun die wichtigsten Konzepte von AspectJ vorgestellt:

9.3.3 Join Points

Join Points sind bestimmte Stellen in den Komponenten (welche die Core Concerns und den Server-Teil der Crosscutting Concerns implementieren), an denen die Aspekte (welche den Client-

[10] Laddad (2003) verwendet in den ersten Kapiteln seines Buchs häufig *Crosscutting Concern* anstelle von Aspekt.

[11] Laddad (2003) verwendet in diesem Zusammenhang den Begriff *Dekomposition*.

[12] Laddad (2003) verwendet in diesem Zusammenhang den Begriff *Rekomposition*.

Teil der Crosscutting Concerns implementieren) andocken können. In diesem Sinne repräsentieren Join Points die in der Abbildung 9-3 dargestellten Kreuzungspunkte zwischen Komponenten und Aspekten. Am besten stellt man sich einen Join Point als eine *(Folge von) Anweisung(en)* vor. Weil sie definieren, wo das „crosscutting" Verhalten eingewoben (engl. woven) werden soll, sind die Join Points ein fundamentales Konzept der aspektorientierten Programmierung.

> **Definition a)** *Join Point*. Element der Komponenten-Sprachsemantik, mit dem sich die Aspekt-Programme koordinieren (Kiczales et al., 1997).
>
> **Definition b)** *Join Point*. Eine identifizierbare Stelle in der Ausführung eines Programms (Laddad, 2003).

In der älteren Definition a) ist ein Join Point ausdrücklich einer *Komponente* zugeordnet, während die neuere Definition b) den Join Point allgemeiner als „Stelle in der Ausführung eines *Programms*" beschreibt. Die Definition b) erscheint zwar unpräziser, trifft aber den Kern der Sache besser, was AspectJ betrifft. Damit in AspectJ auch Aspekte implementiert werden können, die andere Aspekte quer schneiden, betreffen Join Points nicht nur Komponenten, sondern auch Aspekte. *Programm* umfasst also in der Definition b) sowohl Komponenten als auch Aspekte.

Die in einem Aspekt gewünschten Join Points werden durch *Pointcuts* (siehe im Abschnitt 9.3.4) definiert.

Es gibt verschiedene **Kategorien von Join Points**. Wie die Tabelle 9-4 zeigt, unterscheiden sie sich durch ihren Wirkungsbereich.

Kategorie	Wirkungsbereich
Methode	Es gibt zwei Arten von Join Points für Methoden: • Aufruf (engl. call): Der Join Point betrifft den Aufruf einer bestimmten Methode. • Ausführung (engl. execution): Der Join Point betrifft die Ausführung (den Rumpf) einer bestimmten Methode. Diese beiden Join Points werden am häufigsten verwendet.
Konstruktor	Es gibt zwei Arten von Join Points für Konstruktoren: • Aufruf: Der Join Point betrifft den Aufruf eines bestimmten Konstruktors. • Ausführung: Der Join Point betrifft die Ausführung eines bestimmten Konstruktors.
Variablen-zugriff	Es gibt zwei Arten von Join Points für Variablenzugriffe: • Lesezugriff: Der Join Point betrifft das Ansprechen einer bestimmten Variablen. • Schreibzugriff: Der Join Point betrifft die Wertzuweisung an eine bestimmte Variable.
Ausnahme-behandlung	Ein Join Point für Ausnahmebehandlungen betrifft den `catch`-Block einer bestimmten Ausnahme.
Initialisierung	Es gibt zwei Arten von Join Points für Initialisierungen: • Klasse: Der Join Point betrifft das Laden einer bestimmten Klasse und das Initialisieren ihrer statischen Variablen. • Objekt: Der Join Point betrifft das Initialisieren eines bestimmten Objekts.
Vor-Initialisierung	Ein Join Point für Vor-Initialisierungen betrifft den Aufruf von `super()` im Konstruktor eines bestimmten Objekts. Dieser Join Point ist selten.
Advice-Ausführung	Ein Join Point für Advice-Ausführungen betrifft die Ausführung eines beliebigen Advice (siehe im Abschnitt 9.3.6).

Tabelle 9-4: Kategorien von Join Points

Von praktischer Bedeutung sind vor allem die Methoden-Join Points `call` und `execution`. Auf sie wird im weiteren Verlauf dieser Arbeit immer wieder Bezug genommen. Alle übrigen Join Points kommen in den Anwendungsbeispielen selten bis nie vor.

9.3.4 Pointcuts

Pointcuts spezifizieren eine Menge von *Join Points*, an denen das „crosscutting" Verhalten eingewoben werden soll.

> **Definition** *Pointcut*. Ein Programmkonstrukt, das eine Menge von *Join Points* definiert und sich bei Bedarf Kontextinformation an diesen Join Points beschafft.

Der Unterschied zwischen Join Points und Pointcuts ist am Anfang nicht leicht zu verstehen. Man kann sich das so vorstellen, dass die Pointcuts die abstrakten Weaving-Regeln definieren, während die Join Points konkrete Situationen darstellen, die diesen Weaving-Regeln entsprechen. Ein einfacher Pointcut lautet zum Beispiel:

```
call (void Account.debit (float))
```

Dieser Pointcut definiert, dass das „crosscutting" Verhalten überall dort eingewoben werden soll, wo die `debit()`-Methode der `Account`-Klasse aufgerufen wird. Die äussere Klammer enthält die Signatur dieser Methode. Ein solcher Pointcut könnte verwendet werden, um in einer Bankanwendung sämtliche Kontobelastungen in einem Logbuch festzuhalten.

Anonyme und benannte Pointcuts. Pointcuts können *anonym* oder *benannt* sein. Beim obigen Beispiel handelt es sich um einen anonymen Pointcut. Er wird – ähnlich wie eine anonyme Klasse – am Ort seiner Verwendung definiert und kann später nicht mehr angesprochen werden. Ein semantisch gleicher benannter Pointcut könnte wie folgt aussehen:

```
pointcut debitPayments() : call (void Account.debit (float))
```

Der Ausdruck nach dem Doppelpunkt wird auch als *Pointcut-Definition* bezeichnet. Ein benannter Pointcut kann später in einem *Advice* (siehe im Abschnitt 9.3.6) über seinen Namen (`debitPayments`) wieder angesprochen werden.

Eigenschaftsbasierte Pointcuts. Damit in einem Pointcut nicht alle Methoden einzeln aufgezählt werden müssen, können in der Methodensignatur folgende *Wildcards* für Join Points mit gemeinsamen Eigenschaften verwendet werden: Der Stern (`*`) steht für eine beliebige Folge von Zeichen exkl. Punkt, zwei Punkte (`..`) stehen für eine beliebige Folge von Zeichen inkl. Punkt, das Pluszeichen (`+`) steht für eine Subklasse oder ein Subinterface. Der anonyme Pointcut

```
call (* Account.* (..))
```

bedeutet beispielsweise, dass ein Join Point identifiziert (engl. capture) wird, sobald irgendeine Methode der `Account`-Klasse mit beliebigen Argumenten und beliebigen Rückgabewerten aufgerufen wird.

Operatoren. Um Pointcuts miteinander zu verknüpfen, stehen die gleichen Operatoren wie bei Java zur Verfügung: der unäre Operator ! für die Negation, die binären Operatoren || für die *oder*-Verknüpfung und && für die *und*-Verknüpfung.

Arten. Es gibt zwei grundlegend verschiedene Arten von Pointcuts. Die so genannten *kategorisierten* (engl. kinded) *Pointcuts* identifizieren *eine Menge gleichartiger Join Points* aus einer der im Abschnitt 9.3.3 erwähnten Kategorien, z.B. lauter Methodenaufrufe oder lauter Lesezugriffe auf Variablen. Die Tabelle 9-5 zeigt die Syntax der in AspectJ verwendbaren kategorisierten Pointcuts.

Name = Join Point-Kategorie	Syntax
Methodenaufruf	`call(Methodensignatur)`
Methodenausführung	`execution(Methodensignatur)`
Konstruktoraufruf	`call(Konstruktorsignatur)`
Konstruktorausführung	`execution(Konstruktorsignatur)`
Lesezugriff auf eine Variable	`get(Variablensignatur)`
Schreibzugriff auf eine Variable	`set(Variablensignatur)`
Ausnahmebehandlung	`handler(Typsignatur)`
Initialisierung einer Klasse	`staticinitialization(Typsignatur)`
Initialisierung eines Objekts	`initialization(Konstruktorsignatur)`
Vor-Initialisierung	`preinitialization(Konstruktorsignatur)`
Advice-Ausführung	`adviceexecution()`

Tabelle 9-5: Kategorisierte Pointcuts

Alle übrigen Pointcuts identifizieren *sämtliche Join Points innerhalb eines bestimmten Bereichs*, z.B. alle Join Points innerhalb eines Kontrollflusses oder Quellcodesegments. Die Tabelle 9-6 zeigt die Syntax und den Wirkungsbereich der in AspectJ verwendbaren übrigen Pointcuts.

Name	Syntax	Wirkungsbereich
Kontroll-fluss	`cflow(Pointcut)`	Alle Join Points im Kontrollfluss der Join Points eines anderen Pointcut *inkl.* der Join Points dieses Pointcut.
	`cflowbelow (Pointcut)`	Alle Join Points im Kontrollfluss der Join Points eines anderen Pointcut *exkl.* der Join Points dieses Pointcut.
Quellcode	`within (Klasse|Aspekt)`	Alle Join Points im Quellcode einer bestimmten Klasse oder eines bestimmten Aspekts (siehe im Abschnitt 9.3.9).
	`withincode (Signatur)`	Alle Join Points im Quellcode einer bestimmten Methode oder eines bestimmten Konstruktors.
Aus-führungs-objekt	`this(Klasse)`	Alle Join Points, deren aktuelles Objekt Instanz einer bestimmten Klasse ist.
	`target(Klasse)`	Alle Join Points (z.B. Methodenaufrufe), deren Zielobjekt Instanz einer bestimmten Klasse ist. Dieser Pointcut wird häufig in Kombination mit einem `call`-Pointcut verwendet.
		Beide Pointcuts können verwendet werden, um Kontextinformation zu beschaffen.
Argument	`args(Typ)`	Alle Join Points (z.B. Methodenaufrufe) mit einem bestimmten Argumenttyp. Dieser Pointcut kann verwendet werden, um Kontextinformation zu beschaffen.
Bedingungs prüfung	`if(boolescher Ausdruck)`	Alle Join Points, die eine bestimmte Bedingung erfüllen.

Tabelle 9-6: Übrige Pointcuts

Die **Kontrollfluss-Pointcuts** sind anfänglich nicht leicht zu verstehen und werden daher in der Abbildung 9-5 anhand eines UML-Sequenzdiagramms illustriert.

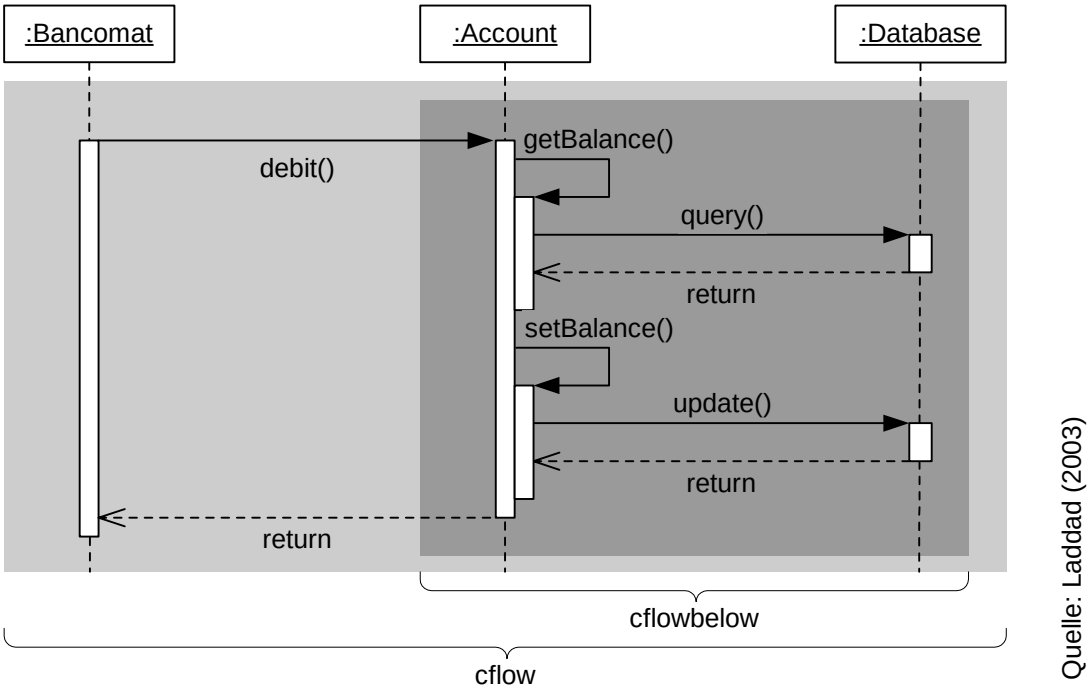


Abbildung 9-5: Die `cflow`- und `cflowbelow`-Pointcuts

Der `cflow`-Pointcut (hellgrau schattiert) umfasst alle Join Points im Kontrollfluss irgendeines Aufrufs der `debit()`-Methode der `Account`-Klasse *einschliesslich* des Aufrufs der `debit()`-Methode selbst:

```
cflow (call (* Account.debit (..)))
```

Der `cflowbelow`-Pointcut (dunkelgrau schattiert) umfasst die gleichen Join Points, jedoch *nicht* den Aufruf der `debit()`-Methode selbst:

```
cflowbelow (call (* Account.debit (..)))
```

Eine typische Anwendung von `cflowbelow` ist die Unterscheidung zwischen dem ersten und weiteren rekursiven Methodenaufruf(en). Ein Beispiel:

```
call (<rekursiveMethode>) && !cflowbelow (call (<rekursiveMethode>))
```

Dieser Pointcut identifiziert jeweils nur den ersten Aufruf von `<rekursiveMethode>` in einem Kontrollfluss. Alle weiteren Aufrufe innerhalb des gleichen Kontrollflusses werden dank `!cflowbelow` ignoriert.

Im Zusammenhang mit den **Ausführungsobjekt-Pointcuts** spielen zwei Objekte eine Rolle: Das *aktuelle Objekt* (engl. current object) wird mit `this` bezeichnet. Es entspricht dem aufrufenden Objekt, das eine bestimmte Nachricht sendet. Das *Zielobjekt* (engl. target object) wird mit `target` bezeichnet. Es entspricht dem aufgerufenen Objekt, das eine bestimmte Nachricht empfängt.

9.3.5 Dynamisches und statisches Crosscutting

In AspectJ gibt es zwei verschiedene Arten, wie Aspekte Komponenten quer schneiden können. Sie werden als *dynamisches* und *statisches Crosscutting* bezeichnet (nicht zu verwechseln mit statischem und dynamischem Weaving). *Dynamisches Crosscutting* entspricht dem klassischen Fall. Es erlaubt, die Programmausführung mit Hilfe von Advice (siehe im Abschnitt 9.3.6) an wohl definierten Punkten um zusätzliches Verhalten zu erweitern, und verändert damit das gesamte Verhalten des Systems zur Laufzeit. Demgegenüber verändert das s*tatische Crosscutting* nicht das Verhalten, sondern lediglich die statische Struktur des Systems zur Übersetzungszeit. Es unterstützt die Implementierung von dynamischem Crosscutting, indem mit Hilfe von *Introductions* (siehe im Abschnitt 9.3.7) neue Variablen und Methoden zu bestehenden Klassen und Interfaces hinzugefügt werden können, oder es erzeugt mit Hilfe von *Declarations* (siehe im Abschnitt 9.3.8) zur Übersetzungszeit Komponenten-übergreifende Warnungen und Fehlermeldungen.

9.3.6 Advice

Advice heisst auf Deutsch „Hinweis, Ratschlag“, wird aber im Zusammenhang mit AspectJ am besten mit „Anweisung“ übersetzt. Advice[13] sind die „Methoden der Aspekte“.

> **Definition** *Advice.* Code, der an einem Join Point auszuführen ist, der durch einen Pointcut identifiziert wurde.

Es gibt drei Arten von Advice: *Before Advice*, *After Advice* und *Around Advice*.

Before Advice werden, wie der Name sagt, *vor* den identifizierten Joint Points (z.B. Methodenaufrufen) ausgeführt. Das folgende Beispiel zeigt einen Before Advice, der mit Hilfe eines anonymen Pointcut alle Auftraggeber von Kontobelastungen authentisiert, bevor die eigentliche Kontobelastung durchgeführt wird. (Die Syntax der Advice-Rümpfe entspricht derjenigen von Methoden.)

```
before() : call (* Account.debit (..)) {
     //Advice-Rumpf mit Authentisierungsoperationen
}
```

After Advice werden *nach* den identifizierten Join Points ausgeführt, entweder immer (`after`), nur nach erfolgreicher Ausführung der Join Points (`after returning`) oder nur nach fehlerhafter Ausführung der Join Points (`after throwing`). Das folgende Beispiel zeigt einen After Advice, der alle erfolgreich durchgeführten Kontobelastungen in einem Logbuch festhält.

```
after returning() : call (* Account.debit (..)) {
     //Advice-Rumpf mit Logging-Operationen
}
```

Bei Bedarf kann im Advice-Rumpf auf die Rückgabe- bzw. Ausnahmeobjekte der Join Points zugegriffen werden, sofern sie in der Klammer nach `returning()` bzw. `throwing()` deklariert sind.

Around Advice umschliessen (engl. surround) die identifizierten Join Points. Dabei können sie die Join Points vollständig *ersetzen* (engl. bypass) oder um zusätzliches Verhalten *erweitern*. Im letzteren Fall bewirkt `proceed()` innerhalb des Advice-Rumpfs, dass das ursprüngliche Verhalten an den Join Points (z.B. Methodenaufrufe) ausgeführt wird. `proceed()` kann einmal oder mehrmals, mit den gleichen oder mit veränderten Argumenten aufgerufen werden.

Das UML-Sequenzdiagramm in der Abbildung 9-6 zeigt das Beispiel eines Around Advice, der sämtliche Kontobelastungen unterdrückt und durch ein anderes Verhalten, z.B. das Werfen einer Ausnahme, *ersetzt*.

[13] Advice existiert auf Englisch – wie information – nur im Singular. Daher erhalten Advice kein Plural-s.

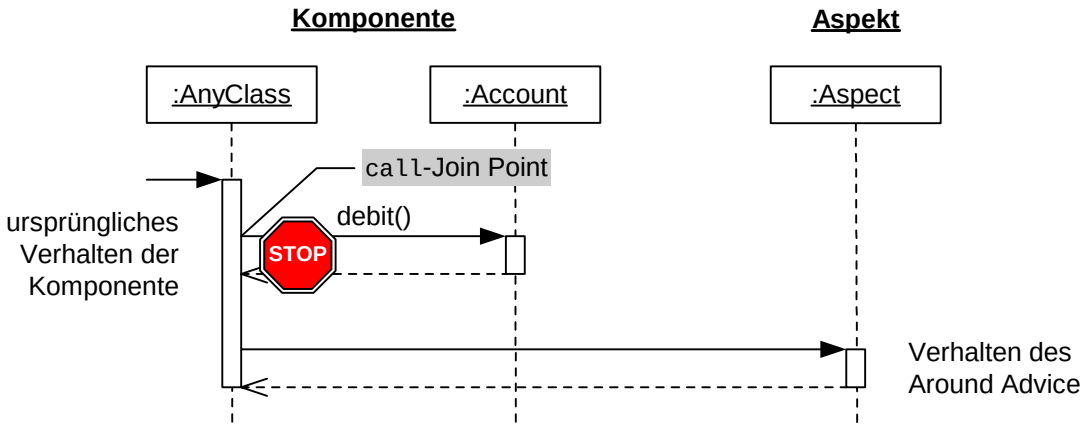


Abbildung 9-6: Around Advice, der die identifizierten Join Points ersetzt

Im folgenden Beispiel *erweitert* der Around Advice alle Kontobelastungen zu Beginn und am Ende ihrer Ausführung um Tracing-Verhalten. Dazwischen werden die Kontobelastungen unverändert durchgeführt.

```
void around() : execution (* Account.debit (..)) {
     //Tracing-Operationen
     proceed();
     //Tracing-Operationen
}
```

Das UML-Sequenzdiagramm in der Abbildung 9-7 stellt das gleiche Beispiel grafisch dar.

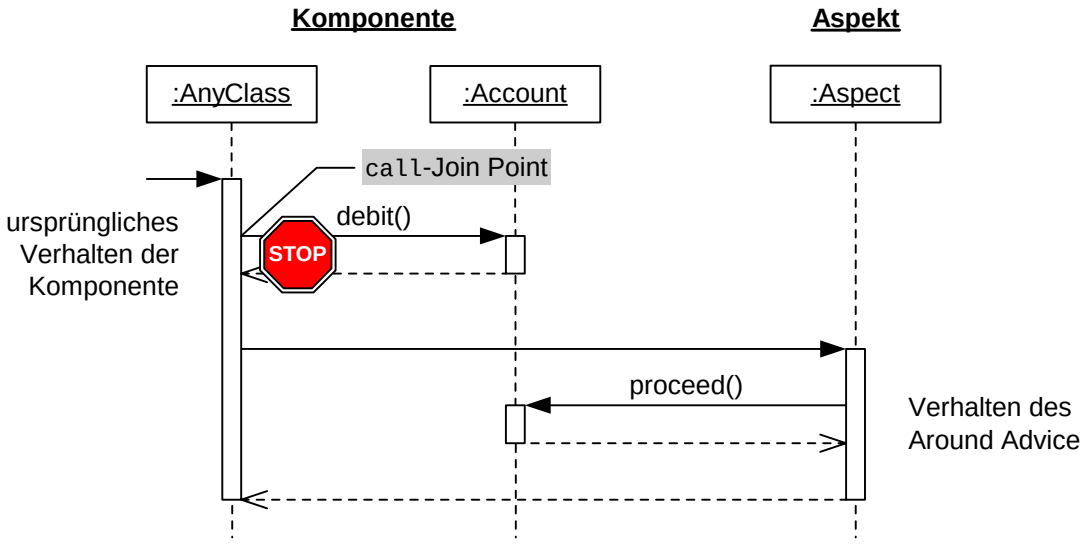


Abbildung 9-7: Around Advice, der die identifizierten Join Points erweitert

Da Around Advice Methodenaufrufe umschliessen können, müssen sie Rückgabewerte (im Beispiel: `void`) deklarieren. Ruft ein Around Advice an einem `call`-Join Point mit `proceed()` die ursprüngliche Methode auf, soll der Around Advice den Rückgabewert (oder das Rückgabeobjekt) dieser Methode unverändert zurückgeben. Warum dies so ist, wird im Abschnitt 14.4 erläutert.

Kontextinformation. Advice haben die Möglichkeit, auf Informationen aus dem Ausführungskontext der Join Points zuzugreifen. Werden benannte Pointcuts verwendet, müssen sich diese den Kontext beschaffen und den Advice zur Verfügung stellen. Ein Beispiel:

```
pointcut debitPayments (Account account, float amount) :
     call (void Account.debit (float))
     && target (account)
     && args (amount);

before (Account account, float amount) :
     debitPayments (account, amount) {
          //Advice-Rumpf mit Zugriffen auf account und amount
     }
```

Jedes Mal, wenn die `debit()`-Methode der `Account`-Klasse aufgerufen wird, beschaffen sich die Pointcuts `target (account)` bzw. `args (amount)` das Zielobjekt bzw. das Argument und übergeben diese beiden Informationen dem Before Advice.

Reflexive Information über Join Points. Mittels Reflexion können alle Advice auf statische und dynamische Informationen über die Join Points zugreifen. Diese Art reflexiver Information ist insbesondere bei der Implementierung von Logging- und ähnlicher Funktionalität von Bedeutung. In jedem Advice sind folgende Informationen verfügbar:

- `thisJoinPoint` liefert dynamische Informationen über den jeweiligen Join Point (z.B. einen Methodenaufruf): `getThis()` das aktuelle Objekt, `getTarget()` das Zielobjekt und `getArgs()` die Argumente.
- `thisJoinPointStaticPart` liefert statische Informationen über den jeweiligen Join Point: `getKind()` die Kategorie, `getSignature()` die Signatur und `getSourceLocation()` die Stelle im Quellcode.
- `thisEnclosingJoinPointStaticPart` liefert statische Kontextinformationen.

Beim Zugriff auf reflexive Informationen ist darauf zu achten, dass dynamische Informationen teurer sind als statische, da sie bei jeder Ausführung des Advice aktualisiert werden müssen.

Advice und Methoden. Advice und Methoden sind sich bis zu einem gewissen Grad ähnlich, unterscheiden sich aber in wesentlichen Punkten:

- Advice haben keine Namen.
- Advice können nicht direkt aufgerufen werden, das ist die Aufgabe des Laufzeitsystems.
- Die Sichtbarkeit von Advice kann nicht spezifiziert werden. (Das ist auch nicht nötig, da sie nicht direkt aufgerufen werden können.)
- Advice können auf reflexive Information zugreifen (siehe weiter oben in diesem Abschnitt).

9.3.7 Introductions

Mit den *(Member) Introductions* können Variablen oder Methoden zu einer Klasse oder einem Interface hinzugefügt werden. Introductions sind ein Konzept des statischen Crosscutting (siehe im Abschnitt 9.3.5). In einem Advice können diese Variablen dann angesprochen bzw. diese Methoden aufgerufen werden. Ein Beispiel:

```
private float Account.minimumBalance;

public float Account.getAvailableBalance() {
     return getBalance() – minimumBalance;
}
```

Die erste Anweisung fügt eine Variable `minimumBalance` zur `Account`-Klasse hinzu. Die zweite Anweisung fügt eine Methode `getAvailableBalance` zur `Account`-Klasse hinzu.

9.3.8 Declarations

Die *Declarations* sind wie die *Introductions* ein Konzept des statischen Crosscutting und haben ganz verschiedene Anwendungsbereiche. Mit `declare parents` kann die Typenhierarchie verändert werden. Die Anweisung

```
declare parents : Account implements BankingEntity;
```

führt beispielsweise dazu, dass die `Account`-Klasse zusätzlich das `BankingEntity`-Interface implementieren muss. Mit `declare precedence` kann die Priorität der Aspekte (siehe im Abschnitt 9.3.9) untereinander geregelt werden. Zum Beispiel führt

```
declare precedence : AuthenticationAspect, AuthorizationAspect;
```

dazu, dass die Advice des Authentisierungsaspekts an jedem Join Point vor den Advice des Autorisierungsaspekts ausgeführt werden. Ferner können mit `declare warning` bzw. `declare error` Warnungen bzw. Fehlermeldungen bei der Übersetzung erzeugt werden, um damit Programmierrichtlinien (z.B. Methoden, die nicht aufgerufen werden dürfen) durchzusetzen. Dies wird auch mit *Policy Enforcement* bezeichnet. Das Beispiel

```
declare warning : call (void Persistence.save (Object)) :
     “Consider using Persistence.saveOptimized()”;
```

weist den Entwickler darauf hin, dass anstelle der `save`-Methode die `saveOptimized`-Methode vorzuziehen ist. Schliesslich können mit `declare soft` geprüfte (engl. checked) Ausnahmen abgeschwächt (engl. softened) werden. In Java gibt es geprüfte und ungeprüfte Ausnahmen. Geprüfte Ausnahmen müssen entweder in einem `try/catch`-Block abgefangen oder mit `throws <Exception>` weitergereicht werden. Bei ungeprüften Ausnahmen (`RuntimeException` oder `Error`), die jederzeit auftreten können, ist das nicht nötig. Die Anweisung

```
declare soft : <Exception> : <call-Pointcut>;
```

erlaubt, geprüfte Ausnahmen, die an einem bestimmten Join Point geworfen werden, wie ungeprüfte zu behandeln. Der AspectJ-Precompiler packt dabei die betroffenen Methodenaufrufe in einen `try/catch`-Block, der eine `SoftException` (erweitert `RuntimeException`) wirft. Das Abschwächen von Ausnahmen unterstützt die Modularisierung von Fehlerbehandlungsaspekten, ist jedoch relativ gefährlich, weil unter Umständen Ausnahmen abgeschwächt werden, die eigentlich behandelt werden müssten.

9.3.9 Aspekte

Die *Aspekte* schliesslich sind die zentrale Einheit von AspectJ, so wie die Klassen die zentrale Einheit von Java sind. Aspekte fassen die Weaving-Regeln mit Hilfe von Pointcuts, Advice, Introductions, Declarations und gewöhnlichem Java-Code zusammen. Der folgende Aspekt wurde aus einigen der obigen Beispiele zusammengefügt:

```
public aspect ExampleAspect {
     after returning() : call (* Account.debit (..)) {
          System.out.println (“Debit operation performed”);
     }
     declare parents : Account implements BankingEntity;

     declare warning : call (void Persistence.save (Object)):
          “Consider using Persistence.saveOptimized()”;
}
```

Aspekte werden üblicherweise so entworfen, dass zuerst die Join Points festgelegt werden, deren Verhalten ersetzt oder erweitert werden soll. Anschliessend werden die Advice spezifiziert.

Abstrakte Aspekte. Aspekte können – ähnlich wie Klassen – mit dem Schlüsselwort `abstract` als abstrakt deklariert werden. Dies erlaubt, Implementierungsdetails in konkrete Aspekte zu verlagern. Die abstrakten Aspekte werden dabei als *Basisaspekte* bezeichnet, die konkreten Aspekte als *Subaspekte*. Das Grundschema eines abstrakten Aspekts besteht aus einem abstrakten Pointcut, der in einem Subaspekt konkretisiert werden muss, und einem konkreten Advice. Ein Beispiel:

```
public abstract aspect AbstractLogging {

    public abstract pointcut logPoints();

    before() : logPoints() {
        //Advice-Rumpf mit konkreten Logging-Operationen
    }
}
```

Ein solcher abstrakter Aspekt eignet sich ausgezeichnet zur Wiederverwendung (siehe auch im Abschnitt 15.5): Das Logging-Verhalten, von dem angenommen wird, dass es anwendungsneutral ist, wird bereits im Basisaspekt konkret implementiert, während die vom Logging betroffenen Join Points anwendungsspezifisch sind und erst in den Subaspekten konkretisiert werden.

Aspektpriorität (engl. precedence). Wie im Abschnitt 9.3.8 erwähnt, kann die Priorität von Aspekten und damit die Reihenfolge, in der die Advice konkurrierender Aspekte an einem Join Point abgearbeitet werden, explizit mit `declare precedence` festgelegt werden. Eine solche *Declaration* kann in einen beliebigen Aspekt eingefügt oder in einen eigens dafür geschaffenen *Koordinations-Aspekt* verlagert werden. Da die erste Möglichkeit zu einer (unerwünschten) Kopplung der Aspekte führt, ist die zweite Möglichkeit vorzuziehen. Wird keine Aspektpriorität deklariert, ist die Reihenfolge der Advice zufällig mit einer Ausnahme: Die Subaspekte haben Priorität vor den Basisaspekten. Sind an einem Joint Point mehrere Aspekte abzuarbeiten, geschieht dies nach folgender Prioritätsregel: Je höher die Priorität eines Aspekts ist, desto weiter entfernt vom Join Point werden seine Advice ausgeführt, d.h. der Aspekt mit der höchsten Priorität kommt bei einem Before Advice zuerst und bei einem After Advice zuletzt an die Reihe. Dies illustriert die Abbildung 9-8.

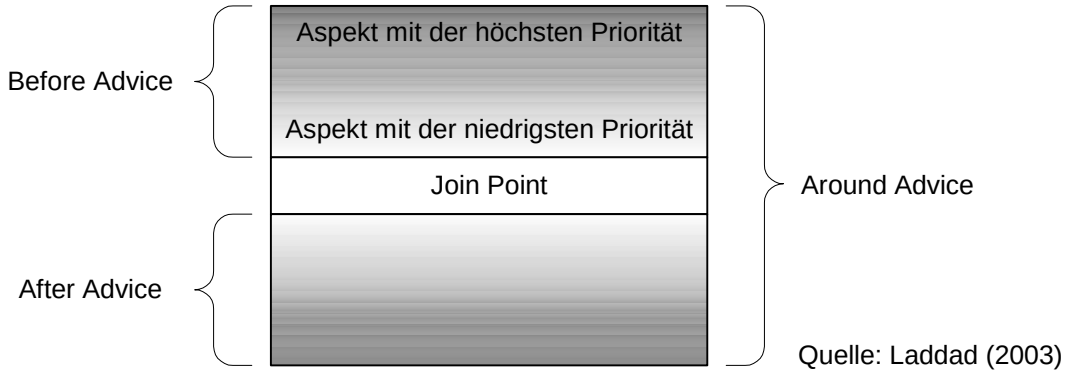


Abbildung 9-8: Prioritätsregel von Aspekten

Sind an einem Join Point mehrere Advice des gleichen Aspekts abzuarbeiten, geschieht dies anhand der Reihenfolge der Advice im Quellcode.

Kiczales et al. (2001a) beschreiben die Prioritätsregeln etwas pointierter:

- Wenn zwei Advice zum gleichen Aspekt gehören, hat der Advice Priorität, der im Quellcode des Aspekts zuerst deklariert wurde.

- Wenn der Aspekt A1 den Aspekt A2 erweitert (`extends`), haben die Advice von A1 Priorität vor den Advice von A2. (Hinter dem Überschreiben von Advice steckt die gleiche Idee wie hinter dem Überschreiben von Methoden.)
- Wenn der Aspekt A1 Priorität vor dem Aspekt A2 hat (`declare precedence`), haben die Advice von A1 Priorität vor den Advice von A2.
- In allen anderen Fällen (unabhängige Aspekte entsprechen dem Normalfall!) ist die relative Priorität undefiniert.

Aspektassoziationen. Im Normalfall existiert in einer virtuellen Maschine – ähnlich wie bei einer Singleton-Klasse – genau eine Instanz pro Aspekt, was in den meisten Fällen genügt. Der Zustand dieser Aspektinstanz wird von allen Objekten gemeinsam genutzt (engl. shared). Es gibt aber auch Fälle, in denen pro Objekt oder pro Kontrollflusssegment eine Aspektinstanz benötigt wird. Aus diesem Grund gibt es drei verschiedene so genannte *Aspektassoziationen* in AspectJ:

- Default ist die **Pro-virtuelle-Maschine-Assoziation**. Ein Pooling-Aspekt, dessen Ressourcen-Pool von allen beteiligten Objekten gemeinsam genutzt wird, ist ein Beispiel dafür.
- Eine **Pro-Objekt-Assoziation** ist erforderlich, wenn ein Aspekt Informationen über mehrere durch einen Pointcut identifizierte *Objekte* benötigt. Dies ist besonders bei abstrakten Aspekten wichtig, bei denen die vom Aspekt betroffenen Objekte nicht im Voraus bekannt sind und der *Introduction*-Mechanismus folglich nicht in Frage kommt. Mit `perthis`[14] bzw. `pertarget (<Pointcut>)` in der Aspektdeklaration wird für jedes aktuelle Objekt bzw. Zielobjekt, das durch den `Pointcut` identifiziert wird, eine neue Aspektinstanz erzeugt. Tatsächlich wird der Aspektzustand damit zu einem Teil des Objektzustands. Ein Aspekt zur Cache-Verwaltung, der sich für jedes Objekt im Cache den Zeitpunkt des letzten Zugriffs merken muss, ist ein Beispiel dafür.
- Eine **Pro-Kontrollfluss-Assoziation** ist erforderlich, wenn ein Aspekt Informationen über mehrere durch einen Pointcut identifizierte *Kontrollflusssegmente* benötigt. Mit `percflow` bzw. `percflowbelow (<Pointcut>)` in der Aspektdeklaration wird für jeden durch den `Pointcut` identifizierten Kontrollfluss eine neue Aspektinstanz erzeugt. Ein Aspekt zur Transaktionsverwaltung, der wissen muss, ob eine Operation innerhalb des Kontrollflusses einer anderen Operation läuft, ist ein Beispiel dafür.

Durch die Pro-Objekt- und Pro-Kontrollfluss-Assoziationen wird der Geltungsbereich (engl. scope) der Advice implizit eingeschränkt, und zwar auf die Join Points, mit denen eine Aspektinstanz verknüpft ist. Zwei statische Methoden, die in jedem Aspekt verfügbar sind, erlauben den Zugriff auf Aspektinstanzen: `aspectOf()` gibt die Aspektinstanz eines Objekts bzw. Kontrollflusses zurück, `hasAspect()` prüft, ob mit einem Objekt bzw. Kontrollfluss eine Aspektinstanz verknüpft ist.

Privilegierte Aspekte. Aspekte können mit dem Schlüsselwort `privileged` als privilegiert deklariert werden. Damit haben ihre Advice Zugriff auf `private` Variablen und Methoden von Klassen. Diese Möglichkeit sollte zurückhaltend eingesetzt werden, da sie die Kopplung zwischen dem Aspekt und den betroffenen Klassen verstärkt (siehe auch im Abschnitt 14.4).

Beispiele für Aspekte. Laddad (2003) bietet eine umfangreiche Sammlung von Aspektimplementierungen in folgenden Bereichen an:

- **Monitoring**. Darunter fallen Aspekte für das *Logging* bestimmter Methoden vor oder nach dem Aufruf, für das *Tracing* bestimmter Methoden zu Beginn oder am Ende der Ausführung, für das Logging bestimmter Ausnahmen mit `after throwing()` sowie für das *Profiling*, das Festhalten des Zeitbedarfs bestimmter Operationen.

[14] `this`, `target`, `cflow` und `cflowbelow` haben die gleiche Bedeutung wie im Abschnitt 9.3.4.

- **Durchsetzung von Programmierrichtlinien** (engl. policy enforcement). Was in einer konventionellen Umgebung nur mit Dokumentation, Ausbildung oder Code-Reviews erreichbar ist, kann mit AspectJ automatisiert werden. Einerseits werden `declare error` und `declare warning` angewendet. Damit können Regelverletzungen (z.B. Aufrufe unerwünschter Methoden, Verwendung unerwünschter Pakete, Zugriffe auf `public` oder `static` Variablen etc.) zwar bereits zur Übersetzungszeit erkannt werden, die Möglichkeiten sind aber beschränkt. Anderseits kommen Entwicklungsaspekte (siehe im Abschnitt 7.5) zum Einsatz, um die Einhaltung von Programmierrichtlinien, z.B. der *Ein-Thread-Regel* von Swing (siehe im Abschnitt 9.3.10), zu prüfen.
- **Nebenläufigkeit**. Dieser Bereich umfasst Aspekte für die Implementierung der *Ein-Thread-Regel* von Swing und für die *Lese-/Schreibsynchronisation*.
- **Optimierung**. Dazu gehören Aspekte für das *Caching* und das *Ressourcen-Pooling*. Dieses kann durch einen Aspekt implementiert werden, der mit Hilfe eines Around Advice den direkten Zugriff auf eine Ressource durch einen Zugriff auf den Ressourcen-Pool ersetzt.
- **Sicherheit**. *Authentisierung* und *Autorisierung* werden mit Hilfe von JAAS[15] (Java Authentication and Authorization Service) aspektorientiert implementiert. Dabei werden zwei Entwurfsmuster, das Arbeiterobjekt-Erzeugungsmuster und das Entwurfsmuster für neue Ausnahmen, eingesetzt (siehe im Abschnitt 9.3.10).
- **Transaktionsverwaltung**. In diesem Bereich wird eine auf JDBC[16] (Java Database Connectivity) und eine auf JTA[17] (Java Transaction API) basierende aspektorientierte Transaktionsverwaltung implementiert.
- **Geschäftsregeln** (engl. business rules). Es gibt unzählige Möglichkeiten, Geschäftsregeln als Aspekte zu implementieren. Laddad (2003) stellt Aspekte aus dem Bankbereich vor: Ein Aspekt stellt sicher, dass der Saldo von Sparkonti nicht unter die für Sparkonti geltende Limite fällt. Ein anderer Aspekt beschafft sich selbständig Geld von einem anderen Konto, wenn auf einem Konto zu wenig Geld für die Einlösung eines Checks vorhanden ist.
- **Fehlerbehandlung**. Aspekte zur Fehlerbehandlung werden mit Hilfe von `declare soft` oder mit Hilfe des Entwurfsmusters für neue Ausnahmen implementiert.

Aspekte und Klassen. Aspekte und Klassen sind sich offensichtlich ähnlich. In folgenden Punkten stimmen ihre Konzepte denn auch überein:

- Aspekte können Variablen und Methoden enthalten.
- Die Sichtbarkeit von Aspekten kann mit den Schlüsselwörtern `public`, `private`, `protected` bzw. `packaged` (Default) spezifiziert werden.
- Aspekte können abstrakt sein (siehe weiter oben in diesem Abschnitt).
- Aspekte können abstrakte Aspekte erweitern, aber auch – was eher unüblich ist – Interfaces implementieren und Klassen erweitern.
- Aspekte können als verschachtelte Aspekte in Klassen und Interfaces eingebettet werden.

Es gibt aber auch wesentliche Punkte, in denen sich Aspekte von Klassen unterscheiden:

- Aspekte können nicht direkt instanziert werden, das ist die Aufgabe des Laufzeitsystems.
- Aspekte können zwecks Reduktion der Komplexität des AspectJ-Precompilers nicht von konkreten Aspekten erben, sondern nur von abstrakten.
- Aspekte können als `privileged` markiert werden (siehe weiter oben in diesem Abschnitt).

[15] www.java.sun.com/products/jaas/ (23.07.2004)
[16] www.java.sun.com/products/jdbc/ (23.07.2004)
[17] www.java.sun.com/products/jta/ (23.07.2004)

9.3.10 Entwurfsmuster und Idiome

Ähnlich wie in der objektorientierten Programmierung gibt es auch in der aspektorientierten Programmierung erste Entwurfsmuster und Idiome, also Lösungen für wiederkehrende Entwurfsaufgaben mit Hilfe von Aspekten. Zu den in Laddad (2003) vorgestellten Entwurfsmustern und Idiomen gehören:

Das **Arbeiterobjekt-Erzeugungsmuster** (engl. worker object creation) wird, da es besonders interessant ist, etwas ausführlicher beschrieben. Ausgangslage ist die *Ein-Thread-Regel* von Swing[18], die folgendes besagt: Sobald eine Swing-Komponente sichtbar ist, müssen alle Zugriffe auf den Zustand dieser Komponente im Event-Dispatching-Thread ausgeführt werden. Für die Implementierung dieser Regel bietet sich das Arbeiterobjekt-Erzeugungsmuster an. Dieses leitet – transparent für die Komponenten – sämtliche Aufrufe von Methoden, die auf den Zustand einer Swing-Komponente zugreifen, an den Event-Dispatching-Thread um, anstatt sie direkt auszuführen. Der Kern dieses Entwurfsmusters, ein Advice, ist folgendermassen aufgebaut:

```
void around() : <Pointcut> {
     Runnable worker = new Runnable() {
          public void run() {
               proceed();
          }
     }
     EventQueue.invokeLater (worker);
}
```

Der `Pointcut` identifiziert alle Join Points, die im Event-Dispatching-Thread ausgeführt werden müssen. An jedem Join Point wird anstelle des ursprünglichen Verhaltens ein `worker`-Objekt einer anonymen Klasse erzeugt, die das `Runnable`-Interface implementiert. Das `worker`-Objekt wird anschliessend der `EventQueue` zur Ausführung übergeben. Sobald dort die `run()`-Methode aufgerufen wird, führt `proceed()` das ursprüngliche Verhalten des Join Point aus.

Das **Wurmlochmuster** löst das Problem der Weitergabe von Kontextinformation in einer Aufrufhierarchie, ohne dass Parameter von Objekt zu Objekt weitergegeben müssen. Zu diesem Zweck werden zwei Pointcuts definiert, die sich die gewünschte Kontextinformation (aktuelles Objekt, Zielobjekt oder Argumente) beim obersten bzw. untersten Aufruf der Aufrufhierarchie beschaffen. Ein weiterer `cflow()`-Pointcut identifiziert die untersten Aufrufe im Kontrollfluss des obersten.

Entwurfsmuster für neue Ausnahmen. Wenn ein Advice eine eigene geprüfte Ausnahme werfen möchte, die an den Join Points weder abgefangen noch deklariert wird, führt das zu einem Übersetzungsfehler. Also umhüllt (engl. wrap) der Advice die geprüfte Ausnahme mit einer ungeprüften und wirft diese – im Gegensatz zum Abschwächen von Ausnahmen (siehe im Abschnitt 9.3.8) – mit der geprüften Ausnahme als Ursache (engl. cause), so dass die geprüfte Ausnahme bei Bedarf an den Aufrufer am Join Point zurückgegeben oder in einem zusätzlichen Aspekt wiederhergestellt werden kann.

Teilnehmermuster. Mit den im Abschnitt 9.3.4 beschriebenen Pointcuts können nicht alle gewünschten Join Points identifiziert werden, z.B. Methoden mit bestimmten Eigenschaften. Da die Klassen die Eigenschaften ihrer Methoden am besten kennen, werden solche Pointcuts in die Klassen verlagert. Dazu wird ein abstrakter Aspekt mit abstrakten Pointcuts und konkreten Advice implementiert. Jede Klasse, die am abstrakten Aspekt teilnehmen will, implementiert einen Subaspekt mit den konkreten Pointcuts. Dieses Entwurfsmuster ist untypisch, da der Aspekt die Kernfunktionalität verändert und damit *nicht transparent* ist. Es sollte daher zurückhaltend eingesetzt werden.

[18] www.java.sun.com/products/jfc/ (23.07.2004)

Idiom zur Vermeidung endloser Rekursionen. Join Points innerhalb von Aspekten können endlose Rekursionen verursachen. Zu ihrer Vermeidung werden die betroffenen Pointcuts einfach um `&& !within(<Aspekt>)` ergänzt. Damit werden alle Join Points innerhalb von `Aspekt` ignoriert.

Idiom zum Deaktivieren von Advice. Durch Hinzufügen von `&& if(false)` im entsprechenden Pointcut können Advice logisch gelöscht werden. Da `if(false)` nie als wahr erkannt wird, identifiziert der Pointcut mit Sicherheit nie einen Join Point.

Idiom für leere Pointcut-Definitionen. In einem Subaspekt muss jeder abstrakte Pointcut des Basisaspekts konkret implementiert werden. Ein Pointcut, der für den Subaspekt nicht relevant ist, kann leer implementiert werden, indem der Doppelpunkt sowie die folgende Pointcut-Definition weggelassen werden.

Idiom zur Default-Implementierung von Interfaces. Java erlaubt in Interfaces keine Implementierungen. Mit dem *Introduction*-Mechanismus können Variablen und Methodenrümpfe auch in Interfaces eingefügt werden. Damit kann das Default-Verhalten eines Interface als Aspekt innerhalb des Interface implementiert werden, und die implementierenden Klassen müssen Methoden mit Default-Verhalten nicht unbedingt selber implementieren. Dieses Idiom sollte mit Vorsicht eingesetzt werden, da es die Kopplung von Interfaces und implementierenden Klassen verstärkt.

9.3.11 Grenzen von AspectJ

Auch AspectJ ist kein Allheilmittel und stösst an bestimmte Grenzen. Einige davon werden in diesem Abschnitt aufgezeigt.

Persistenz. Ein Persistenzaspekt ist grundsätzlich nur dann sinnvoll, wenn die Komponenten persistente und transiente Objekte gleich behandeln können. Die Persistenz muss also für die Komponenten transparent sein. Rashid, Chitchyan (2003) zeigen, dass dies nur teilweise der Fall ist:

- Beim **Einfügen** und **Ändern** ist die Persistenz für die Komponenten *transparent*. Sie können alle Objekte gleich behandeln, egal, ob sie persistent oder transient sind.
- Beim **Abfragen** ist die Persistenz für die Komponenten *nicht transparent*. Einerseits sind Abfragen auf persistente Objekte häufig mit komplizierten Bedingungen (z.B. `where`-Klausel einer `select`-Anweisung in SQL[19]) verknüpft, anderseits ist die Granularität der Zugriffe eine wichtige Entwurfsentscheidung.
- Das **Löschen** persistenter Objekte durch einen Aspekt ist gar nicht möglich. Zu diesem Zweck müssten die Komponenten Objekte explizit löschen können. Das ist in Java nicht vorgesehen; das Löschen von Objekten übernimmt der *Garbage Collector* im Hintergrund.

Datenautorisierung. Bei der Datenautorisierung geht es – häufig als Ergänzung zur Funktionsautorisierung – darum, den Zugriff auf bestimmte Daten einzuschränken. Da sich die Pointcuts auf den Programmablauf und nicht auf die Datenstrukturen beziehen, ist es schwierig bis unmöglich, auf einfache Weise Pointcuts für die Datenautorisierung zu definieren. Es muss auf Datentypen, Namenskonventionen oder Konstrukte wie z.B. Enumeration ausgewichen werden (Bodkin, 2004).

9.3.12 Beurteilung

AspectJ ist die meistverbreitete, reifste und mächtigste aspektorientierte Programmiersprache. Kein anderer Ansatz kann mit Konstrukten von der Mächtigkeit der Pointcuts aufwarten. Ein grosser Vorteil von AspectJ ist, dass die Konzepte trotz ihrer Mächtigkeit einfach und gut verständlich sind;

[19] www.ansi.org (22.07.2004)

die Funktionsweise von Pointcuts kann mit UML-Sequenzdiagrammen ausgezeichnet visualisiert werden. Die Separation of Concerns ist erreichbar, aber nicht so offensichtlich wie bei Hyper/J (siehe im Abschnitt 9.6), sobald der Client-Teil eines Crosscutting Concern als Aspekt, der Server-Teil als Komponente implementiert wird. Ein gewisser Nachteil von AspectJ ist die Orientierung am Programmablauf. Join Points sind als „identifizierbare Stellen *in der Ausführung eines Programms"* definiert (Laddad, 2003). Damit sind sämtliche Join Points, die natürlicherweise *nicht* auf Programmabläufen basieren, in AspectJ nicht oder nur auf Umwegen definierbar, z.B. die Feststellung bestimmter Objekteigenschaften oder die Traversierung von Objektstrukturen. Dieser Nachteil wird durch die adaptive Programmierung (siehe im Abschnitt 9.4) teilweise kompensiert. Die Möglichkeit, Aspekte innerhalb von Klassen zu deklarieren, verletzt die Transparenz, wie das Teilnehmermuster im Abschnitt 9.3.10 zeigt.

9.4 Adaptive Programmierung, DemeterJ und die DJ Library

Die Beschreibung der adaptiven Programmierung und ihrer Realisierungen, der Sprache DemeterJ und der DJ (Demeter/Java) Library, stammen von Orleans, Lieberherr (2001) und Lieberherr, Orleans, Ovlinger (2001). Bei DemeterJ handelt es sich um einen *linguistischen*, bei der DJ Library um einen *objektorientierten Ansatz*.

Das **Gesetz von Demeter**[20] (engl. Law of Demeter, kurz LoD) ist die Ausgangslage der adaptiven Programmierung. Es ist eine Stilregel der objektorientierten Programmierung und besagt, dass jede Einheit nur beschränktes Wissen über andere Einheiten haben sollte, und darüber hinaus nur über Einheiten, die mit ihr eng verbunden sind. Damit entspricht es in etwa dem Prinzip der losen Kopplung. Eine Methode soll beispielsweise nur beschränktes Wissen über das Klassenmodell (Assoziationen und Vererbungsbeziehungen) haben. Damit sind Änderungen am Klassenmodell leichter zu bewerkstelligen. Die Kehrseite des Gesetzes von Demeter ist, dass es zu einer grossen Anzahl kleiner Methoden führt, welche die Verständlichkeit erschweren.

Dank der **adaptiven Programmierung** kann anstelle der vielen Methoden, die zur Traversierung einer komplexen Objektstruktur benötigt würden, eine *Traversierungsstrategie* definiert werden. Typische Anwendungsszenarien der adaptiven Programmierung sind iterativ, z.B. die Berechnung einer Summe aus Werten der einzelnen Objekte.

DemeterJ und die DJ Library. Für die adaptive Programmierung stehen zwei Ansätze zur Verfügung, die ältere Sprache DemeterJ und die neuere DJ Library. In *DemeterJ* generiert ein Präprozessor (mit Parser) zur Übersetzungszeit automatisch Java-konforme Methoden. Demgegenüber ist die *DJ Library* ein Java-Paket, das die Fähigkeit hat, Traversierungsstrategien mit Hilfe der Reflexion von Java zur Laufzeit zu interpretieren. Damit können Traversierungsstrategien auf einfache Art zu Java-Programmen hinzugefügt werden, ohne dass ein Präprozessor benötigt wird. Die DJ Library erlaubt auch das Traversieren von Objektstrukturen, deren Quellcode nicht verfügbar ist. Die Klassengraphen, Traversierungsstrategien und zu besuchenden Klassen sind – dank Reflexion – Laufzeit-Parameter. Folglich ist die DJ Library generischer, flexibler und dynamischer als DemeterJ.

Eine **Traversierungsstrategie** muss die Wurzelklasse, die Zielklasse, allfällige Wegpunkte und Einschränkungen definieren. Eine Traversierungsstrategie folgt der Syntax

```
from <Wurzelklasse> [via | bypassing <Bedingung>] to <Zielklasse>
```

und wird in Form von Java-Strings ausgedrückt. Die Traversierungsstrategien werden in DemeterJ und der DJ Library identisch formuliert.

[20] Demeter ist die griechische Göttin des Ackerbaus.

Adaptiver Besucher. Während die Traversierungsstrategie definiert, welche Objekte zu traversieren sind, spezifiziert der *adaptive Besucher* in der DJ Library, was mit jedem traversierten Objekt zu tun ist. Der adaptive Besucher ist eine Subklasse von `Visitor`. Im Unterschied zum Besuchermuster in Gamma et al. (1996) müssen weder eine Methode pro Zielobjekt der Traversierung noch eine `accept`-Methode implementiert werden, sondern nur diejenigen Methoden, die das Verhalten bei den Zielobjekten beschreiben. Ein weiter gehender Vergleich zwischen dem Besuchermuster und dem adaptiven Besucher findet sich im Abschnitt 14.3. Der adaptive Besucher enthält Methoden namens `before` oder `after`, die mit dem traversierten Zielobjekt als Argument aufgerufen werden. In DemeterJ dient anstelle des adaptiven Besuchers eine *Verhaltensdatei* (engl. behaviour file) mit allen Methoden als Eingabe für den Präprozessor.

Ein **Klassengraph** ist ein vereinfachtes UML-Klassendiagramm mit Assoziationen und Vererbungsbeziehungen zwischen den Klassen. In der DJ Library wird ein Objekt der `ClassGraph`-Klasse mit Hilfe von Reflexion aus den Klassen des Default-Pakets erzeugt. Wird dem Konstruktor ein String mit einem Paket-Namen als Argument übergeben, wird das `ClassGraph`-Objekt anhand dieses Pakets erzeugt. Dabei durchsucht der Konstruktor anhand des Paket-Namens den Klassenpfad nach allen `.class`-Dateien in Unterverzeichnissen. Diese werden dann geladen. Mit Hilfe der `addPackage`- und `addClass`-Methoden können später andere Pakete und Klassen zum `ClassGraph`-Objekt hinzugefügt werden. Die `traverse`-Methode ist zweistufig implementiert: Zuerst wird aus dem Klassengraph und der Traversierungsstrategie ein Traversierungsgraph berechnet, dann wird die Objektstruktur traversiert, und die Visitor-Methoden werden aufgerufen. Die `asList`-Methode unterstützt das *generische Programmieren*: Der Traversierungsgraph wird in eine Liste gepackt, in der die Zielobjekte mit Hilfe eines Iterators abgearbeitet werden können. In DemeterJ dient anstelle der `ClassGraph`-Klasse eine so genannte *Class Dictionary*-Datei als Eingabe für den Präprozessor. Der Traversierungsalgorithmus ist in DemeterJ und der DJ Library identisch.

Eine **Adaptive Methode** fasst einen Klassengraph, eine Traversierungsstrategie und einen adaptiven Besucher zusammen. Das folgende mit DJ implementierte Beispiel aus Lieberherr, Orleans, Ovlinger (2001) summiert die Werte aller `Salary`-Objekte, die über Assoziationen von einem `Company`-Objekt aus erreicht werden können:

```
class Company {
     static ClassGraph cg = new ClassGraph();                    (1)
     Double sumSalaries() {
          String s = "from Company to Salary";                   (2)
          Visitor v = new Visitor() {                            (3)
               private double sum;
               public void start () {                            (4)
                    sum = 0.0;
               }
               public void before (Salary host) {                (5)
                    sum += host.getValue();
               }
               public Object getReturnValue() {                  (6)
                    return new Double (sum);
               }
          };
          return (Double) cg.traverse (this, s, v);              (7)
     }
     //Rest der Klasse Company
}
```

Die statische Variable `cg` (1) ist ein Klassengraph-Objekt. Es wird in der `traverse`-Methode (7) benutzt, um die Traversierungsstrategie (2) zu interpretieren und die Traversierung durchzuführen. Der `traverse`-Methode werden drei Parameter übergeben: die Wurzel der zu traversierenden

Objektstruktur (hier das `Company`-Objekt `this`), ein String mit der Traversierungsstrategie (hier `s`) und ein adaptiver Besucher (3) (hier `v`), der weiss, was an den traversierten Punkten (Zielobjekten) zu tun ist. Die `traverse`-Methode beginnt bei der Wurzel, traversiert den spezifizierten Pfad und führt unterwegs alle passenden Methoden des adaptiven Besuchers aus: Beim Start (4) wird die Summe initialisiert, bei jedem `Salary`-Objekt (5) wird sie um den Wert des `Salary`-Objekts erhöht und am Ende (6) wird der Rückgabewert aufbereitet.

Diese Trennung von Traversierungsstrategie (wohin gehen), adaptivem Besucher (was tun) und Klassengraph (in welchem Kontext suchen) erlaubt die Wiederverwendung adaptiver Methoden.

Die **Adaptive Programmierung und AspectJ** sind sich in vielem ähnlich, und die Konzepte von adaptiver Programmierung und AspectJ lassen sich denn auch aufeinander abbilden. Ein Aspekt entspricht konzeptionell einer adaptiven Methode, ein Pointcut einer Traversierungsstrategie und ein Advice einem adaptiven Besucher. Neuerdings wird zudem versucht, Konzepte von AspectJ in die DJ Library zu integrieren. Beispielsweise wird nun zusätzlich zu den `before`- und `after`-Methoden des adaptiven Besuchers auch eine `around`-Methode angeboten, und den Methoden des adaptiven Besuchers soll zusätzlich auch Kontextinformation zur Verfügung gestellt werden.

DAJ (ausgesprochen als „dodge“) von Sung, Lieberherr (2002) ist eine Erweiterung von AspectJ um Konzepte der adaptiven Programmierung, namentlich Klassengraphen, Traversierungsstrategien und adaptive Besucher. Grundlage von DAJ ist eine so genannte *Traversierungsdatei* mit der Endung .trv, die Deklarationen für Klassengraphen und Traversierungen sowie eine Aspektdefinition enthält. Diese Datei wird dann in einem mehrstufigen Prozess durch verschiedene Generatoren und Precompiler bearbeitet, bevor sie schliesslich dem AspectJ-Precompiler als Eingabe übergeben wird. Die folgende AspectJ erweiternde DAJ-Notation zeigt eine dem obigen Beispiel entsprechende Klassengraph- und Traversierungsdeklaration:

```
ClassGraph cg;                                          //Default
declare traversal s(cg) = “from Company to Salary”;
```

Beurteilung. Die adaptive Programmierung ist ein Nischenprodukt und ergänzt AspectJ. Obwohl sich die adaptive Programmierung und AspectJ offenbar konzeptionell aufeinander abbilden lassen, könnte die obige Summierung mit AspectJ nicht so leicht implementiert werden. Die adaptive Programmierung operiert im Gegensatz zu AspectJ auf Objekt- und nicht auf Methodenebene. Um Join Points identifizieren und Advice ausführen zu können, ist in AspectJ immer die Ausführung eines Komponentenprogramms erforderlich. Demgegenüber traversiert eine adaptive Methode den Klassengraphen von sich aus, um die Methoden des adaptiven Besuchers ausführen zu können. Damit ergänzt die adaptive Programmierung AspectJ. Die adaptive Programmierung verfügt aber insgesamt über wesentlich weniger Gestaltungsmöglichkeiten als AspectJ: Die Traversierungsstrategien sind weniger mächtig als die Pointcuts, während die adaptiven Besucher etwas mächtiger sind als die Advice: Alle Join Points in einem Pointcut erhalten den gleichen Advice, während sich der gleiche adaptive Besucher je nach Situation unterschiedlich verhalten kann. Die Dynamik der DJ Library ist ein Vorteil gegenüber AspectJ. Die Tatsache, dass AspectJ mit DAJ um Konzepte der adaptiven Programmierung erweitert wird, untermauert die Dominanz von AspectJ unter den Ansätzen der aspektorientierten Programmierung.

9.5 AspectC++

Neben AspectJ haben Gal, Spinczyk, Schröder-Preikschat (2002) AspectC++ als Pendant zu AspectJ für C++ entwickelt. AspectC++ orientiert sich an der C++-Syntax, ist aber sonst etwa gleich ausdrucksmächtig wie AspectJ. Die Autoren führen AspectC++ anhand von drei interessanten Beispielen vor:

- **Verteilung**: Ein Around Advice ersetzt einen lokalen Server-Aufruf durch einen entfernten (engl. remote) Server-Aufruf. So können die Spezifika der Middleware-Architektur in einem Aspekt gekapselt werden. Ein Nachteil dieser Lösung ist, dass die Verteilung normalerweise nicht auf Klassen/Methodenebene gelöst wird, sondern auf einer höheren Abstraktionsebene. Dafür existieren in AspectC++ keine Sprachkonstrukte (wie z.B. Aspektschablonen oder Reflexion), so dass pro Klasse/Methode ein Aspekt implementiert werden muss.
- **Fehlertoleranz**: Der Verteilungsaspekt wird erweitert, indem so lange entfernte Aufrufe parallel an alle beteiligten Server geschickt werden, bis mindestens eine Antwort eintrifft.
- **Ausführungszeitüberwachung**: Ferner wird mit AspectC++ eine Ausführungszeitüberwachung implementiert. Beim Aufruf einer zu überwachenden Operation beginnt ein Timer zu laufen, und es wird eine Fehlermeldung erzeugt, wenn nach der Operation das vorgegebene Zeitbudget überschritten ist. Damit kann Komponentencode mit oder ohne Ausführungszeitbeschränkung verwendet werden.

9.6 Multi-Dimensional Separation of Concerns (MDSOC) und Hyper/J

Die **Multi-Dimensional Separation of Concerns (MDSOC)** ist ein Modellierungs- und Implementierungsparadigma, das die Separation überlappender Concerns entlang mehrerer Kompositions- und Dekompositionsdimensionen unterstützt (Clarke und Walker, 2001a). Als Dimensionen werden Daten, Funktionen (engl. feature), Geschäftsregeln, Varianten, Systeme etc. betrachtet (Ossher, Tarr, 2001). Konventionelle Entwicklungsmethoden, -sprachen und -werkzeuge unterstützen a) nur eine oder höchstens wenige Dekompositionsdimensionen und b) immer nur eine Dimension gleichzeitig. Wenn ein System jedoch nur entlang einer Dimension (z.B. Daten) zerlegt werden kann, geschieht dies auf Kosten aller anderen Dimensionen (z.B. Funktionen). Dies wird auch als *Tyrannei der dominanten Dekomposition* bezeichnet (Tarr et al., 1999). Die dominante Dekomposition befriedigt zwar wichtige Bedürfnisse, jedoch immer auf Kosten anderer Bedürfnisse. Dadurch entstehen die im Abschnitt 5.5 erwähnten *Scattering*- und *Tangling*-Probleme. Diese löst die MDSOC dadurch, dass sie die gleichzeitige Dekomposition eines Systems entlang mehrerer Dimensionen erlaubt und diese Dimensionen auch wieder zu einem Ganzen vereinigt.

Hyper/J wurde zur Unterstützung der MDSOC für Java-Softwarekomponenten als *linguistischer Ansatz* entwickelt. Hyper/J ist der einzige symmetrische Ansatz der aspektorientierten Programmierung. Sowohl die MDSOC als auch Hyper/J basieren auf der Idee der *Subjektorientierung* (Clarke et al., 1999). Hyper/J ist ein Werkzeug und keine Spracherweiterung von Java. Es operiert auf Binär- und nicht auf Quelldateien, arbeitet also mit den `.class`-Dateien von Java. Dies hat den Vorteil, dass der Quellcode nicht unbedingt verfügbar sein muss (Ossher, Tarr, 2001).

Der Hyper/J-Homepage[21] ist zu entnehmen, dass Hyper/J folgende Möglichkeiten bietet:

- **Flexible Separation of Concerns**. Hyper/J erlaubt die Identifikation, Kapselung und Manipulation von in Java geschriebenen Concerns, und zwar sowohl bei der erstmaligen Entwicklung als auch während der Nutzungsphase, wenn sich neue Anforderungen ergeben.
- **Komposition und Integration**. Aus einer Menge von Modulen, die verschiedene Concerns implementieren, können bestimmte Concerns herausgelöst und zu Komponenten oder Systemen zusammengefügt werden (engl. mix-and-match). Dies erlaubt, Erweiterungen und massgeschneiderte Konfigurationen bestehender Systeme zu erzeugen, ohne die Software ändern zu müssen, und fördert damit die Wiederverwendung und (Rück-)Verfolgbarkeit.

[21] www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm (22.07.2004)

- **Evolution**. Welche Auswirkungen eine Änderung hat, hängt von der Art der Modularisierung ab. Beispielsweise hat die Änderung einer Datenstruktur in einem objektorientierten System geringe Auswirkungen, während die Änderung einer Funktion bedeutende Auswirkungen haben kann. Hyper/J ermöglicht, die Art von Modularisierung zu verwenden, auf die eine gegebene Änderung die geringsten Auswirkungen hat.
- **Nicht-invasive Remodularisierung**. Anforderungen an neue Concerns können die Einführung einer neuen Art von Modularisierung erfordern. Hyper/J erlaubt eine Remodularisierung auf Wunsch (engl. on-demand), also die Identifikation, Kapselung und Modularisierung von Concerns basierend auf bestehenden Concerns.

Hyper/J besteht aus *Hyperspaces*, *Hyperslices* und *Hypermodulen*, wie die Abbildung 9-9 zeigt. Ein Hyperspace wird mittels *Concern Mapping* (siehe weiter unten in diesem Abschnitt) in Hyperslices aufgeteilt. Dies entspricht der Separation. Die Hyperslices werden anschliessend mit Hilfe von *Kompositionsregeln* zu Hypermodulen kombiniert. Dies entspricht der Integration.

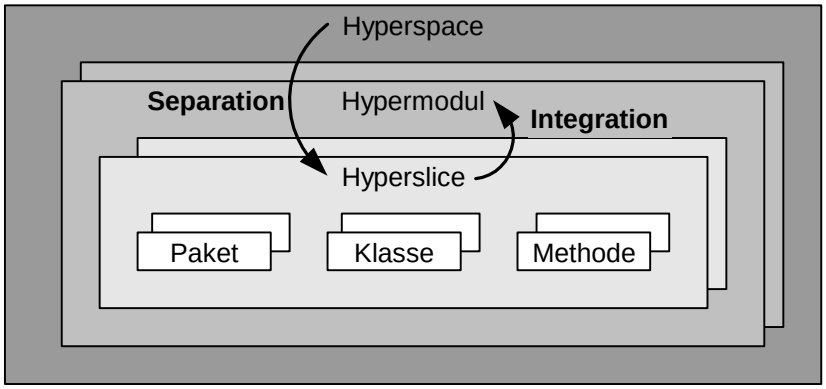


Abbildung 9-9: Hyperspaces, Hyperslices und Hypermodule

Ein **Hyperspace** umfasst den gesamten Concern-Raum mit allen Dimensionen. In Hyper/J wird er durch eine Menge von Java `.class`-Dateien beschrieben, die den Systemumfang festlegen (Clarke, Walker, 2001a).

Ein **Hyperslice** ist eine Menge konventioneller Module, die einen bestimmten Concern in einer Dimension kapseln, die nicht der dominanten Dimension entspricht. Er enthält genau die Module (üblicherweise als „Einheiten" bezeichnet), die diesen Concern betreffen. (In der Subjektorientierung entspräche ein Hyperslice nach Elrad et al. (2001) einem Subjekt.) Die Einheiten werden dabei als unteilbar betrachtet. Die Granularität der Einheiten kann theoretisch irgendwo zwischen Paketen und Anweisungen liegen, sollte aber nicht so fein sein, dass üblicherweise verborgene Einheiten (z.B. die Implementierung einer Methode) zum Vorschein kommen. Hyper/J definiert daher Methoden als feinste Einheiten. Da eine Einheit in mehreren Hyperslices vorkommen kann, können Hyperslices überlappen. Ein System wird als Menge von Hyperslices verstanden, die alle wichtigen Concerns in so viele Dimensionen aufgeteilt wie nötig (Tarr et al., 1999).

Deklarative Vollständigkeit. Eine Folge des Hyper/J-Ansatzes ist, dass Hyperslices Methoden referenzieren können, die sie nicht selber implementieren. Dies ist nicht Java-konform. Daher müssen alle Methoden, die in einem Hyperslice referenziert, aber nicht implementiert werden, als abstrakt deklariert werden. Damit wird *deklarative Vollständigkeit* erreicht: Jeder Hyperslice ist ein korrektes, in sich geschlossenes, nicht aber notwendigerweise vollständiges (und für sich allein lauffähiges oder direkt verwendbares) Java-System. Dennoch sind die Hyperslices nur lose gekoppelt, da sie sich nicht gegenseitig referenzieren (Ossher, Tarr, 2001).

Das **Concern Mapping** ermöglicht die Separation und beschreibt, welche Java-Einheiten aus dem Hyperspace zu welchen Dimensionen und Hyperslices gehören. Das folgende Beispiel aus einer Personalanwendung zeigt ein Concern Mapping (Ossher, Tarr, 2001):

```
package Personnel: Feature.Personnel                    (1)
operation position: Feature.Payroll                     (2a)
operation pay: Feature.Payroll                          (2b)
```

Die generelle Regel (1) besagt, dass alle Methoden, Klassen und Interfaces des `Personnel`-Pakets zum `Personnel`-Hyperslice der `Feature`-Dimension gehören. Die folgenden beiden spezifischen Regeln (2a und 2b) beschreiben eine Ausnahme von der Regel (1) für alle Operationen namens `position` oder `pay`: Sie gehören nicht zum `Personnel`-Hyperslice, sondern zum `Payroll`-Hyperslice. Die in diesem Beispiel dargestellte Ergänzung einer generellen Regel (1) um spezifische Regeln (2a und 2b), die Spezial- und Ausnahmefälle behandeln, ist typisch für Hyper/J.

Ein **Hypermodul** beschreibt die Integration und kombiniert eine Menge von Hyperslices anhand von *Kompositionsregeln* zu kompletten und konsistenten (Sub-)Systemen. Die Kompositionsregeln spezifizieren, wie die Hyperslices kombiniert werden müssen. Hypermodule sind selber auch wieder Hyperslices, können also verschachtelt sein (Tarr et al., 1999). Das folgende Beispiel zeigt ein Hypermodul aus der obigen Personalanwendung (Ossher, Tarr, 2001):

```
hypermodule PayrollPlusPersonnel
     hyperslices: Payroll, Personnel;
     relationships: mergeByName;
end hypermodule
```

Das Hypermodul `PayrollPlusPersonnel` umfasst die Hyperslices `Payroll` und `Personnel`. Die Kompositionsregel `mergeByName` bedeutet, dass alle Einheiten mit gleichem Namen und gleichem Typ aus verschiedenen Hyperslices zu neuen Einheiten verschmolzen werden. Die Klassen `Payroll.Employee` und `Personnel.Employee` würden beispielsweise zu einer neuen Klasse `PayrollPlusPersonnel.Employee` verschmolzen. Mit der Anweisung `order` kann ausserdem die Reihenfolge der Methodenrümpfe in einer verschmolzenen Methode festgelegt werden.

Weitere Kompositionsregeln erlauben es, auch Einheiten mit ungleichen Namen zu verschmelzen (`nonCorrespondingMerge` in Kombination mit `equate`), Einheiten zu überschreiben (`override`), zu erweitern (`bracket` in Kombination mit `before` oder `after`) oder die relevante Einheit erst zur Laufzeit anhand einer Bedingung zu bestimmen. Kompliziertere Kompositionsregeln können mittels *Composition Patterns* (siehe im Abschnitt 10.2) modelliert werden (Clarke et al., 1999, und Clarke, Walker, 2001a).

Hyper/J und AspectJ. Hyper/J ist ein symmetrischer Ansatz und macht daher – im Gegensatz zu AspectJ – keinen Unterschied zwischen Komponenten und Aspekten, sondern geht von gleichwertigen Hyperslices (= Concerns) aus. Konsequenterweise gibt es in Hyper/J kein Advice-ähnliches Konstrukt, sondern lediglich Methoden. Die Methoden in den Hyperslices von Hyper/J entsprechen dabei etwa den Advice von AspectJ, die Kompositionsregeln in den Hypermodulen den Pointcuts. Hyper/J ist in der Lage, beliebige unabhängig voneinander implementierte Concerns zusammenzuführen, während die Aspekte von AspectJ die Komponenten, welche die Core Concerns implementieren, um „crosscutting“ Verhalten erweitern.

Beurteilung. Der obige Vergleich zwischen Hyper/J und AspectJ ergibt, dass sich Hyper/J eher für die Integration vollständiger Subsysteme (Subjekte) eignet, während sich AspectJ eher für die Erweiterung bestehender Komponenten um relativ einfaches „crosscutting“ Verhalten eignet. Ausserdem ist Hyper/J abstrakt und nicht so einfach zu verstehen wie AspectJ. Die Kompositionsregeln von Hyper/J bieten insgesamt weniger Möglichkeiten als die Pointcuts von AspectJ. Es gibt,

mit Ausnahme der `mergeByName`-Kompositionsregel, in Hyper/J kein Konstrukt von der Mächtigkeit der Pointcuts von AspectJ, so dass die Join Points tendenziell einzeln definiert werden müssen. Aus diesem Grund ist Hyper/J in noch viel stärkerem Masse als AspectJ von der Einhaltung von Namenskonventionen abhängig. Ausserdem ist die Anwendbarkeit von Hyper/J für grosse bzw. heterogene Systeme beeinträchtigt. Dafür stimmt in Hyper/J die dynamische Prozessstruktur besser mit der statischen Programmstruktur (siehe Dijkstra, 1968) überein als in AspectJ.

Der immer wieder betonte Vorteil von Hyper/J, dass der Quellcode nicht verfügbar sein muss, hat bei genauerem Hinsehen auch einen beträchtlichen Mangel: Wie will man zu einem bestimmten Concern gehörende Methoden identifizieren und sinnvolle Kompositionsregeln definieren, wenn der Quellcode der Klasse nicht bekannt ist? Mit einer Komposition nur auf Klassenebene können viele „crosscutting" Probleme nicht gelöst werden.

Die deklarative Vollständigkeit hat den Vorteil der Java-Konformität, jedoch den Nachteil, dass sie die betroffenen Hyperslices stärker koppelt. Hyperslices, die nur deklarativ vollständig sind, können niemals für sich allein verwendet werden, sondern sind abhängig von der Existenz eines Hyperslice, der die entsprechenden abstrakten Methoden implementiert. Insgesamt ist die Separation of Concerns aber bei Hyper/J ausgezeichnet.

Die eingangs erwähnten Aussagen auf der Hyper/J-Homepage versprechen teilweise mehr, als sie halten können:

- **Identifikation**. Die Aussage, wonach Hyper/J die Identifikation von Concerns erlaubt, ist zu relativieren. Der Prozess der Identifikation von Concerns wird durch Hyper/J in keiner Weise methodisch unterstützt. Hyper/J erlaubt lediglich, bereits identifizierte Concerns mit Hilfe des Concern Mapping zu spezifizieren.
- **Remodularisierung**. Hyper/J behandelt Methoden als atomare Einheiten. Methoden können folglich beim Concern Mapping nicht aufgeteilt werden. Dies hat zur Folge, dass es nicht möglich ist, eine Menge von Anweisungen, die zu einem bestimmten Concern gehören, aus einer „tangled" Methode herauszulösen, die mehrere (Fragmente von) Concerns implementiert. Mit Hyper/J kann man also Klassen und Methoden zu neuen (Sub-) Systemen kombinieren, methodeninternes Tangling kann man aber nicht beheben. Dies widerspricht bis zu einem gewissen Grad der Behauptung der nicht-invasiven Remodularisierung aus der Hyper/J-Homepage. Den Autoren bleibt denn auch nichts anderes übrig als zu empfehlen, die Methoden in solchen Fällen – invasiv – aufzuteilen (engl. split).

9.7 Composition Filters (CF)

Die Composition Filters sind ein linguistischer Ansatz und werden in Bergmans, Aksit (2001a) und Bergmans, Aksit (2001b) beschrieben. Sie erlauben, Crosscutting Concerns als Erweiterungen von Objekten zu implementieren. Die Art und Weise, wie Composition Filters Objekte erweitern, ist

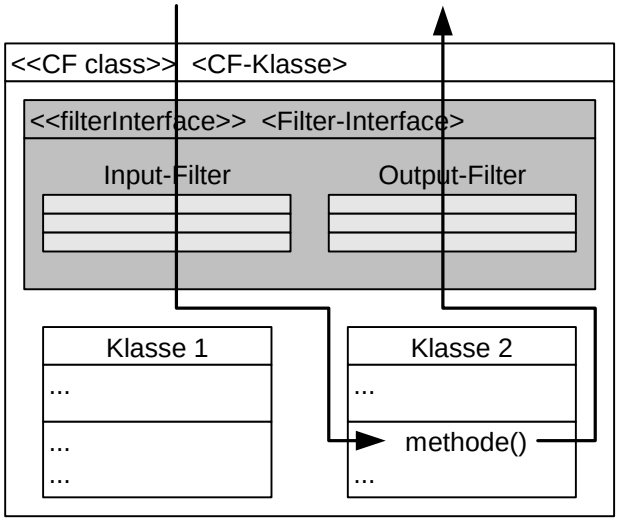
- **modular**. Composition Filters sind *unabhängig* von der Implementierung(ssprache). Filter können an Objekte in verschiedenen Sprachen angehängt werden, ohne sie zu verändern.
- **orthogonal**. Composition Filters sind *zusammensetzbar*. Die Semantik eines Filters ist unabhängig von der Semantik anderer Filter.

Diese beiden Eigenschaften erleichtern die Wiederverwendung und Pflege.

Die **Filter** sind Funktionen, die an Objekte eingehende (*Input-Filter*) und von Objekten ausgehende (*Output-Filter*) Nachrichten bearbeiten. Da das beobachtbare Verhalten eines Objekts durch die Nachrichten bestimmt wird, die es sendet und empfängt, können Filter eine breite Palette von Concerns wie Vererbung, Delegation, Synchronisation, Echtzeit-Einschränkungen (z.B. Zugriffskontrolle) und Interobjekt-Protokolle ausdrücken. Filter können eine Nachricht akzeptieren (engl.

accept) oder zurückweisen (engl. reject). Sind mehrere Filter erforderlich, werden sie hintereinander in Serie geschaltet.

Eine so genannte **CF-Klasse** fasst 0 – n Klassen (z.B. eine Klasse plus ihre Superklasse) zusammen und kombiniert ihr Verhalten unter Verwendung von 1 – n Filtern. CF-Klassen können auch verschachtelt sein. CF-Klassen werden in der UML mit dem Stereotyp `<<CF class>>` dargestellt. Die Abbildung 9-10 illustriert eine CF-Klasse.



Quelle: Bergmans, Aksit (2001a)

Abbildung 9-10: CF-Klasse

Jede ein- bzw. ausgehende Nachricht muss zuerst die Input- bzw. Output-Filter der CF-Klasse passieren. Input- und Output-Filter bilden zusammen das Filter-Interface. Filter-Interfaces werden in der UML mit dem Stereotyp `<<filterInterface>>` dargestellt. Welche Auswirkungen diese Filter auf die Nachrichten haben, hängt von den angewendeten Filtertypen ab.

Es gibt verschiedene **Filtertypen** für Input-Filter:

- **Dispatch** leitet eine Nachricht zwecks Ausführung an ihr Zielobjekt weiter, falls sie akzeptiert wird, andernfalls wird sie an den nächsten Filter weitergereicht. Dispatch kann beispielsweise eine Vererbungsbeziehung simulieren. Dies würde wie folgt ausgedrückt:

  ```
  Dispatch = {<Subklasse>.*, <Superklasse>.*}
  ```

  Dieser Filter sagt aus, dass bei jeder eingehenden Nachricht als erstes die `<Subklasse>` nach einer passenden Methode durchsucht wird, an welche die Nachricht zur Ausführung weitergeleitet werden kann. Die Wildcard (*) steht für „eine beliebige Methode“. Falls in der `<Subklasse>` keine passende Methode existiert, wird als nächstes die `<Superklasse>` durchsucht.

- **Error** leitet eine Nachricht an den nächsten Filter weiter, falls sie akzeptiert wird, andernfalls wird eine Fehlermeldung erzeugt. Damit kann ein Zugriffsschutzmechanismus implementiert werden, beispielsweise, welche Benutzerklasse welche Methoden aufrufen dürfen:

  ```
  Error = {<bedingung1> => {<nachricht1>, <nachricht2>},
       <bedingung2> => {<nachricht3>, <nachricht4>},
       True ~> {nachricht1, nachricht2, nachricht3, nachricht4} }
  ```

  Falls die `<bedingung1>` (z.B. eine bestimmte Benutzerklasse) zutrifft, werden nur die eingehenden Nachrichten `<nachricht1>` oder `<nachricht2>` akzeptiert. Falls die `<bedingung2>` zutrifft, werden nur die eingehenden Nachrichten `<nachricht3>` oder

`<nachricht4>` akzeptiert. Alle anderen Nachrichten werden immer (`True` trifft mit Sicherheit zu!) akzeptiert. Der Operator ~> steht für „mit Ausnahme von“.

- **Wait** leitet – wie Error – eine Nachricht an den nächsten Filter weiter, falls sie akzeptiert wird. Andernfalls wird die Nachricht so lange in eine Warteschlange gestellt, bis eine bestimmte Bedingung erfüllt ist, so dass sie nicht mehr zurückgewiesen wird. Damit kann eine Synchronisation implementiert werden, zum Beispiel:

  ```
  Wait = {NoActiveThreads => *}
  ```

  Solange die Bedingung (`NoActiveThreads`) zutrifft, werden alle Nachrichten akzeptiert. Sonst werden sie zurückgewiesen und in eine Warteschlange gestellt, bis keine Threads mehr aktiv sind.
- **Meta** leitet eine Nachricht `<nachricht1>` als Parameter einer anderen (Meta-)Nachricht `<nachricht2>` an ein bestimmtes Objekt `<objekt>` weiter, falls sie akzeptiert wird. Andernfalls wird sie an den nächsten Filter weitergereicht:

  ```
  Meta = {[<nachricht1>] <objekt>.<nachricht2> }
  ```

  Das Objekt, das die Nachricht erhält, kann die Nachricht an ihr ursprüngliches Zielobjekt weiterleiten. Damit kann beispielsweise Logging implementiert werden.
- **Realtime** erlaubt, der Ausführung von Methoden zeitliche Einschränkungen aufzuerlegen.

Operatoren. Die Bedeutung der in den Filterausdrücken bereits teilweise verwendeten Operatoren ist in der Tabelle 9-7 ersichtlich.

Operator	Bedeutung
;	und
,	oder
~>	Ausschluss (engl. exclusion)
=>	Freigabe (engl. enable)

Tabelle 9-7: Bedeutung der Operatoren in den Composition Filters

Überlagerung (engl. superimposition). Mittels Überlagerung können Filter-Interfaces 0 – n Objekte erweitern. Damit wird das eigentliche „crosscutting“ Verhalten über mehrere Komponenten implementiert. Ein Beispiel für eine Überlagerung:

```
selectors
     allTasks = {*=<Klasse1>, *=<Klasse2>, …)
filterInterfaces
     allTasks <- <filter-Interface>;
```

Damit werden sämtliche Objekte der `<Klasse1>` und der `<Klasse2>` um das an einer anderen Stelle definierte `<filter-Interface>` erweitert. Ein Selektor ist einem Pointcut aus AspectJ ähnlich und spezifiziert die Stellen der Kernfunktionalität, an denen das „crosscutting“ Verhalten zum Tragen kommen soll.

Eigenschaften. Die Composition Filters zeichnen sich durch folgende Eigenschaften aus:

- **Deklarativ**. Die Concerns können mit Hilfe einer einfachen Sprache deklarativ formuliert werden. Für die Composition Filters gibt es verschiedene Implementierungen: ComposeJ[22]

[22] http://trese.cs.utwente.nl/prototypes/composeJ/ (24.07.2004)

integriert die Filter mittels *Inlining* in die passenden Methoden, Sina[23] implementiert die Filter als Metaobjekte.

- **Filter-Semantik**. Die Filtertypen kapseln die Semantik des Akzeptierens und Zurückweisens von Nachrichten.
- **Offen**. Das Composition Filter-Modell kann problemlos um neue Filtertypen und Operatoren erweitert werden.
- **Gekapselt**. Der Wirkungsbereich der Composition Filters ist auf die Interface-Ebene beschränkt. Dadurch werden Implementierungsdetails ausgeblendet, und die Kapselung der Objekte bleibt erhalten.
- **Modular**. Das Composition Filter-Modell verknüpft die objektorientierte mit der aspektorientierten Programmierung. Crosscutting Concerns können als modulare Erweiterungen von Objekten implementiert werden.
- **Zusammensetzbar**. Filter können beliebig zusammengesetzt, d.h. hintereinander geschaltet, werden.

Beurteilung. Die Composition Filters sind einfach zu verstehen und zu implementieren, aber aufgrund der Beschränktheit der meisten ihrer Filtertypen weniger universell einsetzbar als AspectJ. Lediglich der Meta-Filtertyp erlaubt ähnliche Implementierungen wie z.B. die Advice von AspectJ. Der Überlagerungs-Mechanismus entspricht den Pointcuts von AspectJ, beschränkt sich jedoch auf die Pointcuts von Methodenaufrufen. Analogien zu den Introductions und Declarations von AspectJ fehlen ganz. Die Separation of Concerns ist gewährleistet, aber nicht sehr offensichtlich.

9.8 Frameworks

Constantinides et al. (2000) beschreiben das Aspect Moderator Framework (AMF), ein Beispiel für einen Framework-basierten *objektorientierten Ansatz*, der sich primär auf das *Concurrent Object-Oriented Programming*, also die Nebenläufigkeit von Objekten, konzentriert. Das Weaving erfolgt zur Übersetzungszeit. Das Framework setzt das *Vertragsprinzip* (siehe auch im Abschnitt 14.4) um: Die Interaktion zwischen Aspekten und Komponenten wird über Verträge spezifiziert. Die Grundzüge des Framework sind in der Abbildung 9-11 grafisch dargestellt (zur Vereinfachung ohne Interfaces).

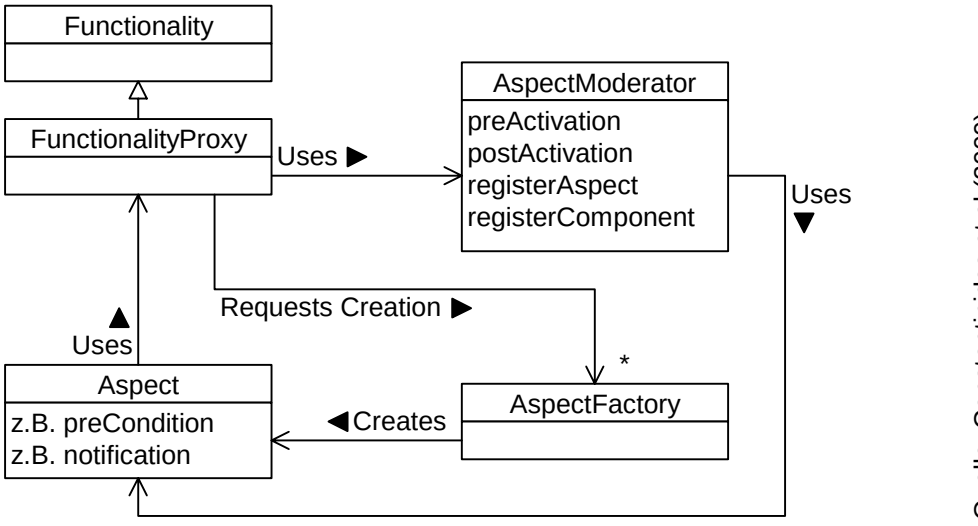


Abbildung 9-11: Das Aspect Moderator Framework im Überblick

[23] http://trese.cs.utwente.nl/sina/ (24.07.2004)

Die **Functionality** repräsentiert die Komponenten. Nach Netinant, Elrad, Fayad (2001) ist es wichtig, dass ihre Methoden, die in diesem Ansatz als *funktionale Methoden* bezeichnet werden, völlig von aspektbezogenen Eigenschaften befreit sind.

Das **Functionality Proxy** hat drei Aufgaben: Erstens lässt es die Aspekte bei der `AspectFactory` als Subklassen von `Aspect` erzeugen, zweitens lässt es die Aspekt- und Komponentenobjekte beim `AspectModerator` registrieren und drittens fängt es die funktionalen Methodenaufrufe an die Komponenten ab und ruft vor jeder Methodenausführung die `preActivation`-Methode (zur Sicherstellung der Voraussetzungen) und nach jeder Methodenausführung die `postActivation`-Methode (zur Sicherstellung der Ergebniszusicherungen) des `AspectModerator` auf.

Der **Aspect Moderator** hat die Aufgabe, Aspekte und Komponenten zu koordinieren. Zu diesem Zweck implementiert er die folgenden Methoden:

- `registerAspect` zur Registrierung eines Aspekts
- `registerComponent` zur Registrierung einer Komponente
- `preActivation` zur Sicherstellung der Voraussetzungen (z.B. `preCondition`-Methode) der relevanten Aspekte vor jedem funktionalen Methodenaufruf
- `postActivation` zur Sicherstellung der Ergebniszusicherungen (z.B. `notification`-Methode) der relevanten Aspekte nach jedem funktionalen Methodenaufruf

Daneben ist der `AspectModerator` verantwortlich dafür, dass die Aspekte in der richtigen Reihenfolge ausgeführt werden, z.B. die Authentisierung vor der Synchronisation.

Layered Aspect Moderator Framework (L-AMF). Netinant, Elrad, Fayad (2001) beschreiben zusätzlich das Layered Aspect Moderator Framework (L-AMF). Es ist primär für die Implementierung von objektorientierten Betriebssystemen gedacht und erweitert das Aspect Moderator Framework um Dienstleistungsschichten. Wie beim Aspect Moderator Framework besteht eine *Schicht* aus Komponenten und aspektbezogenen Eigenschaften, die vom Aspect Moderator gesteuert werden. Die Dienstleistungen innerhalb der Schichten (engl. intralayer) werden von ihren aspektbezogenen Eigenschaften (z.B. Sicherheit, Performance und Zeitplanung (engl. scheduling)) getrennt. So kann eine aspektbezogene Eigenschaft mit verschiedenen Strategien (engl. policy) implementiert werden. Ein *System* besteht aus Schichten und Schnittstellen zwischen den Schichten (engl. interlayer), die ebenfalls vom Aspect Moderator gesteuert werden. Um die Abstraktionen zwischen den Schichten zu definieren und das dynamische Binden der Aspekte zu unterstützen, werden *Entwurfsmuster* (Abstrakte Fabrik, Adapter und Brücke) aus Gamma et al. (1996) eingesetzt. Im Unterschied zum Aspect Moderator Framework erfolgt das *Weaving* zur Laufzeit.

Beurteilung. Das Aspect Moderator Framework erlaubt die Umsetzung des Vertragsprinzips, d.h. es benutzt Aspekte, um Voraussetzungen und Ergebniszusicherungen von Methodenaufrufen sicherzustellen.

Das Aspect Moderator Framework ist in seiner Wirkungsweise ähnlich beschränkt wie die Composition Filters: Das „crosscutting“ Verhalten kann lediglich unmittelbar vor und nach einem Methodenaufruf eingewoben werden. (Im Gegensatz dazu bieten die Pointcuts von AspectJ weiter gehende Möglichkeiten.) Während diese Beschränkung bei den Composition Filters, einem linguistischen Ansatz, nicht von heute auf morgen beseitigt werden kann, ist dies beim Aspect Moderator Framework, einem objektorientierten Ansatz, jederzeit und anwendungsspezifisch möglich.

Die im Abschnitt 7.4 beschriebene Eigenschaft, dass Aspekte transparent und aktiv sein sollen, ist im Aspect Moderator Framework nur teilweise erfüllt: Zwar müssen die Komponenten nichts über die einzelnen Aspekte wissen, aber den Aspektmoderator müssen sie kennen und aktiv involvieren. (In den meisten anderen Ansätzen sind die Komponenten passiv und die Aspekte aktiv). Dadurch löst das Aspect Moderator Framework das *Tangling*-Problem nicht.

Das Aspect Moderator Framework ist nicht ganz einfach zu verstehen, jedoch leicht änder- und erweiterbar. Damit besteht – wie bei allen Frameworks – die Gefahr, dass seine Implementierungen bei unsachgemässer Anwendung unübersichtlich und unlesbar werden. Auch die *Separation of Concerns* stellt sich nicht von selbst ein, da sich Komponenten und Aspekte nicht auf Anhieb klar auseinander halten lassen.

9.9 Reflexive Ansätze

Völter (2001) beschreibt Anwendungsmöglichkeiten von Metaobjektprotokollen für die aspektorientierte Programmierung. Metaobjektprotokolle definieren keine aspektspezifischen neuen Sprachkonstrukte und gehören daher zu den objektorientierten Ansätzen. Die Möglichkeiten, welche die diversen objektorientierten Sprachen hinsichtlich Metaobjektprotokollen bieten, sind unterschiedlich. Daher eignen sich nicht alle objektorientierten Sprachen gleichermassen für die reflexive Implementierung von Aspekten.

Metaobjektprotokolle sind Schnittstellen für den Zugriff auf Metaobjekte. Damit kann nach Sullivan (2001) das Default-Verhalten einer Programmiersprache geändert werden. Es werden drei Arten von Zugriffen unterschieden:

1. **Introspektion**. Ein Programm hat *lesenden Zugriff* auf seine eigene Struktur. Es kann sich Typinformation, Information über Klassen, deren Attribute und Operationen, sowie Information über die Vererbungshierarchie beschaffen. Beispielsweise kann ein Java-Programm mit `getClass()` abfragen, zu welcher Klasse ein bestimmtes Objekt gehört, es kann die Methoden dieser Klasse herausfinden und eine dieser Methoden aufrufen.
2. **Reflexion**. Ein Programm hat *schreibenden Zugriff* auf seine eigene Struktur. Es kann Klassendefinitionen durch Hinzufügen, Löschen oder Überschreiben von Operationen ändern, und es kann die Klassenhierarchie durch Einfügen von Basisklassen ändern.
3. **Reifikation**. Ein Programm kann durch Änderung der Semantik von Instanzierungen, Methodenaufrufen und Attributzugriffen sein eigenes Verhalten ändern, ggf. sogar zur Laufzeit.

Aspekte in Metaobjektprotokollen. Von Metaobjektprotokollen kann man Gebrauch machen, um „crosscutting" Verhalten zu implementieren. Das folgende Beispiel soll zeigen, wie man sich einen mittels Metaobjektprotokoll realisierten Logging-Aspekt vorzustellen hat. In der Abbildung 9-12 ist das entsprechende UML-Klassendiagramm abgebildet.

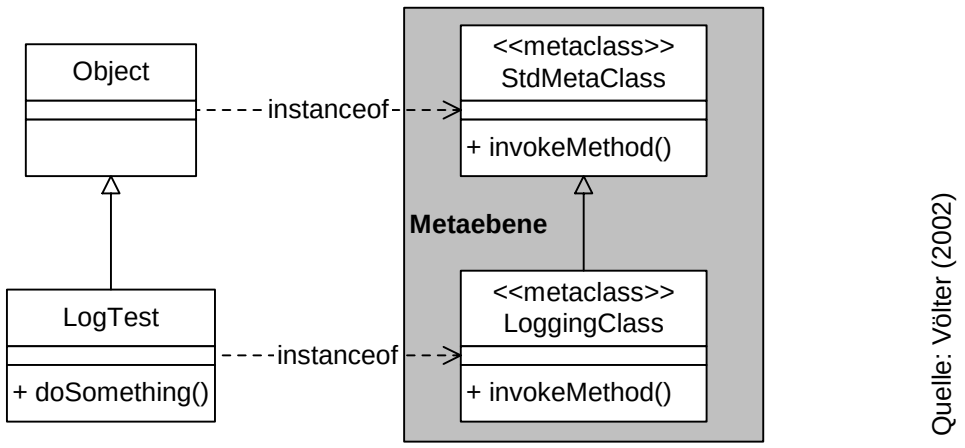


Abbildung 9-12: UML-Klassendiagramm für die Anwendung eines Metaobjektprotokolls

Die `LogTest`-Klasse stellt die Komponente dar, deren `doSomething`-Methode um das Logging-Verhalten erweitert werden soll. Da Java nicht über die erforderlichen Konstrukte verfügt, wird zur Darstellung der Implementierung die Sprache *MetaJava*[24] verwendet, ein nach Angaben von Völter (2001) hypothetisches Java mit einem Metaobjektprotokoll. Die `LogTest`-Klasse und ihre `LoggingClass`-Metaklasse werden mit MetaJava folgendermassen programmiert:

```
public class LogTest extends Object metaclass LoggingClass
     public void doSomething() {
          //do something
     }
}

public class LoggingClass extends StdMetaclass {
     public void invokeMethod(Object dest, String name, Object[] params) {
          System.out.println(name+" called on "+dest+" with "+params);
          super.invokeMethod(dest, name, params);
     }
}
```

Die Interaktionen sind in der Abbildung 9-13 anhand eines UML-Sequenzdiagramms dargestellt.

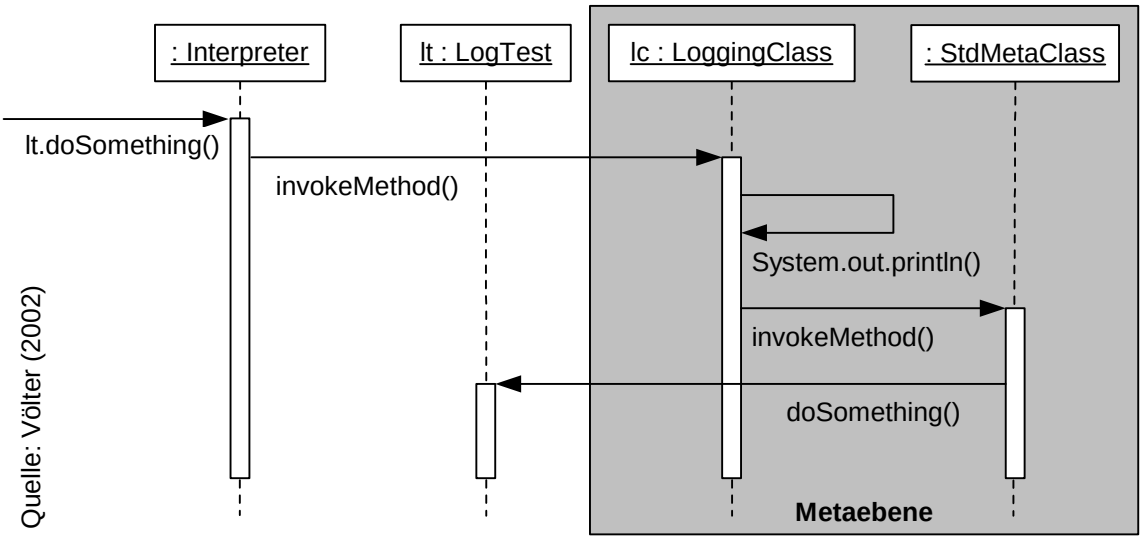


Abbildung 9-13: UML-Sequenzdiagramm für die Anwendung eines Metaobjektprotokolls

Die `doSomething`-Methode eines `LogTest`-Objekts wird aufgerufen. Anstatt sie beim `LogTest`-Objekt auszuführen zu lassen, ruft der Interpreter die `invokeMethod` ihres `LoggingClass`-Metaobjekts auf, und zwar mit der ursprünglichen Nachricht als Argument. (Das entspricht genau dem Verhalten des *Meta*-Filtertyps der Composition Filters aus dem Abschnitt 9.7). Die `invokeMethod` des `LoggingClass`-Metaobjekts führt zuerst das Logging durch (hier mit `System.out.println`) und ruft anschliessend die `invokeMethod` (wieder mit der ursprünglichen Nachricht als Argument) ihrer Superklasse `StdMetaClass` auf. Diese lässt schliesslich die `doSomething`-Methode im `LogTest`-Objekt ausführen.

Metaobjektprotokolle und AspectJ. Das obige Beispiel wäre in AspectJ mit einem Before Advice implementierbar. Es lässt sich leicht vorstellen, dass Implementierungen von After und Around Advice ganz ähnlich aussehen würden.

[24] Das hier verwendete MetaJava darf nicht mit dem erweiterten Java-Interpreter namens MetaJava verwechselt werden, der 1997 an der Universität Erlangen entwickelt wurde (www4.informatik.uni-erlangen.de/metajava (22.07.2004)).

Vorteile von Metaobjektprotokollen. Es gibt keine statische Übersetzungsphase. Der Aspektcode kann Entscheidungen basierend auf Werten zur Laufzeit treffen. Das aspektorientierte Verhalten kann zur Laufzeit hinzugefügt/entfernt werden. Nach Diaz Pace, Campo (2001) ist ausserdem die Separation of Concerns gewährleistet, da sich die Aspekte auf der Metaebene und die Komponenten auf der Basisebene befinden.

Beurteilung. Das Beispiel zeigt, dass Metaobjektprotokolle nicht zu transparenten Aspekten führen, d.h. die Komponenten-Klasse muss – invasiv – leicht angepasst werden. Dies könnte aber gewiss automatisiert werden.

Anhand des Schlüsselworts `metaclass` in einer Klassendeklaration fängt der Interpreter *sämtliche* Nachrichten an Objekte dieser Klasse ab und leitet sie an das entsprechende Metaobjekt um. Dies hat den Nachteil, dass erst auf der Metaebene anhand der Nachricht entschieden wird, ob die zugehörige Methode überhaupt vom „crosscutting" Verhalten betroffen ist. Dies erfordert Fallunterscheidungen in der Metaklasse und führt zu unnötigem Overhead.

Metaobjektprotokolle sind mächtig und nicht einfach zu verstehen. Da damit viel Unfug getrieben werden kann, sollten sie nur von wirklich guten und erfahrenen Programmierern verwendet werden. Das obige Beispiel zeigt nur eine von vielen Möglichkeiten, wie man Metaobjektprotokolle für die Implementierung von Aspekten nutzen kann. Angesichts dieser Vielfalt ergeben sich automatisch Probleme bei der Wiederverwendung von Aspekten. Ein – standardisiertes – Framework zur Implementierung von Aspekten mit Metaobjektprotokollen könnte hier Abhilfe schaffen.

9.10 Empirische Untersuchungen zur aspektorientierten Programmierung

9.10.1 Vergleich eines linguistischen, objekt- und architekturorientierten Ansatzes

Diaz Pace, Campo (2001) wollten anhand eines Experiments herausfinden, wie weit Techniken der aspektorientierten Programmierung gute Entwürfe fördern. Neben den erwähnten *linguistischen* und *objektorientierten* Ansätzen bezogen Diaz Pace, Campo (2001) auch so genannte *architekturorientierte* Ansätze in den Vergleich mit ein. Bei diesen werden die Crosscutting Concerns bereits in einer früheren Phase des Entwicklungsprozesses, also vor der Entwurfs- und Implementierungsphase, auf einer höheren Abstraktionsebene identifiziert und beschrieben und später mittels verschiedener Implementierungstechniken (z.B. Frameworks) auf Aspekte abgebildet.

Experiment. Ein System zur Simulation einer Raumtemperatursteuerung mittels eines mathematischen Modells diente als Vorgabe für die Implementierung. Als Aspekte, die das System quer schneiden, wurden das mathematische Modell, die Nebenläufigkeit/Synchronisation und die Zeitplanung (engl. scheduling) bestimmt. Vier Gruppen von Entwicklern implementierten dieses System mit AspectJ, einem reflexiven Framework namens TaxonomyAop, einem Architektur-Framework namens Bubble und ausserdem konventionell mit Java. Die vier entstandenen Systeme wurden anschliessend anhand von verschiedenen Kriterien verglichen.

Ergebnisse. Die interessantesten Ergebnisse aus diesem Experiment waren:

- **Performance**. Die CPU-Belastung (in ms) war bei drei der vier Ansätze nahezu identisch. Ausnahme war der reflexive Ansatz mit einer um Faktor 3 (!) grösseren CPU-Belastung.
- **Programmgrösse**[25]. Die konventionelle Implementierung führte insgesamt zu wesentlich (1.5- bis 3-mal) weniger Methoden und Klassen, die Klassen waren jedoch im Schnitt

[25] Als Mass für die Programmgrösse wurden NCSS (non-commented source statements) gewählt.

grösser und umfassten mehr Methoden als bei den aspektorientierten Implementierungen. Das andere Extrem bildeten die objekt- und architekturorientierten Ansätze. Sie hatten insgesamt am meisten Methoden, Klassen und Codezeilen, jedoch waren die Methoden und Klassen im Schnitt am kleinsten. Der linguistische Ansatz lag dazwischen. Dieser Befund erstaunt nicht angesichts des *Tangling*-Problems, das bei konventionellen Implementierungen auftritt. Was hingegen erstaunt, ist die Feststellung, dass die konventionelle Implementierung insgesamt mit Abstand am wenigsten Codezeilen umfasste.

- **Komplexität**. Die zyklomatische Komplexität war bei allen Implementierungen etwa gleich gross. Damit scheinen sich zwei entgegen gesetzte Auswirkungen der aspektorientierten Programmierung auf die Komplexität aufzuheben: Einerseits die Reduktion durch die Separation, anderseits die Erhöhung durch die Integration.
- **Wiederverwendbarkeit**. Hier schnitten die linguistische und die objektorientierte Implementierung am besten ab. Sie wurden mit *mittel* bewertet, die architekturorientierte und die konventionelle Implementierung mit *tief*. Die Wiederverwendbarkeit war dort am besten, wo bereits vorhandene Teile integriert werden konnten. Zu vermuten ist, dass die linguistische Implementierung besser abgeschnitten hätte, wenn vorgefertigte abstrakte Aspekte für die Synchronisation und die Zeitplanung vorhanden gewesen wären.
- **Anpassbarkeit**. Auch hier lag die konventionelle Implementierung im Hintertreffen. Sie wurde mit *tief* bewertet, die drei aspektorientierten Implementierungen mit *mittel* bis *hoch*. Dieses Ergebnis entspricht den Erwartungen.
- **Interaktion**. Die Interaktionen zwischen Komponenten und Aspekten waren bei einer schmalen Aspektschnittstelle (z.B. bei der Synchronisation und Zeitplanung) einfacher zu verstehen als bei einer breiten Aspektschnittstelle (z.B. beim mathematischen Modell). Auch das ist nicht weiter verwunderlich.

Erkenntnisse. Das abschliessende – und für überzeugte Anhänger der aspektorientierten Programmierung vielleicht ernüchternde – Fazit der Untersuchung ist, dass die Architektur einer Anwendung wichtiger ist als die Technik, die zur Implementierung von Aspekten verwendet wird. Aspekte sollten also möglichst früh in die Architektur integriert werden. Schliesslich haben sich die linguistischen Ansätze als geeignete Grundlage für gute Entwürfe erwiesen. Dabei ist zu berücksichtigen, dass weder diese noch die nächste Untersuchung repräsentativ ist.

9.10.2 Auswirkungen auf Entwicklung und Pflege

Murphy et al. (2001) wollten untersuchen, ob die aspektorientierte Programmierung, verkörpert durch AspectJ, die Entwicklung und Pflege von Software erleichtert. Zu diesem Zweck führten sie zwei Experimente sowie zwei Fallstudien durch.

Experimente. Das eine war ein Experiment zur Fehlersuche und -behebung in einem Programm mit mehreren *Threads*, das andere war ein Experiment zum Ändern eines verteilten Systems. Die Systeme wurden einmal mit AspectJ, einmal mit einer Kontrollsprache (Java für das erste, Emerald (Black et al., 1986) für das zweite Experiment) implementiert.

Ergebnisse. Im ersten Experiment gelang es dem AspectJ-Team schneller als der Kontrollgruppe, die Fehler zu suchen und zu beheben. Ausserdem fanden im AspectJ-Team weniger Diskussionen über die Codesemantik statt. Im zweiten Experiment hingegen waren beide Teams gleich schnell. Dies wird auf die Tatsache zurückgeführt, dass das AspectJ-Team nicht genug Zeit aufwendete, den Code vor der Änderung gründlich zu analysieren, sondern fälschlicherweise davon ausging, das Problem könne allein mit der Aspektsprache behoben werden.

Erkenntnisse. Erstens kommt es darauf an, wie weit sich der Aspektcode auf den Komponentencode auswirkt. Beim ersten Experiment konnten die Entwickler das Problem allein aufgrund des

Aspektcode lösen, beim zweiten war dies nicht möglich. Zweitens verändert die Anwesenheit von Aspekten im Code die Art und Weise, wie die Teams an ihre Aufgabe herangehen: Das AspectJ-Team suchte immer zuerst nach einer Lösung, die sich in einem Aspekt modularisieren liess. Gelang dies, waren sie schneller als die Kontrollgruppe. Gelang dies nicht, waren sie langsamer.

Fallstudie Atlas. Ferner wurde mit AspectJ eine Fallstudie durchgeführt. Das zu implementierende System war eine verteilte Web-basierte Lernumgebung namens Atlas. Folgende Aspekte wurden bestimmt: Konfiguration, Kapselung von Entwurfsmustern, Debugging und Tracing. Die wichtigste Erkenntnis war, dass die Beziehungen zwischen Komponenten und Aspekten sorgfältig analysiert werden müssen.

Fallstudie zum Vergleich von AspectJ und Hyper/J. Eine zweite Fallstudie umfasste einen Vergleich von *Separation of Concerns*-Technologien (u.a. AspectJ und Hyper/J) auf der Basis existierender Programme.

Erkenntnisse. Aus den beiden Experimenten und den beiden Fallstudien liessen sich die folgenden Erkenntnisse gewinnen:

- Crosscutting Concerns, die nicht von Anfang an als Aspekte erkannt und entsprechend entworfen werden, erfordern später die Restrukturierung des Komponentencodes. Zu diesem Zweck wären Werkzeuge hilfreich.
- Die Entwicklung und Pflege ist einfacher, wenn die Aspekt-Schnittstellen *schmal* und *unidirektional* sind, d.h. wenn die Aspekte wohl definierte Auswirkungen auf den Komponentencode haben. Auch hier wären Werkzeuge hilfreich, um zu zeigen, wie sich die Aspekte auf die Komponenten auswirken.
- Aspekte sind einfacher zu verstehen und zu verwalten, wenn sich der Aspektcode auf die Integration der Aspekte in die Komponenten beschränkt. Die eigentliche Funktionalität eines Crosscutting Concern soll nicht als Aspekt, sondern in der Komponentensprache implementiert und über einen Aspekt mit dem restlichen Komponentencode verbunden werden. (Diese Erkenntnis bezieht sich auf die im Abschnitt 8.2 erwähnte Unterscheidung zwischen dem Client- und dem Server-Teil eines Crosscutting Concern.)

Offene Punkte. Offen und weiter zu untersuchen bleibt, ob die aspektorientierte Programmierung auch in grossen Projekten mit vielen Entwicklern funktioniert, für welche Arten von Problemen sie sich am besten eignet und womit man Crosscutting Concerns am besten spezifiziert.

9.11 Vor- und Nachteile der aspektorientierten Programmierung

Zusammenfassend werden die grundsätzlichen Vor- und Nachteile der aspektorientierten Programmierung aufgezeigt.

9.11.1 Vorteile der aspektorientierten Programmierung

Die aspektorientierte Programmierung hat folgende Vorteile gegenüber der konventionellen Programmierung. Sie beziehen sich zwar auf AspectJ, gelten aber sinngemäss auch für die anderen Ansätze der aspektorientierten Programmierung. Die ersten sieben Vorteile stammen von Laddad (2003), die beiden letzten von Kiczales et al. (2001b).

- **Verantwortung**. Jedes Modul hat eine klare Verantwortung. Dies führt im Endeffekt zu einer besseren (Rück-) Verfolgbarkeit.
- **Modularisierung**. Der Code ist durch die Vermeidung von Scattering und Tangling insgesamt besser modularisiert. Dies reduziert die Redundanz und erhöht die Verständlichkeit und Pflegbarkeit der Software. Ob sich durch die bessere Modularisierung auch der Code-

umfang reduziert, ist umstritten. Kiczales et al. (1997) meinen, dass sich der Codeumfang beträchtlich reduziert (siehe auch im Abschnitt 13.10), während die Untersuchung von Diaz Pace, Campo (2001) das Gegenteil zu belegen scheint. Intuitiv hat man den Eindruck, die aspektorientierte Programmierung müsste den Codeumfang eigentlich reduzieren. Mit grosser Wahrscheinlichkeit hängt dies von der Anzahl der Komponenten ab, die ein Aspekt quer schneidet: je mehr Komponenten dies sind, desto mehr „scattered" Code kann eingespart werden.

- **Pflegbarkeit**. Die Pflege wird einfacher. Die Komponenten wissen nichts (engl. oblivious) von den Aspekten, die sie quer schneiden. Aspekte können daher *additiv* hinzugefügt, geändert oder entfernt werden, also nicht *invasiv* wie bei einer konventionellen Implementierung. Dies birgt die Gefahr, dass den Crosscutting Concerns in den frühen Phasen des Entwicklungsprozesses zu wenig Beachtung geschenkt wird in der irrigen Meinung, die entsprechenden Aspekte könnten später problemlos hinzugefügt werden. Dass dies ein Trugschluss ist, zeigt der Abschnitt 13.1.
- **Einfachheit**. Die späte Integration[26] von Entwurfsentscheidungen macht es möglich, sich während des gesamten Entwicklungsprozesses auf die aktuell relevanten Anforderungen zu beschränken. Architektur und Entwurf können sich auf das Wesentliche konzentrieren und müssen nicht versuchen, zukünftige Anforderungen vorauszusehen und zu berücksichtigen. Dadurch wird das System so einfach wie möglich gehalten. Dies steht im Einklang mit der Forderung von *Extreme Programming* (Beck, 2000) nach Einfachheit des Entwurfs.
- **Wiederverwendung**. Aspektmodule sind loser gekoppelt, als sie es bei einer konventionellen Implementierung wären. Ausserdem sind Aspekte naturgemäss weniger anwendungsspezifisch als Komponenten. Dies führt dazu, dass Aspekte im Allgemeinen gut wiederverwendbar sind.
- **Time-to-market**. Die späte Integration von Entwurfsentscheidungen erlaubt einfachere Entwürfe und führt damit zu einer Reduktion der Entwurfs- und Implementierungszeiten. Die saubere Modularisierung erlaubt eine bessere Arbeitsteilung, eine parallele Entwicklung und damit eine höhere Produktivität. Wiederverwendung reduziert die Entwicklungszeit. Alles in allem können Systeme schneller entwickelt und am Markt eingeführt werden.
- **Kosten**. Die Einfachheit und die schnelle Time-to-market wirken sich einerseits positiv auf die Entwicklungskosten aus. Anderseits kann das System früher genutzt werden, wodurch sich die Entwicklungskosten früher amortisieren. Doch nicht nur das: Aufgrund der verbesserten Pflegbarkeit werden die Lebenszykluskosten der Software insgesamt reduziert.
- **Schrittweise Einführung**. Entwicklungsaspekte bieten Unternehmen, die konservativ und gegenüber neuen Technologien skeptisch sind, einen idealen Einstiegspunkt in die Aspektorientierung. Sie können während der Entwicklungs- und Testphase erste Gehversuche mit Aspekten machen (z.B. mit Debugging-Aspekten) und diese vor der produktiven Inbetriebnahme wieder entfernen.
- **Stabilität**. Die Implementierungen werden stabiler. Wenn beispielsweise eine Komponentenmethode hinzugefügt oder überschrieben wird, ist die Gefahr kleiner als bei einer konventionellen Implementierung, dass dabei die Crosscutting Concerns vergessen gehen.

Die Wechselwirkungen von einigen der erwähnten Vorteile der aspektorientierten Programmierung lassen sich grafisch zusammenfassen, und dies kann wie in der Abbildung 9-14 dargestellt werden. Dabei bedeutet ein Pfeil mit einem Pluszeichen „hat einen positiven Einfluss auf…".

[26] Laddad (2003) verwendet hier den Begriff *spätes Binden*, was in diesem Zusammenhang zu Missverständnissen führen kann.

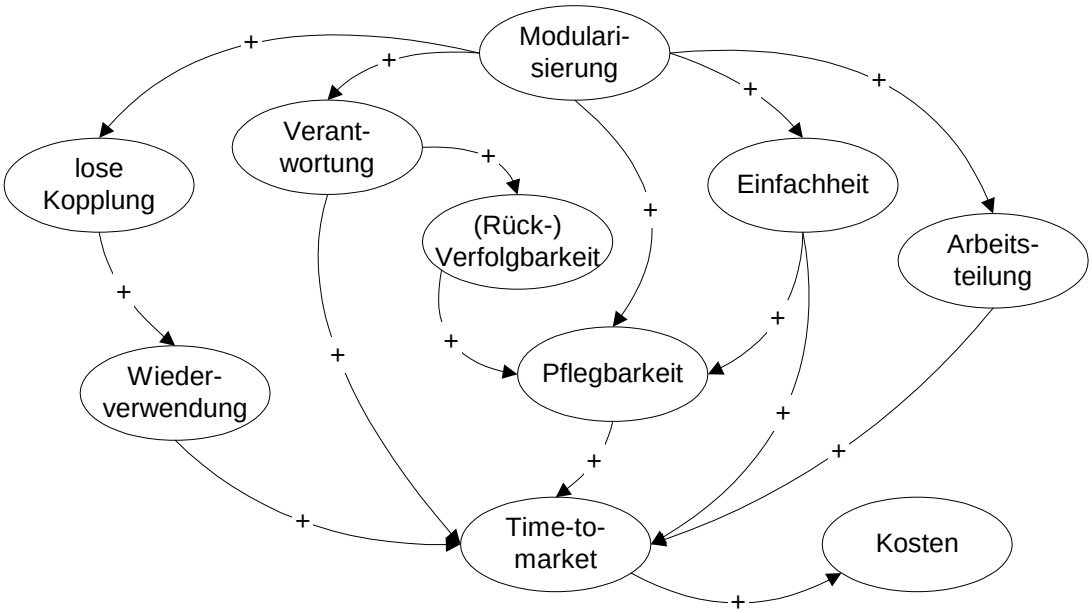


Abbildung 9-14: Wechselwirkungen der Vorteile der aspektorientierten Programmierung

9.11.2 Nachteile der aspektorientierten Programmierung

Bei all diesen Vorteilen stellt sich natürlich sofort die Frage nach den Nachteilen. Diese sind in der Tat nicht so offensichtlich. Zu erwähnen sind:

- **Performance**. Insbesondere bei dynamischem Weaving sind Performance-Nachteile zu gewärtigen. Bei statischem Weaving ist die Performanceverschlechterung aufgrund des Inlining nicht spürbar.
- **Disziplin**. Die aspektorientierte Programmierung erfordert von den Entwicklungsteams eine bis dahin nicht gekannte Disziplin, z.B. in der Einhaltung von Namenskonventionen. Die Frage ist, ob sich diese bei grösseren Entwicklungsteams oder in ganzen Unternehmen durchsetzen lässt.
- **Debugging**. In einem aspektorientierten System ist es nicht einfach, bei der Fehlersuche dem Kontrollfluss zu folgen, da dieser nicht ohne weiteres aus der Struktur des Quellcode abgeleitet werden kann. (Ein ähnliches Problem stellt sich auch in der objektorientierten Programmierung mit der dynamischen Bindung). Wie Dijkstra (1968) in seinem Essay „Go To Statement Considered Harmful“ ausführt, ist in der Programmierung anzustreben, dass die (dynamische) Prozessstruktur möglichst mit der (statischen, als Text vorliegenden) Programmstruktur des entsprechenden Programms übereinstimmt. Werte von Variablen lassen sich beispielsweise nur in Bezug auf den Prozessfortschritt korrekt interpretieren. Dieser ist aber nicht unter Kontrolle des Programmierers und muss anhand der Programmstruktur bestimmt werden. Dass dies am einfachsten ist, wenn die Programmstruktur der Prozessstruktur entspricht, liegt auf der Hand. Unter anderem aus diesem Grund sind die berühmten „go to“-Anweisungen so verpönt.
- **Kapselung**. Die aspektorientierte Programmierung bricht die Kapselung, indem eine Klasse nicht mehr ihr gesamtes Verhalten selber unter Kontrolle hat.
- **Kopplung**. Auch wenn die aspektorientierte Programmierung die Kopplung zwischen den Modulen an gewissen Stellen reduziert, erhöht sie sie gleichzeitig an anderen Stellen: Theoretisch müssen zwar die Komponenten nichts von den Aspekten wissen. Die Tatsache, dass die meisten Pointcuts über Namen von Programmkonstrukten wie Paketen, Klassen,

Methoden und Variablen definiert werden, führt zu einer neuen Art von Kopplung zwischen Komponenten und Aspekten: Das korrekte Funktionieren eines Aspekts hängt von der Namensgebung sämtlicher (!) Programmkonstrukte in den Komponenten ab. Namen dürfen in den Komponenten keinesfalls verändert werden. Neu hinzugefügte Programmkonstrukte werden in einem Aspekt nur dann berücksichtigt, wenn sie sich a) strikt an die Namenskonventionen halten und b) ihre Behandlung im Aspekt vorgesehen ist.

- **Vertragsprinzip**. Das Vertragsprinzip kann durch die Aspektorientierung kompromittiert werden, wie das im Abschnitt 14.4 detailliert ausgeführt wird.
- **Manipulation**. In Ansätzen mit mehrheitlich transparenten Aspekten, die additiv hinzugefügt bzw. entfernt werden können, besteht die Gefahr, dass jemand die Software auf unlautere Art manipuliert, ohne dass dies die Verantwortlichen für die Komponenten oder das Testpersonal bemerken. Diesem Nachteil kann durch geeignete Kontrollen begegnet werden.

Weiter sind noch einige – wahrscheinlich – temporäre Nachteile zu erwähnen, die bestehen, weil es sich bei der Aspektorientierung noch um ein junges (Forschungs-) Gebiet handelt:

- **Werkzeugunterstützung für linguistische Ansätze**. Werkzeuge sind zwar vorhanden, aber insgesamt ist die Werkzeugunterstützung noch mager (siehe auch im Abschnitt 16.2).
- **Erfahrung**. Praktische Erfahrungen mit aspektorientierter Programmierung in grösseren Systemen fehlen weitgehend. Somit sind vielleicht etliche Nachteile der aspektorientierten Programmierung noch gar nicht offensichtlich. Und es muss sich erst weisen, ob die oben erwähnten – potenziellen – Vorteile auch wirklich zum Tragen kommen.
- **Entwicklungsprozess**. Es gibt zurzeit noch keine durchgängige Methode für die Behandlung von Crosscutting Concerns in allen Phasen des Entwicklungsprozesses. Dies bedeutet, dass die rechtzeitige Entdeckung von Crosscutting Concerns als Vorgabe für Entwurf und Implementierung bis zu einem gewissen Grad dem Zufall überlassen ist.

9.12 Gegenüberstellung der Konzepte der aspektorientierten Ansätze und Sprachen

Will man die Konzepte der aspektorientierten Ansätze und Sprachen einander gegenüberstellen, muss zwischen ansatzspezifischen und ansatzneutralen Konzepten unterschieden werden. *Ansatzspezifische Konzepte* kommen nur in bestimmten Ansätzen und Sprachen vor und können folglich auch nur dort eingesetzt werden, während *ansatzneutrale Konzepte* allen Ansätzen und Sprachen gemeinsam sind und als *Essenz der aspektorientierten Programmierung* betrachtet werden können. Diese Unterscheidung ist wichtig, weil den ansatzneutralen Konzepten in der Anforderungsphase, die bekanntlich unabhängig vom gewählten Implementierungsansatz sein sollte, besondere Beachtung geschenkt werden muss. Im Kapitel 13 werden ausschliesslich ansatzneutrale Konzepte behandelt. Ausserdem können die Ansätze der aspektorientierten Anforderungstechnik (siehe im Kapitel 12) daran gemessen werden, wie weit sie ausschliesslich ansatzneutrale Konzepte einsetzen.

9.12.1 Ansatzneutrale Konzepte

Als ansatzneutrale Konzepte können betrachtet werden:

Separation. In allen Ansätzen und Sprachen gibt es Möglichkeiten, die Crosscutting Concerns getrennt von den Core Concerns zu implementieren. Daraus entstehen Aspekte und Komponenten.

Integration. In allen Ansätzen und Sprachen gibt es Möglichkeiten, die Regeln für die Zusammenführung von Aspekten und Komponenten zu implementieren. (Die *eigentliche* Integration

findet erst anlässlich der Übersetzung oder zur Laufzeit statt, so dass nur die Integrations*regeln* zu implementieren sind.) Diese Regeln bestehen aus zwei Teilen:

1. **Stellen der Zusammenführung**. Dieser Regelteil beschreibt das *Wo*, nämlich die Stellen, an denen Aspekte und Komponenten zusammengeführt werden. Um das Wo zu definieren, stehen in den betrachteten Ansätzen und Sprachen zwei grundsätzlich verschiedene Möglichkeiten zur Verfügung. Die eine Möglichkeit ist das *Weaving*, wie man es beispielsweise aus AspectJ und den meisten anderen Ansätzen kennt: Man definiert eine oder mehrere Stelle(n) im Komponentencode, an denen der Aspektcode integriert werden soll. Die andere Möglichkeit ist die *Traversierung*, wie sie in der DJ Library und in DemeterJ implementiert ist: Man definiert in den Komponenten eine Klassenstruktur, deren Objekte traversiert und dabei um das Aspektverhalten erweitert werden sollen.
2. **Art der Zusammenführung**. Dieser Regelteil beschreibt das *Wie*, nämlich die Art, wie Aspekte und Komponenten zusammengeführt werden. Auch für die Definition des Wie stehen in den betrachteten Ansätzen und Sprachen zwei grundsätzlich verschiedene Möglichkeiten zur Verfügung: Entweder wird das *Verhalten* der Komponenten ersetzt oder erweitert (wie z.B. in AspectJ und den meisten anderen Ansätzen) oder *Nachrichten* werden manipuliert (wie z.B. in den Composition Filters und Metaobjektprotokollen).

Die Umsetzung dieser Regeln unterscheidet sich in den betrachteten Ansätzen und Sprachen teilweise beträchtlich. Die Tabelle 9-8 zeigt, mit welchen Sprachkonstrukten die ansatzneutralen Konzepte in den verschiedenen Ansätzen und Sprachen realisiert wurden.

Ansatz, Sprache	Separation	Integration – Wo	Integration – Wie
AspectJ	Advice	Pointcut	Schlüsselwörter `before`, `after` und `around`
DJ Library, DemeterJ	adaptiver Besucher	Traversierungs-strategie	Schlüsselwörter `start`, `before` etc.
Hyper/J	Hyperslice, Klasse	Hypermodul	Schlüsselwörter `mergeByName`, `nonCorrespondingMerge` in Kombination mit `equate`, `over-ride`, `bracket` in Kombination mit `before` oder `after` etc.
Composition Filters	Filter	Überlagerung	Art (Input oder Output), Typ und Reihenfolge der Filter
Aspect Moderator Framework	Aspekt-klasse	Aspekt-moderator	`preActivation`- und `postActivation`-Methode
Metaobjektprotokoll mit MetaJava	Metaklasse	Klasse deklariert Metaklasse als `metaclass`	implizit durch die Folge der Methodenaufrufe

Tabelle 9-8: Sprachkonstrukte der ansatzneutralen Konzepte

In einer symmetrischen Sprache wie Hyper/J können Aspekte und Komponenten a priori nicht unterschieden werden, da sie mit den gleichen Sprachkonstrukten (z.B. Methoden, Klassen) implementiert sind. Aus diesem Grund wird in Hyper/J mit dem Hyperslice ein zusätzliches Sprachkonstrukt benötigt, um die Zugehörigkeit einer Methode oder Klasse zu einem bestimmten Concern zu definieren.

9.12.2 Ansatzspezifische Konzepte

Demgegenüber zeigt die Tabelle 9-9 eine Auswahl ansatzspezifischer Sprachkonstrukte der betrachteten Ansätze und Sprachen. Sie können in der Anforderungsphase vernachlässigt werden.

Ansätze und Sprachen	ansatzspezifische Sprachkonstrukte
AspectJ	Aspekt, Declaration, Introduction
DJ Library, DemeterJ	adaptive Methode, Klassengraph
Hyper/J	Concern Mapping, Hyperspace, deklarative Vollständigkeit
Composition Filters	CF-Klasse, Filter-Interface
Aspect Moderator Framework	Funktionalitäts-Proxy, Aspektfabrik
Metaobjektprotokoll mit MetaJava	Standard-Metaklasse

Tabelle 9-9: Sprachkonstrukte der ansatzspezifischen Konzepte

Dass der Aspekt von AspectJ und die adaptive Methode der DJ Library und von DemeterJ unter den ansatzspezifischen Sprachkonstrukten figurieren, hat seinen Grund: Sowohl Aspekte als auch adaptive Methoden fassen Separations- und Integrationskonzepte zusammen. Ein solches Sprachkonstrukt existiert in den anderen Ansätzen und Sprachen nicht.

10. Ansätze des aspektorientierten Entwurfs

Es sind einige wenige Ansätze des aspektorientierten Entwurfs zu finden, die – wie die Ansätze der aspektorientierten Programmierung auch – sehr unterschiedlich sind. Ihre Beurteilung erfolgt anhand der folgenden Kriterien:

- Ist der Ansatz gut verständlich und einfach zu erlernen?
- Kann der Ansatz auf Ansätze der aspektorientierten Programmierung abgebildet werden?
- Eignet sich der Ansatz für den Entwurf grösserer Systeme?

10.1 Subjektorientierung

Ausgangslage für die *Subjektorientierung* (Clarke et al., 1999) ist das Problem, dass die Abstraktion und Dekomposition von Anforderungen im Normalfall benutzerorientiert ist und sich auf die Funktionalität bezieht, während Entwurf und Code später (z.B. im Fall von objektorientierter Implementierung) nach Klassen gegliedert und modularisiert werden. Dieser Strukturbruch führt zu *Scattering* (eine Anforderung ist über mehrere Klassen verstreut) und *Tangling* (eine Klasse implementiert mehrere Anforderungen) und folglich zu schlechter Verständlichkeit, Wiederverwendbarkeit, (Rück-) Verfolgbarkeit und Pflegbarkeit. Die Subjektorientierung versucht, diesen Strukturbruch zu überwinden. In der Entwurfsphase werden Teile der Funktionalität, die aus Benutzersicht zusammen gehören (so genannte *Subjekte*), unabhängig von anderen Subjekten modelliert und implementiert. Dabei erfolgt die Modellierung der einzelnen Subjekte ganz normal objektorientiert. In der UML werden die Subjekte als Pakete (engl. package) dargestellt. Damit werden auch Entwurf und Code – zumindest auf oberster Ebene – nach Funktionalität gegliedert, und die Gliederung nach Klassen findet erst innerhalb der Subjekte statt. Die spätere Integration der Subjekte zum Gesamtsystem erfolgt aufgrund so genannter *Kompositionsbeziehungen*. Sie definieren, an welchen Stellen und in welcher Art die Subjekte überlappen.

Sowohl die MDSOC und Hyper/J als auch die Composition Patterns (siehe im Abschnitt 10.2) entwickelten sich aus den Ideen der Subjektorientierung.

Hauptvorteil der Subjektorientierung ist, dass sie den Strukturbruch zwischen der Anforderungs- sowie der Entwurfs- und Implementierungsphase überwindet. Damit verbessern sich Verständlichkeit, (Rück-) Verfolgbarkeit und Pflegbarkeit der Software. Zudem können Subjekte (z.B. ein Logging-Subjekt) wiederverwendet werden. Aufgrund der Möglichkeit des *Mix-and-match* von Funktionalität mit Hilfe der Kompositionsbeziehungen können Änderungen additiv statt invasiv vorgenommen werden. Da jedes Subjekt seine eigene Sicht überlappender Entwurfselemente spezifiziert, ist paralleles Entwerfen möglich. Natürlich kann die Subjektorientierung auch bereits während der Anforderungsphase eingesetzt werden.

Ein Nachteil der Subjektorientierung ist, dass sie das Scattering- und Tangling-Problem eigentlich nicht löst, sondern lediglich um eine Ebene nach unten verlagert: Innerhalb der Subjekte bleiben das Scattering und das Tangling bestehen. Aus dem Vorteil der Unabhängigkeit der Subjekte entspringt ein weiterer nicht zu unterschätzender Nachteil: Je unabhängiger voneinander die Subjekte entworfen werden, desto grösser ist später der Aufwand, sie zu einem Gesamtsystem zu integrieren.

Beurteilung. Die Subjektorientierung kann als Operationalisierung der *Separation of Concerns* betrachtet werden, sie ist aber eher eine Philosophie als ein konkreter Ansatz. Das Wissen über die Subjektorientierung als Ursprung der MDSOC und von Hyper/J ist wichtig, um den feinen Unterschied zwischen Hyper/J und anderen Ansätzen (z.B. der aspektorientierten oder adaptiven Programmierung) zu verstehen. Zur Begriffswahl: Die Subjekte entsprechen den Concerns aus dem Kapitel 5. Es ist wichtig zu beachten, dass die Subjektorientierung und alle von ihr abgeleiteten Ansätze symmetrisch sind und keinen Unterschied zwischen Core und Crosscutting Concerns bzw. zwischen Aspekten und Komponenten machen. Alle Subjekte sind gleichwertig.

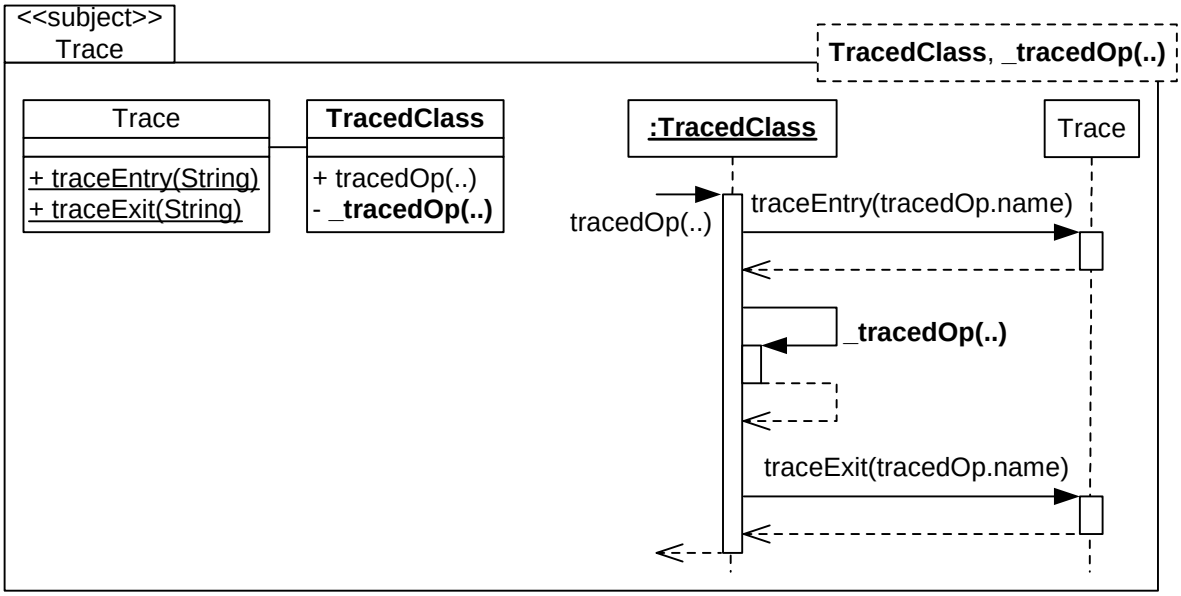
10.2 Composition Patterns

Composition Patterns ermöglichen den separaten und wiederverwendbaren Entwurf von „crosscutting“ Anforderungen. Sie basieren auf der Idee der Subjektorientierung und werden mit Hilfe einer leicht erweiterten UML und von Kompositionsbeziehungen modelliert. Wo nichts anderes erwähnt ist, stammt die Beschreibung dieses Ansatzes von Clarke, Walker (2001b).

> **Definition** *Composition Pattern*. Entwurfsmodell, das den Entwurf von „crosscutting“ Subjekten unabhängig von anderen Subjekten ermöglicht.

Zur Darstellung von Subjekten werden in der UML, wie im Abschnitt 10.1 erwähnt, Pakete verwendet. Um Composition Patterns darstellen zu können, wird die UML um *Paket-Schablonen* (engl. Templates) erweitert. Diese enthalten Parameter als Platzhalter für die spätere Ersetzung durch tatsächliche (d.h. nicht-generische) Elemente. Die Kompositionsbeziehungen zwischen einem Composition Pattern und anderen Subjekten definieren die tatsächlichen Elemente, welche die Parameter im Composition Pattern ersetzen. Die Standard-UML reicht für die Modellierung von Composition Patterns nicht aus, da keine Semantik für das Verschmelzen von „crosscutting“ Verhalten mit anderem Verhalten vorhanden ist (Clarke, Walker, 2001a).

Separation. In einem ersten Schritt wird jedes „crosscutting“ Subjekt als Paket-Schablone dargestellt. Die Abbildung 10-1 zeigt, wie das Tracing-Beispiel aus dem Abschnitt 9.1 mit einem Composition Pattern modelliert werden kann, das aus einem UML-Klassendiagramm und einem UML-Sequenzdiagramm besteht.

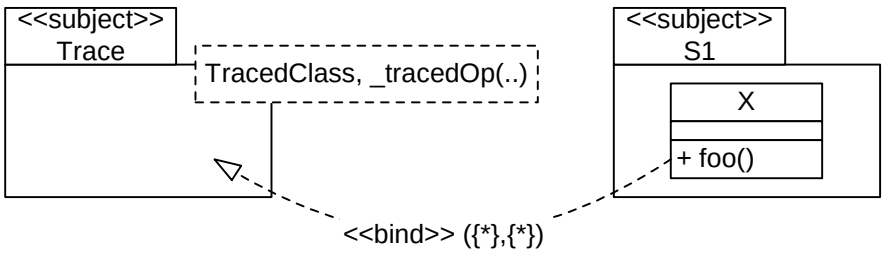


Quelle: Clarke, Walker (2001b)

Abbildung 10-1: Spezifikation eines Composition Pattern

Die parametrisierbare `TracedClass`-Klasse und die parametrisierbare `_tracedOp`-Methode (fett gedruckt) sind die Parameter des Composition Pattern. `TracedClass` repräsentiert eine beliebige Klasse, deren `_tracedOp`-Methode um Tracing-Verhalten erweitert werden soll. Im Beispiel aus dem Abschnitt 9.1 war dies die `foo`-Methode der `X`-Klasse. Die Notation `(..)` bedeutet, dass die parametrisierbare Methode durch eine tatsächliche Methode mit einer beliebigen Signatur ersetzt werden darf. Das UML-Sequenzdiagramm zeigt das Verhalten der `tracedOp`-Methode, welche die parametrisierbare `_tracedOp`-Methode um Tracing-Verhalten erweitert: Zuerst wird `traceEntry()` ausgeführt, dann die parametrisierbare `_tracedOp`-Methode und schliesslich `traceExit()`. Im Gegensatz zum Beispiel aus dem Abschnitt 9.1 ist das Tracing-Verhalten nun vollständig im Composition Pattern enthalten und nicht mehr in der `X`-Klasse. Damit treten in dieser Lösung auch keine *Scattering*- und *Tangling*-Probleme auf.

Integration. In einem zweiten Schritt werden die Parameter des Composition Pattern durch tatsächliche Klassen und Methoden ersetzt. Die Parameterersetzung wird wie in der Abbildung 10-2 gezeigt durch eine Verfeinerungsbeziehung mit dem Stereotyp `<<bind>>` dargestellt.



Quelle: Clarke, Walker (2001b)

Abbildung 10-2: Parameter-Ersetzung eines Composition Pattern

Dabei bedeutet `({*},{*})`, dass alle Methoden aus allen Klassen im tatsächlichen Subjekt S1 um das Tracing-Verhalten erweitert werden sollen. In diesem Beispiel ist dies die tatsächliche Methode `foo()` aus der tatsächlichen Klasse `X`.

Composition Patterns und AspectJ. Die Subjekte inkl. Composition Patterns werden möglichst unabhängig voneinander implementiert, z.B. mit AspectJ oder Hyper/J. Dadurch lässt sich die mit den Composition Patterns erreichte *Separation of Concerns* bewahren. Die Konzepte von AspectJ lassen sich wie in der Tabelle 10-1 aufgelistet auf die Konzepte der Composition Patterns abbilden:

AspectJ	Composition Patterns
Aspekt	„crosscutting" Subjekt
Pointcut	Parametrisierbare Methode im Sequenzdiagramm, die durch eine oder mehrere tatsächliche Methode(n) ersetzt wird
Advice	Verhalten, das im Sequenzdiagramm spezifiziert ist
Introduction	In einer parametrisierbaren Klasse zusätzlich spezifizierte nicht parametrisierbare Variablen und Methoden

Tabelle 10-1: Abbildung von AspectJ-Konzepten auf Composition Pattern-Konzepte

Von den Composition Patterns zu AspectJ und Hyper/J. Clarke, Walker (2001a) wagen den Versuch, Composition Patterns anhand eines Beispiels auf AspectJ und Hyper/J abzubilden. Ziel ist eine nahtlose und rückverfolgbare Abbildung zwischen Entwurf (Composition Patterns) und Code (AspectJ bzw. Hyper/J). Beispielanwendung ist eine Bibliothekverwaltung, die mit dem Beobachtermuster (Gamma et al., 1996) realisiert werden soll. Beobachter ist die Buchverwalter-Klasse, Subjekt ist die Buch-Klasse. Sobald ein neues Buch erfasst wird, meldet sich der Buchverwalter bei diesem Buch als Beobachter an. Wird ein Buch gelöscht, meldet sich der Buchverwalter als Beobachter dieses Buchs ab. Wird ein Buch ausgeliehen oder zurückgegeben, wird der Buchverwalter benachrichtigt, damit er den Ausleihstatus aktualisieren kann. Die Abbildung des Besuchermusters auf ein Composition Pattern sowie die Realisierung mit AspectJ und Hyper/J werden hier aus Platzgründen nicht detailliert, sondern nur im Überblick erklärt.

Abbildung des Besuchermusters auf ein Composition Pattern. Zwei Subjekte[27] werden entworfen, ein gewöhnliches Bibliotheksubjekt und ein parametrisierbares Beobachtersubjekt. Das *Bibliotheksubjekt* enthält die Klassen der Bibliothekverwaltung. Das *Beobachtersubjekt* enthält die parametrisierbaren Klassen des Beobachtermusters: einerseits eine `Subjekt`[28]-Klasse mit gewöhnlichen Methoden namens `meldeAn`, `meldeAb` und `benachrichtige` sowie einer parametrisierbaren `ändereZustand`-Methode, welche die Benachrichtigung der Beobachter bewirkt; anderseits eine `Beobachter`-Klasse mit parametrisierbaren Methoden namens `starte`, `stoppe` und `aktualisiere`. Die Parameter-Ersetzung im Beobachtersubjekt zeigt die Tabelle 10-2:

parametrisierbare Klasse	ersetzt durch
Subjekt	Buch
Beobachter	Buchverwalter

parametrisierbare Methode	ersetzt/erweitert durch
ändereZustand	leiheBuchAus, gibBuchzurück
starte	erfasseBuch
stoppe	löscheBuch
aktualisiere	aktualisiereAusleihstatus

Tabelle 10-2: Parameter-Ersetzung im Beispiel der Bibliotheksverwaltung

Implementierung des Composition Pattern mit AspectJ. Zur Realisierung dieses Composition Pattern durch einen konkreten Aspekt werden die folgenden AspectJ-Konstrukte benutzt:

- **Introduction**: Die gewöhnlichen `meldeAn`-, `meldeAb`- und `benachrichtige`-Methoden werden zur `Buch`-Klasse hinzugefügt.

[27] Darunter sind Subjekte in der Terminologie der Subjektorientierung zu verstehen. Der Begriff wird bis zum Ende des Beispiels nur noch in zusammengesetzter Form (Bibliotheksubjekt bzw. Beobachtersubjekt) verwendet.

[28] Darunter sind Subjekte in der Terminologie des Beobachtermusters zu verstehen.

Die Implementierung der `benachrichtige`-Methode, welche die `aktualisiere`-Methode des Beobachters aufruft, ist ein Spezialfall: Da die `aktualisiere`-Methode im Unterschied zu den anderen parametrisierbaren Methoden die tatsächliche Methode der Buchverwalter-Klasse nicht erweitert, sondern ersetzt, kann die `benachrichtige`-Methode direkt die `aktualisiereAusleihstatus`-Methode der Buchverwalter-Klasse aufrufen.

- Der **Pointcut** führt die Parameter-Ersetzung des Composition Pattern durch. Pro parametrisierbare Methode (`ändereZustand`, `starte`, `stoppe`) wird ein Pointcut definiert, der die tatsächlichen Methoden identifiziert, zum Beispiel alle Ausführungen der `erfasseBuch`-Methode der `Buchverwalter`-Klasse:

```
pointcut starte(Buch buch, Buchverwalter buchverwalter) :
     this (buchverwalter) &&
     execution (void erfasseBuch(buch));
```

- Der **Advice** implementiert das Sequenzdiagramm des Composition Pattern. Pro Pointcut (`ändereZustand`, `starte`, `stoppe`) wird ein Advice definiert, der die tatsächlichen Methoden der `Buch`-Klasse (`benachrichtige`, `meldeAn`, `meldeAb`) an den Join Points einfügt. Der folgende Advice führt das obige Beispiel fort und ruft die `meldeAn`-Methode der Buch-Klasse auf, nachdem die `erfasseBuch`-Methode ausgeführt wurde:

```
after (Buch subjekt, Buchverwalter beobachter) :
     starte(subjekt, beobachter) {
     subjekt.meldeAn(beobachter);
}
```

- **Beurteilung**. Die Abbildung der Composition Patterns auf konkrete Aspekte hat den Vorteil, dass sie automatisiert werden kann, sofern ein geeignetes Werkzeug zur Verfügung steht. Der Nachteil ist, dass mehrere konkrete Aspekte implementiert werden müssen, wenn das Beobachtersubjekt mehrfach verwendet werden soll. Dies führt letztlich zu einem *Scattering* des Beobachtersubjekts. Abhilfe würde die Implementierung mit einem abstrakten Aspekt schaffen.

Implementierung des Composition Pattern mit Hyper/J. Aufgrund der Verwandtschaft zwischen Composition Patterns und Hyper/J (beide konkretisieren die Idee der Subjektorientierung) ist die Abbildung dieses Composition Pattern auf Hyper/J relativ einfach.

- **Hyperspace**. Das Bibliothek- und das Beobachtersubjekt werden je in ein Paket eingefügt. Der Hyperspace besteht aus den beiden Paketen.
- Das **Concern Mapping** definiert das `Beobachter`-Paket als `Beobachter`-Feature und das `Bibliothek`-Paket als `Bibliothek`-Feature:

```
package Beobachter : Feature Beobachter
package Bibliothek : Feature Bibliothek
```

  Eine Einschränkung gilt es zu beachten: Zu verschmelzende Operationen müssen in Hyper/J die gleiche Signatur haben. Die Flexibilität der Composition Patterns in Bezug auf die Methodensignaturen (`(..)`-Notation) findet in Hyper/J keine Entsprechung. Aus diesem Grund müssen die parametrisierbaren Methoden im `Beobachter`-Paket genau die gleichen Signaturen haben wie die tatsächlichen Methoden im `Bibliothek`-Paket. Mit anderen Worten: Das `Beobachter`-Paket ist in Hyper/J nicht so leicht wiederverwendbar wie als Composition Pattern.
- Das **Hypermodul** spezifiziert, dass die beiden Pakete mittels `nonCorrespondingMerge` integriert werden sollen. Die parametrisierbaren Klassen werden mittels `equate class` ersetzt. So bedeutet die Anweisung

```
equate class
     Feature.Bibliothek.Buchverwalter
     Feature.Beobachter.Beobachter;
```

dass die `Buchverwalter`-Klasse aus dem `Bibliothek`-Feature die `Beobachter`-Klasse aus dem `Beobachter`-Feature ersetzt. Den Sequenzdiagrammen des Composition Pattern kann entnommen werden, ob die tatsächlichen Methoden die parametrisierbaren Methoden ersetzen oder erweitern. Eine *Ersetzung* wird mit `override` spezifiziert. Die Anweisung

```
override action
     Feature.Beobachter.Beobachter.aktualisiere with
     Feature.Bibliothek.Buchverwalter.aktualisiereAusleihstatus;
```

bedeutet, dass die `aktualisiere`-Methode der `Beobachter`-Klasse durch die `aktualisiereAusleihstatus`-Methode der `Buchverwalter`-Klasse ersetzt wird. Eine *Erweiterung* wird mit `bracket` spezifiziert. Die Anweisung

```
bracket "erfasseBuch" with
     (after Feature.Beobachter.Beobachter.starte,
          "Buchverwalter");
```

implementiert den gleichen Sachverhalt wie der Pointcut und der Advice im obigen Beispiel und bedeutet, dass nach der `erfasseBuch`-Methode der `Buchverwalter`-Klasse die `starte`-Methode der `Beobachter`-Klasse aufzurufen ist. (In der `starte`-Methode wird die `meldeAn`-Methode aufgerufen.)

- **Beurteilung**. Die Nachteile der Realisierung von Composition Patterns mit Hyper/J sind, dass `bracket` auf `before` und `after` beschränkt ist, dass wie erwähnt nur Methoden mit gleicher Signatur verschmolzen werden können, und dass sich daher Hyper/J für komplizierte Interaktionen schlecht eignet.

Beurteilung. Einer der Vorteile der Composition Patterns ist ihre Unabhängigkeit von einem bestimmten Programmierparadigma. Die obigen Beispiele zeigen, dass ein Entwurf mit Composition Patterns zu sauberen und eleganten Implementierungen mit AspectJ und Hyper/J führt. Wünschenswert wäre eine gute Werkzeugunterstützung, nicht zuletzt auch für die Automatisierung der Abbildung von Composition Patterns auf gängige Programmiersprachen. Die Composition Patterns sind abstrakt und konzeptionell nicht ganz einfach zu verstehen. Dafür kommen sie weitgehend mit Standard-UML-Konstrukten aus, so dass ein erfahrener UML-Entwerfer schnell damit zurechtkommt. Schneidet ein „crosscutting" Subjekt mehrere Subjekte quer, dürfte die Darstellung rasch einmal schwierig und unübersichtlich werden. Dann wäre ggf. eine tabellarische Darstellung der Integration vorzuziehen. Schneiden verschiedene „crosscutting" Subjekte das gleiche Subjekt quer, fehlt ein Mechanismus zur Definition der Prioritäten.

Interessant ist, dass die Composition Patterns die Kompositionsbeziehungen aus der Subjektorientierung zweiteilig umsetzen: Die Stelle, an der das „crosscutting" Verhalten eingefügt werden soll (in AspectJ wären das die Schlüsselwörter `Before`, `After` oder `Around`), wird implizit durch das Sequenzdiagramm in der Spezifikation des Composition Pattern beschrieben. Die Methoden und Klassen, die vom „crosscutting" Verhalten betroffen sind (in AspectJ wären das die Pointcuts), werden durch die Parameterersetzung spezifiziert.

10.3 Theme/UML

Theme/UML[29] ist die Entwurfsmethode zu Theme/Doc (siehe im Abschnitt 12.10). Zusammen bilden sie den Theme-Ansatz, ein Nischenprodukt der Aspektorientierung. Auch der Theme-Ansatz basiert auf der Subjektorientierung, nur dass die Subjekte hier *Themen* genannt werden. Theme/UML modelliert die Anforderungsmodelle von Theme/Doc mit Standard-UML-Konstrukten (aus Entitäten werden Klassen, aus Minor Actions werden Methoden) und ergänzt die UML um Kompositionsbeziehungen zur Integration der Themen. Dieser Ansatz wird hier nicht weiter verfolgt.

10.4 Rollenmodelle

Die von Georg, Ray, France (2002) beschriebenen Rollenmodelle sind den *Composition Patterns* von Clarke, Walker (2001b) ähnlich. Die Rollenmodelle entsprechen Entwurfsmustern für Aspekte, auf die später passende UML-Modellelemente (z.B. Klassen) abgebildet werden müssen. Rollen beschreiben die Eigenschaften, welche diese UML-Modellelemente aufweisen müssen, um der Rolle zu entsprechen. Ähnlich wie die Composition Patterns werden die Rollen durch ein s*tatisches Rollenmodell* (SRM, z.B. ein UML-Klassendiagramm) und ein i*nteraktives Rollenmodell* (IRM, z.B. ein UML-Kollaborations- oder Sequenzdiagramm) beschrieben. Dieser Ansatz wird hier nicht weiter verfolgt, da er methodisch weniger gut ausgearbeitet ist als die Composition Patterns.

10.5 Erweiterung von UML und UXF

Suzuki, Yamamoto (1999) erweitern einerseits die UML und anderseits das UXF (UML eXchange Format), eine eigene, auf der XML basierende Sprache zu Beschreibung und zum Austausch von UML-Modellen zwischen verschiedenen CASE-Werkzeugen.

UML-Erweiterung. Dem UML-Metamodell wird eine *Aspekt-Metaklasse* mit dem Stereotyp `<<aspect>>` hinzugefügt, welche von der Classifier-Metaklasse erbt. Die Operationen der Aspekt-Metaklasse enthalten die Weaving-Regeln. Sie sind mit dem Stereotyp `<<weave>>` gekennzeichnet und zeigen die vom Weaving betroffenen Klassen, Methoden und Variablen, ähnlich wie die Pointcuts in AspectJ. Die Advice-Arten (before, after, around) und die Introductions von AspectJ werden als Bedingungen in geschweiften Klammern beigefügt. Die Beziehung zwischen einem Aspekt und den betroffenen Klassen wird als Abhängigkeitsbeziehung vom Typ *Abstraktion* mit dem Stereotyp `<<realize>>` dargestellt. Die Abbildung 10-3 zeigt ein Fragment des Beobachtermuster-Beispiels aus dem Abschnitt 10.2. Es beschreibt einen `Beobachter`-Aspekt mit einer `benachrichtige()`-Operation, die zur `Buch`-Klasse hinzugefügt werden soll.

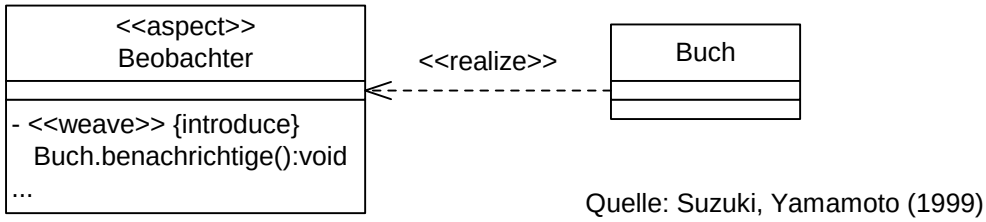


Abbildung 10-3: Aspektorientiertes Klassendiagramm, Teil des Beobachtermusters

Soll die aus dem Weaving resultierende Klasse dargestellt werden, erhält sie das Stereotyp `<<woven class>>`.

[29] www.dsg.cs.tcd.ie/index.php?category_id=355 (26.07.2004)

UXF-Erweiterung. Das UXF/a (UML eXchange Format, aspect extension) erweitert das UXF um die oben erwähnte Beschreibung von Aspekten. Zwecks Forward und Reverse Engineering existieren Konverter, z.B. zwischen Rational Rose und AspectJ. Das folgende Beispiel formuliert das Klassendiagramm aus der Abbildung 10-3 in das UXF/a um:

```
<UXF:Aspect>                                    //Definition des Aspekts
     <UXF:Name>Beobachter</UXF:Name>
     <UXF:Operation>
          <UXF:Name>Buch.benachrichtige</UXF:Name>
          <UXF:Stereotype>
               <UXF:Name>weave</UXF:Name>
          </UXF:Stereotype>
          <UXF:Constraint>
               <UXF:Body>introduce</UXF:Body>
               <UXF:ConstrainedElement>
                    Buch.benachrichtige
               </UXF:ConstrainedElement>
          </UXF:Constraint>
     </UXF:Operation>
</UXF:Aspect>
<UXF:Abstraction>              //Definition der Abstraktionsbeziehung
     <UXF:Stereotype>
          <UXF:Name>realize</UXF:Name>
     </UXF:Stereotype>
     <UXF:Supplier>Beobachter</UXF:Supplier>
     <UXF:Client>Buch</UXF:Client>
</UXF:Abstraction>
<UXF:Class>                                      //Definition der Klasse
     <UXF:Name>Buch</UXF:Name>
</UXF:Class>
</UXF>
```

Beurteilung. Der Ansatz umfasst lediglich eine Sprach-, jedoch keine Prozessdefinition. Er ist sehr AspectJ-lastig und daher einfach zu verstehen. Die Advice können – als Operationen der Aspekt-Metaklasse – definiert werden. Wie mächtig die Definitionsmöglichkeiten für Pointcuts sind, ist aus der Beschreibung des Ansatzes nicht ersichtlich. Angesichts der Nähe zu AspectJ ist fraglich, ob eine grafische Darstellung überhaupt sinnvoll ist oder ob nicht besser (Pseudo-)AspectJ verwendet werden soll, zumal sich die grafische Darstellung nicht für grössere Systeme eignet.

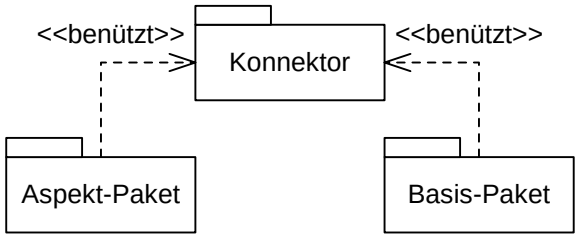
10.6 AML (Aspect Modeling Language)

Auch die AML von Groher, Baumgarth (2002) basiert auf der Standard-UML, so dass bestehende CASE-Werkzeuge benutzt werden können. Bemerkenswert ist, dass die Autoren zuerst die Anforderungen definieren, denen ein Ansatz des aspektorientierten Entwurfs ihrer Meinung nach genügen muss:

- Die Notation soll einfach zu verstehen und anzuwenden sein.
- Die Modellierung soll durch CASE-Werkzeuge unterstützt werden.
- Die Notation soll die Modellierung der gebräuchlichsten aspektorientierten Paradigmen unterstützen.
- Die Modelle sollen einfach zu verstehen sein und eine klare *Separation of Concerns* gewährleisten.
- Eine direkte Abbildung der Notation auf aspektorientierte Programmiersprachen soll eine automatische Code-Generierung erlauben.

- Die Notation soll praxistauglich und Teil eines aspektorientierten Entwicklungsprozesses sein.

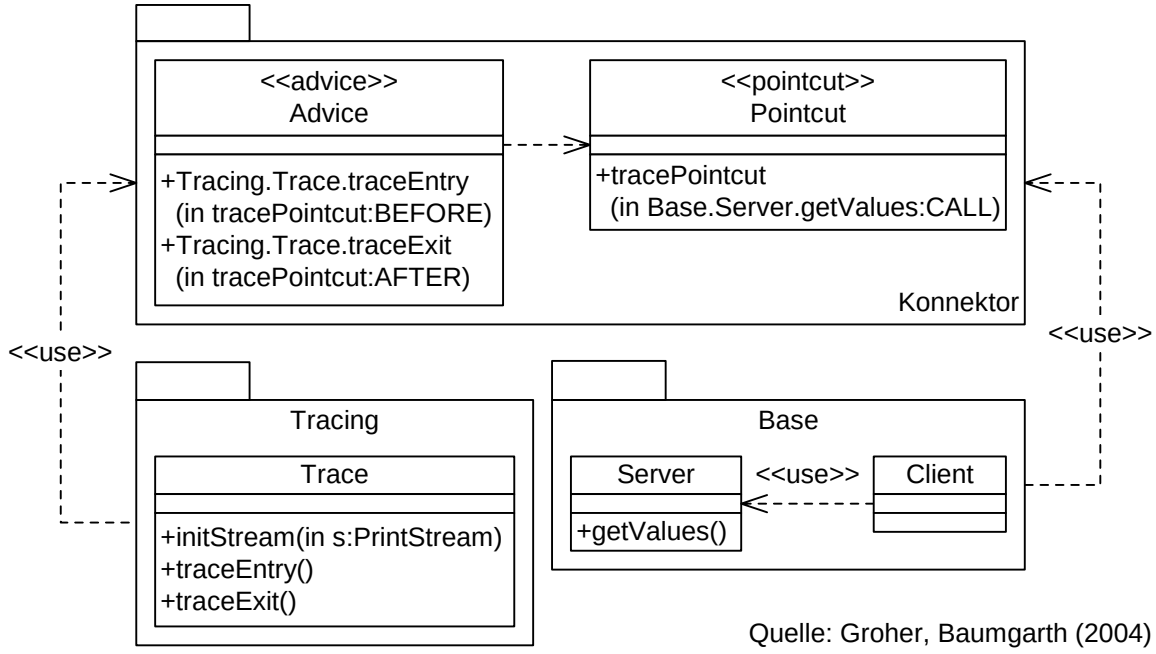
Die Modellierung von Core Concerns und Crosscutting Concerns wird strikt getrennt. Wie die Abbildung 10-4 zeigt, werden die Core Concerns in einem so genannten *Basispaket* dargestellt, die Crosscutting Concerns unabhängig davon in einem *Aspektpaket*. Die beiden Pakete werden durch einen *Konnektor* verbunden.



Quelle: Groher, Baumgarth (2004)

Abbildung 10-4: Anwendung der Pakete in der AML

Ein Tracing-Aspekt würde mit der AML im Hinblick auf eine Implementierung mit AspectJ beispielsweise wie in der Abbildung 10-5 modelliert.



Quelle: Groher, Baumgarth (2004)

Abbildung 10-5: Tracing-Aspekt in der AML

Die Core Concerns sind im `Base`-Paket dargestellt, der „crosscutting" Tracing-Concern im `Tracing`-Paket. Der `tracePointcut` besagt, dass das Tracing-Verhalten jedes Mal ausgeführt werden soll, wenn die Methode `getValues()` der `Server`-Klasse im `Base`-Paket aufgerufen (`CALL`) wird. Der Advice `traceEntry` der `Trace`-Klasse im `Tracing`-Paket wird dabei jeweils unmittelbar vor (`BEFORE`) dem Methodenaufruf durchgeführt, `traceExit` unmittelbar danach (`AFTER`). Im Original von Groher, Baumgarth (2004) sind die Pakete, Klassen und Methoden durch Dollarzeichen ($$) voneinander getrennt. Die $$ wurden hier aus Gründen der Lesbarkeit durch einen Punkt (.) ersetzt.

Eine Hyper/J-orientierte Darstellung ist nach Angaben der Autoren in Arbeit.

Beurteilung. Die Trennung von Basis- und Aspektpaketen führt zu einer saubere Separation of Concerns. Auch der Client- und der Server-Teil des Tracing-Aspekts werden in der AML sauber voneinander getrennt. Die Anwendung von Standard-UML erlaubt den Einsatz von bestehenden CASE-Werkzeugen. Die AML ist einfach und auf den ersten Blick verständlich, wenn man sich mit den Konzepten von AspectJ ein wenig auskennt. In jedem Fall ist sie einfacher zu verstehen als beispielsweise die Composition Patterns. Ausserdem lässt sich aus der AML, wie die Autoren anhand eines Beispiels demonstrieren, automatisch AspectJ-Code generieren. In diesem Sinne erfüllen die Autoren ihre eigenen Anforderungen an eine Entwurfsmethode.

Dass die Übersicht auch bei grösseren Systemen gewahrt bleibt, zeichnet die AML gegenüber allen anderen Ansätzen aus den Kapiteln 10 und 12 aus. Dies ist darauf zurückzuführen, dass die AML alle Pointcuts und Advice in einem Paket zusammenfasst, und dass die Beziehungen zwischen Komponenten und Aspekten auf Paket- und nicht auf Klassenebene grafisch dargestellt werden. Dadurch können beliebig viele Aspekte beliebig viele Komponenten quer schneiden, ohne dass die Übersicht verloren geht.

Hingegen orientiert sich der Ansatz derart stark an AspectJ, dass sich die Frage stellt, ob überhaupt eine (zusätzliche) grafische Darstellung erforderlich ist, oder ob man den gleichen Sachverhalt nicht einfacher gleich direkt mit AspectJ (unter Umständen mit leeren Advice-Rümpfen) formulieren könnte. Es geht im Entwurf nicht um eine Sprache, die auch vom Auftraggeber verstanden werden muss, sondern um eine Sprache, die sich in erster Linie an Entwickler richtet.

11. Ansätze der aspektorientierten Architektur

Ansätze der aspektorientierten Architektur sind selten, genau genommen konnte nur einer, die Stratified Frameworks, gefunden werden. Und dieser entpuppt sich bei genauerem Hinsehen eigentlich nicht als aspektorientiert.

11.1 Stratified Frameworks

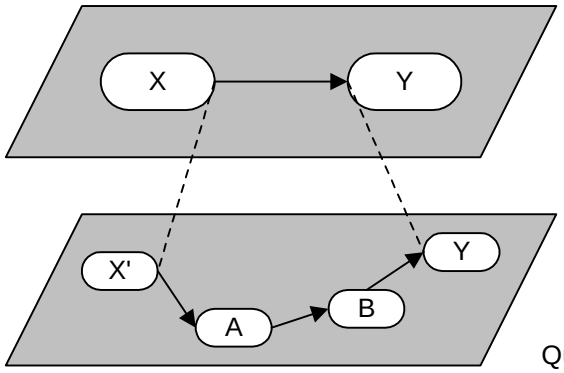
Atkinson, Kühne (2003) gehen der Frage nach, wie die Vorteile der aspektorientierter Programmierung mit anderen Ansätzen, die ebenfalls die *Separation of Concerns* zum Thema haben, kombiniert werden können. Solche Ansätze sind die komponentenbasierte Entwicklung, die Framework-basierte Entwicklung und die Model Driven Architecture (MDA)[30]. Dabei wird anstelle des Weaving eine Schichtenarchitektur verwendet, um *Crosscutting Concerns* zu behandeln.

Architekturperspektiven. Architekturen können aus verschiedenen Perspektiven betrachtet werden, z.B. strukturell, dynamisch, physisch. Die strukturelle Perspektive wird üblicherweise als die wichtigste betrachtet, da sie die Komponenten des Systems und ihre Verbindungen definiert. Ausserdem kann sie auf verschiedenen Abstraktionsebenen dargestellt werden. Häufig wird die tiefste Abstraktionsebene als die echte betrachtet, während die höheren lediglich als Zwischenergebnisse dienen.

Mit einer **Architekturschichtung** kann man aber auch systematisch Sichten erschliessen, um wichtige Concerns zu detaillieren. Die Architekturschichtung soll helfen, die verschiedenen Architekturrepräsentationen so zu organisieren, dass sie die Crosscutting Concerns eines Systems darstellen und miteinander in Beziehung setzen. Dabei werden die Abstraktionsebenen als Schichten (engl. strata) im Zug einer schrittweisen Verfeinerung (engl. stepwise refinement) betrachtet. Verfeinert,

[30] www.omg.org/mda/executive_overview.htm (22.07.2004)

d.h. in einer tieferen Schicht detailliert, werden die Verbindungen. Dies zeigt die Abbildung 11-1. Die Darstellung der Komponenten und ihrer Verbindungen kann in einer beliebigen Sprache (z.B. UML, Programmiersprache) erfolgen.



Quelle: Atkinson, Kühne (2003)

Abbildung 11-1: Verfeinerung einer Verbindung auf einer tieferen Schicht

Dabei können sich auch die Komponenten verändern. Die Client-Komponente X erhält einen neuen Kommunikationspartner, A anstelle von Y, und wird daher zu X'. Demgegenüber bleibt Y unverändert, da es als Server-Komponente nichts über seine Client-Komponente wissen muss.

Verfeinerungstransformationen. Sie halten die Beziehung zwischen einer Verbindung auf der oberen Schicht und den Komponenten und Verbindungen auf der unteren Schicht fest. Um die Art der Transformation zu beschreiben, können die Verbindungen mit standardisierten Vermerken (engl. annotation) versehen werden, z.B. {`remoteSecure`, `logged`}. Solche Vermerke können sich namentlich auf Concerns beziehen. Die Verfeinerungen erfolgen im Idealfall systematisch aufgrund einer bestimmten Transformationsart. Beispielsweise führt die Verfeinerung aller mit {`remote`} gekennzeichneten Verbindungen zu einer Kommunikationsschicht.

Schichten und Ebenen. Schichten sind keine Ebenen (engl. layer)! Eine Ebene repräsentiert einen zusammenhängenden Teil der Funktionalität eines Systems. Um zu verstehen, was das gesamte System tut, müssen alle Ebenen gleichzeitig betrachtet werden. Eine Schicht hingegen beschreibt immer das gesamte System. In einer strikten Ebenenarchitektur herrscht *Transparenz*, d.h. es ist nur möglich, Leistungen der nächst tieferen Ebene zu beziehen. (Dadurch können Ebenen leicht ausgetauscht werden.) Da dies restriktiv ist, stützen sich Schichtenarchitekturen auf *Abstraktion* anstelle von Transparenz. Diese Fähigkeit, wenn möglich von Details tieferer Schichten zu abstrahieren, aber wenn nötig auf diese Details zuzugreifen, ist wichtig für (sich gegenseitig beeinflussende) Crosscutting Concerns. Eine gute Verfeinerung stellt jeden Crosscutting Concern in seinem natürlichen Detaillierungsgrad dar. Jeder Concern soll auf der Schicht sichtbar sein, auf der er relevant ist. Ein Concern (z.B. Verteilung), der auf einer bestimmten Schicht modelliert wird, kann zum Bedarf nach einem weiteren Concern (z.B. Sicherheit) führen, der dann auf einer tieferen Schicht modelliert wird. Dadurch werden die Concerns in der Schichtenhierarchie klassifiziert. Dies ist im Übrigen ganz ähnlich wie beim NFR-Framework (siehe im Abschnitt 12.6).

Abstraktionsebenen. Aspektorientierte Ansätze operieren auf der Codeebene und können – neben Paketen, Klassen und Methoden – keine Abstraktionen darstellen, höchstens über Namenskonventionen. In einer Schichtenarchitektur kann jeder Crosscutting Concern seiner natürlichen Abstraktionsebene zugeordnet werden. Diese Fähigkeit von Schichtenarchitekturen, Informationen zu entwirren, kann auch bei der Verwendung von Mustern (Architekturstilen, Frameworks, Entwurfsmustern, Idiomen) genutzt werden. Auch Muster können ihrer natürlichen Abstraktionsebene zugeordnet werden. Diese Ähnlichkeit zwischen Aspekten und Mustern ist nicht weiter verwunderlich, eignen sich doch beide zur Implementierung von Concerns.

Frameworks. Zurzeit existieren – im Gegensatz zur aspektorientierten Programmierung – keine Werkzeuge zur Modellierung und Implementierung von Schichtenarchitekturen, was die Anwendung des Ansatzes arbeitsintensiv macht. Frameworks können Abhilfe schaffen. Durch wiederholte Instanzierung des Frameworks kann der Initialaufwand für die Separation of Concerns mittels Schichtung amortisiert werden.

Beurteilung. Die Stratified Frameworks sind ein Architekturansatz zur Dekomposition eines Systems in *einer* Dimension, nämlich einer Hierarchie. Ein Crosscutting Concern ist zwar in gewisser Weise modular darstellbar, indem die Schicht, auf der er relevant ist (seine natürliche Abstraktionsebene) eingeblendet wird. Das *Scattering-Problem* wird dabei jedoch nur scheinbar gelöst. Sobald eine andere Schicht eingeblendet wird, ist der Crosscutting Concern wieder „scattered“. Die Stratified Frameworks sind daher nicht zu den aspektorientierten Ansätzen zu zählen.

12. Ansätze der aspektorientierten Anforderungstechnik

Es herrscht Einigkeit darüber, dass die Vorteile der Aspektorientierung aus dem Abschnitt 9.11 nur dann zum Tragen kommen, wenn die Crosscutting Concerns bereits in den frühen Phasen des Software-Entwicklungsprozesses berücksichtigt werden. Ansonsten ist es für Entwickler schwierig bis unmöglich, Crosscutting Concerns in den Anforderungsdokumenten überhaupt zu entdecken. Sousa et al. (2004) sehen folgende Gründe für das Vordringen der Aspektorientierung in frühe Phasen des Software-Entwicklungsprozesses:

- Der Nutzen der Aspektorientierung erschliesst sich auch für die Anforderungs-, Analyse- und Entwurfsaktivitäten.
- Das Nachdenken über Aspekte und ihre Auswirkungen auf die Software-Entwicklung setzt früher ein.
- Ein aspektorientiertes System ist leichter verständlich, wenn nicht nur implementierte Artefakte vorliegen, sondern auch die Anforderungs-, Analyse- und Entwurfsmodelle.

Die Beurteilung der Ansätze erfolgt anhand der folgenden Kriterien:

- Umfasst der Ansatz ein Vorgehensmodell?
- Eignet sich der Ansatz zur Identifikation, Beschreibung und Darstellung von Crosscutting Concerns? Mit anderen Worten: Eignet sich der Ansatz zur Kommunikation mit dem Auftraggeber?
- Eignet sich der Ansatz als Grundlage für den aspektorientierten Entwurf? Mit anderen Worten: Eignet sich der Ansatz zur Kommunikation mit den Entwicklern?
- Berücksichtigt der Ansatz alle ansatzneutralen Konzepte (Separation, Integration) aus dem Abschnitt 9.12? Dieses Kriterium ist, um es vorwegzunehmen, bei allen Ansätzen mehr oder weniger erfüllt.
- Eignet sich der Ansatz für den Entwurf grösserer Systeme?

12.1 PREview

PREview (Process and Requirements Engineering viewpoints) ist eine *Viewpoint*-orientierte Methode für die Anforderungstechnik und Prozessanalyse und stammt von Sommerville, Sawyer (1997). PREview ist kein aspektorientierter Ansatz, bildet aber die Grundlage für einige aspektorientierte Ansätze. Aus diesem Grund wird PREview hier behandelt.

> **Definition** *Viewpoint*. Eingekapselte Information über einen Teil der Anforderungen an ein System aus einer bestimmten Perspektive.

Zu unterscheiden sind die Viewpoints der Benutzer, der übrigen Beteiligten, der Umgebung (z.B. Umsysteme) und Viewpoints aufgrund von Einschränkungen des Anwendungsgebiets (z.B. Sicherheit). Letztere sind teilweise trivial, teilweise in den Viewpoints der Beteiligten verborgen, teilweise aber auch unklar und schwierig zu eruieren.

PREview schlägt das folgende Vorgehen, den *Entdecken-analysieren-verhandeln-Zyklus*, zur Gewinnung und Dokumentation von Viewpoints und Anforderungen vor:

1. **Anforderungsgewinnung** (engl. requirements elicitation). Die Viewpoints des Systems werden identifiziert. Verschiedene Quellen (Beteiligte, Dokumente etc.) werden konsultiert, um das Problem und den Anwendungsbereich zu verstehen und die Anforderungen zu gewinnen.
2. **Anforderungsanalyse**. Die Viewpoints und Anforderungen werden analysiert, um fehlende Anforderungen, Inkonsistenzen, Konflikte und Redundanzen zu entdecken.
3. **Anforderungsverhandlung** (engl. negotiation). Inkonsistenzen werden behoben. Im Konflikt stehende Anforderungen werden nach Wichtigkeit, Machbarkeit, Kosten etc. bewertet, und die Konflikte werden gemeinsam mit allen Beteiligten aufgelöst.
4. **Anforderungsdefinition**. Die Viewpoints (Name, Perspektive, Concerns, Quellen, Liste von Anforderungen, Änderungsgeschichte) und Anforderungen (Identifikator, Beschreibung, Begründung, Änderungsgeschichte) werden tabellarisch dokumentiert.

Nicht alle Anforderungen stammen aus Viewpoints. Es gibt auch globale so genannte *externe Anforderungen*, die aus so genannten *Concerns* (z.B. Zuverlässigkeit, Sicherheit, Kosten, Pflegbarkeit) abgeleitet werden. Dabei kann es sich um Rahmenbedingungen, funktionale, Leistungs-, Schnittstellen- oder Bedienbarkeits-Anforderungen handeln. Sie schränken den Lösungsraum für die Viewpoint-orientierten Anforderungen ein und gelten als nicht verhandelbar, da sie für den Projekterfolg massgeblich sind. Sie werden wie folgt in das obige Vorgehen integriert: Zu Beginn der Anforderungsgewinnung werden die Concerns identifiziert und daraus die externen Anforderungen abgeleitet. Alle Viewpoint-orientierten Anforderungen werden mittels so genannter *Concern-Fragen* grob auf ihre Konsistenz mit den externen Anforderungen geprüft. Während der Anforderungsanalyse wird die Einhaltung der externen Anforderungen systematisch geprüft. Die Notwendigkeit, die Concerns zu integrieren, führt schliesslich zum Vorgehensmodell von PREview, wie es die Abbildung 12-1 zeigt:

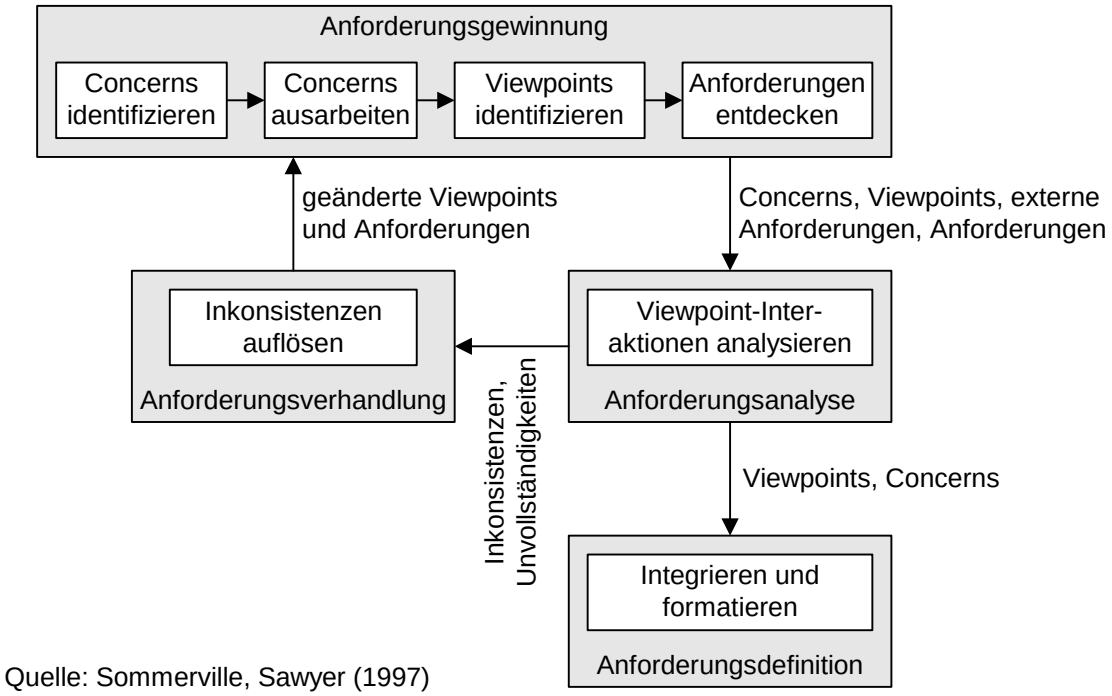


Abbildung 12-1: Vorgehensmodell von PREview

PREview hat die folgenden Vorteile:

- PREview verbessert die Vollständigkeit der Anforderungen. Dank der verschiedenen Viewpoints werden Anforderungen weniger leicht übersehen.
- Die Separation of Concerns ermöglicht eine von anderen Viewpoints isolierte Entwicklung von Teilspezifikationen.
- Die (Rück-)Verfolgbarkeit von Anforderungen zu ihren Quellen ist gewährleistet.
- PREview dient vor allem der Gewinnung und Priorisierung von Anforderungen und kann andere Methoden der Anforderungstechnik ergänzen.

Beurteilung. PREview ist die Urform der aspektorientierten Anforderungstechnik. Zwar erfolgt keine explizite Integration von Viewpoint-orientierten und externen Anforderungen, aber immerhin werden sie einander zwecks Konsistenzprüfung in einer Matrix gegenüber gestellt. PREview eignet sich zur Identifikation von Crosscutting Concerns und zur Kommunikation mit dem Auftraggeber, jedoch weder zur Darstellung und Beschreibung von Crosscutting Concerns noch als Grundlage für den aspektorientierten Entwurf. PREview kann durchaus auch in grösseren Systemen angewendet werden. Zur Begriffswahl: Die Viewpoint-orientierten Anforderungen entsprechen den Core Concerns aus dem Kapitel 5, die externen Anforderungen den Crosscutting Concerns.

12.2 AORE

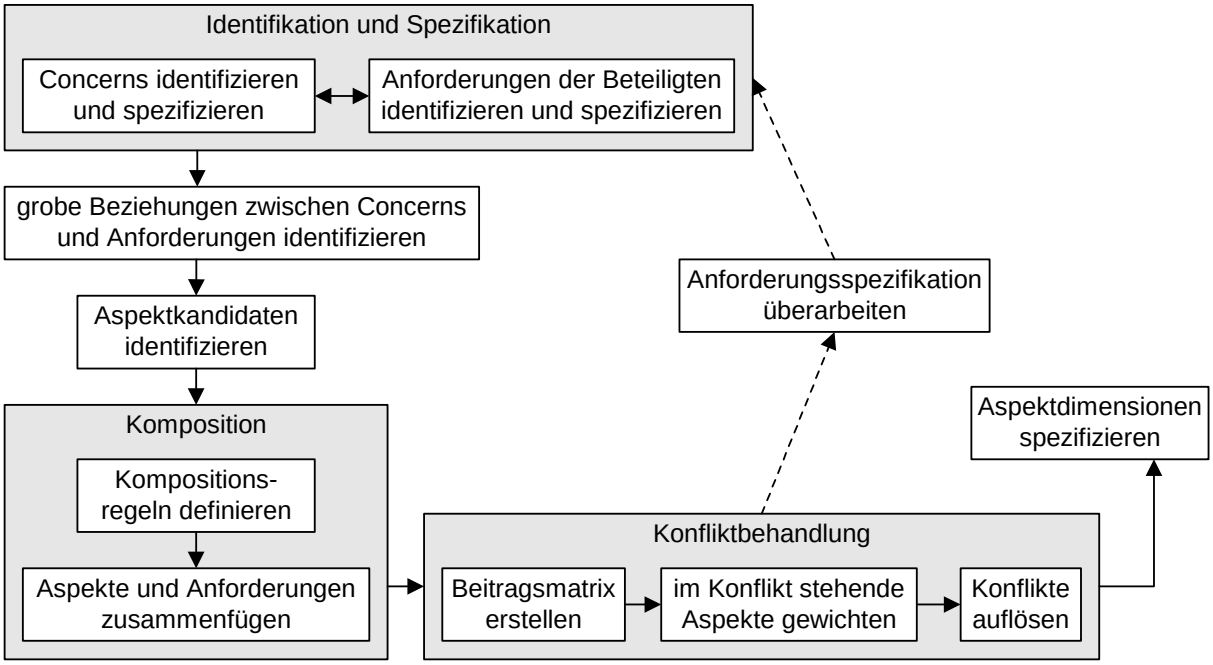
Der in Rashid et al. (2002) grob beschriebene und in Rashid, Moreira, Araujo (2003) verfeinerte Ansatz nimmt für sich in Anspruch, ein allgemeines Modell für die aspektorientierte Anforderungstechnik (engl. aspect-oriented requirements engineering, AORE) zu sein.

In der Forschung herrscht keine Einigkeit darüber, welche Rolle Aspekte in den frühen Phasen des Software-Entwicklungsprozesses spielen und wie sie auf Artefakte späterer Phasen abzubilden sind. Crosscutting Concerns wie Kompatibilität, Verfügbarkeit oder Sicherheit können nicht in einem

einzigen Viewpoint oder Anwendungsfall gekapselt werden. Daraus entsteht das Bedürfnis, Aspekte als fundamentale Modellierungselemente der Anforderungsphase zu betrachten, um

- die Trennung von funktionalen und nicht-funktionalen „crosscutting“ Eigenschaften in der Anforderungsphase zu unterstützen und
- die Abbildung und den Einfluss von Aspekten auf Artefakte späterer Entwicklungsphasen zu identifizieren.

Das Vorgehensmodell von AORE präsentiert sich wie in der Abbildung 12-2 dargestellt:



Quelle: Rashid, Moreira, Araujo (2003)

Abbildung 12-2: Vorgehensmodell von AORE

Identifikation und Spezifikation. In einem ersten Schritt werden die Concerns und die Anforderungen der Beteiligten identifiziert und spezifiziert. Für die Anforderungen der Beteiligten wenden Rashid, Moreira, Araujo (2003) die *Viewpoints* aus dem Abschnitt 12.1 an; es können aber auch andere Mechanismen wie z.B. *Anwendungsfälle*, *Ziele* (siehe van Lamsweerde, 2001) oder *Problem Frames* (siehe Jackson, 2000) eingesetzt werden. Die Viewpoints und Concerns werden in Rashid et al. (2002) tabellarisch, in Rashid, Moreira, Araujo (2003) mit der XML dargestellt. Das folgende XML-Beispiel stammt aus einer Fallstudie zur automatischen Gebührenbelastung an einer Mautstelle. Zunächst werden zwei Viewpoints (Vehicle und Gizmo[31]) sowie ein Concern (Response Time) beschrieben:

```
<Viewpoint name="Vehicle">
     <Requirement id="1">
          The vehicle enters the system when it is within ten meters
          of the toll gate.
     </Requirement>
     <Requirement id="2">
          The vehicle enters the toll gate.
     </Requirement> …
</Viewpoint>
```

[31] Damit ist ein kleines Gerät gemeint, das in allen Fahrzeugen installiert ist.

```
<Viewpoint name="Gizmo">
     <Requirement id="1">
          The gizmo identifier is read by the system.
          <Requirement id="1.1">
               The gizmo identifier is validated by the system.
          </Requirement>
          <Requirement id="1.2">
               The gizmo is checked by the system for being active or
               not.
          </Requirement>
          …
     </Requirement>
     …
</Viewpoint>

<Concern name="ResponseTime">
     <Requirement id="1">
          The system needs to react in-time in order to:
          <Requirement id="1.1">
               read the gizmo identifier;
          </Requirement>
          <Requirement id="1.2">
               turn on the light (to green or yellow);
          </Requirement>
          <Requirement id="1.3">
               display the amount to be paid;
          </Requirement>
          …
     </Requirement>
     …
</Concern>
```

Anschliessend werden die gewonnenen Concerns und Anforderungen in einer Matrix einander gegenübergestellt. Daraus ist ersichtlich, welche Concerns welche Anforderungen quer schneiden. Concerns, die mehrere Anforderungen quer schneiden, werden als *Aspektkandidaten* (engl. candidate aspects) bezeichnet. Da nicht jeder Aspektkandidat später auch tatsächlich auf einen Aspekt abgebildet wird, werden die beiden Begriffe unterschieden.

Komposition. Sobald die Aspektkandidaten identifiziert sind, werden in einem nächsten Schritt detaillierte *Kompositionsregeln* (engl. composition rules) definiert, und zwar in der Granularität einer einzelnen Anforderung. Dadurch ist es möglich zu spezifizieren, wie eine aspektbezogene Anforderung das Verhalten einer Menge von nicht-aspektbezogenen Anforderungen beeinflusst bzw. einschränkt. Auch die Komposition wird mit der XML dargestellt.

```
<Composition>
     <Requirement aspect="ResponseTime" id="1.1">
          <Constraint action="enforce" operator="between">
               <Requirement viewpoint="Vehicle" id="1"/>
               <Requirement viewpoint="Vehicle" id="2"/>
          </Constraint>
          <Outcome action="satisfied">
               <Requirement viewpoint="Gizmo" id="1" children="include"/>
          </Outcome>
     </Requirement>
     …
</Concern>
```

`<Constraint>` beschreibt mittels Aktion und Operator, auf welche Weise eine aspektbezogene Anforderung eine Viewpoint-Anforderung einschränkt. In Rashid, Moreira, Araujo (2003) sind einige Aktionen (enforce, ensure, provide, applied und exclude) und Operatoren (during, between, on, for, with, in, XOR) vordefiniert. `<Outcome>` definiert das Resultat der Einschränkung. Auch dafür sind einige Operatoren (satisfied, fulfilled) vordefiniert. Aktionen und Operatoren sind bewusst informell gehalten, damit die Beschreibung für alle Beteiligten verständlich ist. So lautet z.B. die obige Komposition im Klartext: „`ResponseTime 1.1` requirement must be `enforced between Vehicle 1` and `Vehicle 2` requirements with the `outcome` that `Gizmo 1` requirement `including` its `children` is `satisfied`".

Konfliktbehandlung. Schliesslich werden Konflikte unter den Aspektkandidaten identifiziert und aufgelöst. Zu diesem Zweck wird eine *Wechselwirkungsmatrix* (engl. contribution matrix) erstellt, aus der ersichtlich ist, welcher Aspektkandidat zu welchen anderen Aspektkandidaten einen positiven (+) oder negativen (–) Beitrag leistet. Die Aspektkandidaten mit einem negativen Beitrag werden anschliessend mit einer Zahl zwischen 0 und 1 gewichtet, welche die Priorität des Aspektkandidaten darstellt. Diese Priorisierung hilft dabei, die Konflikte anschliessend gemeinsam mit den Beteiligten aufzulösen. Unter Umständen muss dabei die Anforderungsspezifikation (d.h. die Anforderungen der Beteiligten, die aspektbezogenen Anforderungen oder die Kompositionsregeln) überarbeitet werden.

Identifikation der Aspektdimensionen. Als letzte Aktivität werden die so genannten *Dimensionen* jedes Aspektkandidaten identifiziert und tabellarisch dargestellt.

- **Abbildung** (engl. mapping). Kann der Aspektkandidat auf eine Funktion (z.B. Methode), eine Entscheidung (z.B. Architekturentscheidung) oder einen Aspekt (z.B. Antwortzeit) abgebildet werden?
- **Einfluss**. Welchen Einfluss hat der Aspektkandidat auf die späteren Phasen des Entwicklungsprozesses (z.B. Architektur, Entwurf)?

Werkzeugunterstützung. Zur Unterstützung der Spezifikationsaktivitäten existiert ein Werkzeug ARCaDe (Aspectual Requirements Composition and Decision support tool), welches aber nicht näher beschrieben ist.

Beurteilung. AORE ist ein guter erster Schritt zur Unterstützung der aspektorientierten Anforderungstechnik. Insbesondere das Vorgehensmodell vermag zu überzeugen. AORE eignet sich vor allem für die Identifikation und Beschreibung von Crosscutting Concerns, aufgrund der Wahl von XML weniger für die Darstellung und die Kommunikation mit dem Auftraggeber. Teilformale Darstellungen fehlen ganz. Die Anwendung von AORE für grössere Systeme dürfte zu unübersichtlichen Resultaten führen, die nicht einfach zu validieren sind. Die erwähnten Aspektdimensionen (Abbildung und Einfluss) werden leider nicht näher ausgeführt. Damit bleibt die Frage unbeantwortet, wie sich die Autoren Entwurf und Implementierung der Aspektkandidaten vorstellen. Zur Begriffswahl: Die Concerns entsprechen den Crosscutting Concerns aus dem Kapitel 5, die Anforderungen und Viewpoints den Core Concerns.

12.3 Vision

Vision von Araujo, Coutinho (2003) ist eine weitere Viewpoint-orientierte Anforderungsmethode, die unter anderem auf PREview und der UML basiert. Vision wird basierend auf AORE um aspektorientierte Konzepte erweitert. Funktionale Anforderungen werden als Anwendungsfälle modelliert, „crosscutting" Anforderungen als so genannte *aspektbezogene Anwendungsfälle*.

Das Vorgehensmodell von Vision präsentiert sich wie in der Abbildung 12-3 dargestellt:

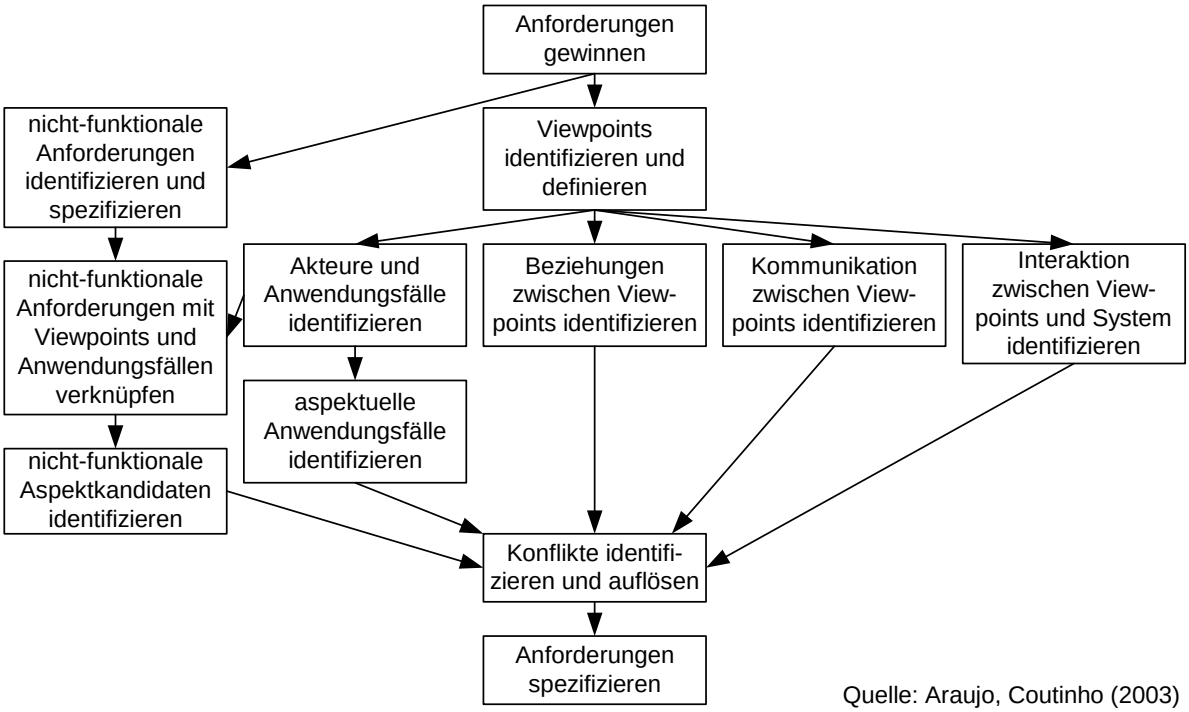


Abbildung 12-3: Vorgehensmodell von Vision

Anforderungen gewinnen. Durch Interviews, Fragebögen, Handbücher und andere Dokumente werden die essentiellen Anforderungen gewonnen. Ergebnis ist ein Dokument mit allen Anforderungen.

Nicht-funktionale Anforderungen identifizieren und spezifizieren. Die gewonnenen Anforderungen werden analysiert. Daraus werden die nicht-funktionalen Anforderungen extrahiert und tabellarisch (Name, Beschreibung, Einfluss auf Viewpoints und Anwendungsfälle, Wechselwirkungen, Priorität) beschrieben.

Viewpoints identifizieren und definieren. Auch die relevanten Viewpoints werden aus den gewonnenen Anforderungen extrahiert und tabellarisch (Name, Fokus, Quellen, Akteure, nicht-funktionale Anforderungen, Anwendungsfälle, Aggregationen, Beziehungen, Spezialisierungen, Änderungsgeschichte) beschrieben. Ein Viewpoint kann auch als UML-Klasse mit dem Stereotyp `<<Viewpoint>>` dargestellt werden.

Akteure und Anwendungsfälle identifizieren. Aufgrund der gewonnenen Anforderungen werden Akteure und Anwendungsfälle identifiziert und tabellarisch (Name, Beschreibung, Akteure, Viewpoints, Szenario, erweitert, schliesst ein, nicht-funktionale Anforderungen) spezifiziert.

Nicht-funktionale Anforderungen mit Viewpoints und Anwendungsfällen verknüpfen. Dabei wird festgestellt, welche Viewpoints und Anwendungsfälle durch welche nicht-funktionalen Anforderungen beeinflusst werden bzw. welche nicht-funktionalen Anforderungen welche Viewpoints und Anwendungsfälle betreffen. Concerns, die mehr als einen Viewpoint betreffen, werden in Anlehnung an AORE *nicht-funktionale Aspektkandidaten* genannt, um sie von den funktionalen *aspektbezogenen Anwendungsfällen* zu unterscheiden.

Aspektbezogene Anwendungsfälle (engl. aspectual use cases) **identifizieren**. Anhand der spezifizierten Anwendungsfälle wird geprüft, ob ein Anwendungsfall existiert, der von mehr als einem Anwendungsfall eingeschlossen (engl. include) wird, bzw. ob ein Anwendungsfall existiert, der

mehr als einen Anwendungsfall erweitert (engl. extend). Ist das der Fall, handelt es sich um einen *aspektbezogenen Anwendungsfall*, da er mehrere Anwendungsfälle quer schneidet. Er kann als Aspekt entworfen und implementiert werden.

Beziehungen zwischen Viewpoints identifizieren. Es wird festgestellt, ob Aggregations-, Assoziations- oder Generalisierungs-/Spezialisierungs-Beziehungen zwischen Viewpoints existieren. Diese Beziehungen werden in einem so genannten *Viewpoint-Diagramm* (einem Klassendiagramm ähnlich) dargestellt.

Kommunikation zwischen Viewpoints und Interaktionen zwischen Viewpoints und System definieren. Die Kommunikation wird durch Szenarien beschrieben, die Interaktion mittels UML-Sequenzdiagrammen.

Konflikte identifizieren und auflösen. Diese Aktivität wird wie in AORE durchgeführt.

Anforderungen spezifizieren. Diese Aktivität ist nicht Teil des Vorgehensmodells von Vision.

Beurteilung. Vision ist ähnlich wie AORE: Die Vorgehensmodelle sind verwandt, teilformale Darstellungen fehlen, und es ist unklar, wie die Anforderungsmodelle auf Entwurfsmodelle abgebildet werden können. Im Vergleich zu AORE ist Vision eher verwirrend: Zwar wird zwischen nicht-funktionalen Aspektkandidaten (Beispiele: Sicherheit und Performance) und aspektbezogenen Anwendungsfällen (Beispiel: Checks drucken) differenziert, der Unterschied wird aber nirgends erklärt. Im Unterschied zu AORE sind fast alle Beschreibungen tabellarisch, was immerhin die Kommunikation mit dem Auftraggeber etwas erleichtern dürfte und die Eignung für grössere Systeme erhöht. Hingegen droht der Überblick über all diese Tabellen verloren zu gehen. Zur Begriffswahl: Die Viewpoints entsprechen den Core Concerns aus dem Kapitel 5, die nicht-funktionalen Aspektkandidaten und die aspektbezogenen Anwendungsfälle entsprechen den Crosscutting Concerns. Es ist zu vermuten, dass sie sich analog zu den Punkten a) und b) aus dem Abschnitt 8.2 unterscheiden.

12.4 Aspektorientierte Anforderungen mit der UML

Einige Ansätze beschäftigen sich mit der Frage, wie Aspekte in die UML[32] integriert werden können. UML steht für Unified Modeling Language und ist eine durch die OMG (Object Management Group) standardisierte teilformale Sprache zur Spezifikation, Visualisierung und Dokumentation von Modellen für Software-Systeme.

Araujo et al. (2002) gehen der Frage nach, wie mit „crosscutting“ nicht-funktionalen Anforderungen in der Anforderungsphase umgegangen werden soll. Der Ansatz basiert auf AORE, wobei die Viewpoint-orientierten Beschreibungen durch UML-Diagramme ersetzt werden. Das Vorgehensmodell unterscheidet drei Hauptphasen:

Modellieren der funktionalen Concerns. Die funktionalen Anforderungen werden konventionell mit UML-Diagrammen, hauptsächlich Anwendungsfall- und -Sequenzdiagrammen, modelliert.

Modellieren der Crosscutting Concerns. Nach der Gewinnung der nicht-funktionalen Anforderungen wird identifiziert, welche davon „crosscutting“, d.h. Aspektkandidaten sind. Eine nicht-funktionale Anforderung ist ein Crosscutting Concern, wenn sie mehr als einen Anwendungsfall traversiert/betrifft. Die Beschreibung erfolgt tabellarisch (Name, Beschreibung, Priorität, Liste von Anforderungen, Liste von Modellen).

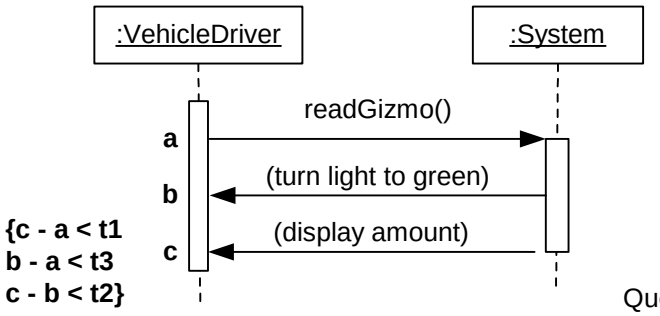
[32] www.uml.org (22.07.2004)

Komposition der Anforderungen. Die funktionalen Anforderungen werden mit den Aspekten (so werden die Aspektkandidaten der Einfachheit halber häufig genannt) verknüpft, dann werden Konflikte identifiziert und aufgelöst. Für die Komposition gibt es drei Möglichkeiten:

- **Überlappen**. Die Aspektanforderungen erweitern die funktionalen Anforderungen, die sie traversieren/betreffen.
- **Überlagern** (engl. override). Die Aspektanforderungen ersetzen (engl. superpose) die funktionalen Anforderungen, die sie traversieren/betreffen.
- **Umhüllen** (engl. wrap): Die Aspektanforderungen kapseln/wickeln die funktionalen Anforderungen ein, die sie traversieren/betreffen.

Zur Darstellung der Komposition wird die UML verwendet: (Als Beispiel wird das gleiche Gebührenbelastungssystem verwendet wie bei AORE.)

- **Anwendungsfalldiagramm**. Die Anwendungsfalldiagramm-Notation wird um zwei Stereotypen erweitert: Die Crosscutting Concerns werden als Stereotypen von Anwendungsfällen dargestellt (z.B. `<<ResponseTime>>`). Die Kompositionen werden als Stereotypen von Abhängigkeitsbeziehungen dargestellt (z.B. `<<wrappedBy>>` für eine umhüllende Komposition). Die Pfeilrichtung geht von den Anwendungsfällen zu den Crosscutting Concerns.
- **Sequenzdiagramm**. Die Sequenzdiagramm-Notation wird wie in der Abbildung 12-4 gezeigt erweitert: Die relevanten Ereignisse (das Versenden und Empfangen von Nachrichten) werden durch Kleinbuchstaben (hier fett gedruckt) markiert. Die durch Crosscutting Concerns (hier im Beispiel: Antwortzeit) auferlegten Bedingungen werden mittels zeitlicher Ausdrücke in geschweiften Klammern festgehalten. Sie enthalten in diesem Beispiel die relative Zeit, die höchstens zwischen zwei Ereignissen liegen darf.



Quelle: Araujo et al. (2002)

Abbildung 12-4: Sequenzdiagramm mit Crosscutting Concern

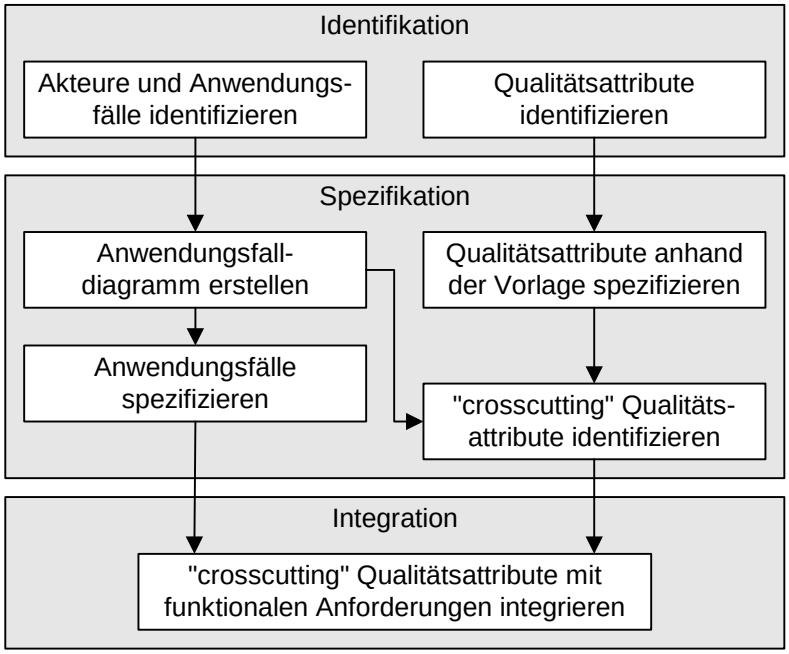
Bestimmungsfaktoren für die Komposition sind *Vollständigkeit* (engl. completeness), d.h. in jedem Aspekt sind alle benötigten Anforderungen aufgeführt, und *Hinlänglichkeit* (engl. sufficiency), d.h. jede Anforderung in einem Aspekt hat eine Auswirkung auf die Komposition.

Beurteilung. Der Fokus dieses Ansatzes liegt nicht auf der Identifikation und Beschreibung von Crosscutting Concerns, sondern auf ihrer Darstellung. Da die UML-Erweiterungen nur anhand eines Beispiels gezeigt werden, fehlt ihnen die Allgemeingültigkeit. Die Erweiterung des Anwendungsfalldiagramms ist nahe liegend, aber nur von beschränktem praktischem Nutzen. Im Fall von Crosscutting Concerns, die viele Anwendungsfälle quer schneiden, wird die Darstellung rasch unübersichtlich, zumal sich Anwendungsfalldiagramme generell eher für kleinere Systeme eignen. Die Erweiterung des Sequenzdiagramms ist interessant, da sie sich einfach auf Konzepte von AspectJ abbilden lässt: die Markierungen der Ereignisse entsprechen den Join Points von AspectJ, die Ausdrücke in geschweiften Klammern lassen sich auf Advice abbilden. Die Konzepte für das Überlappen, Überlagern und Umhüllen werden leider nicht weiter ausgeführt. Sie weisen eindeutig eine gewisse Ähnlichkeit zu den Advice-Arten (before, after, around) auf, lassen sich aber nicht 1:1 darauf abbilden. Insgesamt eignet sich diese Erweiterung des Sequenzdiagramms für die Darstel-

lung von Crosscutting Concerns in der Anforderungsphase und zur entsprechenden Kommunikation mit dem Auftraggeber. Zur Begriffswahl: Crosscutting Concerns, „crosscutting“ nicht-funktionale Anforderungen und Aspektkandidaten werden als Synonyme verwendet.

12.5 Anforderungsmodell für Qualitätsattribute

Das durch Brito, Moreira, Araujo (2002) und Moreira, Araujo, Brito (2002) beschriebene Anforderungsmodell für Qualitätsattribute lehnt sich stark an den im Abschnitt 12.4 beschriebenen Ansatz von Araujo et al. (2002) an, was den Einsatz von UML betrifft. Es verwendet jedoch andere Begriffe (*Qualitätsattribute* anstelle von *Concerns*) und ein etwas anderes Vorgehensmodell, wie die Abbildung 12-5 zeigt.



Quelle: Brito, Moreira, Araujo (2002)

Abbildung 12-5: Vorgehensmodell des Anforderungsmodells für Qualitätsattribute

Identifikation. Zuerst werden alle Anforderungen an das System identifiziert. Diese Anforderungen können in funktionale und nicht-funktionale Anforderungen aufgeteilt werden: *Funktionale Anforderungen* sind Aussagen darüber, welche Leistungen ein System zur Verfügung stellen soll, wie das System auf bestimmte Eingaben reagieren soll und wie sich das System in bestimmten Situationen verhalten soll. Demgegenüber sind *nicht-funktionale Anforderungen* Eigenschaften, die das System als Ganzes betreffen, wie z.B. Antwortzeit, Korrektheit, Sicherheit und Zuverlässigkeit. Der Begriff *Qualitätsattribut* wird in diesem Ansatz als Synonym für den Begriff *nicht-funktionale Anforderung* verwendet.

Spezifikation. Die funktionalen Anforderungen werden anwendungsfallorientiert modelliert und mittels UML-Anwendungsfall-, -Sequenz- und -Klassendiagrammen sowie Szenarien spezifiziert. Die Qualitätsattribute werden anhand der in der Tabelle 12-1 dargestellten Vorlage tabellarisch beschrieben. Aufgrund dieser Tabelle können nun die „crosscutting“ Qualitätsattribute identifiziert werden: Die Information, ob ein Qualitätsattribut mehr als einen Anwendungsfall quer schneidet, kann der Zeile mit der Überschrift „Wo“ entnommen werden. Schneidet ein Qualitätsattribut mehr als eine Anforderung quer, ist dies in der Zeile mit der Überschrift „Anforderungen“ ersichtlich.

Name	Name des Qualitätsattributs
Beschreibung	Beschreibung des Qualitätsattributs
Fokus	System (d.h. Endprodukt) oder Entwicklungsprozess
Quelle	Informationsquellen für dieses Qualitätsattribut (z.B. Beteiligte, Dokumente)
Dekomposition	Dekomposition der Qualitätsattribute in Sub-Qualitätsattribute mit AND-Beziehungen, falls alle Sub-Qualitätsattribute erfüllt sein müssen, bzw. mit OR-Beziehungen, falls nicht alle Sub-Qualitätsattribute erfüllt sein müssen.
Priorität	Wichtigkeit des Qualitätsattributs für die Beteiligten: MAX, HIGH, LOW, MIN
Verpflichtung	Optional oder obligatorisch
Wo	Liste von Akteuren, welche durch das Qualitätsattribut beeinflusst werden, und Liste von Modellen (z.B. Anwendungsfälle und Sequenzdiagramme), für die das Qualitätsattribut erforderlich ist
Anforderungen	Liste von Anforderungen, für die das Qualitätsattribut gilt
Wechselwirkung	Wechselwirkung zu anderen Qualitätsattributen: positiv (+) oder negativ (-)

Tabelle 12-1: Vorlage für die Spezifikation von Qualitätsattributen

Integration. Die Integration verknüpft die Qualitätsattribute mit den funktionalen Anforderungen zum Gesamtsystem. Zwecks Darstellung der Integration wird das Anwendungsfalldiagramm wie bei Araujo et al. (2002) um ein Anwendungsfall-Stereotyp erweitert, mit dem die Qualitätsattribute dargestellt werden können (z.B. `<<ResponseTime>>`). Die Verknüpfung wird im Gegensatz zu Araujo et al. (2002) mittels `<<include>>`-Abhängigkeitsbeziehung von den Anwendungsfällen zum Qualitätsattribut-Stereotyp dargestellt. Die Konzepte für das Überlappen, Überlagern und Umhüllen werden aus Araujo et al. (2002) übernommen, wobei das Überlappen nun *vorher* oder *nachher* erfolgen kann. Um eine Überlappung auszudrücken, wird das Anwendungsfalldiagramm um ein abenteuerliches Konstrukt mit `<<before>>`- und `<<after>>`-Stereotypen erweitert.

Beurteilung. Der Ansatz ist bis auf einige Details ähnlich wie derjenige von Araujo et al. (2002). Daher gelten die im Abschnitt 12.4 gemachten Bemerkungen auch hier. Wie im Abschnitt 13.4 erläutert, ist die Verwendung von `<<include>>` für die Integration der Qualitätsattribute ins UML-Anwendungsfalldiagramm nicht korrekt, da die Anwendungsfälle nichts von den Qualitätsattributen wissen sollten. Zur Begriffswahl: Die Qualitätsattribute entsprechen den Concerns aus dem Kapitel 5. Demzufolge entsprechen die „crosscutting“ Qualitätsattribute den Crosscutting Concerns.

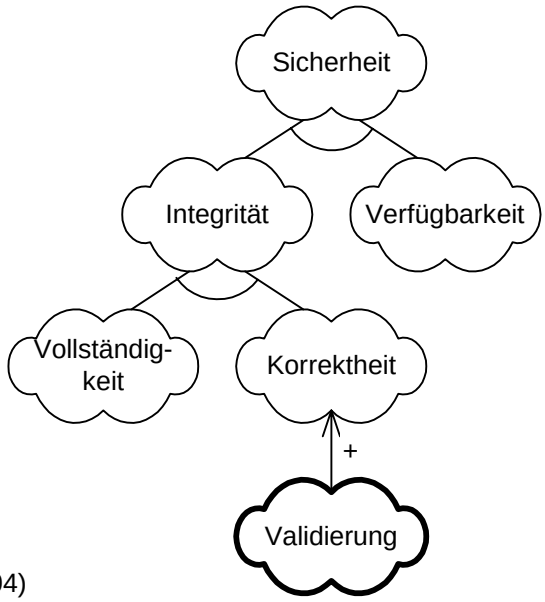
12.6 Modellierung von Crosscutting Concerns mit dem NFR-Framework

In diesem Abschnitt wird zuerst das NFR-Framework kurz beschrieben, und dann werden zwei Ansätze vorgestellt, die auf dem NFR-Framework basieren.

12.6.1 Das NFR-Framework

Das *NFR-Framework* wurde durch Chung et al. (2000) beschrieben und ist nach Sousa et al. (2004) ein systematischer Ansatz zur Behandlung von nicht-funktionalen Anforderungen (engl. non-functional requirements, NFR) in Software-Entwicklungsprojekten. Dabei werden die nicht-funktionalen Anforderungen (z.B. Sicherheit, Performance) als Ziele betrachtet und, weil sie oft schwer fassbar sind, als *Softgoals* bezeichnet. Das NFR-Framework basiert auf einem grafischen Darstellungsmittel, dem *Softgoal Interdependency Graph (SIG)*, dessen Anwendung durch so genannte *Wissenskataloge* unterstützt wird.

Softgoal Interdependency Graph (SIG). Jedes Softgoal wird so lange in immer spezifischere Softgoals hierarchisch zerlegt, bis es operationalisiert, d.h. durch konkrete und präzise Mechanismen (z.B. Operationen, Geschäftsregeln, Entwurfsentscheidungen) beschrieben werden kann. Alle *Operationalisierungen* zusammen erfüllen das Softgoal an der Spitze der Hierarchie, d.h. die ursprüngliche nicht-funktionale Anforderung. Im Laufe dieser Dekomposition werden Wechselwirkungen zwischen den Softgoals erkannt und allfällige Konflikte durch Priorisierung aufgelöst (Sousa et al., 2004). Der *Softgoal Interdependency Graph (SIG)* stellt die Softgoals, ihre Beziehungen und Operationalisierungen grafisch dar. Die Abbildung 12-6 zeigt ein einfaches Beispiel eines SIG aus Brito, Moreira (2004).



Quelle: Brito, Moreira (2004)

Abbildung 12-6: Beispiel für einen Softgoal Interdependency Graph (SIG)

Die Knoten repräsentieren die Softgoals und werden als Wolken dargestellt, während die Kanten die Dekompositionsbeziehungen zwischen über- und untergeordneten Softgoals zeigen. Es gibt zwei Arten von Dekompositionsbeziehungen:

- Eine AND-Beziehung wird durch einen einfachen Bogen zwischen den Kanten dargestellt und zeigt an, dass sämtliche untergeordneten Softgoals (z.B. Vollständigkeit und Korrektheit) erfüllt sein müssen, damit das übergeordnete Softgoal (z.B. Integrität) erfüllt ist.
- Eine OR-Beziehung wird durch einen Doppelbogen zwischen den Kanten dargestellt und zeigt an, dass mindestens ein untergeordnetes Softgoal erfüllt sein muss, damit das übergeordnete Softgoal erfüllt ist.

Die Operationalisierungen (z.B. Validierung) werden durch eine dick umrandete Wolke und einen Pfeil mit + („unterstützt“) oder ++ („erfüllt“) dargestellt, der zum operationalisierten Softgoal (z.B. Korrektheit) zeigt. Daneben gibt es unzählige weitere Mittel zur Darstellung und Beschreibung von Softgoals, ihren Beziehungen und Wechselwirkungen.

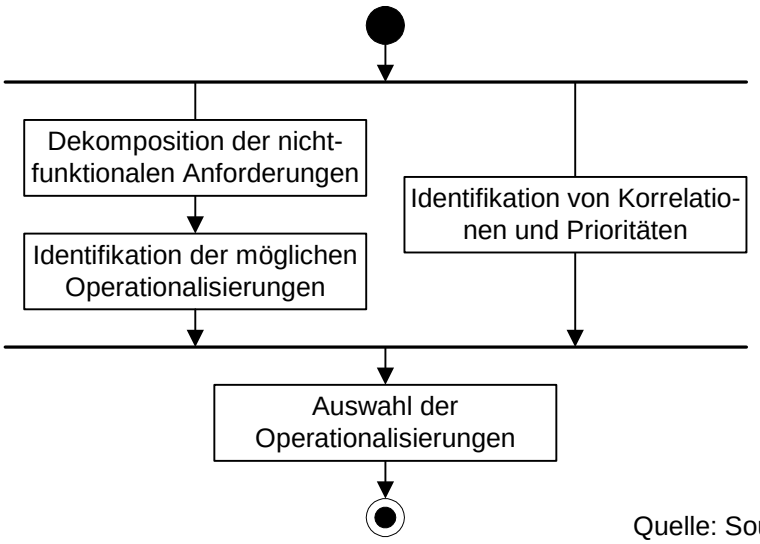
Wissenskataloge. Die Anwendung des NFR-Framework wird durch so genannte *Wissenskataloge* (engl. knowledge catalogues) unterstützt, in denen das verfügbare Wissen über nicht-funktionale Anforderungen in der Anforderungs- und Entwurfsphase gesammelt wird. Chung et al. (2000) erwähnen drei Wissenskataloge:

- Der **Typenkatalog** führt eine einheitliche Terminologie und Kategorisierung von nicht-funktionalen Anforderungen ein und kann in späteren Entwicklungsprojekten als *Checkliste* für die Identifizierung von nicht-funktionalen Anforderungen benutzt werden.

- Der **Methodenkatalog** umfasst Verfahren zur Dekomposition und Operationalisierung von nicht-funktionalen Anforderungen.
- Der **Korrelationskatalog** zeigt die Wechselwirkungen zwischen nicht-funktionalen Anforderungen in Form einer Matrix auf.

Die Wissenskataloge sollen laufend um neues Wissen erweitert werden. Neues Wissen kann aus Dokumentationen, Handbüchern, von Spezialisten aus der Wirtschaft und der Wissenschaft sowie von Projektmitarbeitern aus der eigenen Organisation stammen.

Vorgehen. Das Vorgehensmodell des NFR-Framework definiert, wie ein Software-Entwicklungsprojekt mit nicht-funktionalen Anforderungen umgehen soll. Zuerst muss nach Chung et al. (2000) Wissen über den Anwendungsbereich gesammelt werden. Dazu gehören funktionale Anforderungen und Prioritäten. Dann wird anhand der Wissenskataloge grundsätzliches Wissen über die Terminologie, Kategorisierung und Operationalisierung von nicht-funktionalen Anforderungen erworben. Schliesslich werden die nicht-funktionalen Anforderungen mit Hilfe der Wissenskataloge identifiziert, operationalisiert und in Form von Softgoal Interdependency Graphs (SIG) dargestellt. In der Abbildung 12-7 sind die Hauptaktivitäten zur Operationalisierung von nicht-funktionalen Anforderungen aufgeführt.



Quelle: Sousa et al. (2004)

Abbildung 12-7: Hauptaktivitäten des NFR-Framework

Beurteilung. Das NFR-Framework ist kein eigenständiger Ansatz der aspektorientierten Anforderungstechnik, da es keine Konzepte zur Integration umfasst, es ist jedoch eine sinnvolle Ergänzung für die Identifikation und Beschreibung von Crosscutting Concerns. Der Softgoal Interdependency Graph (SIG) zur Operationalisierung der nicht-funktionalen Anforderungen ist interessant, weil aus der Operationalisierung schliesslich funktionale Anforderungen oder Entwurfsentscheidungen resultieren. Die Idee mit den Wissenskatalogen ist bestechend. Gerade weil in allen Systemen immer wieder ähnliche nicht-funktionale Anforderungen vorkommen, können die Wissenskataloge als Checklisten benutzt werden: die Typenkataloge zur Identifikation nicht-funktionaler Anforderungen, die Methoden- und Korrelationskataloge zur Unterstützung der Operationalisierung. Dadurch tragen sie zur Steigerung der Effizienz und Effektivität des Entwicklungsprojekts bei. Basierend auf einem Typenkatalog auf Anforderungsebene liesse sich vielleicht sogar eine Aspektbibliothek auf Implementierungsebene herstellen. Zur Begriffswahl: Im nicht operationalisierten Zustand entsprechen die nicht-funktionalen Anforderungen den Crosscutting Concerns aus dem Kapitel 5.

12.6.2 Anpassung des Anforderungsmodells für Qualitätsattribute

Das im Abschnitt 12.5 beschriebene Anforderungsmodell für Qualitätsattribute wird von Brito, Moreira (2004) an einigen Stellen leicht angepasst und in folgenden Punkten substanziell erweitert. Insbesondere wird neu der Begriff *nicht-funktionaler Concern* anstelle des Begriffs *Qualitätsattribut* verwendet.

Spezifikation von nicht-funktionalen Concerns. Die nicht-funktionalen Concerns werden mit Hilfe des NFR-Framework identifiziert und spezifiziert. Nicht-funktionale Concerns können anhand der Wissenskataloge aus dem NFR-Framework identifiziert werden. Zu ihrer Spezifikation dient ein Softgoal Interdependency Graph (SIG).

Match Points. Der Begriff *Match Point* wird eingeführt. Er bezeichnet die Punkte, an denen die Komposition stattfindet. Ein Match Point definiert, welche Crosscutting Concerns mit einem gegebenen Concerns zusammengeführt werden sollen. Für jeden Match Point muss eine Kompositionsregel definiert werden. Die Match Points werden in einer Tabelle dargestellt. Pro Concern und Beteiligtem gibt es einen Match Point mit einer Liste von Crosscutting Concerns.

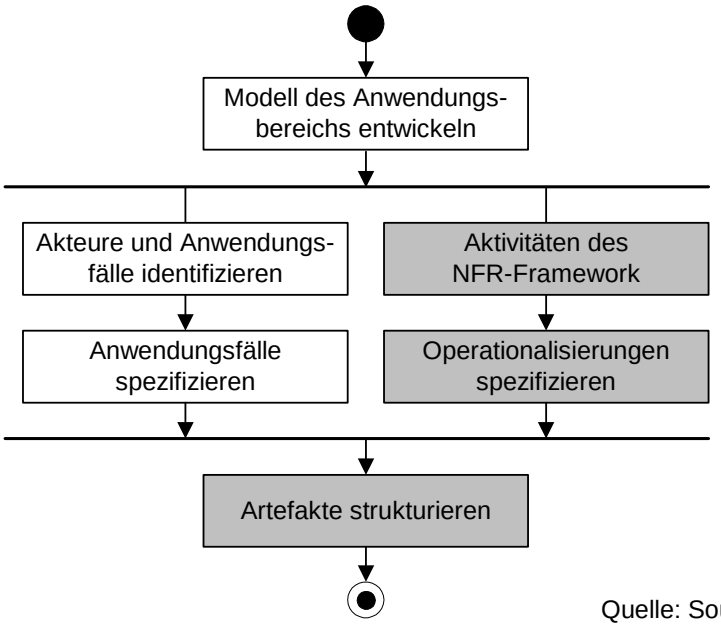
Kompositionsregeln. Die Kompositionsregeln definieren die Reihenfolge, in der die Concerns an einem bestimmten Match Point abgearbeitet werden, mit der Spezifikationssprache LOTOS (siehe ISO, 1989):

- **Sequenz**. Die Sequenz wird in der Form $C_1 >> C_2$ dargestellt, d.h. das Verhalten von C_2 beginnt erst, nachdem C_1 erfolgreich terminiert hat.
- **Unterbruch**. Der Unterbruch wird in der Form $C_1 [> C_2$ dargestellt, d.h. C_2 unterbricht das Verhalten von C_1.
- **Parallelität**. Die Parallelität wird in der Form $C_1 ||| C_2$ dargestellt, d.h. das Verhalten von C_1 und C_2 ist unabhängig voneinander (Nebenläufigkeit ohne Interaktion).
- **Synchronisation**. Die Synchronisation wird in der Form $C_1 || C_2$ dargestellt, d.h. das Verhalten von C_1 und C_2 muss synchronisiert werden (Nebenläufigkeit mit Interaktion).

Beurteilung. Der Begriff *Match Point* ist irreführend und nur verständlich, wenn man berücksichtigt, dass Concerns in diesem Ansatz als atomare Einheiten betrachtet werden. Die Kompositionsregeln schliesslich haben eine Ähnlichkeit mit den Advice-Arten (before, after, around) von AspectJ, sind aber mächtiger als diese. Zur Begriffswahl: Die nicht-funktionalen Concerns entsprechen den Crosscutting Concerns aus dem Kapitel 5.

12.6.3 Anpassung des Unified Software Development Process (USDP)

Der Ansatz von Sousa et al. (2004) basiert auf den anwendungsfallorientierten Aktivitäten des *Unified Software Development Process (USDP)* von Jacobson, Booch und Rumbaugh (1999) und hat zum Ziel, die Separation of Concerns mit Hilfe des NFR-Framework zu unterstützen. Der Informatiker soll mit Techniken ausgestattet werden, welche die systematische Identifikation, Separation, Repräsentation und Komposition von Crosscutting Concerns ermöglichen. Das Vorgehensmodell des um die Aktivitäten des NFR-Frameworks erweiterten Unified Software Development Process (USDP) zeigt die Abbildung 12-8. Die neuen oder geänderten Aktivitäten sind grau schattiert.



Quelle: Sousa et al. (2004)

Abbildung 12-8: Angepasste Aktivitäten des Unified Software Development Process (USDP)

Die Aktivität *Artefakte strukturieren* entspricht im Wesentlichen den Kompositions- bzw. Integrationsaktivitäten aus anderen Ansätzen. Folgende Techniken werden eingesetzt:

Anwendungsfalldiagramm. Nach Jacobson (2003) könnte der Erweiterungsmechanismus verwendet werden, um Aspekte ins Anwendungsfalldiagramm zu integrieren: Der erweiternde Anwendungsfall entspräche einem Aspekt aus AspectJ, der Erweiterungspunkt (engl. extension point) einem Join Point. Sousa et al. (2004) plädieren jedoch für einen neuen Mechanismus: Das „crosscutting" Verhalten wird in einen separaten Anwendungsfall verlagert, und dieser Anwendungsfall wird mittels einer neuen Beziehung (`<<crosscut>>`) mit den Anwendungsfällen verknüpft, die er quer schneidet. Die Abhängigkeitsbeziehungen werden wie folgt voneinander abgegrenzt:

- **`<<include>>`**. Anwendungsfall A schliesst Anwendungsfall B ein, wenn die Ausführung von B erforderlich ist, um das primäre Ziel von A zu erreichen, d.h. A hängt von B ab.
- **`<<extend>>`**. B erweitert A, wenn die Ausführung von B einen komplexen alternativen Pfad repräsentiert, der spezifisch für A ist.
- **`<<crosscut>>`**. B schneidet A quer, wenn B nicht erforderlich ist, damit A sein primäres Ziel erreicht, und B nicht spezifisch für A ist und auch in anderen Anwendungsfällen verwendet werden kann.

Kompositionstabelle. Zur detaillierten Beschreibung der `<<crosscut>>`-Beziehungen aus dem Anwendungsfalldiagramm dient eine separate Kompositionstabelle, wie sie die Tabelle 12-2 zeigt.

„crosscutting" Anwendungsfall <Name>			
betroffener Anwendungsfall	Bedingung (optional)	Operator der Kompositionsregel	betroffener Punkt
<Name>	Bedingung der Erweiterung	{überlappt nach I überlappt vor I überlagert I umhüllt}	Schritt im Szenario

Tabelle 12-2: Kompositionstabelle für „crosscutting" Anwendungsfälle

Die Bedingung ist optional. Fehlt sie, wird das „crosscutting" Verhalten immer ausgeführt. Die Operatoren sind die gleichen wie bei Moreira, Araujo, Brito (2002). Der betroffene Punkt be-

schreibt die Stelle im betroffenen Anwendungsfall, an dem das „crosscutting“ Verhalten ausgeführt werden soll.

Klassendiagramm. Ein neues Stereotyp `<<crosscutting>>` für Klassen wird eingeführt. Diese Klassen implementieren das „crosscutting“ Verhalten. Sie werden später im Entwurf je nachdem auf Aspekte oder auch auf Entwurfsmuster abgebildet. Im Fall von Aspekten wird dies als aspektbezogene Klassen mit dem Stereotyp `<<aspect>>` dargestellt. Sie werden durch eine gerichtete Beziehung mit dem Stereotyp `<<crosscut>>` mit den betroffenen Klassen verbunden.

Um die Wiederverwendbarkeit und Pflegbarkeit der aspektbezogenen Klassen zu bewahren, werden sie in einer wiederum etwas modifizierten Kompositionstabelle detailliert beschrieben, wie die Tabelle 12-3 zeigt:

aspektbezogene Klasse <Name>				
betroffene Klasse	„crosscutting“ Eigenschaften			
	statisch	dynamisch		
		Operator der Kompositionsregel	betroffener Punkt	„crosscutting“ Verhalten
<Name>	<Name der Eigenschaft>	{überlappt nach I überlappt vor I überlagert I umhüllt}	<Beschreibung>	<Name der Operation>

Tabelle 12-3: Kompositionstabelle für aspektbezogene Klassen

Kollaborationsdiagramm. Objekte von `<<crosscutting>>` Klassen werden nicht direkt ins Kollaborationsdiagramm integriert, um die Separation of Concerns zu bewahren. Ihre Auswirkungen werden in der um Objekte und Nachrichten erweiterten Kompositionstabelle beschrieben.

Beurteilung. Dies ist ein durchdachter, praxistauglicher Ansatz und ausserdem der einzige, der auch in Richtung Entwurf und Implementierung weitergeführt wird. Er eignet sich für die Identifikation, Darstellung und Beschreibung von Crosscutting Concerns sowie für die Kommunikation mit dem Auftraggeber. Die Aussage, dass Operationalisierungen von nicht-funktionalen Concerns generell „crosscutting“ sind, wird dreimal gemacht, aber leider nicht weiter erläutert. Interessant sind die Kompositionstabellen: Sie scheinen praxistauglich zu sein, eignen sich auch für den Einsatz in grösseren Systemen und können zudem relativ direkt z.B. auf AspectJ abgebildet werden. Interessant ist auch die Unterscheidung zwischen der Auftraggeber- und der Entwicklersicht: In allen Diagrammen, die primär den Entwickler betreffen (z.B. Kollaborationsdiagramm), werden die Concerns konsequent separiert, während sie in allen Diagrammen, die den Auftraggeber betreffen (z.B. Anwendungsfalldiagramm) integriert werden. Dies vermutlich, um das abstrakte und erklärungsbedürftige Prinzip der Separation of Concerns vor dem Auftraggeber zu verbergen.

12.7 Scenario-Based Requirements

Im Ansatz von Araujo, Whittle, Kim (2004) werden Anforderungen als *Interaktionen* (= Szenarien) modelliert. Diese zeigen die Kommunikation zwischen mehreren Komponenten (globale Sicht) und eignen sich vorzüglich zur Modellierung von Anforderungen, da sie natürlich und intuitiv sind. Aspektbezogene und nicht-aspektbezogene Interaktionen werden unabhängig voneinander modelliert. Zwecks Komposition und Validierung werden die Interaktionen mit einem Algorithmus in *endliche Automaten* (engl. Finite State Machine, FSM) transformiert. Diese zeigen das innere Verhalten der Komponenten (lokale Sicht). Die aspektbezogenen und nicht-aspektbezogenen endlichen Automaten werden schliesslich zu ausführbaren (!) endlichen Automaten kombiniert. Damit können die Anforderungen leicht validiert werden. Die Abbildung 12-9 zeigt das Prozessmodell der Scenario-Based Requirements.

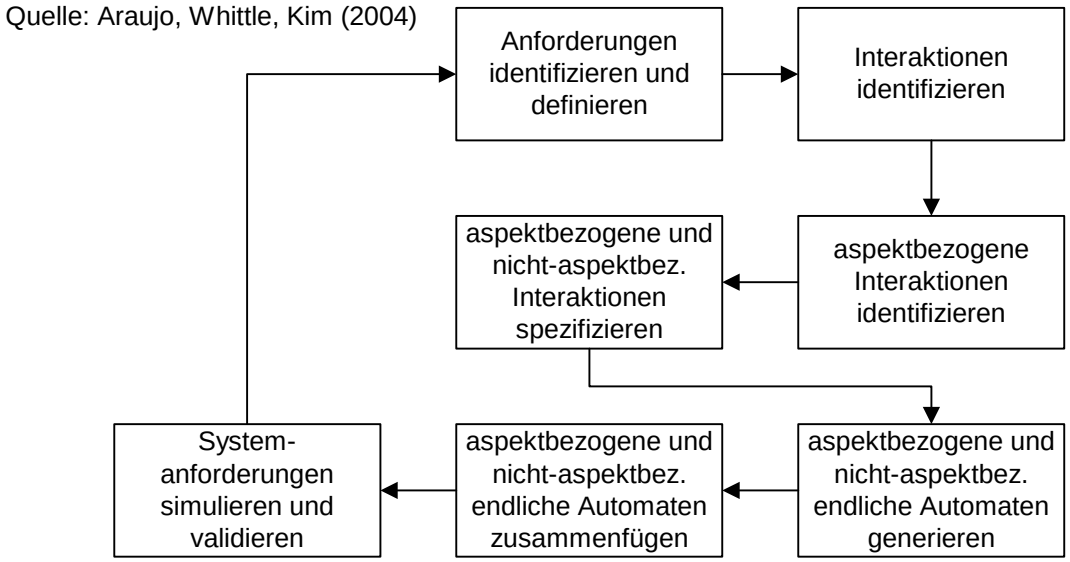


Abbildung 12-9: Prozessmodell der Scenario-Based Requirements

Anforderungen identifizieren und definieren. Anwendungsfälle werden identifiziert und in einem UML-Anwendungsfalldiagramm dargestellt.

Interaktionen identifizieren. Aus dem Anwendungsfalldiagramm werden die Interaktionen extrahiert und tabellarisch aufgelistet. Spezialfälle werden als eigenständige Interaktionen betrachtet.

Aspektbezogene Interaktionen identifizieren. Interaktionen, die andere Interaktionen quer schneiden, werden als aspektbezogene Interaktionen gekennzeichnet.

Aspektbezogene und nicht-aspektbezogene Interaktionen spezifizieren. Nicht-aspektbezogene Interaktionen werden mit UML-Sequenzdiagrammen dargestellt, für aspektbezogene Interaktionen werden so genannte *Interaktionsmuster-Spezifikationen* (engl. Interaction Pattern Specification, IPS) verwendet. IPS sind im Prinzip auch Sequenzdiagramme, die – ganz ähnlich wie bei den Composition Patterns – *Rollen* anstelle von konkreten Modellelementen für Nachrichten und Objekte enthalten können. Rollen sind an einem senkrechten Strich vor dem Namen erkennbar (z.B. │Action) und werden später anlässlich der Komposition durch konkrete Modellelemente ersetzt. Ein Modell ist *musterkonform*, wenn die Modellelemente, welche die Rollen des Musters spielen, die in den Rollen definierten Eigenschaften aufweisen.

Aspektbezogene und nicht-aspektbezogene endliche Automaten generieren (Synthese). Aus den Sequenzdiagrammen und den IPS werden endliche Automaten generiert. Die Sequenzdiagramme werden dabei zu UML-Zustandsdiagrammen, die IPS zu so genannten *Zustandsautomaten-muster-Spezifikationen* (State Machine Pattern Specification, SMPS). SMPS sind – analog zu den IPS – im Prinzip auch Zustandsautomaten, die Rollen anstelle von konkreten Modellelementen für Transitionen und Zustände enthalten können Die Transformation erfolgt in zwei Schritten:

1. Aus jedem Sequenzdiagramm wird ein Zustandsdiagramm pro Objekt erzeugt. Dabei werden eingehende Nachrichten als Ereignisse, ausgehende Nachrichten als Aktionen und die verschiedenen Objektzustände[33] als Zustände betrachtet.
2. Alle Zustandsdiagramme eines Objekts werden anhand von ähnlichen Zuständen zusammengeführt. Ähnlich sind zwei Zustände dann, wenn sie gemeinsame ein- und ausgehende Transitionen besitzen.

[33] Diese müssen allerdings zuvor im Sequenzdiagramm vermerkt worden sein.

Aspektbezogene und nicht-aspektbezogene endliche Automaten zusammenfügen (Komposition). Die Komposition erfolgt in zwei Schritten, die oft gemeinsam durchgeführt werden.

1. **Instanzierung**. Jede SMPS wird für alle Zustandsdiagramme instanziert, die sie quer schneidet, d.h. konkrete Modellelemente werden an die Rollen der SMPS gebunden (engl. bind). Dabei wird eine Transitionsrolle auf genau ein konkretes Modellelement abgebildet, während mehrere Zustandsrollen auf mehrere konkrete Modellelemente abgebildet werden können. Letzteres erfordert Benutzerinput und beeinflusst den nachfolgenden Schritt.
2. **Mischung** (engl. merging). Die Mischung eines instanzierten aspektbezogenen mit einem nicht-aspektbezogenen endlichen Automaten erfordert einen ausgeklügelten Algorithmus, dessen detaillierte Darstellung den Rahmen dieser Arbeit sprengen würde.

In anderen Ansätzen erfolgt die Komposition bereits auf der Ebene der Interaktionen statt auf der Ebene der endlichen Automaten. Das ist von Vorteil, wenn die Entwickler nicht gewohnt sind, mit endlichen Automaten umzugehen. Jedoch ist die Granularität der Komposition dann eher grob, und die Benutzer müssen die Kompositionsoperatoren selber spezifizieren.

Systemanforderungen validieren und simulieren. Anhand der resultierenden endlichen Automaten können die Systemanforderungen validiert und simuliert werden.

Beurteilung. Die Scenario-Based Requirements sind auf den ersten Blick attraktiv. Sie basieren auf Algorithmen und sind daher teilweise automatisierbar. Es dürfte aber trotzdem aufwändig sein, sie für die Spezifikation grösserer Systeme einzusetzen. In gewissen Bereichen ist der Ansatz unklar. So ist beispielsweise nicht beschrieben, wie man bei der Instanzierung am besten herausfindet, welche Zustandsdiagramme ein bestimmtes SMPS quer schneidet. Die Hauptkritik ist aber eine ganz andere: Ein wichtiges Ziel der Aspektorientierung ist es, die *Separation of Concerns* möglichst über alle Phasen des Entwicklungsprozesses zu bewahren und die Komposition der Concerns zum Gesamtsystem möglichst spät durchzuführen. Folglich sollen die Ansätze der aspektorientierten Anforderungstechnik die Komposition *keinesfalls selber durchführen*, wie es dieser Ansatz tut, sondern lediglich die *Kompositionsregeln beschreiben*. In den Scenario-Based Requirements geht die Separation of Concerns mit der Komposition verloren. Daher eignet sich die resultierende Spezifikation nicht als Grundlage für eine aspektorientierte Implementierung, sondern lediglich als Grundlage für eine Validierung oder Simulation. Ob der Aufwand dafür gerechtfertigt ist, muss im Einzelfall geprüft werden. Möglicherweise könnte das System gleich generiert werden. Zudem: Aussagen zur Validierung werden in Araujo, Whittle, Kim (2004) zwar angekündigt, aber nicht gemacht. Zur Begriffswahl: Aspektbezogen ist ein Synonym für crosscutting.

12.8 AOCRE (Aspect-Oriented Component Requirements Engineering)

AOCRE von Grundy (1999) ist eine Anforderungsmethode für komponentenbasierte Software-Systeme. Warum eine spezielle Methode für komponentenbasierte Software-Systeme? Traditionelle Methoden der Anforderungstechnik eignen sich nicht für wiederverwendbare Komponenten, da bei diesen häufig weder die Beteiligten noch die Anforderungen bekannt sind und alle Komponenten dynamisch konfigurierbar sein müssen. Und existierende Methoden für komponentenbasierte Systeme beschränken sich im Normalfall auf Entwurf und Implementierung.

Vorgehen. Pro Komponente werden die funktionalen und nicht-funktionalen Anforderungen an vordefinierte so genannte *Schlüsselaspekte* (z.B. Benutzungsoberfläche, Kollaboration, Persistenz, Verteilung oder Konfiguration) identifiziert und spezifiziert. Diese Schlüsselaspekte werden von jeder Komponente entweder angefordert oder zur Verfügung gestellt. Pro Komponente und Schlüsselaspekt werden dann die so genannten *Aspektdetails* identifiziert, für welche die Komponente

Leistungen anbietet (+) oder bei anderen Komponenten nachfragt (–). Die Aspekte werden anschliessend wie in der Abbildung 12-10 dargestellt verfeinert und ausserdem textlich beschrieben.

Quelle: Grundy (1999)

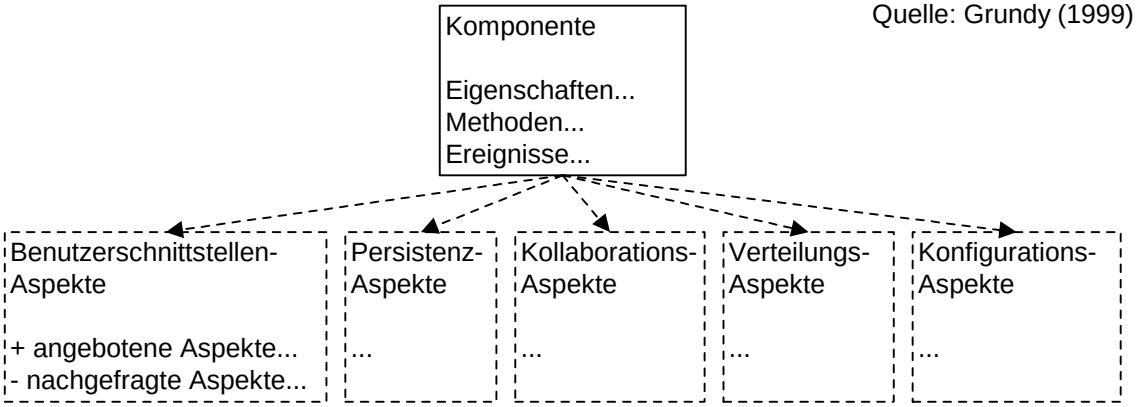


Abbildung 12-10: Darstellung von Komponenten und Aspekten

Schliesslich werden immer wieder vorkommende Aspekte zu so genannten *aggregierten Aspekten* zusammengefasst. Damit entstehen aspektorientierte Anforderungen für eine Menge von Komponenten oder das gesamte System. Aspekte können später mittels Interfaces, Reflexion oder Entwurfsmustern entworfen und implementiert werden.

Beurteilung. AOCRE wurde hier erwähnt, weil der Ansatz von Grundy (1999) in der aspektorientierten Literatur relativ häufig zitiert wird. Bei den Aspekten handelt es sich nicht um implementierte Crosscutting Concerns im Sinne von Abschnitt 7.1, sondern vielmehr um eine Kategorisierung von immer wieder vorkommenden Eigenschaften von Komponenten mit dem Ziel, die Schnittstellen zwischen den Komponenten zu standardisieren.

12.9 Facetten von Concerns

Nach Bogdan (2004) sind Facetten von Concerns Fragen, die man sich zu jedem Concern C stellen sollte:

- **Definition**. Wie wird C definiert?
- **Beispiel**. Welches sind Dinge, welche der Definition von C entsprechen?
- **Behandlung**. Wann lässt sich sagen, C werde behandelt?
- **Lösung**. Wann lässt sich sagen, C sei abgeschlossen?
- **Generalisierung**. Welche anderen Concerns haben weniger restriktive Definitionen als C?
- **Spezialisierung**. Welche anderen Concerns entsprechen der Definition von C mit zusätzlichen Einschränkungen?
- **Lebenszyklus-Phase**. Welche anderen Concerns werden in der gleichen Lebenszyklus-Phase behandelt oder abgeschlossen wie C?
- **Arbeitsablauf**. Welche anderen Concerns werden im gleichen Arbeitsablauf behandelt oder abgeschlossen wie C?
- **Artefakt**. Welche anderen Concerns werden im gleichen Artefakt realisiert wie C?
- **Rolle**. Welche anderen Concerns sind für die gleiche Rolle relevant wie C?
- **Abhängigkeit**. Von welchen anderen Concerns ist C abhängig?

Zu jeder Facette F gibt es ausserdem Subfacetten, welche die folgenden Fragen beantworten (dabei ist C.F die Facette F des Concerns C):

- **Prüfung**. Wie können Elemente von C.F geprüft werden?
- **Bewertung**. Wie können Elemente von C.F bewertet werden?

Beurteilung: Die Facetten von Concerns sind kein eigenständiger Ansatz. Sie können jedoch als *Checkliste* zur Identifikation von Concerns im Rahmen von anderen Ansätzen dienen.

12.10 Theme/Doc

Theme/Doc und Theme/UML (siehe im Abschnitt 10.3) bilden zusammen den Theme-Ansatz. Der Theme-Ansatz weist eine gewisse Verwandtschaft mit der Subjektorientierung auf.

> **Definition** *Thema* (engl. theme). Merkmal eines Systems. Mehrere Themen können in einem multidimensionalen Modell (Ossher, Tarr, 2001) zu einem funktionierenden Ganzen kombiniert werden. Es gibt Basis- und „crosscutting" Themen (Aspekte).

Theme/Doc von Baniassad, Clarke (2004) ist ein Werkzeug, das die Visualisierung und Analyse der Dokumentation von Anforderungen und die Identifikation von „crosscutting" Verhalten erlaubt. Theme/Doc basiert auf der Annahme, dass Aktionen (Verhalten), die in der gleichen Anforderung beschrieben werden, auch miteinander in Beziehung stehen, wobei diese Beziehung zufällig/falsch, hierarchisch oder „crosscutting" sein kann. „Crosscutting" bedeutet, dass das dynamische Verhalten in einem Thema im Anschluss an das Verhalten in anderen Themen ausgelöst wird. Folgendes Vorgehen ist zu wählen:

Anforderungen formulieren. Der Auftraggeber formuliert die Anforderungen mit Hilfe von vorgegebenen so genannten *Schlüsselaktionen* (z.B. `Auswertung`) und *Schlüsselentitäten* (z.B. `Ausdruck`) möglichst präzis in natürlicher Sprache. Hier als Beispiel ein Teil der Anforderungen an ein System, das Ausdrücke auswertet.

R1 Die Auswertung ermittelt das Ergebnis des auszuwertenden Ausdrucks.

R2 Die Anzeige stellt den Ausdruck textlich dar.

R3 Die Syntaxprüfung bestimmt, ob ein Ausdruck syntaktisch korrekt ist.

R4 Das Logging protokolliert die Aktivitäten der Auswertung, Anzeige und Syntaxprüfung.

Aktionen und Entitäten identifizieren. Mit Hilfe vordefinierter Schlüsselwörter werden anhand der Anforderungen sinnvolle Aktionen und Entitäten identifiziert. Im obigen Beispiel sind dies die Aktionen `Auswertung`, `Anzeige`, `Ermittlung`, `Syntaxprüfung` und `Logging` sowie die Entität `Ausdruck`. Daraus wird die so genannte *Action View* generiert, wie sie in der Abbildung 12-11 dargestellt ist. Die Rechtecke entsprechen den Anforderungen, die Rauten den Aktionen.

Quelle: Baniassad, Clarke (2004)

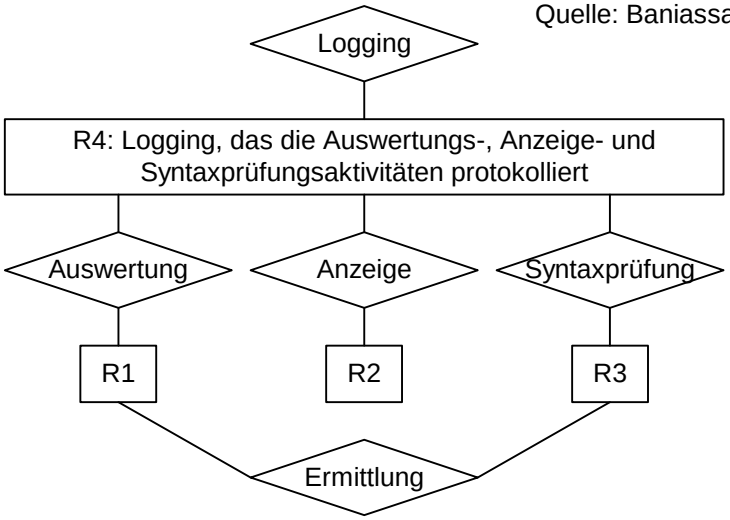


Abbildung 12-11: Action View von Theme/Doc

Aktionen nach Themen kategorisieren: In diesem Schritt werden anhand der *Action View* die so genannten *Themen* identifiziert. Das sind alle Aktionen, die intuitiv so wichtig sind, dass sie später separat modelliert werden sollten. Sie werden auch *Major Actions* genannt. Daraus entsteht die so genannte *Major Action View*. Die übrigen Aktionen, die *Minor Actions*, werden später zu Methoden. Zum obigen Beispiel: Die `Ermittlung` scheint im Vergleich mit den übrigen Aktionen von untergeordneter Bedeutung zu sein und wird daher zur *Minor Action* degradiert. Die *Major Action View* sieht gleich aus wie die *Action View* in der Abbildung 12-11, jedoch ohne die `Ermittlung`.

„Crosscutting" Themen identifizieren. Anhand der *Major Action View* kann festgestellt werden, ob es sich bei einem Thema um ein so genanntes *Basisthema* oder einen so genannten *Aspekt* handelt. Aus Anforderungen, die mit mehr als einem Thema verknüpft sind (z.B. R4), resultieren Aspekte (z.B. `Logging`), aus allen anderen Anforderungen (z.B. R1) resultieren Basisthemen (z.B. `Auswertung`). Diese Unterscheidung führt zur so genannten *Clipped Action View*. Ihr Name rührt daher, dass im Fall von Aspekten die Verbindung zwischen der Anforderung und den Basisthemen gelöst (engl. clip) wird. Im obigen Beispiel wird die Verbindung zwischen der Anforderung R4 und den Themen `Auswertung`, `Anzeige` und `Syntaxprüfung` gelöst. Die Abbildung 12-12 zeigt die entsprechende *Clipped Action View*. Die grauen Pfeile zeigen die „crosscutting"-Hierarchie.

Quelle: Baniassad, Clarke (2004)

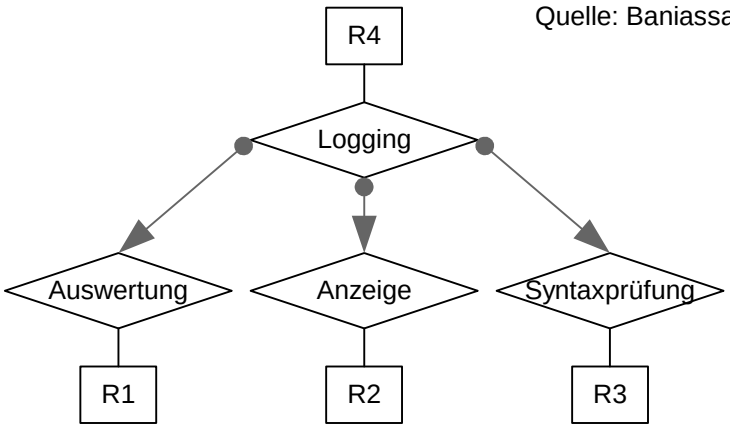


Abbildung 12-12: Clipped Action View von Theme/Doc

Individuelle Themensicht erstellen. Themen können sowohl individuell als auch in den *Action Views* gruppiert dargestellt werden. Die individuelle Themensicht zeigt die Entitäten, Anforderungen und Minor Actions, die mit einem Thema, einer Major Action, verknüpft sind. Diese Sicht ist die Grundlage für den Entwurf mit Theme/UML (siehe im Abschnitt 10.3).

Beurteilung. Was an Theme/Doc auffällt, sind die unübliche Begriffswahl und die unübliche Notation. Das führt dazu, dass der Ansatz exotisch anmutet und schwierig zu verstehen ist. Theme/Doc könnte in der Anforderungsphase zur Identifikation von Concerns eingesetzt werden. Theme/Doc sieht vor, die Anforderungen anhand von vordefinierten Schlüsselwörtern zu formulieren. Aus diesem Grund eignet es sich weniger für grössere Systeme oder unbekannte Anwendungsbereiche. Die Semantik der Verbindungen in den Grafiken bleibt leider unerklärt. Zur Begriffswahl: Was Theme/Doc mit dem Begriff Aspekt bezeichnet, entspricht dem Begriff Crosscutting Concern aus dem Kapitel 5.

12.11 Cosmos (Concern-Space Modeling Schema)

Concerns werden in Methoden und Werkzeugen als Modellierungskonstrukte zweiter Klasse behandelt. Eine globale Betrachtung von Concerns, die den ganzen Software-Lebenszyklus abdeckt und unabhängig von bestimmten Entwicklungswerkzeugen ist, fehlt nach wie vor. *Cosmos* von Sutton, Rouvellou (2002) soll diese Lücke füllen.

> **Definition** *Concern-Raum*. Organisierte Repräsentation von Concerns und ihren Beziehungen.

Ein universelles Modellierungsschema für Concern-Räume soll folgenden Anforderungen genügen:

- Es unterstützt die Repräsentation beliebiger (auch zusammengesetzter) Concerns und beliebiger Beziehungen zwischen Concerns.
- Es unterstützt die Zuordnung von Concerns zu beliebigen Software-Einheiten, Arbeitsergebnissen oder Systemelementen.
- Es ist sprachunabhängig.
- Es ist methodenunabhängig.
- Es kann während des gesamten Software-Lebenszyklus angewendet werden.
- Es unterstützt eine Vielzahl von Software Engineering-Aufgaben.

Cosmos (Concern-Space Modeling Schema) ist ein solches universelles Modellierungsschema für Concern-Räume. Es enthält drei Arten von Elementen: Concerns, Beziehungen und Prädikate:

12.11.1 Concerns

Es gibt logische und physische Concerns. Sie können unabhängig voneinander betrachtet und modelliert werden.

Logische Concerns. Sie stellen die konzeptionellen Sachverhalte dar, die an einem Software-System interessieren. Beispiele: Funktionalität, Verhalten, Performance, Robustheit, Kopplung, Grösse, Kosten etc. Es gibt fünf Typen von logischen Concerns:

- **Klassifikationen** sind Dimensionen im Concern-Raum. Da Concern-Räume mehrdimensional sind, kann eine bestimmte Instanz gleichzeitig in verschiedene Klassifikationen eingeteilt werden. Beispiele: Funktionalität, Verhalten.
- **Klassen** sind Concerns, die andere Concerns klassifizieren. Beispiele: Kernverhalten, Logging-Verhalten, Garbage Collection-Verhalten.

- **Instanzen** sind spezifische Concerns, die keine anderen Concerns klassifizieren oder charakterisieren. Beispiele: spezifische Funktionalität wie „Objekt hinzufügen“, spezifisches Verhalten wie „Eingabe prüfen“.
- **Eigenschaften** sind Concerns, die andere (logische) Concerns, namentlich Klassen und Instanzen, charakterisieren. Beispiele: Performance, Grösse, Erweiterbarkeit.
- **Themen** sind beliebige Sammlungen von Concerns. Beispiel: Konfigurierbarkeit.

Physische Concerns. Sie stellen die tatsächlichen Elemente eines Systems dar, welche die logischen Concerns repräsentieren, implementieren, unterstützen oder anderweitig betreffen. Es gibt drei Typen von physischen Concerns:

- **Instanzen** sind spezifische Elemente eines Systems, einschliesslich Arbeitsergebnissen, Hardware-Einheiten, Systemen und Leistungen.
- **Kollektionen** sind Gruppen physischer Concerns. Kollektionen können homogen oder heterogen sein. Beispiele: Pakete mit Quelldateien, Workstations an einem Netz.
- **Attribute** sind Werte, die physische Instanzen und Kollektionen charakterisieren. Typischerweise repräsentieren sie die interessierenden Eigenschaften der logischen Concerns.

12.11.2 Beziehungen

Es gibt kategorisierende, interpretierende, physische und Abbildungs-Beziehungen.

Kategorisierende Beziehungen. Sie verbinden Concerns auf der Basis von Kategorien. Es gibt sieben Typen von kategorisierenden Beziehungen:

- **Klassifizierung** ordnet Klassen einem Klassifizierungssystem zu. Beispiel: die Funktionalität ist eine Klassifizierung (`ClassificationOf`) der Kernfunktionalität, das Verhalten ist eine Klassifizierung des Logging-Verhaltens.
- **Generalisierung** ordnet Klassen einer Vererbungsbeziehung zu. Beispiel: das Logging von Operationen ist eine Subklasse (`SubclassOf`) des Logging-Verhaltens.
- **Instanzierung** ordnet Instanzen den Klassen zu, zu denen sie gehören. Beispiel: das Hinzufügen eines Objekts ist eine Instanz (`InstanceOf`) der Kernfunktionalität.
- **Charakterisierung** ordnet Eigenschaften den Klassen oder Instanzen zu, auf die sie zutreffen. Beispiel: Performance ist eine Charakterisierung (`PropertyOf`) der Funktionalität.
- **Mitgliedschaft** ordnet physische Instanzen den Kollektionen zu, zu denen sie gehören. Beispiel: Cache_Core.java ist ein Mitglied (`MemberOf`) des Pakets com.ibm.ws.abr.gps.
- **Zuordnung** verbindet Attribute mit physischen Instanzen oder Kollektionen. Beispiel: Grösse ist ein Attribut (`AttributeOf`) von Cache_Core.java
- **Thematisierung** ordnet beliebige Concerns einem Thema zu. Beispiel: die Eigenschaft „Konfigurierbarkeit des Logging“ und die Klasse „Logging-Verhalten“ gehören zum Thema (`SubjectOf`) Logging.

Interpretierende Beziehungen. Sie widerspiegeln semantische Verbindungen zwischen logischen Concerns und hängen primär von der kontextbehafteten Interpretation der Concern-Semantik ab. Es gibt keine vordefinierten interpretierenden Beziehungen, sie werden abhängig vom Anwendungsbereich bestimmt. Beispiele:

- **Beitrag**. Ein logischer Concern trägt zu einem anderen bei. Beispiel: Optimierung trägt zur Performance bei (`ContributesTo`).
- **Anstoss**. Ein logischer Concern gibt Anstoss zu einem anderen. Beispiel: die Recovery-Fähigkeit gibt Anstoss (`Motivates`) zum Logging.

- **Logische Implementierung**. Ein logischer Concern (z.B. ein grafisches Darstellungsmittel) implementiert (`LogicallyImplements`) einen anderen logisch.
- **Zulassung**. Ein logischer Concern lässt einen anderen zu. Beispiel: (erst) das Logging lässt das Puffern des Logs zu (`Admits`), d.h. macht das Puffern des Logs überhaupt sinnvoll.

Physische Beziehungen. Sie verbinden physische Concerns. Beispiel: Verknüpfung von Komponenten zu einer Applikation. Auch hier gibt es keine vordefinierten Beziehungen.

Abbildungs-Beziehungen. Sie verbinden logische und physische Concerns. Beispiel: Die Beziehung „implementiert physisch" bedeutet, dass ein physischer Concern (z.B. ein Stück Code) einen logischen Concern (z.B. eine Funktion) physische implementiert. Auch hier gibt es keine vordefinierten Beziehungen.

12.11.3 Prädikate und Konsistenz

Die Konsistenz ist ein wichtiges Thema im Bereich der Concern-Räume, aber weil die Konsistenzbedingungen je nach Anwendungsbereich unterschiedlich sind, gibt es nur wenige vordefinierte Prädikate:

- Konsistenzbedingungen, welche die Integrität eines Cosmos-Modells sicherstellen.
- Konsistenzbedingungen für die interpretierenden Beziehungen: Eine Anstossbeziehung impliziert eine Beitragsbeziehung (aber nicht umgekehrt), die Anstossbeziehung und die logische Implementierungsbeziehung schliessen einander aus.

12.11.4 Zusammenfassung

Ein Cosmos-Modell hält Information über Concerns und ihre Beziehungen fest. Damit lassen sich grundlegende Fragen zu einem Software-System beantworten. Anstossbeziehungen (`Motivates`) liefern Antworten auf Fragen nach dem Warum, z.B. warum ist die Konfigurierbarkeit des Logging wichtig? Beitragsbeziehungen (`ContributesTo`) liefern Antworten auf Fragen nach Auswirkungen, z.B. worauf wirkt sich die Änderung des Logging-Verhaltens aus? Enthält das Cosmos-Modell auch physische Concerns und Abbildungsbeziehungen, ermöglicht dies die (Rück-) Verfolgbarkeit zwischen Concerns und Phasen bzw. Arbeitsergebnissen. Weiter können damit auch die Auswirkungen abgeschätzt werden, die Änderungen der logischen Ebene auf die physische Ebene haben.

Cosmos erfüllt die eingangs erwähnten Anforderungen. Ausserdem ist es universell und flexibel, indem es den Benutzern die Definition von eigenen Klassifikationen, Klassen und Instanzen von Concerns ermöglicht, ebenso die Definition von zusätzlichen Beziehungstypen. Zusammenfassend kann Cosmos als eine Art *semantischer Hyper-Index* eines Software-Systems betrachtet werden. Dieser ist für die Entwicklung von einer gewissen Bedeutung, insbesondere aber für die Integration und Pflege. Leider existiert für Cosmos noch keine Werkzeug-Unterstützung. Dies ist aber geplant.

12.11.5 Beurteilung

Cosmos ist kein eigenständiger Ansatz der aspektorientierten Anforderungstechnik. Es ist nicht in einen Prozess eingebettet und definiert lediglich eine Sprache zur Beschreibung von Concerns und ihren Beziehungen. Aufgrund des Anspruchs, universell zu sein, ist diese Sprache wohl flexibel und erweiterbar, dies jedoch auf Kosten der präzisen Definition der Sprachkonstrukte. Da die Sprachkonstrukte keinem gängigen Standard entsprechen, relativ kompliziert sind und nicht durch ein Werkzeug unterstützt werden, ist Cosmos nicht praxistauglich und dürfte ein Nischendasein fristen.

12.12 Zusammenfassung der Ansätze

Mit Ausnahme der USDP-Anpassung von Sousa et al. (2004) erfüllt kein einziger Ansatz alle zu Beginn des Kapitels aufgeführten Beurteilungskriterien. Zusammenfassend können die Ansätze wie folgt beurteilt werden:

12.12.1 Vorgehensmodell

Fast alle Ansätze umfassen ein Vorgehensmodell. Ausnahmen bilden die Facetten von Concerns und Cosmos, die aber nicht als vollwertige Ansätze zu betrachten sind. Die Vorgehensmodelle sind sich im Wesentlichen sehr ähnlich. In den meisten sind drei Hauptaktivitäten zu erkennen: die Gewinnung der aus den Core Concerns resultierenden Anforderungen, die Gewinnung der aus den Crosscutting Concerns resultierenden Anforderungen und schliesslich die Komposition der Core Concerns mit den Crosscutting Concerns. Hinter den ersten beiden Hauptaktivitäten verbirgt sich die *Separation*, während sich hinter der letzten die *Integration* verbirgt.

12.12.2 Darstellung und Beschreibung; Kommunikation mit dem Auftraggeber

Als Darstellungs- und Beschreibungsmittel für Crosscutting Concerns und ihre Integration in die Core Concerns werden überwiegend Tabellen und UML-Diagramme (Anwendungsfall-, Sequenz- und Zustandsdiagramme) eingesetzt, die zu diesem Zweck um „crosscutting“ Konzepte erweitert werden. In der USDP-Anpassung wird der Softgoal Interdependency Graph (SIG) verwendet. Diese Darstellungsmittel sind intuitiv verständlich und eignen sich für die Kommunikation mit dem Auftraggeber. Daneben kommt in AORE die XML zum Einsatz, und Theme/Doc verwendet ein eigenes, etwas exotisch anmutendes Darstellungsmittel. Bei diesen beiden dürfte sich die Kommunikation mit dem Auftraggeber ohne vorgängige Einführung oder Schulung als eher schwierig erweisen.

12.12.3 Grundlage für den Entwurf; Kommunikation mit den Entwicklern

Nur die USDP-Anpassung schafft mit der Anwendung von Tabellen und UML-Klassendiagrammen gewisse Grundlagen für den späteren Entwurf und die Implementierung der spezifizierten Crosscutting Concerns. Dass in allen anderen Ansätzen nicht einmal Klassendiagramme eingesetzt werden, erstaunt doch ein wenig.

12.12.4 Eignung für grössere Systeme

Mit Ausnahme von AORE (wegen der XML), der Scenario-Based Requirements (wegen der grossen Menge von Diagrammen) und von Theme/Doc (wegen der Formulierung der Anforderungen mit Hilfe von Schlüsselwörtern) eignen sich alle Ansätze grundsätzlich auch für die Spezifikation grösserer Systeme. Dabei gilt die Einschränkung, dass der Einsatz von Anwendungsfalldiagrammen in grösseren Systemen grundsätzlich schwierig ist. Dies gilt aber auch bei der konventionellen Entwicklung.

12.12.5 Begriffe

Die Begriffswahl in den Ansätzen der aspektorientierten Anforderungstechnik ist alles andere als einheitlich. Die Tabelle 12-4 zeigt, wie die Concerns, die Crosscutting Concerns und die Core Concerns aus dem Kapitel 5 in den verschiedenen Ansätzen bezeichnet werden.

	Concern	Crosscutting Concern	Core Concern
PREview	–	Concern, externe Anforderung	Viewpoint-Anforderung
AORE	Concern	Crosscutting Concern, Aspektkandidat, aspektbezogene Anforderung	Beteiligten-Anforderung, nicht-aspektbezogene Anforderung, Viewpoint
Vision	–	nicht-funktionaler Aspektkandidat, aspektbezogener Anwendungsfall	Viewpoint
Aspektorientierte Anforderungen mit der UML	Concern	Crosscutting Concern, “crosscutting” nicht-funktionale Anforderung, Aspektkandidat	funktionaler Concern, funktionale Anforderung
Anforderungsmodell f. Qualitätsattribute	–	“crosscutting” Qualitätsattribut	funktionale Anforderung
NFR-Framework	Anforderung	nicht-funktionale Anforderung, Softgoal	funktionale Anforderung
USDP-Anpassung	Concern	Crosscutting Concern, nicht-funktionaler Concern	–
Theme/Doc	Theme, Major Action, Key Action	Crosscutting Theme, Aspect	Base Theme
Cosmos	Logical Concern	–	–
Subjektorientierung	Subject	–	–

Tabelle 12-4: Der Concern-Begriff in verschiedenen Ansätzen

Teil III:
Die Aspektorientierung im Software Engineering

Fast alle im Teil II beschriebenen Ansätze behandeln die Aspektorientierung isoliert in einem eng begrenzten Gebiet des Software Engineering. Der Teil III versucht, die Aspektorientierung in den Zusammenhang bekannter Prozesse und Konzepte des Software Engineering zu stellen. Das Kapitel 13 behandelt die Aspektorientierung im Software-Entwicklungsprozess. Das Kapitel 14 zeigt, wie sich die Aspektorientierung in verschiedene Konzepte des Software Engineering einordnen lässt. Die folgenden Kapitel 15 und 16 befassen sich mit den Trends und Lücken im Forschungsgebiet der Aspektorientierung.

13. Aspektorientierung im Software-Entwicklungsprozess

Dieses Kapitel versucht zu zeigen, wodurch sich die Entwicklung eines aspektorientierten Systems von der Entwicklung eines konventionellen Systems unterscheidet. Es macht Vorschläge, wie Crosscutting Concerns identifiziert, dargestellt, spezifiziert und validiert werden können. Es gibt Denkanstösse, was bei der Architektur und beim Entwurf aspektorientierter Systeme zu beachten ist. Und schliesslich zeigt es, wie aspektorientierte Systeme getestet und gemessen werden können.

13.1 Grundregel der aspektorientierten Entwicklung

Die aspektorientierte Entwicklung hat zum Ziel, die Concerns möglichst von Anfang an zu separieren und die Separation of Concerns anschliessend so lange wie möglich zu bewahren. Die *Grundregel der aspektorientierten Entwicklung* bringt dies kurz und prägnant zum Ausdruck:

Frühe Separation – späte Integration

Ersteres kann durch eine aspektorientierte Anforderungsphase erreicht werden, letzteres durch eine aspektorientierte Entwurfs- und Implementierungsphase.

13.1.1 Frühe Separation

Damit sich das Nutzenpotenzial der aspektorientierten Programmierung (siehe im Abschnitt 9.11) in den späteren Phasen des Entwicklungsprozesses entfalten kann, ist es erforderlich, die Concerns ganz zu Beginn der Entwicklung, d.h. bereits in der Anforderungsphase, systematisch zu separieren. Je besser dies gelingt, desto grösser ist die Wahrscheinlichkeit, dass die Vorteile der Aspektorientierung in den späteren Phasen zum Tragen kommen. Die Begründung dafür ist einfach: Sind die Concerns einmal „scattered“ und „tangled“, so lässt sich das nur noch mit mühsamer Handarbeit wieder rückgängig machen. Oder anders formuliert: Was einmal vermischt ist, lässt sich nicht mehr so leicht in seine Bestandteile zerlegen. (Aschenbrödel kann ein Lied davon singen.) Murphy et al. (2001) gelangen in ihrer Untersuchung zur gleichen Erkenntnis (siehe auch im Abschnitt 9.10). Hinzu kommt, dass Concerns oft im Konflikt zueinander stehen. Daher ist eine frühe Separation auch aus wirtschaftlicher Sicht sinnvoll: je eher solche Konflikte sichtbar werden und – durch Priorisierung oder entsprechende Massnahmen – gelöst werden können, desto besser.

Aspektorientierte Anforderungs-, Entwurfs- und Implementierungsphase. Eine aspektorientierte Implementierung ist nur dann wirklich nutzbringend, wenn auch die Anforderungen bereits aspektorientiert spezifiziert werden. Das Ziel ist eine nahtlose, *integrierte aspektorientierte Entwicklung*. Dies erfordert zwingend Ansätze der aspektorientierten Anforderungstechnik, d.h. Verfahren und Sprachen zur aspektorientierten Identifikation, Darstellung, Beschreibung und Validierung von Crosscutting Concerns. Die im Kapitel 12 vorgestellten Ansätze erfüllen diesen Anspruch nur teilweise. Während die Identifikation von Crosscutting Concerns in den verschiedenen Ansätzen ausführlich beschrieben wird, zeigen sich bei der Darstellung, Beschreibung und Validierung noch Lücken. Diese versucht dieses Kapitel zu schliessen.

Dem ist beizufügen, dass die Separation of Concerns auch bei konventionellen Anforderungsspezifikationen teilweise bereits vollzogen wird: Benutzt man die Kapitelstruktur aus Sommerville, Sawyer (1997) oder Glinz (2003a) als Checkliste, ist dort eine Trennung von funktionalen und nicht-funktionalen Anforderungen vorgegeben, wobei die nicht-funktionalen Anforderungen durchaus im Sinne von Crosscutting Concerns verstanden werden können. Demgegenüber wird die Separation of Concerns in konventionellen Modellierungssprachen wie UML völlig ignoriert. Will

man in der Anforderungsphase eine vollständige Separation of Concerns erreichen, gilt es also besonderes Augenmerk auf die Modellierungssprachen zu richten.

Konventionelle Anforderungsphase, aspektorientierte Entwurfs- und Implementierungsphase. Die Entwicklung von Ansätzen der aspektorientierten Anforderungstechnik steckt bekanntlich noch in den Kinderschuhen. Somit ist anzunehmen, dass die Anforderungsphase aufgrund fehlender Möglichkeiten oft noch konventionell durchgeführt wird, auch wenn die Crosscutting Concerns später aspektorientiert implementiert werden. Dies hat zur Folge, dass zwischen der Anforderungs- sowie der Entwurfs- und Implementierungsphase ein Strukturbruch zu überwinden ist: Die „scattered" und „tangled" Crosscutting Concerns sind in der Entwurfs- und Implementierungsphase zu identifizieren und als sauber modularisierte Aspekte zu implementieren. Damit werden die folgenden im Abschnitt 9.11 erwähnten Vorteile der aspektorientierten Programmierung preisgegeben:

- **(Rück-) Verfolgbarkeit**. Die Information, welche Module welche Concerns implementieren, ist schwierig zu bewahren bzw. geht ganz verloren.
- **Wiederverwendung**. Die Wiederverwendung wird insofern eingeschränkt, dass allfälliges Wiederverwendungspotenzial von Crosscutting Concerns in der Anforderungsphase und zu Beginn der Entwurfsphase nicht erkennbar ist.
- **Time-to-market**. Die Separation of Concerns wird in die Entwurfs- und Implementierungsphase verlagert und nimmt dort zusätzlich Zeit in Anspruch. Es ist anzunehmen, dass sich dadurch die Entwicklungszeit insgesamt erhöht.
- **Kosten**. Da die Entwicklung länger dauert, erhöhen sich auch die Entwicklungskosten.
- **Pflegbarkeit**. Alle anderen Vorteile der aspektorientierten Programmierung bleiben zwar prinzipiell erhalten, werden jedoch oft *nicht systematisch, sondern zufällig* erreicht, nämlich dann, wenn die Entwickler zufälligerweise einen Aspekt entdecken und ihn entsprechend implementieren. Damit wird auch die spätere Pflegbarkeit der Software beeinträchtigt.

Aspektorientierte Anforderungsphase, konventionelle Entwurfs- und Implementierungsphase. Ganz anders sieht es im umgekehrten Fall aus: Eine aspektorientierte Anforderungsphase mit einer frühen Separation of Concerns kann durchaus auch sinnvoll sein, wenn das System später konventionell implementiert wird. Auch ohne aspektorientierte Implementierung beeinflusst die Separation of Concerns nämlich die Qualität der Anforderungsspezifikation, des Spezifikationsprozesses und der Software positiv:

- **Vollständigkeit**. Verschiedene Concerns beleuchten ein System aus verschiedenen Blickwinkeln. Dadurch erhöht sich die Vollständigkeit der Anforderungsspezifikation.
- **Prüfbarkeit**. Die Separation of Concerns erleichtert die Validierung der Anforderungsspezifikation. Warum dies so ist, wird im Abschnitt 13.6 erklärt.
- **Effizienz**. Die Separation of Concerns erleichtert die Arbeitsteilung und erhöht dadurch die Effizienz des Spezifikationsprozesses.
- **Verständlichkeit und Kundenorientierung**. Die Arbeitsteilung ermöglicht eine fachliche Spezialisierung. Dies kann die Kundenorientierung und die Verständlichkeit der Anforderungsspezifikation positiv beeinflussen.
- **Evolution**. Die Separation of Concerns in der Anforderungsphase erlaubt die Priorisierung von Concerns und unterstützt damit die Anwendung eines inkrementellen Vorgehensmodells (mehr davon im Abschnitt 14.1).
- **Portabilität**. Die Separation of Concerns in der Anforderungsphase erleichtert eine allfällige spätere Portierung von einer konventionellen in eine aspektorientierte Umgebung (siehe auch im Abschnitt 14.6).
- **Modularisierung**. Die Separation of Concerns in der Anforderungsphase ermöglicht es, auch mit einer konventionellen Programmiersprache viele Ideen der aspektorientierten Pro-

grammierung umzusetzen. Scattering und Tangling können durch eine gute Modularisierung bis zu einem gewissen Grad vermieden oder zumindest gekennzeichnet werden; die Server-Teile von Crosscutting Concerns sind nach allen Regeln der Kunst modularisierbar.

Grenzen der frühen Separation. Aus der Sicht des Auftraggebers ist eine konsequente frühe Separation auch mit Nachteilen verbunden: Zum einen dürften streng separierte Concerns für den Auftraggeber abstrakt und nicht leicht zu verstehen sein. Zum andern führt eine frühe Separation dazu, dass er sein System in den Modellen und Dokumenten der Anforderungsphase nirgends mehr als Ganzes sieht und beurteilen kann. Es gibt verschiedene Möglichkeiten, diesem Nachteil zu begegnen. Eine Möglichkeit ist, die Separation of Concerns zwar zu vollziehen, die Concerns aber in den Modellen, welche die Benutzersicht darstellen (z.B. UML-Anwendungsfalldiagramme, Szenarien), gemeinsam darzustellen. Weitere Möglichkeiten sind Prototyping und Simulationen.

13.1.2 Späte Integration

Einer der wesentlichen Unterschiede zwischen der aspektorientierten und der konventionellen Entwicklung liegt im *Zeitpunkt der Integration* von Core und Crosscutting Concerns bzw. Komponenten und Aspekten. Bei der aspektorientierten Entwicklung erfolgt die Integration erst spät, nämlich frühestens in der Testphase, entweder beim Übersetzen (statisches Weaving) oder zur Laufzeit (dynamisches Weaving). Die Integration wird also weder in der Anforderungs- noch in der Entwurfs- und Implementierungsphase *vollzogen*, sondern lediglich *spezifiziert*. Dies steht im Gegensatz zur konventionellen Entwicklung, wo die Integration spätestens in der Entwurfsphase *vollzogen* werden muss, da keine aspektorientierte Programmiersprache zur Verfügung steht.

Wie die Abbildung 13-1 zeigt, findet die Integration bei der aspektorientierten Entwicklung zu einem deutlich späteren Zeitpunkt im Entwicklungsprozess statt. Oder mit anderen Worten: Die *Separation of Concerns* wird über die Entwurfs- und Implementierungsphase hinaus bewahrt, so dass sich all ihre Vorteile auch in den Entwürfen und im Quellcode manifestieren. Das ist der entscheidende Nutzen der späten Integration.

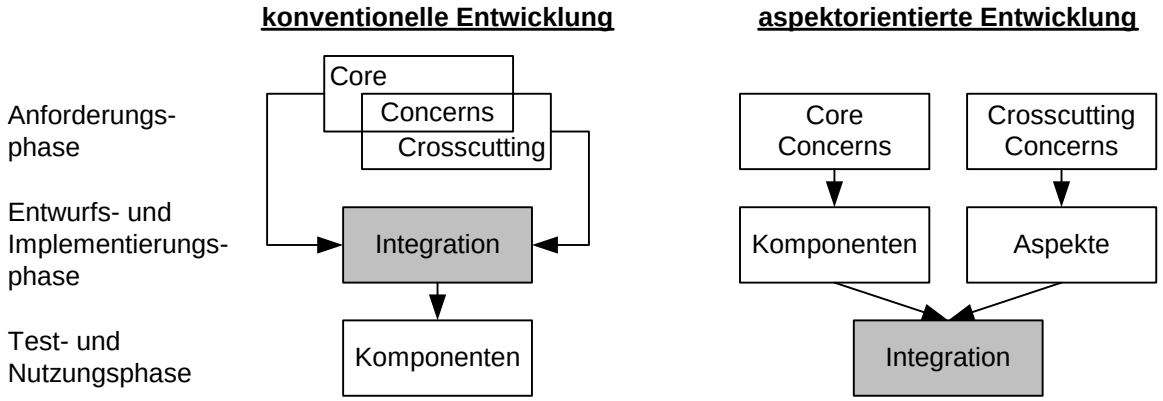


Abbildung 13-1: Integrationszeitpunkt bei konventioneller und aspektorientierter Entwicklung

In der Entwurfs- und Implementierungsphase wird der Entscheid, ob Komponenten und Aspekte getrennt werden können oder integriert werden müssen, ganz wesentlich durch die Möglichkeiten der jeweiligen Programmiersprache bestimmt. Selbstverständlich lassen sich separierte Core und Crosscutting Concerns auch bereits in der Entwurfs- und Implementierungsphase integrieren und damit konventionell implementieren.

13.2 Symmetrisches und asymmetrisches Paradigma

Es gibt nach Harrison, Ossher, Tarr (2002) zwei unterschiedliche Paradigmen für die aspektorientierte Entwicklung. Da es sich um grundlegend andere Denkweisen handelt, wirkt sich die Wahl des Paradigmas auf den gesamten Entwicklungsprozess, also auch auf die Anforderungsphase, aus.

Das **symmetrische Paradigma** macht keinen Unterschied zwischen Core Concerns und Crosscutting Concerns (bzw. zwischen Komponenten und Aspekten). Alle Concerns sind gleichwertig und werden als eigenständige, unabhängige Subsysteme betrachtet. Folglich werden sowohl die Core Concerns als auch die Crosscutting Concerns mit den gleichen Mitteln spezifiziert, entworfen und implementiert. Die Integration der Concerns erfolgt mittels so genannter *Kompositionsbeziehungen*. Vertreter des symmetrischen Paradigmas sind die Subjektorientierung, die MDSOC und Hyper/J.

Das **asymmetrische Paradigma** geht davon aus, dass die Core Concerns (bzw. Komponenten) zusammen ein Basismodell bilden, in das die Crosscutting Concerns (bzw. Aspekte) integriert werden. Dies bedeutet, dass zwischen den Core Concerns und den Crosscutting Concerns ein struktureller Unterschied gemacht werden muss, d.h. sie werden mit unterschiedlichen Mitteln spezifiziert, entworfen und implementiert. Alle anderen im Teil II beschriebenen Ansätze sind Vertreter des asymmetrischen Paradigmas.

Vergleich. Das asymmetrische Paradigma hat den Vorteil, dass es einfacher zu verstehen und anzuwenden ist als das symmetrische, wenn es darum geht, das Verhalten eines gegebenen Basismodells an bestimmten Punkten zu erweitern. In solchen Fällen besteht beim symmetrischen Paradigma die Gefahr, dass der Überblick über das Gesamtsystem verloren geht. Die Vorteile liegen auf der Seite des symmetrischen Paradigmas, wenn es darum geht, ganze Subsysteme einzufügen, auszulagern oder wiederzuverwenden. Wenn sich die Core Concerns ändern, müssen beim asymmetrischen Paradigma unter Umständen die Crosscutting Concerns angepasst werden, beim symmetrischen Paradigma schlimmstenfalls die Kompositionsbeziehungen.

13.3 Identifikation von Crosscutting Concerns

Wie erwähnt beschreiben die Ansätze der aspektorientierten Anforderungstechnik die Identifikation von Crosscutting Concerns ausführlich. Zusammenfassend sind in vielen Ansätzen (z.B. PREview, AORE) die folgenden Schritte auszumachen:

1. **Identifikation der Concerns**. Die Concerns werden identifiziert und beschrieben, beispielsweise mit Hilfe von Anwendungsfällen, Viewpoints, Zielen, Problem Frames oder Wissenskatalogen aus dem NFR-Framework (siehe im Kapitel 12).
2. **Identifikation der Crosscutting Concerns**. Als Crosscutting Concerns werden diejenigen Concerns identifiziert, welche die Core Concerns oder andere Crosscutting Concerns quer schneiden.
3. **Spezifikation der Integrationsregeln**. Die Regeln zur Integration der Crosscutting Concerns in die Core Concerns werden definiert.
4. **Konfliktbehandlung**. Ergeben sich aus den Crosscutting Concerns Widersprüche oder Konflikte (z.B. Sicherheit vs. Antwortzeit), so sind diese mit dem Auftraggeber und den Beteiligten zu klären und durch Priorisierung oder andere geeignete Massnahmen (z.B. Beschaffung zusätzlicher Hardware) aufzulösen.

13.4 Darstellung und Beschreibung von Crosscutting Concerns

In diesem Abschnitt wird ein Vorschlag zur Darstellung und Beschreibung von Crosscutting Concerns gemacht, der Konzepte aus den Ansätzen in den Kapiteln 10 und 12 kombiniert. Angesichts der Tatsache, dass es sich bei der UML um einen universellen, breit akzeptierten und erweiterbaren Standard handelt, wird die UML als Grundlage für die Darstellung und Beschreibung von Crosscutting Concerns benutzt. Dabei wird versucht, die Änderungen an der UML minimal zu halten.

Bevor detailliert auf mögliche Darstellungen und Beschreibungen von Crosscutting Concerns eingegangen werden kann, sind noch zwei Fragen zu klären: Wie werden die Core Concerns dargestellt und beschrieben? Und welche Anforderungen sind an die Darstellung und Beschreibung von Crosscutting Concerns zu stellen?

13.4.1 Darstellung und Beschreibung von Core Concerns

Auf die Darstellung und Beschreibung von Core Concerns wird hier nicht detailliert eingegangen. Es wird angenommen, dass sie mittels Standard-UML-Diagrammen (z.B. Anwendungsfall-, Sequenz-, Klassendiagramm etc.) dargestellt und ergänzend mit Prosatext oder Tabellen beschrieben werden. Die Tatsache, dass daneben noch Crosscutting Concerns existieren, hat keinen Einfluss auf die Darstellung und Beschreibung der Core Concerns.

13.4.2 Anforderungen an die Darstellung und Beschreibung von Crosscutting Concerns

Die Darstellung und Beschreibung von Crosscutting Concerns muss (wie auch die Darstellung und Beschreibung von Core Concerns) den Bedürfnissen zweier Personen(-gruppen), dem Auftraggeber und den Entwicklern, gerecht werden.

- Der **Auftraggeber** hat das Bedürfnis, die Darstellung und Beschreibung der Crosscutting Concerns nachvollziehen zu können. Es möchte möglichst selbständig und ohne grossen Schulungsaufwand prüfen können, ob er richtig verstanden wurde, ob also die modellierten Crosscutting Concerns und ihre Integration in die Core Concerns seinen Anforderungen entsprechen. Dies erhöht die *Effektivität* der Entwicklung: Die Wahrscheinlichkeit steigt, dass das fertige System später seinen Zweck erfüllen wird.
- Der **Entwickler** hat das Bedürfnis, die Crosscutting Concerns möglichst ohne Strukturbrüche entwerfen und implementieren zu können. Die Darstellung und Beschreibung der Crosscutting Concerns soll also, auf einer höheren Abstraktionsebene, Modellkonstrukte verwenden, die sich später möglichst nahtlos – im Idealfall sogar automatisch – auf die gewählte Implementierungssprache abbilden lassen. Dies erhöht die *Effizienz* der Entwicklung: Die Produktivität steigt, und damit sinken die Time-to-market und die Kosten. Ausserdem erleichtert eine nahtlose Abbildung die (Rück-)Verfolgbarkeit zwischen den Anforderungen und der Implementierung und damit die Pflege des fertigen Systems.

Um einen Crosscutting Concern umfassend darstellen und beschreiben zu können, wird er aus zwei verschiedenen Perspektiven betrachtet, einerseits aus der Perspektive der Struktur, der *statischen Sicht* (sie dient vor allem als Grundlage für die nachfolgende Entwurfs- und Implementierungsphase), und anderseits aus der Perspektive des Verhaltens, der *dynamischen Sicht* (sie dient vor allem zur Kommunikation mit dem Auftraggeber).

Die **statische Sicht** stellt die Struktur des Crosscutting Concern dar. Die Tabelle 13-1 zeigt, welche Elemente zur Beschreibung der statischen Sicht eines Crosscutting Concern gehören.

Beschreibungselemente	Analogie: USDP-Anpassung	Analogie: AspectJ
Name des Crosscutting Concern	Name der aspektbezogenen Klasse	Name des Aspekts
Bezeichnung des „crosscutting“ Verhaltens	Spalte „crosscutting Verhalten“ in der Kompositionstabelle	Advice-Signatur
Operationen der Core Concerns, die vom „crosscutting“ Verhalten betroffen sind	Spalte „betroffene Punkte“ in der Kompositionstabelle	Pointcuts
Klassen der Core Concerns, die das „crosscutting“ Verhalten unterstützen	Klassen der Core Concerns	Komponenten-Klassen
für das „crosscutting“ Verhalten benötigtes Wissen	Attribute	Variablen

Tabelle 13-1: Beschreibung der statischen Sicht eines Crosscutting Concern

Dazu ein Beispiel: Das „crosscutting“ Verhalten eines Logging-Concern besteht aus dem Eintragen einer bestimmten Information in ein Logbuch. Zu diesem Zweck muss beschrieben werden, welche Information für welche Operationen der Core Concerns ins Logbuch einzutragen ist, und ausserdem muss ein Logbuch zur Verfügung stehen.

Die **dynamische Sicht** stellt das Verhalten des Crosscutting Concern und die Interaktionen zwischen dem Crosscutting Concern und den Core Concerns dar. Das Verhalten des Crosscutting Concern muss ins Verhalten der Core Concerns eingewoben werden. Zur Beschreibung der dynamischen Sicht eines Crosscutting Concern gehören die Elemente aus der Tabelle 13-2.

Beschreibungselemente	Analogie: USDP-Anpassung	Analogie: AspectJ
Verhalten der Core Concerns	Anwendungsfalldiagramm, Kollaborationsdiagramm	Kontrollflüsse
„crosscutting“ Verhalten	Anwendungsfalldiagramm	Advice
Anwendungsfälle und Operationen der Core Concerns, die vom „crosscutting“ Verhalten betroffen sind	Spalten „betroffener Anwendungsfall“ bzw. „betroffene Punkte“ in den Kompositionstabellen	Pointcuts
zeitliche Beziehung zwischen dem Verhalten der Core Concerns und dem „crosscutting“ Verhalten	Spalte „überlappt, überlagert, umhüllt“ in der Kompositionstabelle	Schlüsselwörter before, after, around

Tabelle 13-2: Beschreibung der dynamischen Sicht eines Crosscutting Concern

Ein Beispiel: Die dynamische Sicht des Logging-Concern aus dem vorhergehenden Abschnitt zeigt, an welchen Stellen das Verhalten des Logging-Concern in das Verhalten der von ihm betroffenen Anwendungsfälle und Operationen einzuweben ist.

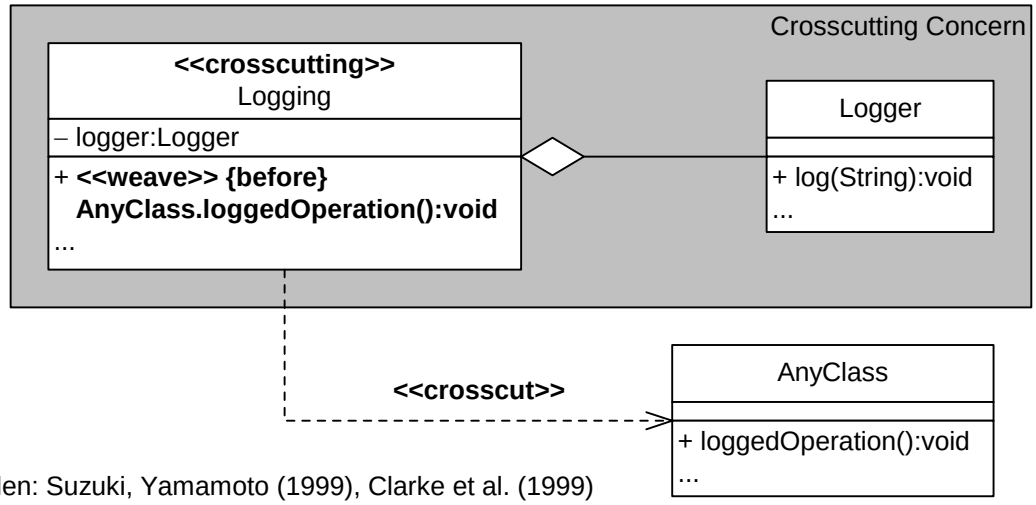
Im Bereich der Operationen der Core Concerns, die vom „crosscutting“ Verhalten betroffen sind, überschneiden sich die statische und die dynamische Sicht. Dies ist gewollt. Einerseits ist diese Information in beiden Sichten erforderlich, damit sie vollständig sind, anderseits erlaubt diese Überschneidung, die statische und die dynamische Sicht miteinander zu verknüpfen.

13.4.3 Darstellung und Beschreibung der statischen Sicht

Als Darstellungs- und Beschreibungsmittel für die statische Sicht von Crosscutting Concerns werden UML-Klassendiagramme vorgeschlagen, da sie bestens bewährt sind und sich als Grundlage für die spätere Entwurfs- und Implementierungsphase eignen. Die gewählten Modellkonstrukte unterscheiden sich abhängig davon, ob das symmetrische oder das asymmetrische Paradigma angewendet wird. Am Ende des Abschnitts werden Alternativen zum Klassendiagramm erwähnt.

Symmetrisch. Wie erwähnt, werden Core und Crosscutting Concerns als gleichwertig betrachtet und mit den gleichen Modellkonstrukten dargestellt und beschrieben. Für die Darstellung und Beschreibung von Crosscutting Concerns stehen keine eigenen Modellkonstrukte zur Verfügung. Jeder Concern, egal ob Core oder Crosscutting, wird als eigenständiges Subsystem betrachtet und mit einem eigenen, unabhängigen Klassendiagramm dargestellt und beschrieben.

Asymmetrisch. Für die Modellierung der statischen Sicht der Core Concerns sind Klassendiagramme gegeben. Für die Crosscutting Concerns müssen zusätzliche Modellkonstrukte gefunden werden. Aufgrund der strukturellen Ähnlichkeit zwischen Core und Crosscutting Concerns (sowohl für Core Concerns als auch für Crosscutting Concerns sind Attribute, Beziehungen und Operationen festzuhalten) wird vorgeschlagen, die Crosscutting Concerns ins Klassendiagramm zu integrieren und als *„crosscutting" Klassen* mit dem Stereotyp `<<crosscutting>>` darzustellen, wie es Sousa et al. (2004) vorschlagen. Ein Logging-Concern würde beispielsweise wie in der Abbildung 13-2 dargestellt, wobei die gegenüber einem Standard-UML-Klassendiagramm erforderlichen Änderungen fett gedruckt sind.



Quellen: Suzuki, Yamamoto (1999), Clarke et al. (1999)

Abbildung 13-2: Darstellung eines Crosscutting Concern im UML-Klassendiagramm

Die Operationen der „crosscutting" Klasse enthalten das „crosscutting" Verhalten, im obigen Beispiel den Aufruf der `log`-Operation der `Logger`-Klasse. Der Operationsrumpf ist im Klassendiagramm – wie bei gewöhnlichen Operationen – nicht dargestellt. Wie im Abschnitt 9.3 beschrieben, ist einer der wesentlichen Unterschiede zwischen Methoden und Advice, dass Advice nicht aktiv aufgerufen werden können und daher auch keine Namen haben. Genau so verhält es sich mit den „crosscutting" Operationen, deren Signatur wie folgt definiert wird (in EBNF):

```
"<<weave>> {"("before"¦"after"¦"around")"} "Operation[{, Operation}]
```

Das Stereotyp `<<weave>>` bedeutet, dass es sich um eine „crosscutting" Operation handelt. Mit dem Ausdruck `("before"¦"after"¦"around")` wird definiert, in welcher zeitlichen Beziehung die Ausführung der „crosscutting" Operation zur Ausführung der gewöhnlichen Operationen steht. Die Semantik entspricht derjenigen von AspectJ, könnte aber bei Bedarf angepasst werden. Im obigen Beispiel zeigt `{before}`, dass die „crosscutting" Operation *vor* den gewöhnlichen Operationen ausgeführt wird. Die „crosscutting" Operation kann nicht aktiv aufgerufen werden, sondern muss in die im Ausdruck `Operation[{, Operation}]` deklarierten gewöhnlichen Operationen eingewoben werden. Im obigen Beispiel wird die „crosscutting" Operation in eine einzige Operation `AnyClass.loggedOperation():void` eingewoben. In den Operationssignaturen können wie in AspectJ auch Wildcards (*) verwendet werden. Um die Übersicht zu bewahren, können die `<<crosscut>>`-Beziehungen in solchen Fällen weggelassen werden.

Die `Logger`-Klasse und ihre Aggregationsbeziehung zur „crosscutting“ `Logging`-Klasse zeigen, dass nicht der gesamte Crosscutting Concern mit „crosscutting“ Klassen spezifiziert werden muss. Der `Logger` hat selber keinerlei „crosscutting“ Eigenschaften und wird daher als gewöhnliche Klasse modelliert.

Die *Benutzt-Abhängigkeitsbeziehung* mit dem Stereotyp `<<crosscut>>` zeigt, dass die „crosscutting“ `Logging`-Klasse die gewöhnliche `AnyClass`-Klasse quer schneidet. Laddad (2003) verwendet ebenfalls eine Benutzt-Abhängigkeitsbeziehung mit dem Stereotyp `<<aspects>>`, um die Beziehung zwischen einem Aspekt und einer Komponente darzustellen. Suzuki, Yamamoto (1999) hingegen schlagen zur Darstellung des gleichen Sachverhalts eine *Abstraktions-Abhängigkeitsbeziehung* mit dem Stereotyp `<<realize>>` in umgekehrter Pfeilrichtung vor; der Pfeil würde also von `AnyClass` zu `Logging` zeigen. Über die Pfeilrichtung kann man geteilter Meinung sein. Nach Oestereich (2001) zeigt der Pfeil in einer Abhängigkeitsbeziehung vom abhängigen auf das unabhängige Element. Ob die „crosscutting“ Klasse von den gewöhnlichen Klassen abhängt oder umgekehrt, ist eine Ermessensfrage:

- Man kann argumentieren, die „crosscutting“ Klasse sei von den gewöhnlichen Klassen abhängig, weil die gewöhnlichen Klassen unabhängig von der „crosscutting“ Klasse existieren können, während die Existenz der „crosscutting“ Klasse nur im Zusammenspiel mit den gewöhnlichen Klassen sinnvoll ist. Dies ergibt die Pfeilrichtung in der Abbildung 13-2.
- Man kann aber auch argumentieren, die gewöhnlichen Klassen seien von der „crosscutting“ Klasse abhängig, um ihr gesamtes Verhalten zu realisieren. Dies ergibt die Pfeilrichtung von Suzuki, Yamamoto (1999)

„Crosscutting“ Klassen können gewöhnliche Attribute (wie im obigen Beispiel `logger:Logger`) und Operationen haben, Interfaces implementieren und von anderen „crosscutting“ Klassen erben. Daher können sämtliche für Klassen üblichen Beziehungen auch für „crosscutting“ Klassen verwendet werden.

Mögliche Alternativen zu den „crosscutting“ Klassen als Darstellungs- und Beschreibungsmittel für Crosscutting Concerns sind:

- Eine **tabellarische Darstellung**, wie sie in der USDP-Anpassung (siehe im Abschnitt 12.6) vorgeschlagen wird. Dies empfiehlt sich besonders, wenn keine Werkzeuge zur Darstellung und Beschreibung von Crosscutting Concerns zur Verfügung stehen, sowie bei grösseren Systemen, bei denen das Klassendiagramm rasch unübersichtlich wird.
- Die **AML** auf einer etwas höheren Abstraktionsebene als im Abschnitt 10.6. Da die AML zur Separation of Concerns UML-Pakete benutzt und die Beziehungen zwischen Core und Crosscutting Concerns auf Paketebene darstellt, wird die Darstellung auch bei grösseren Systemen nicht so schnell unübersichtlich. Bemerkenswert ist, dass die AML die Separation of Concerns so vorbildlich umsetzt, dass sie auch als Darstellungsmittel für das symmetrische Paradigma in Frage kommt.
- **(Pseudo-) AspectJ**. Aspekte werden auf einer höheren Abstraktionsebene spezifiziert, z.B. mit Pointcuts und Advice-Signaturen, jedoch ohne Advice-Rümpfe. Die Pointcuts entsprechen dem Ausdruck `Operation [{, Operation}]` einer „crosscutting“ Operation, die Advice-Signaturen entsprechen dem Rest der Signatur der „crosscutting“ Operation. Dabei geht allerdings die Sprachneutralität verloren, so dass sich diese Alternative nur empfiehlt, wenn AspectJ auch als Implementierungssprache verwendet wird. Dafür ist dann die Abbildung der Crosscutting Concerns auf AspectJ besonders einfach. Zudem muss man sich bewusst sein, dass eine Validierung durch den Auftraggeber ohne vorgängige Schulung in AspectJ (namentlich in der Pointcut-Syntax) kaum in Frage kommt. Das gleiche würde für eine allfällige Beschreibung mit der XML gelten, wie sie in AORE vorgeschlagen wird (siehe im Abschnitt 12.2).

13.4.4 Darstellung und Beschreibung der dynamischen Sicht

Als Darstellungs- und Beschreibungsmittel für die dynamische Sicht von Crosscutting Concerns werden UML-Anwendungsfalldiagramme, UML-Sequenzdiagramme und strukturierter Text vorgeschlagen. Die Anwendungsfalldiagramme zeigen die Anwendungsfälle im Überblick, mit den Sequenzdiagrammen und mit dem strukturierten Text werden einzelne wichtige Anwendungsfälle im Detail spezifiziert. Auch bei der dynamischen Sicht spielt die Unterscheidung zwischen dem symmetrischen und dem asymmetrischen Paradigma eine Rolle.

Symmetrisch. Wiederum wird zwischen Core und Crosscutting Concerns kein Unterschied gemacht. Sie werden mit den gleichen Modellkonstrukten dargestellt und beschrieben. Für die Darstellung und Beschreibung von Crosscutting Concerns stehen keine eigenen Modellkonstrukte zur Verfügung. Jeder Concern, egal ob Core oder Crosscutting, wird als eigenständiges Subsystem betrachtet und mit eigenen, unabhängigen Anwendungsfall- und Sequenzdiagrammen dargestellt und beschrieben.

Hingegen werden zusätzliche Modellkonstrukte zur Darstellung und Beschreibung der Regeln benötigt, nach denen die Concerns integriert werden. Aufgrund der Gleichwertigkeit der Concerns ist es nicht angezeigt, zu diesem Zweck die UML-Diagramme um Integrationsregeln zu erweitern. Das Ziel der symmetrischen Ansätze ist es ja gerade, die Concerns unabhängig voneinander zu modellieren. Am besten werden die Integrationsregeln in einer zusätzlichen, Hyper/J-ähnlichen Syntax ausgedrückt, auch hier mit dem Nachteil, dass die Sprachneutralität verloren geht und dass die Ergebnisse durch den Auftraggeber ohne vorgängige Schulung nicht validiert werden können.

Asymmetrisch. Für die Modellierung der Interaktionen der Core Concerns untereinander sind Anwendungsfall- und Sequenzdiagramme gegeben. Für die Interaktionen der Core Concerns mit den Crosscutting Concerns müssen zusätzliche Modellkonstrukte gefunden werden. Wie schon bei der statischen Sicht werden auch bei der dynamischen Sicht die Crosscutting Concerns in die UML-Diagramme der Core Concerns integriert.

Was die **Anwendungsfalldiagramme** betrifft, so werden die Crosscutting Concerns, wie Sousa et al. (2004) vorschlagen, als eigene Anwendungsfälle dargestellt und mit einer `<<crosscut>>`-Beziehung ins Anwendungsfalldiagramm der Core Concerns eingefügt. Die Abbildung 13-3 zeigt ein Beispiel, wobei die gegenüber einem Standard-Anwendungsfalldiagramm erforderliche Änderung fett gedruckt ist.

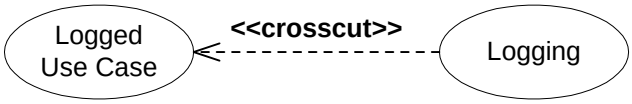


Abbildung 13-3: Darstellung eines Crosscutting Concern im UML-Anwendungsfalldiagramm

Die Frage, warum nicht `<<include>>` oder `<<extend>>` für den Einbezug von „crosscutting“ Anwendungsfällen verwendet werden, lässt sich wie folgt beantworten:

- **`<<include>>`** kommt grundsätzlich nicht in Frage. Nach Jacobson (2002) würde `<<include>>` bedeuten, dass der „crosscutting“ Anwendungsfall unter Kontrolle des gewöhnlichen Anwendungsfalls abläuft. In einer konventionellen Umgebung mag dies zutreffen, in einer aspektorientierten ist das Gegenteil der Fall. Die Pfeilrichtung von `<<include>>` sendet also ein falsches Signal aus. Dies zeigt die Abbildung 13-4.

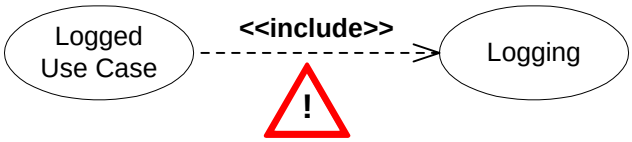
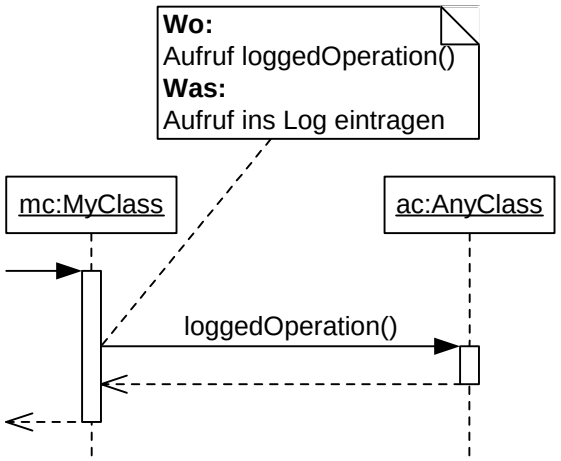


Abbildung 13-4: <<include>> im Anwendungsfalldiagramm

- **<<extend>>** kommt grundsätzlich in Frage. Jacobson (2003) schlägt sogar ausdrücklich vor, <<extend>> für die Darstellung von „crosscutting“ Anwendungsfällen zu verwenden. <<extend>> ist nichts anderes als die Umkehrung von <<include>> und besagt, dass der gewöhnliche Anwendungsfall unter Kontrolle des „crosscutting“ Anwendungsfalls erweitert wird. Dies entspricht der aspektorientierten Realität.
- **<<crosscut>>**. Sousa et al. (2004) kommen zum Schluss, weder <<include>> noch <<extend>> zu verwenden, da beide bereits mit einer Bedeutung behaftet sind. Aus diesem Grund schlagen sie eine unverbrauchte Beziehung mit einer präzisen Semantik für die aspektorientierte Entwicklung vor.

Um die Anwendungsfälle detailliert zu beschreiben, können Sequenzdiagramme oder strukturierter Text verwendet werden. Für einfache, benutzerorientierte Interaktionen eignet sich strukturierter Text bestens; für kompliziertere Interaktionen sollten Sequenzdiagramme eingesetzt werden, da sie eine grössere Ausdruckskraft und Präzision besitzen.

In den **Sequenzdiagrammen** können die Crosscutting Concerns mit Hilfe von UML-Notizen dargestellt werden. Ein Beispiel zeigt die Abbildung 13-5.



Quelle: Laddad (2003)

Abbildung 13-5: Darstellung eines Crosscutting Concern im Sequenzdiagramm

Sobald die `loggedOperation` des `AnyClass`-Objekts aufgerufen wird, soll dieser Aufruf ins Logbuch eingetragen werden.

Es wäre auch möglich, die Crosscutting Concerns als zusätzliche Objekte in die Sequenzdiagramme zu integrieren, ähnlich wie dies in den Abbildungen 9-6 und 9-7 gemacht wurde. Allerdings kann der Kontrollfluss nicht sauber dargestellt werden: Ähnlich wie bei der <<include>>-Beziehung im Anwendungsfalldiagramm würde die Pfeilrichtung der Nachrichten von und zu den Crosscutting Concerns nicht mit der aspektorientierten Realität übereinstimmen. Es müsste ein neuer Nachrichtentyp (analog zur <<extend>>-Beziehung im Anwendungsfalldiagramm) eingeführt werden.

Als Alternative zu den Sequenzdiagrammen können UML-Kollaborationsdiagramme verwendet werden. Sie sind semantisch äquivalent zu den Sequenzdiagrammen.

Um Crosscutting Concerns mit **strukturiertem Text** in Typszenarien (siehe Glinz, 2003a) beschreiben zu können, müssen diese – neben dem Normalfall und den Ausnahmefällen – um einen neuen Abschnitt „Crosscutting Concerns“ ergänzt werden, wie das Beispiel in der Abbildung 13-6 zeigt. Anstelle von `after` können auch `before` oder `around` mit der gleichen Bedeutung wie in AspectJ verwendet werden.

Typszenario: …

Akteur: …

Normalfall:
1 …
2 Die loggedOperation() wird aufgerufen.
3 …

Ausnahmefälle:
3 …

Crosscutting Concerns:
before 2 Logging: Der Aufruf der loggedOperation() wird ins Logbuch eingetragen.

Quelle: Glinz (2003a)

Abbildung 13-6: Darstellung eines Crosscutting Concern mit strukturiertem Text

Die in diesem Abschnitt gezeigten Darstellungsmöglichkeiten stehen untereinander in Beziehung. Entweder stellen sie eine andere Abstraktionsebene oder eine andere Perspektive des gleichen Sachverhalts dar:

- **Andere Abstraktionsebene**. Die `<<crosscut>>`-Beziehung im Anwendungsfalldiagramm einerseits und die Notizen im Sequenzdiagramm bzw. die Crosscutting Concerns in den Typszenarien anderseits stellen den gleichen Sachverhalt auf unterschiedlichen Abstraktionsebenen dar.
- **Andere Perspektive**. Die „crosscutting“ Operationen im Klassenmodell einerseits und die Notizen im Sequenzdiagramm bzw. die Crosscutting Concerns in den Typszenarien anderseits stellen den gleichen Sachverhalt auf der gleichen Abstraktionsebene, jedoch aus verschiedenen Perspektiven dar. Aufgrund dieser Überlappung können die statische und die dynamische Sicht miteinander verknüpft werden.

13.4.5 Kombinierte Darstellung und Beschreibung der statischen und dynamischen Sicht

Will man die statische und dynamische Sicht kombiniert darstellen und beschreiben, eignen sich die Composition Patterns aus dem Abschnitt 10.2. Hierbei gilt es zu beachten, dass es sich bei den Composition Patterns ursprünglich um eine Entwurfstechnik handelt, die auf einer tieferen Abstraktionsebene anzusiedeln ist.

13.5 Dokumentation von Crosscutting Concerns

Wie erwähnt, ist in vielen konventionellen Anforderungsdokumenten eine gewisse Separation of Concerns durch die Trennung von funktionalen und nicht-funktionalen Anforderungen durch die Kapitelstruktur vorgegeben (siehe Sommerville, Sawyer, 1997, und Glinz, 2003a). Der Schritt zur Einbindung der Anforderungen aus Crosscutting Concerns ist also nicht allzu gross. Das Kapitel 3 „Einzelanforderungen" der Kapitelstruktur wäre wie folgt anzupassen:

Symmetrisch. Kommt das symmetrische Paradigma zur Anwendung, wird zwischen Core und Crosscutting Concerns kein Unterschied gemacht, und es ergibt sich die folgende mögliche Kapitelstruktur:

3. Einzelanforderungen
   - 3.1. Anforderungen aus dem Concern 1
   - 3.2. Anforderungen aus dem Concern 2
   - 3.3. …
   - 3.4. Anforderungen aus dem Concern n
   - 3.5. Kompositionsregeln

In den Abschnitten 3.1 bis 3.n des Anforderungsdokuments werden die Anforderungen aus allen Concerns einzeln und unabhängig voneinander beschrieben, und im Abschnitt 3.n+1 werden schliesslich die Kompositionsregeln spezifiziert.

Asymmetrisch. Kommt das asymmetrische Paradigma zur Anwendung, ist zwischen Core und Crosscutting Concerns zu unterscheiden, was zu folgender möglichen Kapitelstruktur führt:

3. Einzelanforderungen
   - 3.1. Funktionale Anforderungen aus Core Concerns
     - 3.1.1 Anforderungen aus dem Core Concern 1
     - 3.1.2 Anforderungen aus dem Core Concern 2
     - 3.1.3 …
     - 3.1.4 Anforderungen aus dem Core Concern n
   - 3.2. Funktionale Anforderungen aus Crosscutting Concerns (mit entsprechenden Verweisen auf die Core Concerns)
     - 3.2.1 Anforderungen aus dem Crosscutting Concern 1
     - 3.2.2 Anforderungen aus dem Crosscutting Concern 2
     - 3.2.3 …
     - 3.2.4 Anforderungen aus dem Crosscutting Concern n
   - 3.3. Nicht-funktionale Anforderungen aus Crosscutting Concerns (mit entsprechenden Verweisen auf die Core Concerns)
     - 3.3.1 Leistungsanforderungen
     - 3.3.2 besondere Qualitäten
     - 3.3.3 Randbedingungen

Für die funktionalen Anforderungen aus den Abschnitten 3.1 und 3.2 des Anforderungsdokuments können die im Abschnitt 13.4 erwähnten Darstellungs- und Beschreibungsmittel verwendet werden. Die nicht-funktionalen Anforderungen aus dem Abschnitt 3.3 des Anforderungsdokuments sind mit Prosatext oder tabellarisch zu beschreiben.

13.6 Validierung von Crosscutting Concerns

Die Validierung von Crosscutting Concerns besteht aus zwei Teilaufgaben:

- Validierung der statischen Sicht jedes Crosscutting Concern
- Validierung der dynamischen Sicht, d.h. des Verhaltens der Crosscutting Concerns und der Interaktionen zwischen Crosscutting Concerns und Core Concerns

Dafür stehen grundsätzlich zwei verschiedene Möglichkeiten zur Verfügung, Reviews und Simulationen:

Reviews. Die Validierung von Crosscutting Concerns kann konventionell und manuell mittels Reviews erfolgen. Obschon sich am Review-Verfahren grundsätzlich nichts ändert, haben Reviews in der aspektorientierten Entwicklung einen entscheidenden Vorteil gegenüber Reviews in der konventionellen Entwicklung: Die Separation of Concerns erlaubt eine Spezialisierung. Jeder Concern kann von den entsprechenden Fachleuten begutachtet werden, beispielsweise der Sicherheits-Concern von einem Sicherheitsfachmann, der Logging-Concern von einem Revisor etc. Dies erhöht die Effizienz und Effektivität der Reviews beträchtlich. Der Auftraggeber soll selbstverständlich sämtliche Concerns begutachten.

Simulationen sind eine weitere Möglichkeit, Crosscutting Concerns zu validieren. Die im Abschnitt 12.7 beschriebenen Scenario-Based Requirements von Araujo, Whittle, Kim (2004) sind nichts anderes als ein Vorbereitungsschritt zur Simulation von Crosscutting Concerns. Allerdings ist der Aufwand dafür beträchtlich. Was den Aufwand verursacht, ist die Integration von Core und Crosscutting Concerns. Simulationen haben wohl nur dann eine reelle Chance eingesetzt zu werden, wenn sie automatisch aus den Spezifikationen generiert werden können. Dies ist bei den Scenario-Based Requirements leider nur teilweise der Fall, da an einigen Stellen (z.B. beim Instanzieren, beim Mischen) Handarbeit erforderlich ist.

13.7 Architekturentwurf – aus Crosscutting Concerns werden Aspekte

Eine der Hauptaktivitäten des Architekturentwurfs ist nach Glinz (2003a) die Aufteilung des Systems in Module. Da es sich sowohl bei Komponenten als auch bei Aspekten um Module handelt, unterscheidet sich die Architektur eines aspektorientierten Systems grundsätzlich nicht wesentlich von der Architektur eines konventionellen Systems. Das ist wahrscheinlich einer der Gründe, weshalb sich in der Literatur nur ein einziger Ansatz der aspektorientierten Architektur finden liess. Im Vergleich zum konventionellen Architekturentwurf sind beim aspektorientierten Architekturentwurf einige zusätzliche Aktivitäten notwendig:

- **Entscheid über den Einsatz der aspektorientierten Programmierung**. Grundsätzlich ist im Architekturentwurf abschliessend über den Einsatz der aspektorientierten Programmierung zu entscheiden, auch wenn derartige Entscheide meistens bereits in der Anforderungsphase getroffen werden. Dazu gehört natürlich auch die Auswahl der Ansätze und Sprachen des aspektorientierten Entwurfs und der aspektorientierten Programmierung, die eingesetzt werden sollen.
- **Extraktion von „crosscutting" Anforderungen aus der Anforderungsspezifikation**. Sofern das System nicht bereits aspektorientiert spezifiziert wurde, müssen zu Beginn des Architekturentwurfs die „crosscutting" Anforderungen aus der Anforderungsspezifikation extrahiert, modelliert und dokumentiert werden. Andernfalls entfällt diese Aktivität.
- **Operationalisierung von nicht-funktionalen Anforderungen**. Im Architekturentwurf erfolgt die Operationalisierung derjenigen Crosscutting Concerns, die in der Anforderungsspezifikation durch nicht-funktionale Anforderungen beschrieben wurden (siehe im Abschnitt 8.2), z.B. mit Hilfe von Softgoal Interdependency Graphs (SIG).
- **Trennung von Aspekten und Komponenten**. Im Architekturentwurf muss entschieden werden, welche Anforderungen mit „crosscutting" Charakter als Aspekte und welche als Komponenten (z.B. mit Entwurfsmustern; siehe im Abschnitt 14.3) realisiert werden. Auch wenn ein System grundsätzlich aspektorientiert entworfen und implementiert werden soll, kann es stichhaltige Gründe geben, nicht alle Crosscutting Concerns aspektorientiert zu implementieren: Unter Umständen gibt es organisationsinterne Richtlinien, die den Einsatz

bestimmter Middleware-Komponenten (z.B. im Rahmen von J2EE[34]) fordern. Vielleicht wird die Aspektorientierung in der Organisation erst gerade eingeführt, man möchte zuerst Erfahrungen mit einfachen Aspekten sammeln und die komplizierten Probleme noch konventionell lösen. Auch Performance-Überlegungen (insbesondere bei reflexiven Ansätzen) können dazu führen, Crosscutting Concerns konventionell zu implementieren.

Nachdem entschieden wurde, einen bestimmten Crosscutting Concern aspektorientiert zu implementieren, wird sein Client-Teil (siehe im Abschnitt 8.2) als Aspekt, sein Server-Teil als Komponente entworfen.

- **Wiederverwendung**. Da sich abstrakte Aspekte besonders für die Wiederverwendung eignen, soll im Architekturentwurf geprüft werden, ob sich – wenn möglich zertifizierte – vorgefertigte abstrakte Aspekte für die Implementierung von „crosscutting“ Anforderungen finden lassen.
- **Teilkonzept für die Aspektorientierung**. Wie im Architekturentwurf für die Datenbank und die Mensch-Maschine-Schnittstelle Teilkonzepte erstellt werden (siehe Glinz, 2003a), empfiehlt es sich, auch für die Aspektorientierung ein Teilkonzept zu erstellen.

13.8 Detailentwurf/Programmierung von Aspekten

Detailentwurf von Aspekten. Wurde im Architekturentwurf entschieden, eine „crosscutting“ Anforderung aspektorientiert zu implementieren, geht es im Detailentwurf darum, die in der Anforderungsphase oder im Architekturentwurf erstellten Modelle auf Entwurfsmodelle abzubilden. Dafür gibt es verschiedene Möglichkeiten. Bei einfachen Aspekten (z.B. Logging) können die Modellkonstrukte der Anforderungsphase direkt auf die ansatzneutralen Sprachkonstrukte der gewählten aspektorientierten Programmiersprache oder des gewählten aspektorientierten Ansatzes abgebildet werden. Die Tabelle 13-3 zeigt als Beispiel, wie die Modellkonstrukte der aspektbezogenen Klassen der USDP-Anpassung (siehe im Abschnitt 12.6) aus der Tabelle 12-3 auf Sprachkonstrukte von AspectJ abgebildet werden können.

Modellkonstrukte der aspektbezogenen Klassen	Sprachkonstrukte von AspectJ
aspektbezogene Klasse	Aspekt
betroffene Klasse	Klasse im Pointcut
statische “crosscutting” Eigenschaft	Aspektvariable, Introduction
Operator der Kompositionsregel	
überlappt.vor	Before Advice
überlappt.nach	After Advice
überlagert	Around Advice ohne proceed()
umhüllt	Around Advice mit proceed()
betroffener Punkt	Pointcut
„crosscutting“ Verhalten	Advice

Tabelle 13-3: Beschreibung der statischen Sicht eines Crosscutting Concern

Bei komplizierten Aspekten (z.B. Autorisierung, Authentisierung, Transaktionsverwaltung) empfiehlt sich die Erstellung eines separaten Entwurfsmodells. Als Ansätze kommen die im Kapitel 10 vorgestellten Ansätze des aspektorientierten Entwurfs in Frage, wobei die Composition Patterns und die AML besonders geeignet scheinen. Beide lassen sich erwiesenermassen auf AspectJ und Hyper/J abbilden. Die AML hat den Vorteil, einfach und übersichtlich zu sein, und eignet sich primär für grosse Systeme, in denen ein Aspekt viele Komponenten quer schneidet (z.B. Autorisierung, Authentisierung). Demgegenüber werden die Composition Patterns bei grossen Systemen

[34] java.sun.com/j2ee/ (22.07.2004)

rasch unübersichtlich, sie eignen sich aber aufgrund der integrierten Sequenzdiagramme besonders für den Entwurf von Aspekten mit komplizierten Interaktionen (z.B. Transaktionsverwaltung).

Generierung von Aspekten. Wie im Abschnitt 8.2 gezeigt wurde, ist eine nahtlose Abbildung von Crosscutting Concerns (der Anforderungsphase) auf Aspekte (der Entwurfs- und Implementierungsphase) nur in vereinzelten Fällen möglich: Einige Crosscutting Concerns (z.B. Performance-Concern) werden überhaupt erst im Architekturentwurf ausreichend konkretisiert, während von anderen Crosscutting Concerns (z.B. Logging-Concern) nur der Client-Teil auf Aspekte abgebildet wird. Eine vollautomatische Generierung von Aspekten im grossen Stil dürfte also eine Illusion sein. Was jedoch sicher denkbar ist, ist die halbautomatische (d.h. durch den Entwickler begleitete) „first cut"-Abbildung einiger Crosscutting Concerns auf Aspekte. In diesem Sinne unterscheiden sich Aspekte nicht von Komponenten. Weiter fortgeschritten sind Überlegungen zur Generierung von Aspekten aus Entwurfsmodellen: Gleichzeitig mit der AML wurde Together[35] um einen Generator ergänzt, der in der Lage ist, „first cut" AspectJ-Code zu erzeugen. Was die Composition Patterns betrifft, wurde immerhin nachgewiesen, dass sich daraus mit minimalen manuellen Eingriffen theoretisch AspectJ- und Hyper/J-Code erzeugen lässt.

Programmierung von Komponenten und Aspekten. Noch ein Wort zur Programmierung von Komponenten und Aspekten: Eine der wichtigsten Voraussetzungen, damit sich das Nutzenpotenzial der Aspektorientierung (siehe im Abschnitt 9.11) entfalten kann, ist äusserste *Disziplin* beim Programmieren der Core Concerns. Dies gilt besonders für linguistische Ansätze mit Weaving (AspectJ, Hyper/J und Composition Filters). Da dort die Integration von Aspekten und Komponenten über die Namen von Paketen, Klassen und Methoden erfolgt, ist es wichtig, dass für die Programmierung der Komponenten verbindliche Namenskonventionen existieren und auch eingehalten werden. Ein Beispiel aus AspectJ: Wenn mit einem `public *.set*(..)`-Pointcut sämtliche öffentlichen zustandsverändernden Methoden identifiziert werden sollen, müssen sich alle Entwickler an die Namenskonvention halten, dass zustandsverändernde Methoden mit `set` beginnen.

13.9 Prüfen von Aspekten

In diesem Abschnitt werden Überlegungen angestellt, was sich beim Prüfen von aspektorientierten Systemen gegenüber konventionellen Systemen ändert. Dabei wird von einer Implementierung mit AspectJ ausgegangen.

13.9.1 Reviews

Was für Crosscutting Concerns gilt (siehe im Abschnitt 13.6), gilt sinngemäss auch für Aspekte: Sie können durch die jeweiligen Spezialisten unabhängig voneinander begutachtet werden (Laddad, 2003). Das erhöht die Effizienz und Effektivität der Reviews.

13.9.2 Tests

Es wird angenommen, dass der Modul- und Integrationstest strukturorientiert, der System- und Abnahmetest funktionsorientiert durchgeführt werden. Die verwendete Testterminologie stammt aus Glinz (2004). Bei der Testplanung für aspektorientierte Systeme ist zu berücksichtigen, dass die schlechte Übereinstimmung zwischen der dynamischen Prozessstruktur und der statischen Programmstruktur (siehe Dijkstra, 1968) die Fehlersuche erschwert.

Der **Modultest** der Komponenten bleibt sich im Prinzip gleich, nimmt aber im Umfang ab, da die Module nicht mehr „tangled" sind und nur noch die Kernfunktionalität enthalten. Das heisst, dass

[35] www.borland.com/together (22.07.2004)

das gesamte Verhalten, das nun in den Aspekten implementiert ist, in den Komponenten nicht mehr getestet werden muss. Vom Modultest der Aspekte sind vor allem die *Advice* betroffen. Sie können wie die Methoden von Komponenten getestet werden, am besten mit Testtreibern (z.B. Testklassen), die einige der Join Points simulieren. Das kann je nach Pointcut umständlich werden, müssen doch die Namen der Testpakete, -klassen und -methoden zu den Pointcuts passen. In Laddad (2003) finden sich diesbezüglich unzählige Beispiele. Anzustreben ist wie üblich Zweigüberdeckung. Die Pointcuts können im Modultest noch nicht ausführlich getestet werden, da die Join Points aufgrund der Namen von Komponentenpaketen, -klassen und -methoden identifiziert werden, die zu diesem Zeitpunkt noch nicht zur Verfügung stehen.

Im **Integrationstest** werden die Komponenten und Aspekte nicht mehr getrennt voneinander getestet, sondern gemeinsam. Es wird davon ausgegangen, dass alle Komponenten und Advice zuvor einen Modultest durchlaufen haben und einwandfrei funktionieren. Im Vergleich zu konventionellen Systemen muss im Integrationstest von aspektorientierten Systemen zusätzlich geprüft werden, ob die Pointcuts tatsächlich die gewünschten Join Points identifizieren. Um in Bezug auf die Ausführung eines Advice *Anweisungsüberdeckung* zu erreichen, genügt ein einziger Testfall, der einen Join Point identifiziert, an dem dieser Advice ausgeführt wird. Dies ist ungenügend, da das Funktionieren der Pointcuts wesentlich von der Namensgebung in allen Komponenten abhängt. Daher ist auch für die Tests der Pointcuts *Zweigüberdeckung* anzustreben. Zu diesem Zweck muss ein Testfall für jeden denkbaren Join Point definiert werden, da der Advice bekanntlich an jedem einzelnen Join Point in die Kernfunktionalität eingewoben wird. (Dies sind die gleichen Testfälle, die auch definiert werden müssten, wenn das gleiche System konventionell implementiert würde).

Soll nicht das Zusammenspiel zwischen Komponenten und Aspekten getestet werden, sondern nur das Funktionieren der Pointcuts, kann dies ebenfalls mit Testtreibern (z.B. Testklassen) erfolgen, welche nicht – wie im Modultest – die Joint Points simulieren, sondern die betroffenen Komponentenmethoden direkt aufrufen. Zu diesem Zweck müssen die Komponenten aber bereits zur Verfügung stehen.

Der **Systemtest** und der **Abnahmetest** bleiben im Vergleich zu einem konventionell implementierten System unverändert. Sie sind funktionsorientiert und hängen folglich nicht davon ab, ob das System konventionell oder aspektorientiert implementiert wurde. Für den *Systemtest* leiten die Entwickler die Testfälle wie gewohnt aus der Anforderungsspezifikation ab. Ist die Separation of Concerns in der Anforderungsspezifikation explizit berücksichtigt, umfassen die Testfälle ganz automatisch auch das „crosscutting“ Verhalten. In diesem Sinne hat eine gute Anforderungsspezifikation auch eine positive Auswirkung auf die Vollständigkeit der Testfälle. Im *Abnahmetest* benutzt der Auftraggeber wie gewohnt seine eigenen Testfälle.

13.10 Messen von Aspekten

Um den Nutzen der aspektorientierten Programmierung gegenüber der konventionellen Programmierung von Aspekten nachzuweisen, sind Masse erforderlich. Sie wurden in der Literatur mit wenigen Ausnahmen bisher nicht ausführlich behandelt. Einige Masse sind jedoch zu finden:

Mass für die Code-Reduktion. Schon Kiczales et al. (1997) definieren das erste Mass für den Nutzen der aspektorientierten Programmierung. Dieses Mass gibt an, um welchen Faktor sich die Anzahl Codezeilen reduzieren lässt, wenn man die Crosscutting Concerns aspektorientiert implementiert. Aus diesem Grund wird das Mass hier *Code-Reduktionsfaktor* genannt, auch wenn es Kiczales et al. (1997) mit „Blähreduktion“ (engl. reduction in bloat) bezeichnen. Ein Code-Reduktionsfaktor von 6 bedeutet, dass ein konventionell implementierter Crosscutting Concern 6-mal mehr Codezeilen benötigt als ein aspektorientiert implementierter. Die Formel lautet:

$$\text{Code-Reduktionsfaktor} = \frac{\text{Grösse des gesamten Code} - \text{Grösse des Komponentencode}}{\text{Grösse des Aspektcode}}$$

Kiczales et al. (1997) haben in Experimenten herausgefunden, dass der Code-Reduktionsfaktor bis zu 98 betragen kann. Er hängt von der Anzahl und der Grösse der Aspekte und von der Anzahl der Komponenten ab, die ein Aspekt quer schneidet. Ein hoher Code-Reduktionsfaktor hat einen positiven Einfluss auf alle Masse, die in irgendeiner Form von der Anzahl Codezeilen abhängen, namentlich auf die Produktivität, die Wirtschaftlichkeit und die Pflege.

Masse für die Modularisierung, Wiederverwendbarkeit, Verständlichkeit und Pflegbarkeit. Zakaria, Hosny (2003) schlagen vor, die folgenden für objektorientierte Systeme definierten Masse von Chidamber, Kemerer (1994) auf aspektorientierte Systeme zu übertragen. Tsang, Clarke, Baniassad (2004) haben diesen Vorschlag übernommen, die entsprechenden Messgrössen in einem aspektorientierten System erhoben und sie mit den Messgrössen eines analogen objektorientierten Systems verglichen. Beide kommen zu ähnlichen Schlüssen:

- **Anzahl Methoden** (engl. weighted methods per class, WMC). Die WMC sagt aus, wie viele Methoden eine Klasse hat[36] und misst damit die Wiederverwendbarkeit, die Verständlichkeit und die Pflegbarkeit. Im aspektorientierten System haben Tsang, Clarke, Baniassad (2004) die Aspekte als Klassen und die Advice als Methoden gezählt. Dabei stellten sie eine leichte Zunahme der WMC fest, während Zakaria, Hosny (2003) von einer Abnahme der WMC ausgehen. Letztlich hängt dies vermutlich vom Entwurf des objektorientierten Systems ab.
- **Tiefe des Vererbungsbaums** (engl. depth of inheritance tree, DIT). Die DIT sagt aus, wie gross die Distanz von einer bestimmten Klasse bis zur Wurzel des Klassenbaums ist, und misst damit die Verständlichkeit und die Wiederverwendbarkeit. Subklassen, die nur existieren, weil sie ein spezielles „crosscutting" Verhalten implementieren, werden in aspektorientierten Systemen zu Aspekten. Für sie nimmt die DIT ab.
- **Anzahl direkter Nachfahren** (engl. number of children, NOC). Die NOC sagt aus, wie viele direkte Subklassen eine bestimmte Klasse hat und misst damit die Wiederverwendbarkeit. Für Klassen, deren Subklassen zu Aspekten werden, nimmt die NOC ab.
- **Kopplung** (engl. coupling between object classes, CBO). Die CBO sagt aus, mit wie vielen anderen Klassen eine bestimmte Klasse durch Methodenaufrufe und Attributzugriffe verbunden ist. Damit misst die CBO die Güte der Modularisierung, die Wiederverwendbarkeit, die Verständlichkeit und die Pflegbarkeit. Im aspektorientierten System haben Tsang, Clarke, Baniassad (2004) nur die Anzahl der explizit (also nicht über Wildcards) referenzierten Methoden berücksichtigt. Dabei stellten sie eine Abnahme der CBO fest. Zakaria, Hosny (2003) kommen zum gleichen Schluss.
- **Anzahl möglicher Methodenaufrufe** (engl. response for a class, RFC). Die RFC sagt aus, wie viele Methoden als Reaktion auf eine Nachricht an ein Objekt einer bestimmten Klasse ausgeführt werden können. Damit misst die RFC die Verständlichkeit und die Pflegbarkeit. Tsang, Clarke, Baniassad (2004) haben im aspektorientierten System die Join Points zur RFC hinzugezählt und dadurch eine deutliche Zunahme der RFC festgestellt.
- **Mangel an Kohäsion** (engl. lack of cohesion in methods, LOCM). Der LOCM sagt aus, ob den Methoden einer bestimmten Klasse die Kohäsion fehlt. Er wird aufgrund der Variablen berechnet, welche jeweils zwei Methoden einer Klasse gemeinsam nutzen. Damit misst der LOCM die Güte der Modularisierung und die Wiederverwendbarkeit. Tsang, Clarke, Baniassad (2004) haben Pointcuts und Advice als Methoden von Aspekten betrachtet und eine Abnahme des LOCM festgestellt. Zakaria, Hosny (2003) kommen zum gleichen

[36] Die Gewichtung (Komplexität der Methoden) wird hier aus Gründen der Einfachheit nicht berücksichtigt.

Schluss und begründen dies damit, dass Aspekte das „crosscutting“ Verhalten aus den Komponenten entfernen, was deren Kohäsion erhöht.

Durch Kombination dieser Masse kommen Tsang, Clarke, Baniassad (2004) zu folgenden Aussagen über das aspektorientierte System, das sie untersucht haben:

- **Modularisierung**. Zur Messung der Modularisierung werden die CBO (Kopplung) und der LOCM (Mangel an Kohäsion) kombiniert. Bei beiden Massen zeigt das Untersuchungsergebnis, dass die Aspektorientierung die Modularisierung tatsächlich verbessert.
- **Wiederverwendbarkeit**. Zur Messung der Wiederverwendbarkeit werden alle Masse mit Ausnahme der RFC kombiniert. Daraus schliessen Tsang, Clarke, Baniassad (2004), dass die Aspektorientierung zu leicht verbesserter Wiederverwendbarkeit führt.
- **Verständlichkeit und Pflegbarkeit**. Zur Messung der Verständlichkeit und Pflegbarkeit werden die WMC, die DIT, die CBO und die RFC kombiniert. Da die RFC in aspektorientierten Systemen deutlich zunimmt, verschlechtert sich die Verständlichkeit und Pflegbarkeit aspektorientierter Systeme nach Tsang, Clarke, Baniassad (2004) insgesamt.

Es wäre nützlich, wenn all diese Masse in Werkzeuge zur Unterstützung der aspektorientierten Programmierung eingebaut werden könnten.

14. Aspektorientierung in konventionellen Software Engineering-Ansätzen

Dieses Kapitel zeigt, wie sich die Aspektorientierung auf bekannte Prinzipien, Konzepte und Methoden des Software Engineering auswirkt.

14.1 Vorgehensmodelle

Crosscutting Concerns und Aspekte lassen sich grundsätzlich mit allen Vorgehensmodellen (z.B. Wasserfallmodell, inkrementelle Modelle, agile Modelle) behandeln. Es gibt aber einige Vorgehensmodelle, die aufgrund ihrer Eigenschaften besonders gut auf den Umgang mit Crosscutting Concerns und Aspekte zugeschnitten sind. Diesen ist dieser Abschnitt gewidmet.

14.1.1 Inkrementelle Modelle

Inkrementelle Modelle zerlegen die Spezifikation, den Entwurf und die Implementierung der Software in eine Folge von *Erweiterungen*, deren Durchführung nacheinander erfolgt (Sommerville, 2001). Zu Beginn der Entwicklung werden die groben Anforderungen festgelegt, und die Systemarchitektur wird entworfen. Ferner wird eine Anzahl von lieferbaren Teilsystemen und Erweiterungen vereinbart, wobei jede Erweiterung eine Teilmenge der Funktionen des Systems umfasst. Die Funktionen mit der höchsten Priorität werden zuerst implementiert und ausgeliefert. Die detaillierten Anforderungen werden jeweils im Rahmen der Entwicklung der Teilsysteme erhoben.

Die Vorteile der inkrementellen Entwicklung liegen auf der Hand: Die wichtigsten Funktionen des Software-Systems stehen schnell zur Verfügung, und die Benutzer können damit erste Erfahrungen sammeln. Das Risiko, dass das Projekt scheitert, wird reduziert. Ändern sich die Anforderungen im Laufe der Zeit, ist das kein Problem. Diesen Vorteilen stehen jedoch gewichtige Nachteile gegenüber: Crosscutting Concerns sind zu Beginn der Entwicklung schwierig zu erkennen, da die Anforderungen nur grob festgelegt werden. Durch die laufenden Erweiterungen besteht die Gefahr, dass sich die Modularisierung im Laufe der Zeit verschlechtert, so dass die Verständlichkeit und Pflegbarkeit leiden.

Die Aspektorientierung kann helfen, diesen Nachteilen zu begegnen, ohne die Vorteile preiszugeben. Mit Hilfe der im Abschnitt 13.3 beschriebenen Ansätze kann versucht werden, die Crosscutting Concerns bereits zu Beginn der Entwicklung systematisch zu identifizieren. Die Priorisierung der (Crosscutting) Concerns erleichtert es, den Umfang für die nächsten Erweiterungen zu definieren. Crosscutting Concerns, die zu Beginn der Entwicklung tatsächlich nicht erkennbar sind, können nachträglich in Form von Aspekten eingefügt werden, im Idealfall additiv, ohne die Komponenten ändern zu müssen. Ausserdem haben aspektorientierte Software-Systeme das Potenzial für gute Verständlichkeit und Pflegbarkeit. Dadurch eignen sie sich besonders für die inkrementelle Entwicklung. Voraussetzung dafür ist allerdings eine gute Modularisierung, die von Beginn weg und über alle Erweiterungen gepflegt wird.

14.1.2 Agile Modelle am Beispiel von Extreme Programming (XP)

Viele Software-Projekte werden verzögert eingeführt oder scheitern ganz, viele Software-Systeme sind fehlerhaft oder für den Auftraggeber unbefriedigend. Beck (2000) versucht, diesen Missständen mit einem völlig neuen Ansatz, dem *Extreme Programming* (XP), zu begegnen. Im Prinzip ist XP nichts anderes als die konsequente Weiterentwicklung der inkrementellen Modelle. Da die Aspektorientierung ganz ähnliche Ziele (Time-to-market, Kosten, Beschränkung auf das Wesentliche) verfolgt wie XP, wird anhand der folgenden Merkmale von XP untersucht, wie gut die Ideen von XP mit den Ideen der Aspektorientierung harmonieren:

Das Planungsspiel. In XP wird der Umfang des nächsten Release anhand von Prioritäten und Aufwandschätzungen in enger Zusammenarbeit zwischen dem Auftraggeber und den Entwicklern festgelegt. Ein solches Vorgehen ist auch für die aspektorientierte Entwicklung von Vorteil. Während es relativ einfach ist, dem Auftraggeber Aussagen über die Core Concerns zu entlocken, bleiben seine Anforderungen und Prioritäten hinsichtlich Crosscutting Concerns ohne hartnäckiges Nachfragen oft im Dunkeln. Durch das Planungsspiel lernen die Entwickler die wahre Bedürfnisse des Auftraggebers besser kennen und einschätzen.

Kurze Releasezyklen. XP zeichnet sich durch kurze Releasezyklen aus. Dies lässt sich gut mit einer aspektorientierten Entwicklung vereinbaren. Die Aspektorientierung verbessert potenziell die Modularisierung und dadurch die Verständlichkeit, Pflegbarkeit und (Rück-) Verfolgbarkeit von Software-Systemen. Aspekte können im Idealfall additiv hinzugefügt/entfernt werden. Diese Merkmale der Aspektorientierung sind eine Voraussetzung für kurze Releasezyklen.

Metapher. XP-Projekte werden durch eine übergreifende Metapher geleitet. Dies ist für die Aspektorientierung weder von besonderem Vorteil noch von besonderem Nachteil.

Einfacher Entwurf. Entwürfe in XP sollen die kleinstmögliche Anzahl von Klassen und Methoden enthalten und namentlich keine Redundanzen aufweisen. Ausserdem ist es verpönt, mögliche zukünftige Anforderungen im Entwurf bereits zu berücksichtigen. In diesem Punkt harmonieren XP und die Aspektorientierung besonders gut, da die Aspektorientierung in idealer Weise die Erfüllung dieser Forderungen unterstützt (siehe auch im Abschnitt 9.11).

Testen. Die Idee von XP besteht darin, dass die Testfälle vor den Programmen erzeugt werden und sämtliche Tests automatisiert sind. In diesem Punkt kann die Aspektorientierung mit Entwicklungsaspekten Unterstützung leisten. Hingegen erschwert die Tatsache, dass die Korrektheit der Aspekte nicht ohne die Komponenten geprüft werden kann, das automatisierte Testen.

Refactoring. XP-Entwickler sollen laufend nach Möglichkeiten suchen, den bestehenden Code zu vereinfachen. Auch das Refactoring wird – wie die kurzen Releasezyklen – durch die aspektorientierte Programmierung erleichtert.

Programmieren in Paaren. Die Entwickler arbeiten grundsätzlich zu zweit. Dies wird durch die Aspektorientierung weder besonders gefördert noch verhindert.

Gemeinsame Verantwortlichkeit. Bei XP sind alle Entwickler gemeinsam für den Code verantwortlich. Wie sinnvoll das ist, ist letztlich eine philosophische Frage und betrifft die Aspektorientierung nicht direkt. Die gemeinsame Verantwortlichkeit widerspricht aber den Aussagen von Laddad (2003), der es gerade als Vorteil der Aspektorientierung betrachtet, dass die Verantwortung für den Code aufgrund der Modularisierung auf einzelne Personen aufgeteilt werden kann. Eine solche Aufteilung der Verantwortung erlaubt eine Spezialisierung und erhöht damit die Produktivität, hingegen ist das Wissen über den Code auf eine Person beschränkt.

Fortlaufende Integration. In XP wird der Code jeweils spätestens am Ende jedes Tages ins Gesamtsystem integriert und getestet. Auch in dieser Hinsicht lassen sich XP und die Aspektorientierung gut kombinieren. Da die Korrektheit von Aspekten wie erwähnt nicht ohne die Komponenten geprüft werden kann, ist eine fortlaufende Integration für die Aspektorientierung von Vorteil.

40-Stunden-Woche. Damit ist gemeint, dass die Entwickler nicht systematisch Überstunden leisten sollen. Dies ist unabhängig davon, ob ein Software-System aspektorientiert oder konventionell entwickelt wird.

Kunde vor Ort. In jedem XP-Projekt ist ein kompetenter Vertreter des Auftraggebers verfügbar, um Fragen zu beantworten, Streitpunkte zu klären und Prioritäten zu setzen. Auch hier gilt, was bereits beim Planungsspiel geschrieben wurde: Gerade Crosscutting Concerns und implizite Anforderungen erfordern eine enge und vertrauensvolle Zusammenarbeit mit dem Auftraggeber.

Aus den obigen Betrachtungen lässt sich zusammenfassend folgern, dass sich XP und die Aspektorientierung gut ergänzen. In einigen wichtigen Bereichen verfolgen XP und die Aspektorientierung ähnliche Ziele und unterstützen sich gegenseitig. Auf einige Merkmale von XP hat es keinen Einfluss, ob ein Software-System aspektorientiert oder konventionell entwickelt wird. Hingegen gibt es kein Merkmal von XP, auf das sich die aspektorientierte Entwicklung schädlich auswirken würde. Lediglich das automatisierte Testen, das in XP eine grosse Bedeutung hat, ist in aspektorientierten Systemen leicht komplizierter.

14.2 Architekturmuster

Einige Architekturmuster aus Glinz (2003a) können mit Aspekten realisiert werden:

14.2.1 Hollywood-Muster

Das Hollywood-Muster („Don't call us, we call you") ist seinem Wesen nach „crosscutting". Der Ereignisverwalter, der alle Eingabeereignisse registriert, sollte als Aspekt implementiert werden können. Dies gilt übrigens auch für alle Java-Methoden, die abstrakte Methoden von `Listener`-Interfaces implementieren. Nachteil ist, dass die resultierenden Aspekte vermutlich nicht homogen sind, d.h. sie müssen Fallunterscheidungen durchführen und haben daher eine Art von logischer Kohäsion. Ob diesem Nachteil durch die Implementierung von abstrakten Aspekten begegnet werden kann, wäre zu untersuchen.

14.2.2 Model-View-Controller-Muster (MVC)

Der Controller oder der oft kombinierte View-Controller des MVC-Musters lassen sich durch einen Aspekt realisieren. Der Aspekt leitet alle Eingabeereignisse ans Model und ggf. an die View weiter.

Dies lässt sich aus der Tatsache schliessen, dass das MVC-Muster häufig durch das Beobachtermuster (siehe Gamma et al., 1996) realisiert wird, und dass das Beobachtermuster seinerseits aspektorientiert implementiert werden kann (siehe im Abschnitt 10.2).

14.2.3 Client/Server-Muster

Aspekte eignen sich ausgezeichnet für die Realisierung von Client/Server-Mustern auch in verteilten Umgebungen. Im Abschnitt 9.5 wurde ein mit AspectC++ implementierter Aspekt vorgestellt, der lokale Methodenaufrufe abfängt und durch entfernte Aufrufe ersetzt.

14.3 Entwurfsmuster

Die Entwurfsmuster von Gamma et al. (1996) sind bewährte Lösungen für wiederkehrende Entwurfsprobleme und stehen in einem vielfältigen Verhältnis zur Aspektorientierung.

Aspekte zur Implementierung bestehender Entwurfsmuster. Einerseits können Aspekte benutzt werden, um Entwurfsmuster zu implementieren. Damit lassen sich die Vorteile der Aspekte mit den Vorteilen der Entwurfsmuster kombinieren. Da Aspekte zumindest in linguistischen Ansätzen keine signifikanten Performance-Einbussen zur Folge haben, sind aus dieser Kombination keine wesentlichen Nachteile zu erwarten. So wird im Abschnitt 10.2 grob gezeigt, wie das *Beobachtermuster* mit AspectJ und Hyper/J implementiert werden kann, und in Clarke, Walker (2001a) ist es detailliert nachzulesen. Die DJ Library ist im Prinzip nichts anderes als ein Framework zur Implementierung des *Besuchermusters*. Im nächsten Abschnitt 14.3.1 werden die DJ Library und das Besuchermuster miteinander verglichen.

Aspekte als Alternative zu bestehenden Entwurfsmustern. Anderseits können die Entwurfsmuster auch als Alternative zu Aspekten betrachtet werden. Tarr et al. (1999) ersetzen die konventionelle Implementierung eines Software-Systems, das den Stil, die Syntax und die Semantik von Ausdrücken in einem Syntaxbaum prüft, durch das das *Besuchermuster*. Für jede Art von Prüfung (Stil, Syntax, Semantik) wird ein Besucher mit allen Prüfmethoden implementiert; die Prüfmethoden der Klassen des Syntaxbaums werden durch `accept`-Methoden ersetzt. Das gleiche Software-System wird später auch noch mit Hyper/J implementiert. Gegenüber der aspektorientierten Implementierung hat das Besuchermuster die folgenden Nachteile:

- Änderungen sind *invasiv*. Bei der erstmaligen Implementierung des Besuchermusters müssen die Klassen des Syntaxbaums geändert werden; wird später ein neuer Ausdruck hinzugefügt, müssen alle Besucher geändert werden. Anzustreben sind *additive* Änderungen.
- Die Wiederverwendung der Besucher ist eingeschränkt, da sie die Klassen des Syntaxbaums kennen müssen.
- Die (Rück-)Verfolgbarkeit zwischen Anforderungen und Entwurf ist schwierig, da die Prüfungsmethoden über alle Subklassen der Besucher verteilt („scattered“) sind.
- Der Code wird komplexer.

Im gleichen System ist ein Logger implementiert, der die Ausführung sämtlicher Prüfmethoden festhält. Die Prüfmethoden der Klassen des Syntaxbaums sind selber für den Aufruf der entsprechenden Logger-Methode verantwortlich. Um dieses Scattering zu reduzieren, implementieren Clarke et al. (1999) das Logging mit dem *Beobachtermuster*. Das Beobachtermuster hat gegenüber der aspektorientierten Implementierung die folgenden Nachteile:

- Der Code wird komplexer, besonders wenn neben dem Beobachtermuster auch das Besuchermuster implementiert wird. Dies schränkt die Verständlichkeit und Pflegbarkeit ein.
- Die Performance kann beeinträchtigt werden.

- Da die Prüfmethoden nun den Beobachter über ihre Ausführung benachrichtigen müssen, wird das Scattering nicht behoben, sondern nur durch ein anderes ersetzt.

Als Alternative zum Beobachtermuster implementieren Clarke et al. (1999) das Logging ausserdem mit dem *Dekorierermuster*. Die Benachrichtigung wird von den Prüfmethoden in die Dekorierer verlagert. Das Problem beim Dekorierermuster besteht darin sicherzustellen, dass sämtliche Nachrichten an die Dekorierer und nicht mehr an die Objekte des Syntaxbaums geschickt werden.

Anderseits haben Entwurfsmuster gegenüber Aspekten auch Vorteile, so dass von Fall zu Fall entschieden werden muss, ob sich nun ein Entwurfsmuster oder ein Aspekt für die Lösung eines bestimmten Problems besser eignet:

- Entwurfsmuster können mit bestehenden Sprachkonstrukten in jeder beliebigen objektorientierten Programmiersprache implementiert werden.
- Entwurfsmuster sind bekannt, tausendfach bewährt und haben insgesamt den Vorteil grösserer Reife.

Aspekte als neue Entwurfsmuster. Drittens können gewisse objektorientierte Ansätze durchaus als neue Entwurfsmuster betrachtet werden. Man denke beispielsweise an das Aspect Moderator Framework aus dem Abschnitt 9.8, welches zwar in diesem Fall tatsächlich ein Framework ist, aber durchaus auch aus Entwicklungsmuster konzipiert werden könnte.

14.3.1 Vergleich zwischen der DJ Library und dem Besuchermuster

Die *DJ Library* (siehe im Abschnitt 9.4) und das *Besuchermuster* dienen einem ähnlichen Zweck. Beide ermöglichen es, Operationen an verschiedenen Knoten einer Objektstruktur auszuführen, und zwar möglichst unabhängig von der zugrunde liegenden Klassenhierarchie.

In der **DJ Library** ist dafür ein so genannter *adaptiver Besucher* definiert, der den Code implementiert, der an jedem Knoten in der zu traversierenden Objektstruktur ausgeführt werden muss. Der adaptive Besucher und die Objektstruktur können unabhängig voneinander geändert werden.

Demgegenüber fasst das **Besuchermuster** die verwandten Operationen jeder Klasse in einem separaten Objekt, dem *Besucher*, zusammen, der die Knoten der Objektstruktur der Reihe nach besucht. Zu diesem Zweck muss jeder Knoten eine `accept`-Methode implementieren, die der Besucher aufrufen kann. Im Rumpf dieser `accept`-Methode kann der Knoten dann die gewünschte Operation des Besuchers aufrufen. Das Besuchermuster unterscheidet sich in folgenden Punkten von der DJ Library.

- Es erlaubt pro Knoten der Objektstruktur unterschiedliche Implementierungen der Operationen.
- Dem Besucher muss die Objektstruktur bekannt sein, damit er sie traversieren kann, d.h. eine Änderung der Objektstruktur erfordert eine entsprechende Anpassung des Besuchers. Hingegen können – wie in der DJ Library auch – im Besucher Operationen hinzugefügt, geändert oder entfernt werden, ohne dass die Objektstruktur geändert werden muss.
- Jeder Knoten der Objektstruktur muss eeine `accept`-Methode mit dem Aufruf der gewünschten Operation implementieren. Dies ist einerseits *invasiv* (Clarke et al., 1999), anderseits führt es zu *Scattering* (die `accept`-Methode ist in allen Knoten implementiert) und *Tangling* (in allen Knoten vermischt sich die `accept`-Methode mit dem eigentlichen Verhalten des Knotens).

Eignung. Aus diesen Unterschieden ergeben sich unterschiedliche Eignungen des Besuchermusters und der DJ Library:

- Die DJ Library eignet sich besonders, wenn die Objektstruktur unbekannt ist oder sich häufig ändert. Die DJ Library eignet sich ebenfalls, wenn an allen Objekten der Objektstruktur die gleichen Operationen ausgeführt werden müssen.
- Das Besuchermuster eignet sich besonders, wenn die Objektstruktur bekannt ist und sich selten ändert. Das Besuchermuster eignet sich ebenfalls, wenn an den Objekten der Objektstruktur unterschiedliche Operationen ausgeführt werden müssen.

14.4 Vertragsprinzip

Das Vertragsprinzip (engl. design by contract) stammt ursprünglich von Meyer (1997). Es fordert, dass Modulschnittstellen als Verträge in Form von so genannten *Zusicherungen* (engl. assertions) spezifiziert werden, welche der Anbieter und die Nutzer eines Moduls miteinander vereinbaren. Zusicherungen können sein (Glinz, 2004):

- **Voraussetzungen** (engl. preconditions), die der Nutzer eines Moduls erfüllen muss, bevor er das Modul aufruft,
- **Verpflichtungen** (engl. obligations), denen der Nutzer eines Moduls irgendwann nach dem Aufruf des Moduls nachkommen muss,
- **Ergebniszusicherungen** (engl. postconditions), die den Anbieter eines Moduls zur Erbringung eines bestimmten Resultats verpflichten, oder
- **Invarianten**, die der Anbieter eines Moduls nicht verändern darf.

Verträge werden zwischen Anbietern und Nutzern von Komponenten geschlossen. Aspekte können die Erfüllung solcher Verträge beeinflussen, indem sie das Verhalten der Komponenten quasi im Verborgenen verändern. Da die Komponenten davon nichts wissen (können), haben sie auch keine Kontrolle über das veränderte Verhalten. Das bedeutet, dass Anbieter und Nutzer von Komponenten Verträge eingehen, deren Einhaltung sie nicht garantieren können, sobald Aspekte im Spiel sind. Das Vertragsprinzip muss folglich auf Aspekte ausgedehnt werden.

In diesem Abschnitt wird versucht, die Probleme in diesem Bereich zu identifizieren und Lösungsmöglichkeiten zu skizzieren. Um die Untersuchung einfach zu halten, wird vom (häufigen) Fall ausgegangen, dass AspectJ eingesetzt wird und dass sich die Verträge auf Methoden-Schnittstellen zwischen Komponenten beziehen. Ausserdem beschränkt sich die Betrachtung auf `call`-Pointcuts, da diese weitaus am häufigsten vorkommen. Die Verpflichtungen werden nicht behandelt, da sie mit der Methodenausführung im engeren Sinne nichts zu tun haben und zu einem beliebigen Zeitpunkt erfüllt werden können.

14.4.1 Identifikation der Probleme

Je nach Art der verwendeten Advice ergeben sich unterschiedliche Probleme. Jedoch kann jede der drei Arten von Advice die vertragskonforme Methodenausführung gefährden.

- **Before Advice** erweitern die aufrufende Komponente vor dem eigentlichen Methodenaufruf um zusätzliches Verhalten. Damit können sie erfüllte Voraussetzungen zerstören, auch wenn sich die aufrufende Komponente vertragskonform verhalten hat.
- **After Advice** erweitern die aufrufende Komponente nach dem eigentlichen Methodenaufruf um zusätzliches Verhalten. Damit können sie erfüllte Ergebniszusicherungen oder Invarianten zerstören, auch wenn sich die ausführende Komponente vertragskonform verhalten hat.
- **Around Advice** mit `proceed()` entsprechen einer Kombination von Before und After Advice. Damit können sie sämtliche Zusicherungen zerstören, auch wenn sich beide Komponenten vertragskonform verhalten haben. Around Advice ohne `proceed()` unterbinden den Methodenaufruf vollständig und können damit die Vertragserfüllung generell verhindern.

Die erwähnten Probleme beziehen sich auf `call`-Pointcuts, können aber analog auch bei `execution`-Pointcuts auftreten. In diesem Fall wird das Verhalten der aufgerufenen Komponente verändert. Diese Probleme stellen vermutlich nur die „Spitze des Eisbergs“ dar. Würde die Untersuchung auf andere Arten von Pointcuts (z.B. `cflow`-Pointcuts) ausgedehnt, kämen möglicherweise weitere Probleme hinzu.

14.4.2 Skizzierung der Lösungsmöglichkeiten

Für die erwähnten Probleme bieten sich verschiedene Lösungsmöglichkeiten an.

Zusicherungen für Aspekte. Lagaisse, Joosen, De Win (2004) fordern, dass nicht nur die Komponenten, sondern auch die Aspekte ihre Voraussetzungen und Ergebniszusicherungen deklarieren. Dabei kommen theoretisch ganz ähnliche Regeln zur Anwendung, wie sie in der Objektorientierung für die *Substitution* im Rahmen der Vererbung gelten (Glinz, 2003a).

- **Voraussetzungen**. Kein Before oder Around Advice darf die Voraussetzungen der Komponentenmethoden, die er quer schneidet, verstärken; er darf sie höchstens abschwächen.
- **Ergebniszusicherungen**: Kein After oder Around Advice darf die Ergebniszusicherungen der Komponentenmethoden, die er quer schneidet, abschwächen; er darf sie höchstens verstärken.
- **Invarianten**. Kein Advice darf die Invarianten der Komponenten, die er quer schneidet, verletzen.

Werden diese Regeln befolgt, ist sichergestellt, dass die erwähnten Probleme nicht auftreten können. Die praktische Umsetzung ist allerdings ohne Sprach- oder Werkzeugunterstützung nahezu unmöglich, muss doch jeder Aspekt die Zusicherungen sämtlicher aktueller und zukünftiger (!) Komponenten berücksichtigen, die er quer schneidet. (Da die Komponenten im Idealfall von den Aspekten nichts wissen, ist die Gefahr gross, dass niemand daran denkt, die Korrektheit der Aspekte zu überprüfen, wenn sich die Zusicherungen der Komponenten ändern.)

Einige Vererbungsgrundsätze aus der Objektorientierung können *sinngemäss* auf die aspektorientierte Programmierung mit AspectJ übertragen werden. Im Unterschied zur Objektorientierung bezieht sich die Vererbung bei AspectJ nicht auf Klassen, sondern auf Methoden, und die Methoden werden nicht dynamisch, sondern statisch gebunden. Im Prinzip erbt jeder Around Advice[37] das Verhalten der Komponentenmethoden, die er quer schneidet, und fügt ihnen eigenes Verhalten hinzu. Auch die Unterscheidung zwischen Substitution, Spezialisierung und Steinbruch aus Glinz (2003a) gilt sinngemäss für Komponentenmethoden und Around Advice:

- **Methodensubstitution**. Ein Around Advice, der die obigen Regeln befolgt, ist eine korrekte Implementierung der Komponentenmethoden, die er quer schneidet, und kann diese folglich ersetzen. Ein Beispiel dafür ist ein Around Advice, der Methoden vor und nach ihrer Ausführung um Tracing-Verhalten erweitert. Er verstärkt keine Voraussetzungen, schwächt keine Ergebniszusicherungen ab und verletzt keine Invarianten.
- **Methodenspezialisierung**. Ein Around Advice, der keine Invarianten verletzt, ist ein Spezialfall der Komponentenmethoden, die er quer schneidet. Ein Beispiel dafür ist ein Around Advice, der einen Methodenaufruf abfängt und in eine Warteschlange stellt, anstatt dass er direkt ausgeführt wird. Er verletzt zwar keine Invarianten, schwächt aber die Ergebniszusicherungen ab.
- **Methodensteinbruch**. Darunter ist ein Around Advice zu verstehen, der die Komponentenmethoden, die er quer schneidet, zwar erweitert oder ersetzt, aber keinerlei Rücksicht auf ihre Zusicherungen nimmt.

[37] Before und After Advice sind dafür zu wenig mächtig; sie können Komponentenmethoden nur erweitern, nicht aber ersetzen.

Sichtbarkeit. Lagaisse, Joosen, De Win (2004) setzen sich ausführlich mit der Sichtbarkeit auseinander. Ein kluger Umgang mit der Sichtbarkeit der Variablen und Methoden in den Komponentenmodulen trägt viel zur Vermeidung von Vertragsverletzungen durch Aspekte bei. Grundsätzlich ist die Sichtbarkeit restriktiv zu handhaben. Lagaisse, Joosen, De Win (2004) zeigen aber, dass eine allzu restriktive Sichtbarkeit nicht in allen Fällen praktikabel ist: Ein Persistenzaspekt muss beispielsweise `private` Variablen, die vor dem Zugriff durch andere Komponentenmodule geschützt sein sollen, in der Datenbank speichern können. Zur Lösung solcher Probleme gibt es zwei Möglichkeiten:

- **Privilegierte Aspekte**. In AspectJ gibt es privilegierte Aspekte (siehe im Abschnitt 9.3), die auf Variablen und Methoden zugreifen können, auch wenn deren Sichtbarkeit `private` ist. Damit lösen sie zwar das Problem der allzu restriktiven Sichtbarkeit, sie gehen aber viel zu weit: Privilegierte Aspekte haben Zugriff auf *sämtliche* `private` Variablen und Methoden *sämtlicher* Komponentenmodule und sind daher unbedingt zu vermeiden.
- **Erweiterungen der Verträge von Komponenten**. Lagaisse, Joosen, De Win (2004) fordern, dass die Verträge von Komponentenmodulen erweitert werden müssen. Die Komponentenmodule sollen in einem so genannten *Aspektintegrationsvertrag* (engl. aspect integration contract) deklarieren, welche Aspekte ihre mit `private` deklarierten Methoden und Variablen lesen oder verändern dürfen. Die Erfüllung dieser Forderung bewirkt jedoch, dass die Transparenz der Aspekte abnimmt.

Nebenwirkungsfreie Aspekte. Aspekte sollen möglichst *nebenwirkungsfrei* im Sinne von Glinz (2004) sein, d.h. sie sollen den Systemzustand nur verändern, wo dies wirklich notwendig ist. Eine gute Modularisierung der Aspekte und Komponenten (siehe im Abschnitt 7.3) trägt dazu bei, die möglichen Nebenwirkungen der Aspekte auf die Komponenten auf ein Minimum zu reduzieren:

- Die für Aspekte geforderte *funktionale Kohäsion* bewirkt, dass jeder Aspekt genau eine bestimmte Aufgabe zu lösen hat, wodurch allfällige Nebenwirkungen kontrollierbar werden.
- Die für Komponenten geforderte *objektbezogene Kohäsion* ermöglicht die Datenkapselung mit einer restriktiven Sichtbarkeit, so dass die Aspekte keine Invarianten der Komponenten verletzen können.
- Die für die Interaktion zwischen Komponenten und Aspekten geforderte *Datenkopplung* zwischen Aspekten und Komponenten sorgt dafür, dass die Aspekte genau über die Informationen verfügen, die sie wirklich brauchen.

Eine Verletzung dieser Regeln kann übrigens zumindest teilweise mit Programmierrichtlinienaspekten (siehe im Abschnitt 7.5) automatisch verhindert oder zumindest protokolliert werden.

Sind Aspekte und Komponenten auf diese Weise sauber modularisiert, besteht die Gefahr von Nebenwirkungen eigentlich nur noch bei Around Advice, die mit `proceed()` Komponentenmethoden aufrufen. (Before und After Advice haben keine Möglichkeit, die Komponentenmethoden aufzurufen.) Diesen Nebenwirkungen kann wie folgt begegnet werden:

- Es ist darauf zu achten, dass die Around Advice die Komponentenmethoden mit unveränderten Argumenten aufrufen.
- Es ist darauf zu achten, dass die Around Advice die Rückgabewerte bzw. -objekte der Komponentenmethoden unverändert zurückgeben.

14.5 Entwicklungsaspekte zur Testunterstützung

Wie im Abschnitt 7.5 erwähnt, können Entwicklungsaspekte auch zur Testunterstützung eingesetzt werden, und zwar sowohl in konventionellen als auch in aspektorientierten Systemen. Die Entwicklungsaspekte werden nach Abschluss der Testphase aus dem System entfernt bzw. inaktiviert und können vor einem späteren Regressionstest wieder ins System integriert bzw. aktiviert werden. Wie

Laddad (2003) zeigt, werden zu testende Aspekte am besten „mit ihren eigenen Waffen geschlagen“, d.h. es wird zur Unterstützung des Testens ein Tracing-Aspekt definiert, der die Ausführung der Methoden und Advice an bestimmten Join Points aufzeichnet. Damit können die durchgeführten Tests kontrolliert werden, und im Falle von Fehlern wird das Debugging erleichtert.

14.6 Re-Engineering

Konventionell implementierte Crosscutting Concerns, da mag das Software-System sonst noch so gut modularisiert sein, führen zu *Scattering* und *Tangling* und beeinträchtigen die Verständlichkeit und damit die Pflege der Software. Falls ein bestehendes konventionelles Software-System (engl. legacy system) noch ausreichend lange genutzt werden soll, ist es unter Umständen wirtschaftlich sinnvoll, zumindest einen Teil der Crosscutting Concerns nachträglich aspektorientiert zu implementieren. Eine sorgfältige Kosten-/Nutzenanalyse ist dabei unerlässlich, sind doch sowohl die Kosten als auch die Risiken beträchtlich.

Ein solches Vorhaben wird in der Fachsprache mit *Re-Engineering* bezeichnet. Nach Klösch, Gall (1995) ist Re-Engineering die Untersuchung und Änderung eines bestehenden Systems mit dem Ziel, das System in einer neuen, geänderten Form zu implementieren. Re-Engineering besteht üblicherweise aus zwei Phasen:

1. **Reverse Engineering** analysiert ein existierendes Software-System, um die Systemkomponenten und ihre Abhängigkeiten zu identifizieren und dann entsprechende Repräsentationen des Systems auf einer höheren Abstraktionsebene zu erzeugen. Auf die Aspektorientierung übertragen, bedeutet Reverse Engineering folgendes: Die in einem konventionellen Software-System *verborgenen Aspekte*[38] werden identifiziert und aspektorientiert dokumentiert.
2. **Restrukturierung** (engl. restructuring) ist die Transformation von einer Repräsentation des untersuchten Systems in eine andere, auf derselben Abstraktionsebene. Auf die Aspektorientierung übertragen, bedeutet Restrukturierung folgendes: Die beim Reverse Engineering identifizierten und dokumentierten verborgenen Aspekte werden aus dem bestehenden Software-System entfernt und als tatsächliche Aspekte implementiert. Dies wird in der Fachsprache auch mit *code-to-code-Transformation* bezeichnet.

Aspektorientiertes Reverse Engineering von konventionellen Software-Systemen. Die schwierigste Aufgabe ist die *Identifikation* von verborgenen Aspekten. Das Thema wird in der Literatur kaum behandelt. Eine Ausnahme ist der Ansatz von Hannemann, Kiczales (2001), der sogar durch ein Werkzeug, das *Aspect Mining Tool (AMT)*[39], unterstützt wird. Der Ansatz umfasst eine textbasierte und eine typbasierte Analyse von Java-Quellcode. Die *textbasierte Analyse* sucht mittels *Pattern Matching* nach Klassen und Methoden gleichen Namens und funktioniert folglich nur, wenn sich das konventionelle Software-System strikt an Namenskonventionen hält. Die *typbasierte Analyse* sucht nach der Verwendung gleicher Klassen und Methoden.

Es gäbe durchaus noch weitere Indikatoren, die das Vorhandensein von verborgenen Aspekten in bestehenden Software-Systemen anzeigen. Hier einige Denkanstösse:

- **Tangling in einer Methode**. Tangling, d.h. ein Modul löst mehr als eine Aufgabe (z.B. eine Hauptaufgabe und mehrere Nebenaufgaben), ist immer auch ein Hinweis auf Scattering. Gerade die Nebenaufgaben können verborgene Aspekte sein. Sie bilden daher einen Ausgangspunkt für die in den nächsten beiden Abschnitten beschriebenen Untersuchungen.

[38] Diese „scattered“ Code-Fragmente von Crosscutting Concerns werden in Hannemann, Kiczales (2001) verborgene Concerns (engl. hidden concerns) genannt.

[39] www.cs.ubc.ca/~jan/amt/ (22.07.2004)

- **Gleiche Anweisungen**. Wird die gleiche (Folge von) Anweisung(en) in verschiedenen Modulen ausgeführt, weist dies auf Scattering hin. Lässt sich in der Umgebung der (Folge von) Anweisung(en) ausserdem ein *Muster* erkennen, steckt hinter der (Folge von) Anweisung(en) möglicherweise ein verborgener Aspekt. Solche Muster können sein: Die (Folge von) Anweisung(en) steht in den Methodenrümpfen immer an der gleichen Stelle (z.B. zu Beginn oder am Ende), die (Folge von) Anweisung(en) steht immer vor oder nach dem Aufruf von Methoden mit einer bestimmten Eigenschaft (z.B. Methoden, die den Saldo eines Kontos verändern) etc.
- **Gleiche Methoden**. Wird die gleiche Methode in verschiedenen Modulen aufgerufen, weist dies ebenfalls auf Scattering hin. Wie im obigen Abschnitt gilt auch hier: Lässt sich in der Umgebung der Aufrufe ein Muster erkennen, steckt hinter der Methode möglicherweise ein verborgener Aspekt.
- **Anforderungsspezifikation**. Wie im Abschnitt 13.1 erwähnt, ist in vielen konventionellen Anforderungsdokumenten eine gewisse Separation of Concerns durch die Trennung von funktionalen und nicht-funktionalen Anforderungen in der Kapitelstruktur vorgegeben. Dieser Umstand kann zur Identifikation von möglichen verborgenen Aspekten ausgenutzt werden. Voraussetzung ist aber, dass die Anforderungsspezifikation gepflegt wurde und mit der Software übereinstimmt.
- **Typische Crosscutting Concerns**. Viele Crosscutting Concerns (z.B. Authentisierung, Autorisierung, Logging etc.) sind in fast allen Software-Systemen in irgendeiner Form implementiert. Nach Code-Fragmenten von solchen typischen Crosscutting Concerns kann in der Software aktiv gesucht werden.

Sind mögliche verborgene Aspekte einmal identifiziert, muss geprüft werden, ob es sich dabei wirklich um „scattered“ Code-Fragmente von Crosscutting Concerns handelt. Wenn ja, werden sie als tatsächliche Aspekte dokumentiert. Zu diesem Zweck kann einer der im Kapitel 10 vorgestellten Ansätze des aspektorientierten Entwurfs benutzt werden. Ziel der Dokumentation ist es, die Struktur und das Verhalten der Aspekte zu beschreiben sowie auf die Komponenten, die sie quer schneiden, zu verweisen.

Anschliessend muss entschieden werden, wie man mit den nunmehr dokumentierten Aspekten weiter verfährt. Entweder man belässt es beim Reverse Engineering: Rein durch die Dokumentation ist im Hinblick auf die Verständlichkeit und Pflege der Software bereits viel gewonnen, der Aufwand hält sich Grenzen, und es besteht kein Risiko von Nebenwirkungen. Oder aber man geht einen Schritt weiter:

Aspektorientierte Restrukturierung von konventionellen Software-Systemen. Die verborgenen Aspekte manuell zu entfernen und als tatsächliche Aspekte zu implementieren, erfordert zahlreiche invasive Eingriffe in die Software, ist aufwändig und birgt das Risiko von Nebenwirkungen. Gesucht sind also Ansätze, welche die Restrukturierung unterstützen oder im Idealfall sogar automatisieren. Vereinzelte Ansätze zur Restrukturierung sind vorhanden. Beispielsweise können mit Hyper/J Klassen und Methoden zu neuen (Sub-)Systemen kombiniert werden (Stichwort: nicht-invasive Remodularisierung). Dies dürfte jedoch in den seltensten Fällen genügen. Wie im Abschnitt 9.6 gezeigt wurde, kann gerade die Restrukturierung von „tangled“ Methoden auch in Hyper/J nur manuell und invasiv gelöst werden. Laddad (2003) empfiehlt, eine allfällige Restrukturierung in jedem Fall schrittweise, d.h. Aspekt für Aspekt, durchzuführen, um den Aufwand und die Risiken zu minimieren. Voraussetzung für die Restrukturierung ist aber in jedem Fall, dass das bestehende Software-System bereits sauber modularisiert ist und dass Namenskonventionen existieren und eingehalten werden. Andernfalls ist nicht daran zu denken, konventionelle Software-Systeme aspektorientiert zu restrukturieren, weder manuell noch automatisch.

15. Trends

Dieses Kapitel versucht aufzuzeigen, in welche Richtung sich die Aspektorientierung zurzeit bewegt. Wirft man einen Blick auf die Programme und Ergebnisse der Konferenzen zur aspektorientierten Software-Entwicklung (Aspect-Oriented Software Development (AOSD), Early Aspects) seit 2002, kristallisieren sich die folgenden Trends heraus:

15.1 Darstellung von Crosscutting Concerns mit der UML

Die meisten der in den Kapiteln 10, 11 und 12 beschriebenen Ansätze verwenden – neben Tabellen und eigenen Notationen – die UML in irgendeiner Form zur Darstellung von Crosscutting Concerns. In diesem Sinne spielt die UML für die Early Aspects eine ähnliche Rolle wie AspectJ für die aspektorientierte Programmierung. *Dass* die UML zur Darstellung von Crosscutting Concerns verwendet werden soll, darin sind sich die Forscher offensichtlich weitgehend einig. Hingegen herrscht weniger Einigkeit darüber, *wie* die UML zur Darstellung von Crosscutting Concerns eingesetzt werden soll, und es hat sich demzufolge noch kein Standard herauskristallisiert (siehe auch im Abschnitt 16.1). Es ist anzunehmen, dass sich dies demnächst ändern wird. Ein klarer Trend ist nur bei der Verwendung der UML-Diagrammtypen auszumachen: Hier stehen – wenig verwunderlich – eindeutig die Anwendungsfalldiagramme, die Sequenzdiagramme und die Klassendiagramme im Vordergrund.

15.2 AspectJ als Marktführer

Nachdem sich AspectJ als bedeutendster unter den Ansätzen der aspektorientierten Programmierung etabliert hat, bilden viele Forschergruppen ihre Ansätze auf AspectJ ab oder definieren entsprechende Erweiterungen von AspectJ. Beispiele:

- **DAJ** von Sung, Lieberherr (2002) ist eine Erweiterung von AspectJ, die es erlaubt, die Konzepte der adaptiven Programmierung (Traversierungsstrategie, adaptiver Besucher, Klassengraph) in AspectJ zu formulieren (siehe auch im Abschnitt 9.4).
- **Composition Patterns**. Clarke, Walker (2001a) bilden ihre Composition Patterns auf AspectJ (und Hyper/J) ab.

Ausserdem werden laufend Erweiterungen für AspectJ vorgeschlagen, um bestimmte Probleme überhaupt erst lösen zu können. Beispiele:

- **`dflow`-Pointcut**. Kawauchi, Masuhara (2004) schlagen einen `dflow-Pointcut` vor, um Datenflüsse von verbotenen Quellen identifizieren zu können. Damit kann sich eine Web-Anwendung mit Hilfe eines Aspekts gegen Angriffe von Dritten schützen.
- **DJcutter** ist eine AspectJ-Erweiterung zur Implementierung von verteilten Aspekten. Ein so genannter *entfernter* (engl. remote) *Pointcut* ist in der Lage, Join Points sowohl auf dem lokalen als auch auf einem entfernten Rechner zu identifizieren. Der entsprechende Advice ist das aspektorientierte Pendant zu einem Methodenaufruf mit Java RMI[40] (remote method invocation), er wird also auf dem Rechner ausgeführt, auf dem der Join Point identifiziert wurde. Das Weaving erfolgt zur Ladezeit. (Nishizawa, Chiba, Tatsubori, 2004)
- **Assoziationsaspekte** (engl. association aspects) sind eine Erweiterung der Aspektassoziationen von AspectJ. Mit Hilfe eines Assoziationsaspekts (`perobjects`) kann eine Aspektinstanz mit einer Kollektion von Objekten verknüpft werden. Damit kann sie beispielsweise die Interaktionen zwischen diesen Objekten koordinieren. (Sakurai et al., 2004)

[40] www.java.sun.com/products/jdk/rmi/ (22.07.2004)

Dieser Trend in Richtung AspectJ wird sich zweifellos fortsetzen. Gleichzeitig ist zu beobachten, dass keine grundsätzlich neuen Ansätze mehr publiziert wurden, und dass weniger verbreitete Ansätze in der Literatur kaum mehr Erwähnung finden. Einzige Ausnahme ist – neben AspectJ – noch Hyper/J. Offensichtlich ist die im Abschnitt 16.1 erwähnte Konsolidierung des Marktes zumindest für die Ansätze der aspektorientierten Programmierung bereits in vollem Gange.

15.3 Erweiterung konventioneller Ansätze um Aspekte

Viele konventionelle Konzepte, Sprachen und Werkzeuge werden um aspektorientierte Konzepte erweitert. Beispiele:

- Araujo, Coutinho (2003) erweitern ihre Viewpoint-orientierte Anforderungsmethode *Vision* um aspektorientierte Konzepte (siehe auch im Abschnitt 12.3).
- Mousavi et al. (2002) erweitern ihre formale Spezifikationssprache *GAMMA* nach der „Formel“ Aspects + GAMMA = AspectGAMMA.

Bei anderen Ansätzen (z.B. bei den Stratified Frameworks; siehe im Abschnitt 11.1) kann nur vermutet werden, dass sie ursprünglich nicht aspektorientiert waren. Sowohl die ursprünglichen als auch die erweiterten Methoden und Sprachen werden vermutlich in Zukunft eher ein Nischendasein führen.

15.4 Theoretische Grundlagen und Formalisierung

Es ist festzustellen, dass sich die Forschung teilweise mit den theoretischen Grundlagen der Aspektorientierung und der Formalisierung von Aspekten bzw. der Separation of Concerns befasst, sei dies im Bereich der mathematischen Grundlagen, sei dies im Bereich von formalen Spezifikationssprachen. Beispiele:

- Mili, Elkharraz, Mcheick (2004) gehen der Frage nach, wie sich die Separation of Concerns formal beschreiben lässt.
- Blair, Blair (1999) beschreiben Aspekte mit LOTOS (Language of Temporal Ordering Specifications), einer formalen Spezifikationssprache für verteilte und nebenläufige Systeme (ISO, 1989).
- Die *Foundations of Aspect-Oriented Languages (FOAL)*, ein Workshop, der jeweils im Rahmen der AOSD-Konferenzen stattfindet, befassen sich im Wesentlichen mit den theoretischen Grundlagen der Aspektorientierung sowie mit bestehenden Sprachen (z.B. AspectJ) aus formaler Sicht.

15.5 Wiederverwendung vorgefertigter Aspekte in AspectJ

Wie im Abschnitt 9.3 gezeigt, besteht das Grundschema von wiederverwendbaren abstrakten Aspekten aus abstrakten Pointcuts und konkreten Advice. Sind die Advice *anwendungsneutral*, bietet es sich an, einen Aspekt abstrakt zu implementieren und ihn damit wiederverwendbar zu machen. Dies ist bei den meisten Aspekten der Fall, beispielsweise bei Entwicklungs-, Sicherheits-, Nebenläufigkeits- und Optimierungsaspekten. Wenn die Advice *anwendungsspezifisch* sind, eignen sich die Aspekte weniger für die Wiederverwendung. Dies dürfte normalerweise bei Geschäftsaspekten der Fall sein. In solchen Fällen bieten sich die Massschneiderung bestehender Aspekte oder die Verwendung von *Aspektmustern* (z.B. das Arbeiterobjekt-Erzeugungsmuster) bzw. *Aspektidiomen* (z.B. das Idiom zum Deaktivieren von Advice) als Alternative an.

In jedem Fall liegt die Wiederverwendung vorgefertigter Aspekte im Trend, und es werden immer mehr Aspekte, Entwurfsmuster und Idiome für Aspekte öffentlich verfügbar gemacht, insbesondere in AspectJ. Beispiele:

- Über die Homepage von AspectJ[41] sind vorgefertigte wiederverwendbare Aspekte zugänglich.
- Die JSAL (Java Security Aspect Library) von Huang, Wang, Zhang (2004) umfasst abstrakte und damit wiederverwendbare Aspekte für Sicherheit (Ver-/Entschlüsselung, Authentisierung, Autorisierung und Audit).
- Boström (2004) schlägt Aspekte für die Ver-/Entschlüsselung vor.
- Laney, Van der Linden, Thomas (2004) modellieren eine digitale Signatur als Aspekt.

16. Lücken

Da die Aspektorientierung ein relativ junges Forschungsgebiet ist, weisen die Forschungsergebnisse durchaus noch Lücken auf, von denen in diesem Kapitel einige diskutiert werden.

16.1 Ansätze zur aspektorientierten Software-Entwicklung

Ein wirklich überzeugender, durchgängiger und praxistauglicher Ansatz zur Unterstützung der aspektorientierten Software-Entwicklung ist noch nicht auszumachen. Ein solcher Ansatz müsste ein Vorgehensmodell beinhalten, Mittel zur Identifikation, Darstellung und Beschreibung von Crosscutting Concerns bereitstellen, ihre Abbildung auf Aspekte ermöglichen sowie durch ein Werkzeug unterstützt sein. Im Idealfall werden die aspektorientierten Konzepte in einen bestehenden Ansatz zur Unterstützung der konventionellen Software-Entwicklung, z.B. in den Unified Software Development Process (USDP) und die Unified Modeling Language (UML), integriert. Zwar sind in einigen Ansätzen (z.B. der USDP-Anpassung von Sousa et al., 2004, oder der AML von Groher, Baumgarth, 2004) durchaus viel versprechende Fragmente eines solchen Ansatzes zu finden, aber die Durchgängigkeit und Werkzeugunterstützung fehlt.

Dieser Umstand ist möglicherweise darauf zurückzuführen, dass sich in den letzten Jahren ein eigentlicher Markt für neue aspektorientierte Ansätze entwickelt hat. Im Anfangsstadium ist ein solcher Markt immer von Vielfalt und Wildwuchs geprägt; das liess sich während der letzten Jahre auf dem Telekommunikationsmarkt gut beobachten. Beginnt der Markt, sich zu konsolidieren, kristallisieren sich im Normalfall ganz automatisch Standards heraus. Es ist anzunehmen, dass dies in den nächsten Jahren auch unter den Ansätzen der aspektorientierten Entwicklung der Fall sein wird. Erste Indizien dafür sind im Abschnitt 15.1 zu finden. In der Informatikgeschichte hat es immer wieder solche Konsolidierungsphasen gegeben, z.B. als sich Grady Booch, James Rumbaugh und Ivar Jacobson zusammengeschlossen haben, um die UML zu entwickeln. Es ist anzunehmen und zu hoffen, dass im Zug einer solchen Konsolidierung auch die Begriffe vereinheitlicht werden.

Im Bereich der Ansätze aspektorientierten Architektur und des aspektorientierten Entwurfs sind bisher nur wenige Ansätze veröffentlicht worden. Es ist zu hoffen, dass sich dies in naher Zukunft ändern wird. Ausserdem ist festzustellen, dass nur selten nach konventionellen Alternativen zur Aspektorientierung Ausschau gehalten wird. Beispielsweise wäre eine Gegenüberstellung von Entwurfsmustern und Aspekten sehr nützlich.

[41] www.eclipse.org (22.07.2004)

16.2 Werkzeugunterstützung

Besonders für die Ansätze der aspektorientierten Anforderungstechnik ist die Werkzeugunterstützung noch sehr mager. In vielen Arbeiten wird aber unter „Future Work“ die Absicht bekundet, sich demnächst um die Werkzeugunterstützung der beschriebenen Konzepte kümmern zu wollen. Beispiele:

- Rashid, Moreira, Araujo (2003) wollen AORE mit ihrem Werkzeug ARCaDe unterstützen (siehe auch im Abschnitt 12.2).
- Araujo, Coutinho (2003) wollen Vision mit einem Werkzeug unterstützen (siehe auch im Abschnitt 12.3).
- Sutton, Rouvellou (2002) wollen Cosmos mit einem Werkzeug unterstützen (siehe auch im Abschnitt 12.11).

Die Werkzeugunterstützung ist der Schlüssel für die Verbreitung und den Erfolg sämtlicher aspektorientierter Ansätze, die neue Sprachkonstrukte einführen. Die objektorientierten Ansätze der aspektorientierten Programmierung und die Ansätze, die auf der Standard-UML basieren, haben diesbezüglich einen Vorteil. Da sie auf bestehenden Sprachkonstrukten basieren, können bestehende Werkzeuge verwendet werden.

Für AspectJ sind auch bereits einige Entwicklungswerkzeuge verfügbar. Beispiel: Kiczales et al. (2001a) haben AspectJ nach eigenen Angaben in *Emacs*[42], *JBuilder*[43] und *Forte for Java* (neu: *Sun Java Studio*[44]) integriert. Unterstützt werden das Navigieren durch die Aspektstruktur und Verweise auf die Aspekte in den Komponenten. Ausserdem ist AspectJ in *Eclipse*[45] integriert.

Auch für die Testphase, besonders für das Debugging von aspektorientierten Systemen, sind Werkzeuge wichtig. da in aspektorientierten Systemen die dynamische Prozessstruktur von der statischen Programmstruktur abweicht (siehe Dijkstra, 1968). Dieses Problem kann aber notfalls auch mit einem Tracing-Aspekt gelöst werden.

16.3 Prüfung

Die Prüfung von aspektorientierten Systemen ist in der Literatur bis auf ganz wenige Hinweise zu Tests und Reviews (Laddad, 2003) und Verifikation (Katz, 2004) kein Thema. Nach H. Ossher im Interview von Elrad et al. (2001) ist die Sicherstellung der Korrektheit von Aspekten und ihren Interaktionen mit den Komponenten einer der wichtigsten offenen Punkte in der Aspektorientierung. Schon in konventionellen Systemen ist die Sicherstellung der Korrektheit von Modulen und ihren Interaktionen ein Problem, obwohl die auf Nachrichten basierenden Interaktionen zwischen den Modulen noch vergleichsweise einfach sind. Die auf Join Points basierenden Interaktionen zwischen Aspekten und Komponenten sind wesentlich komplizierter und erschweren sowohl die Spezifikation als auch das Testen.

[42] www.gnu.org/software/emacs/emacs.html (22.07.2004)
[43] www.borland.com/jbuilder (22.07.2004)
[44] www.java.sun.com (22.07.2004)
[45] www.eclipse.org (22.07.2004)

16.4 Vertragsprinzip

Die Aspektorientierung ist aus der Sicht des Vertragsprinzips (siehe auch im Abschnitt 14.4) ein zweischneidiges Schwert. Auf der einen Seite kann das Vertragsprinzip durch geeignete Aspekte unterstützt werden, auf der anderen Seite können Aspekte die Umsetzung des Vertragsprinzips beträchtlich erschweren. Beiden Fällen wurde bisher in der Forschung kaum Beachtung geschenkt.

Unterstützung des Vertragsprinzips. Obwohl sich die Aspektorientierung eignen würde, das Vertragsprinzip zu unterstützen, findet das Thema in der aspektorientierten Literatur mit Ausnahme von Ishio et al. (2004) und Constantinides et al. (2000) wenig Beachtung. Nimmt man AspectJ als Grundlage, bieten sich Aspekte an, um systemweit gültige Zusicherungen (engl. assertions) zu prüfen: Before Advice zur Prüfung von Voraussetzungen (engl. preconditions), After Advice zur Sicherstellung von Ergebniszusicherungen (engl. postconditions) und zur Prüfung von Invarianten. (Für die Prüfung von Invarianten könnte der `cflow`-Pointcut von Nutzen sein.) Allerdings hat der Einsatz von AspectJ auch einen schwerwiegenden Nachteil, wie Bryant et al. (2002) bemerken: Wird eine Methode geändert, müssen ihre Zusicherungen überprüft werden. Dabei entpuppt sich die Verwendung von Aspekten als Nachteil, weiss doch die Methode im Idealfall nichts von den Aspekten, die sie quer schneiden. Dieser Nachteil kann jedoch durch ein gutes aspektorientiertes Entwicklungswerkzeug wettgemacht werden, das in der Lage ist, zu jeder Methode die relevanten Aspekte anzuzeigen. Zusammenfassend wären die Voraussetzungen, Lösungsmöglichkeiten und Einschränkungen für die Unterstützung des Vertragsprinzips durch Aspekte zu untersuchen.

Erschwerung des Vertragsprinzips. Wie im Abschnitt 14.4 gezeigt, können Aspekte die Einhaltung von Verträgen kompromittieren, ohne dass den beteiligten Modulen Vertragsbruch vorgeworfen werden kann. Auch dieses Thema wird in der Literatur mit Ausnahme von Lagaisse, Joosen, De Win (2004) kaum behandelt. Dabei wäre es von grossem Nutzen, wenn das Vertragsprinzip auf Aspekte ausgedehnt würde, sei dies durch Spracherweiterungen, Werkzeugunterstützung oder Programmierrichtlinien.

16.5 Re-Engineering

Die Frage, wie „scattered“ Code-Fragmente von Crosscutting Concerns (so genannte *verborgene Aspekte*) in bestehenden Software-Systemen identifiziert, dokumentiert und ggf. als Aspekte neu implementiert werden können, wurde im Abschnitt 14.6 angeschnitten. In diesem Bereich sind in der aspektorientierten Literatur – mit Ausnahme von Hannemann, Kiczales (2001) – keinerlei Ansätze auszumachen. Dies könnte darauf zurückzuführen sein, dass ein solches Unterfangen als praktisch aussichtslos beurteilt wird.

16.6 Praktische Erfahrungen

Praktische Erfahrungen mit der Aspektorientierung scheinen weitgehend zu fehlen. Das Funktionieren von Konzepten und Sprachen wird in der Literatur fast ausschliesslich anhand von konstruierten Fallstudien illustriert. So einleuchtend die Konzepte und Sprachen dort erscheinen mögen, im harten Praxiseinsatz haben sie sich offensichtlich noch nicht bewährt. Dafür mag es zu früh sein, dauert es doch erfahrungsgemäss in den Informatik mehr als 10 Jahre, bis sich ein theoretisches Konzept in der Praxis durchzusetzen vermag.

Teil IV: Zusammenfassung

Der Teil IV fasst die im Laufe dieser Arbeit gewonnenen Erkenntnisse zusammen und versucht, sie kritisch zu würdigen.

17. Fazit

Dieses abschliessende Kapitel geht der Frage nach, welche Erkenntnisse im Laufe dieser Arbeit gewonnen werden konnten, wie diese Erkenntnisse zu beurteilen sind, wo noch Forschungsbedarf besteht und schliesslich welche Bedeutung der Aspektorientierung generell beigemessen wird.

17.1 Erkenntnisse

17.1.1 Begriffe

Die Begriffswelt der Aspektorientierung ist keineswegs so gefestigt wie es den Anschein macht. In einigen Ansätzen (z.B. Aspektorientierte Anforderungen mit der UML, siehe im Abschnitt 12.4) werden nicht-funktionale Anforderungen mit Crosscutting Concerns gleichgesetzt. In anderen Ansätzen (z.B. Theme/Doc, siehe im Abschnitt 12.10) ist schon in der Anforderungsphase von Aspekten die Rede. Die im Teil I dieser Arbeit gemachten Begriffsdefinitionen zwingen zu einer klaren Unterscheidung von Concerns, Anforderungen und Aspekten. Dass diese Unterscheidung sinnvoll und auch praktikabel ist, zeigt der Teil III, in dem diese Begriffe definitionsgemäss angewendet wurden. Wenn diese Arbeit zur Klärung der Begriffe der Aspektorientierung beigetragen hat, ohne die bestehende Begriffswelt allzu sehr zu verändern, hat sie eines ihrer wichtigsten Ziele erreicht.

Im Abschnitt 7.3 konnte gezeigt werden, dass Aspekte – mit kleinen Abweichungen – nach den gleichen Kriterien modularisiert werden sollten wie Komponenten. Damit können sowohl Aspekte als auch Komponenten als Unterbegriffe von Modul betrachtet werden. Die Forderung nach funktionaler Kohäsion von Aspekten ist als Vorschlag zu betrachten. Es wäre interessant zu wissen, ob diese Forderung auch einer vertieften Betrachtung standhält.

Die im Abschnitt 7.4 beschriebenen Eigenschaften von Aspekten müssen teilweise relativiert werden, sobald sie im Konflikt zu anderen Prinzipien des Software Engineering geraten. Dies ist beispielsweise bei der Forderung nach transparenten Aspekten der Fall: Sie steht im Widerspruch zur Forderung von Lagaisse, Joosen, De Win (2004) nach einem Aspektintegrationsvertrag im Rahmen des Vertragsprinzips.

17.1.2 Ansätze zur aspektorientierten Software-Entwicklung

Die Ansätze zur aspektorientierten Software-Entwicklung sind im Teil II dieser Arbeit beschrieben. Bei den Ansätzen der aspektorientierten Programmierung ist ein klarer Trend in Richtung von AspectJ zu beobachten (siehe auch im Abschnitt 15.2). AspectJ ist der reifste, mächtigste und universellste aller Ansätze, hat jedoch nach wie vor noch gewisse Lücken. Bei den Ansätzen der frühen Entwicklungsphasen fehlt ein solcher Trend, so dass eine grosse Vielfalt verschiedener Vorgehensweisen auszumachen ist. Was die Spezifikationssprache betrifft, scheint sich die UML durchzusetzen, wobei die verwendeten Spracherweiterungen noch uneinheitlich sind. Angesichts der Dominanz von AspectJ im Bereich der aspektorientierten Programmierung ist anzunehmen, dass sich in den frühen Entwicklungsphasen Ansätze durchsetzen werden, deren Modellkonstrukte sich möglichst elegant auf AspectJ-Konstrukte abbilden lassen.

Um die teilweise doch sehr unterschiedlichen Ansätze der aspektorientierten Programmierung fundiert vergleichen zu können, wären umfangreiche empirische Untersuchungen erforderlich, welche die Zeit und die Möglichkeiten dieser Arbeit übersteigen. Im Abschnitt 9.10 finden sich erste derartige Untersuchungen. Dennoch wäre es wünschbar, mehr darüber zu erfahren, wie sich die Ansätze hinsichtlich Performance, Wiederverwendbarkeit, Modularisierung, Pflegbarkeit etc. voneinander und von konventionellen Ansätzen unterscheiden.

17.1.3 Aspektorientierung im Software Engineering

Damit aspektorientierte Software-Systeme nicht nur implementiert, sondern auch spezifiziert und entworfen werden können, müssen die Prozesse und Sprachen der frühen Phasen um aspektorientierte Konzepte erweitert werden. Der Teil III dieser Arbeit versucht zu zeigen, an welchen Stellen des Entwicklungsprozesses welche Erweiterungen und zusätzlichen Überlegungen erforderlich sind. Da sowohl das Software Engineering als auch die Aspektorientierung breite, vielfältige Gebiete sind, konnte diese Arbeit im besten Fall Denkanstösse geben und Lösungsskizzen aufzeigen. Namentlich die folgenden Bereiche bedürfen einer vertieften Betrachtung:

- **Identifikation, Darstellung und Beschreibung von Crosscutting Concerns**. Die Vorschläge für die Identifikation, Darstellung und Beschreibung von Crosscutting Concerns aus dem Kapitel 13 sind in keiner Weise erprobt und folglich als Denkanstösse zu verstehen. Sie gehen von der Annahme aus, dass in der Anforderungsphase nur diejenigen Konzepte der Aspektorientierung zu behandeln sind, die sich als essenziell erweisen. Diese Essenz wurde im Abschnitt 9.12 aus einem Vergleich der Ansätze der aspektorientierten Programmierung gewonnen und sollte einer kritischen Prüfung unterzogen werden.
- **Symmetrisches und asymmetrisches Paradigma**. Der Unterschied zwischen dem symmetrischen und dem asymmetrischen Paradigma konnte aus zeitlichen Gründen nicht so ausführlich analysiert werden wie erhofft. Hier besteht noch Untersuchungsbedarf. Harrison, Ossher, Tarr (2002) geben dem symmetrischen Paradigma den Vorzug, sie sind aber als Urheber einiger symmetrischer Ansätze in dieser Angelegenheit nicht objektiv. Anderseits beruht die überwiegende Mehrheit der Ansätze auf dem asymmetrischen Paradigma.
- **Vertragsprinzip**. Aspekte können die Umsetzung des Vertragsprinzips erschweren, indem sie Zusicherungen von Komponenten verletzen, die sie quer schneiden. Die Auswirkungen der Aspektorientierung auf das Vertragsprinzip wurden zwar in den Abschnitten 14.4 und 16.4 angeschnitten, sie sind aber so kompliziert und vielschichtig, dass eine vertiefte Analyse den zeitlichen Rahmen dieser Arbeit gesprengt hätte. Das Thema ist jedoch von einiger Brisanz und sollte unbedingt weiter erforscht werden.
- **Prüfung**. Die Prüfung von aspektorientierten Systemen wird in den Abschnitten 13.9 und 16.3 kurz behandelt. Wie weit sie sich tatsächlich von der Prüfung von konventionellen Systemen unterscheidet, sollte ebenfalls Thema einer vertieften Analyse sein.
- **Re-Engineering**. Das Thema wird zwar in den Abschnitten 14.6 und 16.5 behandelt, es wird aber insgesamt als eher unwahrscheinlich beurteilt, dass in der Praxis verborgene Aspekte aus konventionellen Software-Systemen extrahiert und aspektorientiert implementiert werden. Zu hohe Kosten und Risiken stehen einem zu kleinen Nutzen gegenüber.

17.2 Bedeutung der Aspektorientierung

Welche Bedeutung kommt nun der Aspektorientierung innerhalb der gesamten Informationstechnologie zu? Heinrich (1999) unterscheidet vier Technologiearten:

- **Basistechnologie** (engl. basic technology) ist eine vorhandene Technologie, deren Veränderungspotenzial weitgehend ausgeschöpft ist.
- **Schlüsseltechnologie** (engl. key technology) ist eine vorhandene Technologie, die noch über ein erhebliches Veränderungspotenzial verfügt.
- **Schrittmachertechnologie** (engl. pacemaker technology) ist eine Technologie, die sich im Entwicklungsstadium befindet; sie lässt ein erhebliches Veränderungspotenzial erwarten.
- **Zukunftstechnologie** (engl. future technology) ist eine sich abzeichnende Technologie, von der ein erhebliches Veränderungspotenzial erwartet wird.

Aufgrund der Erkenntnisse dieser Arbeit lässt sich die Aspektorientierung zwischen Schlüssel- und Schrittmachertechnologie einordnen. Einerseits sind Ansätze der aspektorientierten Programmierung durchaus vorhanden, anderseits befinden sich die Ansätze der aspektorientierten Anforderungstechnik, der aspektorientierten Architektur und des aspektorientierten Entwurfs noch im Entwicklungsstadium. Der Sprung in die Praxis ist noch nicht geschafft. Das Veränderungspotenzial der Aspektorientierung wird in jedem Fall als erheblich betrachtet: Die Aspektorientierung hat das Potenzial, das Prinzip der Separation of Concerns umzusetzen und dadurch besser modularisierte Systeme zu ermöglichen.

Anhang

Im Anhang A sind die Fragen aus der Aufgabenstellung detailliert aufgeführt. Im Anhang B befindet sich das Literaturverzeichnis.

A. Fragen aus der Aufgabenstellung

Zusammenfassend sind hier die Fragen aus der Aufgabenstellung aufgeführt, auf die diese Arbeit Antworten suchte. Es handelt sich nicht um Zitate, sondern um eine eigene Interpretation der Fragen.

- **Begriffe**. Wie sind Aspekte definiert? Gibt es in der Literatur brauchbare Definitionen? Wie lassen sich Aspekte von Nicht-Aspekten abgrenzen? Welches sind die wichtigsten Begriffe im Umfeld von Aspekten? Wie hängen all diese Begriffe zusammen? Wie können Aspekte kategorisiert werden? Wie unterscheiden sich der Aspektbegriff von ADORA (siehe Joos, 1999) und der Aspektbegriff der Aspektorientierung? Die Antworten auf diese Fragen sind im Teil I zu finden.
- **Aspekte in der Anforderungsphase**. Wie können Aspekte in der Anforderungsphase von funktionalen bzw. nicht-funktionalen Anforderungen abgeleitet werden? Gibt es Aspekte, die nicht aus funktionalen bzw. nicht-funktionalen Anforderungen der Anforderungsphase abgeleitet werden können? Die Antworten auf diese Fragen sind im Teil I zu finden.

  Zu welchem Zeitpunkt und wie sollen die Aspekte in der Anforderungsphase identifiziert werden? Wie können Aspekte in der Anforderungsphase dargestellt und beschrieben werden? Wie weit eignen sich gebräuchliche Modellierungstechniken aus allen Phasen des Entwicklungsprozesses für die Darstellung und Beschreibung von Aspekten in der Anforderungsphase? Wie können Aspekte und Nicht-Aspekte in der Anforderungsphase zusammengeführt werden? Die Antworten auf diese Fragen sind im Teil III, Kapitel 13, zu finden.
- **Aspekte in der Entwurfs- und Implementierungsphase**. Wie können die in der Anforderungsphase spezifizierten Aspekte in die Architektur, den Entwurf und die Implementierung überführt werden? Wie können die in der Anforderungsphase spezifizierten Aspekte die Entwurfs- und Implementierungsphase unterstützen, und welche Voraussetzungen müssen dafür erfüllt sein? Die Antworten auf diese Fragen sind im Teil III, Kapitel 13, zu finden.
- **Aspekte im konventionellen Software Engineering**. Wie beeinflussen sich Aspekte und konventionelle Software Engineering-Konzepte? Wo sind Anpassungen an konventionellen Software Engineering-Konzepten erforderlich, wenn die Aspektorientierung eingesetzt wird? Die Antworten auf diese Fragen sind im Teil III, Kapitel 14, zu finden.

Daneben gab die Aufgabenstellung vor, den Stand der Forschung zusammenzufassen (Teil II) sowie Trends (Teil III, Kapitel 15) und Lücken (Teil III, Kapitel 16) im Forschungsgebiet zu identifizieren.

B. Literaturverzeichnis

Araujo, J., A. Moreira, I. Brito, A. Rashid (2002): Aspect-Oriented Requirements with UML, Workshop Paper, Aspect Oriented Modeling with UML, 5th International Conference on the Unified Modeling Language and its Applications (UML 2002), Dresden, Germany

Araujo, J., P. Coutinho (2003): Identifying Aspectual Use Cases Using a Viewpoint-Oriented Requirements Method, Workshop Paper, Early Aspects 2003, 2nd Aspect-Oriented Software Development Conference (AOSD 2003), Boston, Massachusetts, USA

Araujo, J., J. Whittle, D.-K. Kim (2004): Modeling, Composing and Validating Scenario-Based Requirements with Aspects, to be published in: **Proceedings of the 12th IEEE International Requirements Engineering Conference (RE 2004)**, Kyoto, Japan

Atkinson, C., T. Kühne (2003): Aspect-Oriented Development with Stratified Frameworks, in: **IEEE Software**, www.computer.org/software (22.07.2004), January 2003, 81-89

Baniassad, E., S. Clarke (2004): Finding Aspects in Requirements with Theme/Doc, Workshop Paper, Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Beck, K. (2000): **Extreme Programming, Das Manifest**, Addison-Wesley

Bergmans, L., M. Aksit (2001a): Composing Crosscutting Concerns using Composition Filters, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 51-57

Bergmans, L., M. Aksit (2001b): Composing Multiple Concerns Using Composition Filters, http://trese.cs.utwente.nl/ (24.07.2004)

Black, A., N. Hutchinson, E. Jul, H. Levy (1986): Object Structure in the Emerald System, in **Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)**, Portland, Oregon, USA, ACM Press, 78-86

Blair, L., G. Blair (1999): A Tool Suite to Support Aspect-Oriented Specification, in: **Proceedings of the 3rd Aspect-Oriented Programming Workshop at the 13th European Conference on Object Oriented Programming (ECOOP'99)**, Lisbon, Portugal, Summary in: **Object-Oriented Technology, ECOOP'99 Workshop Reader**, Springer Verlag, LNCS 1743, 296-297

Bodkin, R. (2004): Enterprise Security Aspects, Workshop Paper, AOSDSEC 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Bogdan, C. (2004): Facets of Concerns, Workshop Paper, Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Boström, G. (2004): Database encryption as an Aspect, AOSDSEC 2004, Workshop Paper, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Brito, I., A. Moreira, J. Araujo (2002): A requirements model for quality attributes, Workshop Paper, Early Aspects 2002, 1st Aspect-Oriented Software Development Conference (AOSD 2002), Enschede, The Netherlands

Brito, I., A. Moreira (2004): Integrating the NFR framework in a RE model, Workshop Paper, Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Bryant, A., A. Catton, K. De Volder, G.C. Murphy (2002): Explicit Programming, in: **Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)**, Enschede, The Netherlands, ACM Press, 10-18

Burge, J., D.C. Brown (2002): NFRs: Fact or Fiction?, Technical Report, WPI-CS-TR-02-01, Artificial Intelligence in Design Research Group, Computer Science Department, Worcester Polytechnic University, Massachusetts, USA

Chidamber, S.R., C.F. Kemerer (1994): A Metric Suite for Object Oriented Design, in: **IEEE Transactions on Software Engineering**, Vol. 20, Issue 6, IEEE Press, Piscataway, New Jersey, USA, 476-493

Chung, L., B.A. Nixon, E. Yu, J. Mylopoulos (2000): **Non-functional Requirements in Software Engineering**, Kluwer Academic Publishers, Boston, Dordrecht, London

Clarke, S., W. Harrison, H. Ossher, P. Tarr (1999): Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code, in: **Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)**, Denver, Colorado, USA, 325-339

Clarke, S., R.J. Walker (2001a): Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J, Technical Report, TCD-CS-2001-15 and UBC-CS-TR-2001-05, Trinity College, Dublin, Ireland, and University of British Columbia, Vancouver, Canada

Clarke, S., R.J. Walker (2001b): Composition Patterns: An Approach to Designing Reusable Aspects, in: **Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)**, Toronto, Ontario, Canada, IEEE Computer Society Press, 5-14

Constantinides, C.A., A. Bader, T.H. Elrad, M.E. Fayad, P. Netinant (2000): Designing an Aspect-Oriented Framework in an Object-Oriented Environment, in: **ACM Computing Surveys (CSUR)**, Vol. 32, Issue 1es, ACM Press, Article No. 41

Deininger, M., H. Lichter, J. Ludewig, K. Schneider (1992): **Studien-Arbeiten: ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktor-arbeiten am Beispiel Informatik**, vdf Verlag der Fachvereine Zürich, B. G. Teubner Stuttgart

Diaz Pace, J.A., M.R. Campo (2001): Analyzing the Role of Aspects in Software Design, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 67-73

Dijkstra, E.W. (1968): Go To Statement Considered Harmful, in: **Communications of the ACM**, March 1968, Vol. 11, No. 3, 147-148

Dijkstra, E.W. (1976): **A Discipline of Programming**, Prentice-Hall, Englewood Cliffs, New Jersey

Duden (1997): **Duden, Das Herkunftswörterbuch, Etymologie der deutschen Sprache**, Band 7, Dudenverlag, Mannheim, Leipzig, Wien, Zürich

Elrad, T., R.E. Filman, A. Bader, Guest Editors (2001): Aspect-Oriented Programming, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 29-32

Elrad, T., M. Aksit, G. Kiczales, K. Lieberherr, H. Ossher, Panelists (2001): Discussing Aspects of AOP, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 33-38

Gal, A., O. Spinczyk, W. Schröder-Preikschat (2002): On Aspect-Orientation in Distributed Real-time Dependable Systems, in: **Proceedings of the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)**, San Diego, California, USA, IEEE Computer Society, 261-270

Gamma, E., R. Helm, R. Johnson, J. Vlissides (1996): **Entwurfsmuster, Elemente wieder-verwendbarer objektorientierter Software**, Addison-Wesley

Georg, G., I. Ray, R. France (2002): Using Aspects to Design a Secure System, in: **Proceedings of the 8th International Conference on Engineering Complex Computing Systems (ICECCS 2002)**, Greenbelt, Maryland, USA, ACM Press, 117-128

Glinz, M. (2003a): Spezifikation und Entwurf von Software, Vorlesungsskript, Wintersemester 2003/04, Institut für Informatik, Universität Zürich, Schweiz

Glinz, M. (2003b): Software Engineering I, Vorlesungsskript, Wintersemester 2003/04, Institut für Informatik, Universität Zürich, Schweiz

Glinz, M. (2004): KV Software Engineering, Vorlesungsskript, Sommersemester 2004, Institut für Informatik, Universität Zürich, Schweiz

Groher, I., T. Baumgarth (2004): Aspect-Orientation from Design to Code, Workshop Paper, Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Grundy, J. (1999): Aspect-oriented Requirements Engineering for Component-based Software Systems, in: **Proceedings of the 4th IEEE International Symposium on Requirements Engineering (RE 1999),** Limerick, Ireland, IEEE Computer Society Press, 84-91

Hannemann, J., G. Kiczales (2001): Overcoming the Prevalent Decomposition of Legacy Code, Workshop on Advanced Separation of Concerns at the 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Ontario, Canada

Harrison, W.H., H.L. Ossher, P.L. Tarr (2002): Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition, IBM Research Report, RC22685 (W0212-147), IBM Research Division, Yorktown Heights, New York

Heinrich, L.J. (1999): **Informationsmanagement**, 6. Auflage, Oldenbourg Verlag, München

Huang, M., C. Wang, L. Zhang (2004): Toward a Reusable and Generic Security Aspect Library, Workshop Paper, AOSDSEC 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

IEEE (2000): IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std. 1471-2000

Ishio, T., T. Kamiya, S. Kusumoto, K. Inoue (2004): Assertion with Aspect, Workshop Paper, SPLAT 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

ISO (1989): LOTOS, A formal description technique based on the temporal ordering of observational behaviour, International Standard 8807:1989, www.iso.org (22.07.2004)

Jackson, M. (2000): **Problem Frames: Analyzing and Structuring Software Development Problems**, Addison-Wesley

Jacobson, I., G. Booch, J. Rumbaugh (1999): **The Unified Software Development Process**, Addison-Wesley

Jacobson, I. (2002): Use Cases – Yesterday, Today, and Tomorrow, Rational Software, 11/26/2002

Jacobson, I. (2003): Use Cases and Aspects – Working Seamlessly Together, in: **Journal of Object Technology**, Vol. 2, No. 4, July-August 2003, 7-28

Joos, S. (1999): **ADORA-L - Eine Modellierungssprache zur Spezifikation von Software-Anforderungen**, Dissertation, Institut für Informatik, Universität Zürich, Schweiz

Katz, S. (2004): Diagnosis of Harmful Aspects Using Regression Verification, Workshop Paper, FOAL 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Kawauchi, K., H. Masuhara (2004): Dataflow Pointcut for Integrity Concerns, Workshop Paper, AOSDSEC 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin (1997): Aspect-Oriented Programming, in: **Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)**, Jyväskylä, Finland, Lecture Notes in Computer Science 1241, Springer-Verlag, 220-242

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold (2001a): An Overview of AspectJ, in: **Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)**, Budapest ; Hungary, Lecture Notes in Computer Science 2072, Springer-Verlag, 327-353

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold (2001b): Getting Started with AspectJ, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 59-65

Klösch, R., H. Gall (1995): **Objektorientiertes Reverse Engineering: Von klassischer zu objektorientierter Software**, Springer-Verlag, Berlin, Heidelberg

Laddad, R. (2003): **AspectJ in Action, Practical Aspect-Oriented Programming**, Manning Publications Co.

Lagaisse, B., W. Joosen, B. De Win (2004): Managing semantic interference with aspect integration contracts, Workshop Paper, SPLAT 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

van Lamsweerde, A. (2001): Goal-Oriented Requirements Engineering: A Guided Tour, in: **Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)**, Toronto, Canada, IEEE Computer Society Press, 249-263

Laney R., J. Van der Linden, P. Thomas (2004): Evolution of Aspects for Legacy System Security Concerns, Workshop Paper, AOSDSEC 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Link Everything Online (LEO) (2004): Deutsch-Englisches Wörterbuch, Fakultät für Informatik der Technischen Universität München, www.leo.org (22.07.2004)

Lieberherr, K., D. Orleans, J. Ovlinger (2001): Aspect-Oriented Programming with Adaptive Methods, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 39-41

McMenamin, S.M., J.F. Palmer (1988): **Strukturierte Systemanalyse**, Carl Hanser Verlag München Wien

Merriam-Webster (2004): Merriam-Webster Online Dictionary, www.merriam-webster.com (22.07.2004)

Meyer, B. (1997): **Object-Oriented Software Construction**, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey

Mili, H., A. Elkharraz, H. Mcheick (2004): Concerned about Separation, Workshop Paper, Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Moreira, A., J. Araujo, I. Brito (2002): Crosscutting Quality Attributes for Requirements Engineering, in: **Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)**, Ischia, Italy, ACM Press, 167-174

Mousavi, M., G. Russello, M. Chaudron, M.A. Reniers, T. Basten, A. Corsaro, S. Shukla, R. Gupta, D.C. Schmidt (2002): Aspects + GAMMA = AspectGAMMA, A Formal Framework for Aspect-Oriented Specification, Workshop Paper, Early Aspects 2002, 1st Aspect-Oriented Software Development Conference (AOSD 2002), Enschede, The Netherlands

Murphy, G.C., R.J. Walker, E.L.A. Baniassad, M.P. Robillard, A. Lai, M.A. Kersten (2001): Does Aspect-Oriented Programming Work?, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 75-77

Netinant, P., T. Elrad, M.E. Fayad (2001): A Layered Approach to Building Open Aspect-Oriented Systems, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 83-85

Nishizawa, M., S. Chiba, M. Tatsubori (2004): Remote Pointcut – A Language Construct for Distributed AOP, in: **Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)**, Lancaster, UK, ACM Press, 7-15

Oestereich, B. (2001): **Objektorientierte Softwareentwicklung, Analyse und Design mit der Unified Modeling Language**, Oldenbourg Verlag München Wien

Orleans, D., K. Lieberherr (2001): DJ: Dynamic Adaptive Programming in Java, in: **Proceedings of the 3rd International Conference on Meta-level Architectures and Separation of Crosscutting Concerns (Reflection 2001)**, Kyoto, Japan, Springer Verlag, 73-80

Ossher, H., P. Tarr (2001): Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 43-50

Parnas, D.L. (1972): On the Criteria To Be Used in Decomposing Systems into Modules, in: **Communications of the ACM**, December 1972, Vol. 15, No. 12, 1053-1058

Rashid, A., P. Sawyer, A. Moreira, J. Araujo (2002): Early Aspects: A Model for Aspect-Oriented Requirements Engineering, in: **Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE 2002)**, Essen, Germany, IEEE Computer Society Press, 199-202

Rashid, A., A. Moreira, J. Araujo (2003): Modularisation and Composition of Aspectual Requirements, in: **Proceedings of the 2nd International Conference on Aspect-oriented Software Development (AOSD 2003)**, Boston, Massachusetts, USA, ACM Press, 11-20

Rashid, A., R. Chitchyan (2003): Persistence as an Aspect, in: **Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)**, Boston, Massachusetts, USA, ACM Press, 120-129

Sakurai K., H. Masuhara, N. Ubayashi, S. Matsuura, S. Komiya (2004): Association Aspects, in: **Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)**, Lancaster, UK, ACM Press, 16-25

Schach, S.R. (1999): **Classical and Object-Oriented Software Engineering with UML and Java**, 4th edition, McGraw-Hill International Editions

Scholz, C. (2000): **Personalmanagement**, 5. Auflage, Verlag Franz Vahlen GmbH, München

Sommerville, I., P. Sawyer (1997): **Requirements Engineering – A Good Practice Guide**, John Wiley & Sons

Sommerville, I. (2001): **Software Engineering**, 6. Auflage, Pearson Studium

Sousa, G., S. Soares, P. Borba, J. Castro (2004): Separation of Crosscutting Concerns from Requirements to Design: Adapting and Use Case Driven Approach, Workshop Paper, Early Aspects 2004, 3rd Aspect-Oriented Software Development Conference (AOSD 2004), Lancaster, UK

Sullivan, G.T. (2001): Aspect-Oriented Programming Using Reflection and Metaobject Protocols, in: **Communications of the ACM**, October 2001, Vol. 44, No. 10, 95-97

Sung, J.J., K. Lieberherr (2002): DAJ: A Case Study of Extending AspectJ, Technical Report, NU-CCS-02-16, Northeastern University, Boston, Massachusetts, USA

Sutton Jr., S.M., I. Rouvellou (2002): Modeling of Software Concerns in Cosmos, in: **Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)**, Enschede, The Netherlands, ACM Press, 127-133

Suzuki, J., Y. Yamamoto (1999): Extending UML with Aspects: Aspect Support in the Design Phase, in: **Proceedings of the 3rd Aspect-Oriented Programming Workshop at the 13th European Conference on Object Oriented Programming (ECOOP'99)**, Lisbon, Portugal, Summary in: **Object-Oriented Technology, ECOOP'99 Workshop Reader**, Springer Verlag, LNCS 1743, 299-300

Tarr, P., H. Ossher, W. Harrison, S.M. Sutton Jr. (1999): N Degrees of Separation: Multi-Dimensional Separation of Concerns, in: **Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)**, Los Angeles, California, USA, IEEE Computer Society Press, 107-119

Tsang, S.L., S. Clarke, E. Baniassad (2004): Object Metrics for Aspect Systems: Limiting Empirical Inference Based on Modularity, submitted to the ECOOP 2004

Völter, M. (2001): Metaprogramming, Introspection, Reflection, was ist das? und was bringt‘s?, Mathema AG, www.voelter.de/data/presentations/meta.ppt (22.07.2004)

Zakaria, A.A., H. Hosny (2003): Metrics for Aspect-Oriented Software Design, Workshop Paper, 3rd International Workshop on Aspect-Oriented Modeling (AOM 2003), 2nd Aspect-Oriented Software Development Conference (AOSD 2003), Boston, Massachusetts, USA