Architektur komponentenbasierter Systeme mit LOOM: Aspekte, Muster, Werkzeuge

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Doctor rerum naturalium)

eingereicht am
Fachbereich Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dipl.-Inf. Wolfgang Schult

Betreuer: Prof. Dr. Andreas Polze
Hasso-Plattner-Institut für Softwaresystemtechnik
Universität Potsdam

Zusammenfassung

Eine gute Modularisierung von Software ist die Grundvoraussetzung für eine verteilte Anwendungsentwicklung. Das wurde bereits von Parnas (1972) beschrieben. Bei einem Modul handelt es sich um eine abgeschlossene Funktionseinheit, die einen Dienst an das System liefert und bei Bedarf ausgetauscht werden kann.

Wünschenswert ist es, wenn die Belange einer Software - also Dinge, die für die Entwicklung, den Betrieb oder anderweitig von Interesse für die Software sind - jeweils separat in einzelnen Modulen implementiert werden können. Gängige Formalismen unterstützen das nur beschränkt: Oft müssen Belange über mehrere Module verstreut implementiert werden oder ein Modul ist mit der Implementation mehrer Belange durchsetzt. In der Forschung wurde bereits nachgewiesen, dass das zu einer höheren Zahl von Programmierfehlern führt und die Softwarequalität verschlechtert.

Die Arbeit präsentiert mit dem LOOM-Programmierparadigma einen neuen Ansatz, mit dem das Problem der Streuung und Durchsetzung unter gleichzeitiger Beibehaltung der Modulgrenzen gelöst werden konnte. Die Interaktion zwischen Aspekten und anderen Modulen erfolgt bei LOOM stets über definierte Schnittstellen, die allen Beteiligten bekannt sind. LOOM ist ein ganzheitliches Paradigma für die Entwicklung komponentenbasierender Softwaresysteme, das - angefangen beim Entwurf, über die Programmierung, dem Erstellen und Testen, bis zum Betrieb - in jeder Phase des Softwareentwicklungsprozesses einen Beitrag zur Effizienz und Qualitätssteigerung liefert.

Abstract

In 1972, Parnas came to the conclusion that a good modularization is both a necessity and a solid foundation for successful distributed software systems engineering. A module comprises a self-contained functional entity that serves as a service to a system and that can simply be replaced if necessary.

Concerns of a software system are aspects that are necessary for the development, maintenance, or operation of the system but are not strictly related to a particular module. They are rather scattered (or cross-cutting) throughout the modules comprising the software system. Even worse, some modules may implement more than one concern. It is thus desirable that such concerns can be independently implemented in a separate module. Recent research has shown that this issue leads to a higher number of errors and negatively affects software quality. However, solutions to this issue are rather limited.

This thesis presents the LOOM programming paradigm that solves the problem of cross-cutting concerns while maintaining the boundaries of participating modules. The interaction between modules and aspects is established by interfaces known to all participants. LOOM is an integral paradigm for the development of component based software systems. LOOM increased efficiency and quality in all the phases of the software development process, i.e. design, implementation, building and testing, and maintenance.

*Für meinen Vater Hans-Jürgen Schult (*10.11.1948 - †01.03.1996),*
der mich bereits in meiner Kindheit für Computer und das Programmieren begeistert hat.

Danksagung

Mein besonderer Dank gilt Herrn Prof. Dr. rer. nat. Andreas Polze für die Betreuung und die Unterstützung bei der Fertigstellung der Dissertationsschrift.

Insbesondere Herrn Dr. rer. nat. Martin von Löwis, Herrn Dr. rer. nat Andreas Rasche und Dipl.-Inf. Alexander Schmidt möchte ich für die exzellente und freundschaftliche Zusammenarbeit und die vielen Anregungen danken. Auch bei der kritischen Evaluation und Diskussion der Ergebnisse waren sie unverzichtbare Ansprechpartner.

Weiterhin möchte ich mich bei meiner Fachgruppe für Betriebssysteme und Midddleware, namentlich Dipl.-Inf. Bernhard Rabe und M. Sc. Michael Schöbel für die kollegiale und kooperative Zusammenarbeit bedanken.

Ich danke Herrn Dr. Stefan Ried, Herrn Jürgen Kraß sowie Herrn Robert Leaman, die mir es ermöglicht haben, dass ich neben meiner Tätigkeit bei der Deutschen Post IT-Services GmbH mein Promotionsvorhaben durchführen konnte.

Außerdem gilt der Dank den Studenten, die mit ihren Ideen und ihrem Engagement geholfen haben, das LOOM-Projekt während der letzten Jahre voranzubringen. Hier möchte ich besonders Herrn Kai Köhne, Frau Janin Jeske, Herrn Benjamin Schüler und Herrn Frank Feinbube nennen.

Besonders danke ich meiner Mutter Helga Schult für die unermüdliche sprachliche Überarbeitung meiner Dissertationsschrift.

Schließlich möchte ich meiner Familie für ihre Liebe und selbstlose Unterstützung danken. Ohne deren Aufmunterung während dieser für uns anstrengenden Zeit wäre diese Dissertation kaum zustande gekommen.

Inhaltsverzeichnis

1 Einleitung **11**
1.1 Begriffe 12
1.2 Aspektorientierung und Modularität 13
1.3 Die Evolution von aspektorientierten Programmen 14
1.4 Sichtbarkeit von Aspekten 15
1.5 Statische und dynamische Joinpoints 17
1.6 Aspekte und Datenzugriff 17
1.7 Schlussfolgerungen 18
1.8 Der wissenschaftliche Beitrag dieser Arbeit 19
1.9 Hinweise für den Leser 21

2 Neue Konzepte zur Modularisierung von Software **23**
2.1 Modularisierung in objektorientierten Sprachen 23
2.2 Die Komposition objektorientierter Anwendungen 27
2.3 Der Annotationen in objektorientierten Sprachen 28
2.4 Der LOOM-Ansatz 29
2.5 Die Annotationsrelation 31
2.6 Joinpoints in LOOM 35
2.7 Die Modellierungssprache LOOM.UML 36
2.7.1 Die Definition von Aspekten 37
2.7.2 Annotationen in LOOM.UML 38
2.7.3 Aspekte zur Laufzeit 39
2.7.4 Definition von Advices 40
2.7.5 Ausführung von Advices 42
2.7.6 Definition von Introduktionen 44
2.7.7 Joinpoint-Variablen 45
2.8 Weitere LOOM-Konzepte 47
2.8.1 Der Ausführungskontext von Advices und Introduktionen 47
2.8.2 Advice-Initialisierer 48
2.8.3 Nachrichtenmanipulation 48
2.9 Zusammenfassung 50

3 Zwei LOOM-Aspektweber **53**
3.1 Die Common Language Infrastructure 53
3.2 Metaprogrammierung in der CLI 54
3.3 Die Geschichte des LOOM.NET-Projektes 55
3.4 Eine Abbildung der LOOM.NET-Spracherweiterung auf die CLI 56
3.4.1 Definition einer Aspektklasse 56
3.4.2 Advices und Introduktionen 57
3.4.3 Joinpoint-Parameter und der Ausführungskontext 58
3.4.4 Die Definition von Joinpoint-Variablen 58
3.4.5 Ein Beispiel 59
3.4.6 Das LOOM.NET-Metamodell 59

3.5 Der dynamische Aspektweber Rapier-Loom.Net 60
3.5.1 Der Stellvertreter 61
3.5.2 Schnelle Exemplarbildung 63
3.5.3 Serialisierung von Objekten 66
3.5.4 Der Verwebungsprozess 68
3.6 Der statische Aspektweber Gripper-Loom.Net 71
3.6.1 Microsoft Phoenix 73
3.6.2 Verweben von Assemblies 73
3.6.3 Bewertung des Aspektwebers 74
3.7 Ein Vergleich beider Aspektweber 74
3.8 Zusammenfassung 78

4 Entwurfsmuster 79
4.1 Eine Metrik zur Bewertung der Entwurfsmuster 79
4.2 Das Einzelstückmuster 81
4.2.1 Die Struktur des LOOM-Einzelstückmusters 81
4.2.2 Beispiel 82
4.2.3 Vergleich zu anderen Lösungen 82
4.2.4 Bewertung des Musters 83
4.3 Das Dekorierermuster 83
4.3.1 Die Struktur des LOOM-Dekorierermusters 83
4.3.2 Beispiel 85
4.3.3 Bewertung des Musters 85
4.4 Das Adaptermuster 86
4.4.1 Die Struktur des LOOM-Adaptermusters 88
4.4.2 Beispiel für einen aspektorientierten Adapter 89
4.4.3 Bewertung des Musters 89
4.5 Das Klassenkompositionsmuster 90
4.5.1 Struktur des LOOM-Klassenkompositionsmusters 90
4.5.2 Beispiel 91
4.5.3 Vergleich zu anderen Lösungen 91
4.5.4 Bewertung des Musters 92
4.6 Das Beobachtermuster 92
4.6.1 Die Struktur des LOOM-Beobachtermusters 93
4.6.2 Beispiel 94
4.6.3 Vergleich zu anderen Lösungen 95
4.6.4 Bewertung des Musters 95
4.7 Das Besuchermuster 96
4.7.1 Die Struktur des LOOM-Besuchermusters 99
4.7.2 Beispiel 99
4.7.3 Vergleich zu anderen Lösungen 101
4.7.4 Bewertung des Musters 103
4.8 Das Stellvertretermuster 104
4.8.1 Die Struktur des LOOM-Stellvertreters 105
4.8.2 Beispiel 106
4.8.3 Vergleich zu anderen Lösungen 108
4.8.4 Bewertung des Musters 110
4.9 Der Ereignisabonnent 110
4.9.1 Die Struktur des LOOM-Ereignisabonnenten 112
4.9.2 Beispiel 114
4.9.3 Varianten des Musters 115
4.9.4 Bewertung des Musters 116

4.10 Zusammenfassung 117

5 Programmierkonzepte 119
5.1 Kontextorientiertes Programmieren mit Context# 119
5.1.1 Kontextorientiertes Programmieren 119
5.1.2 Eine Umsetzung von COP mit LOOM.NET 121
5.1.3 Vergleich von Context# mit ContextJ 124
5.1.4 Implementationsdetails 125
5.1.5 Diskussion der Lösung 129
5.2 Design by Contract mit LOOM.NET 129
5.2.1 Design by Contract 131
5.2.2 Implementationsdetails 133
5.2.3 Diskussion der Lösung 137
5.3 Zusammenfassung 138

6 Projekte mit LOOM.NET 139
6.1 Eine Studie im industriellen Umfeld 139
6.1.1 Ziele der Studie 140
6.1.2 Ergebnisse der Analyse 141
6.1.3 Verbesserung von Logging und Tracing 142
6.1.4 Revision der Ausnahmebehandlung 143
6.1.5 Ergebnisse der Studie 146
6.2 Dynamische Softwareaktualisierung mit DSUP 146
6.2.1 Allgemeine Definitionen 148
6.2.2 Sicherstellen der Ausführungsintegrität 149
6.2.3 Sicherstellen der strukturellen Integrität 153
6.2.4 Umgang mit anwendungsspezifischen Invarianten 155
6.2.5 Implementationsdetails 158
6.2.6 Evaluation der Lösung 163
6.2.7 Diskussion der Lösung 164
6.3 Weitere Projekte mit LOOM.NET 165
6.3.1 Adapt.NET 165
6.3.2 Ein Managementframework für die Windows Fernwartungsschnittstelle 166
6.3.3 Das Distributed Control Lab 167
6.3.4 Ein Optimierungsframework für Komponenteninteraktionen 168
6.4 Zusammenfassung 168

7 Verwandte Arbeiten 169
7.1 Ansätze zur Trennung von Belangen 169
7.1.1 Subjektorientierte Programmierung und HyperJ 169
7.1.2 Das Composition-Filter-Modell 170
7.1.3 Mixins und Traits 172
7.1.4 Erweiterungsmethoden 173
7.2 Aspektorientierte Programmierung 173
7.2.1 AspectJ 174
7.2.2 JBoss-AOP 177
7.2.3 Spring 178
7.2.4 AspectS 179
7.2.5 Weitere Lösungen für die .NET-Plattform 179
7.2.6 Weitere Lösungen für Java 180
7.3 Aspektorientierte Modellierung 181
7.3.1 Der Ansatz von Suzuki und Yamamoto 182

7.3.2 Der Ansatz von Chavez und Lucena 182
7.3.3 Der Ansatz von Stein u.a. 182
7.3.4 Fuentes und Sánchez 184
7.4 Zusammenfassung 185

8 Zusammenfassung und Ausblick 187

Literaturverzeichnis 191

1 Einleitung

Während in den frühen Jahren der Computer noch „nah an der Maschine“ in der Sprache des Prozessors programmiert wurde, ist mit modernen Programmiersprachen und Kompilern heute ein Abstraktionsniveau erreicht, das es erlaubt, Programme weitgehend unabhängig vom unterliegenden System zu implementieren. Mit den Konzepten der Objektorientierten Programmierung, die in fast jeder aktuellen Programmiersprache zu finden sind, ist beispielsweise die Modellierung von Problemen der realen Welt einfach, da sich die realen Objekte auch in der Programmierung wiederfinden. Neben der Abstraktion ist aber auch ein weiteres Merkmal - die Modularisierung - relevant, um den Anforderungen von Softwareentwicklungsprozessen gerecht zu werden.

Bereits [Parnas 1972b] erkannte, dass eine gute Modularisierung hilft, die Softwareentwicklung zu vereinfachen, insbesondere wenn es viele Programmierer gibt, die ein Softwareprojekt realisieren. Bei einem *Modul* handelt es sich um eine abgeschlossene Funktionseinheit, die einen Dienst an das System liefert und bei Bedarf ausgetauscht werden kann. Außerdem soll es mit seiner Umwelt nur über klar definierte Schnittstellen kommunizieren. Ein Programmierer muss bei Verwendung eines Moduls keine Kenntnisse über die innere Arbeitsweise haben, die Integrität eines Moduls kann ohne Kenntnisse seiner Einbettung in das Gesamtsystem überprüft werden.

Programmiersprachen unterstützen die Modularisierung bis zu einem gewissen Grad durch Mechanismen der Komposition und Dekomposition. Mit *Methoden* lassen sich wiederkehrende Aufgaben kapseln. Objektorientierte Sprachen kennen zusätzlich das Konzept der *Klasse*, um Methoden mit ihren Daten als eine Einheit zusammenzufassen. Auch auf binärer Ebene erfolgt eine Modularisierung. *Komponenten* bieten den Vorteil, dass sie als binäre Auslieferungseinheit eine Wiederverwendung ermöglichen, ohne dass der ursprüngliche Quelltext zur Verfügung stehen muss [Szyperski 2002, S. 357 ff.].

Diese Formalismen sind allerdings nicht ausreichend und unterstützen jeweils nur die Dekomposition in eine dominante Zerlegung [Tarr u. a. 1999]. Im Ergebnis ist der Quelltext der Module durchsetzt mit Dingen, die das Modul neben seiner eigentlichen Bestimmung implementieren muss. Andererseits gibt es Fälle, wo die Implementationen einer Anforderung über mehrere Module verstreut ist, da sie mit objektorientierten Techniken nicht modularisiert werden können. Es ist inzwischen allgemein anerkannt, dass diese Probleme zu einer höheren Zahl von Programmfehlern (Bugs) führen [Eaddy u. a. 2008; Greenwood u. a. 2007; Tsang u. a. 2004].

Die *Aspektorientierte Programmierung* (AOP) wurde entwickelt, um das Problem der dominanten Zerlegung zu lösen. Ein Quelltext, der sich sonst unübersichtlich über viele Stellen verteilt, wird so für den Programmierer leicht verständlich zusammengefasst [Kiczales u. a. 1997]. Dinge, die in der dominanten Zerlegung Module überschneiden würden, können in eigenen Modulen - den *Aspekten* - implementiert werden. Ein *Aspektweber* webt die Aspekte an den „Stellen der Überschneidung“ später ein.

AOP verspricht, dass die Produktivität der Programmierer steigt, Systeme wartbarer und leichter weiterzuentwickeln sind, Programmierfehler abnehmen und ein besser lesbaren Quelltext entsteht [Laddad 2002]. Allerdings gibt es inzwischen einige kritische Stimmen, die genau das in Frage stellen [Constantinides u. a. 2004; Gabriel u. a. 2006; Steimann 2006; Tourwé u. a. 2003]. Sie kommen zu dem Ergebnis, dass durch AOP der Quelltext unübersichtlich

und schlecht lesbar wird. Besonders kontrovers wird die Diskussion um die Frage geführt, ob AspectJ bzw. Aspektorientierte Programmierung im Allgemeinen die Modularität zerstört, statt sie zu verbessern [Gabriel u. a. 2006; Steimann 2006].

Nachfolgend soll diskutiert werden, welche Probleme mit den bekannten Ansätzen entstehen. Da der größte Teil der bekannten AOP-Lösungen [AOSD Wiki 2007] sich mehr oder weniger an den Konzepten von AspectJ orientiert und andererseits AspectJ die bekannteste AOP-Lösung ist, wird die Diskussion den Fokus auf AspectJ legen. Ziel soll dabei sein, die suboptimalen Konzepte zu identifizieren und aus diesen Erkenntnissen alternative Wege aufzuzeigen. Dazu soll diese Arbeit einen wichtigen Beitrag leisten.

1.1 Begriffe

Mit der Aspektorientierten Programmierung werden neue Konzepte eingeführt, die hier kurz vorgestellt werden sollen. Oft ist im Zusammenhang mit AOP von *crosscutting concerns* die Rede. Eine einigermaßen treffende Übersetzung für diesen Begriff ist *überschneidende Belange*.

Ein *Belang* ist nach [van den Berg u. a. 2005] etwas, das für die Entwicklung eines Systems, seines Betriebs oder anderer Angelegenheiten von Interesse ist und kritisch oder anderweitig wichtig für einen oder mehrere Akteure. Überschneidende Belange entstehen, wenn das Modulkonzept einer Programmiersprache es nicht ermöglicht, jeden Belang in eigenen Modulen zu beschreiben. Einige Belange sind dann über mehrere Module verteilt, überschneiden sie also.

Ein *Aspekt* ist eine eigenständige modulare Einheit, die einen Belang beschreibt, der auf mindestens ein anderes Modul angewendet werden soll. Ein Aspekt existiert immer in Koexistenz mit mindestens einem weiteren Modul. Der Formalismus ist durch den Aspektweber vorgegeben und stellt beispielsweise eine Erweiterung einer Programmiersprache dar. Im Falle von AspectJ handelt es sich um eine Weiterentwicklung der Sprache Java [Kiczales u. a. 2001]. Bei anderen Lösungen liegt diesem Formalismus ein spezielles Programmiermodell zugrunde, wobei Konzepte einer regulären Programmiersprache verwendet werden, um Aspekte zu beschreiben. Diejenigen Module, auf die die Aspekte angewendet werden sollen, werden auch als *Basisprogramm* bezeichnet.

Zu einem Aspekt gehören weiterhin *Advices* und *Introduktionen* (auch *Inter-Type-Deklarationen*). Der Begriff Advices geht zurück auf [Teitelman 1966], der in seinem *Pilot*-Programmiersystem für Lisp damit Methoden meint, die im Eintritts- oder Austrittspunkt existierender Methoden eingefügt werden können. Advices können die Ausführung der Methode beeinflussen, indem sie Parameter ändern, den Rückgabewert manipulieren oder die Abwicklung gänzlich verhindern. Sie können aber auch einfach nur zusätzliche Berechnungen ausführen.

In der Aspektorientierten Programmierung ist der Begriff etwas weiter gefasst. Hier beschreibt er einen Softwareartefakt, der an bestimmten Punkten der Ausführung eines Programmes, den *Joinpoints*, eingewoben werden sollen. Ein Joinpoint hat nicht notwendigerweise eine Entsprechung im zu verwebenden Zielquelltext. Joinpoints existieren vielmehr erst während der Laufzeit des Programmes und repräsentieren zu einem Teil den Programmzustand. Wie im folgenden Abschnitt erläutert wird, ist jedoch gerade das als ursächlich für kontroverse Diskussionen anzusehen.

Joinpoints können in *statische* und *dynamische* Joinpoints unterteilt werden. Statische Joinpoints entsprechen Punkten im Quelltext eines Programmes - unabhängig vom Programmzustand. Sie werden in der Literatur oft auch als *Joinpoint-Shadows* bezeichnet. Bei den dynamischen Joinpoints hingegen ist der Programmzustand immanent. Die dynamischen Joinpoints sind eine Obermenge der statischen Joinpoints.

Ein oder mehrere Joinpoints können mit einem *Pointcut* selektiert werden. Ein Pointcut bezeichnet diejenigen Joinpoints, an denen ein Advice aktiv werden soll. Ein Pointcut ist

```
1  public class Baz {
2      public void bar()
3      {
4          foo();
5      }
6
7      private void foo()
8      {
9        ...
10     }
11 }

   ⋮

12 privileged aspect A  {
13     pointcut fooPC(): call(void Baz.foo());
14
15     before(): fooPC()
16     {
17       ...
18     }
19 }
```

Listing 1.1: Verweben privater Methoden in AspectJ

entweder Teil des Advices oder der Advice referenziert einen entsprechenden Pointcut.

Als Introduktionen werden Methoden des Aspektes bezeichnet, die in anderen Klassen eingeführt werden sollen. Einführen meint dabei, dass die Schnittstelle dieser Klassen um die Deklaration der Introduktion erweitert wird. In einigen AOP-Systemen, wie AspectJ, ist es auch erlaubt, neue Attribute einzuführen und die Vererbungshierarchie anzupassen. Das wird oft auch als *Inter-Type*-Deklaration bezeichnet.

1.2 Aspektorientierung und Modularität

Wie bereits erwähnt, ist der Hauptkritikpunkt an AOP, dass durch deren Verwendung die Modularität der Programme zerstört wird. Eine Implementierung, die mit objektorientierten Programmierungstechniken gekapselt wurde, wird durch die aspektorientierten Techniken wieder offengelegt. So erlaubt es AspectJ, Aspektcode in private Methoden einzuweben. Zur Verdeutlichung soll hier - ohne dabei zu sehr im Detail auf AspectJ eingehen zu wollen - der Beispielquelltext aus Listing 1.1 herangezogen werden. Eine umfassende Beschreibung der Sprache kann in [Laddad 2003] nachgelesen werden.

In den Zeilen 1 bis 11 wird die Klasse `Baz` mit den Methoden `foo` und `bar` definiert. Die Methode `foo` in Zeile 7 wurde ferner als *private* deklariert und ist somit nicht Teil der öffentlichen Schnittstelle. In einem anderen Teil des Quelltextes, hier die Zeilen 12 bis 19, wird zusätzlich ein Aspekt `A` definiert. Dieser greift mit seinem Pointcut (Zeile 13) auf die privaten Implementationsdetails der Klasse `Baz` zu und fügt dort den Quelltext aus dem Advice der Zeile 15 bis 18 ein.

Offensichtlich ist für die Programmierung von `aspect A` Wissen über die interne Struktur von `Baz` notwendig. Genau das widerspricht aber dem von Parnas geforderten Prinzips des „*information hiding*“ [Parnas 1972b, S. 1056]:

> „Every module ... is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.“

Informationen über die Implementation eines Moduls sollten nicht an die Programmierer eines anderen Moduls weitergegeben werden. Einzig das Wissen über eine wohl definierte

Schnittstelle und deren Verhalten sollte zwischen den Beteiligten geteilt werden. Das wird oft als „*black box*“- Eigenschaft eines Moduls bezeichnet. Parnas hatte erkannt, dass eine Verteilung von zu vielen Informationen unter den Programmierern einen negativen Einfluss auf die Softwareentwicklung hat. Es führt zu ungewünschten Abhängigkeiten und verhindert unabhängiges Entwickeln, da Programmierer sonst eigene Module mit dem Wissen über Implementationsdetails fremder Module entwickeln. Diese Implementationsdetails sind im Allgemeinen aber sehr fragil und werden sich bei der Evolution des Moduls natürlich ändern.

Gegen eine solche restriktive Forderung wird aus der AOP-Gemeinschaft oft argumentiert, dass überschneidende Belange natürlich auch die definierten Modulgrenzen (die Grenze zwischen öffentlicher Schnittstelle und privater Implementation) durchschneiden. [Kiczales und Mezini 2005; Kiczales und Voelter 2006] argumentieren, dass aus der Komposition und Implementation von Modulen zu einem Gesamtsystem neue sogenannte *überschneidende Schnittstellen* entstehen, beispielsweise dadurch, dass ein Modul in seiner Implementation ein anderes Modul referenziert und benutzt. Diese Schnittstellen sind nicht explizit vom Programmierer im Quelltext notiert, sondern entstehen bei der Komposition und entsprechen letztendlich den bekannten Joinpoints. Damit - so die Argumentation - wird das Modulkonzept wieder hergestellt.

Das führt jedoch das Modulkonzept ad absurdum, da für ein Modul als Entität nicht entschieden werden kann, was seine öffentliche Schnittstelle ist. Aspekte können demnach erst *nach* der Komposition eines Systems in dieses eingefügt werden, da erst zu diesem Zeitpunkt feststeht, gegen welche überschneidenden Schnittstellen sie programmiert werden können. Problematisch ist auch, wenn Aspekte zu einer Komposition hinzugefügt werden, da sich dadurch die überschneidenden Schnittstellen ändern. Es ist somit unmöglich, Schnittstellen stabil zu halten, was aber eine unabdingbare Forderung für die Wiederverwendbarkeit und unabhängige Entwicklung von Modulen ist.

Die Annahme, Aspekte erst nach der Komposition des Systems hinzuzufügen, widerspricht auch der üblichen Vorgehensweise bei der Entwicklung. Ein Systemarchitekt legt die Schnittstellen zwischen den Modulen *vor* der Programmierung in der Entwurfsphase fest und macht den Programmierer Vorgaben für die Implementation. Die überschneidenden Schnittstellen nach [Kiczales und Mezini 2005; Kiczales und Voelter 2006] sind zu diesem Zeitpunkt aber alles andere als konkret und verhindern damit eine vollständige Spezifikation des zu entwickelnden Systems zu diesem Zeitpunkt.

1.3 Die Evolution von aspektorientierten Programmen

Ein weiteres, oft beobachtetes Paradoxon bei der Verwendung aspektorienterter Techniken ist, dass die Wartungsfähigkeit des Programmes wegen der Modularisierung von sonst überschneidenden Belangen eingeschränkt wird. [Tourwé u. a. 2003] sieht die starke Kopplung der Pointcuts an die Struktur des Basisprogramms als Ursache für dieses Phänomen. Änderungen im Basisprogramm haben somit Auswirkungen auf die Pointcuts und ziehen Änderungen im Aspektcode nach sich. Das solche Änderungen - auch an der Struktur - während der Evolution eines Programms häufiger auftreten, ist allgemein anerkannt [Fowler 1999; Opdyke 1992] und manifestiert sich insbesondere in der Existenz von umfangreichen Refactoring-Werkzeugen, die solche Änderungen sehr einfach ermöglichen. Sie sind in aktuellen Enwicklungsumgebungen, wie beispielsweise Eclipse [The Eclipse Foundation 2008b] und Visual-Studio [Microsoft Corporation 2008a], integriert und für den Programmierer problemlos zu benutzen.

Am eingangs aufgeführten Beispiel (Listing 1.1) lässt sich leicht nachvollziehen, dass eine Änderung des Methodennamens `foo` im Basisprogramm den im Aspekt Zeile 13 definierten Pointcut wirkungslos machen würde. Der Programmierer kann allein durch das Lesen des Quelltextes von `Baz` nicht die Auswirkungen seiner Änderung erfassen. Das ist der von [Filman und Friedman 2000, 2005] geforderten Eigenschaft geschuldet, dass Aspekte gegenüber

einem Basisprogramm unsichtbar sein müssen. In dem Artikel wird diese Forderung damit begründet, dass die Programmierer weiter - wie gewohnt - entwickeln können und trotzdem an Vorteilen von AOP partizipieren. Filman und Friedman gehen dabei jedoch von der Annahme aus, dass die Aspekte stets nach dem Basisprogramm entwickelt werden. Bei Software, die stetig weiterentwickelt wird, ist das nicht möglich; bei Projekten mit vielen Programmierern nicht praktikabel.

Üblicherweise wirkt ein Advice auf mehr als einen Punkt im Basisprogramm. Diese *1:n* -Beziehung wird in den meisten Pointcut-Sprachen durch explizites Aufzählen der entsprechenden Bezeichner oder durch die Verwendung von *Platzhaltern* erreicht. Hier wird nur ein Teil des kompletten Bezeichners zur Identifikation der Joinpoints angegeben und der Rest durch einen solchen Platzhalter ersetzt. Beispielsweise würde `set*` für alle Joinpoints stehen, deren Bezeichner mit „set“ beginnen und in Java per Konvention zum Setzen von Eigenschaftswerten verwendet werden. Das birgt die Gefahr in sich, dass auch Joinpoints identifiziert werden, die nicht in der Intention des Programmierers stehen. So würde z. B. auch die Methode `setup` unter diese Beschreibung fallen, die keinen Eigenschaftswert repräsentiert. Wollte der Programmierer ausschließlich das Setzen von Eigenschaftswerten kontrollieren, wäre diese offensichtlich einfache und logische Joinpoint-Beschreibung dennoch falsch.

Platzhalter führen dazu, dass der Aspektentwickler den Entwicklern des Basisprogramms strenge Namenskonventionen auferlegen muss. Konventionen haben jedoch den Nachteil, dass sie verletzt werden können, da deren Einhaltung schwer zu überprüfen ist. Die persönliche Erfahrung des Autors als Systemarchitekt zeigt, dass man in diesem Punkt nicht mit der Einsicht der Programmierer rechnen darf. Verletzt ein Programmierer eine Konvention, ist das Programm syntaktisch weiterhin korrekt. Das kann jedoch dazu führen, dass eine Verwebung an unvorhergesehener Stelle stattfindet bzw. eine Verwebung nicht ausgeführt wird. Werkzeuge zur automatischen Prüfung von Konventionen - zum Beispiel [FxCOP 2006] - helfen hier nicht, da sie die Bedeutung eines Bezeichners und die Intention des Programmierers nicht erkennen können. Ein Werkzeug kann die Semantik der Methode `setup` sowohl nicht erkennen als auch nicht entscheiden, ob diese Methode zum Setzen einer Eigenschaft mit dem Namen „up“ verwendet wird und somit nach Konvention korrekterweise mit „set“ beginnt, oder ob sie eine normale Methode ist.

Weiterhin werden Konventionen aber auch durch allgemeine Richtlinien der Programmierung - z. B. durch den Auftraggeber - vorgegeben. Oder es werden Frameworks verwendet, die solche Vorgaben ausführen. Es ist daher nicht auszuschließen, dass es zu Konflikten mit Konventionen, resultierend aus der Joinpoint-Beschreibung eines Aspektes und anderen Konventionen, kommt.

1.4 Sichtbarkeit von Aspekten

Im Absatz 1.3 wurde bereits die von [Filman und Friedman 2000] erhobene Forderung der so genannten *Nichtsichtbarkeit* (*Obliviousness*) von Aspekten erwähnt. Nach [Sullivan u. a. 2005] ist die Nichtsichtbarkeit von Aspekten eng an die Quantisierung der Joinpoints gekoppelt. Eine ausdrucksstärkere Joinpoint-Beschreibungssprache geht mit einem höheren Level der Nichtsichtbarkeit einher. Das heißt, je detaillierter der Aspektprogrammierer die Joinpoints beschreiben kann, je genauer er seine Advices in das Basisprogramm einweben kann, um so weniger Zuarbeit ist vom Programmierer des Basisprogramms notwendig.

Mit einer mächtigen Joinpoint-Beschreibungssprache existieren praktisch keine expliziten Schnittstellen zwischen Aspekt und Basisprogramm. Das hat zur Folge, dass der Aspekt die für seine Aufgaben benötigten Informationen direkt aus den Interna des Basisprogramms ableiten muss. So etwas kann nur gelingen, wenn der Programmierer des Aspekts eben diese Interna auch kennt. Die Folge ist eine starke Abhängigkeit des Aspektprogrammierers vom Basisprogrammierer, ohne dass es dem Programmierer des Basisprogramms bewusst ist. Aber

auch für diesen bleibt das Eingreifen in seine „Betriebsinterna“ nicht ohne Folgen.

In dem Beitrag *„Go to statement considered harmful“* beschäftigt sich [Dijkstra 1968] mit der Fragestellung, ob die `Go-To`-Anweisung in Hochsprachen noch verwendet werden sollte. Bei seiner Argumentation verneint er diese Frage und begründet es damit, dass die Fähigkeiten eines Programmierers, aus dem statischen Quelltextes eines Programmes die dynamischen Abläufe zur Laufzeit abzuleiten, beschränkt sind. Der Programmierer benötigt ein Koordinatensystem, auf dem er die zeitlichen Abläufe abbilden kann. Das erstellt er durch einen textuellen Index auf den einzelnen Anweisungen in seinem Quelltext. Das Fortschreiten des Prozesses lässt sich einfach durch sukzessives Erhöhen dieses Indexes abstrahieren (was letztendlich dem zeilenweisen „Durchlesen“ des Quelltextes entspricht). Zum anderen gibt es in Schleifenkonstrukten und bedingten Anweisungen dynamische Indizes (durch die Zählvariablen und Bedingungen) an denen sich das Voranschreiten des Prozesses in diesem Kontrollabschnitt erklären lässt. Für verschachtelte Methodenaufrufe reicht ein textueller Index nicht aus. Vielmehr werden diese durch eine sequenzielle Aneinanderreihung solcher Indizes (mit Referenz zur entsprechenden Methode) gesehen, wobei deren Anzahl der Tiefe des Aufrufstapels entspricht.

Für eine bestimmte Belegung der Indizes lässt sich relativ leicht ermitteln, wie weit der Prozess vorangeschritten ist, insbesondere welche Anweisungen ausgeführt wurden, um an diesen Punkt zu gelangen. Für ein `Go To` ist es laut Dijkstra allerdings aufgrund des unstrukturierten Kontrollflusses unmöglich, ein Koordinatensystem zu finden, das den Fortschritt im Prozess hinreichend beschreibt.

Vergleicht man die Advices von Aspekten mit einem `Go To`, findet man einige Parallelen. So argumentieren [Constantinides u. a. 2004], dass Advices wie einfache Methodenaufrufe erscheinen. Es gibt aber keinen textuellen Index für einen Advice, da der Aspekt transparent ist. Das sukzessive Durchlaufen des textuellen Indexes ermöglicht es dem Programmierer nicht mehr, auf den Ablauf des Prozesses zu schließen. Das wird umso deutlicher, wenn Advices Variablen des Basisprogramms ändern, die dort aufgrund von bedingten Anweisungen Einfluss auf den Kontrollfluss haben. Sie verändern damit dynamische Indizes, ohne dass der Nutzer etwas davon bemerkt. Das Ergebnis ist ein Koordinatensystem, das nur scheinbar korrekt ist.

Dagegen argumentieren [Kutner u. a. 2008], dass das Koordinatensystem für den Programmierer trotzdem erhalten bleibt, da er aufgrund der von [Filman und Friedman 2000] proklamierten Nichtsichtbarkeit der Aspekte keine Kenntnis von eventuell eingewobenem Quelltext haben muss. Ihrer Meinung nach lässt sich für die Aspekte jeweils ein weiteres wohlgeformtes Koordinatensystem finden. Diese Sicht ist insofern problematisch, da das Ausführen zusätzlicher Anweisungen aus den Advices natürlich Einfluss auf das Koordinatensystem einer Methode hat und nicht unabhängig in einem eigenen Koordinatensystem gesehen werden kann. Der Einfluss kann sowohl passiv geschehen, da allein ihre Ausführung das Laufzeitverhalten einer Methode beeinflusst, aber auch aktiv, indem Advices Attribute des Basisprogramms ändern.

Mit angepassten Entwicklungsumgebungen wie [The Eclipse Foundation 2008a] wird versucht, das Koordinatensystem für den Programmierer durch eine grafische Lösung wiederherzustellen. Diejenigen Punkte im Quelltext, an denen ein Advice wirksam ist, werden durch eine grafische Markierung des Quelltextes hervorgehoben. Das lindert die Wirkung, behebt aber nicht die Ursache. Insbesondere funktioniert das nur für Advices, die an *statischen* Joinpoints aktiv werden, d. h. bei denen die Aktivierung unabhängig von Programmzustand erfolgt.

Dem kann man natürlich entgegenhalten, dass das objektorientierte Konzept der späten Bindung von (virtuellen) Methoden ähnlichen Charakter hat und auch hier erst zur der Laufzeit feststeht, welche Implementation tatsächlich abgewickelt wird. Demgegenüber steht jedoch im Wesentlichen, dass der Aufruf einer virtuellen Methode stets explizit erfolgt. Auch wenn die konkrete unterliegende Implementation zum Zeitpunkt der Programmierung nicht

bekannt ist, so hat der Programmierer zumindest eine Vorstellung davon, was die Methode bewirken soll und worin die Auswirkungen seines Aufrufes bestehen. Demgegenüber weiß er bei einem Advice nach [Filman und Friedman 2000] nicht, dass dieser überhaupt auf seinen Quelltext wirkt.

1.5 Statische und dynamische Joinpoints

Joinpoints, an denen die Advices wirken, sind Punkte in der Ausführung eines Programms. Nach dieser Definition sind die Advices an den Pointcuts der Dynamik des Programms unterworfen. In AspectJ manifestiert sich das vor allem in Pointcuts, welche Joinpoints selektieren, die in einem bestimmten Kontrollfluss liegen (hierzu existieren die Schlüsselworter `cflow` und `cflowbelow`) sowie in bedingten Pointcuts, die Joinpoints auswählen, bei denen die dynamischen Werte des Programms eine vorgegebene Bedingung erfüllen (das Schlüsselwort `if`).

Nimmt man sich den *Trace* eines Programms, so entspricht jede einzelne Anweisung in diesem Trace einem dynamischen Joinpoint. Mit Trace ist hier die Abfolge von Anweisungen zur Abwicklung eines Programms gemeint. Die Menge aller möglichen Traces für ein Programm bestimmt damit die Menge der dynamischen Joinpoints für das Programm. Die Mächtigkeit der gewählten Pointcut-Sprache bestimmt, wie groß die Menge derjenigen Joinpoints ist, die in dieser Menge selektiert werden können.

Leider lässt sich die Menge der durch einen Pointcut selektierten Joinpoints nicht bestimmen. Man kann lediglich Aussagen darüber treffen, ob ein gegebener Joinpoint durch einen Pointcut selektiert wird oder nicht. Daher kann stets erst zur Laufzeit entschieden werden, ob Advices, die dynamische Joinpoints selektieren, ausgeführt werden oder nicht. Diese Prüfungen gehen mit hohen Laufzeitkosten einher und können nur mit Anpassungen an die Ausführungsumgebung effizient gelöst werden [Haupt 2005].

Es ist die Frage zu klären, ob es überhaupt notwendig ist, auch den Programmzustand in den Joinpoints selektieren zu müssen. Wiegt der Vorteil durch die Betrachtung dynamischer Joinpoints den Nachteil der hohen Komplexität auf? Der Autor ist der Meinung, dass dies nicht der Fall ist. Wann immer es notwendig erscheint, Advices in Abhängigkeit vom Programmzustand auszuführen, kann das auch mit klassischen Kontrollstrukturen, wie `if-then`, erreicht werden. Aber auch die im Kapitel 5 vorgestellte „Kontextorientierte Programmierung“ ist ein Beispiel dafür, wie mit einem expliziteren Konzept Programmteile in Abhängigkeit vom Programmzustand ausgeführt werden können.

1.6 Aspekte und Datenzugriff

In vielen Aspektorientierten Lösungen ist der Zugriff auf ein Attribut ein Joinpoint und kann durch entsprechende Konstrukte mit einem Advice versehen werden. Der Advice erhält die vollständige Kontrolle über das Datum. Dies bedeutet z. B., dass bei einem schreibenden Zugriff auf das Attribut der abzuspeichernde Inhalt und bei einem lesenden Zugriff das gelesene Ergebnis verändert werden kann.

Für den Programmierer des Basisprogramms ist es dabei völlig intransparent, ob ein Datenzugriff

1. tatsächlich auf der von ihm gedachten Speicherstelle ausgeführt wird,
2. in der von ihm angegebenen Form auf dieser Stelle abgespeichert wird und
3. zusätzliche Aktionen ausgeführt werden.

Gerade der letzte Punkt ist aus Sicht des Autors besonders kritisch zu betrachten. Der Programmierer kann nicht mehr entscheiden, welche Komplexität ein einfacher Datenzugriff

hat. Das Ergebnis eines Methodenaufrufes wird man üblicherweise in einer lokalen Variable speichern, insbesondere wen man dieses im folgenden Quelltextabschnitt mehrfach verwenden möchte. Als Entwickler ist man sich darüber im Klaren, dass es aufgrund des hohen Aufwandes nicht sinnvoll ist, die Methode mit denselben Parametern an den Stellen der Verwendung jeweils erneut aufzurufen.

Für ein Attribut ist eine Zwischenspeicherung in eine lokale Variable nicht sinnvoll, da der Zugriff ungefähr genauso teuer ist, wie der Zugriff auf die lokale Variable. Wird der Zugriff jedoch durch einen Advice kontrolliert, ist die Situation plötzlich identisch mit dem Aufruf einer Methode, denn jeder Zugriff impliziert die Abwicklung des Advices. Damit entstehen Aufwände, die der Entwickler des Basisprogrammes vorher nicht absehen konnte.

1.7 Schlussfolgerungen

Mit der vorangegangenen Diskussion sollte deutlich werden, dass eine komplexe Joinpoint-Beschreibungssprache nicht das Allheilmittel ist, um das Problem der überschneidenden Belange zu lösen. Das ermöglicht erst das Einweben von Quelltexten in eigentlich versteckte Implementationsdetails anderer Module, doch das führt wiederum zu neuen Problemen. Es entstehen starke Abhängigkeiten zwischen den Modulen, die sich nicht in einer formalen Schnittstelle wiederfinden lassen.

Dies ist jedoch eine Grundvoraussetzung für eine verteilte Softwareentwicklung. Erst eine eindeutige Schnittstellenbeschreibung ermöglicht die parralele Entwicklung von Modulen und auch Aspekten. Vor allem sollten Aspekte aber auch wiederverwendbar sein. Dies kann aber nicht funktionieren, wenn sie sich - wie dargestellt - stets auf eine konkrete Implementierung beziehen.

Auch ein Aspekt sollte sich bei der der Interaktion mit anderen Modulen ausschließlich auf öffentliche Schnittstellen der Module beziehen und nicht auf deren Implementierung. Das geht einher mit dem von [Gamma u. a. 1995, S. 18] geforderten Prinzip für gute Programmierung:

> „Program to an interface, not an implementation“

In der objektorientierten Programmierung lässt sich dieses Paradigma gut umsetzen. Dabei ist es dann völlig unerheblich, wie die Implementation der Schnittstellen aussieht. Es ist also notwendig, das vorhandene objektorientierte Schnittstellenkonzept um neue Konzepte zu erweitern, mit dem sich überschneidende Belange vermeiden lassen.

Wichtig ist es in diesem Zusammenhang, dass das Koordinatensystem des Programmierers durch das Vorhandensein Advices nicht zerstört werden darf. Beim Lesen des Quelltextes eines Moduls sollte klar sein, welche zusätzlichen Belange dort wirken (obwohl sie an anderer Stelle implementiert sind).

Das bedingt natürlich ein Abrücken von der durch [Filman und Friedman 2000] eingeführten Forderung nach der Nichtsichtbarkeit von Aspekten. In der vorangegangenen Diskussion wurde jedoch deutlich, dass dies kein Nachteil ist. Im Gegenteil, dadurch ergibt sich die Möglichkeit einer aktiven Kommunikation von Modulen, die die Basisfunktionalität implementieren und denjenigen, die als Aspekte ausgeführt sind.

Die Einbeziehung des Programmzustands für die Entscheidung, ob ein Advice an einer bestimmten Stelle eingewoben wird, hat aus Sicht des Autors keinen Vorteil. Auch hier wird es für den Programmierer schwer, ein passendes Koordinatensystem zu finden, da Module nicht mehr für separat betrachtet werden können. Stattdessen kann ein explizites Abfragen des Zustandes in Kontrollstrukturen innerhalb der Advices erfolgen., wie dies unter anderem die im Kapitel 5 vorgestellte „Kontextorientierte Programmierung“ zeigt.

Schließlich sollte auch die strikte Trennung von Datenzugriff und Methodenaufruf erhalten bleiben. Das Ausführen von Advices beim Datenzugriff kann zu unerwarteten Effekten führen und somit auch zu neuen Fehlerquellen.

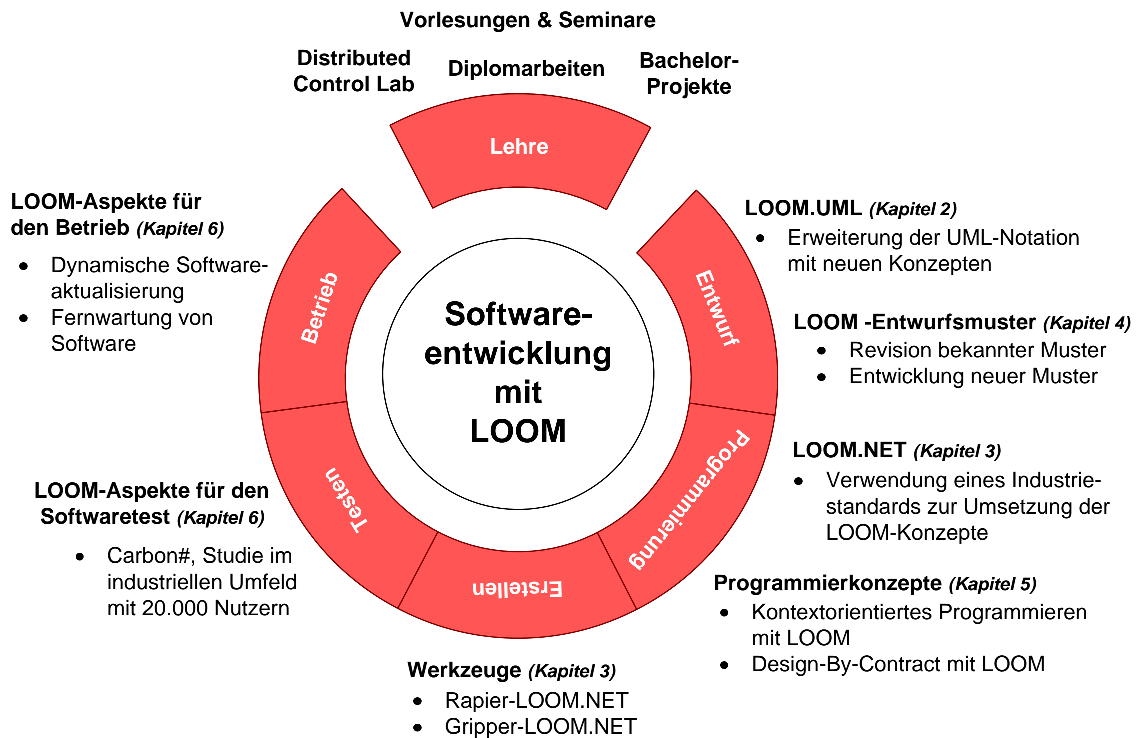


Abbildung 1.1: Einordnung dieser Arbeit im Umfeld der Softwareentwicklung

Um alle diese Punkte zu berücksichtigen Bedarf es verschiedener neuer Konzepte. Grundlage dieser Konzepte ist das vom Autor entwickelte LOOM-Programmierpardigma, welches in Kapitel 2 eingeführt wird. Aufbauend darauf wird gezeigt, wie dieses neue Paradigma Vorteilhaft auf den gesamten Softwareentwicklungszyklus auswirkt. Das Kapitel 3 beschreibt zwei Werkzeuge, einen dynamischen (Rapier-Loom.Net) und einen statischen Weber (Gripper-Loom.Net), die dieses Konzept unterstützen. Wie sich typische Entwurfsmuster mit der vorgestellten Lösung realisieren lassen, wird in Kapitel 4 ausgeführt. Kapitel 5 bespricht weitere Programmierkonzepte, die sich mit LOOM realisieren lassen. Des Weiteren werden in Kapitel 6 konkrete Projekte besprochen, die mit den LOOM.Net-Aspektwebern umgesetzt wurden. Schließlich beschäftigt sich das Kapitel 7 mit der Einordnung dieser Arbeit in den aktuellen Stand der Forschung.

1.8 Der wissenschaftliche Beitrag dieser Arbeit

Die vorliegende Arbeit hatte zum Ziel, ein neues Paradigma zur Entwicklung komponentenbasierender Systeme zu entwickeln, mit denen sich das Problem der überschneidenden Belange vermeiden lässt. Die entstandenen Konzepte sollten dabei insbesondere auf ihre Anwendbarkeit im gesamten Softwareentwicklungsprozess evaluiert werden. Eine These dieser Arbeit ist es, dass diese unter dem Namen LOOM entwickelten Konzepte eine nachhaltige Verbesserung in allen Phasen der Entwicklung erreichen können. Eine Verbesserung vor allem in Hinblick auf Wartbarkeit der Software, Quallität und Effizienz in der Entwicklung. Ein Überblick über die gesamte Arbeit ist in Abbildung 1.1 dargestellt.

Aufbauend auf den Ideen der aspektorientierten Programmierung nach [Kiczales u. a. 1997] werden hier einige neue Programmierkonzepte eingeführt, die nicht zu den in den vorangegangenen Abschnitten beschriebenen Problemen führen. Im Detail sind das die Annotationsrelationen und das Konzept der Überdeckung, Joinpoint-Initialisierer, Joinpoint-Variablen und die Möglichkeit, Nachrichten an den Joinpoints verändern zu können. Alle diese Details

werden in dieser Arbeit ausführlich besprochen.

Angefangen beim Entwurf einer Software wurde im Rahmen dieser Arbeit mit LOOM.UML ferner eine Erweiterung der UML-Modellierungssprache entwickelt, die es erlaubt, das LOOM-Konzept bereits in den frühen Phasen der Softwareentwicklung zu verwenden. Flankiert wird das durch eine Vielzahl von LOOM-Entwurfsmustern, mit denen typische Entwurfsprobleme nach einer vom Autor entwickelten Metrik im Vergleich zu objektorientierten Herangehensweisen effizienter und effektiver gelöst werden können.

Um dieses Konzept in der konkreten Programmierung auch benutzen zu können, wurde auf der Basis eines bestehenden Industriestandards für objektorientierte Sprachen, der *Common Language Infrastructure* (CLI) [Microsoft Corporation u. a. 2006], die Spracherweiterung LOOM.NET entwickelt. Diese ermöglicht eine Aspektbeschreibung - als neues Modularisierungskonzept - unabhängig von der Programmiersprache mit einem standardisierten Binärformat.

Diese Spracherweiterung allein reicht jedoch nicht aus, damit die Konzepte in der entwickelten Software auch wirksam werden. Hier ist vielmehr eine umfangreiche Werkzeugunterstützung vonnöten. Dafür wurden verschiedene Aspektweber entwickelt: RAPIER-LOOM.NET als dynamischer Aspektweber und GRIPPER-LOOM.NET als statischer Aspektweber basieren auf dem Microsoft Phoenix Kompilerframework. Aspektweber des LOOM.NET-Projektes waren auch die ersten öffentlich verfügbaren Aspektweber für die CLI und die Microsoft .NET-Plattform.

Auf der Basis der LOOM.NET Spracherweiterung lassen sich auch weitere Programmierkonzepte auf die CLI aufsetzten, ohne einen eigenen Kompiler entwickeln zu müssen. Konkret wurde das am Beispiel der Kontextorientierten Programmierung und des „Design by Contract“ demonstriert. Das Besondere an beiden Lösungen ist, dass sie mit wenigen LOOM.NET-Aspekten implementiert werden konnten und für alle CLI-konformen Sprachen zur Verfügung stehen.

Im Bereich des Softwaretests leistete eine vom Autor bei der Deutschen Post AG durchgeführte Studie einen wertvollen Beitrag. Das hier evaluierte Carbon#-System ist mit 22.000 Nutzern ein recht großes System mit einem hohen Anspruch an die Softwarequalität. Hier wurde experimentell untersucht, wie mit einfachen LOOM.NET-Aspekten Entwicklungs- und Testaufwände reduziert werden können.

Schließlich wurden auch Lösungen erarbeitet, die eine Verbesserung im produktiven Einsatz von Software bedeuten. Zu nennen sei hier das Projekt *DSUP*, das sich mit dem Problem der dynamischen Softwareaktualisierung beschäftigte. Komponenten der Software können hier im laufenden Betrieb ausgetauscht werden, ohne dass ein Wartungsfenster mit aufwendigem Neustart notwendig wäre. Durch konsquente Verwendung der LOOM-Paradigmen und Konzepte konnte DSUP auf der Standardausführungsumgebung der CLI realisiert werden und erfordert nur unbedeutende Änderungen an der betreffenden Software. Aber auch die vom Autor co-betreute Diplomarbeit von [Olejniczak 2007] zur Fernwartung von Software leistet hier einen wesentlichen Beitrag. LOOM.NET-Aspekte werden eingesetzt, um Softwareeigenschaften mit Fernwartungswerkzeugen beeinflussen zu können.

Teile der Arbeit wurden auf zahlreichen wissenschaftlichen Konferenzen und in Artikeln veröffentlicht. Darunter sind [Schult und Polze 2002a] [Schult und Polze 2002b] [Schult und Polze 2003] [Schult, Tröger und Polze 2003] [Polze und Schult 2004] [Köhne, Schult und Polze 2005] [Rasche, Schult und Polze 2005] [Jeske, Brehmer, Menge, Hüttenrauch, Adam, Schüler, Schult, Rasche und Polze 2006] [Rasche und Schult 2007] [Eaddy, Cyment, van de Laar, Schmied und Schult 2007b]. Weitere Veröffentlichungen fanden in Form von Fachvorträgen zu diesem Thema vor Industriepublikum statt [Schult 2005] [Schult 2007].

Die vom Autor entwickelten Technologien fanden weiterhin in den Dissertationen und Diplomarbeiten von [Kostian 2005; Olejniczak 2007; Puzovic 2005; Rasche 2002, 2008; Troeger 2002] Anwendung. Ausgewählte Arbeiten werden im Kapitel 6 noch detaillierter vorgestellt. Außerdem co-betreute der Autor Bachelorprojekte in Zusammenarbeit mit der Deutschen

Post in den Jahren 2004, 2005 und 2006 mit jeweils vier Studenten sowie die Masterarbeit von [Schöbel 2005] und die Diplomarbeit von [Olejniczak 2007].

Ein weiteres Einsatzgebiet der Forschungsergebnisse des Autors ist das am Lehrstuhl für Betriebssysteme des Hasso-Plattner-Institutes beheimatete *Distributed Control Lab*, eine verteilte Laborinfrastruktur für Forschung und Lehre. Ebenfalls erfolgt eine etwas ausführlichere Darstellung im Kapitel 6.

Die Ergebnisse des LOOM.NET-Projektes wurden vom Autor ab dem Jahr 2002 öffentlich im Internet zur Verfügung gestellt und fanden große Resonanz im Forschungsumfeld und bei Softwareentwicklern. Das manifestiert sich zum einen in über 2000 offiziell registrierten LOOM.NET Benutzern als auch in Blogs und Veröffentlichungen zum Thema, wie zum Beispiel [Bitter 2005; Milovanovic 2004; Richter 2007; Sakuda 2004; Wagner 2003].

Schließlich sollte nicht unerwähnt bleiben, dass die LOOM.NET-Aspektweber auch im kommerziellen Umfeld Verwendung finden. Folgende Firmen sind diesbezüglich mit dem Autor in Kontakt getreten:

- Deutsche Post IT-Solutions, Deutschland,
- Runbox AS, Oslo, Norwegen,
- Walker Technology Consulting LLC, Philadelphia, USA,
- Nowisys IT-Service GmbH, Deutschland,
- evidanza GmbH, Deutschland,
- digital spirit GmbH, Deutschland,
- HAKOM EDV-Dienstleistungsges. m. b. H, Österreich und
- The Sage Group plc, England.

Unterstützung fand die Dissertation zum einen durch den aktuellen Arbeitgeber des Autors, die Deutsche Post IT Services Europe, sowie verschiedene geförderte Forschungsprojekte am Hasso-Plattner-Institut. Zu diesen Forschungsprojekten gehörten:

- Phoenix Direct Funding Award (2007) *(#15899)* [Xu 2007],
- Centracite Community Strategic Sponsor Agreement,
- Rotor I Award: Object and Process Migration in .NET (2003) *(#2003-258)*
- DISCOURSE: The Berlin Distributed Computing Lab (2003/04) und *(#2003-249)*
- Tutorial on Component Programming and Aspect-Oriented Programming with .NET on First .NET-Crash Course at Microsoft Research, Cambridge (2001) *(#2001-61).*

1.9 Hinweise für den Leser

Viele Fachbegriffe in der Informatik haben ihren Ursprung in der englischen Sprache. Oft ist daher das englische Original im deutschen Sprachgebrauch eher anzutreffen als eine entsprechende Übersetzung. Bei einigen Begriffen ist es gar unmöglich, eine passende Entsprechung mit exakt derselben Bedeutung zu finden. Bei denjenigen Begriffen, für die es eine etablierte und verständliche Übersetzung gibt, wird natürlich der deutsche Begriff verwendet. Begriffe, für die es zwar eine deutsche Übersetzung gibt, deren Verwendung sich aber im allgemeinen Sprachgebrauch nicht gefestigt hat, werden bei erstmaliger Verwendung in dieser Arbeit zusammen mit ihrem englischen Originalbegriff eingeführt. Schließlich bleiben Begriffe erhalten,

für die sich keine hinreichend gute deutsche Übersetzung finden lässt. Diese werden, nachdem sie eingeführt wurden, in deutscher Schreibweise weiterverwendet. Als Beispiel sei hier der englische Begriff *join-point* genannt, der in dieser Arbeit fortan als *Joinpoint* bezeichnet wird.

Zur besseren Orientierung werden verschiedene Formatierungen verwendet. *Kursive* Hervorhebungen deuten entweder auf einen neuen Begriff hin oder verweisen auf `Quelltextelemente`. **Schlüsselwörter** verschiedener Programmiersprachen werden hingegen fett dargestellt.

Häufig verwendete Abkürzungen sind *OOP* für objektorientierte Programmierung und *AOP* für aspektorientierte Programmierung.

2 Neue Konzepte zur Modularisierung von Software

Im ersten Kapitel wurde bereits dargelegt, dass eine gute Modularisierung die Grundvoraussetzung für eine verteilte Anwendungsentwicklung ist. Aber auch vor dem Gedanken der Wiederverwendung, Wartbarkeit und dem Verständnis der Software ist eine sinnvolle Aufteilung in Module unabdingbar. Allerdings kann bei der objektorientierten Softwareentwicklung nicht immer eine gute Modularisierung gefunden werden [Kiczales u. a. 1997; Tarr u. a. 1999]. Das soll in diesem Kapitel einmal detaillierter untersucht werden.

Aufbauend auf diesen Analysen wird ein neues Programmierparadigma für eine bessere Modularisierung entwickelt. Ziel soll es sein, einzelne Anforderungen an die Anwendung gut separiert in einzelne Module aufzuteilen. Streuung und Durchsetzung soll vermieden werden, ohne das Modulkonzept selbst zu zerstören. Das LOOM-Programmierparadigma stellt den Kernbeitrag dieser Dissertation dar.

Im Detail werden die neuen Konzepte im dritten Teil des Kapitels unter Verwendung der vom Autor entwickelten LOOM.UML-Modellierungssprache erklärt. LOOM.UML ist als Erweiterung der UML2-Modellierungssprache [Object Management Group 2007a,b] zugleich auch die Basis der Modelle in den folgenden Kapiteln.

2.1 Modularisierung in objektorientierten Sprachen

Bevor der Prozess der Modularisierung genau untersucht werden kann, ist Voraussetzung, den Begriff des Moduls in der objektorientierten Programmiersprache exakt zu definieren. Ausgehend von [Parnas 1972a] müssen für ein Modul folgende Bedingungen gelten:

1. Ein Modul ist eine geschlossene Funktionseinheit, die eine komplexe Dienstleistung an das System liefert und bei Bedarf ausgetauscht werden kann.

2. Ein Modul kommuniziert mit seiner Umwelt nur über klar definierte Schnittstellen.

3. Die Verwendung eines Moduls verlangt keine Kenntnisse über seine innere Arbeitsweise.

4. Die Korrektheit eines Moduls kann ohne Kenntnisse seiner Einbettung in das Gesamtsystem überprüft werden.

Bezogen auf die objektorientierte Programmierung kann der Begriff Modul für verschiedene ineinander gekapselte Strukturierungseinheiten der objektorientierten Programmierung verwendet werden. Die kleinste unabhängige Einheit stellt hierbei eine *referenziell transparente Methode* dar. Referenziell transparent bedeutet, dass das Ergebnis der Methode ausschließlich von ihren Parametern abhängt. Die Schnittstelle einer solchen Methode sind deren Parameter, inklusive deren Rückgabewert. Solche Methoden sind abgeschlossene Funktionseinheiten, die unabhängig verwendet und ausgetauscht werden können, ohne die innere Arbeitsweise zu kennen.

Ist eine Methode nicht referenziell transparent, so hängt das Ergebnis ihrer Ausführung üblicherweise von Daten ab, die wiederum von anderen Methoden beeinflusst wurden. In der

objektorientierten Programmierung werden solche Methoden in der nächstgrößeren Strukturierungseinheit, den *Klassen*, zusammengefasst. In Klassen werden also Daten (die *Attribute* der Klasse) und die Methoden, die auf den Daten operieren, definiert.

Die Schnittstellen einer Klasse sind deren Methoden und Attribute. Einige Programmiersprachen erlauben es, die Schnittstellen durch Festlegung der Sichtbarkeit von Methoden und Attributen einzuschränken. Methoden und Attribute, die als nicht sichtbar deklariert wurden, sind somit nicht Teil der externen - also durch ein anderes Modul verwendbare - Schnittstelle einer Klasse.

Soll eine Schnittstelle unabhängig von einer konkreten Klasse definiert werden, ist das durch die Verwendung eines *Interfaces* möglich. Interfaces können von Klassen implementiert werden; sie müssen dann die im Interface definierten Methoden implementieren. Mit diesem Konzept können sich verschiedene Klassen dieselbe Schnittstelle teilen, obwohl sie diese jeweils unterschiedlich implementieren. Derjenige, der das Interface verwendet, muss die Klassenschnittstelle nicht kennen. Ein Sortieralgorithmus, der beispielsweise gegen das Interface `IComparable` programmiert wurde, kann alle Exemplare von Klassen sortieren, die dieses Interface implementieren. Das Interface `IComparable` erfordert folglich die Implementation der Methode `CompareTo`, mit der Objekte in eine Ordnungsrelation gesetzt werden können.

Oft stehen aber auch ganze Gruppen von Klassen miteinander in Zusammenhang. Zu einem Experiment könnte beispielsweise eine Auswertung gehören, in der die Messwerte aufbereitet wurden. Als Operationen auf der Auswertungsklasse sind Methoden zur grafischen Darstellung denkbar. Thematisch zusammengehörende Klassen lassen sich wiederum in Komponenten kapseln. Diese stellen dabei nicht nur Strukturierungsmittel des Quelltextes dar, sondern werden auch als eigenständige binäre auslieferbare Einheit betrachtet [Szyperski 2002].

Die Schnittstelle der Komponente ergibt sich aus der Summe der Schnittstellen aller in ihr enthaltenen Klassen. Ein großer Vorteil einiger Komponentensysteme besteht darin, dass die Komponenten ihre Schnittstellen auch in ihrer binären Form preisgeben. Das ist z. B. bei Komponenten der CLI [Microsoft Corporation u. a. 2006] der Fall. Dabei enthält jede Komponente Metadaten, die ihre Schnittstellen beschreiben.

Die Frage ist nun, wie eine optimale Modularisierung zu erreichen ist. Die Entwicklung einer neuen Anwendung beginnt üblicherweise mit der Analyse der Anforderungen. Jede Anforderung enthält ein Problem, das es zu lösen gilt und das sich bis zu einem gewissen Grad in Teilprobleme zerlegen lässt. Dieser Prozess verläuft subjektiv und wird von jedem Programmierer unterschiedlich ausgeführt. Deshalb ist er nicht eindeutig, denn er kann nach verschiedenen Kriterien erfolgen, z.B. nach funktionalen Belangen oder aus Sicht der zu verarbeitenden Daten.

In der objektorientierten Programmierung wird oft eine Zerlegung nach zu verarbeitenden Daten bevorzugt, da Klassen sich gut als Abbildung von Dingen der realen Welt eignen. Als Beispiel sei hier die Klasse Experimente genannt, die Experimente in einer Laborumgebung abbilden soll. Die Attribute der Klasse beschreiben den Zustand eines konkreten Experiments, wie z. B. die aktuellen Messwerte. Die Methoden der Klasse wiederum erlauben es, beispielsweise das Experiment zu initialisieren oder die Messwerte auszulesen.

Um den Prozess der Zerlegung etwas anschaulicher darzustellen, soll der Entwurf des *Distributed Control Labs* (DCL) skizziert werden. Das DCL ist eine real existierende Anwendung, die von der Gruppe Betriebssysteme und Middleware am Hasso-Plattner-Institut entwickelt wurde. Wenn in Kapitel 6 Projekte mit LOOM.NET vorgestellt werden, wird auch auf das reale DCL noch einmal eingegangen. An dieser Stelle soll diese Anwendung ausschließlich als Beispiel dienen. Zu diesem Zweck wurden entsprechende Anforderungen und Annahmen sowie die hier skizzierte Architektur für das Verständnis vereinfacht.

Nr.	Anforderung
1.	Darstellung eines realen Experiments in der Anwendung
1.1	Darstellung als Webformular
1.2	Darstellung in einem Visual-Studio Plugin
2	Unterstützung verschiedener Experimenttypen
2.1	Unterstützung für „Hau den Lukas“
2.2	Unterstützung für „Robotersteuerung“
3	Verwaltung der realen Experimente aus der Anwendung
3.1	Steuerprogramm für ein Experiment übertragen und ausführen
3.2	Status eines Experiments abrufen
3.3	Ergebnisse eines Experiments abrufen
4.	Schnelle Antwortzeiten
5.	Unterstützung eines Rollenkonzepts

Abbildung 2.1: Anforderungen (Belange) für das DCL

Das DCL soll eine verteilte Laborinfrastruktur werden, mit der verschiedene Experimente gesteuert werden können. Die Steuerung erfolgt über spezielle Steuerprogramme, die von den Experimentcontrolern abgewickelt werden. In Abbildung 2.1 sind die fünf Anforderungen an das DCL dargestellt. Grundsätzlich sollen verschiedene Experimenttypen unterstützt werden (2.). Einem Bediener soll zum einen der Zustand eines real existierenden Experiments in der Anwendung angezeigt werden können (1.) und zum anderen soll dieses Experiment aber auch über die Anwendung gesteuert werden können (3.). Da die Nutzer der Anwendung als ungeduldig eingeschätzt werden, ist zusätzlich explizit gefordert, dass die Anwendung schnell auf Eingaben reagieren soll (4.). Schließlich muss ein Rollenkonzept sicherstellen, dass nur autorisierte Benutzer die Anwendung verwenden können (5).

Aus der ersten Anforderung ergeben sich verschiedene Teilanforderungen. So ist verlangt, dass verschiedene Plattformen für die Darstellung unterstützt werden sollen. Es soll auf der einen Seite eine Variante geben, mit der die Experimente in einem Webbrowser angezeigt werden können (1.1), um einen einfachen Zugriff auf das DCL zu haben. Zum anderen ist aber eine Integration in die Visual-Studio-Entwicklungsumgebung von Microsoft vorgesehen, da hier die Steuerprogramme für die Experimente komfortabel entwickelt werden können (1.2). Welche Anforderungen an die Verwaltung der Experimente gestellt werden, ist im dritten Anforderungspunkt weiter aufgeschlüsselt.

In Abbildung 2.2 ist die mögliche Überführung der Anforderungen auf eine Modulstruktur dargestellt; es sind aber durchaus auch andere Abbildungen denkbar. Die Anforderungen sind hier mit einem A gekennzeichnet und auf der linken Seite dargestellt. Die Module befinden sich auf der rechten Seite, wobei ein C eine Komponente, ein K eine Klasse und ein M eine Methode bezeichnet. Die gesamte Anwendung wurde hier in drei Komponenten mit insgesamt sechs Klassen zusammengefasst. Methoden wurden nur dargestellt, wenn sie - den oben genannten Kriterien entsprechend - als eigenständiges Modul aufgefasst werden können. Bei der Modellierung wurde versucht, jede (Teil-)Anforderung auf ein eigenes Modul, d. h. in diesem Fall als Klasse oder Methode abzubilden.

Die Klasse K_1 bildet beispielsweise die dritte Anforderung ab, wobei die einzelnen Teilanforderungen jeweils in den Methoden M_1, M_2 und M_3 implementiert wurden. Die verschiedenen Experimente (Anforderung 2) wurden auf eine Klassenhierarchie ($K_2 \dots K_4$) abgebildet. Schließlich wurden die verschiedenen Anzeigearten (Anforderung 1.1 und 1.2) in zwei Komponenten (C_2 und C_3) implementiert.

Das ist nur eine mögliche Modularisierung aus den Anforderungen des DCL, es gibt verschiedene andere Modularisierungen, die die Anforderungen auch umsetzen. Die Frage lautet: Wann ist eine gefundene Modularisierung sinnvoll, bzw. gibt es Kriterien, wonach eine Modularisierung besser als die andere ist?

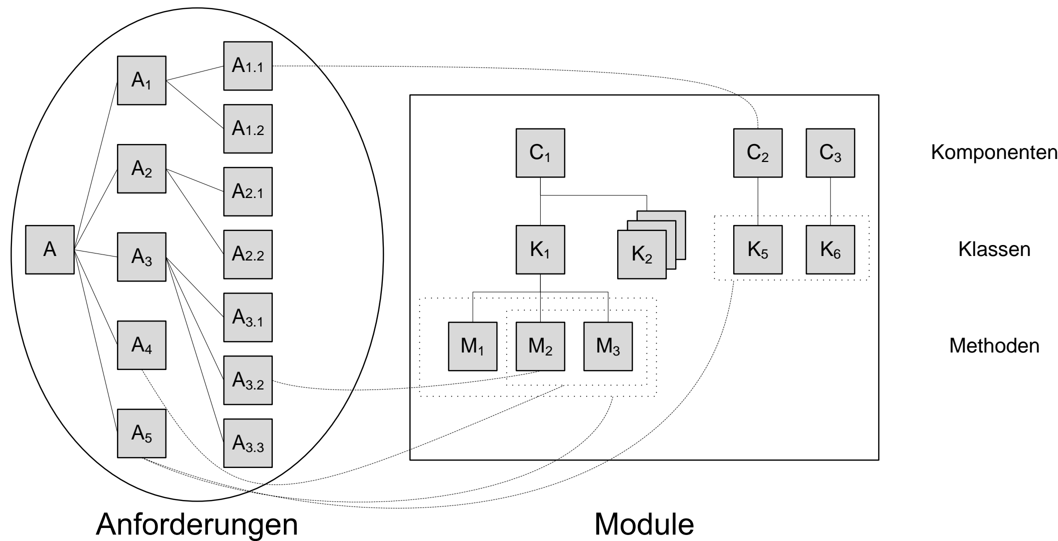


Abbildung 2.2: Abbildung von Belangen auf die Programmstruktur

Grundsätzlich treten zwei Probleme bei der Modularisierung auf:

1. Die Umsetzung einer Anforderung ist über mehrere Module verteilt (*Streuung*) und
2. in einem Modul werden verschiedene Anforderungen umgesetzt (*Durchsetzung*).

Streuung und Durchsetzung sind Probleme, die sich negativ auf die Entwicklung, Wartung und letztlich die Qualität der Anwendung auswirken [Eaddy u. a. 2008; Greenwood u. a. 2007; Laddad 2002; Tsang u. a. 2004]. Insbesondere die Streuung führt dazu, dass die betroffenen Module nicht als unabhängige Einheit betrachtet werden können. Das ist insbesondere bei einer verteilten Anwendungsentwicklung mit vielen Entwicklern problematisch. Aber sie führt auch zu redundantem Quelltext, da die Anforderung in jedem betroffenen Modul erneut implementiert werden muss.

In Abbildung 2.2 ist das Problem der Streuung gut zu erkennen. Die Unterstützung des Rollenkonzepts (A_5) hat Auswirkungen auf fast jedes Modul der Anwendung. So muss in den Methoden M_1,M_2 und M_3 jeweils überprüft werden, ob der Anwender tatsächlich die Berechtigung zum Ausführen der entsprechenden Aktionen hat. In der Präsentationsschicht (Komponente C_2 bzw. C_3) hingegen muss die Identität des Benutzers zuerst ermittelt und vorgehalten werden (z. B. durch ein Anmeldefenster mit entsprechenden Datenstrukturen), und es muss im Fehlerfall auch eine entsprechende Meldung an den Benutzer generiert werden. Die Folge ist, dass das Rollenkonzept nicht losgelöst betrachtet werden kann. Außerdem muss die Überprüfung der aktuellen Rolle in den Methoden M_1,M_2 und M_3 redundant implementiert werden.

Die Durchsetzung eines Moduls mit verschiedenen Anforderungen wird besonders in den Methoden M_2 und M_3 deutlich. Die Methode M_2 ist verantwortlich für die Statusermittlung ($A_{3.2}$). In sie muss das Rollenkonzept implementiert werden (A_5), und zusätzlich auch ein Zwischenspeicher für die schnelle Abwicklung (A_4). Hier entstehen mehrere Probleme: Der Programmierer dieses Moduls muss gleichzeitig ein Experte für verschiedene Problemdomänen sein, also für die Verwaltung der Experimente, die Sicherheit und das Laufzeitverhalten. Auf der anderen Seite ist es schwierig, im Falle der Weiterentwicklung auf geänderte Anforderungen zu reagieren, da die zugehörigen Quelltextteile in diesem Modul erst aufwendig identifiziert werden müssen.

In einer guten Modularisierung würde jede einzelne Anforderung auf genau ein einziges Modul (Komponente, Klasse oder Methode) abgebildet werden. Je geringer die Streuung

und die Durchsetzung einer Modularisierung ist, umso besser ist die Modularisierung. Das lässt sich jedoch mit objektorientierten Formalismen nicht erreichen [Kiczales u. a. 1997; Tarr u. a. 1999]. Der Grund ist, dass es stets eine *dominante Zerlegung* gibt, an der sich die Modulstruktur ausrichtet. In den meisten Fällen wird die Zerlegung getrieben durch die funktionalen Belange der Anwendung. Eine alternative Modularisierung für das DCL hätte z. B. das Rollenkonzept als zentrales Modul.

Bevor jedoch eine Lösung für dieses Problem vorgestellt wird, soll der Fokus noch einmal auf die Schnittstellen der Module und die Komposition einer Anwendung aus Einzelmodulen gerichtet werden.

2.2 Die Komposition objektorientierter Anwendungen

In der Annahme, dass eine Modularisierung aus den Anforderungen gefunden wurde, kann mit der Entwicklung der Anwendung begonnen werden. Die Modularisierung ermöglicht es nun, die Entwicklung der Module auf mehrere Programmierer zu verteilen. Pro Modul wird es einen Programmierer (im Falle von *Extreme Programming* [Beck 1999] eventuell auch zwei) geben, der das implementiert. Alternativ kann auch auf bereits existierende Module zurückgegriffen werden. Entscheidend hierbei ist, dass die Schnittstellen zwischen den Modulen klar definiert sind.

Die Komposition erfolgt dabei stets über das Schema, dass ein Modul einen Dienst über seine Schnittstelle *anbietet* und ein anderes Modul diesen Dienst über die Schnittstelle *nutzt*. Die Nutzung erfolgt dabei durch einen *expliziten* Aufruf an der Schnittstelle. Für den Nutzer ist die konkrete Implementation des Dienstes dabei völlig unerheblich. Diese kann sogar beliebig ausgetauscht werden, solange sich die Schnittstelle nicht ändert. Für den Anbieter auf der anderen Seite ist es ebenso unerheblich, wie die Implementation des Nutzers aussieht. In vielen Fällen wird es sogar die verschiedensten Nutzer mit den unterschiedlichsten Implementationen geben.

Wie die Definition der Schnittstelle erfolgt, wurde bereits erörtert, entweder durch den Anbieter selbst - indem er Komponenten mit öffentlichen Klassen und Klassen mit öffentlichen Methoden definiert - oder unabhängig über das Konzept der Interfaces. Ein Grund für die Streuung und Durchsetzung ist, dass die Schnittstelle eines Moduls stets *explizit* genutzt werden muss.

Bezogen auf das DCL-Beispiel in Abbildung 2.2 könnte man z. B. den Zwischenspeicher für die schnellen Antwortzeiten in einem eigenen Modul implementieren. Trotzdem muss die Schnittstelle des Zwischenspeichermoduls explizit vom Nutzer (den Methoden M_2 und M_3 der Klasse K_1) aufgerufen werden. Die Implementation ist nun zwar nicht mehr direkt mit dem Quelltext für die Zwischenspeicherung durchsetzt, aber das Problem ist trotzdem noch nicht gelöst, denn

1. der Programmierer muss sich trotzdem mit der Schnittstelle der Zwischenspeicherung befassen, obwohl die Zwischenspeicherung nichts mit der Anforderung zu tun hat, die er umzusetzen hat,

2. der Aufruf des Zwischenspeichermoduls ist redundant über mehrere Module verteilt und

3. der Schnittstelle des Nutzers sieht man nicht an, dass sie auch die Anforderung der „schnellen Antwortzeiten“ umsetzt.

Der erste Punkt zwingt dem Programmierer auf, sich mit Anforderungen zu beschäftigen, die ihn normalerweise nicht interessieren müssten. Sein Quelltext ist weiterhin mit Belangen durchsetzt, die nichts mit der Anforderung zu tun haben; auch wenn es in diesem Fall nur

```
1  [Serializable]
2  public class Foo
3  {
4    protected int baz;
5
6    [FileIOPermission(SecurityAction.Demand, Unrestricted=true)]
7    public void Bar()
8    {
9      ...
10   }
11 }
```

Listing 2.1: Annotationen in .NET

der Aufruf des Zwischenspeichermoduls ist. Eigentlich sollte er nur Module verwenden, die tatsächlich für die Umsetzung seiner Anforderung notwendig sind. So entsteht jedoch eine viel zu enge Kopplung zwischen Modulen, die nichts miteinander zu tun haben sollten.

Einfacher wäre es, wenn der Programmierer erklären könnte, dass sein Modul zusätzlich die Zwischenspeicherung verwendet, diese Verwendung aber *unabhängig* von seiner konkreten Implementation ist; was letztendlich die Durchsetzung des Quelltexts vermeidet. Es gibt zwar weiterhin einen Nutzer und einen Anbieter, beide wären aber nicht über deren Implementation, sondern ausschließlich über die Schnittstelle gekoppelt. Da diese Verwendung für beliebige Module erklärt werden kann, ließe sich auch die Streuung vermeiden.

Die Frage ist nun, wie eine solche „Erklärung“ des Programmierers aussehen könnte. Darauf geben neuere objektorientierte Programmiersprachen wie Java und C# mit dem Konzept der *Annotation* eine Antwort.

2.3 Der Annotationen in objektorientierten Sprachen

Mit der Einführung der *Common Language Infrastructure* (CLI) und Microsoft.NET wurden *CLI-Attribute* eingeführt [Microsoft Corporation u. a. 2006, Teil I, S. 84 ff.]. Nicht zu verwechseln mit einem *Attribut*, das ein Merkmal eines Objekts darstellt, sind CLI-Attribute eine besondere Form einer Klasse, die verschiedene Strukturelemente annotieren können. Zu den Strukturelementen gehören unter anderem Klassen, Assemblies (Komponenten), Methoden, Interfaces, aber auch einzelne Parameter. Ziel einer solchen Annotation ist es, diese Strukturelemente mit Informationen anzureichern. Die Klassenbibliothek kennt vordefinierte CLI-Attribute, jedoch kann auch der Entwickler eigene CLI-Attribute definieren. Erstere werden im Allgemeinen von der Laufzeitumgebung oder dem Kompiler evaluiert, Letztere lassen sich zur Laufzeit durch spezielle Bibliotheksfunktionen auswerten. Java implementiert dieses Konzept unter dem Namen *Annotations* ab der Version fünf [Gosling u. a. 2005, S. 281 ff.].

Bei näherer Betrachtung der .NET-Klassenbibliothek ist festzustellen, dass mit Annotationen auch zusätzliches Verhalten zu Modulen hinzugefügt werden kann, ohne dieses explizit zu implementieren. Listing 2.1 zeigt eine Klasse, die mit den CLI-Attributen `Serializable` und `FileIOPermission` annotiert wurde. Der Programmierer drückt hiermit aus, dass Objekte dieser Klasse serialisierbar sein sollen und Aufrufer der Methode `Foo` das Recht haben müssen, Dateioperationen ausführen zu dürfen. Listing 2.2 zeigt demgegenüber, wie die Klasse aussehen müsste, um dieselbe Funktionalität ohne Annotationen zu erreichen.

Dem Quelltext in Listing 2.2 ist vielleicht noch anzusehen, dass es sich um eine serialisierbare Klasse handelt. Doch um zu erkennen, dass die Methode `Bar` zusätzliche Sicherheitsabfragen ausführt, muss genauer analysiert werden. Die Klassendefinition und die Implementation der Methode `Bar` ist offensichtlich mit Quelltext für andere Anforderungen durchsetzt. In Listing 2.1 erfolgt demgegenüber nur eine Erklärung (Deklaration), dass die Klasse und die

```
1  public class Foo:ISerializable
2  {
3    protected int baz;
4
5    protected Foo(SerializationInfo info, StreamingContext context)
6    {
7      this.baz = info.GetInt32("baz");
8    }
9
10   void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
11   {
12     info.AddValue("baz", baz);
13   }
14
15   public void Bar()
16   {
17     FileIOPermission perm = new FileIOPermission(PermissionState.Unrestricted);
18     perm.Demand()
19
20     ...
21   }
22 }
```

Listing 2.2: Implementation ohne Annotationen

Methoden zusätzliche Anforderungen umsetzen. Die Implementation dieser Anforderungen erfolgt aber offensichtlich an anderer Stelle.

Die Implementation dieser beiden Anforderungen ist fest eingebaut in die Laufzeitumgebung. Diesem Ansatz fehlt jedoch die gewünschte Flexibilität: Es ist nicht möglich, CLI-Attribute zu definieren, in denen eine eigene Implementation eingebaut ist.

2.4 Der LOOM-Ansatz

Annotationen eignen sich, eigene Module mit der Information zu versehen, wenn sie mit zusätzlichem Verhalten angereichert werden sollen. Allerdings fehlt ein Ansatz, mit dem die Module bei ihrer Verwendung dann tatsächlich dieses zusätzliche Verhalten aufweisen. Der vom Autor entwickelte LOOM-Ansatz soll diese Lücke schließen.

Mit LOOM werden neue Modularisierungskonzepte eingeführt, mit denen Streuung und Durchsetzung vermieden werden können. Ein *Aspekt* ist eine neue Art von Modul, in dem alle diejenigen Anforderungen implementiert werden, die bei einer Zerlegung andere Module durchsetzen oder über verschiedene Module verstreut sein würden.

Aspekte stehen auf derselben Ebene wie Klassen. Wie die Klassen können Aspekte in Komponenten verpackt werden und gruppieren selbst wieder Methoden und Attribute. Im Gegensatz zu Klassen können Aspekte jedoch weitere Elemente enthalten: *Advices*, *Introduktionen* und *Joinpoint-Variablen*.

In einem Advice wird Verhalten implementiert, das stets in Verbindung mit Methoden anderer Module abgewickelt werden soll. Die Ausführung eines Advices wird nie explizit initiiert, sondern erfolgt immer im Kontext einer anderen Methode.

Mit Introduktionen wird die Schnittstelle anderer Klassen erweitert. Dabei erfolgt die Implementation dieser Schnittstelle nicht in der Klasse, sondern in der Introduktion.

Aspekte, Advices und Introduktionen sind keine neuen Konzepte. Sie werden schon von [Teitelman 1966] und [Kiczales u. a. 1997] beschrieben. Neu ist jedoch deren Verwendung durch das Mittel der Annotation. Während in AOP Aspekte nicht sichtbar sind und es keinen Schnittstellenbezug gibt, existiert bei LOOM ein eigenes sichtbares Modul mit eigener Schnittstelle. Advices und Introduktionen sind Teil dieser Schnittstelle. Die Nutzung des Aspekts muss explizit durch die Annotationsrelation erfolgen, die im nächsten Abschnitt ausführlich erklärt werden soll.

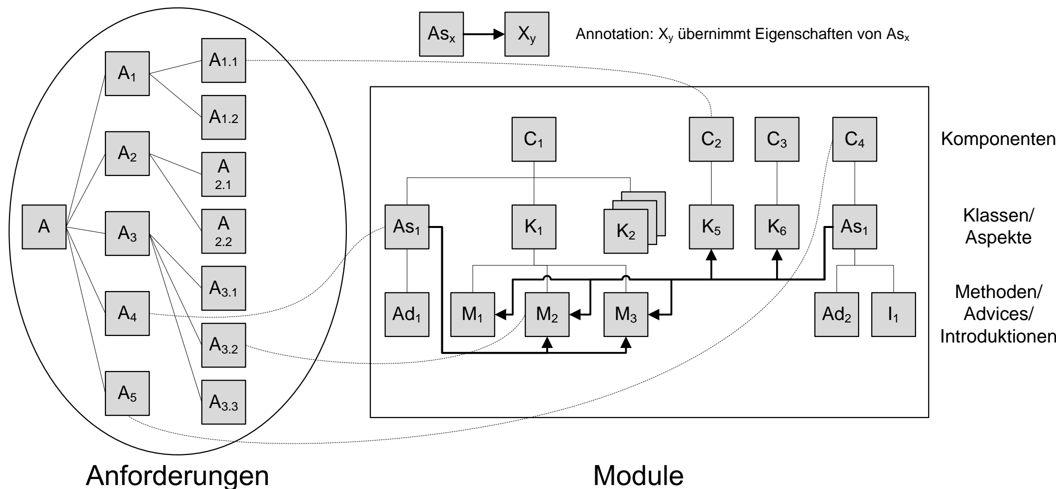


Abbildung 2.3: Abbildung von Belangen mit Vererbung und Annotation

Neben den Advices und Introduktionen können Aspekte spezielle Attribute, die *Joinpoint-Variablen*, deklarieren. Während ein Attribut einer Klasse in deren Exemplaren (oder in der Klasse selbst, wenn es sich um ein statisches Attribut handelt) gespeichert wird, ist das bei den Joinpoint-Variablen der Aspekte nicht der Fall. Joinpoint-Variablen sind ein neues LOOM-spezifisches Konzept und eignen sich zur Modellierung überschneidender Daten. Die Speicherung erfolgt in Exemplaren anderer Klassen oder im Kontext einer anderen Methode.

Schließlich wird mit dem Joinpoint-Kontext außerdem ein Konzept eingeführt, das es einem Aspektprogrammierer erlaubt, die Schnittstelle anderer Module als ein abstraktes Ding zu betrachten. Ohne die Schnittstelle konkret zu kennen, kann er auf ihr Aktionen ausführen und Nachrichten manipulieren.

In Abbildung 2.3 ist eine Modularisierung des DSUP-Beispiels unter Berücksichtigung der LOOM-Konzepte dargestellt. Die Anforderung A_4 für schnelle Antwortzeiten und die Anforderung A_5 wurden jeweils in einen eigenen Aspekt (As_1 und As_2) ausgelagert. Die Annotationen sind durch einen Pfeil, ausgehend von den Aspekten zu denjenigen Modulen, auf denen die Aspekte wirken sollen, gezeichnet.

Obwohl die Modulstruktur auf den ersten Blick komplexer aussieht, wird der Quelltext durch die Entflechtung einfacher und in Summe auch kürzer, da die Redundanzen entfernt wurden. Mit dieser Modulstruktur ist es einfach, verschiedene Programmierer den einzelnen Anforderungen zuzuordnen. Insbesondere ist es leicht, z. B. den Domänenexperten für Sicherheit das Rollenkonzept entwickeln zu lassen.

Bei der Entwicklung seiner Module wird der Programmierer den Prozess der Zerlegung fortsetzen. Für seine Module wird er Hilfsklassen und Methoden definieren, um eine Implementation für bestimmte *Belange* vorzunehmen. Bisher wurde immer von Anforderungen gesprochen, die von Modulen umgesetzt werden. Anforderungen sind die grobe Sicht auf das zu entwickelnde System und sind hauptsächlich durch die Fachlichkeit und den Betrieb der Anwendung getrieben. Die Anforderungen an eine Anwendung sind oft in einem entsprechenden Dokument festgehalten. Belange leiten sich aus den Anforderungen ab, sind aber auch diejenigen Details, die sich bei der Entwicklung ergeben, weil z. B. ein bestimmtes Framework, Betriebssystem oder Ähnliches verwendet wird.

Wie die Anforderungen im Großen können die Belange im Kleinen zu Streuung und Durchsetzung führen. Die LOOM-Konzepte können hier in der selben Art und Weise verwendet werden, um das zu vermeiden. Im Folgenden sollen die Konzepte im Detail vorgestellt werden.

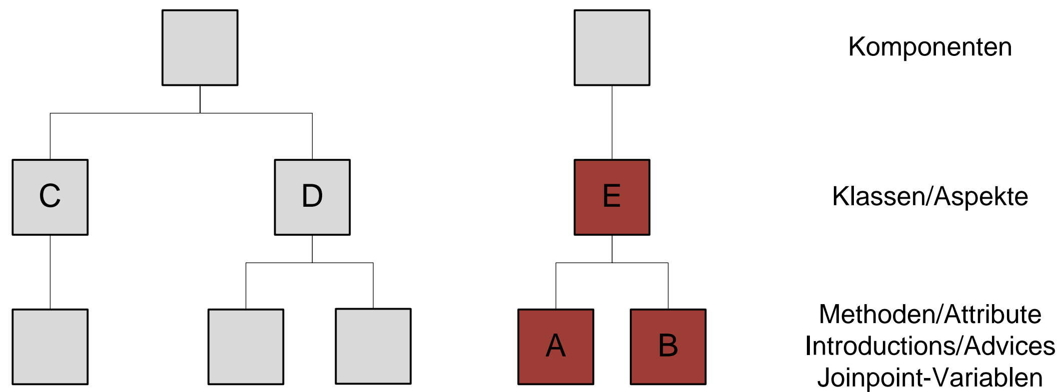


Abbildung 2.4: Abstrakte Darstellung von Aspekten und Modulen

2.5 Die Annotationsrelation

Die *Annotationsrelation* kann zwischen *Aspekten* einerseits und Komponenten, Klassen und Methoden andererseits hergestellt werden und dient dazu, den Wirkungsbereich von Advices, Introduktionen und Joinpoint-Variablen durch den Programmierer explizit festzulegen. LOOM führt hier das neue Konzept der *Überdeckung* ein. Annotiert ein Aspekt ein Modul, so überdeckt er dieses Gruppierungselement und eventuell weitere in dem Modul enthaltene Module. Nicht annotiert werden können Aspekte selbst, deren Methoden sowie die Advices und Introduktionen. Es gibt zwei verschiedene Arten der Überdeckung, die *vertikale* und die *horizontale*.

Ausgangspunkt für die folgenden Erklärungen ist die Darstellung in Abbildung 2.4. Sie beschreibt eine Komponente mit zwei Klassen (C und D), die jeweils eine bzw. zwei Methoden haben. Rechts daneben ist farblich hervorgehoben ein Aspekt E dargestellt, der einen Advice (A) und eine Introduktion (B) enthält.

In Abbildung 2.5 ist die *vertikale Überdeckung* jeweils mit dem Wirkungsbereich für die verschiedenen Module dargestellt. Der Aspekt E überdeckt dabei jeweils eine gesamte Komponente, eine einzelne Klasse sowie eine einzelne Methode. Die Annotation ist durch einen Pfeil vom Aspekt zum entsprechenden Modul gekennzeichnet. Die Überdeckung ist deshalb vertikal, weil sie alles unterhalb (im Sinne von alle enthaltenen Module) und einschließlich des annotierten Moduls betrifft.

Überdeckt ein Aspekt eine Komponente, so wirkt ein Advice (A) potentiell auf jeder in dieser Komponente definierten Methode, erkennbar durch die entsprechende farbliche Markierung. Die Introduktion (B) fügt zu jeder Klasse der Komponente potentiell eine neue Methode hinzu. Potentiell deshalb, da durch entsprechende *Pointcut-Definitionen* sowohl in den Advices als auch bei den Introduktionen der Wirkungsbereich explizit eingeschränkt werden kann. Das wird im nächsten Abschnitt ausführlicher besprochen und vorerst unberücksichtigt bleiben.

Die zweite Grafik in Abbildung 2.5 zeigt eine von einem Aspekt annotierte Klasse. Hier beschränkt sich der Wirkungsbereich des Aspekts auf die Methoden der Klasse und die Klasse selbst. Aber auch hier können Introduktionen neue Methoden einführen und somit die Schnittstelle der Klasse erweitern.

Annotiert ein Aspekt eine einzelne Methode, wie es in der letzten Grafik dargestellt ist, so ist der Wirkungsbereich für die Advices auf diese Methode beschränkt. Mit Introduktionen können „neben“ diese Methode trotzdem weitere Methoden eingeführt werden. Eine typische Konstruktion für die Methodenannotation mit zusätzlicher Introduktion ist, dass ein Aspekt durch einen Advice eine Methode mit einem zusätzlichen Belang bereichert. Die Implementation dieses Belanges benötigt jedoch eine zusätzliche Schnittstelle zur Außenwelt. Solche Introduktionen müssen statisch implementiert werden. Dies wird in Abschnitt 2.7.6

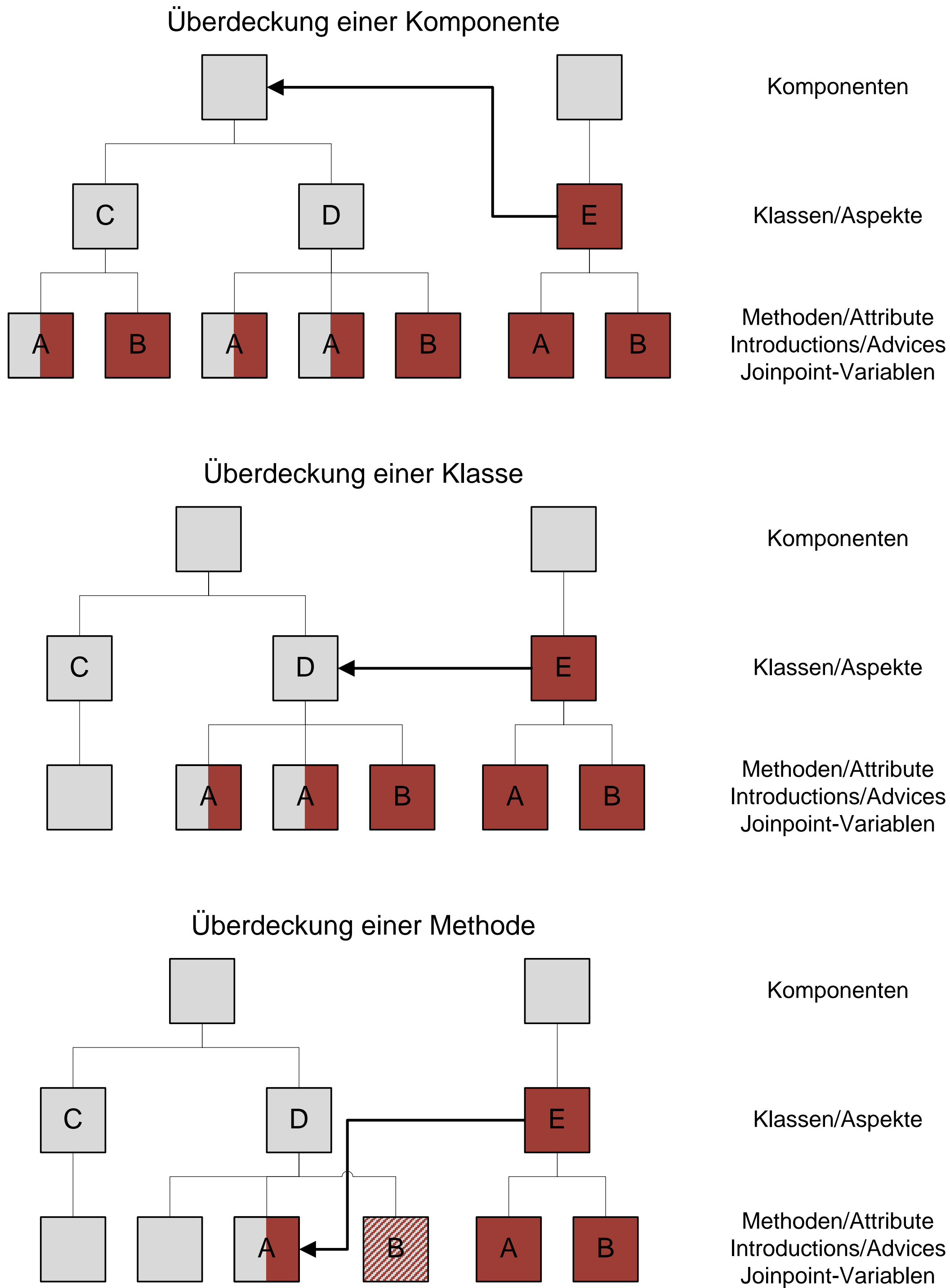


Abbildung 2.5: Vertikale Überdeckung von Aspekten

noch genauer erläutert.

Wie mit Advices Verhalten zu Methoden hinzugefügt und mit Introduktionen neue Methoden in Klassen eingeführt werden können, lassen sich mit Joinpoint-Variablen Attribute zu Methoden hinzufügen bzw. in Klassen einführen. Während Advices und Introduktionen Konzepte sind, um überschneidende Belange zu vermeiden, sind die Joinpoint-Variablen ein Konzept, um überschneidende Daten zu vermeiden. Auch hier gelten dieselben Regeln der Überdeckung, wie sie in Abbildung 2.5 dargestellt werden.

Ist A eine Joinpoint-Variable, die auf Methoden definiert werden soll (im Folgenden präziser: direkt auf dem Joinpoint), so existiert sie durch die Annotation für jede überdeckte Methode genau einmal. Jede in Abbildung 2.5 grau dargestellte Methode erhält zusätzlich ein Attribut A. Im Falle der Überdeckung einer Komponente definiert die Klassen C eine und die Klasse D zwei Methoden, somit ist A insgesamt dreimal in der Struktur vertreten. Das entspricht der Überdeckung von Advices.

In Analogie zu den Introduktionen ist B eine Joinpoint-Variable, die in Klassen eingeführt werden soll. Entsprechend den Überdeckungsregeln findet sie sich folglich in der Komponentenüberdeckung aus Abbildung 2.5 in jeder überdeckten Klasse wieder. Wird hingegen nur eine Klasse annotiert, wird nur in diese Klasse das entsprechende Attribut eingeführt. Bei der Annotation von Methoden gilt, dass B auch in die Klasse eingeführt wird, obwohl die Annotation auf der Ebene der Methoden erfolgte.

Die *horizontale Überdeckung* erweitert den Wirkungsbereich der Aspekte auf die Vererbungshierarchie der Klassen und die Implementation von Interfaces. Horizontal bedeutet, dass sich die Überdeckung auf der Ebene der Klassen und Interfaces über die bestehenden Vererbungs- und Implementationsrelationen fortpflanzt.

Durch die in Abbildung 2.6 dargestellte Ableitung der Klasse C von der durch Aspekt E annotierten Klasse D wird der Wirkungsbereich des Aspekts von der Klasse D auf die Klasse C erweitert. Auf die Methode A_1, die die Klasse C von D erbt, wirkt der Advice A ebenso wie auf die in C neu eingeführte Methode A_2. Die Methode B wird als Introduktion sowohl in die Klasse D als auch in C eingeführt.

Bei der Implementation von Interfaces ergibt sich ein ähnliches Bild. D ist ein Interface mit den beiden Methoden A und B, das von der Klasse C implementiert wird. Da das Interface mit einem Aspekt annotiert ist, werden die Implementationen des Interfaces in der Klasse C folglich auch von diesem Aspekt überdeckt. Der Advice wirkt sich also auch auf die beiden Methodenimplementationen aus.

Annotiert ein Aspekt wie im letzten Fall nur eine einzelne Methodendeklaration eines Interfaces, so sind auch nur dessen Implementationen vom Aspekt überdeckt. Implementationen des restlichen Interfaces bleiben in diesem Fall vom Aspekt unberührt.

Die Annotation von Interfaces erlaubt in LOOM die Definition einer Standardimplementierung für Interfaces. Das stellt zwar indirekt eine Mehrfachvererbung dar, da die Implementation aber aus einem Aspekt stammt, ergeben sich nicht die damit verbundenen Probleme: Eine Klasse kann von einem Aspekt nicht mehrfach überdeckt werden. Obwohl Klasse und Aspekt durch die Überdeckung komponiert werden, hat jeder Teil zur Laufzeit eine eigene unabhängige Identität.

Im Gegensatz zur vertikalen kann die horizontale Überdeckung explizit bei der Definition eines Aspekts ausgeschlossen werden. In diesem Fall beschränkt sich der Wirkungsbereich des Aspekts ausschließlich auf die annotierte Klasse. Eine vertikale Überdeckung von annotierten Komponenten existiert nicht.

Der Programmierer kann mit diesem Konzept wie beim Zusammenfassen von Gemeinsamkeiten in einer Basisklasse überschneidende Belange aus einem oder mehreren Modulen herauslösen.

Mit dem Konzept der Aspektannotation kann somit zusätzlicher Quelltext in Methoden eingefügt und Methoden sowie Attribute zu Klassen hinzugefügt werden. Dieses „Einfügen“

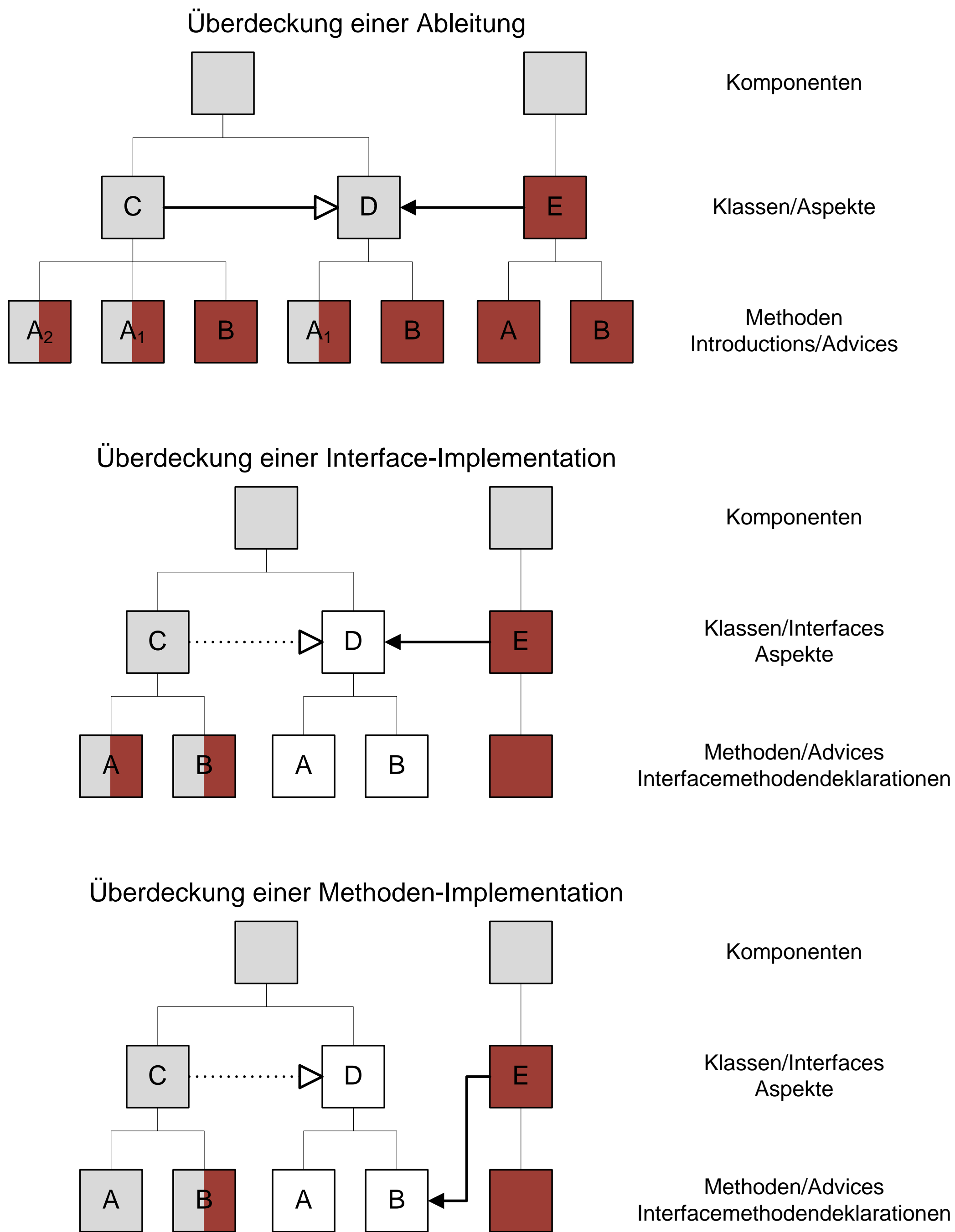


Abbildung 2.6: Horizontale Überdeckung von Aspekten

und „Hinzufügen“ vollzieht sich nicht in der Implementation, sondern wird durch die Annotation und die Aspektdeklaration nur formal beschrieben. Der Programmierer kann mit diesem Konzept wie beim Zusammenfassen von Gemeinsamkeiten in einer Basisklasse überschneidende Belange aus einem oder mehreren Modulen herauslösen.

2.6 Joinpoints in LOOM

Die Annotationsrelation reicht noch nicht aus, das Zusammenspiel zwischen herkömmlichen Modulen und Aspekten ausreichend zu definieren. So hängt es auch vom konkreten Belang des Aspekts ab, an welchen Stellen von ihm überdeckte Module tatsächlich erweitert werden sollen. Der Aspekt As_1, der in Abbildung 2.3 das Rollenkonzept implementiert, wird in der Präsentationsschicht ausschließlich die Methoden zur Berechnung der Bildschirmdarstellung erweitern, um den aktuell angemeldeten Benutzer anzuzeigen. Er überdeckt aber die gesamte Klasse K_5 und würde somit auch auf Methoden Auswirkungen haben, die Benutzerereignisse entgegennehmen.

Es ist erforderlich, dass Aspekte als Teil ihrer Schnittstelle beschreiben, wie ihre Advices wirken. Für Introduktionen ist das bereits klar, denn sie erweitern die Schnittstelle eines Moduls um die jeweilige Introduktion und haben daher keine Auswirkungen auf bestehende Methoden. Es gibt allerdings für Introduktionen genau dann eine Ausnahme, wenn die neu einzuführende Methode in der Schnittstelle schon vorhanden ist.

Die Beschreibung des Wirkungsbereiches von Advices erfolgt über so genannte *Joinpoints*. Dieser Begriff wird im Kontext von LOOM etwas anders verwendet als in der Aspektorientierten Programmierung nach [Kiczales u. a. 1997]. In LOOM sind die Joinpoints eines Modules die *impliziten* und *expliziten* Methoden der Schnittstelle des Moduls. Implizite Methoden sind Methoden, die in der Schnittstellendefinition des Moduls nicht vorkommen, aber trotzdem eine Kommunikation über die Modulgrenze hinweg bewirken. Das ist z. B. das Erzeugen eines Exemplars, für das kein Konstruktor definiert wurde. Explizite Methoden hingegen sind immer in der Schnittstelle des Moduls definiert.

Joinpoints sind die Punkte, an denen ein Aspekt über seine Advices mit einem anderen Modul verbunden werden kann. Die Verbindung dieser beiden Module - Aspekt und Komponente, Klasse oder Methode - basiert auf einer Schnittstelle und ist unabhängig von Implementation und Programmzustand. Das verhindert die in Kapitel 1 beschriebene Zerstörung der Modularisierung und sorgt dafür, dass das Kooordinatensystem für den Programmierer erhalten bleibt.

Jeder Joinpoint zeichnet sich durch Attribute aus, über die er selektiert werden kann. Diese Attribute leiten sich direkt aus der vom Joinpoint abstrahierten Schnittstellenmethode ab und sind:

- die Art der Schnittstellenmethode,
- die Metadaten an der Schnittstellenmethode,
- der Name der Schnittstellenmethode,
- der Name der Schnittstelle und
- eine Beschreibung der an der Schnittstelle übertragenen Daten.

Hinter einer Modulinteraktion zur Laufzeit einer Anwendung steht im Allgemeinen ein Exemplar einer Klasse. Konkret bedeutet das, dass zu einem Aufruf einer Schnittstellenmethode - mit der Ausnahme der statischen Methoden - stets ein Objekt gehört, auf dem der Aufruf abgewickelt wird. Die Art der Schnittstellenmethode klassifiziert die Schnittstellenmethode nach ihrer Verantwortlichkeit für den Lebenszyklus des Objektes, auf dem sie ausgeführt wird. Dieser Lebenszyklus ist in Abbildung 2.7 dargestellt und bedeutet:

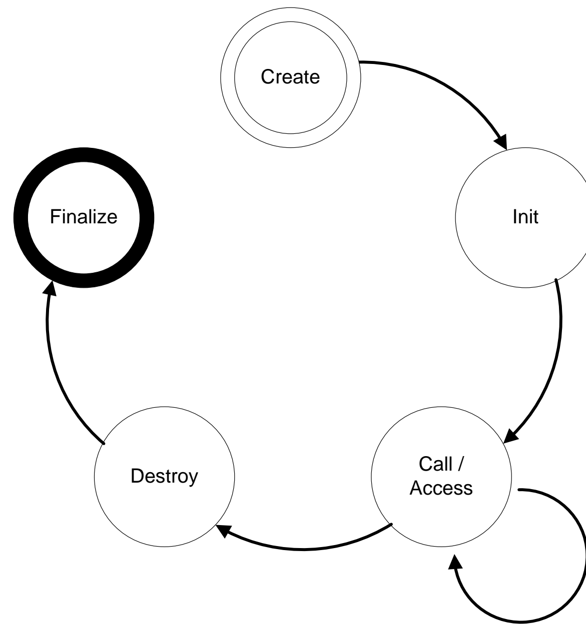


Abbildung 2.7: Lebenszyklus von Objekten

- **create**: ein Exemplar einer Klasse wird erzeugt,
- **initialize**: ein Exemplar einer Klasse wird initialisiert,
- **call**: Methoden eines Exemplars werden gerufen oder
- **access**: auf *Eigenschaften* des Exemplars wird zugegriffen (mit Eigenschaften sind hier spezielle Methoden zum Zugriff auf Attribute des Exemplars gemeint),
- **destroy**: ein Exemplar der Klasse wird zerstört und
- **finalize**: ein Exemplar der Klasse wird freigegeben.

Mithilfe der Joinpointattribute lassen sich Prädikate formulieren, mit denen diejenigen Joinpoints selektiert werden, die für die Implementierung des jeweiligen Belangs notwendig sind. Diese Prädikate müssen nicht Aussagen über alle Attribute des Joinpoints machen. Oft reicht es aus, alle Joinpoints zu selektieren, an denen „*Methoden eines Exeplares gerufen werden*". Für die Rollenverwaltung ist es völlig unerheblich, welchen Namen eine Schnittstellenmethode hat und wie die Schnittstelle selbst heißt, wenn die allgemeine Zugriffsberechtigung auf Methoden eines Exemplars überprüft werden muss.

Ein Joinpoint kann nur dann selektiert werden, wenn er durch den zum Advice gehörenden Aspekt überdeckt wurde. Das Prädikat zur Auswahl einer Menge von Joinpoints ist Teil der Advicedeklaration und damit auch Teil der Schnittstelle eines Aspekts.

Der Nutzer eines Aspekts legt durch die Annotation explizit fest, welche Joinpoints ausgewählt werden können, sieht aber auch über die Aspektschnittstelle, welche Joinpoints tatsächlich selektiert werden. Damit ist ihm stets bewusst, an welchen Stellen seines Moduls eine Erweiterung durch den Aspekt erfolgt.

2.7 Die Modellierungssprache Loom.Uml

Im Folgenden wird nun auf Basis der UML2 (*Unified Modelling Language Version 2*) [Object Management Group 2007a,b], die konkrete Definition der Schnittstelle eines Aspekts erläutert. Dabei können so auch weitere Loom-Konzepte anschaulich eingeführt werden. UML2

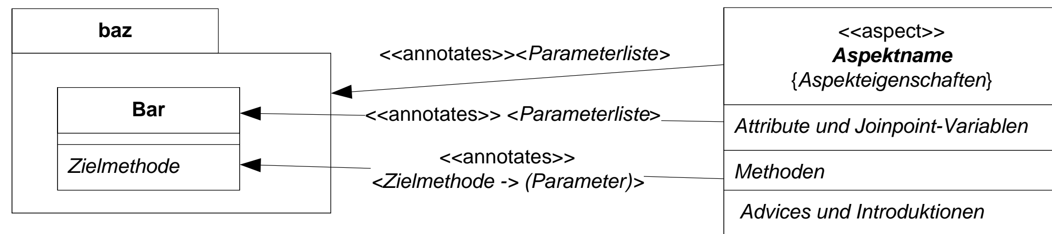


Abbildung 2.8: Erweiterte UML-Darstellung eines Aspekts

als allgemeine objektorientierte Modellierungssprache wurde gewählt, weil sie sich einfach erweitern lässt und eine hohe Verbreitung hat [Google Trends 2008]. Ziel ist, die Konzepte so einfach und verständlich wie möglich in der Modellierungssprache umzusetzen, um eine intuitive Verwendung durch Dritte zu ermöglichen.

Die für die Darstellung der LOOM-Konzepte vom Autor vorgenommenen Erweiterungen der UML2 werden im Folgenden als LOOM.UML bezeichnet. Des Weiteren wird nachfolgend die Abkürzung *UML* verwendet, diese bezieht sich aber stets auf die Version 2, also UML2 und nicht die entsprechende Vorgängerversion.

2.7.1 Die Definition von Aspekten

Abbildung 2.8 zeigt eine allgemeine Darstellung eines Aspekts mit einer Beziehung zu einer Klasse und einer Komponente in LOOM.UML-Notation. Der Aspekt ist eine Erweiterung des UML-Klassenkonzeptes und wird durch den Stereotypen `«aspect»` gekennzeichnet. Des Weiteren erhält er mit `Aspektname` einen eindeutigen Bezeichner.

In *Aspekteigenschaften* können optional zwei zusätzliche Eigenschaften des Aspekts in geschweiften Klammern angegeben werden. Einerseits kann mit dem Schlüsselwort **per** die Exemplarbildung des Aspekts gesteuert werden - ausführlich in Abschnitt 2.7.3 beschrieben -, andererseits durch das Schlüsselwort **noninherited** die horizontale Überdeckung des Aspekts deaktiviert werden. Die horizontale Überdeckung ist dafür verantwortlich, dass auch die Joinpoints von abgeleiteten Klassen sowie die Implementation von Interfaces von einem Aspekt überdeckt werden. Annotiert ein Aspekt mit der **noninherited**-Eigenschaft eine Klasse, so werden Spezialisierungen dieser Klasse von ihm nicht überdeckt. Das gilt auch, wenn der Aspekt ein Interface annotiert. Dabei wird die Realisierung des Interfaces nicht vom Aspekt überdeckt.

Aspekte können wie in Klassen normale Attribute und Methoden definieren. Genau wie bei der Definition einer UML-Klasse erfolgt das im zweiten und dritten Kasten. Zusätzlich kann der zweite Kasten aber auch *Joinpointvariablen* aufnehmen. Diese dienen dazu, Attribute mit annotierten Modulen zu teilen. Eine ausführliche Beschreibung der Joinpointvariablen geschieht im Abschnitt 2.7.7.

Für Advices und Introduktionen wird gegenüber einer UML-Klassendefinition ein weiterer Kasten unter den Methoden aufgeführt. Wie Advices definiert werden und wie die Selektion der Joinpoints in dieser Definition erfolgt, wird ab Abschnitt 2.7.4 dargelegt. Das Konzept der Introduktionen werden in Abschnitt 2.7.6 beschrieben.

Aspekte können wie Klassen vererbt werden. Der Kindaspekt übernimmt dabei alle Eigenschaften des Elternaspekts, inklusive der Joinpointvariablen, der Advices und der Introduktionen.

Ein wesentliches Konzept von LOOM ist die Annotationsrelation. Die Modellierung dieser Relation erfolgt wie in Abbildung 2.8 dargestellt durch einen ausgefüllten Pfeil mit dem Stereotyp `«annotates»`. Sie wird im folgenden Abschnitt in Bezug auf ihre Verwendung in LOOM.UML genauer beschrieben.

```
         Annotation   =   ‘‘«annotates»‘‘ [‘‘<‘‘ Parameterliste ‘‘>‘‘]
     Parameterliste   =   Parameter [‘‘,‘‘ Parameterliste];
          Parameter   =   Aspektparameter | Methode ;
            Methode   =   Methodendeklaration [‘‘-‘‘ Aspektparameter];
    Aspektparameter   =   ‘‘(‘‘ Parameterwert {‘‘,‘‘ Parameterwert} ‘‘)‘‘;
Methodendeklaration       siehe Text
      Parameterwert       siehe Text
```

Abbildung 2.9: LOOM.UML-Syntax für den «annotates» Stereotyp

2.7.2 Annotationen in LOOM.Uml

Die Annotationsbeziehung wird in LOOM.UML als navigierbare Assoziationsbeziehung mit dem Stereotypen `«annotates»` dargestellt. Diese Beziehung verläuft stets von einem Aspekt zu einem zu überdeckenden Element. Das kann eine UML-Klasse oder ein UML-Paket sein.

UML-Pakete werden in LOOM.UML synonym zu Komponenten verwendet, da diese sich aus Sicht des Autors besser zur Darstellung eignen als die gleichnamigen UML-Komponenten. Insbesondere das Konzept, dass sich eine Komponente aus den in ihr enthaltenen Klassen zusammensetzt, wird damit besser wiedergegeben.

Sollen einzelne Methoden von einem Aspekt überdeckt werden, müssen ihre Namen und gegebenenfalls ihre Parametersignatur an der Annotation notiert werden. Das ist notwendig, da das UML-Modellierungskonzept nicht vorsieht, dass Methoden in dieser Darstellungsform mit dem Ende einer Assoziation verknüpft sind.

Die Parameter einer Annotation werden in spitzen Klammern ähnlich der Parametrisierung von Templates [vgl. Born u. a. 2004, S. 259 ff.] angegeben. Die Parameterliste besteht dabei aus zwei Teilen. Im ersten können explizit die zu annotierenden Methoden aufgezählt werden, im zweiten die Argumente, die bei der impliziten Erzeugung eines Aspektexemplars an den passenden Konstruktor des Aspekts übergeben werden. Abbildung 2.9 zeigt die Grammatik der Liste als erweiterte Backus-Naur-Form (EBNF) [International Organization for Standardization 2001].

Für die EBNF wird folgende Syntax verwendet: Nichtterminalsymbole werden jeweils auf der linken Seite einer Produktionsregel definiert. Optionale oder sich wiederholende Teile der Produktionsregel werden von eckigen oder geschweiften Klammern, und Terminalsymbole werden von Anführungszeichen eingeschlossen. Besteht die Wahl zwischen verschiedenen Ersetzungsmöglichkeiten, sind diese durch einen senkrechten Strich (`|`) voneinander getrennt. Eine Ersetzung erfolgt dabei stets mit exakt einer der aufgeführten Möglichkeiten.

Die in Abbildung 2.9 verwendeten Nichtterminalsymbole `Methodendeklaration` und `Parameterwert` werden nicht weiter aufgeschlüsselt, da sie den Methodendeklarationen und Parameterwerten der UML entsprechen.

Die verschiedenen Ausdrucksmöglichkeiten der Annotationsrelation soll an einem Beispiel kurz diskutiert werden. Bei der Annotation:

```
<<annotates>><(Rolle.Admin)>
```

handelt es sich um eine *parametrierte Annotation*. Mit ihr wird ausgedrückt, dass der Wert `Admin` aus der Aufzählung `Rolle` an den Konstruktor des Aspekts bei dessen Erzeugung übergeben wird. Der Zusammenhang zwischen Aspekterzeugung und Annotation wird ausführlich im nächsten Abschnitt behandelt. Die Annotation

```
<<annotates>><GetExperimentStatus, GetExperiment(id:int)>
```

wirkt hingegen nur auf die beiden Methoden `GetExperimentStatus` und `GetExperiment`. Die erste Methode ist hier unvollständig bezüglich ihrer Deklaration angegeben worden. Solange hierdurch keine Mehrdeutigkeiten entstehen und das Verständnis für das Modell darunter nicht leidet, ist das durchaus legitim. Annotiert ein Aspekt mehrere Methoden, kann so der Schreibaufwand etwas reduziert werden.

Möchte man für die Annotation einzelner Methoden eine Parametrierung vornehmen, so ist das in folgender Form möglich:

```
<<annotates>><GetExperimentStatus -> (Rolle.Gast), GetExperiment(id:
    int) -> (Rolle.Nutzer)>
```

Ein Aspekt darf ein Modul jeweils nur einmal überdecken. Überdeckt ein Aspekt ein Modul mehrfach, weil er beispielsweise eine Methode und die zugehörige Klasse annotiert, so ist das eine Fehler.

2.7.3 Aspekte zur Laufzeit

Bisher wurden Aspekte nur in ihrer Beziehung zu den Modulen betrachtet und damit, dass sie diese um bestimmte Belange erweitern können. Es wurde jedoch noch nicht die Frage diskutiert, wie sich Aspekte zur Laufzeit darstellen. In objektorientierten Programmiersprachen werden zur Laufzeit Exemplare von Klassen gebildet. Jedes Exemplar hat seine eigene Identität, unterscheidet sich von anderen Exemplaren, auch wenn alle Exemplare derselben Klasse sind. Die Bildung solcher Exemplare erfolgt explizit durch den Programmierer.

Da Aspekte - wie auch Klassen - Attribute definieren können, ist es wichtig zu wissen, wo diese Attribute zur Laufzeit gespeichert sind und in welcher Relation sie zu den vom Aspekt überdeckten Exemplaren der Klasse stehen. LoOM sieht vor, dass Aspekte zur Laufzeit als Aspektexemplare mit eigener Identität existieren. Diese Exemplare werden nicht explizit erzeugt; die Exemplarbildung erfolgt implizit über die Annotationsbeziehung und der optional in `Aspekteigenschaften` der Abbildung 2.8 angegebenen Vorschrift für die Aspekterzeugung:

- **per instance**: Jedes Exemplar einer durch die Annotation überdeckten Klasse hat ein eigenes Exemplar des annotierenden Aspekts. Das entspricht im Wesentlichen einer Kompositionsbeziehung, da das Aspektexemplar direkt vom Exemplar der Klasse abhängt. Es wird mit der Erzeugung der Klasse gebildet und auch gemeinsam mit ihm zerstört. Wenn keine **per**-Eigenschaft angegeben wurde, ist das das Standardverhalten.

- **per class**: Jede durch die Annotation überdeckte Klasse hat ein eigenes Exemplar des annotierenden Aspekts. Exemplare mit demselben dynamischen Typen teilen sich also jeweils ein Aspektexemplar.

- **per annotation**: Aus jeder Annotationsbeziehung entsteht zur Laufzeit ein eigenes Exemplar des annotierenden Aspekts. Exemplare aller von dieser Annotation überdeckter Klassen werden mit diesem Aspektexemplar verbunden.

- **per aspectclass**: Von jedem Aspekt existiert zur Laufzeit stets nur ein einziges Exemplar. Alle Exemplare aus den vom Aspekt überdeckten Klassen teilen sich dieses Aspektexemplar.

Bezogen auf den Übergang von der UML-Modellebene auf die UML-Objektebene wird aus einer Annotationsbeziehung eine Aggregationsbeziehung zwischen dem Exemplar einer vom Aspekt überdeckten Klasse und dem Aspektexemplar. Der Ursprung der Aggregation ist dabei das Exemplar der überdeckten Klasse. Ein Aspektexemplar kann dabei an verschiedenen Aggregationensbeziehungen beteiligt sein.

Ein Beispiel ist in Abbildung 2.10 unter Verwendung der Vorschrift `per class` dargestellt. Der Aspekt `Rollenverwaltung` existiert zur Laufzeit mit zwei Exemplaren, jeweils eines pro Klasse. Die beiden Exemplare der Klasse `Web` verweisen jeweils auf dasselbe Aspektexemplar.

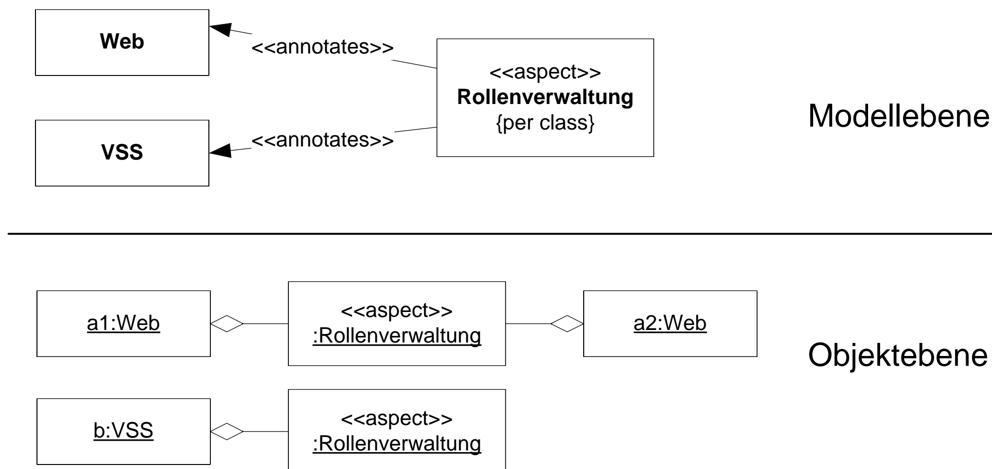


Abbildung 2.10: Die Annotationsbeziehung zur Laufzeit

```
              Advice  =  “advice“ Advicetyp Pointcut;
          Advicetyp   =  “before“ | “around“ | “after throwing“ | “after returning“ | “after“ |
                         “initialize“;
            Pointcut  =  Joinpointtyp [Metadatenliste] [Schnittstellenliste] [“.“]
                         [Methodenliste] “(“ Parameterliste “)“ [Rückgabetyp]
                         [Einschränkungsliste];
        Joinpointtyp  =  “create“ | “init“ | “call“ | “access“ | “destroy“ | “finalize“;
 Metadatenlisteliste  =  “[“ Bezeichnerliste “]“;
 Schnittstellenliste  =  Bezeichnerliste;
       Methodenliste  =  Bezeichnerliste;
      Parameterliste  =  [Parameter] {“,“ Parameter} [“,“] [“...“] ;
         Rückgabetyp  =  “:“ Parametertyp;
 Einschränkungsliste  =  “where“ GenerischerTyp “:“ Typliste [Einschränkungsliste];
            Typliste  =  Typ [“,“ Typliste];
     Bezeichnerliste  =  [“!“] Bezeichner {“,“ Bezeichner};
           Parameter  =  Bezeichner “:“ Typbezeichner;
         Parametertyp =  GenerischerTyp | Typ;
      GenerischerTyp  =  “<“Bezeichner“>“;
                 Typ  =  Bezeichner
          Bezeichner     siehe Text
```

Abbildung 2.11: EBNF von Advice und Introduktion

2.7.4 Definition von Advices

Wie in Abbildung 2.8 dargestellt, werden die Advices in LOOM.UML im vierten Kasten eines Aspekts unter den Methoden definiert. Die EBNF für eine solche Definition ist in Abbildung 2.11 dargestellt. Das nicht definierte Nichtterminalsymbol `Bezeichner` entspricht wieder der üblichen Ersetzungsregel für einen Bezeichner, wie sie in der UML verwendet wird.

Advices definieren Quelltext, der an den von ihnen selektierten Joinpoints ausgeführt werden soll. Die Selektion erfolgt über Prädikate, genauer *Pointcuts*. Ein Pointcut ist ein Prädikat, das aus einer Menge von Joinpoints eine Teilmenge Joinpoints selektiert, an denen der Advice ausgeführt werden soll. Diese Pointcuts werden in Abbildung 2.11 durch das Nichtterminalsymbol `Pointcut` symbolisiert.

In einem Pointcut werden die verschiedenen Joinpointattribute beschrieben, die ein Joinpoint haben soll oder auch nicht. Dabei handelt es sich um die Art der Schnittstellenmethode (`Joinpointtyp`), die Metadaten an der Schnittstellenmethode (`Metadatenlist`), den Name der Schnittstelle (`Schnittstellenliste`), den Name der Schnittstellenmethode (`Methodenliste`) sowie die an der Schnittstelle übergebenen Parameter (`Parameterliste`) inklusive des Rückgabetyps (`Rückgabetyptyp`) und einer Liste von Einschränkungen für die Parameter (`Parametereinschränkungen`).

Metadaten, Schnittstellenname und Schnittstellenmethode sind jeweils Aufzählungen von Bezeichnern, von denen jeweils einer mit dem entsprechenden Joinpointattribut übereinstim-

men muss, oder - wenn die Liste mit einem Ausrufezeichen begonnen wird - von denen keiner mit dem entsprechenden Joinpointattribut eine Übereinstimmung haben darf. Enthält die Liste für die Schnittstellenmethode die Bezeichner

```
GetExperimentStatus , GetExperiment
```

so wird - wenn alle anderen im Pointcut beschriebenen Attribute zutreffen - der Joinpoint für die Methode `GetExperimentStatus` und die Methode `GetExperiment` selektiert. Ist in der Liste für die Metadaten hingegen `!ÄndertZustand` aufgeführt, werden nur Joinpoints selektiert, in deren Metadaten kein `ÄndertZustand` vorkommt. Bezogen auf die objektorientierten Frameworks Java und .NET darf die zugehörige Methode keine gleichnamige Annotation (im Sinne von Java-Annotationen [Gosling u. a. 2005, S. 281 ff.] oder CLI-Attribute [Microsoft Corporation u. a. 2006, Teil I, S. 84 ff.]) enthalten.

Metadaten, Schnittstellenname und Schnittstellenmethode sind optional. Ist eine Liste leer, so wird das entsprechende Joinpointattribut nicht überprüft. Es muss jedoch in jedem Fall der Joinpointtyp und die Parameterliste angegeben werden.

Die Parameterliste beschreibt die an den Joinpoint übergebenen Parameter. Damit ein Joinpoint selektiert werden kann, muss die dort angegebene Liste auf die am Joinpoint übergebenen Parametern abgebildet werden können. Eine Abbildung ist grundsätzlich immer möglich, wenn

- die Anzahl der Parameter des Joinpoints mit der Anzahl der Parameter in der Parameterliste des Pointcuts übereinstimmt, und
- für alle Parameter in der Parameterliste des Pointcuts gilt, dass sie denselben Typen haben wie der am Joinpoint übergebene Parameter auf derselben Parameterposition.

Ein einfacher Pointcut wäre der folgende:

```
foo(bar:double):int
```

Dieser ähnelt einer Methodendeklaration und würde genau diejenigen Joinpoints selektieren, die zu einer Methode mit dem Namen `foo`, einem Parameter vom Typ `double` und dem Rückgabetypen `int` gehören. Da keine Schnittstellen angegeben wurden, kann der Punkt vor der Methode auch weggelassen werden. Ansonsten dient er zur Trennung von Schnittstellen- und Methodenliste.

Parameterlisten können neben konkreten Typen auch *generische Typen* enthalten. Generische Typen dienen als Platzhalter für einen einzelnen Parameter. Sie werden durch den Einschluss in spitze Klammern kenntlich gemacht. In diesem Fall ist der Typ des entsprechenden Parameters des Joinpoints nicht von Belang. Generische Typen können aber am Ende der Deklaration durch eine **where**-Klausel weiter eingeschränkt werden (entsprechend der Ersetzungsregel `Parametereinschränkungen`). In diese Einschränkungen werden ein oder mehrere Typen angegeben, denen der generische Typ mindestens entsprechen muss. Mindestens bedeutet, dass der Parameter am Joinpoint, der auf den generischen Typen abgebildet wird,

- dem Typen entspricht, der in der Einschränkung angegeben wurde,
- sich in den Typ in der Einschränkung konvertieren lässt, oder
- ein in der Einschränkung angegebenes Interface implementiert.

Während mehrere Interfaces als Einschränkung für einen generischen Typen erlaubt sind, darf pro generischem Typen nur ein konkreter Typ als Einschränkung benannt werden. Generischen Typen mit ausformulierten Einschränkungen bieten bei der Implementation der Advices die Möglichkeit, verschiedene Parametersignaturen mit einem Advice zu erfassen,

aber trotzdem typsicher auf den entsprechenden Parameter zuzugreifen. Der Parameter muss genau dem in der Einschränkung angegebenen Typen entsprechen und die angegebenen Interfaces implementieren.

Am Ende einer Parameterliste kann ein allgemeiner Platzhalter (**...**) für eine beliebige Anzahl von Parametern zum Einsatz kommen. In diesem Fall muss die Anzahl der Parameter am Joinpoint nicht mit der Anzahl in der Parameterliste übereinstimmen. Vielmehr können ab der Position des Platzhalters beliebig viele weitere Parameter am Joinpoint übergeben werden.

Der Rückgabetyp als spezieller Parameter kann ebenfalls mit einem generischen Typen beschrieben werden. Ein erweiterter Joinpoint ist beispielsweise der folgende:

```
call [!ÄndertZustand] IController.GetExperimentStatus,GetExperiment(p
    :<T>, ...):<U> where T:ExperimentBeschreibung
```

Dabei werden alle Joinpoints selektiert,

- die nicht mit dem CLI-Attribut `ÄndertZustand` annotiert wurden,
- die sich auf das Interface `IController` beziehen,
- deren Schnittstellenmethode entweder `GetExperimentStatus` oder `GetExperiment` heißt,
- die einen oder mehrere Parameter haben. Der erste Parameter muss dabei vom Typ `ExperimentBeschreibung` oder von einem abgeleiteten Typen sein,
- die einen beliebigen Rückgabewert haben.

Zu einer vollständigen Advicedeklaration gehört noch das Schlüsselwort **advice** und ein `Advicetyp`. Letzterer gibt an, wann der Advice ausgeführt werden soll. Im einfachsten Fall wird in der Advicedeklaration nur der Joinpointtyp beschrieben:

```
advice before call .(...)
```

Er selektiert folglich alle Joinpoints, die vom Aspekt überdeckt werden und den Aufruf einer Methode betreffen. In diesem Fall signalisiert der Punkt vor der Parameterliste, dass Schnittstellen- und Methodenliste leer sind.

Advices können auch als virtuell deklariert werden. Das geschieht, indem sie - wie in der UML üblich - kursiv geschrieben werden. In der Kindklasse wird ein Advice der Elternklasse überschrieben, genau dann, wenn die Pointcutdeklaration beider identisch ist.

2.7.5 Ausführung von Advices

Der in einem Advice definierte Quelltext wird exakt dann zur Ausführung gebracht, wenn der Kontrollfluss einen von ihm selektierten Joinpoint passiert. Genauer spezifiziert das der *Advicetyp* (`Advicetyp`) in Abbildung 2.11:

- **before**: Der Advice wird vor der Ausführung vor der zum Joinpoint gehörenden Methode zur Ausführung gebracht.
- **around**: Der Advice wird anstelle der zum Joinpoint gehörenden Methode zur Ausführung gebracht.
- **after**: der Advice wird nach der Ausführung der zum Joinpoint gehörenden Methode zur Ausführung gebracht,
- **after returning**: der Advice wird zur Ausführung gebracht, wenn die zum Joinpoint gehörende Methode durch eine **return**-Anweisung beendet wurde.

- **after throwing**: der Advice wird zur Ausführung gebracht, wenn die zum Joinpoint gehörenden Methode eine unbehandelte Ausnahme geworfen hat.

Überdecken mehrere Advices dieselbe Methode, so ist deren *Rangordnung* entscheidend für die Reihenfolge der Ausführung. Der Gesamtrang ergibt sich aus dem Rang der Annotation und dem Rang im Aspekt. Der Rang der Annotation ist dabei dem Rang im Aspekt übergeordnet.

Für den Rang der Annotation gilt folgende Reihenfolge, beginnend mit dem höchsten Rang:

- alle Annotationen einer Methode,
- alle Annotationen einer Klasse oder eines Interfaces,
- alle Annotationen einer Komponente (eines Paketes).

Haben nach dieser Definition zwei Annotationen denselben Rang, so ist die Rangordnung nicht definiert.

Der Rang im Aspekt ist festgelegt durch den Advicetypen und die Reihenfolge, in der die Advices im vierten Abschnitt des Aspekts definiert wurden. Alle **before**-Advices haben einen höheren Rang als **around**-Advices und diese wiederum einen höheren Rang als die übrigen Advices. Bei Advices gleichen Typs gilt, dass der erste Advice, der bei der Aspektdeklaration zuerst vorkommt, den höchsten, der letzte den niedrigsten Rang hat.

Für die Abarbeitung der Advices hat die Rangfolge folgende Implikationen:

- Die **before**- und **around**- Advices werden in der Reihenfolge ihrer Rangordnung, beginnend mit dem höchsten Gesamtrang, abgewickelt.
- Ein **before**-Advice höherer Rangordnung kann die Ausführung der Methode und der Advices niedrigerer Rangordnung durch das Werfen einer Ausnahme verhindern.
- Ein **around**-Advice kann die Ausführung der Methode und der Advices niedrigerer Rangordnung explizit veranlassen. Macht er das nicht, kommen die Methode und alle Advices mit einem niedrigeren Rang nicht zur Ausführung.
- Alle **after**-Advices werden in aufsteigender Reihenfolge ihrer Rangordnung ausgeführt. Wenn die Methode zur Ausführung kommt, wird bei dem **after**-Advice mit dem niedrigsten Rang begonnen. Wird die Abarbeitung der Advices durch einen **before** oder **around**-Advice unterbrochen, wird die Abwicklung bei denjenigen **after**-Advices fortgesetzt, die aus derselben Annotation stammen wie der unterbrechende Advice.

Für das in Abbildung 2.12 dargestellte Beispiel würden alle fünf Advices auf die Methode `foo` der Klasse `A` wirken. Nach den zuvor definierten Regeln ergäbe sich folgende Ausführungsreihenfolge:

```
Bar::advice before call .(...)
Baz::advice before call .(...)
Foo::advice before call .(...)
A::foo()
Baz::advice after call .(...)
Baz::advice after call foo(...)
```

Bisher wurde nicht der spezielle Advicetyp `initialize` besprochen. Bei Advices dieses Typs handelt es sich um so genannte *Adviceinitialisierer*, die mit Konstruktoren von Klassen vergleichbar sind. Adviceinitialisierer sind ein spezielles LOOM-Konzept und werden ausführlich in Abschnitt 2.8.2 vorgestellt. An dieser Stelle ist wichtig zu wissen, dass die Adviceinitialisierer immer dann aufgerufen werden, wenn von der zum Joinpoint gehörenden Klasse ein Exemplar gebildet wird. Das ist diejenige Klasse, in der die zum Joinpoint gehörende Methode definiert wurde.

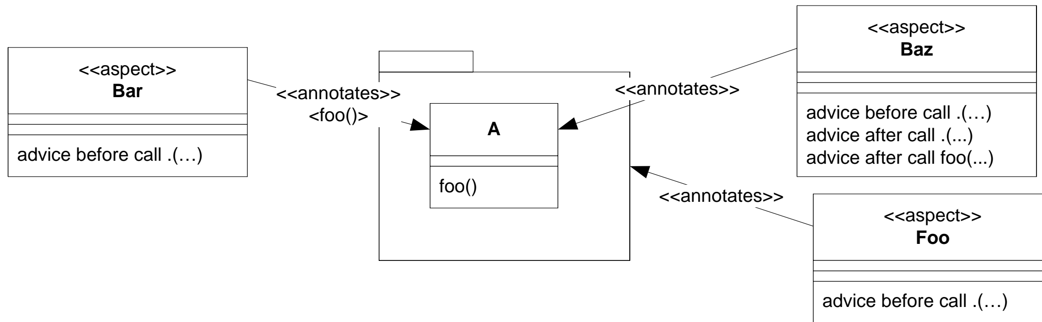


Abbildung 2.12: Beispiel zur Demonstration der Ausführungsreihenfolge von Advices

2.7.6 Definition von Introduktionen

Wie die Advices, werden die Introduktionen auch im vierten Abschnitt des Aspekts definiert. Die Introduktionen werden wie normale Methoden deklariert, allerdings wird der Deklaration das Schlüsselwort **introduce** vorangestellt. Soll mit der Introduktion ein Interface eingeführt werden, wird dieses dem Methodennamen vorangestellt. Voraussetzung ist, dass das Interface bereits bekannt ist und im Aspekt eine vollständige Introduktion aller Interfacemethoden erfolgt. Die Introduktion:

```
introduce IBenutzerInfo.SetBenutzer(b:string)
```

führt die Methode `SetBenutzer` aus der Interfacedefinition `IBenutzerInfo` in alle vom Aspekt überdeckten Klassen ein.

Wenn eine Introduktion eine neue Methode in eine Klasse einführt, ist es möglich, dass eine Methode mit derselben Signatur bereits vorhanden ist. Eine Lösung wäre, die existierende Implementation zu überschreiben, wie es in AspectJ beispielsweise der Fall ist [vgl. Laddad 2003, S.121 ff.].

Eine solche Vorgehensweise bringt einige Probleme: Was passiert, wenn mehrere Aspekte Methoden mit derselben Signatur einfügen? In AspektJ entscheidet die Rangordnung über den „Gewinner“; die ursprüngliche Implementation ist nicht mehr erreichbar.

Die in LOOM propagierte Lösung lässt dem Programmierer etwas mehr Spielraum. Ist eine Introduktion virtuell deklariert, überschreibt die Introduktion eventuell vorhandene Implementierungen. Es ist jedoch möglich, über den Ausführungskontext (Abschnitt 2.8.1) die überschriebene Implementierung zu erreichen. Das entspricht in etwa den **super**-Aufrufen in Java oder den **base**-Aufrufen in C#.

Ist eine Introduktion nicht als virtuell markiert, so ist sie exklusiv. Das bedeutet, dass ein Überschreiben nicht erlaubt ist. Versucht eine solche Introduktion eine bereits vorhandene Methode zu überschreiben, führt das zu einem Fehler bei der Erstellung der Anwendung.

Ein Problem, das bei Introduktionen entstehen kann, ist die richtige Zuordnung der Aspektexemplare. In Abbildung 2.13 soll der Aspekt `Rollenverwaltung` überprüfen, ob der gerade angemeldete Benutzer die notwendigen Rechte hat, die mit dem Aspekt annotierten Methoden abzuwickeln. Der Aspekt speichert in `rolle` den bei der Annotation übergebenen Wert und vergleicht ihn mit der Rolle des in `benutzer` gespeicherten Benutzer. Der aktuelle Benutzer soll jeweils über die Introduktion `SetBenutzer` gesetzt werden.

Die Exemplarbildungseigenschaft **per annotation** sorgt dafür, dass zur Laufzeit für jede Methode der Klasse `Experimentsteuerung` ein Aspektexemplar erzeugt wird. Das ist so gewollt, denn die Methoden setzen unterschiedliche Rollen voraus, die jeweils im Aspektexemplar unter `rolle` gespeichert werden. Wird nun über das Exemplar der `Experimentsteuerung` die Methode `SetBenutzer` aufgerufen, um den aktuellen Benutzer zu setzen, ist nicht klar, auf welchem Aspektexemplar die Methode abgewickelt werden soll.

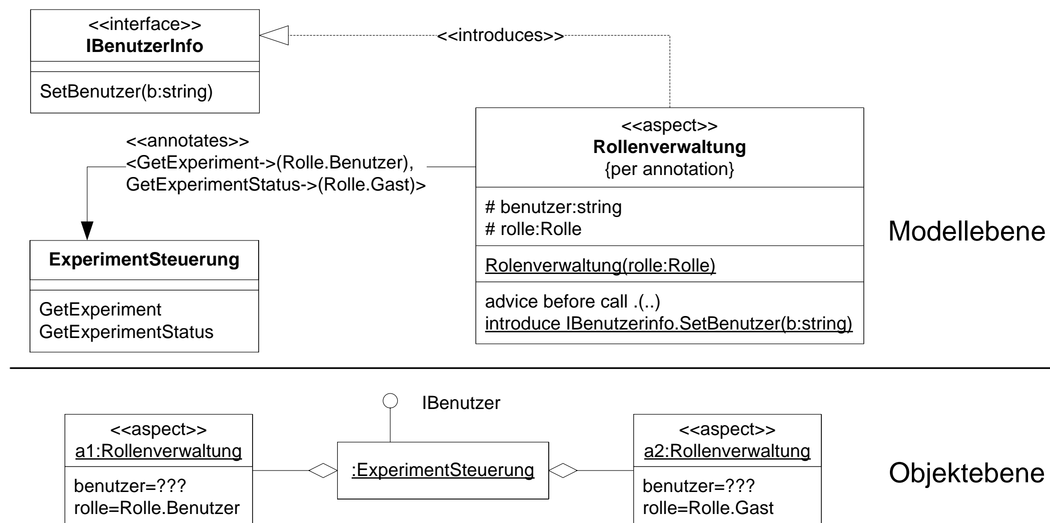


Abbildung 2.13: Introduktionen und Objekte

Eine solche Introduktion führt also aufgrund der Mehrdeutigkeit zu einem Fehler bei der Erstellung der Anwendung. Ein Ausweg besteht darin, die Introduktionen als statisch (exemplarlos) zu definieren. Eine statische Introduktion wird in LooM.UML wie in Abbildung 2.13 unterstrichen, dargestellt. An der Einführung in der überdeckten Klasse ändert sich dadurch nichts; wie statische Methoden haben solche Introduktionen keinen Zugriff auf nichtstatische Attribute des Aspekts.

Für das Attribut `benutzer` bedeutet das, dieses auch statisch zu deklarieren. Eine andere Lösung wäre, das Attribut an die Klasse `ExperimentSteuerung` zu binden. Das wird über die Joinpointvariablen erreicht, die im folgenden Abschnitt erklärt werden.

2.7.7 Joinpoint-Variablen

Joinpoint-Variablen sind ein spezielles LooM-Konzept, um auf Attribute überdeckter Klassen mit einem Aspekt zugreifen bzw. dort neue Attribute definieren zu können. Sie gehen dabei über die Möglichkeiten der *Inter-Type-Deklarationen* verschiedener AOP-Lösungen [vgl. z. B. JBoss AOP 2008; Kiczales u. a. 2001] hinaus, denn

- mit ihnen kann auf bereits vorhandene Attribute zugegriffen und
- mit ihnen können neue Attribute direkt auf einem Joinpoint definiert werden.

Während die Speicherung von normalen Attributen im Aspektexemplar selbst erfolgt, werden Joinpointvariable an anderer Stelle gespeichert. Der Ort der Speicherung ergibt sich aus dem Joinpoint. Das führt dazu, dass eine Joinpointvariable in den meisten Fällen auf mehr als eine Speicherstelle referenziert. Der Zeitpunkt des Zugriffes und die Art der Joinpointvariablen bestimmt die konkrete Speicherstelle.

Es existieren zwei verschiedene Arten von Joinpointvariablen. Sie werden durch die Stereotypen «`target`» und «`joinpoint`» unterschieden. Als zusätzliche Modifizierer sind **virtual** (kursiv geschrieben), **override** (als weitere Attributeigenschaft) und **static** (unterstrichen) erlaubt.

Eine «`target`»-Joinpointvariable kann damit neue Attribute in eine vom Aspekt überdeckte Klasse einführen oder auf bestehende Attribute referenzieren:

- Hat die Joinpointvariable die Sichtbarkeit **private**, wird sie stets als neues Attribut in die Klasse eingeführt,

- andernfalls wird sie als neues Attribut in die Klasse eingeführt, wenn ein gleichnamiges Attribut noch nicht vorhanden ist,
- ansonsten wird die Joinpointvariable auf das in der Klasse vorhandene Attribut mit gleicher Sichtbarkeit abgebildet.

Dabei gelten folgende Regeln:

1. Hat die Joinpointvariable den Modifizierer **override**, muss ein gleichnamiges Attribut bereits vorhanden sein.
2. Hat die Joinpointvariable keinen der Modifizierer **override** und **virtual**, so darf kein gleichnamiges Attribut vorhanden sein.
3. Ist ein gleichnamiges Attribut vorhanden, müssen sowohl die Sichtbarkeit und der Attributtyp mit der Joinpointvariable identisch sein. Hat einer den Modifizierer **static** in seiner Definition, muss dieser Modifizierer in der Definition des anderen auch vorkommen.
4. Joinpointvariablen mit dem Modufizierer **private** dürfen nicht gleichzeitig die Modifizierer **virtual** oder **override** haben. Für solche Joinpointvariablen gelten die Regeln 1 bis 3 nicht.

Wie viele Speicherstellen es für eine `«target»`-Joinpointvariable gibt, hängt davon ab, ob der **static**-Modifizierer verwendet wurde. Ist die Joinpointvariable statisch, existiert pro überdeckte Klasse genau eine Speicherstelle, andernfalls existiert jeweils pro Exemplar der überdeckten Klassen eine Speicherstelle. Wird in der Implementation des Aspekts auf die Joinpointvariable zugegriffen, entscheidet der Kontext, in dem die Implementation ausgeführt wird, welche Speicherstelle gewählt wird. Es ist dann immer diejenige Speicherstelle des Exemplars (oder der Klasse), über dessen Joinpoint der Aspekt aktiviert wurde.

In Abbildung 2.14 wurde im Aspekt `Rollenverwaltung` die bereits bekannte Joinpointvariable `benutzer` als `«target»`-Joinpointvariable deklariert. Aufgrund der Überdeckung wird sie in die Klasse `ExperimentSteuerung` eingeführt. In der zusätzlichen Klasse `AlternativeExperimentSteuerung` ist dieses Attribut bereits vorhanden. Da `benutzer` im Aspekt als virtuell deklariert wurde, wird es auf das bestehende `benutzer` in `AlternativeExperimentSteuerung` abgebildet. Wäre die Joinpointvariable nicht virtuell, würde es beim Erstellen der Anwendung zu einem Fehler kommen. In jedem Exemplar von `ExperimentSteuerung` und `AlternativeExperimentSteuerung` existiert nun eine Speicherstelle für die Joinpointvariable `benutzer`.

Wird in dem Advice auf `benutzer` zugegriffen, so hängt es davon ab, in welchem Kontext der Advice gerade abgewickelt wird. Wurde er über eine Methode der Klasse `ExperimentSteuerung` aktiv, ist es die Speicherstelle des entsprechenden `ExperimentSteuerung`-Exemplars. Wurde der Advice jedoch über eine Methode der Klasse `AlternativeExperimentSteuerung` aktiv, ist es die Speicherstelle des bereits vorhandenen `benutzer`-Attributes des `AlternativeExperimentSteuerung`-Exemplars.

Während `«target»`-Joinpointvariable sich auf Attribute einer Klasse beziehen, können Joinpointvariable mit dem Stereotypen `«joinpoint»` Attribute direkt auf dem Joinpoint einführen. Das entspricht in etwa einer lokalen Variablen in einer Methode. Für diese Joinpointvariablen gelten dieselben Regeln wie für die `«target»`-Joinpointvariablen.

In Abbildung 2.14 ist die Variable `letzterZugriff` mit dem Stereotypen `«joinpoint»` deklariert. Wird die Methode `GetExperiment` der Klasse `ExperimentSteuerung` aufgerufen, wird dem Advice bei einem Zugriff auf `letzerZugriff` eine andere Speicherstelle referenziert, als beim Zugriff über `GetExperimentStatus`.

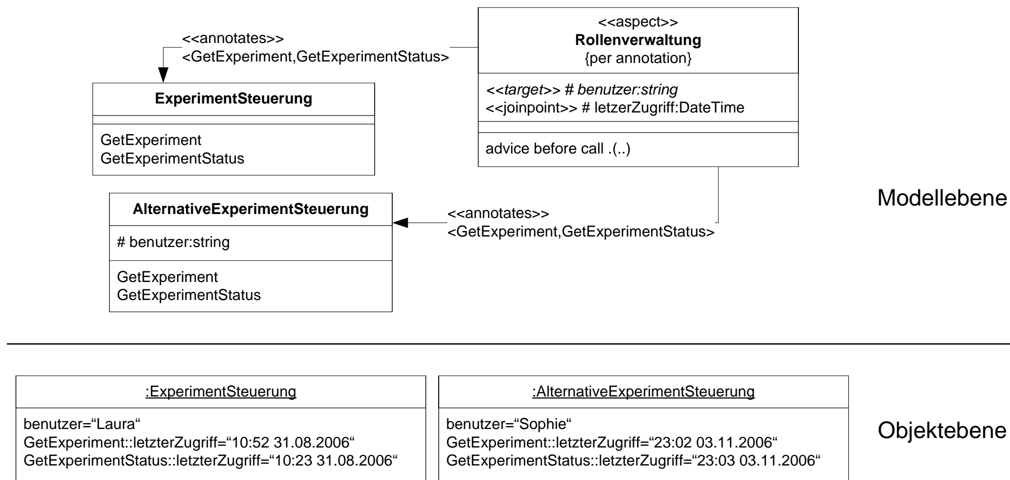


Abbildung 2.14: Joinpoint-Variablen

Joinpointvariablen erlauben also die Speicherung von Daten im Kontext von Klassen und deren Exemplaren sowie von Methoden. Sie erlauben es weiterhin, auf bereits vorhandene Attribute zuzugreifen.

2.8 Weitere LOOM-Konzepte

Im Folgenden sollen LOOM-Konzepte erklärt werden, die direkt die Implementierung von Aspekten betreffen. Es geht insbesondere darum, wie Aspekte mit den von ihnen überdeckten Klassen kommunizieren können, ohne dass sie deren Schnittstellen kennen.

2.8.1 Der Ausführungskontext von Advices und Introduktionen

In objektorientierten Sprachen ist der Kontext, in dem eine Methode (mit Ausnahme statischer Methoden) abgewickelt wird, ein Exemplar der Klasse, in der die Methode definiert wird. Der Programmierer kann auf diesen Kontext je nach Programmiersprache über eine Metavariable (z. B. **this** oder **self**) auf dieses Objekt zugreifen.

Innerhalb eines Aspekt verweist **this** auf das Aspektexemplar, in deren Kontext der Advice, die Introduktion oder die Methode des Aspekts gerade ausgeführt wird. Es gibt jedoch noch einen weiteren Kontext, in dem der Aspekt ausgeführt wird. Das ist das Exemplar derjenigen Klasse, die von dem Aspekt überdeckt und dessen Multiplizität über die **per**-Eigenschaft festgelegt wurde. Die Metavariable, die Zugriff auf dieses Exemplar erlaubt, ist **target**. Da ein Aspekt potentiell viele Klassen überdecken kann, ist der statische Typ von **target** immer der Basistyp aller Klassen, nämlich `object`.

Advices werden stets an den von ihnen selektierten Joinpoints aktiv. Das ist eine weitere Kontextinformation, auf die ein Programmierer über die Metavariable **joinpoint** zugreifen kann. Der Joinpoint wird in LOOM somit zu einem Objekt erster Ordnung und kann an Variablen gebunden werden. Dieses Objekt entsteht dynamisch immer dann, wenn ein Joinpoint passiert wird. Der Datentyp erster Ordnung ist eine spezialisierte Ausprägung von der Klasse `Context`[1]. Für jeden Joinpoint existiert eine eigene Ausprägung.

Die Laufzeitrepräsentation eines Joinpoints (ein `Context`-Objekt) ist letztendlich das, was die Objektinteraktion an diesem Joinpoint ausmacht. Konkret ist das die gesendete Nachricht einschließlich des Empfängerobjektes. Über das `Context`-Objekt lassen sich auch

[1] Die Bezeichnung `Context` mag hier etwas ungewöhnlich sein, hat jedoch historische Ursachen. Tatsächlich gab es in den ersten LOOM-Versionen nämlich nur eine zusätzliche Kontextinformation.

Metainformationen über die Schnittstelle ermitteln, wie zum Beispiel die Klassen- oder der Methodennamen.

Im Kontextobjekt ist neben der Nachricht auch der Fortschritt der Adviceabarbeitung gespeichert. Mit der Methode `Call` der `Context`-Klasse kann explizit (z. B. in **around**-Advices) die Advice-Abarbeitung fortgesetzt werden.

Introduktionen, die als virtuell deklariert wurden, können auch auf den **joinpoint** zugreifen. Existieren mehrere Aspekte, die eine Methode gleicher Signatur in dieselbe Klasse einführen, wird wie bei den Advices entsprechend der Rangordnung mit jedem **joinpoint.Call** die nächste Introduktion aufgerufen, bis die eventuell vorhandene originale Implementation erreicht ist. Ist eine solche Implementation nicht vorhanden, ist die Metamethode **Call** eine leere Anweisung.

2.8.2 Advice-Initialisierer

Advices vom Typ `initialize` wurden bisher noch nicht betrachtet. Während das Konzept der **before-**, **after-** und **around-** Advices bereits relativ alt ist [Teitelman 1966], führt LOOM mit den Advice-Initialisierern ein neues Konzept ein.

In der objektorientierten Welt existieren Konstruktoren zur Initialisierung von Objekten. Sobald ein Speicherplatz für das neue Exemplar reserviert worden ist, wird der entsprechende Konstruktor der zugehörigen Klasse gerufen. Der Konstruktor dient dazu, das eigene Objekt (**this**) zu initialisieren.

In LOOM wurde der **joinpoint** als zusätzlicher Kontext eingeführt. Advice-Initialisierer sind ein Pendant zu den Konstruktoren. Der Programmierer kann einen Advice-Initialisierer definieren, der genau dann ausgeführt wird, wenn für den selektierten Joinpoint eine Laufzeitrepräsentation entsteht. Das ist der Fall, wenn neue Exemplare gebildet und ihre Schnittstellen verfügbar werden.

Advice-Initialisierer werden wie normale Advices definiert und durch das Schlüsselwort **initialize** gekennzeichnet:

```
advice initialize call .(...)
```

Dieser Advice-Initialisierer wird für jede Methode einer vom Aspekt überdeckten Klasse genau einmal aufgerufen, wenn ein neues Exemplar der Klasse gebildet wird. Dabei erfolgt der Aufruf jeweils im Kontext des Joinpoints der Methode. Statt **call** könnten auch andere Joinpointtypen verwendet werden.

Ein Programmierer kann in diesen Initialisierern den Aufrufkontext des Joinpoints abspeichern, um ihn zu einem späteren Zeitpunkt zu verwenden. Er kann so beispielsweise eine Schnittstelleninteraktion auslösen, ohne die Schnittstelle zu kennen. Das verwendet z. B. der *Ereignisabonnenten* in Abschnitt 4.9. Ein Initialisierer eignet sich auch, um lokale Daten, die im Zusammenhang mit den Joinpoints stehen, zu belegen. Ein Beispiel hierfür ist die Initialisierung der Vor- und Nachbedingungen von Methoden, wie sie in den *Design-By-Contract-Aspekten* in Abschnitt 5.2 verwendet werden.

2.8.3 Nachrichtenmanipulation

Im Geltungsbereich eines Aspekts hat der Programmierer über die Metavariable **joinpoint** Zugriff auf den Ausführungskontext des Joinpoints. Dieser repräsentiert unter anderem auch die Nachricht an die Schnittstelle, die zur Aktivierung des Joinpoints führte. Eine solche Nachricht hat die Form:

$$target.method_{id}\,(arg_1, arg_2, \ldots arg_n)$$

	Parameter ändern	**Adressat ändern**	**Methode ändern**	**neuer JP**
Call	ja	nein	nein	nein
CallOn	ja	ja	nein	nein
ReCall	ja	nein	ja	ja
ReCallOn	ja	ja	ja	ja

Abbildung 2.15: Metamethoden zur Nachrichtenmanipulation

Eine Besonderheit von LOOM ist, dass diese Nachricht in ihrer Gesamtheit in einem definierten Rahmen verändert werden kann. Die Veränderung von Nachrichten ist eine häufig genutzte Strategie bei der Implementation der Aspekte, um Aktionen auf unbekannten Schnittstellen ausführen zu können. Ungefähr ein Drittel der in den folgenden Kapiteln vorgestellten Aspekte benutzt dieses LOOM-Konzept. Folgende Teile der Nachricht können verändert werden:

- Ein Parameterwert (arg_m) kann durch den Wert (arg'_m) ersetzt werden, wenn
  - der dynamische Typ von arg'_m in den Parametertypen des m-ten Parameters der Schnittstellendefinition von $method_{id}$ konvertiert werden kann oder
  - arg'_m mit dem wert `null` belegt ist und der Typ des m-ten Parameters der Schnittstellendefinition die Belegung mit Nullwerten erlaubt.
- Der Empfänger der Nachricht ($target$) kann durch einen anderen Empfänger ausgetauscht werden, wenn dieser die Schnittselle hat, auf der $method_{id}$ definiert ist.
- Die Methode $method_{id}$ kann durch eine gleichnamige Überladung $method'_{id}$ und neuen Parametern $arg'_1, arg'_2, \ldots arg'_k$ ausgetauscht werden, wobei sich jeder Parameterwert in den Parametertypen der Schnittstellendefinition von $method'_{id}$ konvertieren lassen muss.

Um eine Nachricht zu manipulieren, muss die für den Nachrichtenempfang vorgesehene Methode durch einen **around**-Advice überdeckt worden sein. Über die im Ausführungskontext definierten Metamethoden **Call**, **CallOn**, **ReCall** und **ReCallOn** kann nun - wie in Abbildung 2.15 dargestellt - die Nachricht verändert werden.

Wird der Adressat oder die Methode geändert, verändert sich automatisch der Ausführungskontext des Joinpoints oder sogar der Joinpoint selbst. Das ändert die Abwicklung der Advices auf dem ursprünglichen Joinpoint: Advices mit einem niedrigerem Rang als dem nachrichtenverändernden **around**-Advice werden nicht mehr ausgeführt. Vielmehr wird der neuen Joinpoint mit einem neuen Joinpointkontext passiert und alle Advices, die diesen Joinpoint selektieren, und die zugehörige Methode wird entsprechend der Regeln in Abschnitt 2.7.5 abgewickelt.

Veränderungen an der Nachricht müssen so ausfallen, dass die zum Joinpoint gehörende Methode auch weiterhin auf die veränderte Nachricht anwendbar ist. Während die Zuordnung der Metamethoden **Call** und **CallOn** eindeutig ist und maximal durch eine Verletzung der genannten Bedingungen zu einer Ausnahme führen kann, ist das bei **ReCall** und **ReCallOn** nicht mehr gewährleistet. Vielmehr muss hier während der Laufzeit zu den übergebenen Argumenten die passende Überladung gefunden werden - also eine Überladung, deren statische Parametertypen am weitesten spezialisiert sind, und die sich auf die Nachricht anwenden lässt. Dieses Verfahren wird *Multidispatch* [Clifton u. a. 2000; Shalit 1996; Steele 1990] bezeichnet.

Von einem *symetrischen Multidispatch* spricht man, wenn alle Argumente für das Auffinden der richtigen Methode gleichberechtigt betrachtet werden. Demgegenüber benutzt *asymetrischen Multidispatch* typischerweise eine lexikografische Ordnung, die u.a. durch die Reihenfolge der Methodendefinitionen definiert ist.

Der in LOOM zugrunde liegende Algorithmus, der bei zwei Methoden entscheidet, ob eine der beiden spezieller ist als die andere, wird in Abbildung 2.16 dargestellt. Wenn $method_1$ spezieller ist als $method_2$ liefert er einen Wert kleiner Null, ist $method_2$ spezieller als $method_1$, liefert er einen Wert größer Null. Wenn beide Methoden nicht in Relation zueinander stehen, liefert er Null als Ergebnis. Für den Algorithmus gelten folgende Randbedingungen:

- $method_x.parameters$ ist ein Feld mit den statischen Argumenttypen der Methode $method_x$, wobei der Feldindex der Position des Argumentes entspricht.
- Die Operatoren ($\lhd, \rhd$) treffen eine Aussage darüber, dass ein Argumenttyp allgemeiner, bzw. spezieller als ein anderer ist.
- $ptype_x.vararg$ ist ein Wahrheitswert, der angibt, ob der statische Argumenttyp $ptype_x$ eine variable Parameterliste definiert. Jedes Argument ist dabei mindestens vom Typ $ptype_x$.
- $ptype_x.generic$ ist ein Wahrheitswert, der angibt, ob der statische Argumenttyp $ptype_x$ ein generischer Parameter ist.
- $dummytype_1$ und $dummytype_2$ dienen als Markierung und sind Typen, die nie als Parametertypen vorkommen.

Der Algorithmus unterstützt damit auch Konzepte, wie generische Parameter und variable Argumentlisten, wie sie in der CLI [Microsoft Corporation u. a. 2006] und in Java [Gosling u. a. 2005] vorhanden sind.

Er wird angewendet, indem diejenigen Überladungen der ursprünglichen Methode inklusive der ursprünglichen Methode ermittelt werden, auf denen die neuen Parameter ausführbar sind. Aus dieser Menge wird nach dem Algorithmus in Abbildung 2.16 die speziellste Methode gewählt. Der Kontrollfluss wird an den Joinpoint dieser Methode umgelenkt.

Zu einem Fehler kann es in zwei Fällen kommen: Es kann keine Methode gefunden werden, die auf die neuen Parameter anwendbar ist, oder es gibt keine speziellste Methode. Letzteres ist der Fall, wenn der Algorithmus nur eine Halbordnung liefert, da nicht jede Methode mit jeder in Relation steht. In beiden Fällen wird von der LOOM.NET-Laufzeitumgebung eine Ausnahme erzeugt.

2.9 Zusammenfassung

In diesem Kapitel wurden neue Konzepte zur Modularisierung besprochen. Es wurde die Konzepte der objektorientierten Programmiersprachen noch einmal diskutiert. In objektorientierten Sprachen kann der Programmierer seinen Quelltext vereinfacht in Methoden, Klassen und Komponenten gruppieren. Je nach Sprache existieren hier noch weitere Ausprägungen. Diese Gruppierungseinheiten werden zum Abstecken von Modulgrenzen verwendet. Komponenten, eine Menge von Klassen - oder in seltenen Fällen auch einzelne Methoden - können unabhängig voneinander entwickelt werden. Durch die Komposition der einzelnen Teile entsteht die fertige Software.

Die objektorientierten Formalismen rechen jedoch nicht aus, um eine eindeutige Zuordnung von Anforderungen auf Module zu gewährleisten. Vielmehr kommt es zu Streuung und Durchsetzung des Quelltextes der einzelnen Module. Ein Modul implementiert dann verschiedene Anforderungen, andere Anforderungen lassen sich nicht auf ein einzelnes Modul abbilden. Das behindert natürlich den Prozess der Softwareentwicklung.

Mit LOOM wurde in diesem Kapitel ein neues Programmierparadigma vorgestellt, mit dem sich genau diese Probleme vermeiden lassen. Als neues Modulkonzept wurden die LOOM-Aspekte mit ihren Advices, Introduktionen und Joinpointvariablen vorgestellt. Außerdem wurde die Annotationsrelation als neues Mittel der Komposition von Modulen erläutert.

```
function CompareMethods(method_1, method_2)
    iPos ← 0
    repeat
        ptype_1 ← dummytype_1
        ptype_1 ← dummytype_2
        while ⟨ (iPos < method_1.parameters.length) ∧
                (iPos < method_2.parameters.length) ∧
                (ptype_1 = ptype_2) ⟩ do
            ptype_1 ← method_1.parameters[iPos]
            ptype_2 ← method_2.parameters[iPos]
            iPos ← iPos + 1
        end while
        if ptype_1.vararg then
            return −1
        end if
        if ptype_2.vararg then
            return 1
        end if
        if ptype_1 ◁ ptype_2 then
            return −1
        end if
        if ptype_1 ▷ ptype_2 then
            return 1
        end if
        if ptype_1.generic ∧ ¬ptype_2.generic then
            return −1
        end if
        if ptype_2.generic ∧ ¬ptype_1.generic then
            return 1
        end if
        diff ← method_1.parameters.length − method_2.parameters.length
    until (diff ≠ 0) ∧ (iPos ≥ method_1.parameters.length)
    return diff
end function
```

Abbildung 2.16: Entscheidungsalgorithmus für das Multidispatch

Schnittstellen spielen bei der Definition von Modulen eine wichtige Rolle. Sie sind der Vertrag, an den sich jeder Beteiligte halten muss. Daher beziehen sich alle LOOM-Konzepte stets auf Schnittstellen und nicht auf Implementationen. Verschiedene in diesem Kapitel vorgestellte Konzepte erlauben es dem Aspektprogrammierer, auch gegen anonyme Schnittstellen zu programmieren. Mit dem Konzept der Joinpointattribute beschreibt er lediglich, welche Eigenschaften eine Schnittstellenmethoden haben muss, um zu ihm kompatibel sein, damit eine Bindung erfolgen kann. Der Nutzer des Aspekts kann wiederum durch die Konzepte der Annotation und Überdeckung steuern, auf welchen Teilen seines Moduls ein Aspekt wirken darf.

Ein weiteres Konzept ist das der Nachrichtenmanipulation an den Joinpoints. Auch das stellt für den Aspektprogrammierer ein Mittel dar, um Aktionen auf ihm unbekannten Schnittstellen ausführen zu können. Innerhalb eines Advices kann über den Ausführungskontext - der den Schnittstellenaufruf an einem Joinpoint als Objekt erster Ordnung darstellt - die Nachricht manipuliert, Parameterwerte und der Empfänger verändert und eine andere Überladung gewählt werden. Letzteres wird durch einen Multidispatch-Algorithmus erreicht.

Das neue Konzept der Advice-Initialisierer erlaubt es, Joinpoint-spezifische Datenstrukturen explizit zu initialisieren. Sie sind das Pendant zu den aus der objektorientierten Programmierung bekannten Konstruktoren. Immer wenn ein neues Exemplar gebildet wurde, das über die vom Advice selektierten Joinpoints erreichbar ist, wird der Advice-Initialisierer im Kontext des Exemplars und des Joinpoints aufgerufen.

Die neuen Konzepte können durch die in diesem Kapitel vorgestellte LOOM.UML, eine Erweiterung der UML2, modelliert werden. Ein Aspekt wird als Stereotyp einer Klasse dargestellt, die Annotationsrelation als Stereotyp einer gerichteten Relation. Für Advices und Introduktionen wurde eine spezielle Sprachsyntax vorgestellt.

3 Zwei LOOM-Aspektweber

Im Kapitel 2 wurden neue Konzepte vorgestellt, mit denen überschneidende Belange im Quelltext vermieden werden können. Im Folgenden soll es darum gehen, diese Konzepte für neue Werkzeuge anwendbar zu machen.

Standardausführungsumgebungen kennen die Konzepte wie Aspekt und Annotation nicht. Eine neue Ausführungsumgebung zu entwickeln, ist zwar möglich, aber schwierig. Das liegt zum einen am hohen Entwicklungsaufwand, zum anderen aber auch an den hohen Markteintrittsbarrieren. Für eine breite Akzeptanz sollte eine solche Ausführungsumgebung auf dem Standard basieren, der von einer bekannten Standardisierungsorganisation akzeptiert wurde.

Die hier vorgestellten Implementierungen gehen einen anderen Weg. Mit ihnen können Aspekte unter der Verwendung eines bestehenden Standards für objektorientierte Sprachen, der *Common Language Infrastructure* (CLI) [Microsoft Corporation u. a. 2006], definiert werden.

Die Definition von Aspekten ist allerdings nur der Anfang. Die Ausführungsumgebung muss das vom Programmierer durch Aspektannotation und Selektion von Joinpoints definierte Verhalten nach den in Kapitel 2 beschriebenen Konzepten auch umsetzen. Da die Ausführungsumgebung jedoch nicht angepasst werden soll, ist hier ein weiterer Schritt notwendig - die Abbildung der aspektorientierten Konzepte auf die objektorientierten. Das geschieht durch einen sogenannten *Aspektweber*.

Es gibt verschiedene Ansätze, wann diese Abbildung passieren soll: Der Quelltext kann in ein objektorientiertes Pendant bzw. in die CIL transformiert bzw. kompiliert werden oder die binären Komponenten werden nach der Kompilierung angepasst. Schließlich kann die Abbildung während der Laufzeit durch Manipulation sogenannter *Metadaten* erfolgen.

Für die letzten beiden Varianten existieren vom Autor entwickelte Werkzeuge, die in diesem Kapitel vorgestellt werden. Die Implementationen können aufgrund ihres inzwischen erheblichen Umfangs - immerhin über 35.000 Zeilen Quelltext - nur ansatzweise vorgestellt werden.

3.1 Die Common Language Infrastructure

Die *Common Language Infrastructure* CLI [Microsoft Corporation u. a. 2006] spezifiziert eine Ausführungsumgebung für sprach- und plattformneutrale Anwendungen. Die CLI selbst ist eine Spezifikation, die von der *International Organization for Standardization* (ISO) und *Ecma International* (ECMA) standardisiert wurde. Hinter der CLI stehen 16 verschiedene Firmen und Organisationen und - es gibt verschiedene Implementierungen des Standards. Die bekannteste Implementierung dürfte dabei Microsoft.NET [Microsoft Corporation 2003b, 2008b] sein. Daneben existieren Umsetzungen, wie das .NET *Compact Framework* für portable Geräte und die Spielekonsole *XBox360* [Microsoft Corporation 2007b; Wigley und Wheelwright 2003], *Net60*, eine Version für Symbian-Basierende Geräte [red FIVE labs 2008], *Mono* als quelloffene Variante für Linux/Unix Betriebssysteme [de Icaza und Andere 2008] sowie *ROTOR*, eine CLI Referenzimplementierung für FreeBSD, Windows und MacOS [Microsoft Corporation 2006; Stutz u. a. 2003].

Der CLI-Standard besteht im Wesentlichen aus der Beschreibung folgender Teile:

- dem gemeinsamen Typsystem (*Common Type System*, CTS) für CLI-kompatible Programmiersprachen,
- den *Metadaten*, um die Struktur von Komponenten unabhängig von der Programmiersprache beschreiben zu können,
- einer Spezifikation, der Programmiersprachen für die CLI genügen müssen (die *Common Language Specification*, CLS) und schließlich
- einer virtuellen Ausführungsumgebung (*Virtual Execution System*, VES), um CLI-kompatible Programme abwickeln zu können.

Alle CLI-kompatiblen Programmiersprachen werden zu einer gemeinsamen Zwischensprache, der *Common Intermediate Language* (CIL) [vgl. Lidin 2002], kompiliert. Diese Zwischensprache abstrahiert von der tatsächlichen Hardware, auf der das Programm abgewickelt werden soll. Erst bei der Abwicklung wird die CIL des Programms von der plattformspezifischen VES-Implementierung in die Maschinensprache der Hardware kompiliert.

Derzeit gibt es laut [Wikipedia 2008b] 50 CLI-kompatible Programmiersprachen. Einige davon sind *C#* [Microsoft Corporation 2003a, 2005a,b], *VB.*NET (Basic), *C++/CLI* (C++), Delphi.NET (Pascal) und *IronPhython* (Phyton). Die in dieser Arbeit verwendete CLI-Sprache wird hauptsächlich C# sein.

3.2 Metaprogrammierung in der CLI

Eine wesentliche Basistechnologie, die in dieser Arbeit verwendet wird, ist die Metaprogrammierung oder auch Reflexion. Sie beschreibt die Fähigkeit von Programmen, Informationen über ihre eigene Struktur zu erhalten bzw. sich selbst zu erweitern. Die Idee geht auf [Smith 1982] zurück, in dessen Modell meta-zirkuläre Interpreter laufen, bei denen jeder Interpreter den nächsten „unter“ sich direkt ausführt. Da ein Programm über den Code der nächsten höheren Ebene reflektieren kann, ist es möglich, einen Code auf einer höheren Metaebenen auszuführen. [Friedman und Wand 1984] beschränken dieses Modell auf eine Basisebene und eine Metaebene. Die Metaebene ist die Konkretion des Programms inklusive seines Zustands, das auf der Basisebene ausgeführt wird. Durch Reflexion kann die Metaebene von der Basisebene abgefragt und verändert werden.

Eine Alternative zu selbst untersuchenden Interpretern ist die objektorientierte Programmierung. In objektorientierten Systemen wird das Verhalten von Objekten in Klassen spezifiziert, in denen wiederum eine Menge von Methoden und Attributen definiert ist. Ein *Metaobjektprotokoll* [Maes 1987] erweitert das objektorientierte System, indem es jedem Objekt ein korrespondierendes Metaobjekt zuordnet. Ein Metaobjekt ist ein Exemplar einer Metaklasse. Die Methoden der Metaklasse spezifizieren das Verhalten auf der Metaebene eines Objekt/Metaobjekt-Paares. Das Verhalten auf der Metaebene beinhaltet u. a. die Funktionsweise der Vererbung, der Exemplarbildung und des Methodenaufrufs.

Auch die CLI erlaubt den Zugriff auf die Metadaten eines Programms über das CTS. Unter dem Namensraum `System.Reflection` sind unter anderem Metaklassen für Komponenten (Assemblies), Klassen, Interfaces, Methoden, deren Parameter und dessen CIL zu finden. Jedes Objekt hat z. B. über die Methode `GetType` Zugriff auf sein korresponierendes Metaobjekt, ein Exemplar der Metaklasse `Type`. Dieses Metaobjekt repräsentiert die Klasse, die das Objekt definiert. Über dieses Metaobjekt lassen sich wiederum weitere Metaobjekte erfragen, wie die Komponente, in der die Klasse definiert wurde (`Assembly`), ihre Methoden mit den Parametern (`MethodInfo` und `ParameterInfo`) und ihre Attribute (`FieldInfo`).

Eine andere Möglichkeit der Verdinglichung von Klassen und Interfaces stellt der **typeof**-Operator dar. Er liefert direkt ein Metaobjekt des angegebenen Typs.

Die CLI erlaubt es weiterhin, Metaobjekte mit benutzerdefinierten Zusatzinformationen - sogenannten *CLI-Attributen* - zu versehen. CLI-Attribute sind spezielle Klassen, die die Klasse `System.Attribute` spezialisieren. Im Quelltext einer Anwendung können sie unter anderem Klassen, Interfaces, Komponenten, Methoden, Parameter und Klassen-Attribute annotieren, indem sie z. B. in der Programmiersprache C# in eckigen Klammern vor das entsprechende Element geschrieben werden. Die Annotation einer Klasse sieht dann wie folgt aus:

```
[BenutzerdefiniertesAttribut]
public class AnnotierteKlasse { ... }
```

Schließlich bietet die CLI über die im CTS definierten Klassen `System.Reflection.Emit` die Möglichkeit, die Metaebene zu erweitern und neue Klassen und Komponenten zur Laufzeit zu erzeugen. Hierzu stellt das CTS jeweils eine spezialisierte Form der bereits genannten Metadatenklassen zur Verfügung. Jede neue Komponente wird durch ein Metaobjekt der Metaklasse `AssemblyBuilder` repräsentiert. Auf diesem können nun beliebig neue Klassen und Interfaces (Exemplare von `TypeBuilder`) erzeugt werden. Ein `TypeBuilder`-Metaobjekt stellt unter anderem Metamethoden zum Erzeugen von Klassenattributen (`FieldBuilder`) und Methoden `MethodBuilder` sowie Konstruktoren (`ConstructorBuilder`) bereit. In letztere kann der Code - die CIL - und somit das Verhalten eingefügt werden.

Eine dynamisch erzeugte Klasse kann erst verwendet werden, wenn ihre Definition explizit abgeschlossen wurde. Das erfolgt durch den Aufruf der Methode `CreateType` auf dem entsprechenden Metaobjekt. Ab diesem Zeitpunkt können die Metatdaten der Klasse nur noch gelesen werden. Das gilt insbesondere auch für alle nicht dynamisch erzeugten Metadaten.

3.3 Die Geschichte des LOOM.Net-Projektes

Seinen Ursprung hat das LOOM.NET-Projekt in einem bereits seit dem Jahre 2001 vom Autor entwickelten Werkzeug mit dem Namen *Wrapper Assistant*. Hierbei handelte es sich um den Ansatz eines statischen Aspektwebers [Schult und Polze 2002b; Schult u. a. 2003], bei dem die Grundidee der Annotationsrelation bereits enthalten war. Aspekte wurden hier in Form von Schablonen definiert, die auf die Metadaten der annotierten Klassen angewendet wurden. Das Ergebnis war eine von der annotierten Klasse abgeleitete und mit dem Quelltext der Schablone dekorierte Klasse. Die Schablonentechnik erwies sich jedoch als zu unflexibel und wenig intuitiv.

Ein neuer Ansatz war der vom Autor entwickelte *Dynamic Weaver*, der Vorläufer von RAPIER-LOOM.NET [Schult und Polze 2002a, 2003]. Dieser ermöglichte, Aspekte mit dem CLI-Standard zu definieren und diese dynamisch zu verweben. Aspekte konnten hier bereits Klassen und Assemblies annotieren - die erste Umsetzung des Konzepts der vertikalen Überdeckung. Unterstützt wurden jedoch nur Advices, die in den Aufruf virtueller Methoden eingewoben werden sollten.

Der dynamische Weber wurde ab Ende des Jahres 2003 unter dem offiziellem RAPIER-LOOM.NET - stetig weiterentwickelt. Es kamen die Unterstützung für Introduktionen und die Möglichkeit der feingranularen Selektion von Joinpoints hinzu. Nach dem Erscheinen des .NET-Framworks 2.0 wurde schließlich ab dem Jahr 2006 die Version 2.0 veröffentlicht. Diese wurde gegenüber der Vorgängerversion grundlegend überarbeitet. Ab dieser Version wurde die Spracherweiterung für die Aspektdefinition komplett vom Aspektweber getrennt. Dadurch wurde es möglich, Aspekte unabhängig von den später verwendeten Verwebungstechnologien zu definieren und in Komponenten zu verpacken. Solche Aspekt-Komponenten können - einmal definiert - sowohl mit RAPIER-LOOM.NET als auch mit dem ab dem Jahr 2007 entwickelten statischen Aspektweber GRIPPER-LOOM.NET verwoben werden.

In der aktuellen Version 2.2 sind inzwischen alle im Kapitel 2 beschriebenen Konzepte umgesetzt worden. Das gesamte Projekt inklusive aller Testfälle umfasst inzwischen etwas

mehr als 35.000 Zeilen Quelltext.

3.4 Eine Abbildung der LOOM.Net-Spracherweiterung auf die CLI

In Kapitel 2 sind die neuen Gruppierungseinheiten Aspekt, Introduktion und Advice sowie das Konzept der Joinpointvariablen und des Ausführungskontextes eingeführt worden. Für jedes dieser Konzepte wurde eine Abbildung auf die CLI für objektorientierte Sprachen entwickelt, die im Folgenden vorgestellt werden soll. Implementiert sind diese Konzepte in einer eigenständigen Komponente, der `loom.dll`. Diese Komponente enthält jedoch *keinen* Aspektweber.

Dass die `loom.dll` selbst keinen Aspektweber enthält, ist aus mehreren Gründen wichtig. So lassen sich Aspekte unabhängig von der später gewählten Verwebungsstrategie definieren. Außerdem bleiben Komponenten, die Aspekte enthalten, klein und können ohne den umfangreichen Code eines Aspektwebers ausgeliefert werden.

In der `loom.dll` ist neben den Sprachelementen, die zum Implementieren der LOOM.NET-Konzepte verwendet werden, auch ein LOOM.NET-Metamodell enthalten. Das Metamodell beschreibt die Bedeutung der Sprachelemente und wird von den Aspektwebern verwendet, um von einer aspektorientierten Beschreibung durch einen Verwebungsvorgang auf eine objektorientierte Abbildung zu kommen.

3.4.1 Definition einer Aspektklasse

Es existieren zwei Möglichkeiten, einen Aspekt mit der `loom.dll` zu definieren: Die erste Möglichkeit ist durch eine Ableitung von dem Interface `Loom.Aspect`. Das Interface hat selbst keine Methoden und benötigt somit auch keine Implementation. Die zweite Variante ist die Ableitung von der Basisklasse `Loom.AspectAttribute`. Letztere stellt eine direkte Ableitung der Klasse `System.Attribute` aus der CLI dar, die verwendet wird, um benutzerdefinierte Metadaten zu definieren. Über diese Variante können Aspekte Gruppierungseinheiten annotieren. So definierte Aspekte werden zukünftig auch als *Aspektklassen* bezeichnet. Das veranschaulicht, dass der Aspekt in der CLI mit Hilfe der `loom.dll` definiert wurde.

Eine Aspektklasse kann - wie jede normale CLI-Klasse auch - beliebige Methoden und Attribute definieren. Eine in einer Aspektklasse definierte Methode kann jedoch auch die Rolle eines Advices oder einer Introduktion annehmen. Das geschieht, indem die Methode mit einem *Aspektmethodenattribut* annotiert wird. Die `loom.dll` definiert unter `Loom.Joinpoints` insgesamt zehn verschiedene Aspektmethodenattribute, die im folgenden Abschnitt näher untersucht werden sollen.

Aspekte können verschiedene Eigenschaften haben. Eine dieser Eigenschaften betrifft die Exemplarbildung, wie sie mit **per** angegeben wird. In der `loom.dll` existiert hierzu das CLI-Attribut `Loom.AspectProperties.CreateAspectAttribute`, das als Annotation auf Aspektklassen angewendet werden kann. Als Parameter wird der Aufzählungstyp `Per` mit den Werten `Instance`, `Class`, `Annotation` und `AspectClass` verwendet.

Eine weitere Aspekteigenschaft ist **noninherited**, mit der die horizontale Überdeckung explizit ausgeschaltet werden kann. Die `loom.dll` definiert hierzu kein eigenes CLI-Attribut, sondern verwendet das `System.AttributeUsage`-Attribut der CLI [Microsoft Corporation u. a. 2006, Teil I, S. 57]. Ist eine Aspektklasse mit diesem CLI-Attribut versehen und der Eigenschaftswert `Inherited` auf **false** gesetzt, wird für die Aspektklasse keine horizontale Überdeckung verwendet.

3.4.2 Advices und Introduktionen

Advices und Introduktionen werden als Methoden in der Aspektklasse definiert und mit einem Aspektmethodenattribut annotiert, damit sie als solche erkannt werden können. Optional kann eine Aspektmethode auch mit einem oder mehreren *Pointcut-Attributen* versehen werden. Ausgehend von der LOOM.NET-Spracherweiterung ergibt sich folgende Abbildung eines Advices auf eine Methodendeklaration in der CLI:

advice before	call	[!Baz] IFoo.!foo,bar	(p:<T>, ...):int where T:FooBar
Advice-Aufzählung	Aspekt-methoden-attribut	Pointcut-Attribute	Methodensignatur

Als Aspektmethodenattribute stehen `Create`, `Init`, `Call`, `Access`, `Destroy` und `Finalize` zur Verfügung. Diese bilden sich auf die gleichnamigen Schlüsselwörter der LOOM-Spracherweiterung ab. Als Parameter erwarten diese CLI-Attribute einen Aufzählungswert der Advice-Aufzählung `Advice`. Mögliche Werte sind hier `Before`, `After`, `AfterThrowing`, `AfterReturning`, `Around` und `Initialize`, die auch eine Entsprechung in den schon bekannten Schlüsselwörtern finden.

Um Advices mit einem einfachen Pointcut zu definieren, reichen diese CLI-Attribute bereits aus: Die Signatur der Aspektmethode und der Advicetyp beschreiben den zu selektierenden Joinpoint bereits vollständig. Die Advicemethode

```
[Call(Advice.Before)]
public void foo(double bar)
```

selektiert alle Joinpoints mit dem Methodennamen `foo` und einem Parameter vom Typ `double`.

Für erweiterte Pointcuts müssen Pointcut-Attribute verwendet werden. Die `loom.dll` definiert folgende CLI-Attribute zur Selektion der Joinpoints:

- `IncludeIfAnnotated/ ExcludeIfAnnotated`: Definiert die Liste der ein- oder auszuschließenden Metainformationen, die die zum Joinpoint gehörende Methode annotieren müssen oder nicht annotieren dürfen.

- `Include/ Exclude` unter Angabe eines Typs: Definiert die ein- oder auszuschließenden Schnittstellen (Interfaces oder Klassen), zu denen der Joinpoint gehört.

- `Include/ Exclude` unter Angabe eines Namens: Definiert die Liste der ein- oder auszuschließenden Methodennamen des Joinpoints.

Innerhalb einer Gruppe dürfen entweder nur `Include*`- oder `Exclude*`-Attribute verwendet werden. Das entspricht dem Weglassen oder Verwenden des Ausrufezeichens in der LOOM-Sprachdefinition für die jeweilige Gruppe.

Ist eine Aspektmethode mit Pointcut-Attributen definiert, wird ihre Signatur für die Selektion der Joinpoints wie folgt interpretiert:

- Der Name der Aspektmethode wird ignoriert.

- Generische Typen in der Parametersignatur werden als Platzhalter für einen einzelnen Parameter interpretiert. Enthält die Methodendefinition eine **where**-Klausel, so schränkt diese den Parameter in Hinblick auf den zu selektierenden Joinpoint entsprechend ein.

- Ein Parameter für variable Argumentlisten (in C# durch das Schlüsselwort **params**) wird als Platzhalter für eine beliebige Anzahl von Parametern interpretiert. In der LOOM-Spracherweiterung wurde das mit „**...**“ angegeben.

Soll die Definition einer Aspektmethode als erweiterter Pointcut interpretiert werden, obwohl kein Pointcut-Attribut definiert wurde, kann das Attribut `IncludeAll` verwendet werden. Das Attribut wird so interpretiert, dass in keiner der Gruppen eine Einschränkung für die Selektion eines Joinpoints existiert. Die Aspektmethode:

```
[Call(Advice.Before), IncludeAll]
public void foo(params object[] args)
```

selektiert alle Joinpoints in der Überdeckung der Aspektklasse.

Für Introduktionen gibt es das Aspektmethodenattribut `Introduce`. Das Attribut erwartet als Parameter zusätzlich einen Interface, um die Introduktion eindeutig diesem einzuführenden Interface zuordnen zu können.

3.4.3 Joinpoint-Parameter und der Ausführungskontext

Der Ausführungskontext, d. h. die Metavariablen **target** und **joinpoint**, stehen in den Aspektmethoden nicht direkt zur Verfügung. Sie müssen vielmehr der Aspektmethode explizit übergeben werden. Dies geschieht durch die optionale Deklaration von *Joinpoint-Parametern*.

Joinpoint-Parametern sind die ersten Parameter einer Aspektmethode und werden durch ein *Joinpoint-Parameter-Attribut* markiert. Bei der Auswertung der Methodensignatur für die Selektion eines Joinpoints werden diese Parameter ignoriert. Eine Aspektmethode kann beliebig viele Joinpoint-Parameter in ihrer Parametersignatur deklarieren.

Der Zugriff auf die Metavariable **target** erfolgt über einen Parameter mit dem Typ `object`, der zusätzlich mit dem Joinpoint-Parameter-Attribut `JPTarget` versehen wurde. Auf die Metavariable `joinpoint` wird über einen Parameter vom Typ `Context` zugegriffen, der mit dem Attribut `JPContext` versehen wurde:

```
[Call(Advice.Around), IncludeAll]
public void foo([JPTarget] object target, [JPContext] Context
    joinpoint, params object[] args)
```

In der `Context`-Klasse sind alle aus Abbildung 2.15 bekannten Methoden definiert. Zusätzlich existieren zu den Methoden `Call` und `CallOn` die Varianten `Invoke` und `InvokeOn`. Diese Varianten nehmen die am Joinpoint übergebenen Parameter (hier `args`) auch als Parameterfeld mit variabler Länge entgegen; die Semantik ändert sich nicht. Der Aufruf der zum Joinpoint gehörenden Methode erfolgt mit:

```
joinpoint.Invoke(args)
```

und würde die Parameter ohne Änderung an diese Methode weiterleiten.

3.4.4 Die Definition von Joinpoint-Variablen

Die Definition von Joinpoint-Variablen erfolgt mithilfe von Joinpoint-Parametern. Der Name und der Typ der Joinpoint-Variable ergibt sich aus dem Namen und dem Typ des Parameters. Das zugehörige Joinpoint-Parameter-Attribut ist `JPVariable`.

Dem Attribut kann als optionalem Parameter eine Bitmaske übergeben werden, welche die Art, die Sichtbarkeit und die Modifizierer der Joinpointvariablen festlegt. Die Bitmaske kann mit der Aufzählung `Scope` kombiniert werden. Die dort definierten Werte sind:

- für die Art: `Scope.Target`, `Scope.Joinpoint`,

Basisprogramm

```
public interface IBar
{
  int Bar(double d);
}

[MyAspect("Irgend␣ein␣String")]
public class A
{
  protected int foobar;

  public virtual int Foo()
  protected virtual int Baz(double d)
}

public class B
{
  protected int foobar;

  [MyAspect("Noch␣ein␣String")]
  public virtual void Foo(string s)
}
```

Aspekt

```
[CreateAspect(Per.Instance)]
public class MyAspect : AspectAttribute
{
  public MyAspect(string config)

  private void InternalMethod()

  [Call(Advice.Initialize), IncludeAll]
  public void Init(
    [JPContext] Context joinpoint,
    [JPVariable(Scope.Protected| Scope.
        Target| Scope.Virtual)]
    ref int foobar,
    params object[] args)

  [Call(Advice.After)]
  [Include(typeof(A)), Include(typeof(B))]
  public void After<T>(
    [JPRetVal] object retval,
    [JPVariable(Scope.Protected | Scope.
        Target | Scope.Virtual)]
    ref int foobar,
    T firstparam)

  [Introduce(typeof(IBar))]
  static public int Bar(
    [JPTarget] object target,
    double d)
}
```

Abbildung 3.1: Ein Beispiel in C#

- für die Sichtbarkeit: `Scope.Private`, `Scope.Protected` und `Scope.Public`,
- als Modifizierer: `Scope.Static`, `Scope.Virtual`, `Scope.Override`.

Natürlich sind nicht alle Kombinationen erlaubt. Für die Art und die Sichtbarkeit muss genau ein Wert gewählt werden, es ist nicht möglich, `Private`, `Virtual` und `Override` paarweise zu kombinieren.

Damit auf eine Joinpoint-Variable auch schreibend zugegriffen werden kann, muss sie in der Parameterliste als Referenzvariable deklariert weren. In C# erfolgt das mit dem Schlüsselwort **ref**.

3.4.5 Ein Beispiel

Abbildung 3.1 zeigt ein LOOM.NET-Beispiel in einer C#-Syntax. Dabei wurden die Methodendefinitionen aus Gründen der Übersichtlichkeit weggelassen. Der Aspekt ist eine reguläre C#-Klasse. Die beiden Advices und die Introduktion wurden in den Methoden `Init`, `Before` und `Bar` definiert. Außerdem definiert der Aspekt die Joinpointvariable `foobar`, die allerdings nur in den beiden Advices verwendet wird. Beide Advices greifen auch auf den Joinpoint-Kontext zu, während die Introduktion den Zielkontext (`target`) verwendet.

Die Annotationsrelation wurde durch die Verwendung des Aspekts als CLI-Attribut umgesetzt. Der Aspekt annotiert die Klasse `A` und die Methode `Foo`. Die Konstruktorparameter werden bei der Annotation direkt angegeben.

3.4.6 Das LOOM.Net-Metamodell

Das LOOM.NET-Metamodell beschreibt, welche Auswirkungen die Sprachelemente haben. Die Metamodellklassen unterteilen sich in zwei Kategorien. Eine Kategorie beschreibt die Aspekte

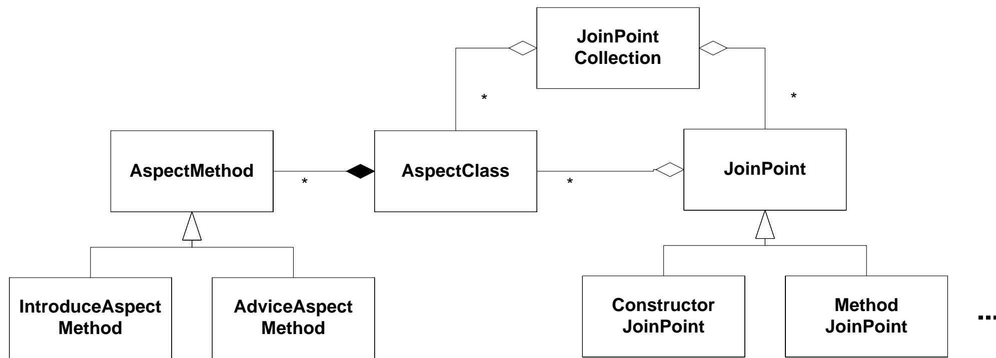


Abbildung 3.2: Ausschnitt aus dem LOOM.NET-Metamodell

mit ihren Advices und Introduktionen, die andere die Joinpoints. Ein LOOM.NET-Aspektweber kann beides über eine interne von der `loom.dll` bereitgestellte Schnittstelle verdinglichen.

Für die Verdinglichung eines Aspekts wird das Typobjekt - selbst ein Metaobjekt - der Aspektklasse benötigt. Über Reflexion der auf diesem Typobjekt definierten Methoden baut die `loom.dll` ein Modell des Aspekts auf. Jede Methode und ihre Parameter werden hierzu auf die in den vorangegangenen Abschnitten beschriebenen benutzerdefinierten CLI-Attribute untersucht. Des Weiteren werden die Aspekteigenschaften durch Abfrage der entsprechenden CLI-Attribute im Modell gesetzt.

Es gibt Fälle, in denen für eine Aspektklasse kein Modell erzeugt werden kann. Das ist z. B. der Fall, wenn die Definition der Aspektmethoden widersprüchlich oder falsch ist. In diesem Fall kommt es beim Abfragen der Metadaten zu einem Fehler, der dann in Form einer Fehlermeldung an den Benutzer weitergegeben wird.

Joinpoint-Metaobjekte werden direkt aus den Metaobjekten der CLI für Methoden, Konstruktoren, Eigenschaftsmethoden, Ereignismethoden und Destruktoren abgeleitet. Ein Joinpoint-Metaobjekt enthält alle Attribute eines Joinpoints, die von Advices und Introduktionen selektiert werden können. Ein Aspektweber erhält die Joinpoint-Metaobjekte, indem er eine `JoinpointCollection` für eine Klasse erstellt. Dargestellt ist dies in Abbildung 3.2.

Die `JoinpointCollection`-Metaobjekt referenziert alle Aspekt-Metaobjekte, die aus der Überdeckung der angegebenen Klasse resultieren. Die Menge ist geordnet bezüglich des Ranges der abgebildeten Aspekte. Ein Joinpoint-Metaobjekt referenziert ebenso Aspekt-Metaobjekte. Das sind diejenigen Aspekte, die eine Methode direkt annotieren.

3.5 Der dynamische Aspektweber Rapier-Loom.Net

Der erste vom Autor entwickelte Aspektweber war der dynamische Weber RAPIER-LOOM.NET. *Dynamisches Weben* bedeutet, dass Advices und Introduktionen sowie Joinpointvariablen erst zur Laufzeit in die vom Aspekt überdeckten Klassen an den selektierten Joinpoints eingebracht werden. Das hat zwar erhöhte Laufzeitkosten zur Folge - da offensichtlich zur Laufzeit die Metadatenstruktur verändert werden muss - aber auch einige Vorteile. So ermöglicht dieses Konzept, eine Überdeckung erst zur Laufzeit zu definieren. Exemplare einer Klasse können mit unterschiedlichen Überdeckungen - mit oder ohne einen bestimmten Aspekt - gebildet werden.

Die Verwebung zur Laufzeit bringt allerdings neben den erhöhten Laufzeitkosten auch einige andere Einschränkungen mit sich. Soll die Laufzeitumgebung (CLR) nicht geändert werden, müssen die Metadaten manipuliert werden. Die CLR lässt nur das Hinzufügen neuer Metadaten zu. Konkret wird ein neues Metaobjekt, ein Stellvertreter erzeugt, der das Verhalten der überdeckten Klasse und aller überdeckenden Aspekte in sich vereint.

Alternative Ansätze verwenden die *Debug-* sowie *Edit & Continue*-Schnittstellen der

.NET-Laufzeitumgebung [Pietrek 2002; Stall 2005], die jedoch ausschließlich für die Microsoft .NET-Implementierung der CLI funktionieren. Aus Sicht des Autors sind diese Schnittstellen auch deshalb ungeeignet, da sie für die Fehlersuche in Programmen und nicht für den Produktivbetrieb vorgesehen sind.

Eine weitere Variante stellen die *Remotig*-Schnittstellen der CLI dar [Microsoft Corporation u. a. 2006, Teil I, S 92 ff.]. Diese Variante hat jedoch viele Einschränkungen: So können nur Klassen erweitert werden, die von der Klasse `MarshalByRefObject` erben. Außerdem fehlen Konzepte, um Introduktionen und Joinpointvariablen umzusetzen.

3.5.1 Der Stellvertreter

RAPIER-LOOM.NET generiert für jede Klasse, die mit einem Aspekt überdeckt wurde, einen Klassenstellvertreter [Schult und Polze 2002a]. Das bedeutet, dass ein Klient nicht auf einem Exemplar der von ihm angeforderten Klasse, sondern auf einer Ableitung von dieser operiert. Im Allgemeinen ist das kein Problem, die Typrelationen bleiben erhalten. Eine Exemplar vom Typ A bleibt ein Exemplar vom Typ A, auch wenn es mit einem Aspekt verwobenen ist. Einzig die Reflexion über den dynamischen Typ des Exemplars liefert ein anderes Ergebnis, da dieser dem Klassenstellvertreter entspricht.

Die Verwebung muss grundsätzlich drei verschiedene Anwendungsfälle umsetzten: Das sind das Vermischen von bestehenden Methoden mit Advices, das Einführen von Introduktionen sowie das Einführen und Verknüpfen von Joinpointvariablen.

Advices werden vom Aspektweber implementiert, indem die hinter einem Joinpoint liegende Methode überladen wird und in der Überladung die Verbindung zwischen den Advices und der originalen Methode hergestellt wird. Im Nachfolgenden wird das als *Verbindungscode* bezeichnet. Eine Ausnahme sind die **create**-Advices, bei denen der Verbindungscode in den später beschriebenen *Klassenobjekten* implementiert wird.

Der Verbindungscode hat folgende Aufgaben:

- Aufruf der originalen Methode,
- Abbilden der übergebenen Parameter und der Joinpointparameter auf die Parameter der Advices,
- Aufruf der Advices in der richtigen Reihenfolge,
- Zugriff auf Joinpointvariablen,
- Erzeugen eines Joinpointkontexts und
- Abfangen von Ausnahmen.

Der Stellvertreter referenziert entsprechend der **per**-Eigenschaft ein oder mehrere Aspektexemplare, auf denen die Advices ausgeführt werden. Der Aufruf der Advices erfolgt dann auf den entsprechend zugeordneten Exemplaren.

Wurden über Joinpointvariablen neue Attribute eingeführt, so werden auch diese im Stellvertreter definiert. Dabei sind die verwendeten Modifizierer der Joinpointvariablen ausschlaggebend für die Definition des zugehörigen Attributs im Stellvertreter:

- Öffentliche und geschützte Joinpointvariablen vom Typ **target** werden als gleichnamiges Attribut mit identischen Modifizierern (**public** oder **protecetd** und ggf. **static**) definiert.
- Private Joinpointvariablen vom Typ **target** werden als privates Attribut (ggf. mit dem Modifizierer **static**) unter einem eindeutigen Namen definiert. Das ist notwendig, um Namenskonflikte mit gleichnamigen Joinpointvariablen anderer Aspekte zu vermeiden.

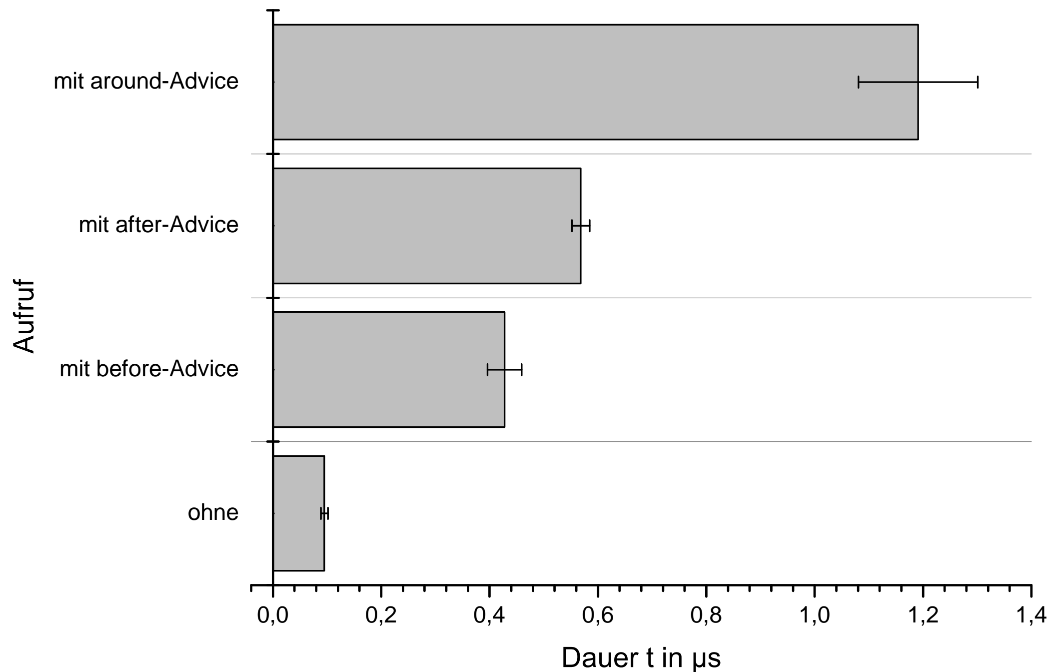


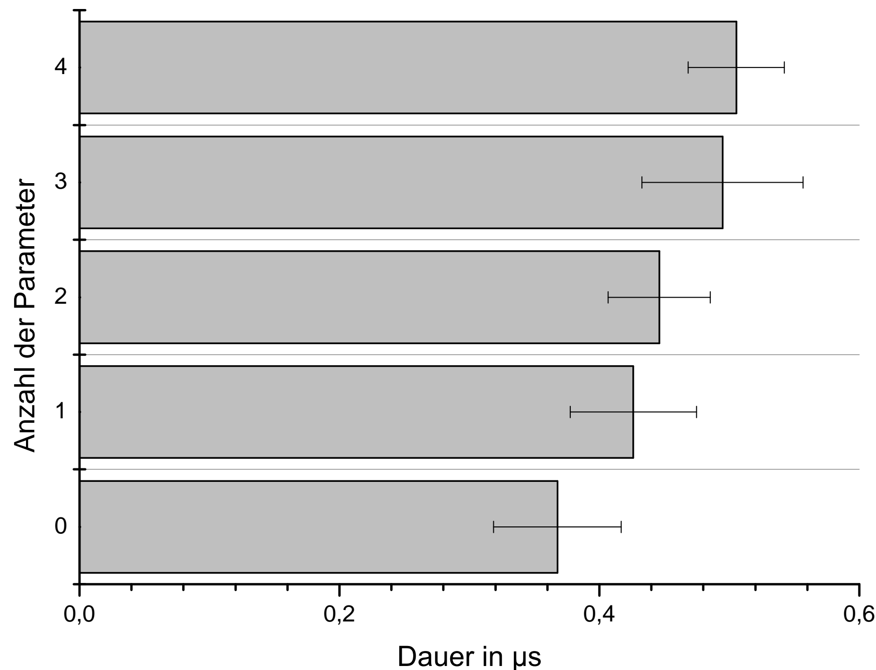
Abbildung 3.3: Durchschnittliche Kosten für des Verbindungscode beim Aufruf einer Methode mit einem Parameter

- Joinpointvariablen vom Typ **joinpoint** werden von jedem Advice selektierten Joinpoint genau einmal unter einem jeweils eindeutigen Namen definiert.

Für Introduktionen wird ein ähnlicher Verbindungscode erzeugt wie bei den Advices. An Stelle der Überladung wird dem Stellvertreter ein neues Interface hinzugefügt, dessen Implementation der Verbindungscode ist. Der Verbindungscode besteht hier im Allgemeinen aus einer direkten Weiterleitung des Aufrufes an die entsprechende Introduktion im Aspekt. Falls die Introduktion einen bereits existierenden Joinpoint überschreibt (mit gesetztem Modifizierer **virtual**), wird sie wie ein **around**-Advice implementiert.

In Abbildung 3.3 sind die durchschnittlichen Kosten dargestellt, die der Verbindungscode bei einem Aufruf einer Methode mit einem Parameter erzeugt. Diese Methode wurde - mit Ausnahme der Messreihe *ohne* - mit verschiedenen Advices verwoben. Wurde kein Advice verwoben (*ohne*), wird auch kein Verbindungscode erzeugt. In diesem Fall folgt auf den Aufruf **call** unmittelbar der Rücksprung **ret**. In anderen Fällen wird der Aufruf vom Verbindungscode behandelt. Dieser ruft dann sowohl die originale Methode als auch den Advice auf. Im Fall von *after* wird zusätzlich eine Ausnahmebehandlung benötigt, da die originale Methode mit einer Ausnahme zurückkehren könnte. Im *around* Fall kommt zusätzlich eine Indirektion hinzu: Die originale Methode wird nicht direkt vom Verbindungscode, sondern indirekt vom Advice über den Ausführungskontext gerufen. Der Verbindungscode erzeugt ein Exemplar des aktuellen Ausführungskontextes und übergibt diesen als Joinpoint-Parameter an den Advice.

Abbildung 3.4 zeigt, wie die durchschnittlichen Kosten in Abhängigkeit von den Methodenparametern ansteigen. In dem hier dargestellten Fall verwendet der verwobene Advice Platzhalter für eine beliebige Anzahl an Parametern. Das wird in der LOOM.NET-Deklaration des Advices mit einem Parameter für variable Argumentlisten ausgedrückt. Dieser Parameter ist ein Feld, welches vom Verbindungscode vor dem Aufruf des Advices mit den tatsächlichen Parametern belegt wird. Das erklärt den steigenden Aufwand bei steigender Parameterzahl.



*Gemessen wurden 1000 Methodenaufrufe auf einem Intel Core Duo T5500, 1,66 GHz

Abbildung 3.4: Durchschnittliche Kosten für des Verbindungscode beim Aufruf einer mit einem before-Advice verwobenen Methode

3.5.2 Schnelle Exemplarbildung

Während in Java mit dem Konzept der *ClassLoader* ein effektives Konzept zur Verfügung steht, die Metadaten einer Klasse transparent zu manipulieren, ist das in der CLI nicht vorgesehen. Hier muss man sich mit dem *Fabrikmuster* [Gamma u. a. 1995, S. 107 ff.] behelfen. Anstelle des **new**-Operators tritt ein expliziter Aufruf des Aspektwebers:

```
Weaver.Create<A>() // anstelle von new A()
```

Die große Herausforderung ist es, die Laufzeitkosten für die Exemplarbildung gering zu halten. Hohe Laufzeitkosten entstehen an verschiedenen Stellen:

1. Für eine angeforderte Klasse muss der zugehörige Stellvertreter gesucht werden.

2. Wurde kein Stellvertreter gefunden, muss ein neuer Stellvertreter erzeugt werden.

3. Für die übergebenen Parameter der `Weaver.Create`-Funktion muss der richtige Konstruktor gefunden werden.

Muss ein neuer Stellvertreter erzeugt werden, kommt es darauf an, die Verwebung so effizient wie möglich zu gestalten. Allerdings tritt dieses Ereignis während der Lebenszeit einer *Applikationsdomäne* für jede Klasse höchstes einmal auf - genau dann, wenn für diese Klasse erstmalig ein verwobenens Exemplar angefordert wird. Eine Applikationsdomäne bezeichnet in der CLI eine isolierte Umgebung, in der die Programme abgewickelt werden. Sie ist vergleichbar mit einem Prozess im Betriebssystem, wobei nicht ausgeschlossen ist, dass die VES mehrere Applikationsdomänen in einem Betriebssystemprozess verwaltet.

Rapier-Loom.Net kann den Verwebungsprozess durch Persistierung des erzeugten Codes beschleunigen. Im Allgemeinen dauert der Verwebungsvorgang mit dem Erzeugen einer neuen Stellvertreterklasse um ein Vielfaches länger, als wenn die Stellvertreterklasse aus einer Komponente vom Sekundärspeicher (Festplatte) geladen wird. Rapier-Loom.Net legt nun

für jede in einer Applikationsdomäne geladenen Komponente eine korrespondierende dynamische Komponente für die Stellvertreterklassen an. Der Stellvertreter einer jeden Klasse wird in der korrespondierenden dynamischen Komponente unter demselben voll qualifizierenden Namen abgelegt. Beim Beenden der Applikationsdomäne wird die dynamische Komponente auf die Festplatte geschrieben. Das geschieht unter demselben Dateinamen wie das Original, jedoch zusätzlich mit dem Präfix „Loom.“.

Beim erneuten Starten der Anwendung wird bei einer Exemplaranforderung zuerst geprüft, ob für die Komponente der Klasse bereits eine gleichnamige mit dem „`Loom.`“ Präfix existiert. In diesem Fall wird der Stellvertreter aus der Komponente geladen.

Ein Problem entsteht, weil sich die Metadaten der dynamischen Komponente nach dem Abspeichern und dem späteren Laden nicht mehr verändern lassen. Werden nun Exemplare für Klassen angefordert, für die noch kein Stellvertreter existiert, muss eine neue dynamische Komponente angelegt werden. Hier wird dasselbe Verfahren zur Persistierung verwendet, nur erhält diese Komponente das Präfix „`Loom_1.`“, die folgende „`Loom_2`“ usw. Je öfter eine Applikation läuft, desto vollständiger wird die Menge der Stellvertreter in diesen Komponenten. Die Fälle, in denen eine Verwebung gestartet werden muss, tendieren dann gegen Null.

Für den zweiten und dritten Punkt - das Auffinden des richtigen Stellvertreters und die Zuordnung des richtigen Konstruktors - generiert RAPIER-LOOM.NET sogenannte *Klassenobjekte* (siehe Abbildung 3.5). Für jeden Stellvertreter existiert genau eine spezialisierte Klasse eines Klassenobjekts in der dynamischen Komponente. Die Spezialisierung erfolgt dabei über mehrere Stufen:

- `Loom.Runtime.ClassObject`: Die abstrakte Basisklasse aller Klassenobjektklassen (kurz: **CCO**).
- `A$ClassObject$0`: Eine spezialisierte, aber noch abstrakte Klassenobjektbasisklasse für jede überdeckte Klasse der Benutzerkomponente (kurz: $\mathbf{CCO_{id}^{base}}$).
- `A$ClassObject`: Die Klassenobjektklasse für den Stellvertreter, in dem alle die originale Klasse überdeckenden Aspekte (`FooAspect`) implementiert sind (kurz: $\mathbf{CCO_{id}^{0}}$).
- `A$ClassObject$1`: Die Klassenobjektklasse für weitere Stellvertreter, die aus der dynamischen Verwebung weiterer Aspekte (`BarAspect`) resultieren (kurz: $\mathbf{CCO_{id}^{n}}$, $\mathbf{n \geq 1}$).

Von den verschiedenen Spezialisierungen $\mathbf{CCO_{id}^{n}}$ existiert jeweils nur ein einziges Exemplar. Das Kürzel *id* steht für den Identifizierer der originalen Klasse.

Exemplare von $\mathbf{CCO_{id}^{n}}$ werden intern in einer Hash-Tabelle (*Hash Table*) als Wert abgelegt, dessen Schlüssel der Typidentifizierer (*id*) der originalen Klasse ist. Bei einer Exemplaranforderung sucht RAPIER-LOOM.NET in dieser Datenstruktur nach einem Klassenobjekt. Konnte ein solches nicht gefunden werden, wird für die originale Klasse der Verwebungsvorgang angestoßen.

Die Klassendefinition eines Klassenobjekts ist jeweils spezialisiert, genau den ihm zugeordneten Stellvertreter zu erzeugen. Verantwortlich hierfür sind die Implementationen von `CreateInstanceInternal` und `CreateTO`.

Die Methode `CreateInstanceInternal` ist in den abstrakten Basisklassen $\mathbf{CCO_{id}^{base}}$ definiert. In dieser Methode wird ein Multidispatch für alle Konstruktoren der originalen Klasse *id* implementiert. Der richtige Konstruktor wird aus den dynamischen Typen der Konstruktorargumente gewählt, wie sie an die Farbrikmethode des Webers übergeben wurden. In Abbildung 3.5 sind es die Konstruktoren der Klasse `A`, und es wird überprüft, ob kein Parameter oder ein Stringobjekt übergeben wurden.

Anstatt die Konstruktoren direkt aufzurufen, ruft die Methode `CreateInstanceInternal` eine auf die Konstruktorparameter spezialisierte virtuelle Methode `CreateAI` auf. Die Implementation von `CreateAI` in $\mathbf{CCO_{id}^{base}}$ ruft wiederum die abstrakte Methode `CreateTO` mit

Abbildung 3.5: Struktur der Klassenobjekte

denselben Parametern auf. Der Grund für diese dreistufige Indirektion wird in der für den Stellvertreter spezialisierten Klasse des Klassenobjekts deutlich.

Jede Implementation von $\mathbf{CCO_{id}^{n}}$ überschreibt (in Abbildung 3.5 `A$ClassObject`) alle `CreateTO` und optional alle `CreateAI` Methoden. In `CreateTO` wird der Stellvertreter erzeugt, indem der Konstruktor mit derselben Parametersignatur gerufen wird. Implementiert der Aspekt ein **create**-Advice, der den Joinpoint für die Objekterzeugung selektiert, wird dieser aus der Methode `CreateAI` heraus aufgerufen. Das geschieht nach demselben Muster, wie die Advice-Aufrufe im Stellvertreter (siehe Abschnitt 3.5.1) implementiert wurden.

Da in Abbildung 3.5 ein **around create**-Advice für die Exemplarbildung mit einem Stringobjekt definiert wird, wird in `A$ClassObject` die Methode `CreateAI(aspects:Aspect[], string s)` überschrieben. Der Parameter `aspects` wird intern verwendet, um die Aspektexemplare mit dem verwobenen Exemplar zu verbinden. Die Implementation von `CreateAI` besteht ausschließlich aus dem Aufruf des Advices. Als Aufrufkontext wird ein Exemplar der spezialisierten Klasse `A$ClassObject.A$CreateContext$1` übergeben. Dessen Implementation der `Call` Methode ruft die Basisimplementierung von `CreateAI` auf. Diese wiederum ruft die virtuelle Methode `CreateTO` und damit die Erzeugung des Stellvertreters auf. Anschaulich dargestellt ist dieser Prozess noch einmal als Sequenzdiagramm in Abbildung 3.6.

Werden zur Laufzeit zusätzlich Aspekte an ein Exemplar gebunden, müssen neue Stellvertreter und Klassenobjektklassen ($\mathbf{CCO_{id}^{1..n}}$) erzeugt werden. Für jeden dynamischen Aspekt wird die Hierarchie jeweils um eine Ableitung erweitert. Wenn beispielsweise von der Klasse `A` aus Abbildung 3.5 ein Exemplar mit dem dynamisch hinzugefügten Aspekt `BarAspect` erzeugt werden soll, wird ein neuer Stellvertreter `A$1` und eine Klassenobjektklasse `A$ClassObject$1` erstellt. Die Methoden werden nach dem bereits beschriebenen Schema überschrieben.

Die Suche nach dem richtigen Klassenobjekt bei der Exemplarbildung läuft wie folgt ab: Ausgehend vom bereits gefundenen Klassenobjekt für die originale Klasse geht die Implementation für jeden dynamisch hinzugefügten Aspekt über die Hash-Tabelle `subClassObj`

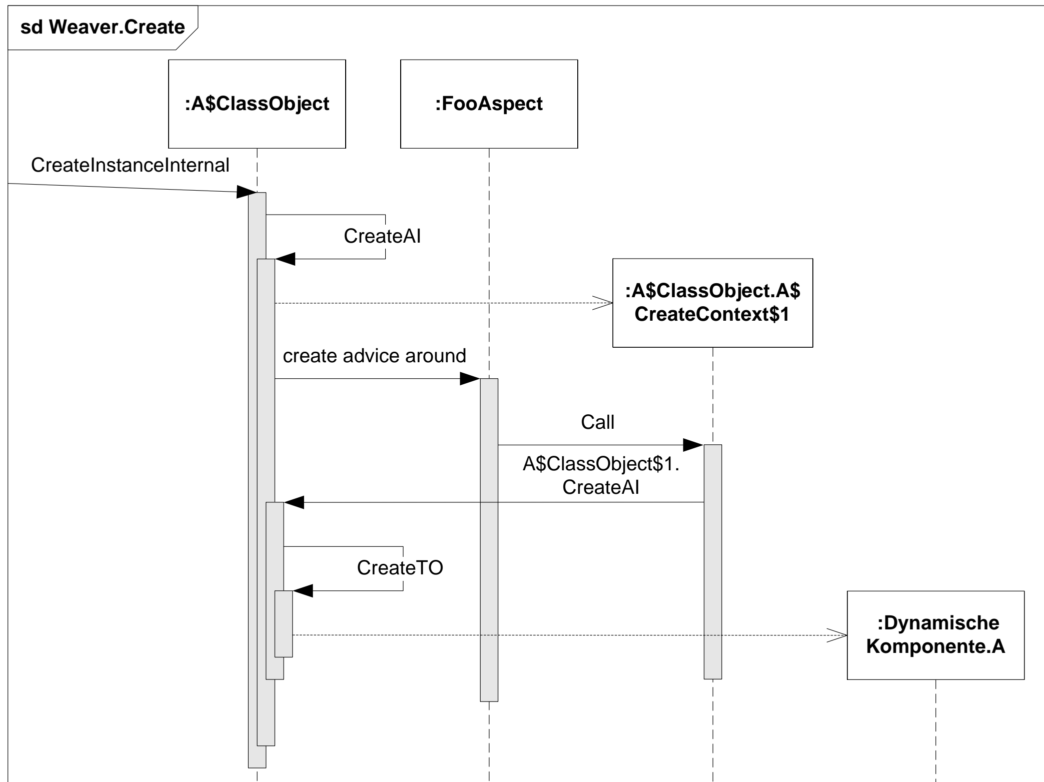


Abbildung 3.6: Ablauf einer Exemplarbildung

zum nächsten Klassenobjekt. Als Schlüssel dient jeweils der Typidentifizierer des Aspekts. Ist die Aspektliste abgearbeitet, so ist das Ergebnis das Klassenobjekt, das einen Stellvertreterexemplar für die angeforderte Klasse erzeugen kann. War die Suche nicht erfolgreich, muss der Verwebungsvorgang für den nicht gefundenen Aspekt angestoßen werden.

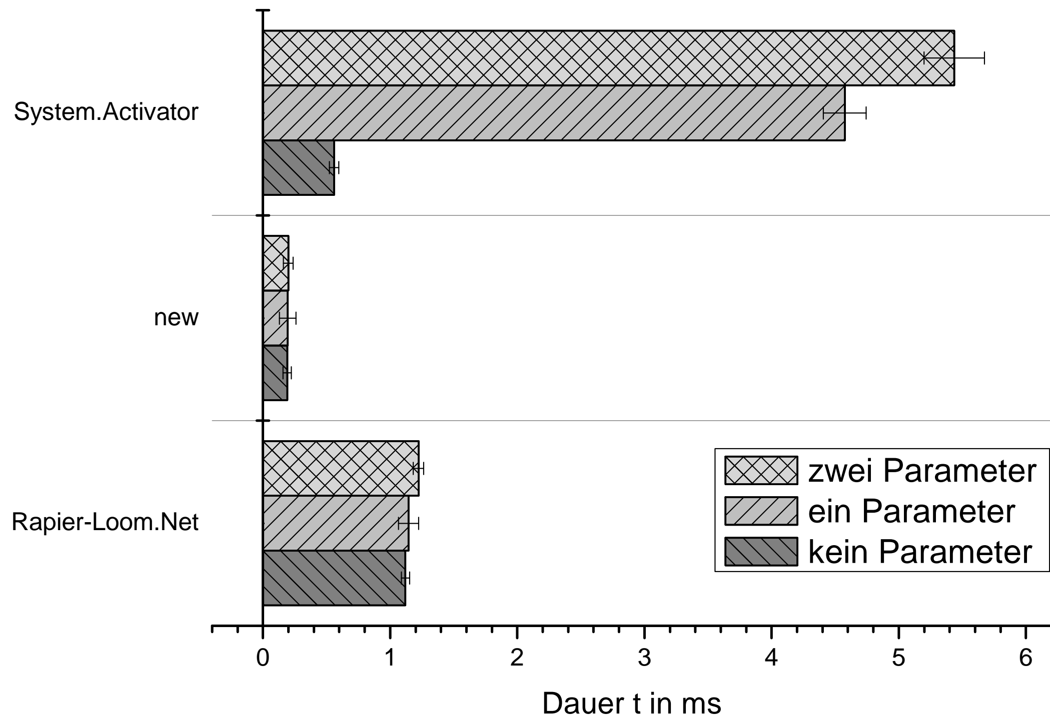
Obwohl die zahlreichen Indirektionen hohe Laufzeitkosten vermuten lassen, ist die tatsächliche Implementierung recht effizient. Die Methoden `CreateAI` und `CreateTO` bestehen ausschließlich aus dem Laden von Parameterwerten vom Stack und dem anschließenden Aufruf von Methoden. Diese Aufrufe können größtenteils als *Tail-Calls* [vgl. Microsoft Corporation u. a. 2006, Teil III, S. 29] implementiert werden. Das bedeutet, dass der Stackrahmen der Methode erhalten bleiben kann und der CIL-Kompiler ggf. sogar ein *Inlining* der Methoden vornehmen kann.

Es bleibt die Synchronisation der Datenstrukturen bei nebenläufigen Zugriffen, der Zugriff auf die Hash-Tabelle - dieser kann bei dynamisch hinzugefügten Aspekten mehrstufig ausfallen - und das Multidispatch der Konstruktoraufrufe. Unter Idealbedingungen, bei denen nur ein Konstruktor definiert wurde und kein Aspekt die Aufrufkette mit einem **create advice** unterbricht, liegt der Aufwand bei ca. dem Sechsfachen eines normalen **new**-Aufrufes (Abbildung 3.7). Hat der Konstruktor einen Parameter, ist die Exemplarerzeugung sogar schneller als die dynamische Erzeugung über die .NET-Klassenbibliothek. Das setzt sich bei steigender Anzahl der Parameter tendenziell fort.

3.5.3 Serialisierung von Objekten

Werden Exemplare von dynamisch verwobenen Klassen serialisiert, kann es vorkommen, dass bei der Deserialisierung diese Klassen noch nicht existieren. Das würde dazu führen, dass diese Exemplare nicht deserialisiert werden können.

Zur Behebung dieses Problems verwendet RAPIER-LOOM.NET sein eigenes Verwebungskonzept, um serialisierbare Klassen mit einem speziellen Serialisierungs-Aspekt zu verweben.



*Gemessen wurden 1000 Explarbildungen auf einem Intel Core Duo T5500, 1,66 GHz

Abbildung 3.7: Durchschnittliche Kosten für die Exemplarbildung

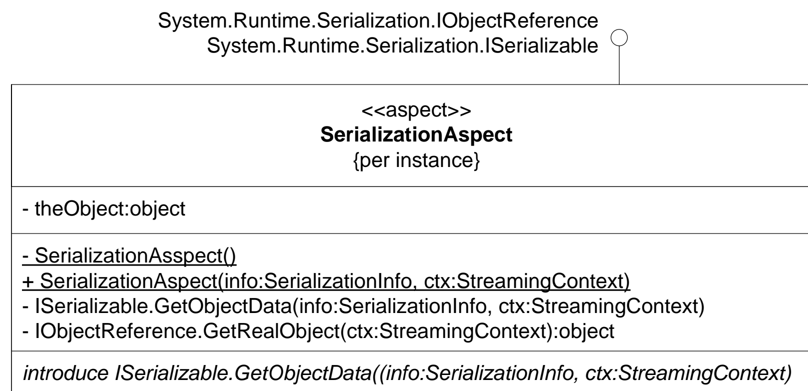


Abbildung 3.8: Der Serialisierungsaspekt der RAPIER-LOOM.NET Laufzeitumgebung

Die Deklaration dieses Aspekts ist in Abbildung 3.8 dargestellt.

Die Aufgabe des Aspekts besteht darin, Serialisierungsanforderungen an ein Exemplar abzufangen. Das geschieht durch Einführen des in der Klassenbibliothek definierten Interfaces `ISerialize` in die entsprechende Klasse. Hat die Klasse dieses Interface selbst bereits definiert, wird es überschrieben, da die Introduktion virtuell ist.

Eine Serialisierungsanforderung führt zum Aufruf von `GetObjectData` und somit zum Aufruf der Introduktion. Zu beachten ist, dass der Aspekt auch dieses Interface definieren muss, diese Implementation jedoch leer ist und nicht verwendet wird. Sie ist notwendig, damit später der Deserialisierungskonstruktor des Aspekts aufgerufen werden kann.

Im Serialisierungsstrom werden für ein Exemplar folgenden Daten abgelegt:

- Ein Typidentifizierer des zu serialisierenden Exemplars,
- Die serialisierten Daten aller Attribute des Exemplars (wobei der Serialisierungsalgorithmus dafür sorgt, dass Exemplare nicht mehrfach serialisiert werden)

Der Trick besteht nun darin, dass der Aspekt seinen eigenen Typidentifizierer in den Serialisierungsstrom (`info`) legt. Anschließend serialisiert er den Typidentifizierer der ursprünglichen Klasse (nicht den des Stellvertreters) und alle dynamisch zur Klasse hinzugefügten Aspekte. Aspekte, die aufgrund ihrer Überdeckung mit der ursprünglichen Klasse verwoben wurden, werden nicht serialisiert - aber der Zustand ihrer Joinpointvariablen.

Implementiert das Exemplar selbst auch das Interface `ISerializable`, ruft der Aspekt die Implementierung über den Aufrufkontext auf. Andernfalls serialisiert er alle übrigen Attribute des Exemplars.

Bei der Deserialisierung passiert Folgendes: Die Laufzeitumgebung liest vom Serialisierungsstrom den Typidentifizierer des Aspekts und erstellt über dessen Deserialisierungskonstruktor ein Exemplar. Im Deserialisierungskonstruktor kann nun mit den Informationen auf dem Serialisierungsstrom das ursprünglich verwobene Objekt wiederhergestellt werden.

Zuerst wird geprüft, ob die ursprüngliche Klasse das Interface `ISerializable` implementiert. Wenn dem so ist, muss ein verwobenes Exemplar mit Hilfe des Deserialisierungskonstruktors der ursprünglichen Klasse erzeugt werden. Wurde das Interface nicht implementiert, wird der Standardkonstruktor gewählt. Die Exemplarerzeugung erfolgt durch den Aufruf der `Weaver.Create` Fabrikmethode.

Die Laufzeitumgebung hält noch eine Referenz auf das Aspektexemplar, obwohl sie eigentlich an dieser Stelle das ursprüngliche Exemplar erwartet. Um dieses Problem zu lösen, muss das Interface `IObjectReference` implementiert werden. Bevor die Referenz auf das gerade deserialisierte Objekt im Objektgraphen aktualisiert wird, prüft die Laufzeitumgebung, ob das Objekt dieses Interface implementiert. In diesem Fall wird die Methode `GetRealObject` aufgerufen. Da der Serialisierungsaspekt das zuvor erzeugte verwobene Exemplar in `theObject` gespeichert hat, kann er die richtige Referenz zurückgeben.

Ein ähnliches Verfahren wird auch verwendet, um den Joinpoint-Kontext serialisierbar zu machen. Auf diese Weise können Methodenaufrufe eingefroren, persistiert und zu einem späteren Zeitpunkt oder an einem anderen Ort ausgeführt werden.

3.5.4 Der Verwebungsprozess

Der Verwebungsprozess besteht in Rapier-Loom.Net aus vier Phasen:

1. Überdeckung und selektierte Joinpoints ermitteln,
2. Verwebungsplan erstellen,
3. Abstrakten Verbindungscode definieren,
4. CIL synthetisieren.

Wird ein Exemplar einer Klasse angefordert, für das kein Klassenobjekt existiert, muss der Verwebungsvorgang gestartet werden. Zuerst ermittelt Rapier-Loom.Net über die Metadatenschnittstelle der `loom.dll` (siehe Abschnitt 3.4.6) die Joinpoints der Klasse. Das Metaobjekt für diese Klasse ist die `JoinPointCollection`. Dieses Objekt ist der Ausgangspunkt für die Verwebung.

Über die `JoinPointCollection` kann ein virtueller Baum der Metaobjekte erzeugt werden, der die folgende Struktur hat: Das Wurzelobjekt ist die `JoinPointCollection`, die die Klasse repräsentiert. Auf der nächsten Ebene befinden sich die Metaobjekte derjenigen Aspekte, die Methoden der Klasse oder die Klasse selbst überdecken. Unter einem Aspekt-Metaobjekt befinden sich diejenigen Joinpoints - repräsentiert durch Joinpoint-Metaobjekte - die von Advices oder Introduktionen des Aspektes selektiert wurden. Werden Interfaces eingeführt oder verwoben, so sind sie ebenfalls auf dieser Ebene zwischen die Joinpoints des

```
1  public class A
2  {
3    aspect MyAspect
4    {
5      public virtual int Foo()
6      {
7        call initialize void MyAspect.Init(
8          jpparam[JPContext, Context joinpoint],
9          jpparam[JPVariable, {Protected,Target,Virtual}, ref int foobar],
10         container[object[] args])
11     }
12
13     public virtual int Baz(double d)
14     {
15       call initialize void MyAspect.Init(
16         jpparam[JPContext, Context joinpoint],
17         jpparam[JPVariable, {Protected,Target,Virtual}, ref int foobar],
18         container[object[] args])
19
20       call after void MyAspect.After<T>(
21         jpparam[JPRetVal, object retval],
22         jpparam[JPVariable, {Protected,Target,Virtual}, ref int foobar],
23         generic[T p])
24     }
25
26     interface IBar
27     {
28       private int Bar(double d)
29       {
30         introduce static int MyAspect.Bar(
31           jpparam[JPTarget, object target],
32           simple[double d])
33       }
34     }
35   }
36 }
```

Abbildung 3.9: Generierte Baumstruktur für eine Verwebung

Interfaces und den Aspekt geschoben. Die Blätter des Baumes bilden die Aspektmethoden-Metaobjekte. Sie repräsentieren diejenigen Advices und Introduktionen, die jeweils den Joinpoint selektieren, unter dem sie sich in der Baumstruktur befinden. Joinpoint-Metaobjekte und Aspektmethoden-Metaobjekte können mehrfach in der Baumstruktur vorkommen.

Abbildung 3.9 zeigt eine solche Baumstruktur für das bekannte Beispiel aus Abbildung 3.1. Der Inhalt der Metadatenobjekte ist in Textform dargestellt. Untergeordnete Elemente sind jeweils in geschweifte Klammern eingeschlossen. In den Metadaten der Aspektmethoden sind zusätzlich die Metadaten ihrer Parametersignatur enthalten.

Die Metadaten der Parametersignatur lassen sich in vier Kategorien unterteilen: Die Kategorie `simple` entspricht einem einfachen Parameter, der direkt mit einem Parameter aus dem Joinpoint korrespondiert. Bei der Kategorie `generic` gibt es zwar einen korrespondierenden Parameter, der Parametertyp der Aspektmethode ist jedoch generisch - in der Definition der Aspektmethode ein Platzhalter für einen Parameter. Ein `container` ist ein Parameter, auf den eine Menge von Parametern aus dem Joinpoint übertragen wird - also ein Platzhalter für mehrere Parameter. Die letzte Kategorie ist `jpparam` und bezeichnet einen Joinpoint-Parameter.

Die Baumstruktur stellt gleichzeitig einen *Verwebungsplan* dar. Sie gibt die Reihenfolge an - beginnend mit dem niedrigsten Rang - in der die aufgeführten Aspekte verwoben werden sollen. Unterhalb der Joinpoints werden die Aspektmethoden - beginnend mit dem höchsten Rang - aufgeführt. Diese entgegengesetzte Reihenfolge hat technische Gründe: Zuerst verwobene Aspekte sind die letzten bei der Abarbeitung an einem Joinpoint - bei der Generierung einer Stellvertretermethode werden die ersten Advices auch zuerst ausgeführt.

Ein Verwebungsplan lässt sich dem Werkzeug `wp.exe` des LOOM.NET-Projektes für alle Klassen einer Assembly anzeigen. Das Werkzeug wird von der Kommandozeile gestartet und erwartet als Eingabe den Namen der anzuzeigenden Assembly. Dies ist eine einfache Methode, um zu überprüfen, ob die Advices und Introduktionen an den richtigen Stellen eingewoben werden.

Nachdem der Verwebungsplan feststeht, erstellt Rapier-Loom.Net einen abstrakten Bauplan für den Stellvertreter. Dieser Bauplan setzt sich aus verschiedenen Objekten zusammen: Ein `TypeBuilder` beschreibt, dass ein neuer Typ erzeugt werden muss. Der `TypeBuilder` kann wiederum `MethodBuilder` enthalten, deren Aufgabe es ist, neue Methoden zu erzeugen. Ein `MethodBuilder` hat wiederum verschiedene *Blöcke*, die `Codebausteine` aufnehmen können.

Codebausteine sind eine abstrakte Beschreibung für eine bestimmte Aktion. Für diese können sie Parameter als Eingabe entgegennehmen und ein Ergebnis als Ausgabe erzeugen. Das Ergebnis kann einem speziellen `Datenplatz`, dem `RetValDataSlot` zugewiesen werden. Ist dieser Datenplatz nicht vorhanden, obwohl der Codebaustein ein Ergebnis produziert, so wird es verworfen. Produziert der Codebaustein kein Ergebnis, obwohl der Datenplatz vorhanden ist, dann wird der Datenplatz mit **null** belegt.

Ein Codebaustein kann nicht direkt auf seine Eingabedaten zugreifen. Er benötigt hierzu einen *Parametercodebaustein*. Dieser ist letztendlich ein Umsetzer, der die Eingabedaten in eine Form bringt, die vom Codebaustein verarbeitet werden kann. Die Parametercodebausteine leiten sich direkt aus den Parameter-Metadaten der vorangegangenen Phase ab. Es gibt Bausteine,

- die auf einen Datenplatz referenzieren (`DataSlotParameter` für den Wert und `DataSlotParameterRef` für die Adresse eines Datenplatzes),
- die einen Parameter aus dem Joinpoint direkt weiterleiten (`Argument`),
- die einen Parameter aus dem Joinpoint an einen generischen Parameter weiterleiten (`GenericArgument`).

Jeder `MethodBuilder` hat fünf verschiedene Blöcke mit jeweils unterschiedlichen Eigenschaften: Codebausteine, die zu einem *before*-Block hinzugefügt werden, werden als erstes in die Methode eingebaut. Von diesen Codebausteinen erzeugte Ergebnisse werden verworfen.

Ein `method`-Block kann nur einen einzigen Codebaustein aufnehmen. Das Ergebnis dieses Codebausteins kann entweder dem `RetValDataSlot` zugewiesen werden oder dient als Rückgabewert der zu erzeugenden Methode.

Der Code, der von Codebausteinen im `afterreturning`-Block erzeugt wird, wird nur dann abgewickelt, wenn in den Blöcken `before` und `after` keine unbehandelte Ausnahme erzeugt wurde. Das Ergebnis jedes Codebausteins wird dem `RetValDataSlot` zugewiesen.

Ein `afterthrowing`-Block nimmt Codebausteine auf, deren Code nur dann aufgerufen werden soll, wenn in den vorangegangenen Blöcken `before`, `method` und `afterreturning` eine unbehandelte Ausnahme erzeugt wurde. Ist der spezielle Datenplatz `ExceptionDataSlot` vorhanden, wird er automatisch mit der Ausnahme belegt.

Schließlich werden Codebausteine im `after` Block am Ende der Methode eingebaut. Der von ihnen generierte Code wird immer abgewickelt, unabhängig davon, ob vorher ein Ergebnis oder eine Ausnahme erzeugt wurde.

Neben den speziellen Datenplätzen können im `TypeBuilder` und im `MethodBuilder` auch allgemeine Datenplätze definiert werden. Diese werden verwendet, um den Joinpoint-Kontext aufzunehmen, oder dienen als Container für Parameterlisten. Im Falle des Joinpoint-Kontextes referenzieren sie dann auf die mit einem `TypeBuilder` erzeugte Spezialisierung der Klasse

`Loom.Context`. Datenplätze können einem `TypeBuilder` oder einem `MethodBuilder` zugeordnet werden. Auf Datenplätze eines `TypeBuilders` können auch alle in ihm enthaltenen `MethodBuilder` zugreifen.

Abbildung 3.10 zeigt den Bauplan für den Verwebungsplan aus Abbildung 3.9. Die Darstellung ist hierarchisch in `TypeBuilder`, `MethodBuilder` und Codebausteine unterteilt. Hier kommen zwei verschiedene Codebausteine zum Einsatz: Die Abwicklung der zum Joinpoint gehörenden Methode (`methodcode`) und der Aufruf einer Aspektmethode (`aspectcall`). In diesem Bauplan wird außerdem deutlich, dass die Advice-Initialisierer in den `MethodBuilder` in den `before`-Block des Konstruktors verschoben wurden. Würde die Klasse `A` weitere Konstruktoren definieren, würden die Advice-Initialisierer dort auch aufgelistet sein.

RAPIER-LOOM.NET kann auf dem Bauplan verschiedene Optimierungen vornehmen. Eine ist die Konsolidierung von Datenplätzen, um Redundanzen zu vermeiden. Eine andere bewirkt, dass bestimmte Datenplätze - z. B. für den Rückgabewert - nur angelegt werden, wenn sie tatsächlich benötigt werden. Weitere Optimierungen finden in der letzten Phase der Verwebung statt - dem *Synthetisieren*.

Beim Synthetisieren wird der Bauplan in den Zielcode umgewandelt. Die Elemente des Bauplans sind jeweils für unterschiedliche Teile des Zielcodes verantwortlich: Ein `TypeBuilder` generiert eine Hülle für eine neue Klasse, ein `MethodBuilder` einen Rahmen für eine Methode, und die Codebausteine generieren CIL-Code. Datenplätze werden zu neuen Attributen, lokalen Variablen, oder sie werden abgebildet auf existierende Attribute.

In Abbildung 3.11 ist die CIL als die in einem Stellvertreter verwobene Methode `Baz` zu sehen. Die Zeilen 6 bis 9 sind verantwortlich für den Aufruf der Basisimplementierung und die Sicherung des Rückgabewertes in einer lokalen Variablen. Da der After-Advice sowohl nach einer regulären Rückkehr Abwicklung Basisimplementierung als auch nach dem Werfen einer Ausnahme ausgeführt werden soll, ist ersterer in einen *try-finnaly* Block eingefasst. Das wird durch Zeile 21 definiert; der `leave`-Befehl in Zeile 10 bestimmt die Aussprungadresse aus diesem Block. Bevor diese Adresse jedoch angesprungen wird, kommt der *finally*-Block (Zeile 11-18) zur Abwicklung. In diesem wird zuerst das Exemplar des verwobenen Aspektes auf den Stack geladen. Hinzu kommen anschließend die Argumente des After-Advices: der Rückgabewert, die Joinpointvariable `foobar` und der erste Parameter der verwobenen Methode.

3.6 Der statische Aspektweber Gripper-Loom.Net

Trotz vieler Optimierungen ist ein Problem der dynamischen Verwebung der erhöhte Ressourcenverbrauch zur Laufzeit; zum einen, da der Weber selbst zur Laufzeit aktiv ist und sowohl Speicher konsumiert als auch die CPU belastet, zum anderen, weil beim Verwebungsvorgang Kompromisse auf Grund von Beschränkungen gemacht werden müssen.

Eine dieser Beschränkungen ist beispielsweise, dass Metadaten in der CLI nicht direkt verändert werden können. Insbesondere lässt sich die Implementation einer Methode zur Laufzeit nicht ändern. GRIPPER-LOOM.NET wurde als Prototyp eines statischen Aspektwebers entwickelt, der auf Microsoft Phoenix [Microsoft Corporation 2008c] - einem Framework für Kompiler - aufsetzt und die Verwebung auf den binären Komponenten ausführen kann. Eine Besonderheit ist dabei, dass kompilierte und in Komponenten verpackte Aspekte zwischen RAPIER-LOOM.NET und GRIPPER-LOOM.NET frei ausgetauscht werden können.

Die Entwicklung des statischen Aspektwebers wurde finanziell durch die Firma Microsoft unterstützt [Xu 2007].

```
1 TypeBuilder[public class Loom.A:A]
2 {
3   d0:DataSlot[MyAspect]
4   d1:DataReference[protected int A.foobar]
5
6   t0:TypeBuilder[Context, int A.Foo()]
7   t1:TypeBuilder[Context, int A.Baz(double d)]
8   t2:TypeBuilder[Context, string Object.ToString()]
9   t3:TypeBuilder[Context, bool Object.Equals(object o)]
10  t4:TypeBuilder[Context, int Object.GetHashCode()]
11
12  MethodBuilder[public A()]
13  {
14    d2:Dataslot[JPContext, t0)
15    d3:Dataslot[container, {}]
16    d4:Dataslot[JPContext, t1]
17    d5:Dataslot[container, {double}]
18    d6:Dataslot[JPContext, t2]
19    d7:Dataslot[container, {object}]
20    d8:Dataslot[JPContext, t3]
21    d9:Dataslot[JPContext, t4]
22
23    before
24    {
25      aspectcall void MyAspect.Init(DataSlotParameter(d0), DataSlotParameter(d2),
          DataSlotParameterRef(d1), DataSlotParameter(d3))
26      aspectcall void MyAspect.Init(DataSlotParameter(d0), DataSlotParameter(d4),
          DataSlotParameterRef(d1), DataSlotParameter(d5))
27      aspectcall void MyAspect.Init(DataSlotParameter(d0), DataSlotParameter(d6),
          DataSlotParameterRef(d1), DataSlotParameter(d3))
28      aspectcall void MyAspect.Init(DataSlotParameter(d0), DataSlotParameter(d8),
          DataSlotParameterRef(d1), DataSlotParameter(d7))
29      aspectcall void MyAspect.Init(DataSlotParameter(d0), DataSlotParameter(d10),
          DataSlotParameterRef(d1), DataSlotParameter(d3))
30    }
31    method
32    {
33      methodcode A.A()
34    }
35    afterreturning {}
36    afterthrowing {}
37    after {}
38  }
39
40  MethodBuilder[public virtual int Baz()]
41  {
42    d10:RetValDataslot[JPRetVal, int]
43
44    before {}
45    method
46    {
47      methodcode int A.Baz(Argument(1)): => d10
48    }
49    afterreturning {}
50    afterthrowing {}
51    after
52    {
53      aspectcall void MyAspect.After<T>(DataSlotParameter(d0), DataSlotParameter(d10)
          , DataSlotParameterRef(d1), GenericArgument(T,1))
54    }
55  }
56
57  MethodBuilder[private int IBar.Bar(double d)]
58  {
59    aspectcall int MyAspect.Bar(Argument(0), Argument(1))
60  }
61 }
```

Abbildung 3.10: Abstrakter Bauplan für eine Verwebung

```
1 .method family hidebysig virtual instance int32 Baz(float64) cil managed
2 {
3     .maxstack 8
4     .locals init (
5         [0] int32 num)
6     L_0000: ldarg.0
7     L_0001: ldarg.1
8     L_0002: call instance int32 [Test]Test.A::Baz(float64)
9     L_0007: stloc.0
10    L_0008: leave L_0021
11    L_000d: ldarg.0
12    L_000e: ldfld class [Test]Test.MyAspect Test.A::__aspect0
13    L_0013: ldloc.0
14    L_0014: ldarg.0
15    L_0015: ldflda int32 [Test]Test.A::foobar
16    L_001a: ldarg.1
17    L_001b: call instance void [Test]Test.MyAspect::After<float64>(int32, int32&,
          !!0)
18    L_0020: endfinally
19    L_0021: ldloc.0
20    L_0022: ret
21    .try L_0000 to L_000d finally handler L_000d to L_0021
22 }
```

Abbildung 3.11: Der CIL-Code der Methode `Baz`

3.6.1 Microsoft Phoenix

Microsoft Phoenix besteht aus verschiedenen Komponenten, die zusammen den Rahmen für ein Kompiler-*Backend* bilden. Als *Backend* wird dabei der Teil eines Kompilers bezeichnet, der die Synthesephase implementiert. Das beginnt bei der Erzeugung eines Zwischencodes, geht über die Programmoptimierung bis schließlich zur Codegenerierung.

Die Ausgabe eines auf Phoenix basierenden Werkzeuges muss nicht notwendigerweise ein kompiliertes Programm sein. Die einzelnen Phasen lassen sich frei konfigurieren und können durch eigene Komponenten implementiert werden. Damit richtet sich die Ausgabe ausschließlich nach den Belangen des Phoenix-Programmierers und kann z. B. auch eine grafische Softwarevisualisierung sein.

Phoenix bietet bereits eine Menge fertiger Komponenten an, die in den einzelnen Phasen eingesetzt werden können. Dazu zählen unter anderem Phasen, um verschiedene Ausgangsformate zu lesen, wie z. B. das CIL-Assemblyformat oder einen abstrakten Syntaxbaum der Programmiersprache C#. Weiterhin existieren vorgefertigte Phasen zur Generierung verschiedener Ausgabeformate. Dazu zählt unter anderem das PE-Dateiformat oder auch das CIL-Assemblyformat. Die Aufgabe des Programmierer ist es, die Phasen über eine Phoenix-Schnittstelle zu konfigurieren und dann deren Abarbeitung anzustoßen.

3.6.2 Verweben von Assemblies

GRIPPER-LOOM.NET ist ein kommandozeilen-basiertes Werkzeug, das als Eingabe eine Assembly erwartet, die den Haupteintrittspunkt einer Anwendung definiert. Das ist in der Programmiersprache C# im Allgemeinen die Komponente mit der `main`-Methode. GRIPPER-LOOM.NET transformiert alle zu der Anwendung gehörenden Komponenten so, dass das durch Aspektannotation und Selektion von Joinpoints definierte Verhalten komplett abgebildet wird. Üblicherweise wird GRIPPER-LOOM.NET als zusätzlicher Schritt bei der Programmerzeugung aufgenommen. Es reiht sich direkt hinter den Kompiler und Linker ein, wenn diese ihre Arbeit erfolgreich abgeschlossen haben.

Im Wesentlichen implementiert GRIPPER-LOOM.NET unter Verwendung der Phoenix-Schnittstellen drei Phasen: Die erste Phase liest eine CIL-Assembly ein und wandelt sie in die Phoenix-Interne Zwischensprache (IR). Auf dieser Zwischensprache wird in der zweiten

Phase die Verwebung vorgenommen. GRIPPER-LOOM.NET bindet hierzu eine eigene Komponente - den `Weaver` - in die Phoenix Phasenliste ein. Schließlich wird als letzte Phase aus der Zwischensprache wieder eine CIL-Assembly generiert. Bei der ersten und letzten Phase verwendet GRIPPER-LOOM.NET bestehende Komponenten des Phoenix Frameworks.

Der `Weaver` analysiert den gesamten Code auf der Zwischensprache und sucht nach der Verwendung von Klassen, die von Aspekten überdeckt sind. Findet er eine solche Klasse, wird wie bei RAPIER-LOOM.NET ein Klassenstellvertreter dieser Klasse inklusive einer Klassenobjektklasse erzeugt. Implementiert wird beides in einer neuen Komponente. Diese Komponente erhält denselben Namen wie diejenige, aus der die originale Klasse stammt, jedoch mit dem Präfix „`Loom.`“.

In einem zweiten Schritt sucht der Weaver auf der Zwischensprache alle jene Stellen, an denen Exemplare von Klassen gebildet werden, für die bereits ein Klassenobjekt generiert wurde. Dieser Code wird nun so umgeschrieben, dass eine Exemplarerzeugung nicht durch direkten Aufruf des Konstruktors über den Opcode **newobj** erfolgt, sondern stattdessen ein Aufruf in das entsprechende Klassenobjekt erfolgt.

Das Verfahren, insbesondere die Erzeugung der Stellvertreterklassen ähnelt stark der Vorgehensweise in RAPIER-LOOM.NET; es gibt jedoch auch einige Unterschiede. Die Definition der Klassenobjektklassen ist in GRIPPER-LOOM.NET wesentlich einfacher, da hier kein Multidispatch implementiert werden muss. Der `Weaver` ist aus der statischen Analyse bereits in der Lage, direkt die richtige `CreateAI`-Methode (vgl. Abschnitt 3.5.2 und Abbildung 3.5) aufzurufen. Der Umweg über `Weaver.Create` und `CreateInstanceInternal` entfällt.

3.6.3 Bewertung des Aspektwebers

Obwohl GRIPPER-LOOM.NET als Prototyp entwickelt wurde, sind die Ergebnisse vielversprechend. Insbesondere die Binärkompatibilität zu RAPIER-LOOM.NET-Aspekten ist ein großer Vorteil, da die Anwendungsbeispiele mit wenigen Modifikationen auch unter GRIPPER-LOOM.NET laufen. Die Modifikationen bestanden ausschließlich darin, die RAPIER-LOOM.NET-Fabrikmethode gegen einen normalen **new** Aufruf auszutauschen - was die Lesbarkeit des Quelltextes noch verbessert und intuitiver für den Programmierer ist.

Der größte Teil der Quelltextbasis für GRIPPER-LOOM.NET stammt derzeit aus dem RAPIER-LOOM.NET-Projekt. So konnte z. B. die Generierung der Stellvertreter und der Klassenobjekte mit wenigen Modifikationen übernommen werden. Zukünftige Weiterentwicklungen könnten hier ansetzen und weitere Funktionen des Phoenix-Frameworks nutzen um noch effizienteren Code zu erzeugen.

3.7 Ein Vergleich beider Aspektweber

In diesem Abschnitt sollen die beiden Verwebungskonzepte anhand von Messungen der Laufzeitkosten verglichen werden. Als Programmbasis wurde hier der Linpack-Algorithms [Dongarra u. a. 2003; Shudo 2005] herangezogen. Linpack ist eine Programmbibliothek zum Lösen linearer Gleichungssysteme, die zur Leistungsmessung von Rechnern verwendet wird. Die einzelnen Methoden stellen einen guten Mix zwischen sehr einfachen, nur wenige Anweisungen enthaltenden und komplexen Methoden mit verschachtelten Schleifen und Verzweigungen dar. Für den Vergleich werden die Methoden `Dscal`, `Abs`, `Idamax` und `Daxpy` sowie die komplette `Linpack`-Klasse aus [Shudo 2005] mit Aspekten annotiert.

Die Methode `Dscal` skaliert einen Vektor um einen konstanten Faktor, `Abs` ermittelt den absoluten Betrag einer Gleitkommazahl, `Idamax` ermittelt den Index des Maximalwertes in einem übergebenen Feld und `Daxpy` führt eine konstante Wiederholung von Vektoradditionen durch. Diese Methoden enthalten keine Aufrufe anderer Methoden.

verwobene Methoden	verwobene Annahme	Prüfungen pro Durchlauf	Anteil
keine		0	
Dscal	`[Requires("n>0")]`	499	0,26%
Abs	`[Ensures("(RESULT==OLD␣d)␣||` `(RESULT== -(OLD d))")]`	1015	0,18%
Idamax	`[Requires("n>=0")]` `[Ensures("RESULT>=0")]`	499	0,41%
Daxpy	`[Requires("n>=0")]`	125749	64,59%
Alle	`[Invariant("total>=0")]`	255538	100%

Abbildung 3.12: Getestete Annahmen der Linpack-Methoden

Zur Annotation dieser Methoden werden die im Abschnitt 5.2 noch ausführlicher dargestellten Aspekte der *Design-by-Contract*-Lösung verwendet. Diese Aspekte erlauben es, Vor- und Nachbedingungen für einzelne Methoden sowie Invarianten für eine gesamte Klasse nach dem Konzept von [Meyer 1992] zu definieren.

Entsprechend der Abbildung 3.12 sind die Methoden mit Vor- oder Nachbedingungen mit unterschiedlicher Komplexität annotiert, beziehungsweise die gesamte Klasse mit einer Invariante versehen. Jede Zeile der Tabelle entspricht einem Testszenario. Als Referenz wird der Algorithmus ohne Annotation vermessen. Dabei sind jeweils die Initialisierung und die Berechnungsdauer von Interesse.

Als Initialisierung gilt hierbei der Zeitraum, der benötigt wird, um ein Exemplar der Linpack-Klasse zu erzeugen. Annotierte Exemplare haben hier bei der erstmaligen Initialisierung einen erheblichen Mehraufwand, da zuerst eine dem Ausdruck entsprechende Evaluationsklasse generiert werden muss. Genauer wird der Vorgang in Abschnitt 5.2 erklärt.

Beim Methodenaufruf fällt der zusätzliche Aufwand für die Prüfung der Annahme an. Die Tabelle in Abbildung 3.12 zeigt, wie oft diese im entsprechenden Szenario bei einem Durchlauf des Linpacks geprüft werden müssen. Die letzte Spalte gibt an, welchen Anteil die annotierte Methode an der Gesamtberechnung hat. Das ist der Anteil, den der Abwickler tatsächlich in dieser Methode verbringt. Ruft diese Methode weitere Methoden auf, gehört die dort verbrachte Zeit nicht zum Anteil der aufrufenden Methode.

Da auch die Art der Verwebung (dynamisch oder statisch) Auswirkung auf die Ergebnisse hat, wurden einzelne Szenarien sowohl mit Rapier-Loom.Net als auch mit Gripper-Loom.Net gemessen. Zu erwarten ist, dass der statische Weber einen geringeren Aufwand bei der Exemplarerzeugung hat. Bei der eigentlichen Berechnung dürften sich keine Unterschiede ergeben.

Wie aus Abbildung 3.13 ersichtlich wird, liegt der Aufwand für die erstmalige Erzeugung von Exemplaren, die mit Annahmen versehen wurden, deutlich über Exemplaren, bei denen keine Annahmen geprüft werden. Der Aufwand steigt auch mit der Anzahl der zu prüfenden Annahmen (Methode `Idamax`). Vergleicht man die Effizienz der statisch mit Gripper-Loom.Net verwobenen Variante mit der des dynamischen Webers Rapier-Loom.Net, so fällt auf, dass letzterer etwas mehr Zeit für die Exemplarerzeugung benötigt. Die Differenz bei der erstmaligen Erzeugung ergibt sich aus dem benötigten Aufwand für die Verwebung der Aspekte zur Laufzeit.

Bei der Erzeugung weiterer Exemplare (Abbildung 3.14) ist die Differenz zum Referenzwert (`new`) - insbesondere bei Gripper-Loom.Net - mit einem Faktor drei bis fünf vergleichsweise gering. Der Unterschied zu Rapier-Loom.Net (Faktor 25) liegt vor allem darin begründet, dass die Exemplarerzeugung des dynamischen Webers über die Fabrikmethode des Webers erfolgen muss. Diese muss erst das passende Klassenobjekt und dann dynamisch den richtigen Konstruktor suchen, bevor das Exemplar erzeugt werden kann. Die entsprechenden

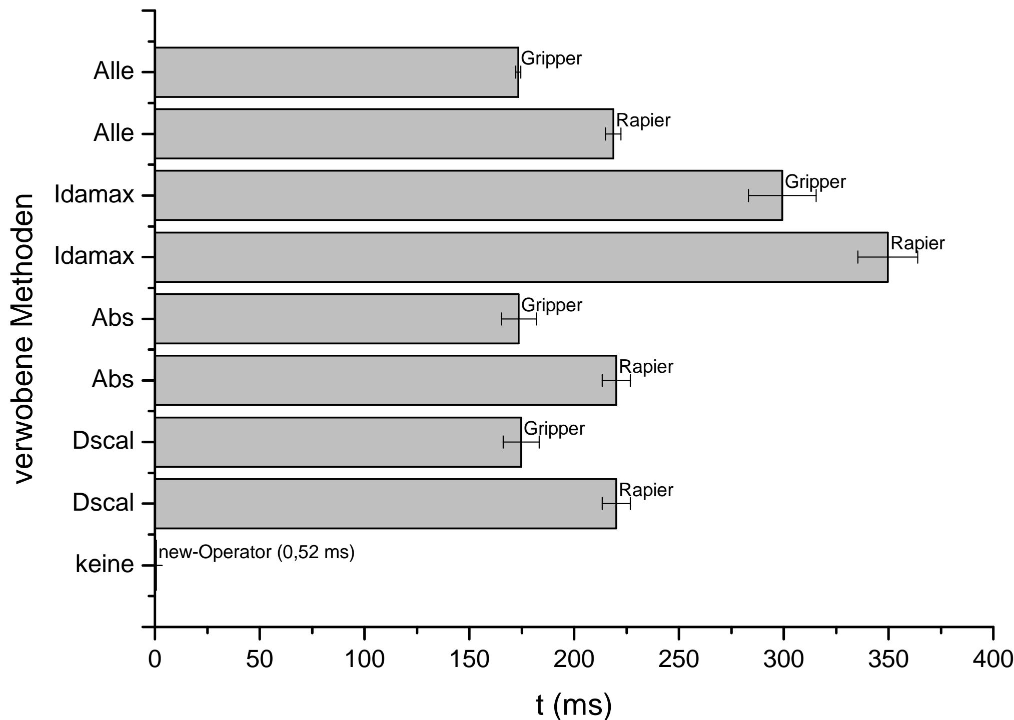


Abbildung 3.13: Durchschnittliche Erzeugungsdauer des ersten Exemplars

Datenstrukturen müssen weiterhin vor nebenläufigen Zugriffen geschützt werden. Demgegenüber steht beim Referenzwert ein einziger CIL-Befehl - `newobj` - zur Exemplarerzeugung.

Auffällig bei den Messungen mit Gripper-Loom.Net ist die Methode `Idamax`. Diese ist mit insgesamt zwei Aspekten verwoben. Jeder Aspekt (`Requires` und `Ensures`) verwebt die Exemplarerzeugung mit einem Advice, insgesamt also zwei Verwebungen mit Advices. Demgegenüber ist in allen anderen Testszenarien (`Alle`, `Abs`, `Dscal`) jeweils nur ein Advice in die Exemplarerzeugung eingewoben. Trotzdem benötigen die Letztgenannten knapp die doppelte Zeit, obwohl der Aufwand offensichtlich geringer ist und es sich um dieselben Aspekte handelt.

Als mögliche Erklärung hierfür kommt nur der *Just-in-time-Kompiler* der .Net-Laufzeitumgebung in Frage. Da die Advices selbst nur wenige Anweisungen enthalten, könnte der Kompiler im `Idamax`-Fall Optimierungen vorgenommen haben, die in den anderen Fällen nicht erfolgt sind. Möglich ist, dass der Kompiler ein *Inlining* [Detlefs und Agesen 1999] der Advices durchgeführt hat. Ein ganz ähnliches Phänomen beschreibt Haupt [Haupt 2005, S. 166] bei den Messungen des *Steam-Loom* Aspektwebers unter Java.

Der zusätzliche Aufwand bei der Berechnung ist in Abbildung 3.15 dargestellt. Hier zeigt sich, dass ein signifikanter Mehraufwand erst bei denjenigen Methoden entsteht, die während der Berechnung häufig aufgerufen werden (`Daxpy` bzw. die Prüfung der Invariante). Die Szenarien `Idamax`, `Abs` und `Dscal` hingegen weisen keinen messbaren Mehraufwand gegenüber dem Szenario ohne Aspekte (`keine`) auf.

Das Szenario mit dem größten Mehraufwand (die Invariante) wurde sowohl mit dem dynamischen als auch mit dem statischen Weber gemessen. Hier zeigt sich, dass sich im Rahmen der Messungenauigkeit keine Unterschiede zwischen beiden Aspektwebern feststellen lassen. Auch bewegt sich der zusätzliche Aufwand mit einem Faktor von knapp 1,3 gegenüber dem Szenario ohne Aspekte in einem durchaus akzeptablen Rahmen.

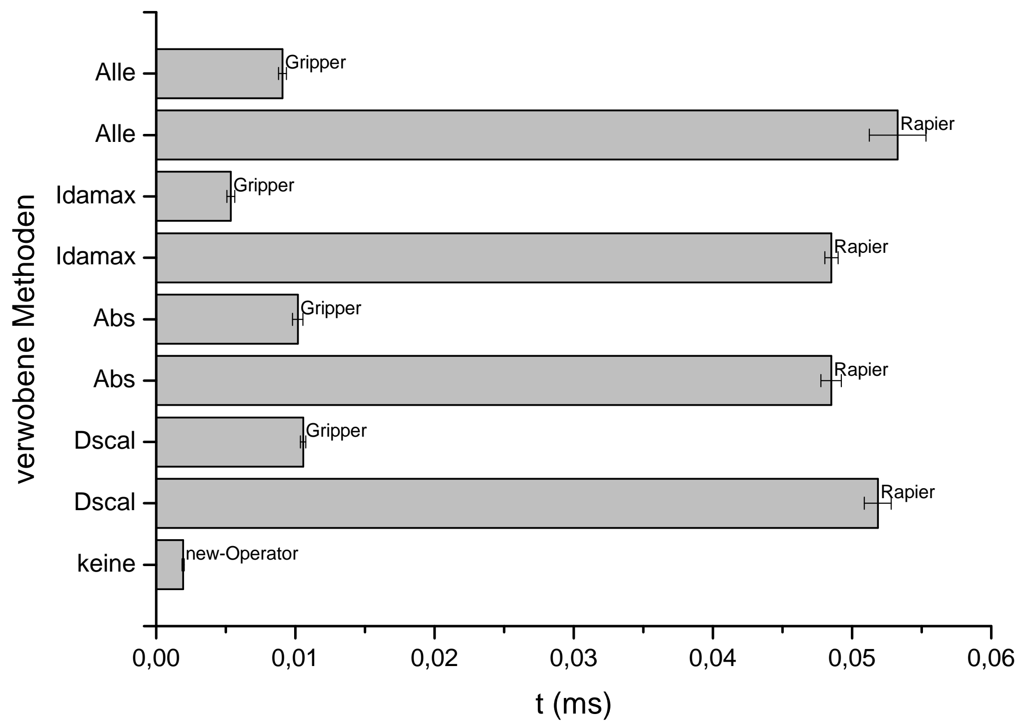


Abbildung 3.14: Durchschnittliche Erzeugungsdauer des n+1'ten Exemplars

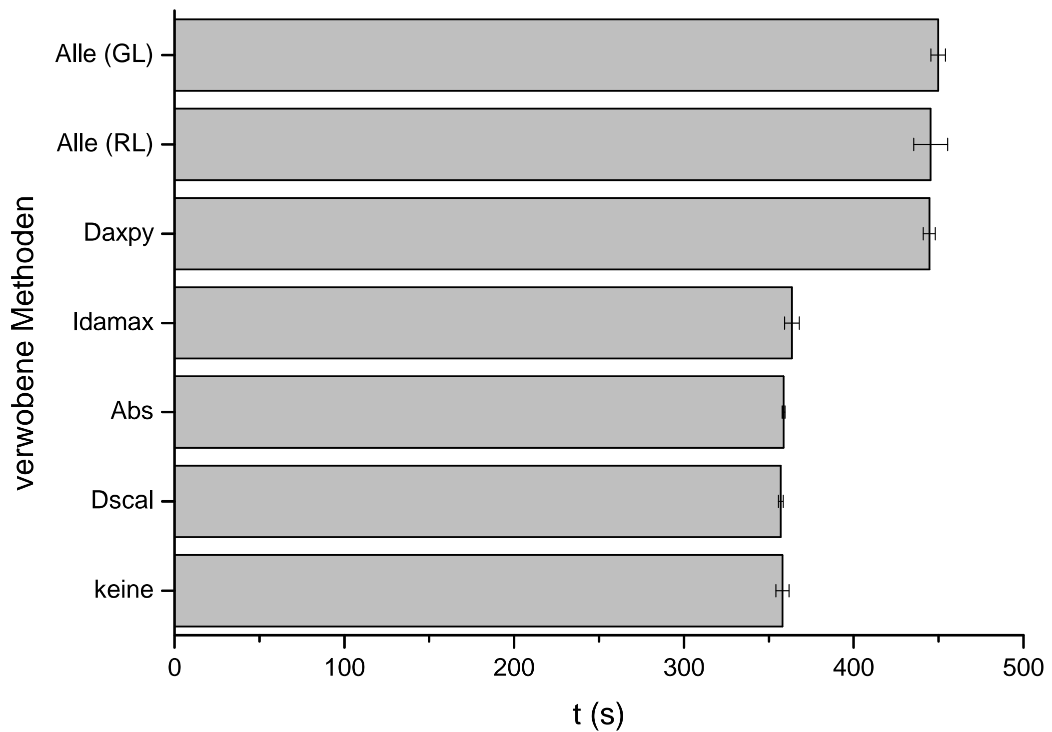


Abbildung 3.15: Zusätzlicher Aufwand für die Überprüfung der Annahmen

3.8 Zusammenfassung

In diesem Kapitel wurden zwei Implementationen, Rapier- und Gripper-Loom.Net, für die Loom.Net-Spracherweiterung vorgestellt. Es wurde dargelegt, dass Aspekte - wie sie in Kapitel 2, definiert sind - mit einem bestehenden Industriestandard, der *Common Language Infrastructure* (CLI) beschrieben werden können. So beschriebene Aspekte sind binärkompatibel zu den beiden vorgestellten Aspektwebern.

Bei Rapier-Loom.Net handelt es sich um einen dynamischen Aspektweber, der zur Laufzeit die Überdeckung der Aspekte ermittelt und die überdeckten Klassen bei der Anforderung des ersten Exemplars verwebt. Das dynamische Weben hat den Vorteil, dass die Entscheidung, ob Klassen von einem Aspekt überdeckt werden sollen, erst bei der Exemplarbildung getroffen werden. Das eröffnet für den Programmierer neue Möglichkeiten beim Systementwurf. Exemplare können jetzt in Abhängigkeit von den Umgebungseigenschaften dynamisch mit zusätzlichem Verhalten versehen werden.

In einem Teil des Kapitels wurden ausgewählte Mechanismen und Optimierungen vorgestellt, die in den Aspektwebern implementiert wurden. Es wurde beschrieben, wie Rapier-Loom.Net über indizierte Zwischenspeicher und generierte Klassenobjekte den Zeitaufwand für die Exemplarbildung auf ein ähnliches Niveau wie herkömmliche **new**-Aufrufe senken kann. Damit ist die Exemplarbildung in den meisten Fällen sogar schneller als die Verwendung entsprechender Reflektionsaufrufe.

Der Ablauf der Verwebung über die verschiedenen Stufen, angefangen beim Verwebungsplan über die abstrakte Implementierung des Verbindungscodes bis zur Synthetisierung der Zwischensprache CIL und dem Emittieren der Metadaten, wurde an einem Beispiel erörtert. Auch hier wurden verschiedene Optimierungen umgesetzt, einen möglichst effizienten Code mit geringen Laufzeitkosten zu erhalten. Das bestätigen unabhängige Messungen von [Kostian 2005, S. 37 ff.] und [Sakuda 2004].

Einige Anforderungen bei der Entwicklung des Rapier-Loom.Net-Aspektwebers wurden selbst mit Aspekten realisiert. Ein Beispiel hierfür ist die Serialisierung von mit Aspekten verwobenen Exemplaren. Ein spezieller Serialisierungsaspekt sorgt für die Integrität solcher Exemplare nach der Deserialisierung.

Schließlich wurde mit Gripper-Loom.Net ein weiterer Aspektweber des Autors vorgestellt. Hierbei handelt es sich um einen statischen Aspektweber, der die Verwebung der Aspekte direkt nach der Kompilierung vornimmt. Mit Microsoft Phoenix wurde zur Realisierung ein bestehendes Kompilerframework verwendet. Bei der statischen Verwebung entfällt die Verwendung einer Fabrikmethode, sodass die Verwebung von Klassen transparent erfolgt.

4 Entwurfsmuster

In diesem Kapitel sollen Entwurfsmuster für die aspektorientierte Softwareentwicklung mit LOOM vorgestellt werden. Entwurfsmuster sind bewährte, wiederverwendbare Vorlagen zur Lösung einer bestimmten Klasse von wiederkehrenden Entwurfsproblemen. Bekannt wurde der Begriff Entwurfsmuster durch [Gamma u. a. 1995], wo 23 typische Muster der objektorientierten Softwareentwicklung beschrieben werden. Sie sind üblicherweise unabhängig von der verwendeten Programmiersprache, obwohl es durchaus Ausnahmen gibt, bei denen die eine oder andere Programmiersprache eine bessere Eignung findet.

Die Entscheidung, ob ein Entwurfsproblem mit einem bekannten Muster implementiert werden soll, wird in der Entwurfsphase der Software getroffen. Sie hat Auswirkungen auf die Klassenstruktur und die Interaktion der Klassen untereinander. Ist eine solche Entscheidung einmal getroffen, lässt sie sich oft schwer rückgängig machen. Das mustertypische Verhalten lässt sich nicht einfach „abschalten", sondern muss aufwendig aus dem Quelltext entfernt werden. Als Beispiel sei die Anwendung des Einzelstückmusters genannt, bei dem man sich im Nachhinein doch für eine Mehrfacherzeugung eines Exemplars entscheidet.

LOOM kann hier neue Wege aufzeigen. Es stellt sich heraus, dass die Belange, die durch viele der bekannten Muster implementiert werden, aufgrund der zugrundeliegenden Formalismen der verwendeten Programmiersprache überschneidend implementiert werden müssen. Gelingt es, die Logik der Muster in Aspekte auszulagern, ergeben sich daraus einige Vorteile. Zum einen kann sie zu jedem Zeitpunkt der Entwicklung eingefügt oder entfernt werden, zum anderen lassen sie sich mitunter so abstrakt formulieren, dass sie als eigenständige Komponenten vielfältig einsetzbar sind. Die implementierte Logik der Muster lässt sich damit kompakter implementieren, ist leichter anzupassen und besser zu warten.

Für viele der von [Gamma u. a. 1995] beschriebenen Entwurfsmuster gibt es bereits Ansätze, wie diese mit aspektorientierten Mitteln - speziell AspectJ - umgesetzt werden können. Beispiele sind [Hachani und Bardou 2002; Hannemann 2005; Hannemann und Kiczales 2002; Laddad 2003]. Hier sollen vor allem die Besonderheiten von LOOM und die Vorteile, die dessen explizites Joinpoint-Modell bietet, herausgearbeitet werden. Inwieweit sich daraus Vorteile ergeben, soll mit einer in diesem Kapitel vorgestellten Metrik gemessen werden.

Neben den klassischen, aus der Objektorientierung bekannten Mustern werden in diesem Kapitel auch Entwurfsmuster besprochen, die sich so nur mit aspektorientierten Mitteln und speziell mit LOOM.NET realisieren lassen. Ein Beispiel ist der *Ereignis-Abonnent*. Für diese Muster gibt es keine objektorientierte Entsprechung. Abbildung 4.1 zeigt alle vom Autor auf die LOOM-Konzepte adaptierten bzw. neu entwickelten Muster im Überblick.

4.1 Eine Metrik zur Bewertung der Entwurfsmuster

Um die Vorteile der LOOM-Konzepte objektiv bewerten zu können, wurde eine Metrik entwickelt, die die Kategorien *Kopplung*, *Modularisierung*, *Redundanz* und *Implementationsaufwand* berücksichtigt. Die Bewertung erfolgt jeweils auf einer Skala von eins bis drei, wobei eine höhere Zahl stets einen besseren Wert bedeutet.

Bei der *Kopplung* wird eine Aussage darüber getroffen, wie groß jeweils die Abhängigkeiten des Quelltextes der Belange des Musters vom restlichen Quelltext ist. Sie wird in drei

Muster	Typ	adaptiert aus OOP
Einzelstück	Erzeugung	ja
Dekorierer	Struktur	ja
Adapter	Struktur	ja
Klassenkomposition	Struktur	nein
Beobachter	Verhalten	ja
Besucher	Verhalten	ja
Stellvertreter	Struktur	ja
Ereignisabonnent	Struktur	nein

Abbildung 4.1: Überblick der vorgestellten Muster

Stufen bewertet: *1=stark*, *2=mittel*, *3=lose*. Eine starke Kopplung bedeutet, dass der musterspezifische Quelltext den fachspezifischen Quelltext stark durchsetzt und nicht ohne weiteres herausgelöst werden kann. Eine mittlere Kopplung liegt vor, wenn der Quelltext des Musters zwar für jeden Anwendungfall angepasst werden muss, der fachspezifische Quelltext jedoch größtenteils unberührt bleibt. Insbesondere enthält die Implementation des Anwendungsfalls keine Referenzen auf den musterspezifischen Quelltext. Bei einer losen Kopplung muss auch der musterspezifische Quelltext nicht mehr angepasst werden.

Muster aus der objektorientierten Programmierung haben üblicherweise eine hohe Kopplung, denn sie werden ausschließlich als Vorlage definiert, die für den speziellen Anwendungsfall angepasst werden muss [Gamma u. a. 1995, S. 3]. Werden diese Muster mit den LOOM-Konzepten neu implementiert, verringert sich die Kopplung mitunter so weit, dass der musterspezifische Quelltext völlig unabhängig vom restlichen Quelltext ist. Diese Unabhängigkeit ist ein wünschenswertes Ziel, denn sie stellt gleichzeitig einen Indikator für die Wiederverwendbarkeit des Quelltextes dar: Ist der Quelltext unabhängig, so kann er für die unterschiedlichsten Szenarien und Anwendungsfälle verwendet werden, ohne dass er angepasst werden muss. Im besten Fall kann er sogar in Binärform als Komponente zur Verfügung gestellt werden.

In eine ähnliche Richtung zielt das zweite Maß, die *Modularisierung*. Eine *hohe* Modularisierung bedeutet, dass die Belange des Musters und die faschspezifischen Belange in jeweils eigene Module implementiert werden können. Bei den objektorientierten Mustern wird oft die musterspezifische Logik direkt in die fachspezifischen Klassen eingebaut. Es fällt dann oft schwer, auseinander zu halten, welcher Quelltext zu welchem Belang gehört. In diesem Fall ist die Modularisierung *schlecht*. Entsprechend werden diese drei Stufen den Werten *3=hoch, 2=mäßig, 1=schlecht* zugeordnet.

Die letzten beiden Punkte betreffen die *Redundanz* und den *Implementationsaufwand* des Quelltextes. Auch hier erfolgt eine Einordnung in *1=hoch*, *2=mittel* und *3=gering*. Eine *hohe* Redundanz bedeutet, dass der Quelltext an verschiedenen Stellen mehrfach implementiert wurde. Der *Implementationsaufwand* wird dahingehend bewertet, wie viel Quelltextzeilen grundsätzlich notwendig sind, um das Muster für eine Anwendungsfall zu implementieren. Ein *1=hoher* Implementationsaufwand bedeutet, dass viel Quelltext benötigt wird, um das Muster verwenden zu können. Demgegenüber steht ein *3=geringer* für wenige Zeilen. Der Implementationsaufwand ist allerdings stets im Vergleich zu anderen Lösungen zu betrachten.

Ein geringer Implementationsaufwand bedeutet nicht notwendigerweise, dass eine Implementation auch *insgesamt* wenig Quelltext benötigt. Weist diese eine hohe Redundanz auf, sind die Kosten zu Beginn zwar gering, für jeden neuen Anwendungsfall kommen aber unter Umständen viel mehr Quelltextzeilen hinzu als bei einer Lösung mit hohen Implementationsaufwand und geringer Redundanz.

Die Bewertung wird für jedes Muster separat vorgenommen. Wenn das Muster eine Entsprechung in der objektorientierten Programmierung hat [Gamma u. a. 1995], wird diese zum Vergleich herangezogen. Oft gibt es aber auch andere Lösungen für das musterspezifische

Problem. In diesem Fall werden diese Lösungen auch diskutiert und in den Vergleich mit einbezogen.

4.2 Das Einzelstückmuster

Das Einzelstückmuster [Gamma u. a. 1995, S. 127 ff.] (auch *Singleton*) gehört zu der Kategorie *Erzeugungsmuster*. Es stellt sicher, dass von einer Klasse nur ein einziges Exemplar erzeugt werden kann.

Die objektorientierte Implementation des Einzelstückmusters für eine Klasse enthält üblicherweise eine statische Variable, die das einzige Exemplar dieser Klasse referenziert. Oft existiert zusätzlich ein privater Konstruktor, um zu verhindern, dass Exemplare außerhalb des Zuständigkeitsbereichs der Klasse angelegt werden.

Es gibt unterschiedliche Varianten des Musters, beispielsweise kann der Zugriff auf das Exemplar der Klasse direkt über die statische Variable erfolgen. Diese ist dann als öffentliche - ausschließlich lesbare - Variable deklariert und mit dem Konstruktoraufruf initialisiert. Eine andere Variante sieht eine explizite Erzeugermethode vor. Ist noch kein Exemplar vorhanden, wird es erzeugt und gespeichert. Der Rückgabewert dieser Methode ist das Exemplar.

Das Problem des objektorientierten Einzelstückmusters ist die vom üblichen Paradigma abweichende Erzeugung von Exemplaren. Anstelle des `new`-Operators muss immer die Zugriffsmethode (oder eine öffentlich lesbare Variable) für das Exemplar der Klasse verwendet werden. Der Quelltext ist daher mit der Einzelstückmuster-typischen Exemplarbildung durchsetzt.

Eine weiteres, viel schwerwiegenderes Problem ist die Ableitung von Einzelstückklassen. Die Programmlogik für die Verwaltung des Einzelstücks ist nicht vererbbar, da für jede Klasse in der Vererbungshierarchie ein eigenes Einzelstück-Exemplar verwaltet werden muss. Zugriffsmethode und Variable sind jedoch statisch und damit an eine Klasse gebunden. [Gamma u. a. 1995, S. 130 ff.] schlagen hierzu die Verwendung einer Registratur vor, bei der sich die Derivate des Einzelstücks anmelden. Bei jeder Exemplaranforderung wird dann in der Registratur nach dem Exemplar des entsprechenden Typs gesucht. Bei vielen unterschiedlichen Klassen ist das sehr aufwendig.

Eine andere Alternative zur Registratur ist die Verwendung einer generischen Einzelstückklasse. Diese hat einen Typparameter, der zur Definition der entsprechenden statischen Exemplarvariablen verwendet wird. Es wird von dieser Klasse abgeleitet, indem als Typparameter der Typ der abgeleiteten Klasse übergeben wird. Somit existiert für jede direkte Ableitung von dieser Einzelstückklasse eine eigene statische Exemplarvariable. Weitere Ableitungen eines Derivats sind wiederum nicht möglich.

4.2.1 Die Struktur des LOOM-Einzelstückmusters

Abbildung 4.2 zeigt die Struktur des Einzelstückmusters. Die Klasse `Singleton` wird durch Annotation mit dem `Einzelstück`-Aspekt als Einzelstückklasse deklariert. Ist eine Klasse ein Einzelstück, sind auch alle Ableitungen der Klasse automatisch Einzelstücke. Ist dieses Verhalten nicht erwünscht, kann es mit der Aspekteigenschaft **noninherited** abgeschaltet werden.

Annotierte Klassen dürfen ausschließlich parameterlose Konstruktoren definieren. Wäre das nicht verboten, könnte das einzige Exemplar dieser Klasse potentiell von vielen Stellen mit unterschiedlichen Parametern angefordert werden. Zur Laufzeit wäre nicht klar, mit welchen Parametern das einzige Exemplar tatsächlich gebildet wurde. Daher sorgt der **error**-Advice dafür, dass Konstruktoren mit Parametern zu einem Verwebungsfehler führen.

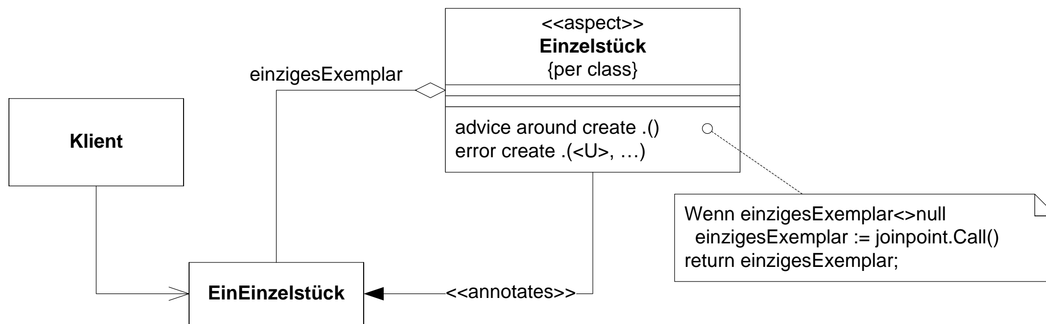


Abbildung 4.2: Das LOOM-Einzelstückmuster

```
1  [CreateAspect(Per.Class)]
2  public class Einzelstück : AspectAttribute
3  {
4      private object einzigesExemplar = null;
5
6      [Create(Advice.Around)]
7      public T Create<T>([JPContext] Context ctx)
8      {
9          if (einzigesExemplar == null) einzigesExemplar = ctx.Call();
10         return (T)einzigesExemplar;
11     }
12
13     [Error(JoinPoint.Create, "Einzelstückkonstruktoren␣müssen␣parameterlos␣sein")]
14     [IncludeAll]
15     public abstract void Error<T>(T p1, params object[] pn);
16 }
```

Listing 4.1: Der Aspekt zur Annotation von Einzelstücken

4.2.2 Beispiel

Listing 4.1 zeigt eine mögliche Implementierung des Aspekts zur Deklaration von Einzelstückklassen. Der Aspekt verwebt in seinem **create**-Advice (Zeile 6-11) die Exemplarbildung aller mit ihm annotierten Klassen. Die private Variable `einzigesExemplar` hält dabei eine Referenz auf das Einzelstück-Exemplar. Ist dieses beim Aufruf von `Create` noch nicht vorhanden, wird es einmalig durch Aufruf des originalen Konstruktors über den Joinpoint-Kontext (`ctx.Call()`, Zeile 11) angelegt. Dass für jede Klasse genau ein Exemplar angelegt wird, stellt die Aspekteigenschaft **per class** (Zeile 1) sicher.

Der Einzelstückaspekt wird durch die Annotation derjenigen Klassen verwendet, von denen jeweils nur ein Exemplar existieren soll:

```
[Singleton]
public class A
{ ... }
```

4.2.3 Vergleich zu anderen Lösungen

[Hannemann 2005; Hannemann und Kiczales 2003] lösen das Einzelstück-Problem über einen abstrakten Aspekt, der für jede Einzelstückklasse konkretisiert werden muss. Der abstrakte Aspekt implementiert hierbei die Semantik des Musters, das Abfangen der `new` Aufrufe. Hanneman nutzt eine Hash-Tabelle, um die Einzelstückexemplare den Klassen zuzuordnen. Die konkrete Ableitung des Aspektes ist notwendig, um eine Klasse als Einzelstück zu deklarieren. Für jede Einzelstückklasse wird somit ein konkreter Einzelstückaspekt benötigt. Das macht die Lösung kompliziert, insbesondere da sich aus der Betrachtung des Quelltexts der Einzelstückklasse ihr Verhalten bei der Exemplarbildung nicht erschließt.

4.2.4 Bewertung des Musters

Kopplung

Während bei der objektorientierte Variante des Musters der Einzelstückbelang stets erneut implementiert werden muss, ist das bei AspectJ und LOOM.NET nicht notwendig. Allerdings ist nur die LOOM-Variante derart entkoppelt, dass der zugrunde liegende Quelltext für neue Einzelstückklassen nicht geändert werden muss.

Modularisierung

Bei der objektorientierten Variante wird das Einzelstückverhalten direkt in die entsprechende Klasse implementiert. Mit den aspektorientierten Lösungen erfolgt die Implementation separat.

Redundanz

Die Einzelstückimplementation muss in der objektorientierten Variante für jede Klasse stets erneut erfolgen. Eine Ausnahme ist die Verwendung einer generischen Einzelstück-Basisklasse, was in vielen Fällen jedoch nicht anwendbar ist. Die LOOM-Lösung enthält hingegen keinen Redundanten-Quelltext. In AspectJ für jedes Einzelstück ein neuer Aspekt angelegt werden.

Implementationsaufwand

Im Gegensatz zu LOOM.NET und zur objektorientierten Variante hat die AspectJ-Variante einen erhöhten Implementationsaufwand, da für ein konkretes Einzelstück zusätzlich eine Ableitung des Aspektes erfolgen muss.

Gesamtbewertung

Kategorie	**OOP-Muster**	**AspectJ-Muster**	**LOOM-Muster**
Kopplung	1	2	3
Modularisierung	1	3	3
Redundanz	1	2	3
Implementationsaufwand	3	2	3

4.3 Das Dekorierermuster

Das Dekorierermuster zählt zur Kategorie Strukturmuster und bildet eine flexible Alternative zur Unterklassenbildung, wenn eine Klasse mit zusätzlicher Funktionalität erweitert werden soll [Gamma u. a. 1995, S. 175]. Im Gegensatz zur Objektorientierten Programmierung ist die Dekoration von Klassen mit zusätzlicher Funktionalität durch Advices ein inhärentes Mittel der Aspektorientierten Programmierung und stellt einen typischen Anwendungsfall für AOP dar.

4.3.1 Die Struktur des LOOM-Dekorierermusters

Ein Dekorierer wird verwendet, um verschiedenartige Module zu dekorieren. Das geschieht durch Annotation des entsprechenden Moduls. Ein Dekorierer kann also als *Paket-*, *Klassen-* und *Methodendekorierer* auftreten, wobei sein Wirkungsbereich die von ihm überdeckten Elemente sind. Weiterhin wird zwischen *unspezifischen* und *spezifischen* Dekorierern unterschieden.

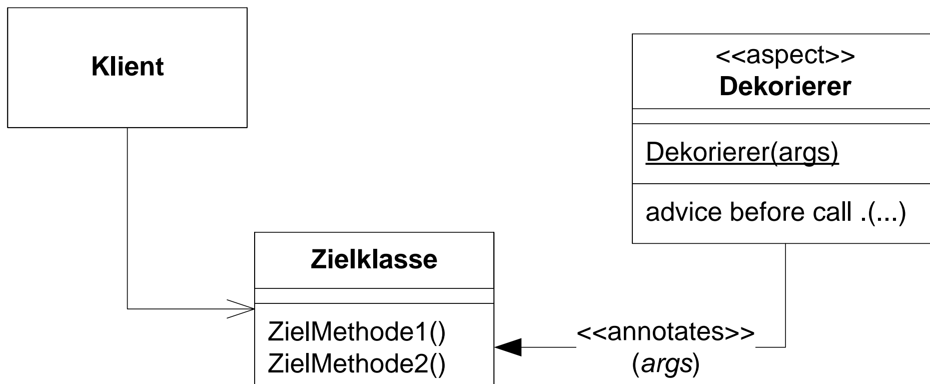


Abbildung 4.3: Ein unspezifische Dekorierer

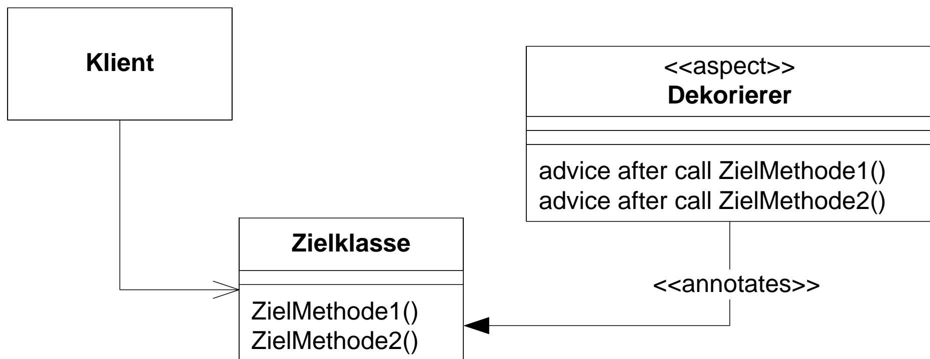


Abbildung 4.4: Ein spezifische Dekorierer

Ein *unspezifischer Dekorierer* Abbildung 4.3 ist ein Dekorierer, der nicht auf einen bestimmten Objekttyp spezialisiert ist. Das im Dekorierer implementierte Verhalten ist unabhängig von der dekorierten Schnittstelle. Sein Verhalten wird in einem Advice implementiert, der mit verschiedenen Zielmethoden verwoben werden kann. Je nachdem, ob es sich um einen Paket-, Klassen- oder Methodendekorierer handelt, werden alle Methoden eines Paketes, einer Klasse oder nur eine einzelne Methode mit zusätzlichem Verhalten versehen. Unspezifische Dekorierer werden oftmals bei der Annotation zusätzlich parametriert, um Einfluss auf ihre Funktion zu nehmen.

Ein Beispiel für einen unspezifischen Dekorierer ist das Aufzeichnen von Methodenaufrufen. Das folgende C#-Beispiel eines Methodendekorierers dekoriert den aufzuzeichnenden Methodenaufruf und legt gleichzeitig das Aufzeichnungsziel fest:

```
[Log("c:\logile.txt")]
public void foo() { ... }
```

Ein *spezifischer Dekorierer* ist ein Dekorierer, der eine bestimmte Klasse oder Ableitungen dieser Klasse bzw. ein bestimmtes Interface dekoriert. Dei Funktionalität des Dekorierers wird in Advices implementiert, die jeweils genau eine Methode der dekorierten Klasse oder des dekorierten Interfaces verweben.

Den spezifischen Dekorierer zeigt Abbildung 4.4. Er besteht aus einer *Zielklasse*, die die zu dekorierenden *Zielmethoden* enthält und einem *Aspekt*, der die Rolle des Dekorierers übernimmt. Jeweils ein Advice des Aspektes wird mit einer Zielmethode der zu dekorierernden Zielklasse verwoben.

Sowohl spezifische als auch unspezifische Dekorierer fügen ausschließlich Verhalten den zu dekorierenden Modulen hinzu. Im Gegensatz zu anderen Mustern, wie z. B. dem Stellvertreter, manipulieren sie keine Nachrichten an diese Module. Typischerweise werden aus diesem Grund ausschließlich **before**- oder **after**-Advices verwendet.

```
1  using System.Drawing;
2
3  public abstract class FigureElement
4  {
5      public abstract Rect BoundingRect { get; }
6      public abstract void Draw(Graphics gdi);
7  }
8
9  public class ElementWithFrame : AspectAttribute
10 {
11     [Call(Advice.Before)]
12     public void Draw([JPTarget] FigureElement fe, Graphics gdi)
13     {
14       using(Pen blackPen = new Pen(Color.Black, 3))
15       {
16         gdi.DrawRectangle(blackPen, fe.BoundingRect);
17       }
18     }
19 }
20
21 [ElementWithFrame]
22 public class Text:FigureElement
23 {
24     public override Rect BoundingRect {get { ... }}
25     public override void Draw(Graphics gdi) {...}
26 }
```

Listing 4.2: Beispiel eines LOOM-Dekorierers

Während beim objektorientierten Dekorierermuster der Dekorierer und die dekorierte Klasse eine gemeinsame Basisklassse oder ein gemeinsames Interface implementieren müssen, ist das beim aspektorientierten Dekorierer nicht notwendig. Der Dekorierer-Aspekt verwebt ausschließlich die Funktionen, die für seine Funktionalität notwenig sind.

4.3.2 Beispiel

Eine Anwendung (Listing 4.2) soll verschiedene Grafikelemente (`FigureElement`) anzeigen. Jedes Grafikelement implementiert hierzu die Methode `Draw`, um sich selbst darzustellen. Einige dieser Elemente haben gemeinsam, dass sie einen Rahmen um das sie umgebende Rechteck anzeigen müssen. Der Aspekt `ElementWithFrame` implementiert hierzu die Darstellung des Rahmens (Zeile 11-18) als Advice. Dieser Advice malt einen Rahmen, bevor die `Draw`-Methode des annotierten Grafikelements aufgerufen wird. Jede Grafikelementklasse, die einen solchen Rahmen besitzen soll, kann mit dem Aspekt annotiert werden.

Als Variante kann die Dekoration der Klassen auch erst zur Laufzeit erfolgen. Durch dynamisches Aspektweben werden die Dekorierer-Aspekte bei der Exemplarbildung an den Zielklassen (hier Grafikelemente) verwoben.

Der Dekorierer kann zusätzlich fallspezifisch parametriert werden. So könnte beispielsweise bei der Annotation die Rahmenfarbe übergeben werden, um unterschiedlichen Grafikelementen Rahmen mit jeweils verschiedenen Farben zu geben.

4.3.3 Bewertung des Musters

Kopplung

Ein unspezifischer Dekorierer ist völlig unabhängig vom zu dekorierenden Quelltext. Insbesondere können Methoden und Klassen dekoriert werden, deren Signatur nicht bekannt ist. Damit können solche Dekorierer als Komponente implementiert und für die verschiedensten Anwendungsfälle verwendet werden, was bei bei der objektorientierten Variante nicht möglich ist. Hier sind die Dekorierer explizit an die dekorierten Klassen gebunden.

Modularisierung

Der Quelltext eines Dekorierers wird in einer eigenen Klasse bzw. in einem eigenen Aspekt implementiert.

Redundanz

In der objektorientierten Variante muss in der zu dekorierenden Klasse explizit der Aufruf des Dekorierers für jede Operation implementiert werden.

Implementationsaufwand

Sowohl der objektorientierte Dekorierer, wie auch die LOOM-Variante haben einen ähnlichen Implementationsaufwand.

Gesamtbewertung

Kategorie	OOP-Muster	LOOM-Muster
Kopplung	1	2^a - 3^b
Modularisierung	3	3
Redundanz	2	3
Implementationsaufwand	3	3

[a] für den spezifischer Dekorierer
[b] für den unspezifischen Dekorierer

4.4 Das Adaptermuster

Ein *Adapter* [Gamma u. a. 1995, S. 139 ff.] ist ein Strukturmuster, das dazu dient, eine Schnittstelle in eine andere zu übersetzen. Adapter finden speziell dann Anwendung, wenn Komponenten Dritter in das eigene System einzubinden sind und Klassen zusammenarbeiten müssen, die aufgrund inkompatibler Schnittstellen dazu nicht in der Lage sind. Der Adapter implementiert die vom System erwartete Schnittstelle und wandelt Schnittstellenaufrufe in Aufrufe einer Schnittstelle der einzubindenden Komponente um. Zur besseren Unterscheidung wird erstere nachfolgend als *Zielschnittstelle* bezeichnet und letztere als *adaptierte Schnittstelle*. Methoden dieser Schnittstellen sind *Adaptermethoden*, respektive *adaptierte Methoden*.

Ein objektorientierter Adapter wird üblicherweise in einer speziellen Klasse implementiert, die die Zielschnittstelle anbietet. Diese Klasse wird dann entweder als *Klassenadapter* durch die Verwendung von Vererbung oder als *Objektadapter* mit Hilfe der Objektkomposition [Gamma u. a. 1995, S. 141] implementiert. Beim Klassenadapter erbt der Adapter von derjenigen Klasse, die die adaptierte Schnittstelle implementiert. Demgegenüber hält der Objektadapter nur eine Referenz auf die adaptierte Schnittstelle. Der Objektadapter hat gegenüber dem Klassenadapter unter anderem den Vorteil, dass mehrere Klassen (adaptierte Schnittstellen) in einem Adapter zusammengefasst werden können. Dabei ist die adaptierte Schnittstelle für den Klienten aufgrund fehlender Vererbungsbeziehung nicht mehr sichtbar.

Der Aufwand für die Implementation der Adaptermethoden hängt von der Ähnlichkeit der adaptierten Schnittstelle zur Zielschnittstelle ab. Häufig unterscheiden sich Adaptermethode und adaptierte Methode nur im Namen und der Reihenfolge der Methodenparameter. Eventuell müssen einige Parameter der adaptierten Methode mit vordefinierten Fixwerten belegt werden oder eine Konvertierung vom Typ des Parameters der Adaptermethode in den Typ des zugehörigen Parameters der adaptierten Methode erfolgen. Adaptermethoden können jedoch auch eine sehr komplexe Programmlogik enthalten, die eine Menge durchaus verschiedener Operationen beinhaltet. Vereinfacht lässt sich die Implementation einer Adaptermethode in folgende Kategorien unterteilen:

- **1:n Adapter**: Ein Aufruf einer Adaptermethode führt zu einer Sequenz von Aufrufen adaptierter Methoden.
- **1:1 Adapter**: Ein Aufruf einer Adaptermethode führt zu einem Aufruf von genau einer adaptierten Methode.
- **n:1 Adapter**: Erst eine Sequenz von Methodenaufrufen auf dem Adapter führt letztlich zum Aufruf einer adaptierter Methode.

In der Praxis werden Adapter eingesetzt, wenn verschiedene Technologien in einem System Anwendung finden sollen. In Windows-Umgebungen werden z. B. an vielen Stellen COM-Komponenten eingesetzt, für die es keine entsprechenden .NET-Klassen gibt. Mithilfe des *COM-Interop* lassen sich solche Komponenten zwar einfach in die .NET-Umgebung importieren, die beim Import automatisch generierten Klassen bieten jedoch oft nicht die gewünschte Schnittstelle. Gerade bei COM-Komponenten werden häufig sogenannte `VARIANT`-Typen als Methodenparameter verwendet, deren konkreter Typ sich erst aus der Benutzung der Schnittstelle ergibt. Das ist damit begründet, dass in COM die Überladung von Methoden - mehrere Methoden mit gleichem Namen aber unterschiedlichen Parametersignaturen - nicht möglich ist und daher eine Methode oft für ähnliche Anwendungsfälle bereitgestellt wird. In .NET wird ein solcher Parametertyp dann auf `object` abgebildet. Ein Adapter kann für jeden Anwendungsfall eine konkrete Methodendefinition mit den korrekten Parametertypen zur Verfügung stellen.

Die Idee des aspektorientierten Adapters besteht nun darin, die Umsetzung von Zielschnittstellen zur adaptierten Schnittstelle für jede Adaptermethode in einem Aspekt zu realisieren. Vorausgesetzt, der Quelltext der Aspekte ist wiederverwendbar und es handelt sich nicht um wenige Adaptermethoden, dann lässt sich durch LOOM der Implementationsaufwand für den Adapter reduzieren. Die Implementation der Adaptermethoden ist jedoch nicht notwendig, da das generisch durch den Aspekt geschieht. Das funktioniert allerdings nur unter bestimmten Voraussetzungen: Die Operationen innerhalb der Adaptermethoden zum Aufruf der adaptierten Methode müssen sich verallgemeinern lassen. Das trifft in den meisten Fällen ausschließlich auf einen 1:1 Adapter zu, bei denen die Adaptermethode genau eine adaptierte Methode aufruft. Verallgemeinerbare Operationen sind hier insbesondere

- das Ändern der Reihenfolge der übergebenen Parameter der Adaptermethode in die adaptierte Methode,
- das Belegen von Parametern der adaptierten Methode mit Nullwerten oder Konstanten,
- das Konvertieren eines Parameters in einen anderen Typ.

In wenigen Fällen ist auch für 1:n Adapter eine verallgemeinerte Umsetzung der Adaptermethoden in einem Aspekt denkbar, genau dann, wenn jeder Aufruf einer Adaptermethode eine Sequenz von Methodenaufrufen auf der adaptierten Schnittstelle zur Folge hat, die sich zur Laufzeit berechnen lässt. Das heißt, dass sich die Sequenz ableiten lassen muss aus:

- den übergebenen Parametern der Adaptermethode,
- den Metadaten der Adaptermethode (gegebenenfalls Annotationen) und
- Rückgabewerten von vorangegangenen Aufrufen adaptierter Methoden.

Mit unterschiedlichen Arten der Fehlerbehandlung lässt sich das gut veranschaulichen. Angenommen, die adaptierten Methoden liefern als Ergebnis einen Wahrheitswert, der anzeigt, ob deren Ausführung erfolgreich war oder nicht. Wenn die Ausführung nicht erfolgreich war, bietet die adaptierte Schnittstelle die Methode `GetLastError`, um die Fehlernummer zu

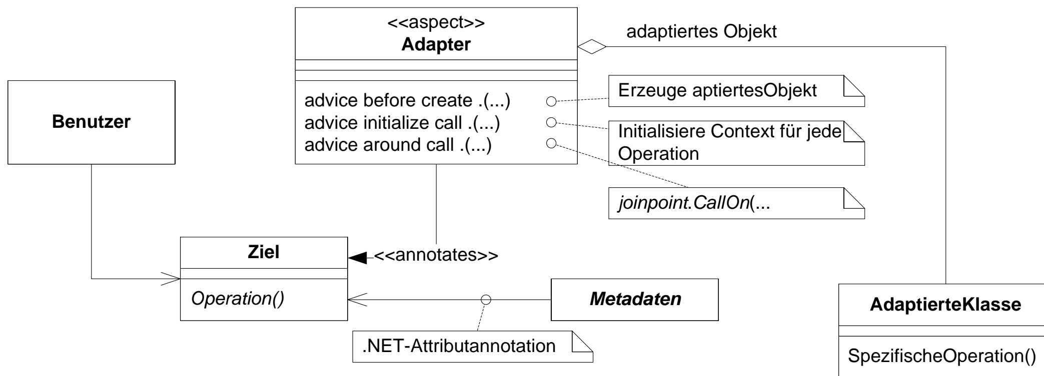


Abbildung 4.5: Das LOOM-Adaptermuster

erfahren. Von der Zielschnittstelle wird jedoch erwartet, dass im Fehlerfall eine Ausnahme geworfen wird, die die Fehlernummer enthält.

Der Adapter muss unabhängig von der konkreten Methode immer folgenden Algorithmus ausführen:

1. Methode aufrufen,
2. Rückgabewert prüfen,
3. Ist der Rückgabewert `true`, wirf eine Ausnahme mit dem Wert von `GetLastError`,
4. Anderfalls kehre normal zum Aufrufer zurück.

4.4.1 Die Struktur des LOOM-Adaptermusters

Das LOOM.UML-Diagramm in Abbildung 4.5 zeigt die Struktur eines LOOM-Adapters. Die abstrakte Klasse `Ziel` definiert die adaptierte Schnittstelle in Form abstrakter Adaptermethoden. Die Klasse ist durch den `Adapter`-Aspekt annotiert. Jede Adaptermethode kann mit zusätzlichen Metadaten annotiert sein, die deklarativ die Umsetzung auf die adaptierte Methode beschreiben. Dazu gehört zum Beispiel, welche Methode der adaptierten Schnittstelle aufgerufen werden soll, wie die Parameter auf die adaptierte Methode umgesetzt und welche Parameter mit festen Werten belegt werden sollen. Die Implementation der Adaptermethoden und damit die Umsetzung der Zielschnittstelle auf die adaptierte Schnittstelle (`AdaptierteKlasse`) wird von den drei Aspektmethoden im `Adapter`-Aspekt vorgenommen. Zur Laufzeit unterscheidet man hier zwei Phasen.

In der *Initialisierungsphase* des Adapters wird bei der Exemplarbildung der `Ziel`-Klasse durch die verwobene **create**-Advice zusätzlich ein Exemplar der Klasse `AdaptierteKlasse` angelegt. Für jede in `Ziel` definierte Adaptermethode werden aus deren Metadaten Datenstrukturen initialisiert, die später für die Umsetzung des Methodenaufrufs auf die adaptierte Schnittstelle benötigt werden. Das geschieht in der **advice call initialize**-Aspektmethode. Die Implementation dieser Methode hängt dabei sehr stark von den Belangen des jeweiligen Adapters ab.

In der *Betriebsphase* des Adapters wird jeder Aufruf einer Adaptermethode von der **advice call around** Methode des `Adapter`-Aspekts abgefangen und entsprechend den Regeln in den vorher initialisierten Datenstrukturen an die entsprechenden adaptierten Methoden weitergeleitet. Auch diese Implementation hängt natürlich stark von den Belangen des Adapters ab.

```
1  public class Adapter : AspectAttribute
2  {
3      [Call(Advice.Around), IncludeAll]
4      public T Adv<T>([JPContext] Context ctx, params object[] args)
5      {
6          StringBuilder sb = new StringBuilder();
7          sb.Append(ctx.CurrentMethod.Name);
8          foreach (object arg in args)
9          {
10             sb.Append(",");
11             sb.Append(arg.ToString());
12         }
13         Zielsystem.InvokeMethod(sb.ToString());
14
15         return default(T);
16     }
17 }
18
19 // Die neue Schnittstelle von AdaptierteKlasse
20 [Adapter]
21 public abstract class Ziel
22 {
23     public Ziel() {}
24     public abstract void foo(int i, double d);
25     public abstract void bar(string s);
26 }
```

Listing 4.3: Beispiel eines aspektorientierten n:1 Adapters

4.4.2 Beispiel für einen aspektorientierten Adapter

In Listing 4.3 ist ein einfacher n:1 Adapter dargestellt, der die Adaption eines Methodenaufrufs von einer Technologie in eine andere demonstrieren soll. Das Zielsystem nimmt Methodenaufrufe ausschließlich als kommaseparierte Zeichenkette entgegen. Der erste Wert entspricht dabei dem Methodenidentifizierer, und alle folgenden Werte entsprechen den Parametern. In stark vereinfachter Form ist das ähnlich der Kodierung von Webserviceaufrufen mit SOAP; denn auch dort werden die Aufrufe in eine von Menschen lesbare XML-Nachricht umgewandelt, um sie dann durch einen Kommunikationskanal an den Empfänger zu schicken - dafür wäre hier die Methode `InvokeMethod` (Zeile 13) verantwortlich.

Dieser relativ einfache Adapter kommt mit einem **around**-Advice aus (Zeile 3-17),in dem die Kodierung des Methodenaufrufes implementiert ist (Zeile 6-12). Über den Joinpoint-Kontext wird der Name der gerufenen Methode ermittelt und dann zusammen mit den Parametern in eine Zeichenkette kodiert.

4.4.3 Bewertung des Musters

Kopplung

Während objektorientierte Adapter nur für eine spezifische Schnittstelle implementiert werden können, lässt die LOOM-Variante einen generischen Ansatz zu. Dieser muss nicht notwendigerweise auf eine konkrete Schnittstelle angepasst sein, sondern kann durch Metadatenbeschreibungen auch unabhängig gestaltet werden.

Modularisierung

Der Belang des Adaptierens ist sowohl bei der LOOM-Variante als auch bei der objektorientierten Variante in einer eigenen Klasse bzw. einem eigenen Aspekt implementiert.

Redundanz

Die LOOM-Variante erlaubt es im Gegensatz zur objektorientierten Variante die Implementationen mehrerer Methoden zusammenzufassen und somit Redundanzen zu vermeiden.

Implementationsaufwand

Ausgehend vom Implementationsaufwand für einen Anwendungsfall (eine Methode), so ist dieser anfänglich für die generische LOOM-Variante höher. Aufgrund der geringeren Redundanz relativiert sich das später, wenn ähnliche Anwendungsfälle (weitere Methoden) zu implementieren sind.

Gesamtbewertung

Kategorie	OOP-Muster	LOOM-Muster
Kopplung	1	2
Modularisierung	3	3
Redundanz	1	3
Implementationsaufwand	3	2

4.5 Das Klassenkompositionsmuster

Das Klassenkompositionsmuster ist ein rein LOOM-basierendes Strukturmuster, nicht zu verwechseln mit dem objektorientierten Kompositum [Gamma u. a. 1995, S. 163 ff.]. Es wird verwendet, um in einer Klassenhierarchie Kindklassen gemeinsame Eigenschaften zuzuweisen, die in deren Elternklassen nicht definiert werden können.

Normalerweise werden gemeinsame Eigenschaften von verwandten Klassen in deren Elternklassen definiert. Durch die Vererbung stehen diese Eigenschaften den Kindklassen zur Verfügung, ohne dass diese erneut beschrieben werden müssen.

Es gibt jedoch auch Situationen, bei denen die gemeinsame Elternklasse nicht geändert werden kann. Dafür gibt es einige Gründe. Die Elternklasse kann beispielsweise in einer Komponente Dritter definiert sein, deren Quelltext nicht zur Verfügung steht. Oder es handelt sich um eine automatisch generierte Klassenhierarchie, bei der eine Änderung beim nächsten Generatorlauf verloren wäre. Es kann jedoch auch sein, dass die Klassenhierarchie von mehreren Klienten verwendet wird und es sich daher verbietet, klientenspezifische Erweiterungen vorzunehmen.

In Abbildung 4.6 ist das Problem dargestellt. `Komponente 1` kann durch den Programmierer nicht geändert werden, die Klassen `D` und `E` haben gemeinsame Eigenschaften (in diesem Fall die Methode `Foo`). Die gemeinsame Basisklasse, in die `Foo` eigentlich implementiert werden könnte, ist die Klasse `A`. Da das jedoch nicht möglich ist, muss die Implementation sowohl in `D` als auch in `E` erfolgen.

Mit dem LOOM-Klassenkompositionsmuster lässt sich das Problem einfach lösen. Die Implementierung der Methode `Foo` erfolgt in einem Aspekt, und dieser wird dann mit den beiden Klassen `D` und `E` verwoben.

4.5.1 Struktur des LOOM-Klassenkompositionsmusters

Abbildung 4.6 zeigt die Struktur des LOOM-Klassenkompositionsmusters. Eine gegebene Klassenhierarchie `A`, `B` und `C` soll um eine gemeinsame Eigenschaft (die Methode `Foo`) erweitert werden. Die Klassenhierarchie selbst ist nicht veränderbar `Komponente 1`. Die gemeinsamen Eigenschaften werden daher als Introduktionen in einem Aspekt (`NeueEigenschaft`) implementiert. Alle Klassen, die aus der Klassenhierarchie mit der neuen Eigenschaft verwendet

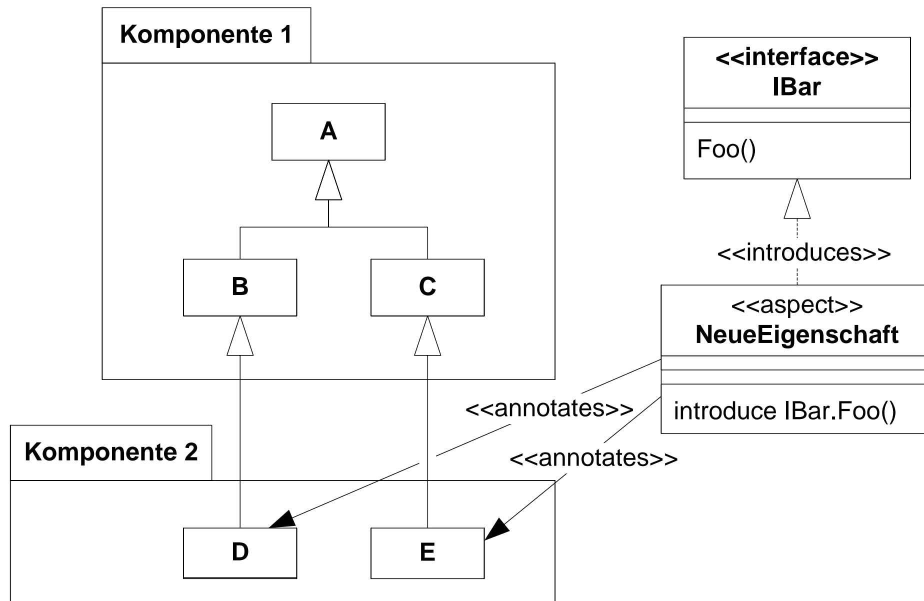


Abbildung 4.6: Die Struktur des LOOM-Klassenkompositionsmusters

werden sollen, werden in einer eigenen Komponente abgeleitet (`Komponente 2`) und mit dem Aspekt annotiert.

Die Introduktionen können optional auch mit dem Modifizierer **virtual** deklariert werden. In diesem Fall können Introduktionen aus unterschiedlichen Aspekten gleichzeitig jeweils eine Methode der Schnittstelle implementieren. Die Implementation der Schnittstelle besteht dann aus der Aneinanderreihung der einzelnen Introduktionen.

4.5.2 Beispiel

Ein Beispiel für das Klassenkompositionsmuster lässt sich in der Implementation des Rapier-Loom.Net Aspektwebers finden. Exemplare, die zur Laufzeit verwoben wurden, können potentiell auch serialisiert werden. Das Exemplar wird in einer Bytefolge gespeichert, um es zu einem späteren Zeitpunkt aus dieser Bytefolge wieder zu rekonstruieren. Bei der Rekonstruktion eines verwobenen Exemplars muss sichergestellt sein, dass sein verwobener Typ bereits existiert. Daher fügt Rapier-Loom.Net in alle als serialisierbar ausgewiesenen Typen dynamisch einen weiteren Aspekt ein. Dieser `SerializationAspect` erweitert die Schnittstelle der Klasse um eine Methode, über die eine spezielle Serialisierung dieser Objekte erfolgt. Dieses Verfahren wird im Abschnitt 3.5.3 genauer beschrieben.

4.5.3 Vergleich zu anderen Lösungen

Das Komponieren von Klassen aus verschiedenen Einzelbestandteilen lässt sich auch durch die Verwendung von *Mehrfachvererbung* [Stroustrup 1998, S. 415 ff.] lösen. Mehrfachvererbung gilt jedoch als komplex und schwer verständlich, insbesondere wegen der Auflösung von Mehrdeutigkeiten. Daher wird Mehrfachvererbung in Sprachen wie Java und CLI-basierenden Sprachen nicht unterstützt.

Die Nachteile der Mehrfachvererbung existieren bei *Mixins* [Bracha und Cook 1990; Moon 1986] und *Traits* [Odersky und Zenger 2005; Schärli u. a. 2003] nicht. Mit ihnen lässt sich das Problem in ähnlicher wie der hier vorgestellten Variante lösen. Mixins und Traits werden in Abschnitt 7.1.3 ausführlicher diskutiert.

Auch mit den *Erweiterungsmethoden* von C# 3.0 [Microsoft Corporation 2005b, §26.2] lassen sich Klassen aus verschiedenen Bestandteilen komponieren. Allerdings können hier nur Methoden zu einer existierenden Klassendefinition hinzugefügt werden, für Attribute gilt das

nicht (siehe auch Abschnitt 7.1.4). Die hinzugefügten Methoden sind auch nur in der aktuellen Kompilierungseinheit sichtbar.

4.5.4 Bewertung des Musters

Kopplung

Bei der Klassenkomposition ist bei allen vorgestellten Varianten eine hohe Kopplung vorhanden, worin das Wesen der Komposition besteht. Viele Teile sollen zu einem zusammengefasst werden. Die Einzelteile sollen dabei spezifisch zueinander passen.

Modularisierung

Alle Varianten zeichnen sich durch ein hohes Maß an Modularisierung aus. Auch das ist ein Grundziel des Musters: Die Einzelteile sollen in eigenen Modulen implementiert werden können, um sie später zusammenzufügen.

Redundanz

Alle vorgestellten Varianten vermeiden redundanten Quelltext.

Implementationsaufwand

Der Implementationsaufwand ist bei allen Varianten in etwa gleich groß.

Gesamtbewertung

Kategorie	**Mehrfachvererbung/ Mixins**	**Erweiterungsmethoden**	**LOOM-Muster**
Kopplung	1	1	1
Modularisierung	3	3	3
Redundanz	3	3	3
Implementationsaufwand	3	3	3

4.6 Das Beobachtermuster

Das Beobachtermuster ist ein Verhaltensmuster, um in einem System mit einer Menge von interagierenden Klassen die Konsistenz der miteinander in Beziehung stehenden Objekte aufrechtzuerhalten. Die Änderung des Zustands eines Objektes führt dazu, dass alle abhängigen Objekte über die Zustandsänderung benachrichtigt werden und entsprechend reagieren können.

Ein typischer Anwendungsfall ist die Trennung der Benutzerschnittstelle von der dahinterliegenden Anwendung. Es gibt im System Objekte, die die Anwendungsdaten enthalten und andere, die die Darstellung repräsentieren. Für ein und dieselben Anwendungsdaten kann es dabei mehrere Darstellungsformen geben. Angenommen, eine Anwendung soll die Infrastruktur eines Rechenzentrums verwalten. Sie soll in der Lage sein, den Zustand des Rechenzentrums sowohl in einem Anwendungsfenster tabellarisch als auch in einem weiteren Anwendungsfenster grafisch als Netztopologie anzuzeigen. Ändert sich der Zustand eines Gerätes, soll sichergestellt sein, dass sich sowohl der Inhalt der Tabelle als auch die Grafik aktualisieren.

Für die Klasse, die die Geräte repräsentiert, muss keine direkte Verbindung zu den für die Darstellung der Infrastruktur verantwortlichen Fensterklassen existieren. Die Benachrichtigung über die Änderung der Daten sollte weitestgehend transparent und unabhängig von

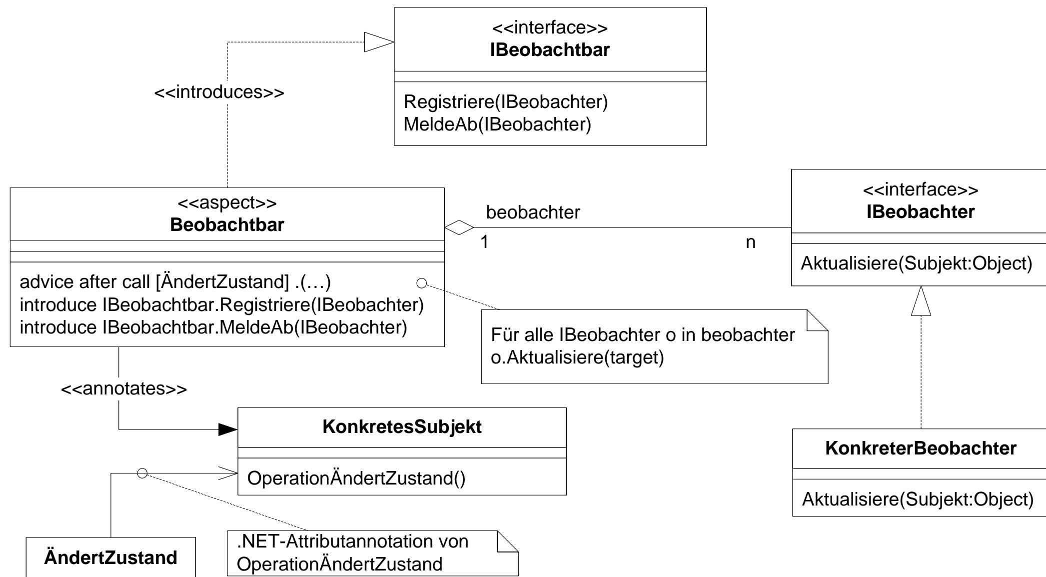


Abbildung 4.7: Das LOOM-Beobachtermuster

der Implementation der Geschäftsklasse erfolgen. Bekannt ist diese Art von Interaktion auch als *Publish-Subscribe*.

In der objektorientierten Variante des Musters [Gamma u. a. 1995, S. 239 ff.] gibt es eine Basisklasse für alle Klassen, deren Exemplare beobachtbar sein sollen. In dieser Basisklasse sind Methoden zum Registrieren und Abmelden der *Beobachter* und Datenstrukturen zum Abspeichern aller angemeldeten Beobachter implementiert.

Die objektorientierte Variante hat jedoch einige Nachteile. Alle beobachtbaren Klassen (*Subjekte*) müssen von einer Basisklasse ableiten. Das schränkt die Möglichkeiten der Vererbung bei Programmiersprachen ein, die keine Mehrfachvererbung unterstützen. Eine Implementation über ein Interface würde das Problem lösen, nur müssten die Methoden zum Registrieren und Abmelden der Beobachter für jedes Subjekt erneut implementiert werden.

Ein weiterer Nachteil ist, dass die Subjekte explizit die Benachrichtigung der Beobachter veranlassen müssen. Der Quelltext der Subjekte ist damit von Benachrichtigungsaufrufen durchsetzt, die nichts mit der Programmlogik des Subjektes zu tun haben.

4.6.1 Die Struktur des LOOM-Beobachtermusters

Abbildung 4.7 zeigt die Struktur des LOOM-Beobachtermusters. Ein `KonkretesSubjekt` ist mit dem `Beobachter`-Aspekt annotiert, das die Schnittstelle `IBeobachtbar` einführt. Diese ist in den Aspektmethoden `Registriere` und `MeldeAb` implementiert. Mit ihnen kann sich ein `KonkreterBeobachter` bei Exemplaren des Subjekts registrieren. Die Liste der registrierten Beobachter wird im verwobenen Aspektexemplar gespeichert (`beobachtbar`).

Der Aspekt ist weiterhin mit jeder Methode des Subjekts verwoben, die mit dem `ÄndertZustand` Attribut markiert ist. Dabei sind nur diejenigen Methoden des Subjekts markiert, die tatsächlich eine Änderung des Zustandes auslösen. Der verwobene Advice ruft jeweils nach der Ausführung dieser Methoden die `Aktualisiere`-Methode aller registrierten Beobachter mit einer Referenz auf das soeben geänderte Subjekt auf.

Wie in Listing 4.5 zu erkennen ist, ist die Implementation eines Subjektes sehr einfach. Die Klasse `Subjekt` ist durch den `Beobachtbar`-Aspekt annotiert. Weiterhin ist die Methode `OperationÄndertZustand` mit dem Attribut `ÄndertZustand` markiert, um auszudrücken, dass diese Methode den Zustand des Reports ändert.

Der Beobachter `Beobachter` zeichnet sich dadurch aus, dass er die `IBeobachtbar`-Schnittstelle implementiert. Er kann mit folgender Zeile an ein konkretes Exemplar des Reports

```
 1 public interface ISubjekt
 2 {
 3     void Registriere(IBeobachter Observer);
 4     void MeldeAb(IBeobachter Observer);
 5 }
 6
 7 public interface IBeobachter
 8 {
 9     void Benachrichtige(ISubjekt observedSubject);
10 }
11
12 public class AendertZustand : Attribute { }
13
14 [AttributeUsage(AttributeTargets.Class)]
15 public class Beobachtbar : AspectAttribute
16 {
17     List<IBeobachter> beobachterliste = new List<IBeobachter>();
18
19     [IncludeAnnotated(typeof(AendertZustand))]
20     [Call(Advice.After)]
21     public void Notify([JPContext] Context ctx, params object[] args)
22     {
23         foreach (IBeobachter beobachter in this.beobachterliste)
24         {
25             beobachter.Benachrichtige((ISubjekt)ctx.Instance);
26         }
27     }
28
29     [Introduce(typeof(ISubjekt))]
30     public void Registriere(IBeobachter beobachter)
31     {
32         if (!this.beobachterliste.Contains(beobachter))
33         {
34             this.beobachterliste.Add(beobachter);
35         }
36     }
37
38     [Introduce(typeof(ISubjekt))]
39     public void MeldeAb(IBeobachter beobachter)
40     {
41         if (this.beobachterliste.Contains(beobachter))
42         {
43             this.beobachterliste.Remove(beobachter);
44         }
45     }
46 }
```

Listing 4.4: Beispiel eines Beobachters

übergeben werden:

```
((IBeobachtbar)report).MeldeAn(beobachter)
```

4.6.2 Beispiel

In Listing 4.4 ist die Implementation eines Beobachter-Aspektes und in Listing 4.5 deren Verwendung für das eingangs genannte Infrastrukturbeispiel dargestellt. Es gibt eine Fensterklasse (`Tabelle`), die als Beobachter fungiert und das zu beobachtende Subjekt `Infrastrukturdaten`. Die eigentliche Beobachter-Logik ist im Aspekt `Beobachtbar` unabhängig vom konkreten Subjekt und Beobachter implementiert. Die Aspektimplementation ist dabei vollkommen unabhängig von der tatsächlichen Benutzung und kann ohne Veränderung wiederverwendet werden.

Der Aspekt enthält einen Advice (Zeile 19-27), der immer dann aufgerufen wird, wenn die Methode `FuegeRechnerHinzu` aus Listing 4.5 abgewickelt wurde. Der Aufruf der Beobachter erfolgt in Zeile 25. Die Introduktionen in Zeile 29 und 38 erweitern das Subjekt

```
public partial class Tabelle : Form, IBeobachter
{
    Infrastrukturdaten daten;

    public Tabelle(Infrastrukturdaten daten)
    {
        this.daten = daten;
        InitializeComponent();
        ((ISubjekt)daten).Registriere(this);
    }

    private void Tabelle_FormClosed(object sender, FormClosedEventArgs e)
    {
        daten.MeldeAb(this);
    }
}

...

[Beobachtbar]
public class Infrastrukturdaten
{
    [AendertZustand]
    void FuegeRechnerHinzu(...)
    { ... }
}
```

Listing 4.5: Verwendung des Beobachters

`Infrastrukturdaten` weiterhin um das `ISubjekt`-Interface. Diese wird vom Beobachter genutzt, um sich zu registrieren und abzumelden.

4.6.3 Vergleich zu anderen Lösungen

Auch bei der Implementierung des Observers schlägt [Hannemann 2005; Hannemann und Kiczales 2002, 2003] wie bereits beim Einzelstückmuster vor, die Logik des Beobachtermusters als abstrakten Aspekt zu implementieren. Die konkrete Ableitung dieses Aspekts (Listing 4.6) definiert nun

- welche Klasse das Subjekt ist und welche Klassen seine Beobachter sind (realisiert durch Introduktionen),
- welche Operationen des Subjekts, eine Benachrichtigung der Beobachter auslösen sollen (Definition eines Pointcuts),
- die Aktion, durch welche die Benachrichtigung des Beobachters erfolgen soll (durch einen Advice, der in dem zuvor definierten Pointcut eingewoben ist).

Für jede neue Subjekt/Beobachter-Kombination muss eine neue Ableitung des Aspekts erfolgen. Der Aufwand hierfür scheint insbesondere im Vergleich zur LOOM-Lösung recht hoch, das in LOOM durch eine einfache Annotation der Klassen erreicht wird.

Die Variante von [Piveta und Zancanella 2003] unterscheidet sich nicht wesentlich von [Hannemann und Kiczales 2002]. Auch hier wird wieder ein abstrakter Aspekt definiert, der für jede neue Subjekt-Beobachter Relation abgeleitet werden muss.

4.6.4 Bewertung des Musters

Kopplung

Die Bestandteile des objektorientierte Beobachtermuster müssen in jedes Subjekt und in jedem Beobachter erneut implementiert werden. Damit ist der Beobachter-Belang eng an

```
public aspect CoordinateObserver extends ObserverProtocol{
  // Das Subjekt: hier die Klasse Point
  declare parents: Point  implements Subject;
  // Die Observer der Klasse Point
  declare parents: Screen implements Observer;

  // Die Methoden im Subjekt, die zur Benachrichtigung führen
  protected pointcut subjectChange(Subject subject):
    (call(void Point.setX(int)) ||
     call(void Point.setY(int)) ) && target(subject);

  // Die Aktion die Ausgeführt werden soll, wird vom Basisaspekt aufgerufen
  protected void updateObserver(Subject subject, Observer observer) {
    ((Screen)observer).display("Screen updated "+
      "(point subject changed coordinates).");
  }
}
```

Listing 4.6: Auszug eines Beobachters [nach Hannemann und Kiczales 2003]

die restliche Implementierung gekoppelt. In der AspectJ-Variante werden die Bestandteile separat in abgeleiteten Aspekten implementiert. Die LOOM-Lösung hingegen erlaubt einen völlig generischen Ansatz, bei dem es nur eine Implementation des Belanges gibt.

Modularisierung

Die enge Kopplung der objektorientierten Variante hat auch eine schlechte Modularisierung zur Folge: Das Subjekt ist von der Beobachter-Implementation durchsetzt. In AspectJ und der LOOM-Variante steht der Quelltext des Musters separat. Mit LOOM.NET kann er sogar als wiederverwendbare Komponente ausgelagert werden.

Redundanz

Jedes Subjekt muss in der objektorientierten Implementation den entsprechenden Quelltext des Musters erneut implementieren. Das ist in der LOOM-Variante nicht notwendig. In AspectJ muss für jedes Subjekt ein neuer Aspekt angelegt werden.

Implementationsaufwand

Der Aufwand für die Implementation ist in beiden Varianten ähnlich hoch. In AspectJ entsteht aber ein zusätzlicher Aufwand für die Implementation des Basisaspekts.

Gesamtbewertung

Kategorie	OOP-Muster	AspectJ	LOOM-Muster
Kopplung	1	2	3
Modularisierung	1	3	3
Redundanz	1	2	3
Implementationsaufwand	3	2	3

4.7 Das Besuchermuster

Das Besucher-Muster [Gamma u. a. 1995, S. 331 ff.] ist ein *Verhaltensmuster*, um eine feste, meist rekursive Datenstruktur mit beliebigen Operationen erweitern zu können. Die Operationen auf den Klassen der Datenstruktur *Datenklassen* werden dabei in eigene Klassen,

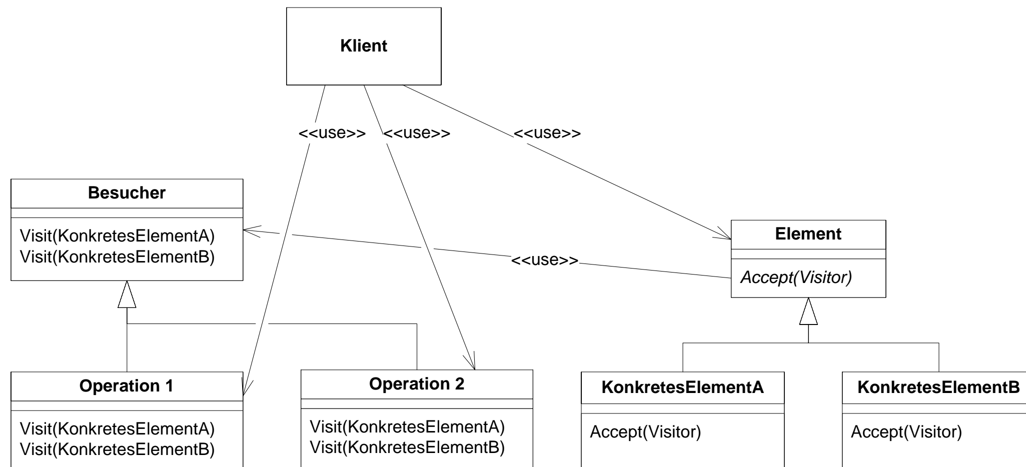


Abbildung 4.8: Das objektorientierte Besuchermuster

Besucherklassen, gekapselt. Das hat den Vorteil, dass nicht miteinander verwandte Operationen zum einen besser modularisiert sind, zum anderen eine neue Operation durch Hinzufügen einer neuen Klasse implementiert werden kann.

Abbildung 4.8 zeigt eine objektorientierte Version des Besuchermusters. Die Basisklasse der Datenklassen ist `Element`. Jene für Besucherklassen ist `Besucher`. Alternative Ansätze implementieren diese Klassen auch als Interface, das ändert jedoch nichts an der Funktionsweise. Ein `Klient` hält eine Referenz auf eine Datenstruktur, wobei ihm bis auf die Basisklasse die konkreten Datenklassen nicht notwendigerweise bekannt sein müssen.

Auf dieser Datenstruktur kann der Klient verschiedene Operationen ausführen, die in jeweils einer Ausprägung einer Besucherklasse (auch *Visitor*) implementiert sind. Er übergibt dazu ein Exemplar der entsprechenden Besuchersklasse an die überschriebene `Accept`-Methode seines referenzierten Exemplars der Datenklasse, woraufhin dieses die `Visit`-Methode des Besuchers mit sich selbst als Referenz aufruft. Durch diese Indirektion wird stets diejenige `Visit`-Überladung der Besucherklasse aufgerufen, die für die Ausprägung der rufenden Datenklasse zuständig ist. In ihrer zur Datenklasse passenden `Visit`-Überladung können die Besucher daraufhin Operationen auf dem übergebenen Exemplar ausführen und die Datenstruktur traversieren, indem sie wieder die `Accept`-Methode der weiteren Exemplare der Struktur aufrufen.

Der Nachteil der klassischen objektorientierten Implementation ist, dass das Hinzufügen neuer Klassen der Datenstruktur eine Erweiterung aller Besucherklassen bedeutet, da in allen Klassen neue `Visit`-Überladungen eingeführt werden müssen. Dieses Problem wurde durch [Wadler 1998] als *Expression Problem* bekannt. Wadler konstruiert hierzu eine einfache imaginäre Sprache *EXP*, um Ausdrücke darstellen zu können. Die Sprache enthält einen Additionsoperator und Literale. Sollen Ausdrücke von *EXP* zusammen mit Operationen, um sie zu manipulieren, als Klassenstruktur dargestellt werden, ist es schwierig, sich entweder für eine einfache Erweiterbarkeit der Ausdrücke oder eine einfache Erweiterbarkeit der Operationen auf diesen zu entscheiden. Einfache Erweiterbarkeit meint dabei, neue Operationen und Ausdrucke hinzuzufügen:

- ohne existierenden Quelltext zu modifizieren oder zu replizieren,
- ausschließlich mit Kenntnis der abstrakten Basisklassen für Ausdrücke und Operationen,
- ohne bestehenden Quelltext neu übersetzen zu müssen (d. h. dieser kann als Komponente vorliegen) und
- unter Beibehaltung der Typsicherheit (keine expliziten Typumwandlungen).

```
1  public abstract class Expression
2  {
3      public abstract void Accept(Visitor visitor);
4  }
5
6  public class Literal : Expression
7  {
8      public Literal(double value) { ... }
9
10     public override void Accept(Visitor visitor)
11     {
12         visitor.Visit(this);
13     }
14
15     ...
16 }
17
18 public class Add : Expression
19 {
20     public Add(Expression left, Expression right) {  .... }
21
22     public override void Accept(Visitor visitor)
23     {
24         visitor.Visit(this);
25     }
26
27     ...
28 }
29
30 public abstract class Visitor
31 {
32     public abstract void Visit(Literal lit);
33     public abstract void Visit(Add add);
34 }
35
36 public class Print:Visitor
37 {
38     public override void Visit(Literal lit) { ... }
39     public override void Visit(Add add) { ... }
40 }
41
42 public class Eval:Visitor
43 {
44     public override void Visit(Literal lit) { ... }
45     public override void Visit(Add add) { ... }
46 }
```

Listing 4.7: Die objektorientierte Besucher-Implementierung von EXP

In Listing 4.7 ist eine Implementierung der Sprache *EXP* mit dem Besuchermuster dargestellt. Die Ausdrücke (`Add` und `Literal`) sowie die Operationen (`Print` und `Eval`) sind jeweils in eigene Klassen gekapselt. Das Hinzufügen neuer Operationen ist mit diesem Ansatz problemlos möglich, es muss nur eine neue Spezialisierung von der Klasse `Visitor` implementiert werden. Möchte man hingegen einen neuen Ausdruck hinzufügen, muss die `Visitor`-Klasse selbst und damit alle ihre Spezialisierungen angepasst werden.

Alternativ existiert natürlich auch die Möglichkeit, alle Operationen direkt in den Klassen der Ausdrücke zu implementieren. Jede Operation bekäme dann eine eigene virtuelle Methode in der Basisklasse der Ausdrücke (`Expression`). Das Hinzufügen neuer Ausdrücke wäre in diesem Fall unproblematisch, das Hinzufügen neuer Operationen würde aber mindestens eine Änderung in der Basisklasse nach sich ziehen. Abgesehen davon hat diese Lösung auch den Nachteil, dass sie schlecht modularisiert ist. Die Implementation von jeweils einer Operation ist über viele Datenklassen verteilt und die Datenklassen wiederum sind mit dem Quelltext vieler Operationen durchsetzt.

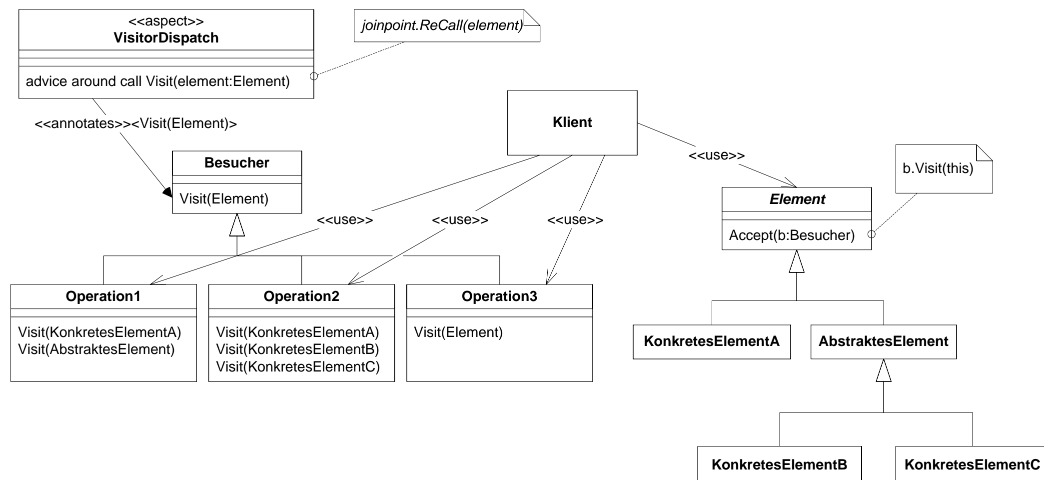


Abbildung 4.9: Die Struktur des LOOM-Besuchermusters

4.7.1 Die Struktur des LOOM-Besuchermusters

In Abbildung 4.9 ist der Aufbau des LOOM-Besuchermusters dargestellt. Neu ist hier, dass ausschließlich `Element`, die abstrakte Basisklasse der Datenklasse, eine `Accept`-Implementierung benötigt. Das aufinden der korrekten `Visit`-Überladung erfolgt erst zur Laufzeit durch den Aspekt `VisitorDispatch`. Dieser hat einen Advice, der mit der einzigen `Visit`-Methode der `Besucher`-Basisklasse verwoben ist.

Erst die konkreten Besucherklassen implementieren für die Datenklassen eine `Visit`-Methode, für die sie auch eine Implementation anbieten. Neu ist ebenfalls, dass sie dabei nicht für alle Datenklassen eine Implementation vorhalten müssen. Vielmehr kann unter Berücksichtigung der Vererbungshierarchie der Datenklassen auch eine allgemeinere Implementation erfolgen. Ist keine vorhanden, wird die `Visit`-Methode der `Besucher`-Basisklasse ausgeführt.

Der Quelltext für das Verhalten des Besuchermusters konstituiert sich ausschließlich in den Klassen `Elements` und `Operations` sowie den Aspekt `VisitorDispatch`. Die Spezialisierungen der Daten- und Besucherklassen hingegen enthalten keinen Quelltext, der dem Besuchermuster zugerechnet wird.

4.7.2 Beispiel

Listing 4.8 zeigt den geänderten Quelltext für die Implementation von *EXP* mit der LOOM-Variante des Besuchermusters. Die Datenklassen `Add` und `Literal` enthalten keinen zum Besuchermuster gehörenden Quelltext. Die Besucherklassen `Print` und `Eval` implementieren für jede Datenklasse eine Überladung der `Visit`-Methode.

Eine neue Datenklasse lässt sich einfach durch Spezialisierung der Klasse `Expression` hinzufügen. Solange die Besucherklassen keine Überladung der `Visit`-Methode für die neue Klasse anbieten, hat diese Datenklasse jedoch in diesen Besucherklassen keine Semantik; es wird die Basisimplementation von `Visit` in der `Visitor`-Klasse aufgerufen. Was in einigen Fällen möglicherweise sinnvoll ist, wird in den meisten Fällen dazu führen, dass in jede vorhandene Operationsklassse eine neue Überladung der `Visit`-Methode hinzugefügt werden muss. Das läßt sich durch die Spezialisierung der Datenklassen vermeiden.

Angenommen, die Sprache *EXP* soll um Subtraktion, Division und Multiplikation erweitert werden. Mit der Addition haben diese Operatoren gemeinsam, das zu ihnen jeweils ein linksseitiger und ein rechtsseitiger Ausdruck gehört. Der Algorithmus zum Anzeigen dieser Ausdrücke unterscheidet sich jeweils nur in dem anzuzeigenden Symbol. Das kann durch Einfügen der Zwischenklasse `BinaryOperation` in die Vererbungshierarchie ausgedrückt werden

```
public class VisitorDispatchAttribute:AspectAttribute
{
    [Call(Advice.Around)]
    public void Visit([JPContext] Context ctx, Expression node)
    {
        ctx.ReCall(node);
    }
}

public abstract class Expression
{
    void Accept(Visitor visitor)
    {
        visitor.Visit(this);
    }
}

public class Literal : Expression
{
    public Literal(double value) { ... }

    ...
}

public class Add : Expression
{
    public Add(Expression left, Expression right) {  .... }

    ...
}

public abstract class Visitor
{
    [VisitorDispatch]
    public virtual void Visit(Expression exp) { ... }
}

public class Print:Visitor
{
    public void Visit(Literal lit) { ... }
    public void Visit(Add add) { ... }
}

public class Eval:Visitor
{
    public void Visit(Literal lit) { ... }
    public void Visit(Add add) { ... }
}
```

Listing 4.8: Die LOOM-Besucherimplementierung von EXP

```
1 public class BinaryOperation:Expression
2 {
3   public Expression Left,Right;
4   public string Symbol;
5 }
6
7 public class Print:Visitor
8 {
9     public void Visit(BinaryOperation op)
10    {
11        op.Left.Accept(this);
12        Console.Write(op.Symbol);
13        ol.Right.Accept(this);
14    }
15 }
```

Listing 4.9: Die Visit-Überladung für BinaryOperation

(Listing 4.9). Diese wird dann zur Basisklasse für alle zweistelligen Operationen, wie der `Add`-Klasse.

Mit dieser Vererbungshierarchie muss die `Print`-Klasse ausschließlich eine `Visit`-Überladung für `Literal` und `BinaryOperation` zur Verfügung stellen. Weitere zweistellige Operatoren lassen sich durch Spezialisierung der `BinaryOperation`-Klasse hinzufügen, ohne dass eine Änderung der `Print`-Klasse notwendig ist. Das heißt, es muss weder der Quelltext des Klienten noch der Daten- und Besucherklassen angepasst und übersetzt werden. Die neue Datenklasse kann in eine eigene Komponente integriert werden.

Für Besucherklassen, die für die neue Datenklasse in jedem Fall eigene `Visit`-Implementation benötigen (wie z. B. die `Eval`-Klasse), gibt es verschiedene Möglichkeiten der Erweiterung. Am einfachsten ist es, der Klasse eine neue Überladung der `Visit`-Methode hinzuzufügen. Allerdings müsste dazu der existierende Quelltext geändert und neu übersetzt werden. Das ist nicht immer möglich und widerspricht Wadlers Forderung.

Eine andere Möglichkeit wäre, die Besucherklasse abzuleiten und die Überladung von `Visit` dort zu implementieren. Eine Erweiterung ist dann jedoch nur linear auf einer Vererbungsachse der Besucherklasse möglich. Wird eine weitere Datenklasse hinzugefügt, müssen die Besucherklassen erneut abgeleitet werden. So entsteht eine Vererbungskette, die ein quasi-paralleles Erweitern der Implementation von mehr als einer Partei unmöglich macht. Weiterhin muss der Quelltext des Klienten an allen Stellen, an denen die Besucherklasse erzeugt wird, auf die speziellste Ableitung der Klasse angepasst werden. Wird das *Fabrikmuster* [Gamma u. a. 1995, S. 107 ff.] verwendet, ist eine Änderung nur an einer Stelle notwendig.

In Verbindung mit dem LOOM-Klassenkompositionsmuster aus Abschnitt 4.5 lassen sich jedoch auch unabhängig Erweiterungen für die Besucherklassen definieren: Die `Visit`-Methoden für eine neue Datenklasse wird in einem Aspekt als Introduktion definiert.

4.7.3 Vergleich zu anderen Lösungen

Das Besucher-Muster findet durch das von [Wadler 1998] aufgebrachte „Expression Problem" in der Forschung eine recht große Beachtung. Die vom Autor vorgestellte Lösung lässt sich grundsätzlich auch mit *Multidispatch*-Programmiersprachen implementieren. Beispiele für solche Sprachen sind *MultiJava* [Clifton u. a. 2000], *Common Lisp* [Steele 1990] und *Dylan* [Shalit 1996]. Hier ist man jedoch an die jeweilige Programmiersprache gebunden. Insbesondere *MultiJava* kommt nicht ohne Erweiterung des Java-Sprachstandards aus. Alternative Implementationen ändern die virtuelle Maschine [Dutchyn u. a. 2001].

[Torgersen 2004] bietet verschiedene Lösungen auf der Basis generischer Klassen in Java. Seine Ansätze lösen zwar grundsätzlich das Problem der Erweiterbarkeit, erfordern jedoch bei jeder Änderung die Übersetzung des gesamten Quelltextes, inklusive der Basisklassen.

```
1  public aspect VisitorDispatcher {
2    public pointcut visitCall(Visitor v):
3      !within(VisitorDispatcher) &&
4      call(void Visitor.visit(*)) && target(v);
5    void around(Visitor v, A a):
6      visitCall(v) && args(a) && !args(B) && !args(C){
7      v.visit(a);
8    }
9
10   void around(Visitor v, B b):
11     visitCall(v) && args(b) && !args(C){
12     v.visit(b);
13   }
14
15   void around(Visitor v, C c):
16     visitCall(v) && args(c) && !args(B){
17     v.visit(c);
18   }
19 }
```

Listing 4.10: Eine Visitor-Implementierung in AspectJ [nach Schmidmeier u. a. 2003, S. 3]

Ein weiterer Nachteil ist, dass eine Erweiterung nur linear auf einer Vererbungsachse der Besucherklassen möglich ist. Wird eine Datenklasse hinzugefügt, müssen alle Besucherklassen für diese jeweils einmal abgeleitet werden. Jede weitere Datenklasse erfordert wiederum das Spezialisieren dieser Ableitungen. Neben dem Problem der Übersetzung macht das ein unabhängiges quasi-paralleles Erweitern der Implementation von mehr als einer Partei unmöglich.

[Schmidmeier u. a. 2003] demonstrieren einen aspektorientierten Ansatz zur Umsetzung des Besuchermusters. Sie betrachten jedoch nicht das *Expression Problem* selbst, sondern konzentrieren sich ausschließlich auf die Modularisierung des überschneidenden Belanges zur Auswahl der richtigen `Visit`-Methode in den Datenklassen als Aspekt. In den Datenklassen entfällt somit die Implementation der `Accept`-Methode. Jede neue Datenklasse erfordert jedoch bei dieser Lösung sowohl das Anpassen des Aspekts als auch jeder einzelnen Operationsklasse. Das hat zur Folge, dass der Quellcode jedes Mal neu übersetzt werden muss. Die Definition der Pointcuts (Listing 4.10) in dem von [Schmidmeier u. a. 2003] vorgeschlagenen Aspekt erscheint dem Autor sehr umständlich: Für jede Datenklasse (`A`, `B` und `C`) ist ein eigener Advice definiert, in dem wiederum explizit die jeweils anderen Datenklassen ausgeschlossen werden (`!args(B) && !args(C)`). Eine Erweiterung der Datenklassen wird somit schnell umständlich und fehleranfällig, da jeder einzelne Advice angepasst werden muss.

[Hachani und Bardou 2002] gehen bei ihrem Ansatz den Weg, alle Operationen als Aspekt zu implementieren und die jeweiligen `Visit`-Methoden als Introduktionen mit den Datenklassen zu verweben. Das grundsätzliche Problem bei diesem Ansatz besteht jedoch darin, dass Operationen nicht mehr als eigenständige Exemplare existieren und eine Verwaltung des Zustandes der Operation nur sehr aufwendig möglich ist. Zustandsvariablen, die im Aspekt definiert werden, sind an die Exemplare der Datenklasse gebunden (in AspectJ durch das Schlüsselwort `perthis` oder `pertarget`), global für jede Datenklasse (durch das Schlüsselwort `perclass`) oder einmalig pro Operation (durch das Schlüsselwort `issingleton`). Alle diese Möglichkeiten erlauben jedoch nicht das sichere nebenläufige Ausführen der Operationen.

[Tarr u. a. 1999] zeigt, wie das „Expression Problem“ durch Separieren der einzelnen Belange mit HyperJ realisiert werden kann. Jede Operation auf den Datenklassen wird eigenen sogenannten *Slices* implementiert. Durch die Definition von *Hypermodulen* werden diese *Slices* zur vollständigen Implementation zusammengesetzt. Die Lösung bietet zwar eine hervorragende Modularisierung einzelner Belange, hat aber Nachteile bei der Erweiterbarkeit neuer Ausdrücke und Operationen, denn eine Erweiterung einzelner *Slices* erfordert immer das komplette Übersetzen des *Hypermodules* und somit des gesamten Quelltextes.

	OOP-Besuchermuster	LOOM-Besuchermuster
Zahl der Methoden insgesamt	$data * (2 + visitors) + 1$	$data + 2 \leq x \leq data * visitors + 2$
Anzahl neuer Methoden durch Hinzufügen einer Datenklasse	$visitors + 2$	$0 \leq x \leq visitors$
Geänderte oder hinzugefügte Besucherklassen durch Hinzufügen einer Datenklasse	$visitors + 1$	$0 \leq x \leq visitors$
Anzahl neuer Methoden durch Hinzufügen einer Besucherklasse	$data$	$1 \leq x \leq data$

Tabelle 4.1: Implementationsaufwand des Besuchermusters

	OOP-Besuchermuster	LOOM-Besuchermuster
Zahl der Klassen insgesamt	9	10
Zahl der Methoden insgesamt	21	9

Tabelle 4.2: Implementationsaufwand von EXP

4.7.4 Bewertung des Musters

Bevor eine Bewertung dieses Musters nach dem bekannten Schema erfolgt, soll hier noch einmal der Vorteil herausgearbeitet werden, der entsteht, wenn neue Daten- oder Besucherklassen hinzufügen werden.

Wenn *data* die Anzahl der Datenklassen und *visitors* die Anzahl der Besucherklassen ist, dann ergibt sich für die objektorientierte Variante folgender Aufwand: Jede Datenklasse und die gemeinsame Basisklasse müssen eine `Accept`-Methode implementieren ($data + 1$), hinzu kommen für jede Datenklasse je eine `Visit`-Methode in der Operationsklasse und deren Basisklasse ($data * (visitors + 1)$).

Die LOOM-Lösung benötigt nur eine einzige `Accept`-Methode in der Basisklasse und eine `Visit`-Methode in der Basisklasse der Besucher. Hinzu kommt, dass für jede Datenklasse auch mindestens eine Besucherklasse eine konkrete Überladung von `Visit` bereitstellt (insgesamt mindestens $data + 2$). Maximal enthält jede Besucherklasse für jede Datenklasse eine Überladung ($data * visitors + 2$).

In Tabelle 4.1 ist die Zahl der zu implementierenden Methoden tabellarisch gegenübergestellt. Es ist zu sehen, dass die LOOM-basierte Lösung in jedem Fall einen geringeren Implementationsaufwand bedeutet. Der Aufwand für die Implementation des Aspekts ist hier nicht weiter berücksichtigt worden, da der Aspekt keinen für die Problemdomäne spezifischen Quelltext enthält - er kann ohne Änderung wiederverwendet werden. Außerdem ist der Änderungsaufwand für das Hinzufügen von Daten- und Besucherklassen aufgeschlüsselt. Tabelle 4.2 zeigt den tatsächlichen Aufwand für die Sprache *EXP* mit vier Operatoren und den zwei Operationen zum Anzeigen und Auswerten der Ausdrücke. Hier wird deutlich, dass zwar eine zusätzliche Klasse (`BinaryOperator`) implementiert wurde, insgesamt aber nur weniger als die Hälfte an Methoden benötigt wird, um das Muster zu implementieren.

Kopplung

Bei der Kopplung wird hier berücksichtigt, inwieweit Teile der Besucherklassen von den Datenklassen abhängen und umgekehrt. Nur die Varianten LOOM und HyperJ bieten hier die Möglichkeit einer relativ unabhängigen Implementation.

Modularisierung

In allen vier Varianten können Operationen unabhängig von ihren Daten gekapselt werden.

Redundanz

In der objektorientierten Variante können ähnliche Operationen nicht zusammengefasst werden. AspectJ erlaubt das durch entsprechende Formulierung der Pointcuts. In LOOM werden sie hingegen durch Vererbungsstrukturen vermieden. HyperJ erlaubt das durch eine angepasste Komposition der Operationsklassen.

Implementationsaufwand

Der Implementationsaufwand für einen Anwendungsfall ist bei der objektorientierten Variante durch die zahlreichen `Accept` und `Visit`-Implementationen relativ hoch. In der LOOM-Variante wird das durch einen einfachen Aspekt abgefangen, der sogar unabhängig von der konkreten Implementation und somit wiederverwendbar ist. Bei den Varianten AspectJ und HyperJ ist die Definition der Pointcuts und Hypermodule hingegen relativ komplex.

Gesamtbewertung

Kategorie	**OOP-Muster**	**AspectJ**	**HyperJ**	**LOOM-Muster**
Kopplung	1	1	2	2
Modularisierung	3	3	3	3
Redundanz	1	2	3	3
Implementationsaufwand	2	2	2	3

4.8 Das Stellvertretermuster

Das Stellvertretermuster ist ein Strukturmuster, um den Zugriff auf ein Objekt zu kontrollieren. Im Allgemeinen ist der Stellvertreter eine Klasse, die als Schnittstelle zu einem Subjekt auftritt.

Nach [Gamma u. a. 1995, S. 208 ff.] gibt es folgende Situationen, in denen das Stellvertretermuster angewendet werden kann:

- Ein *Remote-Proxy* oder auch *Ambassador* [Coplien 1992] stellt eine lokale Repräsentation für ein entferntes Objekt dar. Der Stellvertreter leitet hier Aufrufe auf seine Schnittstelle an das entfernte Objekt weiter. Für den Klienten erscheint es so, als wäre das entfernte Objekt lokal.

- Ein virtueller Stellvertreter erzeugt ressourcenintensive Objekte bei Bedarf. Üblicherweise wird das reale Subjekt erst angelegt, wenn die Schnittstelle des Stellvertreters tatsächlich angesprochen wird.

- Ein *Schutz-Proxy* kontrolliert den Zugriff auf das reale Objekt. Er ist sinnvoll, wenn Objekte unterschiedliche Zugriffsrechte haben. Stellvertreter für Kernobjekte im *Choices* Betriebssystem [Campbell u. a. 1993] kontrollieren beispielsweise auf diese Art den Zugriff.

- Eine *Smart Reference* (auch *Smart Pointer*) kann zusätzliche Aktionen ausführen, wenn auf ein Objekt zugegriffen wird. Das wäre zum Beispiel das Zählen der Referenzen zum realen Objekt, um dieses automatisch freizugeben, wenn es nicht mehr verwendet wird [Edelson 1992]. Andere Möglichkeiten sind z. B. der Zugriffsschutz auf das Objekt bei nebenläufiger Verwendung seiner Schnittstelle.

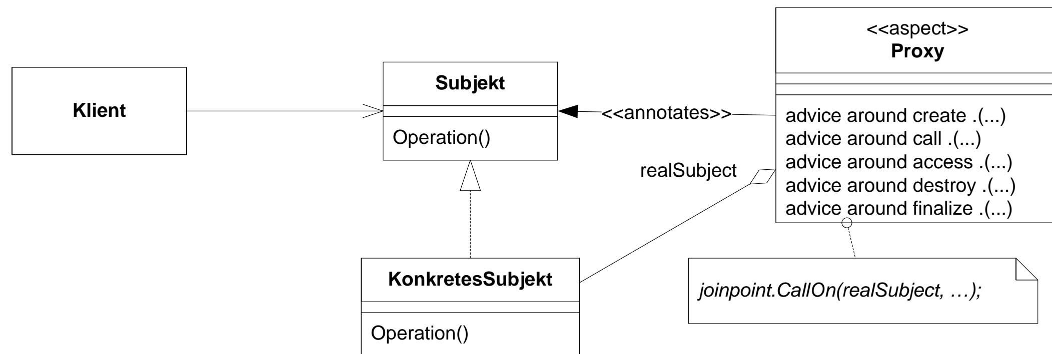


Abbildung 4.10: Struktur des Stellvertretermusters

Die LOOM-Variante des Musters bereichert die Möglichkeiten um eine weitere Facette. Man unterscheidet hier zwischen einem *Klassenstellvertreter* und einem *Methodenstellvertreter*. Während der Klassenstellvertreter sich stets auf das gesamte Subjekt bezieht, bezieht sich der Methodenstellvertreter ausschließlich auf eine oder mehrere explizit ausgewählte Methoden des Subjekts. Ausschließlich Aufrufe dieser Methoden werden vom Methodenstellvertreter behandelt.

Ein weiterer Vorteil des LOOM-basierenden Stellvertreters besteht darin, dass die Stellvertreterlogik völlig unabhängig von der Schnittstelle des Subjekts implementiert werden kann. Das erlaubt generische Stellvertreter, die wiederverwendbar auf die unterschiedlichsten Klassen und Methoden angewendet werden können. Das ist unter anderem mit dem in Kapitel 2 vorgestellten Konzept der Nachrichtenmanipulation möglich.

4.8.1 Die Struktur des LOOM-Stellvertreters

Die Struktur des LOOM-Stellvertreters ist in Abbildung 4.10 dargestellt. Die Aufgabe der Klasse `Subjekt` besteht einzig darin, die Schnittstelle zu definieren. Sie hat üblicherweise keine eigene Logik und wird als Hülle für den Stellvertreter verwendet. Die `Subjekt`-Klasse wird mit dem `Stellvertreter`-Aspekt annotiert.

Der Aspekt definiert Advices, um die Aktionen Erzeugung, Benutzung der Schnittstelle und Zerstörung des Subjekts zu kontrollieren. Diese Advices sind so formuliert, dass sie stets alle Joinpoints des Subjekts selektieren. Im Unterschied zum Dekorierer (Abschnitt 4.3) ist die Implementierung des Stellvertreters immer unspezifisch und unabhängig vom Subjekt. Während ein Dekorierer zusätzliche Funktionen zu einer Methode hinzufügt, kontrolliert der Stellvertreter den Zugriff auf eine Methode.

Erfolgt durch den Klienten ein Methodenaufruf auf der `Subjekt`-Schnittstelle, wird dieser automatisch durch den Aspekt abgewickelt. Über den entsprechenden Advice hat der Aspekt Zugriff auf den Joinpoint-Kontext. Dieser repräsentiert den Aufruf als Objekt und wird zum Aufruf der entsprechenden Methode von `KonkretesSubjekt` verwendet. Dazu wird die Methode `CallOn` (in LOOM.NET auch `InvokeOn`) des Joinpoint-Kontextes mit der Referenz von `KonkretesSubjekt` als Parameter benutzt.

Ein `CallOn` auf einem Joinpoint-Kontext ändert den Adressaten des Methodenaufrufs. Dieser kann vom Advice explizit angegeben werden. Der Aufruf wird so auf ein anderes Objekt umgelegt, ohne dass der Advice Kenntnis von der Methodensignatur oder dem Objekttyp haben muss. Dieser Mechanismus ermöglicht es, Stellvertreter unabhängig vom Subjekt und seinen Methoden zu implementieren.

Besonders hervorzuheben ist hier, dass der Joinpoint-Kontext nicht an den lokalen Adressraum gebunden ist. Als Objekt erster Ordnung kann der Kontext durchaus Serialisiert und Persistiert werden. Das wird durch interne LOOM.NET-Mechanismen explizit unterstützt. Damit kann der Kontext in einen anderen Adressraum transportiert werden und ermöglicht

```
public class Fibonacci
{
    [Memoized]
    public virtual ulong Calculate(uint n)
    {
        if (n < 2)
            return n;
        else
            return Calculate(n - 1) + Calculate(n - 2);
    }
}
```

Listing 4.11: Berechnung von Fibonacci-Zahlen

somit vielfältigste Stellvertreterimplementierungen.

Während Abbildung 4.10 einen typischen Klassenstellvertreter beschreibt, ist der Methodenstellvertreter noch etwas einfacher aufgebaut: Beim Methodenstellvertreter ist die Klasse `Subjekt` nicht notwendig. Der `Stellvertreter`-Aspekt wird hier direkt mit der Klasse `KonkretesSubjekt` verwoben, indem die zu kontrollierende Methode mit dem Aspekt annotiert wird.

4.8.2 Beispiel

Es gibt Situationen, in denen es sinnvoll sein kann, Rückgabewerte von Methoden zwischenzuspeichern. Wird die Methode zu einem späteren Zeitpunkt mit denselben Parametern erneut aufgerufen, kann sofort auf den gespeicherten Wert zurückgegriffen werden. Diese unter dem Namen *Memo-Function* oder *Memoiser* von [Michie 1968] beschriebene Technik ist besonders dann interessant, wenn die entsprechende Methode sehr resourcenintensive Berechnungen durchführt. Voraussetzung zum „Memorisieren" einer Methode ist, dass sie *referenziell transparent* d. h. frei von *Seiteneffekten* ist. Das bedeutet, dass sie bei gleichen Eingabeparametern immer dasselbe Ergebnis liefert.

Mit dem LOOM-Stellvertreter kann die Logik der Zwischenspeicherung separat in einem Aspekt implementiert werden. Diejenigen Methoden, die memoisiert werden sollen, werden dann mit diesem Aspekt annotiert. Dieses Beispiel eines Methodenstellvertreters soll an der Berechnung der n-ten Fibonacci Zahl [Fibonacci 1857, S. 283 f.] veranschaulicht werden.

Listing 4.11 zeigt die Klasse `Fibonacci`, in der die Methode `Calculate` zur rekursiven Berechnung der n-ten Fibonacci-Zahl implementiert ist. Offensichtlich wird für jeden Berechnungsschritt das Ergebnis der beiden vorangegangenen Zahlen benötigt. Diese rekursive Implementation führt dazu, dass `Calculate` eine Komplexität von $O(1.6^n)$ hat, wobei n die Nummer der Fibonacci-Zahl ist. Die Rekursion muss bei der Berechnung immer wieder Fibonacci-Zahlen berechnen, die in vorangegangenen Rekursionsschritten bereits bekannt waren. Dieser Effekt vergrößert sich exponentiell mit der Größe der zu berechnenden Zahl.

Memoisiert man die Methode `Calculate`, kann ein vorher bereits ermitteltes Ergebnis ohne erneute Berechnung zurückgeliefert werden. Erfolgt die Memoisierung der Methode durch Annotation mit dem Aspekt `[Memoized]` (Zeile 3), muss die Implementation von `Calculate` nicht geändert werden. Dieser Aspekt ist in Listing 4.12 abgebildet.

Der Aspekt besteht aus einem Zwischenspeicher (`cache`) (Zeile 3) für die Zuordnung von Parameterwerten und Methodenergebnissen sowie einem Advice (Zeile 7), der die Aufgabe des Stellvertreters übernimmt. Jeder Aufruf einer mit dem Aspekt annotierten Methode (hier `Calculate`) wird zuerst vom Advice behandelt. Dieser versucht den übergebenen Parametern einen Rückgabewert zuzuordnen (Zeile 12). Der Zwischenspeicher ist eine Hash-Tabelle, deren Schlüssel-Wert-Paar jeweils aus einem `ArgVector`-Objekt und dem Rückgabewert besteht. `ArgVector` ist eine hier nicht weiter ausgeführte Hilfsklasse, um die Methodenparameter zusammenzufassen, da die Methode potentiell auch mehrere Parameter haben kann. Mit

```
1  public class Memoized : AspectAttribute
2  {
3      private Dictionary<ArgVector, object> cache =
4        new Dictionary<ArgVector, object>();
5
6      [Call(Advice.Around)]
7      [IncludeAll]
8      public T Memoizer<T>([JPContext] Context ctx, params object[] args)
9      {
10         ArgVector newArray = new ArgVector(args);
11         object result;
12
13         if (!cache.TryGetValue(newArray, out result))
14         {
15             result = ctx.Invoke(args);
16             cache.Add(newArray, result);
17         }
18
19         return (T)result;
20     }
21 }
```

Listing 4.12: Der Memoized Aspekt

```
1  [Versionen(typeof(Version1), typeof(Version2), typeof(Version3)]
2  public class Subjekt
3  {
4    ...
5  }
6
7
8  public class Version1 : Subjekt
9  {
10    ...
```

Listing 4.13: n-Versionen-Programmierung

der Hilfsklasse können die Parameterlisten verglichen werden, und es wird ein eindeutiger Hash-Wert für die Hash-Tabelle zur Verfügung gestellt. Kann für die übergebenen Parameter kein Rückgabewert gefunden werden, wird über den Joinpoint-Kontext die originale Methode aufgerufen und das Ergebnis im Zwischenspeicher abgelegt (Zeile 14-15).

Ist der Memoized-Aspekt einmal definiert, kann er auf beliebige Methoden, die der eingangs besprochenen Voraussetzung der referenziellen Transparenz genügen, angewendet werden. Bis auf die Annotation der Methode sind keine weiteren Änderungen am Quelltext notwendig.

Der Klassenstellvertreter soll am Beispiel des n-Versionen-Stellvertreters erläutert werden. N-Versionen-Programmierung ist ein Programmierparadigma, um fehlertolerante Software zu entwickeln [Arlat u. a. 1990; Avizienis 1985; Cristian 1991; Laprie 1985, 1992]. Werden verschiedene funktional äquivalente Module, ausgehend von derselben Spezifikation, unabhängig voneinander entwickelt, so lautet die These, dass die Wahrscheinlichkeit für das Vorhandensein des gleichen Softwarefehlers in allen Modulen relativ gering ist. An dieser Stelle soll es weniger um die Details einer Fehlertoleranzlösung gehen, vielmehr soll relativ einfach die Umsetzung der n-Versionen-Programmierung mit LOOM.NET und dem Stellvertretermuster im Vordergrund stehen.

Mit dem LOOM-Stellvertretermuster lassen sich hinter einer Klasse verschiedene konkrete Implementierungen dieser Klasse verbergen. Mit relativ einfachen Mitteln lässt sich so eine n-Versionen-Programmierung auf Klassenebene realisieren. Die Schnittstellenspezifikation für die unabhängig voneinander zu entwickelnden Module stellt hierbei die `Subjekt`-Klasse dar. Jede Version wird von dieser Klasse in einer Versionsklasse abgeleitet und muss die Methoden der `Subjekt`-Klasse implementieren (Listing 4.13).

Der nachfolgende Aspekt ist geeignet, Softwarefehler während der Laufzeit zu erkennen und zu kompensieren. Als Softwarefehler gelten hier ausschließlich unbehandelte Ausnahmen oder Berechnungsfehler. Fehler, die sich darin äußern, dass ein Schnittstellenaufruf nicht zum Aufrufer zurückkehrt, können nicht erkannt werden.

Die Aspektimplementation nimmt eine Replikation eines Schnittstellenaufrufs vor und verteilt sie auf die paarweise verschiedenen Versionen. Die Repliklation erfolgt auf der Zeitachse; die Implementierungen werden sequentiell nacheinander abgearbeitet. Es müssen folgende Randbedingungen gelten:

- Die einzelnen Versionen dürfen keine gemeinsamen Ressourcen verwenden.
- Ein Schnitstellenaufruf ist bezüglich seiner Zeitdauer immanent.

Als das richtige Ergebnis wird das angesehen, das von der Mehrheit aller Versionen zurückgeliefert wurde. Existieren zwei Versionen einer Schnittstelle, ist bei korrekter Umsetzung der Spezifikation und gleicher Eingabe dieselbe Ausgabe beider Versionen zu erwarten. Bei einem Totalausfall einer Implementation (eine unbehandelte Ausnahme) kann immer noch das Ergebnis der jeweils anderen Klasse verwendet werden. Unterscheiden sich die Ergebnisse beider Klassen, kann auf einen Fehler geschlossen werden. Wird die Zahl der Versionen erhöht, lassen sich auch Berechnungsfehler durch einen Auswahlalgorithmus kompensieren.

In Listing 4.14 ist ein Stellvertreter-Aspekt für eine einfache n-Versionen-Programmierung implementiert. Mit dem Aspekt wird die `Subjekt`-Klasse, von der alle Versionen ableiten, annotiert. Der Aspekt übernimmt dabei als Parameter die Typobjekte der verschiedenen Versionen (Zeile 7). Fordert ein Klient ein Exemplar der `Subjekt`-Klasse an, so wird zuerst der **create**-Advice des Aspekts ausgeführt (Zeile 12-20). Dieser legt in der Joinpoint-Variable `versionen` für jeden Versionstyp ein Exemplar an (Zeile 18).

Der **call**-Advice ist mit jeder Methode der Basisklasse `Schnittstelle` verwoben und leitet den Aufruf an die in `versionen` gespeicherten Referenzen weiter (Zeile 22-64). Der Advice verwendet hierzu die `InvokeOn`-Methode des Joinpoint-Kontextobjektes (Zeile 34). Die Ergebnisse jedes Versionsobjektes werden in einer Hash-Tabelle (`ergebnisse`) abgespeichert, wobei der Schlüssel das Ergebnis selbst ist und der Wert die Häufigkeit (Zeile 40-48). Wurde beim Aufruf eine Ausnahme geworfen, wird das nicht in die Wertung einbezogen (Zeile 50). Abschließend sucht der Algorithmus das Ergebnis, das am häufigsten aufgetreten ist und gibt es an den Aufrufer zurück (Zeile 52-61). Gibt es keine Mehrheit für ein Ergebnis, wird eine Ausnahme geworfen (Zeile 62).

Als Erweiterung des Aspekts ist z. B. eine Replikation der Schnittstellenaufrufe in verschiedenen Threads denkbar. Mit der Parallelisierung können zusätzlich Zeitüberschreitungen einzelner Versionen erkannt und als Fehler interpretiert werden. Für eine zusätzliche Abschottung ist es möglich, die Versionen in eigenen Adressräumen (Applikationsdomänen) abzuwickeln. Beispiele hierzu sind in der LOOM.NET-Dokumentation [Schult 2008] zu finden.

Auch dieser Stellvertreteraspekt ist so allgemein gehalten, dass er auf die unterschiedlichsten Versionenschnittstellen (Subjekte) angewendet werden kann. Eine Spezialisierung ist nicht notwendig.

4.8.3 Vergleich zu anderen Lösungen

Die Stellvertreterimplementierung von [Hannemann 2005; Hannemann und Kiczales 2003] basiert auf einem abstrakten Aspekt, von dem die konkreten Stellvertreteraspekte ableiten sollen. In diesem Basisaspekt ist ein Pointcut definiert, der sämtliche Aufrufe an ein potentielles Subjekt abfängt. Dieses muss dann in der Ableitung des Aspekts durch Konkretisierung des Pointcuts näher spezifiziert werden.

Jeder konkrete Stellvertreteraspekt muss zusätzlich die virtuelle Methode (`isProxyProtected`) überschreiben. Diese wird bei jedem Zugriff auf die Subjektschnittstelle aufgerufen.

```
1 [AttributeUsage(AttributeTargets.Class, Inherited=false)]
2 public class Versionen:AspectAttribute
3 {
4     private Type[] versionsklassen;
5     static object c_ungueltig = new object();
6
7     public Versionen(params Type[] versionsklassen)
8     {
9         this.versionsklassen = versionsklassen;
10    }
11
12    [Create(Advice.After), IncludeAll]
13    public void Create([JPVariable(Scope.Protected)] ref object[] versionen, params
          object[] args)
14    {
15        versionen = new object[versionsklassen.Length];
16        for (int iPos = 0; iPos < versions.Length; iPos++)
17        {
18            versionen[iPos] = Weaver.CreateInstance(versionsklassen[iPos], args);
19        }
20    }
21
22    [Call(Advice.Around), IncludeAll]
23    public T Call<T>([JPContext] Context ctx, [JPVariable(Scope.Protected)] object[]
          versionen, params object[] args)
24    {
25        object res=c_invalid;
26        int count;
27        Dictionary<object, int> ergebnisse = new Dictionary<object, int>(versionen.
              Length);
28        foreach (object version in versionen)
29        {
30            try
31            {
32              try
33              {
34                res = ctx.InvokeOn(version, args);
35              }
36              catch(ApplicationException e)
37              {
38                res = e;
39              }
40              if (ergebnisse.TryGetValue(res, out count))
41              {
42                  count++;
43                  ergebnisse[res] = count;
44              }
45              else
46              {
47                  ergebnisse.Add(res, 1);
48              }
49            }
50            catch{Exception e} {}
51        }
52        count = 0;
53        foreach(KeyValuePair<object,int> kvpair in ergebnisse)
54        {
55            if (kvpair.Value > count)
56            {
57                count = kvpair.Value;
58                res = kvpair.Key;
59            }
60            else if (kvpair.Value == count) res = c_ungueltig;
61        }
62        if (res == c_ungueltig) throw new ApplicationException();
63        return (T)res;
64    }
65 }
```

Listing 4.14: Ein n-Versionen-Stellvertreter

Möchte der Aspekt den Zugriff behandeln, muss er hier `true` zurückliefern. Das führt in der Folge dazu, dass die Methode `handleProxyProtection` aufgerufen wird und der Stellvertreter seine eigentliche Aufgabe ausführen kann.

Problematisch an der Vorgehensweise ist jedoch, dass keine anonyme Delegation eines Methodenaufrufs (wie mit `CallOn` bzw. `InvokeOn` in LOOM.NET) vom Subjektobjekt auf ein konkretes Subjektobjekt möglich ist. Vielmehr müssen die Methodenaufrufe explizit ausformuliert werden. Das führt zu der Situation, dass für jede Subjektmethode hier ein eigener Advice definiert werden muss. Somit existiert kein Vorteil gegenüber dem objektorientierten Stellvertreter.

4.8.4 Bewertung des Musters

Kopplung

Sowohl die AspectJ- als auch die objektorientierte Variante benötigt stets für jede Schnittstelle eine konkrete Ausprägung eines Stellvertreters. Das bedeutet eine hohe Kopplung an die konkrete Schnittstelle. Die LOOM-Variante lässt hier eine generische Lösung für die unterschiedlichsten Schnittstellen zu. Sie kann unabhängig in einer Komponente implementiert und wiederverwendet werden. Das gilt für die anderen beiden Varianten nicht.

Modularisierung

In allen Varianten wird der Stellvertreter in einer eigenen Klasse bzw. einem eigenen Aspekt implementiert.

Redundanz

In der objektorientierten Variante und in der AspectJ-Lösung muss eine Implementation für jede einzelne Methode der Schnittstelle erfolgen. In der LOOM-Variante geschieht das in einer einzigen Methode.

Implementationsaufwand

Der Implementationsaufwand für eine Methode ist in der objektorientierten Variante und in der AspectJ-Lösung bereits höher als in der LOOM-Variante, da gegen eine konkrete Schnittselle programmiert werden muss.

Gesamtbewertung

Kategorie	OOP-Muster	AspectJ	LOOM-Muster
Kopplung	1	1	3
Modularisierung	3	3	3
Redundanz	1	1	3
Implementationsaufwand	2	2	3

4.9 Der Ereignisabonnent

Ereignisprogrammierung ist ein probates Mittel zur Steuerung des Programmflusses durch externe Ereignisse, wie Benutzereingaben oder Sensorausgaben. Ein Beispiel ist die Programmierung grafischer Benutzeroberflächen. Wenn der Benutzer auf eine Schaltfläche klickt, löst das ein Ereignis aus. In der Folge muss dieses an die richtige Methode zur Bearbeitung übergeben werden. Das Ereignisabonnenten-Muster stellt hierzu eine leichtgewichtige Lösung dar.

In der objektorientierten Welt hat sich für die Ereignisprogrammierung das *Publish-Subscribe*-Muster etabliert, wie es beispielsweise oft Anwendung in verteilten Systemen findet

```
1 public delegate void EreignisTyp(object sender, System.EventArgs e);
2
3 class Ereignisanbieter
4 {
5     public event EreignisTyp Ereignis;
6
7     public void LoeseEreignisAus(System.EventArgs args)
8     {
9       // Ereignis ist null, wen kein Delegate registriert wurde
10      if (Ereignis != null)
11      {
12        Ereignis(this, args);
13      }
14    }
15 }
16
17 class Abonnent
18 {
19   protected void Ereignismethode(object sender, System.EventArgs e)
20   {
21     // ...
22   }
23 }
24
25 Ereignisanbieter anbieter=new Ereignisanbieter();
26 anbieter.Ereignis+=new Ereignistyp(Ereignismethode);
27
28 anbieter.LoeseEreignisAus(new System.EventArgs());
```

Listing 4.15: Beispiel für Ereignisprogrammierung in C#

[vgl. Eugster u. a. 2003]. Hier gibt es zwei Rollen, die ein Objekt annehmen kann: Zum einen gibt es die Rolle des *Anbieters*, das sind Objekte, die Ereignisse veröffentlichen. Zum anderen existiert die Rolle des *Abonnenten*, also Objekte, die Ereignisse konsumieren. Ein Abonnent kann sich zur Laufzeit bei einem Ereignisanbieter an- und abmelden. Tritt das Ereignis ein, werden alle registrierten Abonnenten vom Ereignisanbieter benachrichtigt.

Die in .NET-Umgebungen empfohlene technische Realisierung sieht so aus, dass der Abonnent dem Ereignisanbieter eine Methode übergibt, die der Ereignisanbieter beim Auftreten des Ereignisses aufrufen soll [Microsoft Corporation 2003a, §10.7]. .NET bietet hierzu das Konzept der *Delegates* [Microsoft Corporation u. a. 2006, Teil I, S. 35]. Mit einem Delegate-Exemplar kann man auf eine Methode eines (Abonnenten-)Objektes zeigen und diese aufrufen. Die Programmiersprache C# bietet zusätzlich mit dem Schlüsselwort `event` [Microsoft Corporation 2003a, S. 237 ff.] ein vordefiniertes Muster, um die Registrierung von Delegate-Exemplaren und den Aufruf der gebundenen Methoden zu vereinfachen. Die Ereignisse des Anbieters werden mit diesem Schlüsselwort als Klassenattribut deklariert (Listing 4.15).

Die Delegate-Deklaration `Ereignistyp` (Zeile 1) beschreibt die Deklaration der `Ereignismethode`. Abstrakt kann sie als die Art des auftretenden Ereignisses aufgefasst werden. Das mit dem Schlüsselwort `event` versehene Attribut des Anbieters (Zeile 5) speichert die Delegates, über die die Ereignismethoden beim Auslösen des Ereignisses gerufen werden (Zeile 28).

Die Bindung von Ereignisanbieter und Abonnent erfolgt auf der Objektbasis (Zeile 26). Dabei kann eine Ereignismethode eines Abonnentenobjekts an beliebig viele Anbieterobjekte gebunden werden, die einen passenden Ereignistyp anbieten. Umgekehrt kann ein Ereignisattribut eines Anbieterobjekts mit beliebig vielen passenden Delegates der Ereignismethoden verknüpft werden. Obwohl der Mechanismus flexibel ist, ist er trotz Sprachunterstützung schreibintensiv. Der Programmierer muss für jedes Ereignis:

- einen *Delegatetyp* definieren oder den generischen Typ `EventHandler` verwenden, um später die Ereignismethode des Abonnenten zu binden,

- in der Anbieterklasse das Ereignis mit dem `event` und dem *Delegatetyp* definieren,
- eine Methode definieren, die das Ereignis auslöst, wenn mindestens ein Abonnent vorhanden ist,
- in jedem Abonnenten die Ereignismethode an den Anbieter programmatisch binden.

In Java bietet für die Ereignisbehandlung das Interface `ActionListener` nach dem objektorientierten Besuchermuster an. Jeder Abonnent muss dieses Interface implementieren, um sich bei einem Ereignisanbieter registrieren zu können. Da eine Klasse ein Interface nicht mehrfach implementieren kann, ist bei dieser Implementierung allerdings auch nur eine Ereignismethode pro Klasse möglich. Das dadurch etwas relativiert, dass die Ereignismethoden oft als anonyme Klassen implementiert werden [Gosling u. a. 2005]. Der Schreibaufwand bleibt trotzdem relativ hoch.

Der Ereignisabonnent stellt eine leichtgewichtige Alternative dar. Die Anbieter-Abonnent-Beziehung wird hier nicht erst zwischen zwei Objekten zur Laufzeit hergestellt, sondern bereits deklarativ zum Zeitpunkt der Programmierung. Das Ereignis wird direkt an diejenigen Methoden gebunden, die es behandeln sollen. Das geschieht durch Annotation der Ereignisbehandlungsmethoden mit dem Ereignis(-typ). Beim Lesen des Quelltexts wird sofort ersichtlich, welche Methode welche Ereignisse konsumiert.

Dieses Muster wird vor allem dann angewendet, wenn die Ereignisbindung an eine Ereignismethode eines Objekts dauerhaft während der gesamten Lebenszeit des Objekts erhalten bleiben soll und es sich um eine lokale Ereignisbehandlung handelt. Lokal meint hier, dass das Auslösen und Empfangen des Ereignisses im selben Prozess erfolgt. Verteilte Systeme werden hier nicht betrachtet, da hier eine weit aufwendigere Kommunikation notwendig ist. Ein Ereignisanbieter wird bei diesem Muster nicht benötigt.

Der Ereignisabonnent besitzt einige Parallelen zum Beobachtermuster aus Abschnitt 4.6. Beim Beobachter geht es darum, implizit ein Ereignis auszulösen, wenn sich ein zu beobachtendes Objekt geändert hat, während beim Ereignisabonnenten das Ereignis explizit ausgelöst wird.

4.9.1 Die Struktur des Loom-Ereignisabonnenten

In Abbildung 4.11 sind die beteiligten des Ereignisabonnentenmusters dargestellt. Kern ist der abstrakte Aspekt (`Ereignis`). Jedes konkrete Ereignis wird durch eine Ableitung von diesem Aspekt definiert. Weiterhin existieren beliebig viele Abonnentenklassen (hier die Klasse `Abonnent`). In einer Abonnentenklasse ist die Ereignisbehandlungsmethode definiert, die gerufen werden soll, wenn das konkrete Ereignis eintritt. Sie wird mit dem entsprechenden Aspekt annotiert. Damit wird explizit verdeutlicht, dass diese Methode das Ereignis konsumieren möchte. Als Parameter akzeptiert die Methode ein Exemplar des konkreten Ereignisses:

```
[KonkretesEreignis]
void Ereignismethode(KonkretesEreignis arg)
{
  ...
}
```

Der Typ des Parameters der Methode kann dabei auch ein allgemeinerer Typ als der des konkreten Ereignisses sein. Er muss jedoch mindestens vom Typ `Ereignis` sein. Das gilt insbesondere wenn - was bei diesem Muster durchaus legitim ist - diese Methode mehrere Ereignisse unterschiedlichen Typs konsumiert. In diesem Fall muss der Parameter vom Typ einer gemeinsamen Basisklasse aller Ereignisse, mindestens jedoch vom Typ `Ereignis`, sein.

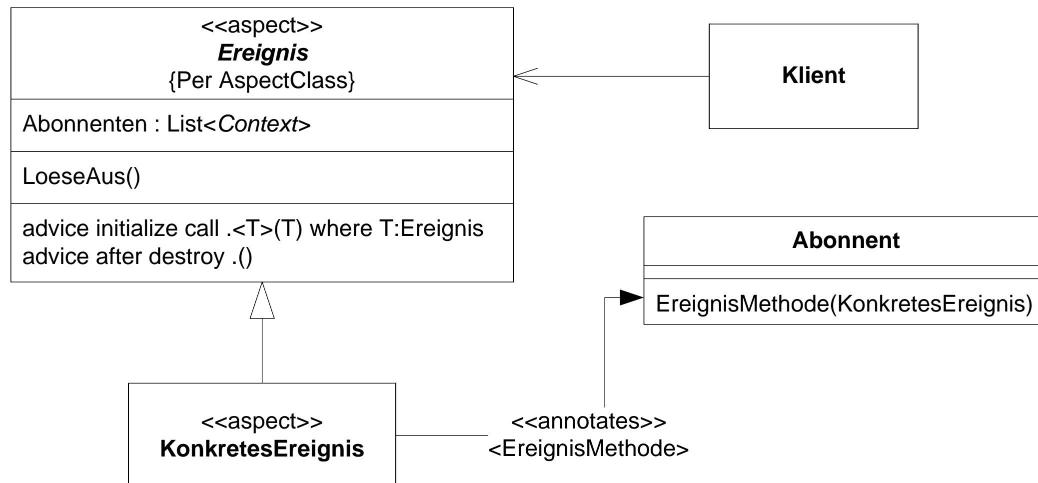


Abbildung 4.11: Die Struktur des Ereignisabonnenten

Der `Ereignis`-Aspekt definiert zwei Advices: Der **initialize**-Advice wird in die Initialisierung eines Abonnenten-Objektes eingewoben. Wird ein Abonnent erstellt, wird für jede Methode, die mit einem von `Ereignis` abgeleiteten Aspekt annotiert ist, dieser Advice-Initialisierer aufgerufen. Der Aspekt merkt sich in der Liste `abonnenten` den Joinpointkontext der verwobenen Methode. Damit ist die Ereignismethode und das zugehörige Objekt nach der Initialisierung an den Aspekt gebunden. Jeder von `Ereignis` abgeleitete Aspekt - und damit jedes konkrete Ereignis - führt eine solche Liste. Das wird implizit durch die Aspekteigenschaft **per aspectclass** realisiert. Sie besagt, dass es für jeden Aspekttyp nur ein Aspektobjekt zur Laufzeit geben wird. Folglich werden die Advices eines konkreten Ereignisses, die der Aspekt vom abstrakten `Ereignis`-Aspekt erbt, immer auf demselben Aspektobjekt ausgeführt.

Durch den **destroy**-Advice wird die Freigabe der Objekte des Abonnenten verwoben. Die Freigabe von Objekten muss durch einen `Dispose`-Aufruf erfolgen. Hier werden alle Aufrufkontexte, die zum freizugebenden Objekt gehören, wieder aus der `Abonnenten`-Liste entfernt. Notwendig ist dieser Schritt, um Referenzen des Aspekts auf den Abonnenten zu löschen, damit dieser von der Freispeicherverwaltung ordnungsgemäß freigegeben werden kann.

Die Methode `LoeseAus` löst das Ereignis aus. Sukzessive wird jede als Kontextobjekt in der Abonnentenliste registrierte Methode mit dem Aspekt selbst als Parameter aufgerufen:

```
KonkretesEreignis e=new KonkretesEreignis();
e.LoeseAus();
```

Auf den ersten Blick erscheint es etwas ungewöhnlich, dass hier explizit ein Exemplar eines Aspekts erzeugt wird. Tatsächlich nimmt der Aspekt hier eine weitere Rolle ein. Er ist nicht nur verantwortlich dafür, die Bindung zu den Ereignismethoden herzustellen, er dient auch als Container für zusätzliche Informationen über das Ereignis. Hierzu wird der Aspekt `KonkretesEreignis` üblicherweise um zusätzliche Attribute erweitert. Dieses explizit gebildete Exemplar wird beim Auslösen des Ereignisses automatisch an alle gebundenen Ereignisse als Parameter übergeben.

Zur Laufzeit existieren dann tatsächlich mindestens zwei Exemplare des Aspektes beim Auslösen des Ereignisses: Ein Exemplar hält die Bindungsinformationen zu allen Ereignismethoden, das zweite dient als Ereignisparameter. Auf letzterem wird die `LoeseAus`-Methode aufgerufen. Wie von diesem Exemplar die Bindungsinformationen erreicht werden können, ist in Listing 4.16 dargestellt. Über die LOOM.NET-Reflektionsinformationen können die Exemplare eines Aspektes erfragt werden, die aufgrund einer Annotationsbeziehung mit den Abbonenten erzeugt wurden (Zeile 23). Für jedes von `Ereignis` abgeleitete konkrete Ereignis existiert aufgrund der **per aspectclass**-Eigenschaft (Zeile 1) stets nur ein einziges

```
1  [CreateAspect(Per.AspectClass)]
2  public abstract class Ereignis:AspectAttribute
3  {
4      private List<Context> abonnenten=new List<Context>();
5
6      [Call(Advice.Initialize), IncludeAll]
7      public void Initialize<T>([JPContext] Context ctx, T theevent) where T:Ereignis
8      {
9          abonnenten.Add(ctx);
10     }
11
12     [Destroy(Advice.After)]
13     public void Destroy([JPTarget] object target)
14     {
15         for(int iPos=0;iPos<abonnenten.Count;iPos++)
16         {
17             if (abonnenten[iPos].Instance == target) abonnenten.RemoveAt(iPos);
18         }
19     }
20
21     public void LoeseAus()
22     {
23         Ereignis evt = (Ereignis)AspectAttribute.GetAspect(GetType());
24         if (evt == null) return;
25         foreach(Context ctx in evt.abonnenten)
26         {
27             ctx.Call(this);
28         }
29     }
30 }
31
32 public class KonkretesEreignis:Ereignis {}
```

Listing 4.16: Der Basisaspekt für den Ereignisabonnenten

Exemplar, genau jenes, das die Bindungsinformationen enthält.

Verglichen mit der eingangs vorgestellten *Publish-Subscribe*-Technik ist der einzig aufwendige Teil die Implementierung des bereits in Listing 4.16 dargestellten abstrakten `Ereignis`-Aspekts. Das geschieht jedoch nur ein einziges Mal und kann sogar wiederverwendbar in eine Komponente ausgelagert werden. Um ein neues Ereignis zu definieren, wird nur eine Ableitung dieses Aspektes ohne weitere Implementation benötigt. Das Ereignis wird durch Annotation der Ereignismethode abonniert. Das Auslösen des Ereignisses erfordert lediglich das Aufrufen der `LoeseAus`-Methode auf einem Exemplar des konkreten Ereignisses.

4.9.2 Beispiel

Eine Anwendung soll die Daten eines externen Messgeräts visualisieren. Die Daten müssen vom Gerät über eine proprietäre Schnittstelle im Polling-Verfahren abgerufen werden. Die grafische Oberfläche der Anwendung soll die Daten auf verschiedenste Weise in unterschiedlichen Fenstern visualisieren. Ein Fenster könnte beispielsweise ausschließlich den Gerätestatus anzeigen, während ein anderes die Daten grafisch aufbereitet. Ein drittes Fenster zeigt die Daten in tabellarischer Form an.

In der Anwendung wird das Gerät über einen separaten Thread durch eine Ereignisschleife permanent abgefragt. Dabei können verschiedene Ereignisse auftreten:

- das Gerät wurde angeschaltet,
- das Gerät ist betriebsbereit,
- das Gerät wurde angehalten (Standby),
- es wurden neue Daten empfangen,

```
new Eingeschaltet().LoeseAus();
while(GeraeteAPI.NeueDaten())
{
  GeraeteDaten daten=GeraeteAPI.HoleDaten();
  Ereignis ereignis=null;
  switch(daten.Type)
  {
    case GeraeteStatus.Betriebsbereit:
      ereignis=new Betriebsbereit();
      break;
    case GeraeteStatus.Angehalten:
      ereignis=new Angehalten();
      break;
    case GeraeteStatus.Daten:
      ereignis=new NeueMesswerte(daten.inhalt);
      break;
  }
  if(ereignis!=null) ereignis.LoeseAus();
}
new Ausgeschaltet().LoeseAus();
```

Listing 4.17: Ereignisschleife zum Abfragen des Gerätes

- das Gerät wurde ausgeschaltet.

Die Ereignisschleife ist unabhängig von möglichen Fensterobjekten, die die Daten empfangen sollen. Eine mögliche Implementierung ist in Listing 4.17 dargestellt. Für das Gerät stehen die beiden Methoden `NeueDaten` und `HoleDaten` zur Verfügung. Die Methode `NeueDaten` (Zeile 2) blockiert so lange, bis das Gerät entweder neue Daten liefert oder ausgeschaltet wurde. Letzteres wird durch den Rückgabewert `false` signalisiert. Sind neue Daten vorhanden, so können diese über die Methode `HoleDaten` (Zeile 4) ausgelesen werden. Der Inhalt kann analysiert und in eines der drei Ereignisobjekte `Betriebsbereit`, `Angehalten` oder `NeueMesswerte` umgewandelt werden. Anschließend wird das Ereignis ausgelöst (Zeile 18). Zwei weitere Ereignisse signalisieren den Beginn und das Ende der Ereignisbearbeitung: `Eingeschaltet` und `Ausgeschaltet`.

Hat sich eine Methode für das Ereignis registriert, indem sie mit dem entsprechenden Aspekt annotiert wurde, wird sie mit dem soeben angelegten Exemplar als Parameter aufgerufen:

```
[NeueMesswerte]
void VisualisiereDaten(NeueMesswerte arg)
{
  foreach(Datenreihe dr in arg.inhalt) { ... }
}
```

Diese Ereignismethode wird immer dann aufgerufen, wenn das Gerät neue Daten liefert. Diese sind im Ereignisparameter `arg` enthalten, so wie sie in der Ereignisschleife Zeile 15 übergeben wurden. Die Implementation der verschiedenen Fenster der Benutzerschnittstelle würde für die unterschiedlichen Ereignisse entsprechende Ereignismethoden zur Verfügung stellen.

4.9.3 Varianten des Musters

Es sind verschiedene Abwandlungen dieses Musters möglich. Anstatt neue Ereignisse durch eine Ableitung zu definieren, kann auch jede neue Annotation einer Ereignismethode als neuer Ereignistyp aufgefasst werden. Es existiert in diesem Fall nur ein nicht-abstrakter Ereignisaspekt. Das Ereignis kann durch den voll qualifizierten Namen der Ereignismethode identifiziert werden.

[Olejniczak 2007] macht von dieser Variante Gebrauch, um Ereignisse für die Windows-Fernwartungsschnittstelle (WMI) zu definieren. Olejniczak bildet jede mit dem Ereignisaspekt annotierte Methode auf ein eigenes WMI-Ereignis ab. Im Kapitel 6 wird diese Lösung detaillierter besprochen.

Eine weitere Variante besteht darin, die Objekte der Abonnentenklassen bei der Ereignisweiterleitung genauer zu betrachten. Wenn beispielsweise ein Mausklickereignis aufgetreten ist, sollen nicht alle Ereignismethoden aufgerufen werden, die mit einem `[BeiMausklick]`-Ereignis annotiert wurden, sondern nur diejenigen, deren Objekt (bzw. deren Schaltfläche) tatsächlich angeklickt wurde.

Hierzu ist eine einfache Abwandlung der `LoeseAus`-Methode notwendig. Anstatt sukzessive alle Ereignismethoden hintereinander aufzurufen, würde gezielt zum Ereignis der passende Joinpoint-Kontext aus der Liste gewählt. Beim Mausklick ließe sich das einfach ermitteln, denn das Betriebssystem liefert zu der Art des Ereignisses (Mausklick) auch einen Identifizierer der getroffenen Schaltfläche. Dieser wiederum kann mit dem aus dem Joinpoint-Kontext ermittelten Abonnentenobjekt abgeglichen werden.

In einer abgewandelten Form eignet sich dieses Muster auch zur *Dependency Injection*. Sie ist ein von [Fowler 2004] entwickeltes Entwurfsmuster, bei dem die Verknüpfung eines Objektes mit seiner Umgebung nicht durch das Objekt selbst, sondern durch einen externen Dritten (meist ein entsprechendes Framework) erfolgt. Dadurch werden Abhängigkeitsbeziehungen zwischen Objekten minimiert. Das Entwurfsmuster des Ereignisabonnenten kann hier die Rolle des externen Dritten übernehmen: Das Ereignis steht für ein abhängiges abstraktes Objekt. Mit dem Ereignis werden diejenigen Setter-Methoden annotiert, die später die abhängigen Objekte entgegen nehmen sollen. Der Programmierer drückt bei der Definition seiner Klasse nur aus, dass er z. B. ein Exemplar eines Datenbankdienstes benötigt. Wie das gebildet wird, interessiert ihn hingegen nicht:

```
[Datenbankdienst]
Datenbankdienst datenbank { set; }
```

Wird das Ereignis `Datenbankdienst` zur Laufzeit an diese *Setter*-Methode bei der Bildung eines Abonnentenexemplars gebunden, kann automatisch das Ereignis ausgelöst werden. Das richtige Datenbankdienstexemplar wird automatisch als Ereignisparameter übergeben. Der Abonnent erklärt nur, dass er einen Datenbankdienst benötigt und an welcher Stelle folglich dieser injiziert werden soll. Die Erzeugung des konkreten Exemplars erfolgt zentral im Datenbankdienst-Ereignis und kann unabhängig angepasst werden.

4.9.4 Bewertung des Musters

Kopplung

Dieses Muster befördert die Entkopplung von Objekten, da zur Laufzeit keine expliziten Verbindungen hergestellt werden müssen. Diese sind ausschließlich deklarativ im Quelltext festgelegt und entstehen automatisch durch die Verwebung.

Modularisierung

Der Belang des Ereignisses ist ausschließlich im entsprechenden Aspekt modelliert.

Redundanz

Redundanter Quelltext kommt bei diesem Muster nicht vor.

Muster	Kopplung	Modularisierung	Redundanz	Implementierung
Einzelstück	+	+ (0)	+	+ (0)
Dekorierer	+	0	+	0
Adapter	+	0	+	-
Klassenkomposition[a]	0	0	0	0
Beobachter	+	+	+ (0)	0
Besucher	+ (0)	0	+ (0)	+
Stellvertreter	+	0	+	+

[a]im Vergleich zu Mehrfachvererbung, Mixins, sowie Erweiterungsmethoden

Abbildung 4.12: Gesamtbewertung der Loom-Muster

Implementationsaufwand

Die Basisklasse des Ereignisses wird einmal implementiert und kann wiederverwendet werden. Für konkrete Ereignisse ist nur eine Ableitung ohne Implementation notwendig.

Gesamtbewertung

Kategorie	LOOM-Muster
Kopplung	3
Modularisierung	3
Redundanz	3
Implementationsaufwand	3

4.10 Zusammenfassung

Mit Hilfe von Entwurfsmustern können typische, immer wiederkehrende Entwurfsprobleme gelöst werden. In Verbindung mit den LOOM-Konzepten kann das an vielen Stellen effizienter geschehen, als es mit den bekannten objektorientierten Konzepten der Fall ist.

In Abbildung 4.12 ist der Vergleich der LOOM-Entwurfsmuster zu den bekannten objektorientierten Entsprechungen bzw. den anderen in den entsprechenden Abschnitten vorgestellten Implementierungen noch einmal als Übersicht dargestellt. Ein *„+“* bedeutet, dass das LOOM-Entwurfsmuster im Vergleich zu den anderen vorgestellten Varianten zu einer Verbesserung geführt hat. Eine *„0“* steht für eine gleichwertige Lösung, ein *„-“* für eine Verschlechterung. Wurde zusätzlich eine Null in Klammern *„(0)“* angegeben, ist eine Verbesserung nur gegenüber der objektorientierten Implementation gegeben. In diesem Fall gibt es auch andere gleichwertige Varianten.

In der Summe zeigt sich, dass die vorgestellten LOOM-Muster eine deutliche Verbesserung gegenüber den bekannten Implementationen bedeuten. Besonders deutlich wird das bei der Kopplung und der Redundanz. Die Kopplung des *Einzelstück-*, *Beobachter-* und des *Stellvertretermusters* konnte sogar so weit reduziert werden, dass diese wiederverwendbar als eigene Komponente implementiert werden können. Das ist bei keiner der anderen evaluierten Varianten dieser Muster möglich. Im begrenzten Umfang gilt diese Aussage auch für das *Adapter-* und das *Dekoriermuster*.

Wird die Definition eines Entwurfsmusters nach [Gamma u. a. 1995, S. 3] betrachtet, so ist ein solches wie folgt definiert:

> Design patterns are not about designs ... that can be encoded in classes and reused as is.
>
> ...

> The design patterns ... are descriptions of communication objects and classes that are customized to solve a general design problem in a particular context.

Die wiederverwendbaren Implementationen wären nach dieser Definition keine Entwurfsmuster. Sie können vollständig als Klassen (Aspekte) implementiert und müssen nicht für neue Anwendungsfälle angepasst werden. Interessant ist, dass das sowohl für Verhaltens-, Struktur- als auch für Erzeugungsmuster gilt. Offensichtlich handelt es sich bei diesen Mustern eigentlich um Belange, die in objektorientierten Sprachen nur überschneidend realisiert werden können. Mit LOOM.NET können diese Belange modularisiert werden.

Da die LOOM-Muster Redundanzen vermeiden, führt das bei der Anwendung auf viele Klassen am Ende zu weniger und übersichtlicherem Quelltext. Das *Besuchermuster* benötigt z. B. im besten Fall nur knapp über die Hälfte der Methoden einer objektorientierten Variante. Die LOOM-Variante des Musters löst auch das Dilemma des „Expression Problems“ [Wadler 1998], das eine einfache Erweiterbarkeit bisher nur entweder für die Verhaltensstrukturen oder für die Datenstrukturen erlaubte. Auch das *Stellvertretermuster* sei hier noch einmal genannt. Während die objektorientierte Variante stets voraussetzt, dass die Stellvertreterlogik für jede Stellvertretermethode erneut zu implementieren ist, ist das bei der hier vorgestellten Lösung nur einmal nötig.

Außerdem wurden zwei neue Muster vorgestellt. Das *Klassenkompositionsmuster* erlaubt es, auf einfachem Wege nachträglich neue Belange in verschiedene Klassen gleichzeitig einzubringen, ohne dabei komplizierte Vererbungsstrukturen aufbauen zu müssen. Mit dem *Ereignisabonnenten* steht dem Programmierer ein Entwurfsmuster zur Verfügung, das eine deklarative lose Kopplung von Ereignissen an Klassen ermöglicht.

Insgesamt demonstrieren die Entwurfsmuster, wie die Konzepte von LOOM in der Programmierpraxis umgesetzt werden können und vor allem, wie der Entwickler überschneidende Belange vermeiden kann.

5 Programmierkonzepte

Programmiersprachen unterstützen verschiedene *Programmierkonzepte*, die sich oft in den Schlüsselwörtern und der Grammatik der Sprache widerspiegeln. Ein Programmierkonzept ist beispielsweise die objektorientierte Programmierung; viele Programmiersprachen, die das Konzept der objektorientierten Programmierung unterstützen, haben ein Schlüsselwort zum Definieren einer Klasse.

Während man in dynamischen Programmiersprachen - wie *Lisp* - durchaus in der Lage ist, neue Programmierkonzepte in die Sprache einzufügen, ist das in statischen Programmiersprachen im Allgemeinen mit einem großen Aufwand verbunden: Das erfordert meist das Anpassen der zur Programmiersprache gehörenden Werkzeuge, insbesondere des Kompilers.

In diesem Kapitel soll gezeigt werden, wie mit dem LOOM-Paradigma neue Programmierkonzepte mit relativ geringem Aufwand in eine statische Programmiersprache implementiert werden können. Das wird am Beispiel der *Kontextorientierten Programmierung* und des *Design-by-Contract* demonstriert.

5.1 Kontextorientiertes Programmieren mit Context#

Mit der Vision des allgegenwärtigen Rechners (*Ubiquitous Computers*) beschreibt [Weiser 1991] eine Welt, in der Rechner nicht mehr als Gegenstand an sich im Mittelpunkt stehen, sondern durch ihre Funktionalität geprägt werden. Eine Umsetzung dieser Vision kann heutzutage in den verschiedensten Bereichen beobachtet werden. Ein Mobiltelefon dient heute nicht mehr nur zum Telefonieren, sondern stellt auch Anwendungen bereit, die vor zwanzig Jahren einen Bürorechner benötigten. So können heute auf einem Telefon problemlos Termine verwaltet, elektronische Post abgefragt und Dokumente bearbeitet werden.

Allerdings ergeben sich damit auch neue Anforderungen an die Anwendung selbst. So müssen in Abhängigkeit vom Umfeld, in dem die Anwendung ausgeführt wird, verschiedene Variationsmöglichkeiten vorgesehen werden. Auf einem kleinen Bildschirm mit geringer Auflösung können dem Benutzer weniger Informationen präsentiert werden als auf einem Büro-Arbeitsplatz. Außerdem ist auf einem mobilen Gerät der Zugriff auf eine zentrale Serverdatenbank nur eingeschränkt möglich.

Gängige Programmiersprachen bieten derzeit gar keine oder nur rudimentäre Unterstützung, um kontextabhängiges Verhalten zu implementieren. Eine elegante Lösung für dieses Problem ist das von [Costanza und Hirschfeld 2005; Hirschfeld u. a. 2008; von Löwis u. a. 2007] vorgeschlagene Konzept der *kontextorientierten Programmierung*. In den folgenden Abschnitten soll gezeigt werden, dass dieses Konzept auf beliebige Programmiersprachen der *Common Language Infrastructure* (CLI) unter Verwendung von LOOM angewendet werden kann.

5.1.1 Kontextorientiertes Programmieren

Die grundlegende Idee der *Kontextorientierten Programmierung* (COP) besteht darin, den Programmfluss einer Anwendung durch den aktuellen *Kontext* beeinflussen zu können. Der Kontext spiegelt hierbei den Zustand der externen Welt wider. Das können zum Beispiel die zur Verfügung stehende Peripherie oder der Typ des Gerätes sein, auf dem die Software ausgeführt wird. Im Ergebnis wird dadurch eine Adaption des Anwendungsverhaltens in

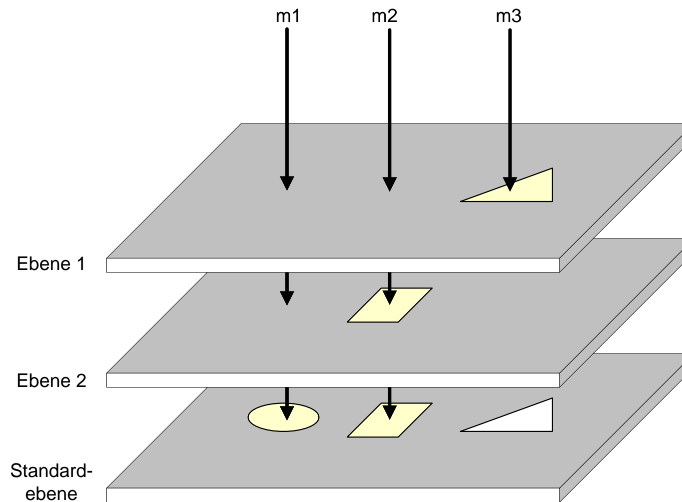


Abbildung 5.1: Methodenaufrufe durch COP-Ebenen

Abhängigkeit von seiner Umgebung erreicht. [Hirschfeld u. a. 2008] führen hierzu folgende Konzepte ein:

1. **Verhaltensvariationen**. Hierzu zählt das Hinzufügen, Modifizieren oder Entfernen von Verhalten für einen bestimmten Kontext.

2. **Ebenen**: In so genannten Ebenen *Layers* sollen die kontextabhängigen Verhaltensvariationen zusammengefasst (gruppiert) werden.

3. **Aktivierung**: Das in den Ebenen zusammengefasste kontextabhängige Verhalten kann zur Laufzeit aktiviert und deaktiviert werden. Das kann explizit im Quelltext geschehen. Letztendlich bedeutet das eine Aktivierung bzw. Deaktivierung von Ebenen.

4. **Kontext**: Der Kontext umfasst potentiell alle Informationen, auf die zur Laufzeit zugegriffen werden kann. Daraus leitet sich ab, welche Verhaltensvariation gewählt wird.

5. **Gültigkeitsbereich**: Der Bereich, in dem Ebenen aktiviert und deaktiviert werden, kann explizit kontrolliert werden. In unterschiedlichen Bereichen (Anweisungsblöcken) einer laufenden Anwendung können dieselben Variationen aktiv oder inaktiv sein.

Die technische Realisierung geschieht in COP durch eine Erweiterung des Auswahlverfahrens bei einem Methodenaufruf. Bei prozeduralen Sprachen ist die Auswahl (Dispatch) der Methoden ausschließlich von der Methodensignatur, dem Methodennamen und den Parametertypen abhängig. Objektorientierte Konstrukte führen eine weitere Dimension - die des Empfängers des Methodenaufrufs - ein. Bei der kontextorientierten Programmierung wird eine dritte Dimension, der Kontext, zur Auswahl der Methode herangezogen.

Beispielhaft seien in Abbildung 5.1 zwei aktive Ebenen dargestellt (`Ebene 1` und `Ebene 2`) sowie eine Ebene, die die Standardimplementierung enthält (`Standardebene`). `Ebene 1` wurde innerhalb des Gültigkeitsbereiches von `Ebene 2` aktiviert und liegt demnach über dieser. Wird eine Methode aufgerufen, so werden sukzessive die aktiven Ebenen von oben nach unten nach Verhaltensmodifikationen für diese Methode durchsucht. Existiert eine solche Modifikation, wird sie ausgeführt, andernfalls wird in der nächsten Ebene weiter gesucht. Ist keine Ebene aktiv oder enthält keine eine Verhaltensvariation, wird die Implementation der Standardebene verwendet. In Abbildung 5.1 gibt es für den Methodenaufruf `m1` in keiner Ebene eine Variation, daher wird dieser auch von der `Standardebene` behandelt. Demgegenüber erfolgt der Methodenaufruf `m3` auf der Verhaltensvariation in `Ebene 1`.

Eine Verhaltensvariation muss nicht notwendigerweise eine komplette Implementation der aufgerufenen Methode enthalten. Es ist auch möglich, den Kontrollfluss aus einer solchen Variation heraus an die nächste Ebene weiterzuleiten. [Hirschfeld u. a. 2008] führen hierzu das von AOP bekannte **Proceed** ein. Trifft ein Abwickler in einer Verhaltensvariation auf ein **Proceed()**, so wird die nächste Verhaltensvariation für die Methode in den folgenden aktiven Ebenen gesucht und schließlich ausgeführt. In Abbildung 5.1 wird beispielsweise für die Methode `m2` erst die Variation auf `Ebene 2` und dann die Implementation in der `Standardebene` abgewickelt.

[Costanza u. a. 2006] schlägt alternativ vor, Verhaltensvariationen, wie AOP-*Advices*, zu beschreiben. Das heißt, es lässt sich zusätzlich spezifizieren, ob eine Variation *vor*, *nach* oder *umschließend* zu folgenden Variationen oder der Standardimplementation ausgeführt wird. Das ist allerdings nur eine syntaktische Variante des *Proceed*-Ansatzes, da durch Letztgenanntes alle drei Fälle abgebildet werden können.

Bisher existieren verschiedene Umsetzungen des Konzepts: *ContextL* ist eine Umsetzung der kontextorientierten Programmierung für die Programmiersprache Lisp. [Costanza u. a. 2006] definiert neue Makros und Metaklassen, um die Konzepte von COP umzusetzen. Lisp ist als Programmiersprache bereits so konzipiert worden, dass eine Erweiterung der Sprachsyntax problemlos möglich ist. Somit integriert sich die ContextL-Spracherweiterung auch nahtlos als Bibliothek in das System, ohne dass eine neuer Kompiler oder eine eigene Laufzeitumgebung erforderlich sind.

*ContextJ** ist eine prototypische Umsetzung von COP für Java. [Hirschfeld u. a. 2008, S. 146 ff.] verwenden hier Standardmechanismen der Java-Programmierspache, wie generische und anonyme Klassen, um die COP-Konzepte umzusetzen. Obwohl diese Lösung mit sehr wenig Aufwand recht effektiv ist, sind die verwendeten Hilfskonstrukte weniger intuitiv. Verhaltensmodifikationen werden in anonymen Klassen definiert. Anweisungsblöcke, für die Ebenen aktiviert oder deaktiviert werden, müssen in solche Klassen ausgelagert werden.

Schließlich ist *ContextS* [Hirschfeld u. a. 2006] eine Variante von COP für Squeak/Smalltalk. Auch hier lassen sich Ebenen als Objekte erster Ordnung implementieren, sodass eine syntaktische Erweiterung der Sprache nicht notwendig ist.

5.1.2 Eine Umsetzung von COP mit Loom.Net

Das Konzept des kontextorientierten Programmierens lässt sich auf verschiedene Weise umsetzen. Die naheliegendste Variante ist sicherlich die Erweiterung einer bestehenden Programmiersprache. [Hirschfeld u. a. 2008] stellen beispielsweise eine bisher noch fiktive Erweiterung der Sprache Java vor. Neu sind hier die Schlüsselwörter `layer` zur Definition und Modularisierung von Ebenen, `with` und `without` zum Aktivieren und Deaktivieren einer Ebene sowie `Proceed` zum Weiterreichen des Kontrollflusses an die nächste zuständige Ebene.

Die Erweiterung einer bestehenden Programmiersprache erweist sich jedoch oft als aufwendig, da ein eigener Kompiler oder zumindest ein Präkompiler entwickelt werden muss. Ein Präkompiler könnte die COP-Sprachkonstrukte in Konstrukte der Basisprache transformieren, während alternativ ein Kompiler die Basissprache und die COP-Erweiterungen übersetzen müsste. Steht kein quelloffenener Kompiler für die Basissprache zur Verfügung, der entsprechend angepasst werden kann, sind die Entwicklungskosten erwartungsgemäß hoch. Zudem besteht dadurch eine starke Abhängigkeit zum Entwicklungszyklus der Basissprache. Entwickelt diese sich weiter, muss auch der eigene Kompiler um die neu hinzugekommenen Konstrukte erweitert werden.

Eine alternativer Vorschlag ist es, die COP-Konzepte als Framework zu realisieren. Ein Framework ist eine Bibliothek, deren Verwendung im Wesentlichen durch definierte Entwurfsmuster vorgegeben ist. Gegenüber der Kompilervariante ist die einfache Implementierbarkeit ein großer Vorteil. *Context#* ist ein solches Framework für die .NET-Laufzeitumgebung und

```
1 // Definition einer neuen Ebene
2 public class Layer1:Layer {}
3
4 [Layered]
5 public class A
6 {
7     // Definition der Standardebene
8     public void foo(int i)
9     {
10       ...
11    }
12
13    // Verhaltensvariation für "Layer1"
14    public void foo(Layer1 layer, int i)
15    {
16      ...
17      layer.Proceed();
18    }
19 }
```

Listing 5.1: Modularisierung auf Klassenebene

benötigt unter Verwendung von LOOM.NET weniger als 200 Zeilen Quelltext, um alle COP-Konzepte zu implementieren. Die Erfahrungen des Autors in großen Industrieprojekten haben auch gezeigt, dass die Barrieren für die Benutzung eines Frameworks bei Entscheidern viel geringer sind als der Einsatz eines neuen, fremden Kompilers.

Das vom Autor entwickelte Framework *Context#* besteht aus fünf Klassen und einem Aspekt. Deren Verwendung erfolgt nach vordefinierten Mustern, die im Folgenden erläutert werden sollen.

Verhaltensvariation als Überladungen

Um COP-Verhaltensvariationen zu beschreiben und Ebenen zu definieren, zu denen die Verhaltensvariationen gehören, existiert in *Context#* die Basisklasse `Layer`.

Regel 5.1 (Ebenendefinition) *Eine Ebene E ist eine Klasse, die von der Basisklasse* `Layer` *abgeleitet wurde.*

Regel 5.2 (Verhaltensvariation) *Eine Methode m' ist eine Verhaltensvariation von m in der Ebene E, gdw. m' eine Überladung von m mit zusätzlichem Parameter von Typ E als ersten Parameter und die Klasse in der m' definiert wurde, wird als* `[Layered]` *ausgewiesen.*

Listing 5.1 zeigt eine Verhaltensvariation der Methode `foo` für die Ebene `Layer1`. In diesem Beispiel findet eine *Modularisierung auf Klassenebene* statt. Das bedeutet, dass alle Verhaltensvariationen auch innerhalb der betreffenden Klasse definiert sind. Die kontextorientierte Programmierung sieht alternativ auch eine *Modularisierung pro Ebene* vor. In der Programmiersprache C# lässt sich das, wie Listing 5.2 zeigt, einfach durch das Konzept der *partiellen* Klassen umsetzen. Die Definition der Klasse `A` wird hierbei in mehrere Teile zerlegt. Diese Teile enthalten dann jeweils die für die entsprechende Ebene relevanten Definitionen.

Wie bereits oben definiert, bekommt eine Verhaltensvariation einer Methode als ersten Parameter ein Exemplar vom Typ `Layer`. Neben der Zuordnung der Methoden zu einer bestimmten Ebene durch die Parametersignatur wird durch dieses Exemplar auch die Weiterleitung des Kontrollflusses an die untergeordneten Ebenen gesteuert. Hierzu dient die in der Basisklasse `Layer` definierte Methode `Proceed`. Wie in Listing 5.2 dargestellt, wird die Methode direkt auf dem übergebenen Exemplar der Ebene aufgerufen.

Als Erweiterung des ursprünglichen Ansatzes von [Costanza und Hirschfeld 2005] können Ebenen in *Context#* spezialisiert werden. Abbildung 5.2 zeigt ein Beispiel für solche Vererbungsrelation. Vererbungsrelationen über Ebenen bieten den Vorteil, die Kontextinformationen viel feingranularer abbilden zu können. Insbesondere wenn Kontextinformationen nicht

```
1 /* Definition der Standardebene */
2 public partial class A
3 {
4     public void foo(int i)
5     {
6       ...
7     }
8 }
9
10
11 /* Definition der Ebene "Layer1" */
12 public class Layer1:Layer {}
13
14 public partial class A
15 {
16    public void foo(Layer1 layer, int i)
17     {
18       ...
19       layer.Proceed();
20     }
21 }
```

Listing 5.2: Modularisierung pro Ebene

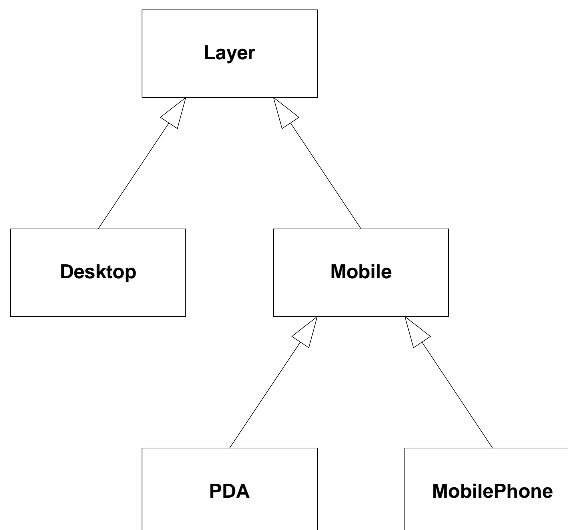


Abbildung 5.2: Polymorphie von Ebenen

orthogonal zueinander stehen, sondern voneinander abhängen, kann dadurch der Quelltext effizienter gestaltet werden. Im oben aufgeführten Beispiel steckt die Information „mobiles Gerät“ indirekt in der Information „Mobiltelefon“ und „PDA“ (*Personal Digital Assistant*, Organizer). Ohne Ebenenhierarchien müsste diese Information aber explizit durch die gleichzeitige Aktivierung mehrerer Ebenen ausgedrückt werden. Im Beispiel wären es `Mobile` und `MobilePhone` oder `PDA`. Alternativ hätte im Beispiel die allgemeine Verhaltensvariation für die Ebenen `MobilePhone` und `PDA` natürlich auch redundant definiert werden können.

In *Context#* können Ebenen als Container für kontextspezifische Informationen verwendet werden. Hierzu werden in der Ebenenklasse entsprechende Attribute und Zugriffsmethoden definiert. Alle Verhaltensvariationen dieser Ebenen können gleichberechtigt auf diese Informationen zugreifen. Die Lebensdauer des Containers ist dabei der *Gültigkeitsbereich* der Ebene.

Aktivierung und Deaktivierung von Ebenen

Die Aktivierung bzw. Deaktivierung von Ebenen erfolgt explizit im Quelltext und zwar immer für einen bestimmten Bereich. In der COP-Implementation für Java wird das beispielsweise

```
1 using (With<Layer4>.Do)
2 {
3   // Ab hier ist "Layer4" aktiv
4   using (Without<Layer4>.Do)
5   {
6     // In diesem Block ist "Layer4" wieder deaktiviert
7   }
8 }
```

Listing 5.3: Aktivierung und Deaktivierung von Ebenen

durch die Einführung der Schlüsselwörter **with** und **without** realisiert. Das Schlüsselwort **with** aktiviert eine Ebene für den folgenden Anweisungsblock. Das Schlüsselwort **without** bewirkt genau das Gegenteil: Solange sich der Kontrollfluss innerhalb des Anweisungsblocks befindet, ist die entsprechende Ebene deaktiviert. Es ist allerdings möglich, die Ebene in einer weiteren Verschachtelung mit dem Schlüsselwort **with** wieder zu aktivieren. Außerdem muss eine in **without** benannte Ebene beim Eintritt in den Anweisungsblock nicht notwendigerweise auch aktiv gewesen sein. In diesem Fall wird die Deaktivierung ignoriert.

In *Context#* wird dieses Verhalten durch eine Hilfskonstruktion über das Schlüsselwort `using` und die Schnittstelle `IDisposable` [Microsoft Corporation 2005a, §20.8.8] erreicht:

Regel 5.3 (Ebenenaktivierung) *Die Ebene E wird durch das Konstrukt* `using(With<E>.Do)` *im nachfolgenden Anweisungsblock aktiviert.*

Regel 5.4 (Ebenendeaktivierung) *Die aktive Ebene E wird durch das Konstrukt* `using (WithOut<E>.Do)` *während der Abwicklung des nachfolgenden Anweisungsblocks deaktiviert. War die Ebene E nicht aktiv, so wird die Anweisung ignoriert.*

Listing 5.3 zeigt die Aktivierung/Deaktivierung von Ebenen noch einmal im Context# Syntax. Die statischen Eigenschaften `Do` der generischen Klassen `With` und `Without` liefern jeweils die Hilfsobjekte, die die im generischen Parameter übergebene Ebene aktivieren bzw. deaktivieren. Damit der Mechanismus funktioniert, muss der Aufruf in der **using** Klausel erfolgen.

Mit der Ebenendefinition, den Verhaltensvariationen sowie der Aktivierung und der Deaktivierung der Ebenen sind die Konzepte von COP in *Context#* vollständig umgesetzt. Im nachfolgenden Abschnitt soll die Verwendung an einem Beispiel erläutert und Context# mit dem ContextJ-Voschlag verglichen werden.

5.1.3 Vergleich von Context# mit ContextJ

Im direkten Vergleich zwischen dem Framework *Context#* und der Spracherweiterung *ContextJ* (Abbildung 5.3) ist festzustellen, dass die Unterschiede gering und ausschließlich syntaktischer Natur sind. Da *Context#* keine neuen Schlüsselwörter definiert, werden Hilfskonstrukte, wie vordefinierte Basisklassen, Methodenüberladungen oder die automatische Freigabe von Objekten, nach Verlassen eines Anweisungsblocks verwendet. Diese Konstrukte benötigen gegenüber *ContextJ* nur unwesentlich mehr Schreibaufwand.

In Listing 5.4 ist das Beispiel aus der ursprünglichen Lösung von [Hirschfeld u. a. 2008] als Context#-Implementation dargestellt. In diesem Beispiel wird eine Klasse `Person` (Zeile 11-39) und eine Klasse `Employer` definiert (Zeile 42-62), die eine Angestellten-Arbeitgeber-Relation repräsentieren. Jede `Person` hat in diesem Beispiel genau einen `Employer`. Neben dem Namen werden in den Klassen auch Details zur Adresse gespeichert. Zusätzlich werden zwei Ebenen definiert, eine Adressebene (`Address`, Zeile 7) und eine Ebene für das Angestelltenverhältnis (`Employment`, Zeile 9).

Exemplare von `Person` und `Employer` können ihren Inhalt über die entsprechende Methode `ToString` wiedergeben. Die Standardimplementation sieht vor, dass ausschließlich der

Beschreibung	**Context#**	**ContextJ**

Definition von Verhaltensvariationen	in einer als `[Layered]` definierten Klasse. Eine Variation ist eine Überladung einer Methode, deren erster Parameter eine Ableitung vom Typ `Layer` ist.
	`void m(Ebene1 e1, ...){...}`
		mit Methoden, die innerhalb eines durch das Schlüsselwort `layer` abgegrenzten Bereiches liegen.
		`layer Ebene1 {`
		`void m(...){...} }`

Aktivierung und Deaktivierung von Ebenen	durch die Konstruktion von Hilfsklassen `With` und `WithOut` innerhalb eines `using`-Blockes. `using(With<Ebene1>.Do){ ... }`	durch Verwendung Schlüsselwörter `with` und `without`. `with Ebene1 { ... }`
Delegation des Kontrollflusses an die nächste Implementierung	durch die `Proceed()`-Methode des Ebenenobjekts. `e1.Proceed()`	durch das Schlüsselwort `proceed`. `proceed`

Abbildung 5.3: Vergleich Context# mit ContextJ

im Objekt hinterlegte Name ausgegeben wird. Zusätzlich implementieren die Klassen jedoch auch jeweils eine Verhaltensvariation der Methode für die Adressebene, um zum Namen auch die vollständige Adresse auszugeben. Die Klasse `Person` implementiert noch eine weitere Verhaltensvariation, um außerdem den Arbeitgeber anzuzeigen.

In der Methode `Main` wird deutlich, wie sich die Aktivierung der Ebenen auf die Ausgabe auswirken (dargestellt jeweils als Kommentar der entsprechenden Quelltextzeile). Insgesamt gibt es in der Methode vier durch geschweifte Klammern geschachtelte Anweisungsblöcke, in denen jeweils unterschiedliche Ebenen aktiv sind:

1. keine Ebene ist aktiv (Zeile 73)
2. Ebene `Address` ist aktiv (Zeile 75)
3. Ebene `Employment` und `Address` sind aktiv (Zeile 77)
4. Ebene `Employment` ist aktiv (Zeile 82)

5.1.4 Implementationsdetails

Die Implementation von *Context#* mit knapp 200 Zeilen Quelltext ist übersichtlich und einfach gehalten. Im Wesentlichen besteht sie aus:

- einer Klasse (`Layer`) zur Definition von Ebenen,
- zwei generischen Klassen (`With` und `WithOut`) zur Aktivierung und Deaktivierung der Ebenen,
- einem Aspekt (`Layered`), der die Zuordnung zur richtigen Verhaltensvariation realisiert.

Wie bereits erwähnt, werden Ebenen als Klasse definiert, deren Basisklasse die Klasse `Layer` ist. Die beiden generischen Hilfsklassen `With` und `WithOut` akzeptieren eine solche Ebenenklasse als generischen Parameter und stellen über die statische Eigenschaft `Do` ein Objekt ihrer Selbst zur Verfügung, das die Aktivierung und Deaktivierung eben dieser Ebenen vornimmt. Beide Klassen implementieren das *Dispose*-Muster [Microsoft Corporation 2005a,

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4   using Loom;
5   using Loom.ContextSharp;
6
7   public class Address : Layer { }
8
9   public class Employment : Layer { }
10
11  [Layered]
12  public abstract class Person
13  {
14      private string name;
15      private string address;
16      private Employer employer;
17
18      public Person(string name, string
            address, Employer employer)
19      {
20          this.name = name;
21          this.address = address;
22          this.employer = employer;
23      }
24
25      public override string ToString()
26      {
27          return "Name: " + this.name;
28      }
29
30      public string ToString(Address
            layer)
31      {
32          return layer.Proceed() + "; 
                Address: " + this.address;
33      }
34
35      public string ToString(Employment
            layer)
36      {
37          return layer.Proceed() + "; 
                Employer: " + this.
                employer;
38      }
39  }
40
41  [Layered]
42  public abstract class Employer
43  {
44      private string name;
45      private string address;
46
47      public Employer(string name,
            string address)
48      {
49          this.name = name;
50          this.address = address;
51      }
52
53      public override string ToString()
54      {
55          return "Name: " + this.name;
56      }
57
58      public string ToString(Address
            layer)
59      {
60          return layer.Proceed() + "; 
                Address: " + this.address;
61      }
62  }
63  namespace ContextSharpExample
64  {
65    class Program
66    {
67      static void Main(string[] args)
68      {
69        Employer employer = Weaver.
              Create<Employer>("HPI","
              Potsdam");
70        Person person = Weaver.Create<
              Person>("Wolfgang Schult", "
              Berlin", employer);
71
72        // Name: Wolfgang Schult
73        Console.WriteLine(person);
74
75        using (With<Address>.Do)
76        {
77          using (With<Employment>.Do)
78          {
79            // Name: Wolfgang Schult;
80            // Address: Berlin;
81            // Employer: Name: HPI;
82            // Address: Potsdam
83            Console.WriteLine(person);
84
85            using (Without<Address>.Do)
86            {
87
88              // Name: Wolfgang Schult;
89              // Employer: Name: HPI
90              Console.WriteLine(person);
91            }
92
93            // Name: Wolfgang Schult;
94            // Address: Berlin;
95            // Employer: Name: HPI;
96            // Address: Potsdam
97            Console.WriteLine(person);
98          }
99
100         // Name: Wolfgang Schult;
101         // Address: Berlin
102         Console.WriteLine(person);
103       }
104
105       // Name: Wolfgang Schult
106       Console.WriteLine(person);
107     }
108   }
109 }
```

Listing 5.4: Ein Context#-Beispiel

```
1 public class With<LAYER> : IDisposable where LAYER : Layer, new(){
2     protected LayerPtr predecessor;
3     protected LAYER layer;
4
5     public static IDisposable Do {
6         get { return new With<LAYER>(); }
7     }
8
9     private With() {
10      // Ebene LAYER aktivieren
11      ...
12    }
13
14    void IDisposable.Dispose() {
15      // Ebene LAYER entfernen
16      ...
17    }
18 }
```

Listing 5.5: Ausschnitt der `With`-Klasse

§20.8.8] und das zugehörige `IDisposable` Interface. Wird auf die `Do`-Eigenschaft innerhalb einer `using`-Klausel zugegriffen, wird nach dem Verlassen des zugehörigen Anweisungsblocks automatisch die `Dispose`-Methode des gelieferten Objekts aufgerufen. Somit kann im Konstruktor der `With`-Klasse die Aktivierung einer Ebene implementiert werden, während in der `IDisposable.Dispose` für die Deaktivierung derselben Ebene sorgt.

Listing 5.5 zeigt einen Auschnitt der `With`-Klasse. Die `WithOut`-Klasse implementiert Konstruktor und `Dispose` mit getauschten Rollen.

Intern werden die aktiven Ebenen als Objekte in einer verketteten Liste verwaltet. Dabei ist diejenige Ebene, die zuletzt aktiviert wurde, auch diejenige, die in dieser Liste an erster Position steht. Zusätzlich zeigen die `With` und `WithOut` Objekte auf die von ihnen aktivierten Ebenen. Abbildung 5.4 zeigt in der ersten Grafik die Liste mit einer aktiven Ebene (`Layer4`). Das `Root`-Objekt ist die Wurzel der Liste. In der zweiten Grafik wurden zusätzlich die Ebenen `Layer3` und `Layer1` aktiviert.

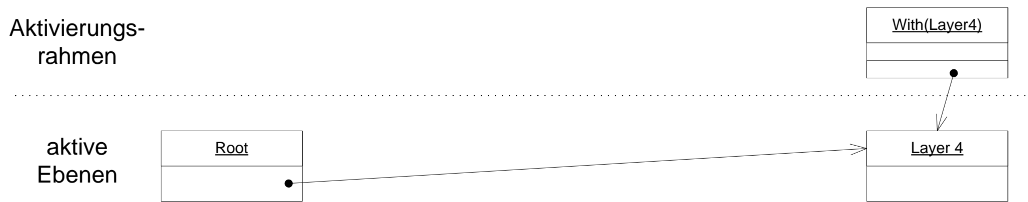
Beim Aufruf einer Methode m wird - ausgehend von der Wurzel für jede Ebene - in der Liste nach einer Verhaltensvariation m' von m gesucht, die als Argumente die gewählte Ebene zuzüglich der Argumente von m als Eingabe akzeptiert. Enthält die Liste - ausgehend von `Root` - keine passende Ebene, wird die Methode m (die Implementation auf der Standardebene) gewählt. Die gefundene Methode wird mit den entsprechenden Parametern ausgeführt.

Ruft eine Verhaltensvariation wiederum ein `Proceed` auf dem ihr übergebenen Ebenenobjekt auf, so wird die Suche nach einer weiteren Verhaltensvariation von m fortgesetzt. Dabei werden nur noch diejenigen aktiven Ebenen betrachtet, die bei der vorangegangenen Suche noch nicht berücksichtigt wurden. Das sind genau diejenigen Ebenenobjekte, die in der Liste hinter dem übergebenen Ebenenenobjekt angehängt sind. Wird eine Variation gefunden, so wird diese aufgerufen, ansonsten wird die Implementation der Standardebene ausgeführt.

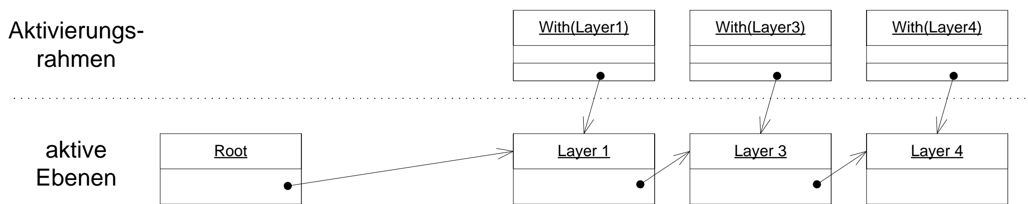
Im Falle der Aktivierung einer bereits aktiven Ebene muss der Algorithmus dafür sorgen, dass kein weiteres Exemplar dieser Ebene in die Liste aufgenommen wird. Das würde zu einem reentranten Aufruf der für diese Ebene verantwortlichen Verhaltensvariation durch das Abarbeiten von `Proceed` führen. Um das zu verhindern, wird die bereits aktive Ebene temporär an den Anfang der Liste verschoben. Das Hilfsobjekt für die Aktivierung (Listing 5.5) merkt sich hierzu die ursprüngliche Position des Ebenenobjekts im Attribut `predecessor`. Die dritte Grafik in Abbildung 5.4 illustriert diese Situation. Beim Verlassen des zur Aktivierung gehörenden Anweisungsblocks wird dieser Vorgang dann wieder rückgängig gemacht.

Die Deaktivierung von Ebenen verläuft vom Mechanismus her ähnlich und ist in der letzten Grafik in Abbildung 5.4 dargestellt. Das Hilfsobjekt für die Deaktivierung (`WithOut`)

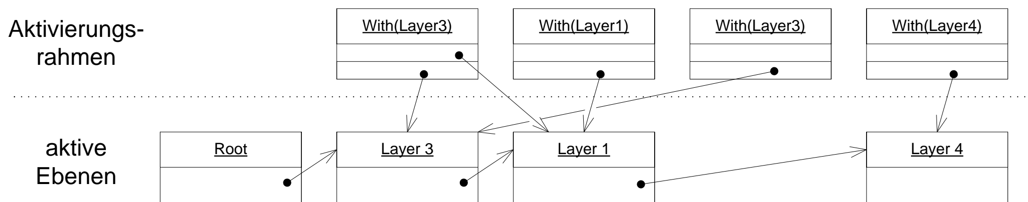
Aktivierung einer Ebene



Hinzufügen weiterer Ebenen



Wiederholte Aktivierung einer Ebene



Deaktivierung einer Ebene

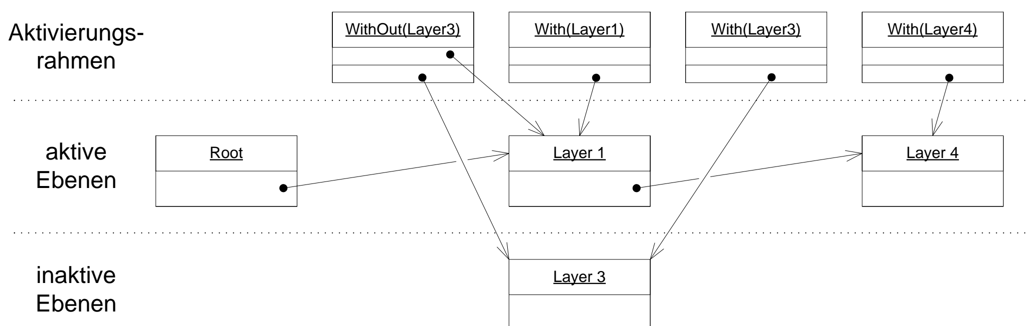


Abbildung 5.4: Aktivierung und Deaktivierung von Ebenen

merkt sich die alte Position des Ebenenobjekts und entfernt es temporär aus der Liste. Auch hier wird nach dem Verlassen des Anweisungsblocks das Ebenenobjekt wieder an der ursprünglichen Position eingefügt.

Die Aktivierung und Deaktivierung von Ebenen ist immer Thread-lokal, da diese jeweils für die entsprechenden Anweisungsblöcke gültig sind. Daher ist der Wurzelzeiger für die Liste der aktiven Ebenen ebenso Thread-lokal.

Um die Verhaltensvariationen in Abhängigkeit von den aktiven Ebenen überhaupt ausführen zu können, wird ein weiterer Mechanismus benötigt. Dieser muss die Ausführung aller Methoden auf einer als `Layered` definierten Klasse durch den oben beschriebenen Algorithmus ersetzen. Realisiert wird das durch den gleichnamigen Aspekt in Listing 5.6. Klassen, in denen Verhaltensvariationen definiert sind, müssen mit dem Aspekt annotiert werden.

Kern des Aspektes ist die in Listing 5.6 dargestellte **around**-Advice `ContextSwitch` (Zeile 17-21). Diese ersetzt jede einzelne Methode der verwobenen Klasse. Statt der Implementation der gerufenen Methoden wird der bereits erläuterte Algorithmus zur Ermittlung der jeweils für den entsprechenden Methodenaufruf gültigen Verhaltensvariation ausgeführt (Zeile 23-52). Hierbei wird das LOOM.NET-Konzept der Nachrichtenmanipulation (Abschnitt 2.8.3) verwendet. Im Advice ist die aufgerufene Methode in Form des LOOM.NET-Aufrufkontextes (`ctx`) bekannt. Durch einen `ReCall`-Aufruf auf diesem Aufrufkontext (Zeile 42) kann eine erneute Zuordnung zu einer Methode auf der Zielklasse erzwungen werden. Ändert man die Argumente so, dass der erste Parameter eine aktive Ebene enthält, landet der Aufruf automatisch in der entsprechenden Verhaltensvariation. Existiert diese nicht, wird das durch eine `ContextInvocationException` signalisiert, und der Algorithmus versucht mit der nächsten aktiven Ebene fortzufahren (Zeile 44 und 50). Bleibt auch das ohne Erfolg, wird schließlich die Standardimplementation mit `ctx.Invoke` aufgerufen (Zeile 33).

Vor der Ausführung einer Verhaltensvariation wird der LOOM.NET-Aufrufkontext in einem ebenenlokalen Stapel (`contextstack`) abgelegt (Zeile 36). Ruft der Benutzer in einer Verhaltensvariation `Proceed` auf, zeigt das oberste Stapelelement genau auf diejenige Methode der Standardebene, die aktuell abgewickelt werden soll. Die Implementation von `Proceed` kann damit die nächste Verhaltensvariation für diese Methode über die `InvokeMethod`-Implementation des Aspekts aufrufen (Listing 5.7). Die Verwendung eines Stapels begründet sich in der Annahme, dass der Benutzer Methoden mit einer Verhaltensvariation auch geschachtelt aufrufen kann. Der Stapel stellt sicher, dass `Proceed` immer die richtige Methode findet.

5.1.5 Diskussion der Lösung

Die Implementation von Context# zeigt, wie mit relativ geringem Aufwand unter Zuhilfenahme der LOOM.NET-Konzepte eine neues Programmierparadigma in eine bestehende Sprache eingebunden werden kann. Die Lösung ist im Umfang übersichtlich und lässt sich einfach als Bibliothek einbinden. Was als neue Schlüsselwörter in die Sprachdefinition hätte aufgenommen werden müssen, wurde über Klassen und Aspekte realisiert. Ein spezieller Kompiler ist somit nicht nötig.

Die Lösung steht potentiell für alle CLI-Sprachen zur Verfügung. Als Einschränkung muss hier jedoch genannt werden, dass das Konzept der **using**-Blöcke nicht in jeder CLI-Programmiersprache unterstützt wird. Daher ist die Aktivierung und Deaktivierung von Ebenen in solchen Sprachen nicht implizit möglich, wie es hier über Anwendungsblöcke in C# gezeigt wurde, sondern muss explizit erfolgen.

5.2 Design by Contract mit LOOM.Net

Die Softwareentwicklung hat den Anspruch, dass sich Komponenten konsistent entsprechend ihrer Spezifikation verhalten. Bei großen komplexen Softwaresystemen ist es wichtig, dass

```
1  public class LayerPtr
2  {
3      internal LayerPtr()
4      {
5      }
6
7      internal Layer nextlayer;
8  }
9
10 public class Layered : AspectAttribute
11 {
12     [ThreadStatic]
13     internal static LayerPtr currentlayer=new LayerPtr();
14
15     [IncludeAll]
16     [Call(Advice.Around)]
17     public T ContextSwitch<T>([JPContext] Context ctx, params object[] args)
18     {
19         ctx.Tag = args;
20         return InvokeMethod<T>(currentlayer.nextlayer, ctx);
21     }
22
23     internal static T InvokeMethod<T>(Layer layer, Context ctx)
24     {
25         object[] args = (object[])ctx.Tag;
26         object[] newargs = new object[args.Length + 1];
27         args.CopyTo(newargs, 1);
28
29         for (;;)
30         {
31             if (layer == null)
32             {
33                 return (T)ctx.Invoke(args);
34             }
35
36             layer.contextstack.Push(ctx);
37             try
38             {
39                 newargs[0] = layer;
40                 try
41                 {
42                     return (T)ctx.ReCall(newargs);
43                 }
44                 catch (ContextInvocationException) { }
45             }
46             finally
47             {
48                 layer.contextstack.Pop();
49             }
50             layer = layer.nextlayer;
51         }
52     }
53 }
```

Listing 5.6: Der Context#-Aspekt

```
1  public abstract class Layer: LayerPtr
2  {
3      internal Stack<Context> contextstack = new Stack<Context>();
4
5      protected Layer():base()
6      {
7      }
8
9      public object Proceed()
10     {
11         Context context = contextstack.Peek();
12         return ContextManaged.InvokeMethod<object>(nextlayer, context);
13     }
14 }
```

Listing 5.7: Die Implementierung der Ebenenklasse

jedes einzelne Teilsystem das Verhalten zeigt, das von ihm erwartet wird. Ist das Verhalten nicht mehr vorhersagbar, kann eine korrekte Funktionsweise des Gesamtsystems nicht mehr garantiert werden. So muß beim Einsatz von Komponenten Dritter oder der Wiederverwendung eigener Komponenten die Vorhersagbarkeit des Verhaltens innerhalb der Spezifikation gewährleistet sein.

Softwarekomponenten werden allgemein durch die Schnittstellen, die sie benötigen und anbieten, spezifiziert [Szyperski 2002]. Daher ist der erste Schritt, um die Vorhersagbarkeit der Komponente sicherzustellen, die Schnittstellen so präzise wie möglich zu definieren. Die meisten objektorientierten Sprachen, wie Java und C# lassen jedoch nur eine eingeschränkte Beschreibung der Schnittstellen zu. So ist es zum Beispiel oft nicht möglich, zusätzliche Bedingungen für die korrekte Benutzung einer Komponente zu formulieren. Ebenso unmöglich ist es, Annahmen, die die Komponente über ihre Ausführungsumgebung macht, zu treffen. Diese Angaben finden sich im besten Fall in den Kommentaren der öffentlichen Schnittstellenbeschreibung wieder. Demgegenüber bietet beispielsweise UML mit der *Object Constraint Language* (OCL) [Object Management Group 2001] schon eine solche Möglichkeit.

Design by Contract (DBC) ist ein Ansatz, bei dem solche Annahmen und Bedingungen für die korrekte Benutzung einer Komponente als Boolsche-Ausdrücke formuliert werden können. In den folgenden Abschnitten soll dargelegt werden, wie mit LOOM gängige Programmiersprachen mit dem DBC-Ansatz erweitert werden können.

Die in diesen Abschnitten vorgestellten Ergebnisse basieren auf Ideen, die vom Autor gemeinsam mit Kai Köhne entwickelt wurden [Köhne, Schult und Polze 2005]. Dennoch sind wesentliche Teile der Arbeit für diese Dissertation überarbeitet und weiterentwickelt worden. Das betrifft insbesondere die technische Umsetzung des Design-by-Contract-Konzeptes mit LOOM.NET. Vorgestellt wurde die erweiterte Version bereits in [Schult 2007].

5.2.1 Design by Contract

Entwickelt wurde das Konzept DBC ursprünglich für die Programmiersprache *Eiffel* [Meyer 1992]. Sie basiert auf der Grundidee, dass Objekte anderen Objekten Dienste anbieten. Dienste sind in diesem Kontext die Methoden, die ein Objekt zur Verfügung stellt. Die Bedingungen sowie die Effekte der Benutzung dieser Dienste werden als formaler Vertrag zwischen beiden Parteien, (also dem Rufer einer Methode und demjenigen, der die Methode implementiert), formuliert. Das Ergebnis ist eine zur klassischen objektorientierten Programmiersprache erweiterte Schnittstellenbeschreibung. Ein Beispiel für eine solche Schnittstellendefinition zeigt Listing 5.8.

Für jede Schnittstellendefinition, die von einem Objekt implementiert wird, werden zusätzliche Bedingungen (Annahmen) definiert. Die Schlüsselwörter **require** und **ensure** werden verwendet, um Vor- und Nachbedingungen zu markieren, **invariant** hingegen beschreibt

```
1  class interface DICTIONARY [ELEMENT]
2
3  feature
4  put (x: ELEMENT; key: STRING) is
5    -- Insert x so that it will be retrievable
6    -- through key.
7    require
8      not_full: count <= capacity
9      key_valid: not key.empty
10   ensure
11     item_avail: has (x)
12     item_stored: item (key) = x
13     size_incr: count = old count + 1
14
15   -- ... Interface specifications of other
16   -- features ...
17
18 invariant
19   not_under_minimum: 0 <= count
20   not_over_maximum: count <= capacity
21
22 end
```

Listing 5.8: Definition von Schnittstellenverträgen in Eiffel

die Invarianten. Alle drei Konzepte sind Annahmen in Form eines Booleschen-Ausdrucks, die den Vertrag widerspiegeln. Wenn ein Ausdruck als Ergebnis den Wahrheitswert *falsch* liefert, bedeutet dies, dass der Vertrag gebrochen wurde. Das führt dazu, dass der normale Kontrollfluss unterbrochen wird, z. B. durch das Werfen einer Exception. Wann der Ausdruck ausgewertet wird, ist abhängig vom verwendeten Schlüsselwort.

Vor- und Nachbedingungen gehören immer zu einer Methode der Schnittstelle. Vorbedingungen müssen durch den Aufrufer der Methode erfüllt werden. Sie werden vor dem Eintritt in die Methode geprüft. Nachbedingungen müssen erfüllt sein, wenn die Methode zum Aufrufer zurückkehrt. Der Aufrufer kann also davon ausgehen, dass alle in den Nachbedingungen formulierten Annahmen gültig sind, wenn der Kontrollfluss zu ihm zurückkommt. Umgekehrt wird die Methode niemals zur Ausführung kommen, wenn die Vorbedingungen nicht erfüllt sind.

Invarianten sind Annahmen, die während des gesamten Lebenszyklusses eines Objektes gelten müssen. Sie werden beim Eintritt und beim Verlassen jeder Methode der Schnittstelle überprüft. Allerdings ist es möglich, dass die Annahmen während der Ausführung einer Methode zeitweise verletzt werden.

Da Design-by-Contract eine Erweiterung des Schnittstellenkonzeptes objektorientierter Sprachen ist, muss es natürlich auch deren Regeln zur Vererbung folgen. Folglich werden sämtliche Annahmen eines Vertrages auch in den abgeleiteten Schnittstellen vererbt und müssen sowohl vom implementierenden als auch vom aufrufenden Objekt erfüllt werden.

Es ist weiterhin möglich, Vor- und Nachbedingungen zu erweitern. Vorbedingungen müssen dann allerdings mindestens gleich oder weniger restriktiv als im Basisinterface sein. Bei den Nachbedingungen kehrt sich diese Forderung um. Sie müssen mindestens genauso oder aber restriktiver als ihre Definition in der Basisschnittstelle sein. Der Grund hierfür ist, dass ein Objekt, das eine abgeleitete Schnittstelle implementiert, von einem anderen Objekt über die Basisschnittstelle benutzt werden kann. Der Aufrufer weiß in diesem Fall von einer restriktiveren Vorbedingung nichts, was somit zu einem (unverschuldeten) Vertragsbruch führen kann. Umgekehrt würde eine weniger restriktive Nachbedingung dazu führen, dass Annahmen des Aufrufers nicht mehr erfüllt sind, obwohl sie aus seiner Sicht Vertragsbestandteil waren.

5.2.2 Implementationsdetails

Es gibt viele verschiedene Möglichkeiten, DBC-Konzept in objektorientierte Sprachen zu integrieren. Beispiele hierfür sind Perl [Conway und Goebel 2001], Python [Plösch 1997], Ruby [Hunt 2002], Ada [Luckham u. a. 1987], Lisp [Hölzl], Smalltalk [Carrillo-Castellón u. a. 1996] and C++ [Guerreiro 2002]. Grundsätzlich muss jedoch folgendes Problem gelöst werden: Wie wird die Schnittstellenbeschreibung um die Annahmen erweitert und wie werden diese zur Laufzeit geprüft? Eine übliche Lösung ist eine Spracherweiterung mit entsprechend erweitertem Kompiler.

Mit LOOM.NET lässt sich eine elegante Alternative finden, die ausschließlich auf sprachinhärenten Mitteln basiert. Die Abbildung der DBC-Konzepte auf die CLI wird wie folgt definiert:

Regel 5.5 (Schnittstellenvertrag) *Ein DBC-Schnittstellenvertrag wird durch eine CLI-Klasse oder ein CLI-Interface abgebildet.*

Regel 5.6 (DBC-Methoden) *DBC-Methoden bilden sich auf .NET-Klassen- oder Interface-Methoden, einschließlich spezieller Methoden für Eigenschaften (Properties) und Ergeignisse (Events) ab. Diese Methoden müssen öffentlich sein.*

Regel 5.7 (Annahmen) *Vor-, Nachbedingungen und Invarianten werden durch die CLI-Attribute* `Requires`*,* `Ensures` *und* `Invariant` *definiert.*

Grundsätzlich gilt, dass die Invarianten Klassen und Interfaces annotieren dürfen. Das sind genau diejenigen Klassen und Interfaces für die sie gelten sollen. `Requires`- und `Ensures`-Annotationen hingegen können ausschließlich auf Methoden angewendet werden. Private Methoden dürfen nicht annotiert werden, eine Prüfung von Invarianten beim Aufruf bzw. dem Verlassen privater Methoden findet nicht statt.

Die Bedingungen werden in Form eines Prädikats formuliert und als Zeichenkettenparameter bei der Annotation übergeben. Innerhalb des Prädikats darf auf öffentliche Variablen und Eigenschaftswerte einer Klasse zugegriffen werden. Für Ausdrücke der Vor- und Nachbedingungen gilt zusätzlich, dass diese auch Parameter der annotierten Methode enthalten kann. In Nachbedingungen kann über das Schlüsselwort `RESULT` auf den Rückgabewert der Methode zugegriffen werden.

Ausdrücke von Nachbedingungen können auf Werte zugreifen, die *vor* dem Aufruf der annotierten Methode gültig waren. Dies können sowohl Methodenparameter, Klassenvariablen oder Eigenschaftswerte sein. Innerhalb des Ausdrucks muss dem Bezeichner hierzu das Schüsselwort `OLD` vorangestellt werden.

Listing 5.9 zeigt eine einfache Klasse mit einer Vor- und Nachbedingung für die drei Methoden `Einzahlung`, `Auszahlung` und `Transferiere` sowie eine Invariante für die gesamte Klasse `Sparkonto`. Ein Sparkonto muss immer einen positiven Saldo aufweisen (Zeile 1). Außerdem wird gefordert, dass nur positive Werte eingezahlt (Zeile 11) und Auszahlungen den Saldo nicht überschreiten dürfen (Zeile 18). Die Nachbedingungen entsprechen der vom Dienstnutzer erwarteten Semantik der jeweiligen Methode: Beispielsweise muss eine Einzahlung den aktuellen Kontostand genau um den übergebenen Wert erhöhen (Zeile 12).

Werden die Annahmen an einer Klasse formuliert, ist der Dienstanbieter damit bereits festgelegt, denn die Klasse selbst enthält auch die Implementation der Schnittstelle. Das Konzept der *Interfaces* lässt aber auch eine unabhängigere Schnittstellendefinition zu (Listing 5.10). Interfaces sind nicht an eine Implementation gebunden. Es kann beliebig viele Implementationen für ein Interface geben. Werden schließlich Interfaces mit Annahmen annotiert, geht das über das ursprüngliche Konzept von DBC hinaus. Der Schnittstellenvertrag gilt dann nicht nur zwischen einem Dienstanbieter und beliebig vielen Dienstbenutzern (1:n),

```
1 [Invariant("Saldo>=0")]
2 public class SparKonto
3 {
4     private decimal saldo;
5 
6     public decimal Saldo
7     {
8         get { return saldo; }
9     }
10 
11     [Requires("wert>0")]
12     [Ensures("Saldo␣==␣OLD␣Saldo␣+␣wert")]
13     public void Einzahlung(decimal wert)
14     {
15         saldo += wert;
16     }
17 
18     [Requires("wert>Saldo")]
19     [Ensures("Saldo␣==␣OLD␣Saldo␣-␣wert")]
20     public void Auszahlung(decimal wert)
21     {
22         saldo -= wert;
23     }
24 
25     [Requires("wert>0␣&&␣von.Saldo␣>=␣wert")]
26     [Ensures("von.Saldo␣==␣OLD␣von.Saldo␣-␣wert␣&&␣Saldo␣==␣OLD␣Saldo␣+␣wert")]
27     public void Transferiere(SparKonto von, decimal wert)
28     {
29         von.Auszahlung(wert);
30         saldo += wert;
31     }
32 }
```

Listing 5.9: DBC-Annahmen mit Annotationen

sondern zwischen beliebig vielen dieser Paare (n:m). Derjenige Dienstanbieter, der ein Interface implementiert, muss somit sicherstellen, dass die dort definierten Annahmen auch eingehalten werden.

In der CLI gibt es im Gegensatz zu Eiffel das Konzept von privaten Elementen einer Klasse. Es ist daher zu diskutieren, ob es erlaubt ist, sich in Annahmen auf die privaten Teile zu beziehen. Private Elemente können nur vom Dienstanbieter gelesen und geschrieben werden. Für denjenigen, der den Dienst in Anspruch nimmt, sind diese Teile nicht sichtbar. Es wäre also schwer nachzuvollziehen, warum der Vertrag zwischen beiden Parteien Bedingungen enthalten soll, die für einen Vertragsteilnehmer nicht sichtbar und somit auch nicht kontrollierbar sind. Es erscheint daher nicht sinnvoll, einen solchen Zugriff zu erlauben.

Bis jetzt besitzen die Annahmen keine Semantik. Sie liegen zwar als Zeichenkette in den entsprechenden Annotationen vor, werden aber noch nicht ausgewertet. Es wird also weder überprüft, ob sie syntaktisch und semantisch korrekt sind, noch wird sichergestellt, dass die Bedingungen auch zur Laufzeit eingehalten werden.

In Abbildung 5.5 ist der Kern der Lösung dargestellt. Er setzt sich im Wesentlichen aus den drei Aspekten `Ensures`, `Requires` und `Invariant` zusammen. Die Aspekte sind die bereits definierten CLI-Attribute zur Beschreibung eines DBC-Schnittstellenvertrags mit der zusätzlichen Implementation zur Überprüfung der mit ihnen definierten Annahmen.

Jeder dieser Aspekte implementiert jeweils ein Advice für die *Initialisierung* des Ausdrucks und mindestens ein Advice für die *Überprüfung* desselben. Bei der *Initialisierung* wird der übergebene Ausdruck in eine dynamisch zur Laufzeit generierte Klasse mit der Schnittstelle `IEvaluator` umgewandelt. Diese `IEvaluator`-Schnittstelle umfaßt wiederum zwei Methoden.

Die Methode `GetOldValues` sichert diejenigen Werte, auf die im Ausdruck über das `OLD`-Schlüsselwort zugegriffen wird. Die Methode `Eval` implementiert die Prüfung des Ausdrucks und liefert einen entsprechenden Wahrheitswert. Beiden Methoden stehen als Methodenpara-

```
1  [Invariant("Saldo>=0")]
2  public interface ISparKonto
3  {
4      decimal Saldo { get; }
5
6      [Requires("wert>0")]
7      [Ensures("Saldo == OLD Saldo + wert")]
8      void Einzahlung(decimal wert)
9
10     [Requires("wert>Saldo")]
11     [Ensures("Saldo == OLD Saldo - wert")]
12     void Auszahlung(decimal wert)
13
14     [Requires("wert>0 && von.Saldo >= wert")]
15     [Ensures("von.Saldo = OLD von.Saldo - wert && Saldo = OLD Saldo + wert")]
16     void Transferiere(SparKonto von, decimal wert)
17 }
```

Listing 5.10: DBC-Annahmen in Interfaces

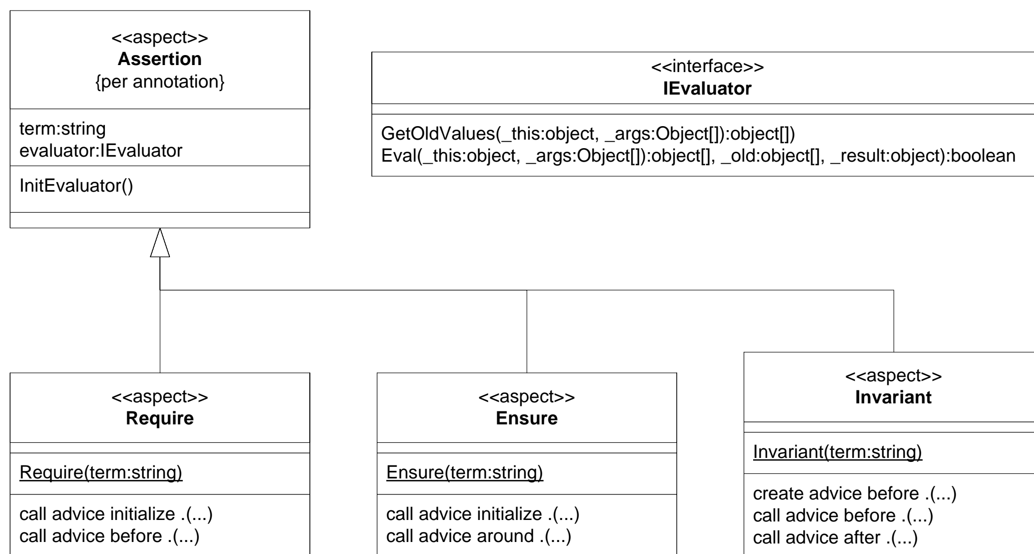


Abbildung 5.5: Der DBC-Aspekt

meter diejenigen Werte zur Verfügung, die zur Ermittelung des Wahrheitswertes der Annahme benötigt werden:

- `_this` enthält das Objekt, auf dem die Annahmen geprüft werden.
- `_args` enthält die Argumente der aufgerufenen Methode.
- `_old` enthält alle Werte, auf die mit dem `OLD`-Schlüsselwort zugegriffen wurde.
- `_retval` enthält den Rückgabewert der aufgerufenen Methode.

In der Initialisierungsphase jedes Aspekts wird der Ausdruck analysiert und dynamisch in die Methode `Eval` transformiert. Die derzeitige Lösung generiert hierzu C#-Quelltext und kompiliert ihn nach folgendem Schema:

1. Werte, auf die mit dem `OLD`-Schlüsselwort zugegriffen wird, werden indiziert und als typisierter Lesezugriff auf das `_old`-Feld kodiert. Zusätzlich werden sie in der Methode `GetOldValues` als indizierte Zuweisung auf ebendieses Feld codiert.
2. Der Wert des Schlüsselwortes `RESULT` wird als typisierter Lesezugriff auf den Parameter `_retval` codiert.
3. Alle Zugriffe auf Parameter der annotierten Methode werden nach der Parameternummer indiziert und als indizierter und typisierter Zugriff auf das Feld `_args` codiert.
4. Zugriffe auf Eigenschaftswerte oder Klassenvariablen werden über den Parameter `_this` codiert.

Die dynamische Implementation der `Eval`-Methode für die Nachbedingung von `Einzahlung` des obigen Beispiels würde also wie folgt aussehen:

```
return ((ISparKonto)_this).Saldo == (decimal)_old[0] + (decimal)
    _args[0];
```

Ist ein Ausdruck fehlerhaft formuliert, schlägt die Erzeugung der Klasse fehl und es kommt zu einem Laufzeitfehler. Während der Verwendung eines annotierten Exemplars kann nun über die so dynamisch generierten Evaluierungsklassen der Ausdruck überprüft werden. Die entsprechenden Advices werden wie folgt eingewoben:

- Invarianten haben zwei Advices, die die Methode `Eval` jeweils *vor* und *nach* jeder Benutzung des Exemplars aufrufen.
- Vorbedingungen haben ein Advice, das *vor* dem Aufruf der jeweils annotierten Methode die `Eval`-Methode ruft.
- Nachbedingungen haben zwei Advices. Der erste ruft *vor* dem Aufruf der jeweils annotierten Methode die Methode `GetOldValues`, der zweite ruft *nach* dem Aufruf die Methode `Eval`.

Durch die Verwendung dynamisch generierter Klassen hat jeder Ausdruck zwar einen entsprechenden Initialisierungsaufwand, dieser fällt jedoch jeweils nur ein einziges Mal während der gesamten Lebensdauer des ausführenden Prozesses an. Dementsprechend ist der zusätzliche Aufwand für die Prüfung der Annahmen relativ gering und nur von der Komplexität des Ausdrucks abhängig.

5.2.3 Diskussion der Lösung

Design by Contract ist ein sehr pragmatischer Ansatz, um das Verhalten von Exemplaren gemäß ihrer Spezifikation sicherzustellen. Obwohl die Ausdruckskraft gegenüber anderen Ansätzen - wie z. B. formalen Methoden zur Programmverifikation - limitiert ist, ist dieser Ansatz leicht zu verstehen und anzuwenden. Der große Vorteil bei der Verwendung von LOOM.NET besteht darin, dass sich jede CLI-Sprache mit dem Design-by-Contract-Konzept erweitern lässt. Die vorgestellte Lösung lässt sich als Bibliothek in jedes CLI-Projekt einbinden. Dabei ergibt sich die Wahl zwischen dem Programmiermodell von RAPIER-LOOM.NET, das die Verwendung einer Fabrikmethode zu Exemplarbildung voraussetzt oder dem Hinzufügen eines nachgelagerten Schrittes im Kompilationsprozess für die Verwebung der Aspekte mit GRIPPER-LOOM.NET.

Performancemessungen aus dem Abschnitt 3.7 haben gezeigt, dass der Mehraufwand nur bei der Erzeugung der Exemplare signifikant ist. Als alternative Implementation könnte - statt der Übersetzung von C#-Quelltext zur Laufzeit - die Verwendung dynamischer Sprachen für .NET, wie IronPhyton [Iron Phyton Team 2008] bzw. die unterliegende *Dynamic Language Rutime* [Hugunin 2007], einen Performancevorteil bringen. Bei der Benutzung dieser Exemplare hingegen fällt die Prüfung der Annahmen kaum ins Gewicht.

Es existieren aber auch bereits andere Lösungen, die sich mit der Umsetzung des DBC-Konzeptes beschäftigen. Die im Bereich der CLI-Sprachen am weitesten fortgeschrittene Umsetzung des Konzeptes stellt *Spec#* [Barnett u. a. 2004a,b] von Microsoft Research dar. Sie ist nicht nur eine einfache Portierung des Design-by-Contract-Konzepts auf die C#-Programmierspreche, sondern bringt auch eine Reihe von eigenen Erweiterungen mit. Dazu zählt, dass der Programmierer explizit steuern kann, wann Invarianten gültig sein müssen oder auch die Möglichkeit, Typen zu deklarieren, die keine Nullwerte annehmen dürfen. Die Sprache Spec# ist eine Obermenge von C#. Reguläre C#-Programme lassen sich mit dem Spec#-Kompiler übersetzen. Die Verwendung anderer CLI-Sprachen ist jedoch nicht möglich.

Eine Portierung der Eiffel-Programmiersprache ist *Eiffel.*NET [Simon und Stapf 2002]. Zu dieser Portierung gehört natürlich auch die vollständige Umsetzung des DBC-Konzepts. Der Kompiler erzeugt .NET-Assemblies, die von anderen .NET-Sprachen verwendet werden können. Bedingungen werden jedoch nur innerhalb der Eiffel.NET-Assemblies geprüft. Auch hier gilt, dass diese Lösung nur für die eine CLI-Sprache gültig ist.

Eine sprachunabhängige Variante - wie die vorgestellte Lösung - schlagen [Arnout und Simon 2001] mit dem *Contract Wizard* vor. Er erlaubt dem Benutzer, Schnittstellen bestehender Assemblies mit Bedingungen zu versehen. Im Anschluss wird für jede mit mindestens einer Bedingung versehene Klasse zur Verwendung des Stellvertretermusters [Gamma u. a. 1995, S. 207 ff.] eine Eiffel#-Klasse erzeugt. Referenzen zur originalen Klasse müssen nun durch den entsprechenden Stellvertreter ersetzt werden. Das erschwert die Lösung in der Anwendung, insbesondere wenn sich Assemblies gegenseitig referenzieren.

[Lippert und Lopes 2000] zeigt wie mit AspectJ Vor-, Nachbedingungen und Invarianten überprüft werden können. Für jede zu prüfende Klasse wird ein Aspekt implementiert, der die Prüfung der Bedingung als Advice implementiert. Die Klasse `Konto` hätte dann beispielsweise den Aspekt `Kontovertrag`, der sämtliche Bedingungen für die Klasse enthält. Bei dieser Lösung ist für einen Programmierer nicht sofort ersichtlich, welche Methoden mit welchen Bedingungen versehen wurden, da die Aspekte separat implementiert sind. [Lippert und Lopes 2000] schlagen daher vor, die Bedingungen zusätzlich in den *Javadoc*-Kommentaren zu notieren.

Weiterhin gibt es viele Arbeiten, die sich damit beschäftigen, wie das Design-by-Contract-Konzept in die verschiedensten Sprachen umgesetzt werden kann. Als Beispiel sei hier genannt *Java* [Bartetzko u. a. 2004; Karaorman u. a. 1999; Kramer 1998; Meemken; Szathmary 2002], *Perl* [Conway und Goebel 2001], *Python* [Plösch 1997], *Ruby* [Hunt 2002], Ada [Luckham u. a. 1987], *Lisp* [Hölzl], *Smalltalk* [Carrillo-Castellón u. a. 1996] und *C++* [Guerreiro 2002].

Die vom Verfasser vorgestellte Lösung zeigt insbesondere in der Kombination von Annotationen, AOP und den LOOM.NET-Aspektwebern ihre Stärken - erlaubt sie es doch, dass Design by Contract fast als integraler Bestandteil einer CLI-Sprache wahrgenommen werden kann. Insbesondere wurde gezeigt, wie einfach sich imperative Sprachen mit deklarativen Sprachelementen erweitern lassen.

5.3 Zusammenfassung

Mit den LOOM-Konzept lassen sich neue Programmierkonzepte auch ohne großen Aufwand in statische Programmiersprachen integrieren. LOOM-Aspekte übernehmen hier die Rolle neuer Schlüsselwörter. Die Bedeutung der Schlüsselwörter ist direkt in den Aspekten codiert. Das wurde an zwei Beispielen ausführlich demonstriert.

Context# ist ein neues Programmierkonzept, das erlaubt, Anwendungen einfach an wechselnde Umgebungsparameter anzupassen. Der Programmierer hat die Möglichkeit, Methoden kontextabhängig zu erweitern. Repräsentiert wird der Kontext über Ebenen, die aktiviert und deaktiviert werden können. Auf jeder Ebene lässt sich für jede Methode eine zusätzliche Implementation hinterlegen. Die Definitien der Ebenen erfolgt durch einen LOOM.NET-Aspekt. Insgesamt benötigt die Lösung weniger als 200 Zeilen Quelltext.

Die Design-by-Contract-Erweiterung ermöglicht es, Schnittstellen mit Annahmen, die für die Schnittstelle gelten sollen, zu versehen. Solche Annahmen sind Vor- und Nachbedingungen sowie Invarianten, deren Einhaltung durch die unterliegenden Aspekte garantiert wird. Auch hier erlaubt LOOM eine nahtlose und intuitive Umsetzung des Konzepts, ohne einen speziellen Kompiler oder eine eigene Ausführungsumgebung zu benötigen. Insbesondere lässt sich diese Lösung auf alle Sprachen der *Common Language Infrastructure* [Microsoft Corporation u. a. 2006] anwenden.

6 Projekte mit LOOM.NET

In diesem Kapitel wird eine Auswahl von Projekten vorgestellt, die unter der Verwendung der LOOM-Konzepte realisiert wurden. Sie zeigen, dass sich das vom Autor entwickelte Paradigma und die von ihm entwickelten Werkzeuge auch im praktischen Einsatz bewährt haben. Zwei Projekte, in die der Autor maßgeblich involviert war, sollen dabei etwas ausführlicher untersucht werden.

Bei dem ersten Projekt handelt es sich um eine Industriekooperation mit der Deutschen Post IT-Services, die in den Jahren 2003 bis 2004 durchgeführt wurde. Hier ging es zum einen darum, die generelle Anwendbarkeit der LOOM-Konzepte in einem großen Industrieprojekt nachzuweisen, zum anderen im Detail darum, LOOM-Aspekte für den Softwaretest und zur Verbesserung der Softwarequalität einzusetzen.

Das zweite Projekt, an dem der Autor maßgeblich mitwirken konnte, ist *DSUP* - eine Plattform zur Softwareaktualisierung. Sie ermöglicht eine Aktualisierung von Anwendungen während der Laufzeit. Durch den Verwendung der LOOM-Konzepte kann ein Anwendungsprogramm mit sehr geringem Aufwand so angepasst werden, dass deren Komponenten im laufenden Betrieb der Anwendung ausgetauscht werden können. Als eine herausragende Besonderheit gilt, dass die Ausführungsumgebung nicht angepasst werden muss; die dynamische Softwareaktualisierung wird bereits durch die Verwendung einer DSUP-Bibliothek ermöglicht.

Schließlich werden noch weitere LOOM-Projekte vorgestellt, an denen der Autor selbst nicht oder nur indirekt beteiligt war, die aber aufgrund ihrer Aufgabenstellung besonders erwähnenswert sind. Das *Managementframework für die Windows Fernwartungsschnittstelle* leistet zum Beispiel einen wesentlichen Beitrag zur Vereinfachung des Softwarebetriebs. Auch hier liegt der Schlüssel in der konsequenten Umsetzung der LOOM-Konzepte.

6.1 Eine Studie im industriellen Umfeld

In den Jahren 2003 und 2004 wurde über sechs Monate in Zusammenarbeit mit dem Hasso-Plattner-Institut bei der Deutschen Post IT-Solutions (heute: Deutsche Post IT-Services Center Europe, kurz DP-ITSC) eine Studie zum Einsatz von AOP in einem großen IT-Projekt durchgeführt. Es handelte sich hierbei um die Neuentwicklung des EPOS-Systems, das in Schalterterminals der Deutschen Post-Filialen zum Einsatz kommt.

Das EPOS-Kassensystem umfasst über 150 Vorgangsarten (d.h. Geschäftsvorfallstypen), die täglich circa 6 Millionen Datensätze erzeugen und 20 Millionen Transaktionen im Backend anstoßen. Das System ist in 8.000 Filialen und an 20.000 Arbeitsplätzen verfügbar und bildet die Geschäftslogik aller Kundenprozesse in den Filialen ab. Die anfallenden Daten werden vom Schalterterminal an das *Backend* (eine zentrales Rechenzentrum) über eine verteilte Kommunikationsinfrastruktur weitergeleitet, archiviert und unter Anwendung von 2.800 Regeln fachlich geprüft und verdichtet. Von dort werden die Daten an 21 Umsysteme weitergeleitet. Das sind eigenständige Systeme, die mit dem Backend direkt kommunizieren. Ein solches Umsystem ist z. B. die SAP-Buchhaltung.

Die Systemarchitektur von EPOS basiert auf zwei technischen Frameworks: CARBON für die Schalterterminals und Titanium für das Backend-System, die eine saubere Trennung der Systemschichten ermöglichen. Technologisch gehört das gesamte Projekt zu einem der

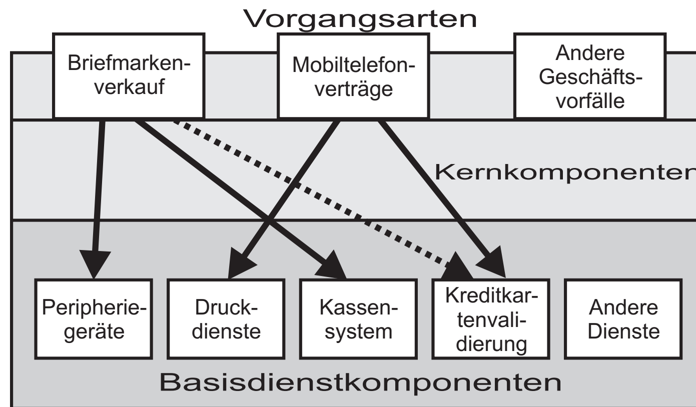


Abbildung 6.1: Aufbau von CARBON

großen .NET-Systeme, geschrieben in C# mit über 1.200.000 Programmzeilen. Die Software-Architektur des Backend basiert auf der Microsoft BizTalk Plattform [Microsoft Corporation 2007a].

In Abbildung 6.1 ist der vereinfachte schematische Aufbau des CARBON-Frameworks dargestellt. Für jeden Geschäftsvorfall wird jeweils eine *VGA-Komponente* vorgesehen (oben). In ihr sind die Bildschirmmasken, die Ablaufsteuerung und die Geschäftslogik für den entsprechenden Vorfall enthalten. Den VGA-Komponenten stehen zusätzlich sogenannte *Basisdienste* zur Verfügung. Über diese wird beispielsweise der Zugriff auf Stammdaten, die Ansteuerung von Peripherie (Drucker, Handscanner usw.) oder auch die Verifizierung von Daten vorgenommen. Das Zusammenspiel der Komponenten (Exemplarbildung, Verknüpfung) wird durch den *Kernel* realisiert.

Ein Geschäftsvorfall wird gestartet, indem der Bediener die zugehörige VGA-Komponente aufruft. Das geschieht im Allgemeinen durch die Eingabe einer zum Vorgang gehörenden eindeutigen Nummer. Der Kern lädt die Komponente und ruft deren Initialisierungsfunktion auf. Daraufhin wird die zum Vorgang gehörende Bildschirmmaske angezeigt. Der Bediener kann nun seine Eingaben tätigen, während die Komponente im Hintergrund den Maskenfluss steuert und die Daten der Eingabe verarbeitet. Hierzu werden gegebenenfalls Basisdienste erzeugt und verwendet. Zum Beispiel kann ein solcher die Gültigkeit einer Kreditkarte validieren oder den aktuellen Kontostand abfragen. Schließlich werden die verarbeiteten Daten an das Backend übertragen.

Beim EPOS-System handelt es sich um eine Anwendung, die besondere Anforderung an das Design, die Entwicklung und den Betrieb stellt. So wird über das EPOS-System der größte Teil der buchungsrelevanten Geschäftsvorfälle aller Postfilialen Deutschlands abgewickelt. Fehlfunktionen oder ein Ausfall des Systems hätten materielle und immaterielle Schäden bei der Deutschen Post zur Folge. Buchungen, die nur fehlerhaft oder gar nicht ausgeführt werden, müssten aufwendig und teuer nachgepflegt werden. Eine ausbleibende Verbuchung von Tagesumsätzen senkt die Liquidität des Unternehmens, was zu zusätzlichen Zinsbelastungen führen kann. Nicht zuletzt hat ein nicht funktionierendes System auch negativen Auswirkungen auf die Reputation des Unternehmens.

Mit einem von der DP-ITSC durchgeführten Forschungs- und Entwicklungsprojekt sollte evaluiert werden, ob die Verwendung von aspektorientierten Programmiertechniken einen Vorteil bei der Erfüllung der hohen Anforderungen an das EPOS-Projekt bringt. Diese Untersuchung wurde in Zusammenarbeit mit dem Hasso-Plattner-Institut durchgeführt.

6.1.1 Ziele der Studie

Das EPOS-Projekt hatte vor dem Beginn der Studie bereits zwei Entwicklungszyklen absolviert und stand kurz vor dem Start der Entwicklung für die Version 3.0. Daher war es mög-

lich, auf bereits gesammelte Erfahrungen zurückzugreifen und die Studie praxisorientierter an konkreten Problemstellungen auszurichten. Hierzu wurden im Vorfeld mit dem verantwortlichen Architekten, dem Projektleiter und einigen Programmierern Gespräche geführt. Dabei kristallisierten sich folgende drei Fragestellungen als für den Auftraggeber besonders wichtig heraus:

- Kann die Softwarequalität durch LOOM verbessert werden?
- Ist es möglich, Querschnittsanforderungen in EPOS mit aspektorientierten Techniken zu implementieren, und ergeben sich dadurch Vorteile?
- Gibt es durch die Anwendung von LOOM Einsparpotential bei der Produktentwicklung?

Ziel sollte es nicht sein, alle überschneidenenden Belange im System zu identifizieren und als Aspekte zu extrahieren, ebenso soll keine umfassende Analyse der Programmierfehler und des Softwareentwicklungsprozesses stattfinden. Vielmehr sollte den Verantwortlichen in der Industrie das Potential der aspektorientierten Programmierung veranschaulicht werden.

Als ersten Schritt wurden die häufigsten Fehlerquellen bei der Programmierung ermittelt. Hierzu wurde eine Analyse des Quelltextes vorgenommen und mit zwei Programmierern des CARBON-Basissystems und drei Entwicklern der Vorgangsarten über die aus Ihrer Sicht vorhandenen Probleme gesprochen.

Aus organisatorischen Gründen war es leider jedoch nicht möglich, mit Verantwortlichen der Qualitätssicherung zu sprechen und aus rechtlichen Gründen stand auch nur für einen Teil des Systems der Quelltext zur Verfügung.

6.1.2 Ergebnisse der Analyse

Bis zur CARBON-Version 2.1 war das sogenannte Tracing von Methodenaufrufen, das heißt das Loggen bei Beginn und Ende einer Methode, Bestandteil der Anforderungen des Auftraggebers: Auf diese Weise sollten auch im Betrieb die CARBON-internen Abläufe nachvollziehbar gemacht werden, um Fehler im Programm leichter lokalisieren zu können. Es stellte sich jedoch heraus, dass sich die bisherige Lösung sehr negativ auf die Effizienz des Programms auswirkte. Auch bei ausgeschalteteten Logging und Tracing waren stets die Aufrufe in das entsprechende Logging- und Tracingmodul in jeder Methode vorhanden, was mitunter dazu führte, dass die Reaktionszeit zwischen Eingabe und Ausgabe mehrere Sekunden betrug.

Neben dem Effizienzproblem wurde aus Sicht der Wartbarkeit die hohe Zahl an Logging- und Tracingpunkten bemängelt. In der Version 2.1 von CARBON summierten sich die Zahl der Aufrufe auf 2133 jeweils für den Eintritt und das Verlassen einer Methode. Die Zahl der Programmzeilen für das gesamte Projekt war mit knapp 70.000 zu diesem Zeitpunkt noch relativ gering, da in dieser Version nur einige wenige Vorgänge umgesetzt waren. Für die Produktivversion wurde eine um den Faktor zehn höhere Zahl an Programmzeilen und demnach auch entsprechender Loggingpunkte erwartet.

Ein weiteres Problem ergab sich aus einer hohen Fluktuation von Programmierern. Zwischen den Versionen wurden teilweise komplette Programmiererteams ausgetauscht, da die alten Teams nach dem Abschluss einer alten Version bereits in anderen Projekten verplant wurden und zu Beginn der Entwicklungsphase der neuen Version nicht mehr zur Verfügung standen. Dieses organisatorische Problem ließ sich auf Grund des vorgegebenen Vorgehensmodells für die Produktentwicklung nicht lösen. Die insbesondere aus der Unerfahrenheit der Programmierer resultierenden Fehler betrafen unter anderem Querschnittsfunktionen der zu implementierenden Geschäftslogik. Zum Beispiel gab es konkrete Anforderungen, wie die Behandlung von Ausnahmen zu erfolgen hat und Ausnahmen an übergeordnete Kernmodule weitergeleitet werden sollen. Diese Anforderungen wurden jedoch oftmals ungenügend umgesetzt.

	Beobachtung	**häufigste Ursache**
1	Ausnahmen wurden überhaupt nicht behandelt.	Die Notwendigkeit der Ausnahmebehandlung war nicht bekannt, bzw. wurde aus Zeitgründen vernachlässigt.
2	Ausnahmen wurden für bestimmte Anweisungsblöcke nicht behandelt.	Es wurde davon ausgegangen, dass die Blöcke keine Ausnahme werfen.
3	Es wurden nicht alle möglichen Ausnahmen gefangen.	es wurde übersehen, dass auch Ausnahmen eines anderen Typs auftreten können.
4	Es wurden falsche Ausnahmen an das Basissystem weitergeleitet.	Tippfehler oder falsche Interpretation der Spezifikation.

Abbildung 6.2: Fehler bei der Ausnahmebehandlung

Abbildung 6.2 zeigt die häufigsten Fehler, die bei der Ausnahmebehandlung auftreten. Sie spiegelt zum einen die Sicht der interviewten Entwickler, andererseits auch die Ergebnisse der Quelltextanalyse wider. Nachfolgend werden nun die Lösungsvorschläge diskutiert.

6.1.3 Verbesserung von Logging und Tracing

Wie bereits erwähnt, werden im CARBON-Framework die Vorgangsarten und die dazugehörigen Basisdienste Tracing für fast alle Methoden eingesetzt. Dabei wird auf die Tracing-Funktionalität der .NET-Bibliothek (insbesondere `System.Diagnostics.Trace`) zurückgegriffen, die durch verschiedene Erweiterungen und Hilfsfunktionen in der CARBON-Laufzeitumgebung für die Benutzung in den VGA's vereinfacht wird.

Zur Filterung von Tracing-Meldungen anhand der Relevanz definiert CARBON verschiedene Ebenen: `InfoBase`, `InfoImportant`, `InfoCritical`, `InfoExceptionLog`, `InfoLog`. Die ersten drei stehen jeweils für das niedrigste, mittlere bzw. höchste Tracing-Level und sind bzgl. ihres Ziels frei konfigurierbar. Bei der Verwendung der übrigen zwei wird garantiert, dass damit gekennzeichnete Meldungen auf jeden Fall in die entsprechende Logdatei geschrieben werden.

Das CARBON-Framework erlaubt das Loggen der Tracing-Einträge in verschiedene Senken: Event-Log, Log-Datei, Ausnahme-Log-Datei und Debug-Queue. Dabei steht Event-Log für die Windows-Ereignisanzeige und Debug-Queue normalerweise für das Ausgabefenster in Visual Studio.NET. Die restlichen beiden stehen für frei konfigurierbare Log-Dateien. In der Konfigurationsdatei für den CARBON-Klienten ist weiterhin pro Level konfigurierbar, welche Einträge in welche Senken geschrieben werden.

Zur Kategorisierung (und damit Filterung) der Tracing-Meldungen nach inhaltlichen Kriterien definiert CARBON insgesamt 25 verschiedene Schalter, von denen einer beim Loggen einer Meldung angegeben werden muss. Zum Zeitpunkt der Studie wurden davon jedoch nur zwei genutzt. Einer zeigt an, dass es sich um eine VGA handelt, der andere wird für Basisdienste genutzt.

Das CARBON-Framework stellt unter anderem die Methoden

- `TracingOperations.BeginMethod` und
- `TracingOperations.EndMethod`

zur Verfügung, um eine einheitliche Formatierung der Ausgaben für das Tracing von Methodenaufrufen zu vereinfachen. Diese Methoden übernehmen als Parameter den anzuwendenden Schalter sowie die Argumente der zu loggenden Methode und liefern einen formatierten String zurück, der dann via `Trace.WriteLine()` dem Logging-System übergeben werden kann.

Um Informationen über die aktuelle Methode zu bekommen, greifen `BeginMethod` und `EndMethod` auf die - gemessen am Laufzeitaufwand - relativ teure Operation `GetCurrentStack-Frame` zurück. Diese liefert den im aktuellen Thread gültigen Aufrufstack. Hieraus wird dann beispielsweise der Name der aufgerufenen Methode extrahiert.

Weiterhin erweist sich als aufwendig, dass alle Parameter der aufgerufenen Methode an `BeginMethod` und `EndMethod` einzeln als formatierte Zeichenkette übergeben werden müssen. Aus diesem Grund wird die Hilfsmethode `GetMethodParametersTracing` zur zuverlässigen Umwandlung der Parameter verwendet. Letztlich sieht ein Tracing-Aufruf zu Begin einer Methode mit den beiden Methodenparametern `returnCode` und `laenderCodeListe` wie folgt aus:

```
Trace.WriteLine(
  TracingOperations.BeginMethod(TraceSupport.SwitchName,
    TraceSupport.GetMethodParametersTracing(returnCode,
        laenderCodeListe)));
```

Und das entsprechende Gegenstück am Ende der Methode:

```
Trace.WriteLine(
  TracingOperations.EndMethod(TraceSupport.SwitchName,
    TraceSupport.GetMethodParametersTracing(returnCode,
        laenderCodeListe)));
```

In der für die Studie herangezogenen CARBON-Version fanden sich 2133 Stellen, an denen dieser Quelltext angepasst auf die jeweilige Methode eingefügt war,und das jeweils zu Beginn und bei Verlassen der Methode. Neben der großen Anzahl redundanter Quelltextzeilen verbirgt diese Vorgehensweise noch ein weiteres Problem. In den meisten Fällen wurde der Aufruf zum Mitschneiden des Verlassens einer Methode nicht in einen `try-finally`-Block programmiert. Das hat zur Folge, dass im Falle einer Ausnahme das Verlassen der Methode nicht protokolliert wird. Der Quelltext entspricht damit nicht der Spezifikation.

Die Umsetzung der Aufgabenstellung mit LOOM.NET entspricht dem in Kapitel 4 vorgestellten Dekorierermuster. Zuerst werden aus dem Quelltext sämtliche Tracing-Aufrufe entfernt. Ein Tracing-Aspekt mit zwei Advices, die vor und nach dem Aufruf einer Methode eingewoben werden, übernimmt den Aufruf der Methoden `BeginMethod` und `EndMethod`. Methodenname und Methodenparameter können mit LOOM.NET-inhärenten Mitteln aus dem Joinpoint gezogen werden.

Die Verwebung der Aspekte kann global auf der Ebene der jeweiligen Assembly erfolgen. Hierzu wird die Assembly einmal mit dem Tracing-Aspekt annotiert. Zum Zeitpunkt der Studie stand der statische Aspektweber GRIPPER-LOOM.NET leider noch nicht zur Verfügung, sodass nur eine Verwebung zur Laufzeit mit RAPIER-LOOM.NET in Frage kam. Da in CARBON die Exemplarerzeugung von Vorgangsarten und Basisdiensten in einer zentralen Fabrikmethode erfolgt, war nur an dieser Stelle eine Änderung des Quelltexts notwendig.

6.1.4 Revision der Ausnahmebehandlung

Ausnahmen sind ein üblicher Mechanismus, um Fehler während des Programmablaufes zu signalisieren und zu behandeln. Hierbei wird von der Funktion, in der der Fehler auftrat, ein Exemplar vom Typ `Exception` erzeugt und mit dem Schlüsselwort `throw` nach oben „geworfen“. Jede Funktion, die im Aufrufstapel oberhalb der werfenden Funktion steht, hat danach die Möglichkeit, die Ausnahme durch eine Ausnahmebehandlung zu „fangen“ und auf den Fehler zu reagieren. In CARBON gibt es verschiedene Ausnahmetypen, anhand derer unter anderem entschieden wird, wer der Adressat der entsprechenden Fehlermeldung ist. Tritt während der VGA-Verarbeitung ein Fehler auf, wird im Allgemeinen eine spezielle

```
1 [Validate(Aspects.FREMDWAEHRUNGSBETRAG)]
2 protected ValidateResult ValidFremdwaehrungsBetrag(DoubleAspect customerName)
3 {
4     try
5     {
6     /// fachlicher Quellkode
7     }
8     catch(Exception ex)
9     {
10        // Ausnahme des FDS FdsWaehrungsInfo
11        throw new BaseServices.Fds.FdsWaehrungsInfo.CmpException(ex.Message);
12    }
13 }
```

Listing 6.1: Umwandlung von Ausnahmen in CARBON

<<aspect>>
ExceptionHandler

type:Type
message:string

ExceptionHandler(type:Type)
ExceptionHandler(type:Type, message:string)

advice after throwing call .(...)

Abbildung 6.3: Aspekt zur Ausnahmebehandlung in VGA's

Ausnahme vom Typ `RetailException` geworfen. Diese führt dazu, dass der Benutzer über den aufgetretenen Fehler informiert wird.

Die VGA verwendet selbst auch Funktionen, die Ausnahmen anderen Typs werfen. Das ist insbesondere beim Aufruf von Funktionen aus der .NET-Klassenbibliothek der Fall. Hier muss die VGA eine Umwandlung in die „richtige" Ausnahme vornehmen. Der entsprechende Quelltext, wie er üblicherweise in den VGA's gefunden wird, ist exemplarisch in Listing 6.1 zu finden. Dieses Muster wiederholt sich im gesamten Quelltext, teilweise zwanzigmal pro VGA. Der einzige Unterschied ist jeweils der Typ der Ausnahme.

Diese Beobachtung führt zu dem Schluss, dass durch den Einsatz von LOOM eine Vereinfachung des Quelltexts möglich ist. Der Programmierer kann die Art der Ausnahme als Attribut an die Methode schreiben. Die eigentliche Ausnahmebehandlung und die Umwandlung der Ausnahme können durch einen Aspekt erfolgen. Spätere Änderungen an der Art der Ausnahmebehandlung wären für den Programmierer transparent.

Bei der Implementierung des Aspekts wurde wieder auf das Dekorierermuster Abschnitt 4.3 zurückgegriffen. Der Aspekt enthält einen Advice, der in die Ausnahmebehandlung der annotierten Methode eingewoben wird. Abbildung 6.3 zeigt den Aufbau des Aspekts. In den Konstruktoren wird die im Fehlerfall zu erzeugende Ausnahme übergeben (`type`). Der zweite Konstruktor übernimmt zusätzlich einen Meldungstext, der im Fehlerfall an das Ausnahmeexemplar weitergereicht werden soll. Der konkrete Ablauf im Advice ist in Abbildung 6.4 dargestellt.

Aus Sicht des VGA-Programmierers gestaltet sich die Verwendung des Aspektes einfach. Er muss nur alle Methoden, die eine Ausnahmebehandlung benötigen, mit `ExceptionHandler` annotieren. Wird der Aspekt als Klassenattribut verwendet, so ist er für alle Methoden der öffentlichen Klassenschnittstelle gültig. Das Listing 6.1 ist in Listing 6.2 noch einmal unter Verwendung des LOOM-Aspekts dargestellt.

Eine Einschränkung bei der Verwendung des Aspekts ergibt sich daraus, dass die Ausnahmebehandlung ausschließlich für die gesamte Methode erfolgt. Benötigen einzelne Anwei-

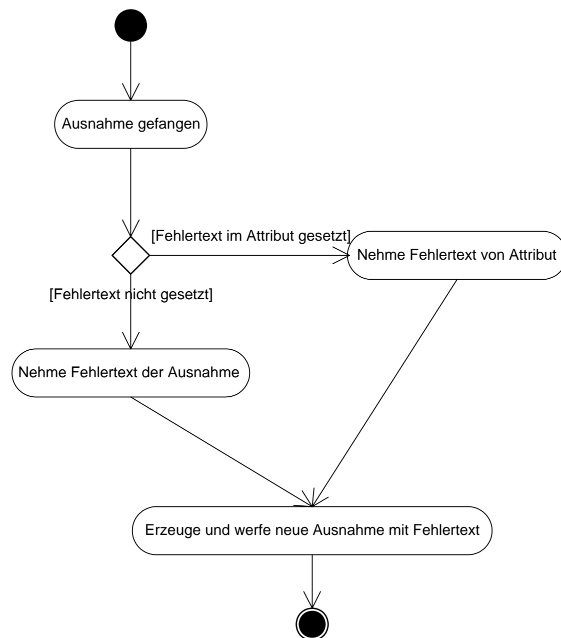


Abbildung 6.4: Ablauf im Fehlerfall

```
1 [Validate(Aspects.FREMDWAEHRUNGSBETRAG)]
2 [ExceptionHandler(typeof(BaseServices.Fds.FdsWaehrungsInfo.CmpException))]
3 protected ValidateResult ValidFremdwaehrungsBetrag(DoubleAspect customerName)
4 {
5     /// fachlicher Quelltext
6 }
```

Listing 6.2: Ausnahmebehandlung in CARBON mit LOOM

sungsblöcke innerhalb einer Methode eine eigene Ausnahmebehandlung, so ist das nur über die Auslagerung des entsprechenden Quelltexts in separate Methoden möglich.

Dieser Aspekt ist ein Beispiel, wie fachlicher Quelltext von immer wiederkehrenden nicht-funktionalem Belang - der Ausnahmebehandlung - getrennt werden kann. Es ist unter anderem mit diesem Aspekt möglich, unterschiedliche Ausnahmebehandlungsstrategien zu implementieren. Für die Qualitätssicherung könnten die Meldungen z. B. mit zusätzlichen Informationen angereichert werden. Durch die strikte Trennung vom fachlichen Quelltext kann für jede Anforderung eine eigene Version des Aspekts kompiliert werden.

Einsparungen ergeben sich dahingehend, dass `try-catch`-Blöcke in der Fachlogik nicht mehr ausprogrammiert werden müssen. Vielmehr werden nur die Klassen bzw. Methoden durch Attribute annotiert, um das gewünschte Verhalten zu realisieren.

6.1.5 Ergebnisse der Studie

Die entwickelten Aspekte wurden in zwei ausgewählten Vorgangsarten eingesetzt. Hierzu wurde zuerst der Quelltext für das Logging und Tracing sowie für die Ausnahmebehandlung entfernt. Anschließend wurden die Vorgangsarten mit den Aspekten annotiert, sodass die ursprüngliche Funktionalität wiederhergestellt war.

Es konnte nachgewiesen werden, dass sich ein Logging und Tracing und die Ausnahmebehandlung mit voller Funktionalität sehr einfach realisieren lassen. Das bedeutet, dass die übliche Instrumentierung des Quelltexts mit Tracing-Anweisungen und Ausnahmebehandlung nicht mehr erfolgen muss. Hieraus ergibt sich ein enormes Einsparungspotential bei der Entwicklung, wobei sich die Qualität des Quelltexts sogar verbessert. Die durch den VGA-Entwickler implementierten Methoden enthalten dann tatsächlich nur noch den funktionalen Quelltext, die Lesbarkeit ist deutlich erhöht.

Insbesondere das aufgezeigte Einsparungspotential durch die Verwendung beider Aspekte ist ein guter Ansatzpunkt für eine konkrete Implementierung im realen Produkt. Es wurde gezeigt, dass bis zu 12% weniger Quelltext durch den Programmierer geschrieben werden muss und trotzdem die gleiche Funktionalität und Performance erreicht wird.

Ziel der Studie war es, die Potentiale der aspektorientierten Programmierung in einem großen Softwareprojekt aufzuzeigen. Die abschließenden Gespräche mit den Beteiligten und Verantwortlichen zeigten hier durchaus positive Resonanz. Wenngleich eine Entscheidung für den Einsatz von LOOM.NET vorerst nicht getroffen wurde, hatte die Studie dennoch Einfluss auf die Weiterführung des Projekts.

In der Version 3.0 wurde das Logging und Tracing komplett aus dem Pflichtenheft gestrichen. Zum einen stand der Mehrwert bei der Qualitätssicherung und Fehlersuche in einem deutlichen Missverhältnis zum Aufwand. Zum anderen wurde bewußt, dass es bei Bedarf relativ einfach ist, das Logging und Tracing mit LOOM.NET zu reaktivieren. Auch die Ausnahmebehandlung wurde aufgrund der Analysen der Studie komplett überarbeitet. Der hier beschriebene Aspekt war somit in der Version 3.0 obsolet geworden.

Vorbehalte zum Einsatz RAPIER-LOOM.NET waren ausschließlich rechtlicher Natur. Es gab Bedenken bezüglich der Verantwortlichkeit bei eventuell vorhandenen Softwarefehlern, die zu Produktionsausfällen führen.

6.2 Dynamische Softwareaktualisierung mit DSUP

Immer kürzere Produktzyklen, die Gefahr von Angriffen auf Sicherheitslücken und das Bedürfnis, sich an wechselnde Umgebungen anzupassen, machen es notwendig, Software möglichst während der Laufzeit aktualisieren zu können. Vor allem langlaufende Anwendungen im Serverbereich erfordern es, dass ihre Dienste ständig zur Verfügung stehen und daher für eine Aktualisierung nicht neu gestartet werden sollten.

Ein Anwendungsprogramm wird üblicherweise unter zwei Gesichtpunkten weiterentwickelt. Zum einen werden neue Anwendungsfälle implementiert, zum anderen sollen Fehler in der vorangegangenen Version behoben werden. Die klassische Weiterentwicklung des Programms, beginnend beim Design, der Implementierung und dem abschließenden Softwaretest, geht dabei von einem statischen Austausch der Softwarekomponenten aus. Die neuen und geänderten Komponenten werden in das System eingespielt, während die Anwendung selbst nicht aktiv ist. Sollen die Änderungen hingegen dynamisch ins laufende System eingespielt werden, spielt der Zustand, den das System hat, eine Rolle. Das hat einerseits Einfluss darauf, *was* aktualisiert werden kann, *wie* aktualisiert werden muss und *wann* aktualisiert werden darf. Neben diesen eher technischen Fragen ist andererseits auch die Frage zu klären, welchen Einfluss die Möglichkeit einer dynamischen Aktualisierung auf das Design der Anwendung hat und wie die Anwendung zu testen ist.

Die Anwendungen bestehen typischerweise aus einer Vielzahl von Komponenten, deren Zusammenwirken ihre Funktionalität darstellt. Bei der dynamischen Softwareaktualisierung geht es darum, einzelne Komponenten durch eine neue Version zu ersetzen. Das kann geschehen, um Fehler in der Komponente zu bereinigen, als auch um eine neue Funktionalität hinzuzufügen.

Für die dynamische Softwareaktualisierung wurde zusammen mit Andreas Rasche ein System, die *Dynamic Software Update Plattform* (kurz: *DSUP*), entwickelt. Teile der vorliegenden Arbeit wurden bereits in [Rasche und Schult 2007; Rasche, Schult und Polze 2005] veröffentlicht. Bei der Entwicklung von DSUP standen folgende Punkte im Mittelpunkt:

- **Effizienz**: Die Zusatzkosten bei der Abwicklung des Anwendungsprogramms sollten gering sein.
- **Flexibilität**: Jeder Teil des Anwendungsprogramms sollte austauschbar sein, die Einschränkungen für die Programmierung sollten gering sein.
- **Robustheit**: Die Gefahr von Fehlern und Abstürzen sollten minimiert werden.
- **Einfachheit**: Für den Entwickler sollte nur wenig zusätzlicher Aufwand entstehen, die unterliegende Laufzeitumgebung sollte nicht verändert werden.

Das Austauschen von Komponenten bedeutet im Allgemeinen eine Änderung an der Struktur und der Logik der Anwendung. Hierbei besteht die Gefahr, dass die Konsistenz der Anwendungsdaten nach der Aktualisierung nicht mehr gewährleistet ist. Das ist insbesondere der Fall, wenn sich während der Abwicklung einer Operation die Struktur der Daten, auf denen sie ausgeführt wird, verändert. Im Gegensatz zur statischen muss bei der dynamischen Softwareaktualisierung daher der Zustand des laufenden Programms berücksichtigt werden.

Im Bereich der dynamischen Rekonfiguration, einem mit der dynamischen Softwareaktualisierung verwandten Gebiet, existieren Algorithmen, die dieses Problem lösen [Bidan u. a. 1998; Kramer und Magee 1990; Miguel Alexandre Wermelinger 1999; Moazami-Goudarzi 1999; Wermelinger 1997]. Grundidee ist es, die Anwendung in einen rekonfigurierbaren Zustand zu bringen, um dann die Rekonfigurationsoperationen auszuführen. Das erfolgt im Allgemeinen durch „Einfrieren" der Anwendung in einem konsistenten Zustand. Die Herausforderung besteht darin, die Anwendung in einen solchen konsistenten Zustand zu überführen.

Für [Kramer und Magee 1990] bedeutet Konsistenz zunächst ein Zustand, von dem die Abwicklung des Programms fortgeführt werden kann, ohne dass diese in einem Fehlerzustand endet. Ein wenig präziser formuliert [Moazami-Goudarzi 1999] die Konsistenz von verteilten Systemen, indem er für die Korrektheit fordert, dass das *System strukturell integer* ist, die *Kommunikationspartner gegenseitig einen gültigen Zustand* haben und die *anwendungsspezifischen Invarianten des Systems gültig sind*. Bezogen auf ein Anwendungsprogramm innerhalb eines Prozesses lassen sich daraus die folgenden drei Bedingungen ableiten:

- **Ausführungsintegrität:** Wird eine Operation auf einem Objekt abgewickelt, so kann dieses Objekt auf anderen, von ihm referenzierten Objekten Operationen ausführen und auf Daten in seinen Attributen zugreifen, um das Ergebnis der Operation zu berechnen. Wird jedoch während der Abwicklung ein Interaktionspartner oder ein Datum von außen unvorhergesehen ausgetauscht, so führt das möglicherweise zu einem falschen Ergebnis.

- **Strukturelle Integrität:** Objekte interagieren miteinander über die von ihnen bereitgestellten Schnittstellen. Die Schnittstellen zwischen den Objekten müssen zu jedem Zeitpunkt beiden Kommunikationsteilnehmern vollständig bekannt sein.

- **Anwendungsspezifische Invarianten:** Anwendungsspezifische Invarianten sind Prädikate, die Aussagen über den Zustand einer Teilmenge des Anwendungsprogramms treffen. Solange diese Prädikate erfüllt sind, ist das System konsistent. Ein Beispiel aus der automatischen Speicherverwaltung besagt, dass der Referenzzähler eines Objekts Null sein muss, wenn es nicht mehr referenziert wird.

Eine Anwendung wird im Folgenden als konsistent bezeichnet, wenn sie alle drei Bedingungen erfüllt. DSUP stellt für jede dieser drei Bedingungen Mechanismen bereit, mit denen diese erfüllt werden können. Für die Ausführungsintegrität wurde ein Algorithmus entwickelt, der diese auch in Anwendungen mit mehreren Threads garantieren kann. Die strukturelle Integrität stellt der sogenannte *Updatemanager* sicher. Er sorgt dafür, dass der Zustand von allen Objekten der alten Version einer Komponente in Objekte der aktualisierten Komponente übertragen wird. Anwendungsspezifische Invarianten können schließlich durch ein spezielles für DSUP entwickeltes Entwurfsmuster, den Aktualisierungskonstruktoren, sichergestellt werden.

6.2.1 Allgemeine Definitionen

Bevor Integrität und Invarianten diskutiert werden können, ist es notwendig, einige Aussagen über die Objekte in einer Anwendung und deren Relationen zueinander zu treffen. In objektorientierten Systemen sind Objekte diejenigen Entitäten, deren Gesamtheit zum einen den Zustand des Anwendungsprogrammes widerspiegeln, auf denen zum anderen auch sämtliche Operationen abgewickelt werden.

Der Zustand eines Objekts wird durch die Belegung seiner Attribute definiert, die möglichen Operationen sind durch den Typ des Objekts definiert. Neben dem dynamischen Typ, also demjenigen, der verwendet wurde, um das Objekt zu erzeugen, kann ein Objekt verschiedene Interfaces implementieren.

Definition 6.1 (Typen und Objekte) *Sei O die Menge aller Objekte und T die Menge aller Typen einer Anwendung, so existieren folgende Abbildungen:*

- ***isclasstype*** $: T \to BOOL$*, isclasstype(type)* $\Leftrightarrow$ *type* $\in T$ *ist eine Klasse*

- ***isinterface*** $: T \to BOOL$*, isinterfacetype(type)* $\Leftrightarrow$ *type* $\in T$ *ist ein Interface*

- ***basetypes*** $: T \to \mathcal{P}(T)$*, bildet von einem Typ type* $\in T$ *auf die Menge seiner Basistypen ab*

- ***typeof*** $: O \to T$*, bildet von einem Objekt* $o \in O$ *auf den dynamischen Typ* $t \in T$ *des Objektes ab*

- ***interfacesof*** $: O \to \mathcal{P}(T)$*, bildet von einem Objekt* $o \in O$ *auf die Menge der von* o *implementierter Interface-Typen ab*

Klassen und Interfaces können ihre Eigenschaften vererben:

Definition 6.2 (Ableitungen) *Ein Typ $a \in T$ kann von einem $b \in T$ ableiten, wenn entweder beide ein Klassentyp oder beide ein Interfacetyp sind. Es gilt:*

$$\lhd, \rhd : T \times T \to BOOL$$

wobei mit $a, b \in T$

$$a \lhd b \Leftrightarrow a \in basetypes(b)$$

$$a \lhd b \Leftrightarrow b \rhd a$$

Diese Definitionen bilden die Basis für das nachfolgend vorgestellte Modell.

6.2.2 Sicherstellen der Ausführungsintegrität

Die Ausführungsintegrität soll sicherstellen, dass während der Abwicklung eines Anwendungsprogramms die Daten, auf denen operiert wird, durch die Aktualisierung nicht verändert werden. Andernfalls bestünde die Gefahr, dass die Operationen in der Anwendung falsche Ergebnisse liefern und es zu einem Absturz kommt. Bevor die Anwendung aktualisiert werden kann, darf sich also der Teil der Anwendung, der aktualisiert wird, nicht in Abwicklung befinden.

Das Anwendungsprogramm wird abgewickelt, indem Operationen auf Objekten des Anwendungsprogramms ausgeführt werden, die in den Anwendungskomponenten definiert wurden. Wird eine Operation auf einem Objekt aufgerufen, so gilt:

Definition 6.3 *Eine Operation auf einem Objekt gilt solange als aktive Operation, bis diese entweder zum Aufrufer zurückgekehrt ist oder eine Ausnahme geworfen wurde, die nicht innerhalb der Operation behandelt werden konnte. Die Erzeugung und Finalisierung eines Objekts gilt jeweils als Operation.*

Der dynamische Typ eines Objekts sowie seine Basistypen können Attribute definieren. Ein Objekt kann andere Objekte durch die entsprechende Belegung dieser Attribute referenzieren. Die Erreichbarkeit von Objekten wird nun wie folgt definiert:

Definition 6.4 (Erreichbarkeit) *Ein Objekt a ist von einem Objekt b erreichbar ($b \leadsto a$), gdw.:*

- *a und b dieselben Objekte sind.*
- *Es gibt ein Objekt c, und c ist erreichbar von b, und a ist erreichbar von c (Transitivität).*
- *Es existiert ein Attribut in b, das ein Objekt c referenziert, und es gilt $c \leadsto a$.*
- *Eine aktive Operation in b hat einen Parameter, der c referenziert und es gilt $c \leadsto a$.*
- *Eine aktive Operation in b hat lokale Variablen, die c referenziert, und es gilt $c \leadsto a$.*

Attribute von Objekten können, wenn sie öffentlich sind, von außen geändert werden. In der Typdefinition des Objekts werden diese Attribute als `public` markiert. Damit ist es möglich, die Belegung der Attribute durch Operationen zu ändern, die nicht zum Objekt gehören. Im Folgenden soll die Einschränkung gelten, dass ein Objekt keine öffentlichen Attribute hat. Später kann diese Einschränkung zum Teil wieder aufgehoben werden.

Hat ein Objekt keine öffentlichen Attribute, so ist eine Änderung des Inhalts aller zum Objekt gehörenden Attribute nur über seine Operationen möglich. Für dieses Modell soll ein Objekt genau dann genau dann konsistent sein, wenn sein Inhalt nicht geändert werden kann:

Definition 6.5 (Konsistenzkriterium für Objekte) *Ein Objekt ist zum Zeitpunkt t genau dann konsistent, wenn es zum Zeitpunkt t keine aktive Operation hat.*

Allerdings sagt die Konsistenz einzelner Objekte noch nichts über die einer Komponente aus. Vielmehr müssen *alle* mit dieser Komponente in Verbindung stehenden Objekte beim Austausch konsistent sein. Das Verbindungsglied zwischen Objekten und Komponenten sind die Typen. Ein Objekt hat einen dynamischen Typ, und dieser wiederum hat ein oder mehrere Basisklassen. Außerdem kann er Interfaces zur Verfügung stellen. Um von Objekten zu Komponenten zu kommen, interessiert, welche Objekte zu einem Typ gehören:

Definition 6.6 (Objekte eines Typs) *Die Funktion* ***derivedinstances*** *liefert für einen Typ type* $\in T$ *genau diejenigen Objekte, die entweder mit dem Typ type gebildet wurden, deren dynamischer Typ eine Ableitung vom Typ type ist oder die das Interface type anbieten:*

$$derivedinstances : T \rightarrow \mathcal{P}(O)$$

mit:

$$type \in T \mapsto \left\{ o \in O \;\middle|\; \begin{array}{l} type = typeof(o) \vee \\ type \in basetypes(typeof(o)) \vee \\ type \in interfacesof(o) \end{array} \right\}$$

Auf einem Objekt werden nicht nur Operationen ausgeführt, die der dynamische Typ definiert, sondern auch jene, die den Basistypen und den Interfaces zugeordnet sind. Für den konkreten Typ `Object` als Basisklasse aller Typen liefert die Funktion *derivedinstances* alle Objekte der Anwendung ($O = derivedinstances(Object)$). Nun lässt sich das Konsistenzkriterium auf Typen erweitern:

Definition 6.7 (Konsistenzkriterium für Typen) *Ein Typ type* $\in T$ *ist zum Zeitpunkt t genau dann konsistent, wenn alle Objekte, die diesen Typ enthalten, konsistent sind:*

$$type \text{ ist konsistent} \Leftrightarrow \forall o(o \in derivedinstances(type) \Rightarrow o \text{ ist konsistent}).$$

Komponenten bestehen aus Typen, Ressourcen und den Metadaten dieser Typen. Metadaten und Ressourcen sind unveränderbare Daten und somit in jedem Fall konsistent. Wenn alle Typen einer Komponente konsistent sind, folgt, dass auch die Komponente selbst konsistent ist:

Definition 6.8 (Typen einer Komponente) *Die Funktion* ***contains*** *liefert für eine Komponente* $c \in C$ *und einen Typen type* $\in T$ *die Zugehörigkeit des Typen type zur Komponente c. C bezeichnet die Menge der Komponenten und T die Menge der Typen der Anwendung:*

$$contains : C \times T \rightarrow \{wahr, falsch\}$$

mit:

$$(c, type) \Leftrightarrow \text{Typ type ist definiert in Komponente } c, \text{ wobei } c \in C, type \in T.$$

Damit lässt sich die Funktion *derivedinstance* auf Komponenten erweitern, sodass sie alle Objekte, die zu einer Komponente gehören, findet:

Definition 6.9 (Objekte einer Komponente) *Die Funktion* ***derivedinstances*** *liefert für eine Komponente* $c \in C$ *genau diejenigen Objekte* $o \in O$*, die Objekte eines Typs sind, der in der Komponente c definiert wurde:*

$$derivedinstances : C \rightarrow \mathcal{P}(O)$$

mit:

$$c \in C \mapsto \{o \in O \,|\, type \in T \wedge o \in derivedinstances(type) \wedge contains(type, c)\}.$$

Womit sich die Konsistenzdefinition auf Komponenten ausweiten lässt:

Definition 6.10 (Konsistenzkriterium für Komponenten) *Eine Komponente $c \in C$ ist zum Zeitpunkt t genau dann konsistent, wenn alle zur Komponente gehörenden Objekte $o \in derivedinstances(c)$ zum Zeitpunkt t konsistent sind.*

Um für eine Komponente einen rekonfigurierbaren Zustand zu erreichen, müssen alle ihre Typen konsistent sein und demzufolge auch alle Objekte, die diesen Typen zugeordnet werden können. Auf diesen Objekten dürfen folglich zu diesem Zeitpunkt keine aktiven Operationen existieren. Es würde genügen, den Aufruf von Operationen auf diesen Objekten zu blockieren und zusätzlich sicherzustellen, dass aktive Operationen abgeschlossen werden. Dabei ist jedoch Folgendes zu beachten:

1. Die Blockieroperation darf zu keiner Verklemmung führen.

2. Die Zusatzkosten für die Blockieroperation sollen so gering wie möglich ausfallen.

Die erste Forderung nach Verklemmungsfreiheit lässt sich einfach an folgendem Beispiel erläutern: Zwei Objekte a und b gehören zu einer Komponente, die aktualisiert werden soll. Auf dem Objekt a wird die Operation Op_1 ausgeführt, diese wiederum ruft auf einem referenzierten Objekt b die Operation Op_2 auf. Bevor letztere tatsächlich ausgeführt werden kann, kommt es zu einer Blockierungsanforderung. Würde man nun Op_2 blockieren, könnte der Kontrollfluss nicht zu Op_1 zurückkehren, und diese Operation würde aktiv bleiben. Dies widerspräche aber dem Konsistenzkriterium, und somit könnte keine Rekonfiguration durchgeführt werden. In diesem Fall darf Op_2 erst blockieren, wenn Op_1 abgeschlossen ist.

Ein verklemmungsfreier Algorithmus, der dieses Problem löst, ist die so genannte Schreib-Lese-Sperre (*Reader-Writer-Lock*) [Ben-Ari 1982, S. 80 ff.]. Dieser Algorithmus wird im Allgemeinen dazu verwendet, um konkurrierende Schreib- und Lesezugriffe auf Ressourcen zu synchronisieren. Dabei wird zwischen *Schreib-* und *Lesesperren* unterschieden. Wenn ein Thread ausschließlich lesend auf eine Ressource zugreifen möchte, benötigt er eine Lesesperre. Eine Schreibsperre hingegen erlaubt es einem Thread, die Ressource zu verändern. Der Vorteil der Schreib-Lese-Sperre entsteht dadurch, dass lesende Zugriffe auf die Ressource solange nicht synchronisiert werden müssen, solange kein Thread eine Schreibsperre anfordert. Es können also beliebig viele Threads gleichtzeitig im Besitz einer Lesesperre sein. Ein schreibender Zugriff auf eine Resource darf jedoch nur exclusiv erfolgen. Fordert ein Thread eine Schreibsperre an, wird diese Anforderung mit allen anderen Anforderungen entsprechend synchronisiert. Nachfolgende Anforderungen von Schreib- oder Lesesperren werden dann solange blockiert, bis der Thread seine Schreibsperre zurückgegeben hat.

Übertragen auf das Problem der dynamischen Softwareaktualisierung ist die Ressource jeweils eine Komponente. Operationen auf Objekten der Komponente werden hier als unkritische Leseoperationen auf die Komponente aufgefasst, die in ihrer Nebenläufigkeit nicht einzuschränken sind. Die Aktualisierung der Komponente hingegen stellt eine Schreiboperation auf die Komponente dar - und im Prinzip ist eine Aktualisierung genau das. Sie darf ausschließlich exklusiv erfolgen.

Eine mögliche Lösung wäre offensichtlich, bei jedem Ausführen einer Operation auf einem Objekt eine Lesesperre zu akquirieren und diese beim Verlassen derselben wieder freizugeben. Die dabei entstehenden Zusatzkosten stehen jedoch in keinem Verhältnis zum Nutzen: Vor allem Operationen mit wenigen Anweisungen würden den Großteil ihrer Ausführungszeit mit dem Akquirieren und Freigeben der Lesesperre verwenden. Das kann sich in Summe signifikant auf das Verhalten der Anwendung auswirken.

Besser ist in diesem Fall ein weniger invasives Vorgehen, bei dem nur ein Teil der Operationen mit Lesesperren versehen wird. Dadurch würde dann auch die zweite Forderung nach geringen Zusatzkosten erfüllt. Konkret wird nur eine Teilmenge von Objekten für die Sperre ausgewählt:

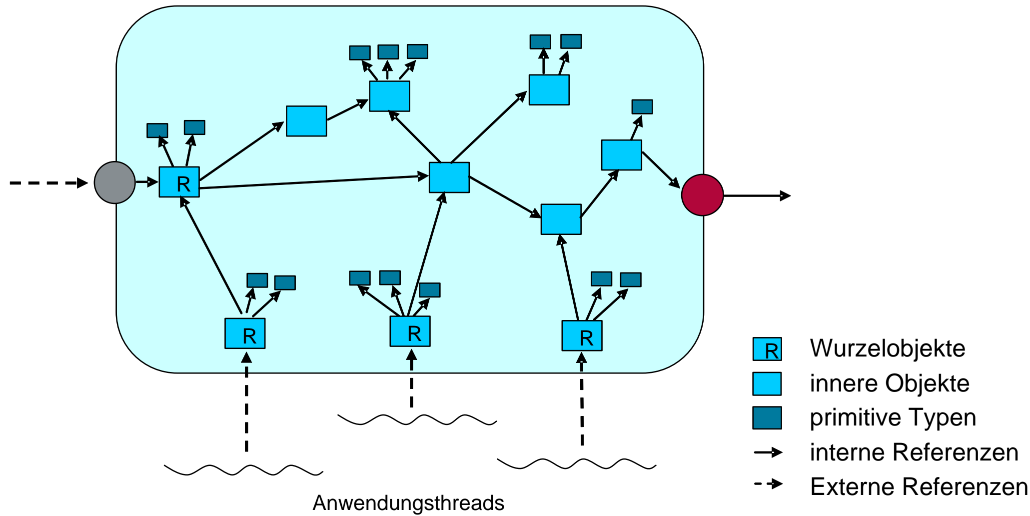


Abbildung 6.5: Wurzelobjekte einer Komponente

Definition 6.11 (Wurzelobjekte) *Sei $O_{root} \subseteq derivedinstances(c)$ eine Wurzelobjektmenge für die Komponente $c \in C$ gdw.:*

$$\forall o \in derivedinstances(c)\, \exists o_{root} \in O_{root} \Rightarrow o_{root} \rightsquigarrow o$$

Das Objekt o_{root} wird als Wurzelobjekt bezeichnet.

Eine Teilmenge der Objekte einer Komponente ist genau dann eine Wurzelobjektmenge, wenn von diesen Objekten alle anderen der Komponente erreicht werden können. Das führt zu folgender Behauptung:

Satz 6.1 *Ist jedes Objekt einer Wurzelobjektmenge der Komponente c zum Zeitpunkt t konsistent, so ist auch die Komponente c zum Zeitpunkt t konsistent.*

Der Beweis hierfür lässt sich indirekt führen: Angenommen, es ließe sich eine konsistente Wurzelobjektmenge $O_{root} \subseteq derivedinstances(c)$ finden und die Komponente $c \in C$ ist nicht konsistent, dann gäbe es nach Definition 6.10 ein Objekt $o \in derivedinstances(c)$, das nicht konsistent ist. Daraus folgt nach Definition 6.5, dass das Objekt o mindestens eine aktive Operation hat (1). Nach Definition 6.11 lässt sich nun ein $o_{root} \in O_{root}$ finden, für das $o_{root} \rightsquigarrow o$ gilt. Da o nach (1) jedoch eine aktive Operation hat, kann o_{root} nach Definition 6.5 nicht konsistent sein. Dann ist jedoch auch $O_{root} \ni o_{root}$ nicht konsistent, was ein Widerspruch zur Voraussetzung wäre $\square$.

Abbildung 6.5 veranschaulicht die Relation zwischen Wurzelobjekten und den restlichen Objekten einer Komponente. Werden die Objekte einer Komponente zusammengefasst, finden sich an den Rändern die sogenannten Wurzelobjekte wieder. Operationsaufrufe in die Komponente hinein erfolgen ausschließlich über diese Wurzelobjekte.

Eine Sperre für einzelne Objekte zu etablieren, gestaltet sich schwierig, wenn die Laufzeitumgebung nicht geändern werden soll. Einfacher ist es, die Kommunikation der Wurzelobjekte mit ihrer Umwelt genauer zu untersuchen und die hierfür verwendeten Schnittstellen mit der Sperrlogik zu verweben. Die Kommunikation kann dabei über Interfaces oder über die Klassenschnittstelle erfolgen. Letztere muss allerdings nicht notwendigerweise dem dynamischen Typ des Objekts entsprechen:

Definition 6.12 (relevanter Klassentyp) *Sei $t_1 \triangleleft t_2 \ldots \triangleleft t_n \ldots \triangleleft t_m$ eine Vererbungskette von Klassentypen mit $isclasstype(t_1) \ldots isclasstype(t_m)$, dann ist t_n der relevante Klassentyp, gdw. alle in den Typen $t_{n+1} \cdots t_m$ definierten öffentlichen Operationen ausschließlich*

durch Überschreiben von Operationen aus den Typen $t_1 \cdots t_n$ entstanden sind.

$$mrt : T \rightarrow T \text{ liefert für einen Typ den relevaten Klassentyp.}$$

Der relevante Klassentyp eines Objektes ist also derjenige Typ, der vollständig ist bezüglich der Definition seiner öffentlichen Schnittstelle. Alle Ableitungen vom relevanten Klassentyp bis zum dynamischen Typ eines Objekts erweitern den öffentlichen Teil der Klassenschnittstelle nicht, sind also für die Objektinteraktion auch nicht relevant. Werden die vom Objekt implementierten Interfaces und der relevanten Klassentyp zusammengefasst, so erhält man alle Operationen, die von einem Nutzer auf dem Objekt ausgeführt werden können.

Daraufhin lässt sich daraus die Menge derjenigen (Wurzel-)Typen ableiten, über die die Wurzelobjekte kommunizieren:

$$T_{root} = \bigcup_{o \in O_{root}} \{mrt(typeof(o))\} \cup interfacesof(o).$$

Zu Beginn dieses Abschnittes wurde die Einschränkung definiert, dass Typen keine öffentlichen Attribute haben dürfen. Diese Einschränkung galt, da sonst Änderungen am Objektinhalt vorgenommen werden können, ohne Operationen des Objekts zu verwenden. Tatsächlich gilt die Einschränkung aber nur für die Typmenge der Wurzelobjekte, da alle anderen Typen der Komponente keine Schnittstellen der Wurzelobjekte sind. Inhalte von Objekten, die diese Typen verwenden, können also weder über Operationsaufrufe noch durch einen Direktzugriff auf die Attribute verändert werden, wenn die Wurzelobjekte konsistent sind.

Daraus ergibt sich die Frage, wie eine solche Typmenge zu erhalten ist. Leider existiert hierzu kein formales Verfahren. Im Allgemeinen sollte der Programmierer aber wissen, welche Objekte einer Komponente mit der Außenwelt über welche Schnittstellen (also Typen) kommunizieren. Ist die Kopplung zwischen den Komponenten relativ lose, ist die Menge der Typen recht klein. Selbst für große Anwendungen - wie später an Beispielen dargestellt werden soll - kann diese Menge relativ schnell gefunden werden.

6.2.3 Sicherstellen der strukturellen Integrität

Über die Wurzelobjekte bzw. deren Typmenge kann für eine oder mehrere Komponenten unmittelbar vor und nach ihrer Aktualisierung ein ausführungskonsistenter Zustand garantiert werden. Bisher wurde jedoch die strukturelle Integrität noch nicht betrachtet. Sie soll sicherstellen, dass bei der Abwicklung von Operationen die Struktur der Daten, auf denen sie operieren, nicht verändert wird.

Die Wurzelobjekte werden von ihrer Umwelt über die Schnittstellen, die sie zur Verfügung stellen, referenziert. Da diese umliegenden Objekte aber nicht ausführungskonsistent sein müssen, verbietet sich eine Aktualisierung dieser Schnittstellen. Eine Operation auf einer Schnittstelle eines Wurzelobjekts könnte genau zum Zeitpunkt der Aktualisierung durch die im Abschnitt erklärte Schreibsperre blockiert worden sein. Wird diese Schnittstelle während der Aktualisierung geändert, liegt das refererenzierte Wurzelobjekt in einer neuen Version vor, und die Schnittstelle passt möglicherweise nicht mehr zum Objekt. Die Signatur der Operation könnte geändert worden sein, oder sie selbst existiert nicht mehr. In der Folge kann die Operation also nicht mehr bearbeitet werden.

In DSUP wird die strukturelle Integrität durch geringe Einschränkungen beim Entwurf des Anwendungsprogramms sichergestellt. Wurde für den ausführungskonsistenten Zustand noch verlangt, dass der relevante Klassentyp der Wurzelobjekte und die implementierten Interfaces, kurz die Menge T_{root} nicht notwendigerweise in der auszutauschenden Komponente definiert sein müssen, so ist es für die strukturelle Integrität sogar notwendig, dass alle diese Typen aus einer anderen, nicht von der Aktualisierung betroffenen Komponenten stammen müssen. Alle Typen aus T_{root} sind nicht aktualisierbar.

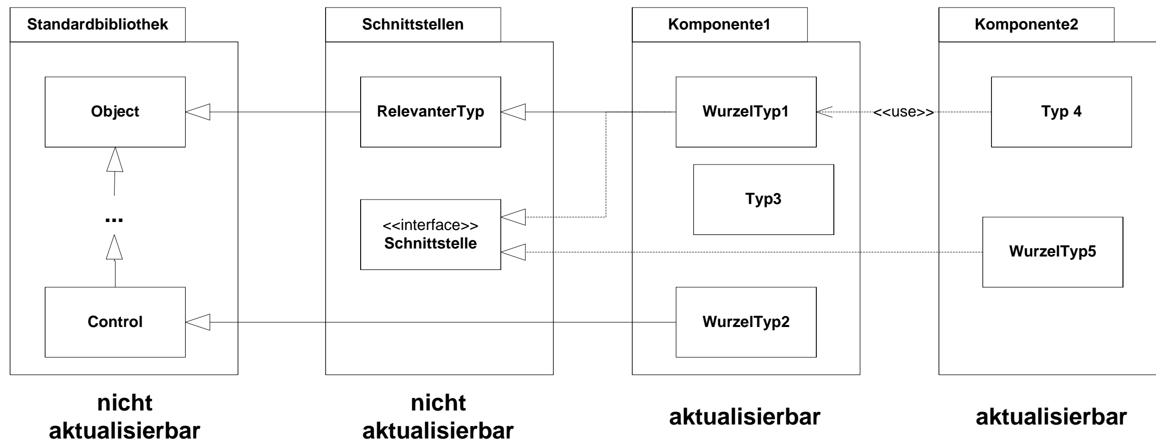


Abbildung 6.6: Bedingungen für strukturelle Integrität

Das bedeutet zwar im ersten Moment eine sehr starke Einschränkung, in der Praxis relativiert sich dieser Eindruck allerdings schnell. Das hat mehrere Gründe. Viele Typen aus T_{root} sind tatsächlich Typen, die in den Standardbibliotheken definiert sind. Komponenten, die Formulare enthalten, sind hierfür ein gutes Beispiel. Die Basisklassen `WinForm` und `Control` enthalten alle Operationen, die von den Wurzelobjekten (z. B. dem Hauptfenster der Anwendung) verwendet werden. Andererseits gilt als guter Programmierstil, Komponenten lose zu koppeln und Komponenteninteraktionen über Interfaces laufen zu lassen. Diese Interfaces können und sollten in eigene Komponenten ausgelagert werden und stellen dann kein Problem dar (siehe Abbildung 6.6).

Eine Aktualisierung wird immer auf einer Teilmenge von allen aktualisierbaren Typen ausgeführt. Diese Typmenge - nachfolgend als T_{upd} bezeichnet - enthält alle Typen der zu aktualisierenden Komponente. Auperdem können jedoch auch noch weitere Komponenten des Anwendungsprogramms exisitieren, die von dieser Komponente direkt oder indirekt abhängen. Das ist der Fall, wenn eine Komponente eine andere referenziert, weil sie deren Typen in irgendeiner Form verwendet. Diese Abhängigkeiten sind zum Zeitpunkt der Anwendungserstellung jedoch bekannt.

Definition 6.13 (aktualisierbare Typen) *Wenn c eine aktualisierbare Komponente ist, dann enthält die Menge T_{upd} alle diejenigen Typen, für die gilt:*

- *sie sind Typ der Komponente c und*
- *sie sind Typ einer Komponenten, die von c direkt oder indirekt referenziert wird.*

Für hinzugekommene abhängige Komponenten muss die Menge der Wurzelobjekte und dementsprechend die Menge T_{root} nach den bekannten Regeln vergrößert werden. Die strukturelle Integrität ist gewährleistet, wenn gilt:

$$(T_{root} \cap T_{upd} = \emptyset) \wedge (\forall\, type\, (type \in T_{upd} \Rightarrow type \text{ ist Ausführungskonsistent}))\,.$$

Strukturänderungen an Typen aus T_{upd} sind grundsätzlich erlaubt. Das wird dadurch realisiert, dass DSUP die Typen selbst nicht verändert, sondern die neuen Versionen der Typen zusätzlich zum System hinzufügt. Eine Aktualisierung kann für jeden Typ entweder durch *automatischen Zustandstransfer* oder *programmatisch* erfolgen.

Für den automatischen Zustandstransfer ist es notwendig, dass eine Abbildung von *alten* Typen auf *neue* Typen existiert. Diese Abbildung muss eineindeutig sein und wird über den Namen des Typs hergestellt. Für die Attribute des neuen Typs müssen bezüglich der zugeordneten alten Version des Typs folgende Bedingungen erfüllt sein:

- Wenn für ein Attribut im neuen Typ ein Attribut gleichen Namens im alten Typ existiert und der alte Attributtyp nicht aus T_{upd} ist, so ist der neue Attributtyp gleich dem alten.

- Wenn für ein Attribut im neuen Typ ein Attribut gleichen Namens im alten Typ existiert und der alte Attributtyp aus T_{upd} ist, so ist der neue Attributtyp der aktualisierte Typ des alten Attributtyps.

Ein neuer Typ kann bezüglich seines alten Typs neue Attribute definieren. Es ist auch nicht notwendig, dass er alle Attribute des alten Typs enthält. Werte aus entfernten Attributen werden verworfen, hinzugefügte Attribute werden mit dem Standardwert des Attributtyps initialisiert. Die Menge der Operationen im alten Typ darf sich von der Menge der Operationen im neuen Typ unterscheiden. Auch die Signatur der Operationen darf sich gegenüber dem alten Typ geändert haben, da aufgrund der Ausführungskonsistenz keine dieser Operationen aktiv ist.

Für die programmatische Aktualisierung eines Typs ist es erforderlich, dass ein entsprechender *Aktualisierungskonstruktor* definiert wird. Mit diesem kann der Programmierer selbst den Zustandstransfer eines Objekts alten Typs auf ein Objekt neuen Typs definieren. Die konkrete Implementierung des Algorithmus zur automatischen Aktualisierung der Objektstrukturen und die Verwendung der Aktualisierungskonstruktoren wird genauer im Abschnitt 6.2.5 beschrieben.

6.2.4 Umgang mit anwendungsspezifischen Invarianten

Anwendungsspezifische Invarianten sind Prädikate, die Aussagen über den Zustand einer Teilmenge des Anwendungsprogramms machen. Diese Prädikate müssen erfüllt sein, damit das Anwendungsprogramm konsistent ist. Die Menge der für das Anwendungsprogramm gültigen Prädikate lässt sich jedoch nicht aus dessen Quelltext ableiten. Es sind vielmehr Anforderungen, die im Entwurf und der Entwicklung der Software zur konkreten Struktur und Ausprägung des Quelltexts geführt haben. Nachfolgendes Beispiel soll das veranschaulichen:

Eine Komponente soll nicht näher spezifizierte Datensätze in einer Tabelle verwalten. Es soll eine Möglichkeit geben, neue Datensätze zu erzeugen und in die Tabelle aufzunehmen. Des Weiteren soll es möglich sein, Datensätze zur Weiterverarbeitung an verschiedene Teilnehmer weiterzugeben. Haben alle Teilnehmer für einen Datensatz signalisiert, dass sie ihn nicht mehr verwenden werden, so kann er aus der Tabelle gelöscht werden.

Zur Umsetzung dieser Anforderung bietet sich an, für jeden Datensatz das Verfahren der *Referenzzählung* zu implementieren. Eine Klasse `DataTable` definiert die Operationen `CreateData`, `AddRef` und `Release` (siehe Abbildung 6.7). Jeder Datensatz speichert intern einen Referenzzähler. Die Operation `CreateData` erzeugt einen neuen Datensatz und ruft die Operation `AddRef` implizit auf. Die Operation `AddRef` erhöht für den angegebenen Datensatz den Referenzzähler um Eins. `Release` schließlich subtrahiert vom Referenzzähler den Wert Eins und löscht den Datensatz aus der Tabelle, wenn der Wert Null erreicht wurde. Jedes `Release` muß ein korrespondierendes `AddRef` haben, um die Konsistenz der Datensätze zu garantieren.

Ein Prädikat der Anwendung ist, dass ein Datensatz dann nicht mehr Element der Tabelle ist, wenn die Zahl der `AddRef`-Aufrufe gleich der Zahl der `Release`-Aufrufe ist.

Angenommen, in einer fehlerhaften Implementation wurde vergessen, den Referenzzählers in `AddRef` hochzuzählen, das führt in der laufenden Anwendung dann dazu, dass Datensätze nicht freigegeben werden, da der Referenzzähler beim Aufruf von `Release` negativ wird und niemals den Wert Null erreicht. Das ist ein typisches Speicherleck, d. h. die Anwendung konsumiert immer mehr Hauptspeicher, ohne ihn wieder freizugeben. Die Funktionalität der Anwendung wird durch diesen Fehler erst einmal nicht beeinträchtigt. Je länger jedoch das

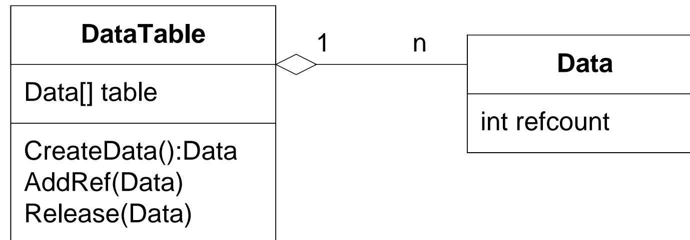


Abbildung 6.7: Beispiel für anwendungsspezifische Invarianten

System läuft, desto höher ist die Wahrscheinlichkeit, dass der Fehler für den Benutzer sichtbar wird, da neue Speicheranforderungen nicht mehr erfüllt werden können.

Eine dynamische Aktualisierung der Anwendung mit der korrekten Implementation der Operation `AddRef` kann jedoch fatale Folgen haben: Angenommen, ein Datensatz wurde mit `CreateData` angefordert. Der Referenzzähler für diesen Datensatz hat aufgrund der fehlerhaften Implementierung den Wert Null. Findet nun eine Aktualisierung der Komponente statt, gefolgt von ein Aufruf der Operation `AddRef` für den Datensatz, steht der Referenzzähler auf Eins. Das führt dazu, dass schon der erste Aufruf von `Release` den Datensatz freigibt und nicht erst der zweite. Die Folge ist, dass spätestens der zweite `Release`-Aufruf auf einen Datensatz zugreift, der nicht mehr in der Tabelle vorhanden ist.

Offensichtlich tritt dieser Fehler nur in Verbindung mit der dynamischen Softwareaktualisierung auf, da beide Versionen des Anwendungsprogrammes für sich separat betrachtet, diesen Fehler nicht enthielten. Die Änderung an der Semantik der Operation `AddRef` hat zu einer Änderung der Interpretation der Daten geführt und damit zum Fehler.

Somit hat die Anwendung allerdings vor der Aktualisierung auch das oben genannte Prädikat nicht erfüllt und war damit auch nicht konsistent im Sinne der Spezifikation und der geltenden Invarianten *nach* der Aktualisierung. *Vor* der Aktualisierung galt offensichtlich eine andere Invariante. Diese besagte, dass angeforderte Datensätze niemals freigegeben werden.

Das Wesen vieler Softwarefehler besteht jedoch darin, dass die Implementation nicht der Spezifikation entspricht. Das bedeutet, dass sich die Invarianten des Anwendungsprogramms mit der Behebung des Fehlers verändern. Um einen solchen Fehler formal korrekt zu beheben, müssen *vor* der Aktualisierung alle Invarianten gültig sein, die auch *nach* der Aktualisierung gelten.

In den meisten Fällen wird ein solcher Zustand für eine fehlerbehaftete Anwendung nicht existieren oder wenn ein solcher Zustand existiert, ist er nicht erreichbar. Im oben beschriebenen Beispiel wäre das der Fall, wenn kein Teilnehmer irgendeinen Datensatz referenziert und die Tabelle folglich leer ist. Werden permanent Datensätze bearbeitet, können die Invarianten nicht erfüllt werden, und eine Aktualisierung ist nicht möglich. Sollte trotzdem aktualisiert werden, ist es notwendig, Kompromisse einzugehen.

Ein Kompromiss wäre in diesem Beispiel, nur Datensätze, die nach der Aktualisierung angefordert wurden, entsprechend den Regeln freizugeben. Datensätze, die vor der Aktualisierung schon im System vorhanden waren, dürfen hingegen niemals freigegeben werden. mit der Unterscheidung nach alten und neuen Datensätzen lassen sich nun wieder alle Invarianten erfüllen. Die Frage ist, wo dieser Kompromiss implementiert werden soll?

[Hicks und Nettles 2005; Microsoft 2008b] lösen dieses Problem, indem zwei Varianten für die neue Version der Komponente implementiert werden: eine für die dynamische und eine für die statische Aktualisierung der Software. In der statischen Variante wird der Fehler nach Spezifikation behoben. Für die dynamische Version werden zusätzlich die Invarianten berücksichtigt. Das erzeugt jedoch in der Implementierung unnötig Redundanz.

DSUP verfolgt hier einen anderen Ansatz. Der Entwickler muss zur Erfüllung der Invarianten in die entsprechenden Typen einen *Aktualisierungskonstruktor* implementieren. Wenn ein Typ einen solchen Aktualisierungskonstruktor definiert, wird der Zustand von Exemplaren

```
 1 class DataTable
 2 {
 3   Data[] table;
 4
 5   protected DataTable(UpdateInfo info)
 6   {
 7     switch(info.Version)
 8     {
 9       case "1.0.0.30":
10         table = info.GetObject("table");
11         foreach(Data d in table)
12         {
13           d.refcount=Int32.MaxValue;
14         }
15         break;
16       case "1.1.0.35":
17         ...
18     }
19
20     ...
21   }
22 }
```

Listing 6.3: Beispiel eines Aktualisierungskonstruktors

dieses Typs von der DSUP-Laufzeitumgebung nicht automatisch übertragen, sondern erfolgt durch den Aufruf dieses Konstruktors. Der Entwickler hat so die Möglichkeit, den Zustand des Anwendungsprogramms programmatisch zu konvertieren.

Das Referenzzählerproblem kann durch die Implementation eines Aktualisierungskonstruktors in der Klasse `DataTable` gelöst werden. Für jeden in der Tabelle existierenden Datensatz wird dieser dessen Referenzzähler auf einen Wert mit dem größtmöglichen Abstand zur Null setzen (z. B. `MaxInt`). Da das genau die alten Datensätze sind, haben sie im weiteren Verlauf einen Wert, der bei spezifikationsgerechter Implementierung von `AddRef` und `Release` den Wert Null nicht erreichen kann. Alle nach der Aktualisierung erzeugten Datensätze wären davon nicht betroffen.

Der Quelltext zur Behebung eines Fehlers wird so nur einmal implementiert, und zwar in der regulären Quelltextbasis. Aktualisierungskonstruktoren hingegen werden nur dann implementiert, wenn es notwendig erscheint. Da diese ausschließlich für den Wechsel von einer Komponentenversion zu einer anderen benötigt werden, können sie in den folgenden Versionen auch wieder aus dem Quelltext entfernt werden.

Listing 6.3 zeigt einen solchen Aktualisierungskonstruktor. Das verwendete Muster entspricht dem *Memento* [Gamma u. a. 1995, 283 ff.] und gleicht dem Deserialisierungskonstruktor der Klassenbibliothek [Microsoft 2008a].

Das `UpdateInfo`-Objekt übernimmt die Rolle des Mementos. Nach [Gamma u. a. 1995] müsste das der Urheber (das alte Objekt) selbst tun, da nur das alte Objekt aufgrund der Kapselung seiner Daten seinen internen Zustand vollständig kennt. Würde an dieser Stelle genau so verfahren, gäbe es jedoch ein Problem: Die Methode zum Persistieren des Zustands im Memento muss zwangsläufig paarweise mit einer Methode zum Deserialisieren des Zustands implementiert werden. Dabei ist zu beachten, dass die Persistierung immer auf der alten und die Deserialisierung immer auf der neuen Version des Objekts ausgeführt werden. Wurde in einer früheren Version noch keine Methode zum Erzeugen eines Mementos implementiert, kann keine manuelle Aktualisierung erfolgen.

Das ist eine unnötige Einschränkung, daher wird der Zustand des alten Objekts durch den `UpdateManager` automatisch in den Memento übertragen. Der Inhalt eines jeden Attributs des Objekts ist dafür zusammen mit dem Namen des Attributes im Memento zu merken. Zugegriffen wird auf diese durch verschiedene `Get`-Methoden. Es existieren Methoden für die primitiven Datentypen (z. B. `GetInt32`) und für Referenztypen. Diese Methoden ermöglichen

es, den Wert des gleichnamigen Attributs im alten Objekt zu ermitteln. Erfolgt der Zugriff über `GetObject`, stellt der *UpdateManager* sicher, dass es sich immer um eine aktualisierte Version des abgefragten Objekts handelt.

Über das `UpdateInfo`-Objekt kann auch die Version des alten Objekts erfragt werden. Es handelt sich hierbei um die Versionsnummer derjenigen Komponente, in der der dynamische Typ des alten Objektes definiert wurde. Dieser Mechanismus erlaubt es, Aktualisierungspfade für unterschiedliche Versionen zu definieren. Das ist insbesondere dann notwendig, wenn sich die Typstruktur über verschiedene Versionen stark geändert hat und vor allem, wenn verschiedene Versionen einer Komponente ausgerollt wurden und mit einer Aktualisierung auf den neuesten Stand gebracht werden sollen.

Sollen komplexere Datenstrukturen konvertiert werden, reicht dieser Mechanismus nicht aus, da er ausschließlich erlaubt, auf Attributen der alten Version des eigenen Objekts zu operieren. Um das zu veranschaulichen, soll das Beispiel der `DataTable` erweitert werden. Es gibt eine Methode, die nach `Data`-Elementen in der Tabelle suchen kann. Hierzu werden die Elemente als Binärbaum gespeichert, um einen effizienten Zugriff zu ermöglichen. Jedes `Data`-Element erhält zusätzlich eine `left`- und `right`- Referenz, jeweils auf den Teilbaum der kleineren und der größeren `Data`-Elemente. Das Attribut `Table` ist nun nicht mehr ein Attribut von `Data`-Objekten, sondern die Wurzel des Binärbaums.

In einer neuen Version soll nun die Struktur von Binärbaum auf eine Hash-Tabelle (`Dictionary`) umgestellt werden, da die Einfügeoperation für neue Datenelemente effizienter ist. Hierzu wird die entsprechende Klasse der Klassenbibliothek verwendet. Für die `Data`-Klasse bedeutet das, dass der `left` und `right`-Zeiger überflüssig werden, denn die Elemente werden direkt im `Dictionary` abgespeichert.

Die Listing 6.4 zeigt die Änderungen im Quelltext und den zugehörigen Aktualisierungskonstruktor im neuen Objekt. Die `UpdateInfo`-Klasse bietet eine Methode `GetNode`, die zum manuellen Traversieren des Objektbaums verwendet wird. Das ist in der Methode `LookupChildren` implementiert. In die neue Struktur werden die Elemente durch den Aufruf von `InsertObject` aufgenommen.

6.2.5 Implementationsdetails

DSUP ist eine auf LOOM.NET aufbauende Klassenbibliothek, deren Programierschnittstelle ermöglicht, dynamisch aktualisierbare Anwendungsprogramme zu schreiben. Die Bibliothek besteht im Wesentlichen aus:

- einem Aspekt, mit dem die Typen der Wurzelobjekte annotiert werden können und
- einem *Updatemanager*, der geänderte Komponenten zur Laufzeit erkennt und die Aktualisierung vornimmt.

Wie bereits erwähnt, ist es Aufgabe des Programmierers, diejenigen Typen im Anwendungsprogramm zu identifizieren, über die die Wurzelobjekte mit der Außenwelt kommunizieren. Jeder in der Typmenge T_{root} enthaltener Typ wird mit dem Aspekt `RootObject` annotiert und in der Folge entsprechend verwoben. Die eingewobene Aspektlogik realisiert hierbei drei Aufgaben:

- Synchronisieren der Objekterzeugung und Anmelden des Objekts beim *Updatemanager*,
- Synchronisieren der Methodenaufrufe auf dem Objekt,
- Austausch der Objektreferenz nach einer Aktualisierung.

Abbildung 6.8 zeigt eine vereinfachte Darstellung der Architektur von DSUP und deren Funktionsweise. Die zwei Typen `WurzelTyp1` und `WurzelTyp2` im Anwendungsprogramm

alte Version

```
class DataTable
{
  Data root;

    ...
}
```

```
class Data
{
  Data left, right;
  ...
}
```

neue Version

```
class DataTable
{
  Dictionary<Data,Object> table;

  // Der Aktualisierungskonstruktor
  protected DataTable(UpdateInfo info)
  {
    table=new Dictionary<Data,Object>();

    InsertObject(
      info.GetObject<Data>("root"));
    LookupChildren(info.GetNode("root"));
  }

  // Traversieren des alten Binärbaums
  private void LookupChildren(UpdateInfo node)
  {
    if(node!=null)
    {
      InsertObject(
        info.GetObject<Data>("left"));
      LookupChildren(info.GetNode("left"));

      InsertObject(
        info.GetObject<Data>("right"));
      LookupChildren(info.GetNode("right"));
    }
  }

  // Aufnehmen der Daten in die neue Struktur
  private void InsertObject(Data data)
  {
    if(data!=null)
    {
      table.Add(data, null);
    }
  }

    ...
  }
}
```

```
class Data
{
  ...
}
```

Listing 6.4: Aktualisierung komplexer Strukturen mit einem Aktualisierungskonstruktor

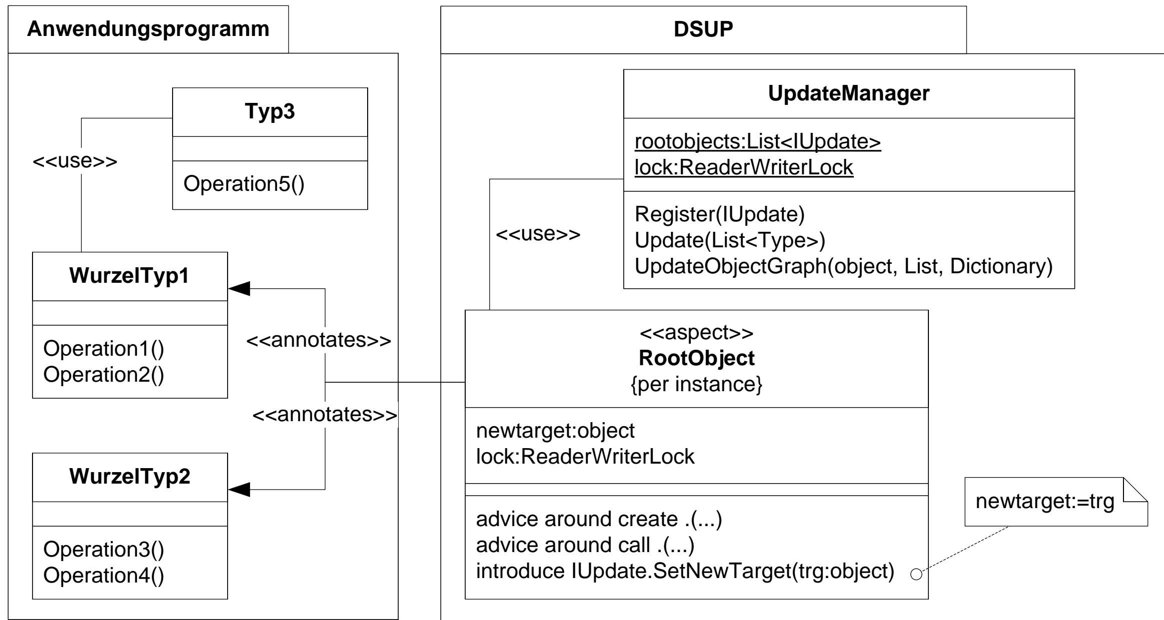


Abbildung 6.8: Vereinfachte Architektur von DSUP

wurden vom Entwickler jeweils als Typ eines Wurzelobjekts identifiziert. Dementsprechend sind sie mit dem Aspekt `RootObjekt` annotiert und verwoben. Der Aspekt kontrolliert für alle Objekte dieser Typen die Erzeugung, Benutzung und Zerstörung.

Listing 6.5 zeigt den Pseudocode der Advices vom `RootObject`-Aspekt: Bei der Objekterzeugung, verwoben mit **create** Advice, wird zuerst aus dem `UpdateManager` eine Referenz auf die globale Schreib-Lese-Sperre (`lock`) geholt (Zeile 3). Auf dieser wird anschließend eine Lesesperre angefordert, denn es wäre möglich, dass zum Zeitpunkt der Objekterzeugung eine Aktualisierung durchgeführt wird. Konnte die Sperre akquiriert werden, so gibt der Aspekt den Kontrollfluss an den ursprünglichen Konstruktor ab und erzeugt so das Objekt (Zeile 5). Das erzeugte Objekt wird anschließend beim `Updatemanager` registriert (Zeile 6). Dieser merkt sich die registrierten Objekte in der Liste `rootobjects`. Schließlich kann die Lesesperre wieder entfernt werden, und das erzeugte Objekt wird an den Aufrufer zurückgegeben.

Methodenaufrufe auf einem (Wurzel-)Objekt werden durch den Aspekt mit **call**-Advice abgefangen. Zuerst wird auch hier eine Lesesperre angefordert (Zeile 13-14). Ist das im Aspekt abgespeicherte Attribut `newtarget` noch nicht belegt, dann war dieses Objekt noch nicht von einer Aktualisierung betroffen. In diesem Fall delegiert der Aspekt den Aufruf an das originale Objekt weiter (Zeile 16). Wurde dieses Objekt jedoch zwischenzeitlich aktualisiert, dann ist in `newtarget` eine Referenz auf eine Version des Objekts gespeichert. In diesem Fall wird der Aufruf durch `InvokeOn` auf das neue Objekt umgeleitet, und das Wurzelobjekt agiert nur als Stellvertreter (Zeile 18). Kehrt der Methodenruf zum Aspekt zurück, kann die Lesesperre wieder entfernt werden. Hatte der Methodenaufruf einen Rückgabewert, so wird dieser an den Aufrufer zurückgegeben (Zeile 19-21).

Der `UpdateManager` hat die Aufgabe, neue Versionen von Komponenten zu erkennen. Hierzu überwacht er Änderungen im Dateisystem. Unter .NET wird eine Assembly jedoch sofort gesperrt, sobald sie in das Anwendungsprogramm geladen wurde. Das Überspielen einer neuen Version ist daher nicht möglich, solange die Anwendung nicht beendet wurde. Die Laufzeitumgebung lässt es jedoch zu, das Laden einer Assembly manuell vorzunehmen. Das ist ausführlich in [Rasche 2008, S. 64 f.] dargestellt. Durch das manuelle Laden kann DSUP verhindern, dass Assembly-Dateien für Änderungen gesperrt sind. Wird eine Assembly geändert, wird automatisch die `Update`-Methode im `Updatemanager` ausgeführt.

```
1 advice around create .(...)
2 begin
3   lock:=UpdateManager.lock
4   lock.AquireReadLock()
5   trg:=joinpoint.Call(argv)
6   UpdateManager.Register(trg)
7   lock.ReleaseReadLock()
8   return trg
9 end
10
11 advice around call .(...)
12 begin
13   lock:=UpdateManager.lock
14   lock.AquireReadLock()
15   if newtarget=null then
16     res:=joinpoint.Call(argv)
17   else
18     res:=joinpoint.CallOn(newtarget, argv)
19   lock.ReleaseReadLock()
20   return res
21 end
```

Listing 6.5: Pseudocode der `RootObject`-Advices

Abbildung 6.9 zeigt den der `Update`-Methode zugrundeliegenden Algorithmus. Die `Update`-Methode aquiriert zuerst eine Schreibsperre und stellt so sicher, dass auf den registrierten Wurzelobjekten keine aktiven Operationen ausgeführt werden, diese also ausführungskonsistent sind. Anschließend wird für jedes Wurzelobjekt durch den Aufruf der Methode `UpdateObjectGraph` eine Aktualisierung des Objektgraphen ausgelöst. Die Methode akzeptiert drei Parameter, das Objekt selbst, eine Liste derjenigen Typen, für die eine neue Version vorliegt (T_{upd}), und eine anfangs leere Menge von Schlüssel- Wertpaaren ($\mathbb{O}_{visited}$), in der die bereits besuchten Objekte und deren neue Version abgespeichert werden.

Kehrt die Methode mit einem anderen Objekt als dem übergebenen Wurzelobjekt zurück, so wurde das Wurzelobjekt aktualisiert. In diesem Fall ist das Wurzelobjekt ungültig und der `RootObject`-Aspekt wird durch den Aufruf der Introduktion `SetNewTarget` davon in Kenntnis gesetzt. Außerdem werden alle Attribute mit Wertetypen auf `null` gesetzt. Das hat zur Folge, dass das Wurzelobjekt nur noch als leere Hülle existiert. Der Aspekt sorgt dafür, dass alle folgenden Aufrufe an die neue Version des Objekts weitergeleitet werden. So ist sichergestellt, dass alle Objekte, die Wurzelobjekte von außerhalb referenzieren, nach der Aktualisierung auf dem richtigen Objekt operieren.

In der Methode `UpdateObjectGraph` werden alle Objekte aktualisiert. Das geschieht durch Traversieren des durch die Wurzelobjekte aufgespannten Objektbaums. Da der Baum Zyklen enthalten kann, wird zuerst in ($\mathbb{O}_{visited}$) überprüft, ob das Objekt schon aktualisiert wurde. Ist das der Fall, zeigt die dort gespeicherte Referenz auf das aktualisierte Objekt, und die Methode kehrt zum Aufrufer zurück. Andernfalls muss überprüft werden, ob das Objekt aktualisiert werden muss. Wenn der dynamische Typ in der Menge T_{upd} enthalten ist, muss ein neues, uninitialisiertes Objekt vom aktualisierten Typ erzeugt werden (`CreateNew`). Das ursprüngliche und das resultierende Objekt werden in die Liste der aktualisierten Objekte als Tupel aufgenommen und so - als bereits besucht - markiert.

Handelt es sich um ein neu generiertes Objekt, hat es möglicherweise einen Aktualisierungskonstruktor. Dieser wird dann aufgerufen und ist für Zustandstransfer verantwortlich. Die Methode `CreateUpdateInfo` erzeugt hierzu ein `UpdateInfo`-Objekt, das den Zugriff auf den Zustand des alten Objekts ermöglicht.

Ist kein Aktualisierungskonstruktor vorhanden, werden automatisch alle Attribute des resultierenden Objekts untersucht und auch deren Inhalt rekursiv durch erneuten Aufruf der Methode `UpdateObjectGraph` aktualisiert. Dies geschieht durch die Methode `fieldsof`, die hier stellvertretend für Aufrufe der entsprechenden Reflektionsklassen steht.

procedure UPDATE($T_{upd} \subseteq T$)
  $\mathbb{O}_{visited} \leftarrow \emptyset$
  $lock$.AquireWriteLock()
  for all $o_{root} \in O_{root}$ **do**
    $o_{trg} \leftarrow$UpdateObjectGraph(o_{root}, $\mathbb{O}_{visited}$, $T_{updated}$)
    if $o_{trg} \neq o_{root}$ **then**
      o_{root}.SetNewTarget(o_{trg})
    end if
  end for
  $lock$.ReleaseWriteLock()
end procedure

procedure UPDATEOBJECTGRAPH($o \in O$, $\mathbb{O}_{visited} \subseteq O \times O$, $T_{upd} \subseteq T$)
  if $(o, o_{trg}) \in \mathbb{O}_{visited}$ **then**
    return o_{trg}
  end if
  if $typeof(o) \in T_{upd}$ **then**
    $o_{trg} \leftarrow$CreateNew(o)
  else
    $o_{trg} \leftarrow o$
  end if
  $\mathbb{O}_{visited} \leftarrow \mathbb{O}_{visited} \cup \{(o, o_{trg})\}$
  if $o_{trg} \neq o \wedge \{o_{trg}.ctor_{upd}\} \in methodsof(typeof(o_{trg}))$ **then**
    $o_{trg}.ctor_{upd}(CreateUpdateInfo(\mathbb{O}_{visited}, T_{upd}))$
  else
    for all $f \in fieldsof(o_{trg})$ **do**
      $o_{trg}.f \leftarrow$UpdateObjectGraph($o_{trg}.f$, $\mathbb{O}_{visited}$, $T_{updated}$)
    end for
  end if
  return o_{trg}
end procedure

Abbildung 6.9: Aktualisierung der Objekte

Die Rekursion terminiert, da die Menge der Objekte, die von den Wurzelobjekten aufgespannt werden, endlich ist. Zyklische Aktualisierungen von sich gegenseitig referenzierenden Objekten werden durch die Einträge in ($\mathbb{O}_{visited}$) vermieden. Wenn alle aufgespannten Objekte einen Eintrag in ($\mathbb{O}_{visited}$) haben, ist die Aktualisierung abgeschlossen, und die Schreibsperre wird wieder abgegeben.

Da die Attribute der Wurzelobjekte während der Aktualisierung auf `null` gesetzt wurden, existiert für alle alten Objekte, die nicht selbst Wurzelobjekt sind, nach der Aktualisierung keine Referenz mehr aus einem aktiven Teil des Anwendungsprogramms. Damit kann die automatische Speicherverwaltung der Laufzeitumgebung diese Objekte freigeben.

6.2.6 Evaluation der Lösung

Um die Anwendbarkeit der vorgestellten Lösung auf reale Anwendungsprogramme zu demonstrieren, wurden zwei größere externe .NET-Projekte, *PaintDotNet* [Brewster 2008] und der *Lumisoft Mail-Server* [Lumi und Lumi 2008], herangezogen. Beide Projekte sind unabhängig vom DSUP-Projekt, und keines der Projekte hatte dynamische Softwareaktualisierung als Anforderung für die Entwicklung. Von beiden Projekten steht der komplette C#-Quelltext zur Verfügung.

PaintDotNet ist ein quelloffenes Bildbearbeitungsprogramm für den semiprofessionellen Bereich. Verglichen mit dem im Windows-Betriebssystem mitgelieferten *Paint* bietet es einen viel größeren Funktionsumfang. Dementsprechend komplex ist auch die zugrundeliegende Quelltextbasis. Das Programm hat ca. 133.000 Zeilen.

Der Lumisoft Mail-Server ist eine Serveranwendung zur Bereitstellung von elektronischen Postfächern. Er implementiert hierzu das *IMAP*-Protokoll [Crispin 2003]. Über dieses Protokoll kann der Benutzer auf dem Server unter anderem auf E-Mails zugreifen und diese manipulieren. Des Weiteren ist es möglich, Postfächer anzulegen, zu löschen und umzubenennen. Auch für den Lumisoft Mail-Server stehen die gesamten Quellen frei zur Verfügung.

Zuerst wurde untersucht, ob sich die Voraussetzungen für die Anwendung des Konsistenzmodells von DSUP bei beiden Anwendungen erfüllen lassen. Dabei wurden entsprechende Wurzeltypen identifiziert. Diese Typen enthielten keine öffentlichen Attribute, sodass auch diese Voraussetzung erfüllt werden konnte. Die Wurzeltypen wurden anschließend mit dem `RootType`-Aspekt annotiert. Das war die einzige Änderung, die an den Quelltexten vorgenommen wurde. Anschließend wurden die Programme neu kompiliert und mit GRIPPER-LOOM.NET verwoben.

Die beiden kompilierten Anwendungen wurden auf einem System ausgeführt und das korrekte Verhalten bei einer Aktualisierung untersucht. Hierzu wurde im Vorfeld jeder Anwendung ein Softwarefehler eingebaut, der jeweils bei der Aktualisierung behoben werden konnte. In PaintDotNet führte der Softwarefehler dazu, dass die Position des Mauszeigers um einige Punkte verschoben angezeigt wurde. Bei Lumisoft Mail-Server wurde die Funktion zum Löschen von Mailboxen durch den Softwarefehler außer Kraft gesetzt.

Zur Aktualisierung der Anwendungen wurden die Softwarefehler behoben und die neu kompilierten Anwendungen in das laufende System eingespielt. Wie zu erwarten, waren die Fehler nach der Aktualisierung nicht mehr zu reproduzieren. Während der Aktualisierung wurden verschiedene Kenngrößen ermittelt. Eine Kenngröße war die Totzeit, also die Zeit in der Teile der Anwendung durch eine Schreibsperre gesperrt sind. Während dieser Zeit kann die Anwendung nur beschränkt Eingaben verarbeiten und Ausgaben produzieren. Weiterhin wurde die Zahl der bei der Aktualisierung traversierten Objekte gezählt und dann erfasst, welche Objekte davon tatsächlich aktualisiert werden mussten.

Abbildung 6.10 zeigt die Ergebnisse. Die aktualisierte Komponente des Lumi-Mailservers hatte im Gegensatz zur PaintDotNet-Komponente zum Zeitpunkt der Aktualisierung weniger Objekte im System, sodass die Totzeit hier im Vergleich relativ gering ausfiel. Das manifestiert sich in der Zahl der aktualisierten Objekte. Trotz allem ist eine Totzeit von fast vier

	PaintDotNet	Lumisoft Mail-Server
Quelltextzeilen	ca. 133.000	ca. 61.000
Totzeit bei Aktualisierung	$3,52 \pm 0,2\,s$	$213 \pm 7,5\,ms$
Traversierte Objekte	728.000	1326
Aktualisierte Objekte	200	3

Abbildung 6.10: Ergebnisse bei ausgesuchten Anwendungen

Sekunden für PaintDotNet durchaus akzeptabel, wenn eine Aktualisierung auf bisherigem Weg zum Vergleich heangezogen wird. Hier müsste der Anwender zuerst alle offenen Dokumente speichern. Dann müsste er die Anwendung beenden, die Komponenten aktualisieren, anschließend die Anwendung neu starten und schließlich alle seine zuvor gespeicherten Dokumente wieder laden.

6.2.7 Diskussion der Lösung

DSUP hat gezeigt, dass es möglich ist, eine komplexe Aufgabe, wie die dynamische Aktualisierung von laufenden Anwendungen, durch die Verwendung von LOOM.NET mit einfachen Mitteln gelöst werden kann. Es ist nicht notwendig, eine Änderung der Laufzeitumgebung vorzunehmen; DSUP lässt sich auf üblichen CLI-Ausführungsumgebungen verwenden. Die Änderungen am Anwendungsprogramm selbst sind gering und belaufen sich ausschließlich auf die Identifikation und Annotation der Wurzeltypen. Die herausragenden Vorteile von DSUP sind die einfache Verwendung, die geringen Laufzeitkosten und die Unterstützung von Anwendungen mit mehreren nebenläufigen Threads.

An zwei völlig verschiedenen real existierenden Anwendungen wurde die flexible Einsetzbarkeit von DSUP gezeigt. Mit dem Einsatz von LOOM.NET konnte die gesamte Synchronisationslogik, die in den Wurzelobjekten benötigt wird, durch einfache Annotation hinzugefügt werden. Daher müssen nur wenige Zeilen in der originalen Anwendung geändert werden. Das DSUP-Konsistenzmodell und das Programmiermodell der Aktualisierungskonstruktoren sorgen dafür, dass die Aktualisierung zur Laufzeit relativ robust gegenüber Programmierfehlern ist.

Es gibt aber auch zahlreiche ähnlich gelagerte Arbeiten, die hier kurz diskutiert werden sollen. *Trap/J* benutzt einen vergleichbaren Ansatz wie DSUP. [Sadjadi u. a. 2004] verwenden AspectJ, um für jede Komponente einen Stellvertreter zu erzeugen. Dieser (*Wrapper-Level*) leitet alle Methodenaufrufe an ein *Meta-Level* weiter. Das *Meta-Level* sind sogenannte *Meta-Level-Objects*, die anfänglich den ursprünglichen Komponentenimplementierungen initialisiert sind. Alternative Implementierungen können jedoch zur Laufzeit in ein *Delegate-Level* nachgeladen werden. Die Methodenaufrufe können dann durch Aktualisierung der *Meta-Level-Objects* auf die neue Implemetation aktualisiert werden.

Um eine Anwendung für eine dynamische Aktualisierung vorzubereiten, ist ein spezieller Kompilationsschritt für die Erzeugung des *Wrapper-Level* notwendig. Zur Laufzeit erfolgen die Aufrufe ins *Meta-Level* über die Java-Reflektionsschnittstellen, was einen großen Nachteil gegenüber DSUP darstellt, da das erhebliche zusätzliche Laufzeitkosten bedeutet [vgl. Forax u. a. 2005]. DSUP verwendet zwar auch die teuren Reflektionsschnittstellen, allerdings ausschließlich während der Rekonfiguration zur Übertragung des Zustands, während das bei Trap/J bei jedem Komponentenaufruf geschieht.

Ein weiterer Nachteil gegenüber DSUP ergibt sich daraus, dass alternative Implementierungen explizit in der separaten *Delegate-Ebene* erfolgen müssen. Das hat zur Folge, dass der Softwareentwicklungsprozess komplexer wird. Änderungen müssen in der *Delegate-Ebene* erfolgen, um eine dynamische Aktualisierung zu ermöglichen, und sie müssen zusätzlich in der Komponente erfolgen.

Das Microsoft Windows-Betriebssystem hat spezielle Mechanismen, um bestimmte Betriebssystemkomponenten zur Laufzeit auszutauschen. In [Microsoft 2008b] werden Eintrittspunkte der Komponenten durch eine bestimmte Sequenz von Maschinencodeanweisung präpariert. Die Sequenz dient hier als Platzhalter für eine mögliche spätere Aktualisierung. Eingefügt wird sie bei der Übersetzung der Komponente durch einen speziellen Schalter. Im Falle einer Aktualisierung wird die Sequenz durch Maschinenbefehle ersetzt, die folglich auf die neue Version der Komponente referenzieren. Dieses Verfahren ist ausschließlich dem Betriebssystem vorbehalten und lässt sich nicht auf Anwendungssoftware übertragen. Insbesondere ist das bei Software, die unter einer virtuellen Maschine läuft, nicht möglich, da hier direkt der Maschinencode geändert wird.

[Hicks und Nettles 2005] diskutieren verschiedene Verfahren, um Anwendungen, die in der Programmiersprache C geschrieben, sind zur Laufzeit zu aktualisieren. Sie schlagen unter anderem vor, die Anwendung so zu instrumentieren, dass alle Methodenaufrufe nicht direkt, sondern über eine *Global Offset Table* (GOT) ausgeführt werden. Infolge einer Aktualisierung wird die GOT auf die neue Implementierung angepasst. Zusätzlich wird eine spezielle Methode benötigt, um den Zustandstransfer zu ermöglichen. Evaluiert wird dieses Verfahren mit *FlashEd*, einem dynamischen Webserver. In der vorgestellten Lösung ist es jedoch Aufgabe des Programmierers, den richtigen Aktualisierungszeitpunkt zu finden. Im Falle von *FlashEd* ist das jedoch einfach, da es sich um eine ereignisgetriebene Anwendung mit einem Thread handelt. Die Aktualisierung erfolgt immer vor der Abarbeitung der Nachrichtenschleife.

Andere Ansätze erweitern existierende Programmiersprachen oder verwenden eine spezielle virtuelle Maschine. *Adaptive Java* von [Kasten u. a. 2002] führt beispielsweise neue Schlüsselwörter in die Java-Programmiersprache ein, um rekonfigurierbares Verhalten zu modellieren. QuO von [Vanegas u. a. 1998] ist eine verteilte Middleware für die Entwicklung verteilter Anwendungen. Der Entwickler kann hier alternative Implementierungen definieren, die in Abhängigkeit von der Umgebung ersetzt werden können.

6.3 Weitere Projekte mit LOOM.Net

In diesem Abschnitt werden andere Projekte vorgestellt, die mit LOOM.NET realisiert wurden. Sie sollen einen Überblick über weitere Einsatzmöglichkeiten des Konzeptes geben.

6.3.1 Adapt.NET

In seiner Dissertation [Rasche 2008] stellt Andreas Rasche, ein langjähriger Kollege des Autors, einen ganzheitlichen Ansatz für die Entwicklung und Ausführung adaptiver komponentenbasierter Anwendungen vor. Im Vordergrund steht hierbei die Anpassung von Anwendungen an wechselnde Umgebungsparameter und Ressourcenverfügbarkeiten. Die von ihm entwickelte Ausführungsplattform *Adapt.NET* ermöglicht es, zur Laufzeit auf alternative Anwendungskonfigurationen umzuschalten, um sich der Umgebung anzupassen.

Damit eine Anwendung (re-)konfigurierbar gestaltet werden kann, schlägt Rasche eine Ausdehnung der Komponentenabstraktion auf die Anwendungslaufzeit vor, bei der sogenannte *Kapseln* über *Konnektoren* interagieren. Dadurch wird eine (Re-)konfiguration der laufenden Anwendung wird möglich, weil die Ausführungsplattform Adapt.NET jederzeit Kontrolle über die Kapseln und deren Interaktionen hat.

Ein zentraler Bestandteil der zugrundeliegenden Rekonfigurationstechniken verwendet LOOM.NET-Aspekte zur effizienten Entwicklung adaptiver Anwendungen. Über die Aspekte wird unter anderem die für die Rekonfiguration notwendige Synchronisationslogik in die Konnektoren der Kapseln eingewoben. Zusätzlich erlauben es die Aspekte, Konfigurationsschnittstellen in die Kapseln einzubringen. Das ermöglicht Konfiguration, Zustandstransfer

```
 1 [WMIManagedAspect("NAMESPACE", "CLASSNAME")]
 2 public class Sample
 3 {
 4   private int iMode=0;
 5   ...
 6   public int iMode
 7   {
 8     [WMIExposedAspect]
 9     get
10     {
11       return this.iMode;
12     }
13     [WMIExposedAspect]
14     set
15     {
16       this.iMode=value;
17     }
18     ...
19     [WMIExposedAspect]
20     public void Terminate()
21     {
22       Console.WriteLine("Sample::Terminate␣called");
23     }
24 }
```

Listing 6.6: Eine WMI-verwaltete Klasse [nach Olejniczak 2007, S. 53]

und Migration der Kapseln. Insgesamt stellt das in [Rasche 2008] vorgestellte Verfahren einen interessanten und sehr komplexen Anwendungsfall für LOOM.NET dar.

6.3.2 Ein Managementframework für die Windows Fernwartungsschnittstelle

In einer vom Autor co-betreuten Diplomarbeit beschäftigt sich Michael Olejniczak damit, wie Software an Fernwartungsschnittstellen des Betriebssystems angeschlossen werden können. Die Möglichkeit, Software im laufenden Betrieb zu warten und zu überwachen, ist besonders bei Anwendungen im Bereich der Hochverfügbarkeit unablässig. Aber auch in anderen Bereichen stellt das einen kostenreduzierender Faktor dar, wenn der Zustand eines Anwendungsprogramms vom Administrator abgefragt und verändert werden kann. Fehlerdiagnose und -behebung sowie Wartung der Software kann so erfolgen, ohne dass der physische Zugang zur ausführenden Maschine erforderlich ist.

Im konkreten Fall geht es hier um die Anbindung an die Windows Fernwartungsschnittstelle (WMI). WMI ist die Windows-Implementierung des WBEM-Standards der [Distributed Management Task Force DMTF Homepage], um verteilte IT-Umgebungen zu verwalten. WMI ermöglicht es unter anderem, auf Ressourcen von Anwendungsprogrammen zuzugreifen und diese zu verändern. Diese Möglichkeiten ließen sich unter Microsoft.NET 2.0 jedoch nur relativ eingeschränkt nutzen. In der Klassenbibliothek vorhandene Funktionen erlauben nur einen lesenden Zugriff auf Ressourcen, ein schreibender Zugriff war nicht vorgesehen.

[Olejniczak 2007] schlägt ein vollkommen neues Programmiermodell vor. Anstatt Ressourcen explizit über die WMI-Schnittstellen anzulegen und zu verwalten, werden in seinem Modell die Ressourcen direkt auf Eigenschaften und Methoden der Klassen des Anwendungsprogramms abgebildet. Elemente, die für WMI sichtbar werden sollen, werden mit LOOM-Aspekten entsprechend annotiert. Die Aspektlogik übernimmt die Kommunikation mit der WMI-Schnittstelle.

Das in Listing 6.6 dargestellte Beispiel ist eine derartige, durch WMI verwaltete Klasse. Für den Programmierer ist die komplexe WMI-Schnittstelle transparent. Er sieht in seinem Quelltext, welche Teile seiner deklarierten Klasse durch die Fernwartung abgefragt und verändert werden können.

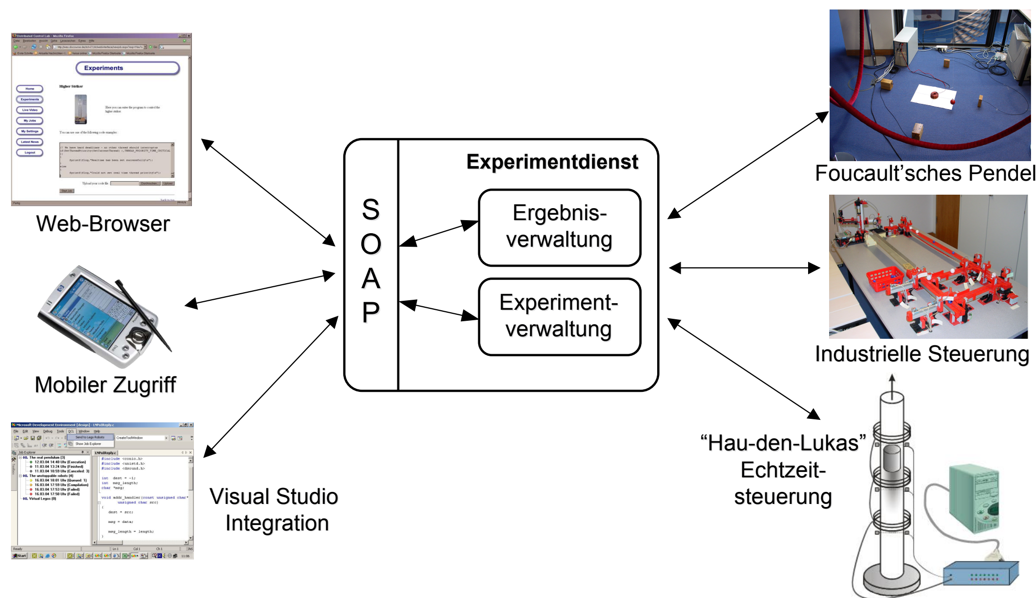


Abbildung 6.11: Architektur des DCL

6.3.3 Das Distributed Control Lab

Das Distributed Control Lab (DCL) ist eine verteilte Laborinfrastruktur, die seit 2002 am Hasso-Plattner-Institut entwickelt und in Lehre und Forschung eingesetzt wird. Es besteht aus verschiedenen Experimenten, die über das Internet gesteuert werden können. Hierzu überträgt der Benutzer ein Programm zur Steuerung eines real existierenden Prozesses. Das Programm verarbeitet die Sensorsignale des Prozesses und steuert die Aktoren. Als Experimente existieren zum Beispiel das Focault'sche Pendel, Hau den Lukas und eine Fischertechnik-Fertigungsstraße.

Eine Herausforderung des DCL ist es, fehlerhafte Steuerungsprogramme zu erkennen und zu entfernen, um Schaden an den Experimenten zu vermeiden. Für bestimmte Experimente ist es auch nötig, spezielle Steuerprogramme auszuführen, die den Urzustand herstellen, bevor ein neues Programm gestartet werden kann.

Abbildung 6.11 zeigt ein grobes Architekturbild des DCL. Kern ist der Experimentdienst, der die Verwaltung der Steuerprogramme entgegennimmt, sie zeitlich dem entsprechenden Experiment zuweist und die Ausführung der Programme überwacht. Anschließend können die Ergebnisse dort wieder abgeholt werden.

Der Benutzer hat verschiedene Möglichkeiten, mit dem Experimentierdienst zu kommunizieren. Üblicherweise erfolgt der Zugriff über eine Webseite aus dem Internet. Die Experimente können aber auch über ein mobiles Gerät gesteuert werden. Über eine *Visual-Studio-*Integration [Microsoft Corporation 2008a] lassen sich die Steuerprogramme aber auch direkt in den Experimentierdienst einspielen. Die Kommunikation mit dem Experimentierdienst ist als Webserviceschnittstelle ausgelegt.

Eine große Herausforderung bei der Implementation des Experimentierdiensts war es, die Vielzahl an Serveranfragen bewältigen zu können. Für die ursprüngliche Lösung war das insbesondere ein Problem, wenn viele unterschiedliche Informationen gleichzeitig abgefragt werden mussten. Das führte nachfolgend zu einer großen Zahl an Webserviceaufrufen. Eine Anfrage, um die Liste aller Experimente zu erhalten, benötigte beispielsweise je nach Auslastung des Servers zwanzig Sekunden und mehr.

Eine Analyse des Problems ergab, dass bei komplexen Anfragen bestimmte Webserviceaufrufe mehrfach redundant ausgeführt wurden. Diese Aufrufe sind unnötig und belasten das Netzwerk. Um das Problem zu lösen, könnte man aufseiten des Klienten ein Zwischenspei-

cher implementiert werden, der redundante Aufrufe erkennt und in diesem Fall das bekannte, zuvor abgespeicherte Ergebnis zurückliefert. Dieses *Caching* müsste mit objektorientierten Methoden als überschneidende Belange in jeder einzelnen Webservicemethode implementiert werden. Daher wurde ein entsprechender Aspekt implementiert und unter Verwendung von Gripper-Loom.Net in das System eingewoben.

Als Ergebnis könnte die Bearbeitungsdauer für die Abfrage der Experimentliste auf unter drei Sekunden reduziert werden. Der Implementationsaufwand war gering, da der Memoized-Aspekt bereits Teil der in Loom.Net enthaltenen Beispiele ist. Zusätzlich zu diesem Aspekt wurden an verschiedenen Stellen außerdem das aspektorientierte *Einzelstückmuster* (Abschnitt 4.2) und ein *Logging*-Aspekt verwendet.

6.3.4 Ein Optimierungsframework für Komponenteninteraktionen

Extesible optimisation framework for .NET virtual machine [Puzovic 2005] ist eine Arbeit, die am Imperial College in London realisiert wurde. Dieses Projekt wurde unabhängig vom Autor durchgeführt.

Eine These von [Puzovic 2005] lautet, dass während der Laufzeit einer komponentenorientierten Anwendung ein zusätzlicher Aufwand entsteht, der aus dem Zusammenspiel der Komponenten resultiert. [Puzovic 2005] stellt ein Framework zur Verfügung, das dieses Problem lösen soll.

Ein zentraler Bestandteil dieses Frameworks ist ein Loom.Net-Aspekt, mit dem Aufrufe in Komponenten abgefangen und analysiert werden können. Als Ergebnis der Analyse nimmt das Framework Optimierungen in der Interaktion zwischen den Komponenten vor. Das kann zum Beispiel das Zusammenfassen mehrerer Fernaufrufe zu einem einzigen Aufruf sein.

6.4 Zusammenfassung

Im Rahmen dieses Kapitels wurden verschiedene Anwendungsszenarien von Loom vorgestellt und diskutiert. Hierbei konnte verdeutlicht werden, dass mit den Loom-Konzepten und den Aspektwebern Rapier-Loom.Net und Gripper-Loom.Net eine große Bandbreite von Programmierproblemen effizient gelöst werden können.

In einer Studie im industriellen Umfeld konnte sich Loom.Net an einem realen Anwendungsfall - einem Frontend-System der Filialen der Deutschen Post - beweisen. Hier wurde gezeigt, wie überschneidende Belange aus dem bestehenden Quelltext extrahiert und mit Loom modularisiert werden können. Dabei ergab sich als konkretes Ergebnis, dass der Aufwand für den Softwaretest verringert werden konnte.

Mit dem DSUP-Projekt wurde ein ganzheitlicher Ansatz für die Enwicklung und Ausführung dynamisch aktualisierbarer Anwendungsprogramme realisiert. Es wurde ein Modell vorgestellt, mit dem es möglich ist, Komponenten der Anwendung zur Laufzeit auszutauschen, ohne die Integrität der Anwendung zu verletzen. Mit DSUP wurde eine Klassenbibliothek geschaffen, die es erlaubt, solche dynamisch aktualisierbaren Anwendungen zu schreiben. Ein besonderes Merkmal dieser Lösung ist es, dass ein neu entwickelter Algorithmus die Aktualisierung von Anwendungen mit mehreren parallel laufenden Anwendungsthreads erlaubt. Der Einsatz von Loom.Net erlaubt ein minimal-invasives Vorgehen, da der Quelltext der Anwendungsprogramme nur an wenigen Stellen angepasst werden muss und keine besondere Ausführungsumgebung benötigt.

7 Verwandte Arbeiten

In diesem Kapitel soll eine Abgrenzung der vorliegenden Arbeit zu anderen Forschungs- und Entwicklungsprojekten erfolgen. Aufgrund der Menge der verschiedenen Lösungen werden hier nur diejenigen mit der größten Nähe zur Arbeit zu LOOM betrachtet. Allein im Bereich der Aspektorientierten Programmierung sind im [AOSD Wiki 2007] 31 verschiedene Lösungen für die unterschiedlichsten Programmiersprachen aufgeführt. Die englische Wikipedia zählt im Juni 2008 auf der Seite zur Aspektorientierten Programmierung [Wikipedia 2008a] 83 Implementierungen auf, davon allein 13 für die .NET-Plattform und 24 für Java.

Ferner sollen auch ähnliche Ansätze diskutiert werden, mit denen überschneidende Belange vermieden werden können, die selbst aber nicht zur Aspektorientierten Programmierung gehören.

In Kapitel 2 wurde mit LOOM.UML auch eine Erweiterung der UML2 zur Modellierung von Aspekten vorgestellt. Hier sollen nun Lösungen vorgestellt werden, die ähnliche Ansätze verfolgen.

7.1 Ansätze zur Trennung von Belangen

In diesem Abschnitt werden Lösungen vorgestellt, mit denen eine Trennung von Belangen realisiert werden kann, die sich aber konzeptionell von AOP und [Kiczales u. a. 1997] unterscheiden.

7.1.1 Subjektorientierte Programmierung und HyperJ

Die *Subjektorientierte Programmierung* [Harrison und Ossher 1993; Ossher u. a. 1994] wurde vor der Aspektorientierten Programmierung mit ähnlichen Zielen entwickelt. Sie versteht sich als eine Erweiterung der Objektorientierten Programmierung und erlaubt eine verteilte Klassendefinition. Ein Entwickler, der neue Operationen zu Klassen hinzufügen möchte, muss den existierenden Quelltext nicht bearbeiten, er kann statt dessen die Klasse durch ein neues *Thema* erweitern. Die verschiedenen Themen werden schließlich so zusammengefügt, dass sie die Anforderungen der Anwendung erfüllen.

Überlappen sich Themen, die zu einer Klasse gehören, da beispielsweise Methoden in beiden Themen definiert wurden, so gibt es verschiedene Möglichkeiten, diesen Konflikt zu lösen. Entweder werden beide Methoden ausgeführt, wobei es verschiedene Möglichkeiten gibt, wie mit dem Rückgabewert umgegangen werden soll. Oder die Methode eines Themas überschreibt die Methode des anderen. Schließlich gibt es noch die Möglichkeit, dass solche Überlappungen als illegal behandelt werden.

Eine Weiterentwicklung der Subjektorientierung stellt *HyperJ* mit dem Ansatz der mehrdimensionalen Trennung von Belangen dar [Tarr u. a. 1999]. Die Trennung der Belange erfolgt durch *Hypermodule* und *Hyperslices*. Dabei werden im Gegensatz zur Subjektorientierten Programmierung nicht nur die Klassen selbst, sondern auch Komponenten und Methoden als separierbare Einheiten betrachtet.

Ein *Hyperslices* ist eine Menge konventioneller Module und fasst alles das zusammen, was einen einzelnen Belang betrifft. In einem Hyperslice können einzelne Methoden, Klassen

und ganze Pakete vertreten sein. Auch hier können Einheiten mehrfach in unterschiedlichen Hyperslices auftreten.

Ein *Hypermodule* ist eine Menge von Hyperslices zusammen mit einer Kompositionsregel, die bestimmt, wie die Hyperslices zusammengefügt werden sollen. Das Ergebnis ist wiederum ein Hyperslice. Hypermodule können daher auch geschachtelt angewendet werden. Die Komposition der Hyperslices bildet schließlich das gesamte System.

Der Ansatz der Hyperslices hat große Ähnlichkeiten zu den im späteren Abschnitt besprochenen *Mixins*. Die Komposition ist hier jedoch nicht nur auf eine Menge von Methoden beschränkt, sie schließt auch andere Einheiten, d.h. Methoden und Namensräume mit ein. In dieser Hinsicht ähnelt die Vorgehensweise dem von LOOM, da auch hier die Ebenen der Komposition Methoden, Klassen oder ganze Pakete darstellen.

Im Gegensatz zu HyperJ wird die Komposition in LOOM über die Aspekte und deren Annotation beschrieben und benötigt kein zusätzliches Sprachmittel, wie die Hyperslices und Hypermodule. Des Weiteren hat der Programmierer mit LOOM durch den Aufrufkontext zur Laufzeit die Kontrolle über die Abarbeitung von verwobenen Methodenaufrufen. In HyperJ entspräche dies dem Zusammenführen gleichnamiger Methoden aus unterschiedlichen Hyperslices, wobei die Abarbeitung schon zum Kompilierungszeitpunkt explizit festgelegt werden muss.

7.1.2 Das Composition-Filter-Modell

Das *Composition-Filter-Modell* (CF-Modell) [Aksit und Bergmans 2001; Aksit und Tekinerdogan 1998a,b] geht zurück auf die *Sina* Programmiersprache [Aksit und Tripathi 1988] und stellt eine modulare Erweiterung zum konventionellen Objektmodell dar. Die Interaktion zwischen Objekten definiert das CF-Modell als Senden und Empfangen von Nachrichten. Die Idee ist, durch definierte *Filter* die Nachrichten zu verändern, um so das Verhalten der Objekte anzupassen.

Um eine solche Manipulation zu erreichen, wird eine Schicht (das *Interface*) über ein sogenanntes *Implementations*-Objekt gelegt. Jedes Objekt im System kann ein solches Implementationsobjekt sein. Das Composition-Filter Modell und seine Elemente sind in Abbildung 7.1 dargestellt. Die *Filter* definieren das (beobachtbare) Verhalten des Objekts. Jeder einzelne Filter kann dazu Nachrichten inspizieren und manipulieren. Dabei können *Eingabefilter* und *Ausgabefilter* Nachrichten, die vom Objekt empfangen und gesendet werden, verändern. Die Filter werden jeweils zu geordneten *Filtermengen* zusammengefasst. Filter können interne und externe Objekte referenzieren (*Referenzen*).

Das Verhalten des Objekts ergibt sich aus dem Zusammenspiel von Implementation (die das eigentliche Objekt darstellt) und internen und externen Objekten, definiert durch die Filter. Die Implementation kann zwei Typen von Methoden enthalten: *reguläre* und *konditionale*. Die regulären Methoden implementieren die funktionalen Eigenschaften des Objekts. Diese werden durch die Nachrichten aufgerufen, (wenn die Filter das erlauben). Konditionale Methoden geben einen Wahrheitswert (**true** oder **false**) zurück, der Informationen über den Zustand des Objekts widerspiegelt. Diese Methoden müssen jedoch frei von Seiteneffekten sein.

Jede Nachricht, die an ein Implementationsobjekt gesendet wird, muss jeden Filter passieren und kann dabei entweder verworfen oder *dispatched* (bearbeitet) werden. Bearbeitet meint, dass die Nachricht entweder auf einer lokalen Methode aufgerufen oder zu einem anderen Objekt delegiert wird. Jeder Filter kann Nachrichten entweder annehmen oder abweisen. Die Semantik für die Akzeptanz oder das Abweisen hängt von dem Typ des Filters ab. Vordefinierte Filter sind zum Beispiel:

- *Dispatch*: Wenn die Nachricht akzeptiert wurde, wird sie zum aktuellen Ziel der Nachricht weitergeleitet. Ansonsten wird sie zum nächsten Filter durchgereicht. Sollte kein

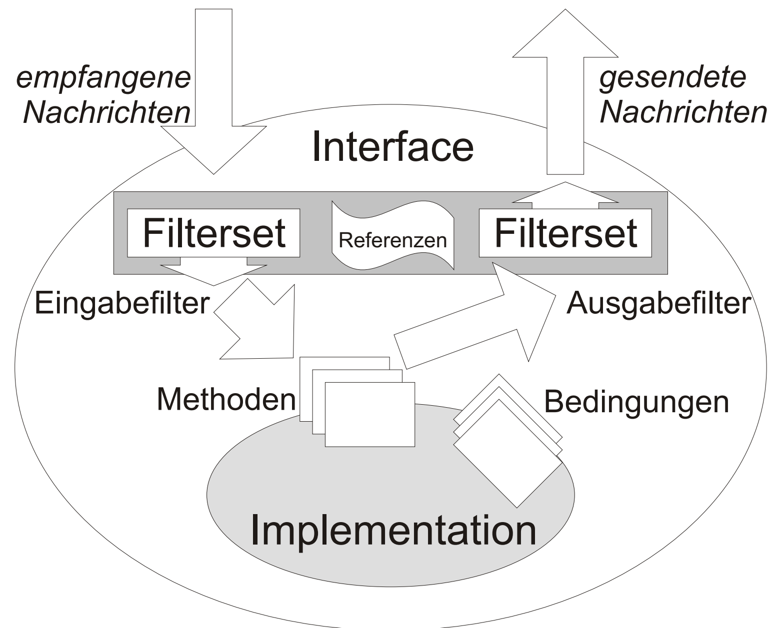


Abbildung 7.1: Vereinfachte Struktur der Composition Filter Objekte [nach Aksit und Tekinerdogan 1998a]

weiterer Filter mehr existieren, wird eine Ausnahme (*Exception*) erzeugt.

- *Error*: Wenn der Filter die Nachricht abweist, wird eine Ausnahme erzeugt. Ansonsten wird die Nachricht zum nächsten Filter in der Menge weitergeleitet.
- *Wait*: Wenn die Nachricht akzeptiert wurde, wird sie zum nächsten Filter weitergeleitet. Ansonsten wird sie solange in einer Warteschlange gehalten, wie der Filter sie ablehnt.
- *Meta*: Akzeptiert der Filter die Nachricht, so wird sie als Parameter verpackt zu einem benannten Objekt (als Metanachricht) geschickt, ansonsten wird mit der Bearbeitung im nächsten Filter fortgefahren. Das Objekt, das die Metanachricht erhält, kann die originale Nachricht untersuchen und Parameter verändern. Danach kann es die veränderte Nachricht zur Ausführung reaktivieren.

Ein Filter besteht immer aus einer Anzahl von *Filterelementen*, wovon jedes die Form hat:

$$\langle Bedingung \rangle \Rightarrow \langle Ausdruck \rangle.$$

Listing 7.1 zeigt eine komplette Filterdefinition. Dieser Filter wird in Abhängigkeit der gesetzten Sicherheitsstufe Nachrichten zu Objekten vom Typ `GUIElement` zulassen oder verweigern. Das erste Filterelement würde, wenn die Bedingung `IsSetAllowed` wahr und die empfangene Nachricht entweder `setX` oder `setY` ist, diese Nachricht akzeptieren und zum nächsten Filter weiterreichen. Sollte die Bedingung falsch sein, so würde das nächste Filterelement evaluiert. Sollte die Nachricht von keinem der Filterelemente akzeptiert werden - der Filter weist die Nachricht ab - würde in diesem Fall der `Error`-Filter eine Ausnahme erzeugen.

Eine Implementierung des CF-Modells unter .NET ist *Compose*.NET*, die die Möglichkeit eröffnet, Filter auf .NET-Assemblies zu definieren. Diese Lösung ist damit prinzipiell auf alle .NET-Sprachen anwendbar. Gegenüber LOOM wird beim CF-Modell ein anderes Konzept verfolgt. Obwohl bei LOOM Nachrichten an Objekte und deren Manipulation (realisiert durch

```
concern GUIElement
  filterinterface GUIWithViews begin
    internals
      calculator:Calc;
    externals { no externals defined by this class }
    conditions
    methods
    inputfilters
      Security: Error={ IsSetAllowed => {setX,setY},
                        IsSizeAllowed => {resize}
                        };
  end filterinterface
```

Listing 7.1: Eine Filterdefinition

die Kontextobjekte) eine zentrale Rolle spielen, steht hier die Komposition von Modulen im Vordergrund. Alles, was durch die Filter im CF-Modell realisierbar ist, kann in LOOM mit Hilfe der Advices und der Kontextobjekte auch abgebildet werden. Im CF-Modell fehlt jedoch eine Entsprechung zu den Introduktionen.

7.1.3 Mixins und Traits

Die Begriffe *Mixin* und *Trait* sind leider nicht eindeutig definiert und wurden im Verlauf der Programmiersprachenentwicklung für immer neue, wenngleich aber ähnliche Konzepte verwendet. Daher herrscht eine gewisse Konfusion über die tatsächliche Bedeutung, diese erschließt sich meist erst im Zusammenhang mit der konkreten Implementation und Programmiersprache. Die Intention von Mixin und Traits ist jedoch grundsätzlich dieselbe. Es geht darum, eine bestimmte Funktionalität in einer modularen Einheit zusammenzufassen, um diese an unterschiedlichen Stellen wiederverwenden zu können.

Mixin hat seinen eigentlichen Ursprung in *Flavors* [Moon 1986], das später in das *Common Lisp Object System* (CLOS) mündete. In Flavours und CLOS ist ein Mixin eine Klasse, deren einzige Aufgabe darin besteht, über Mehrfachvererbung mit anderen Klassen kombiniert zu werden.

Nach der Definition von [Bracha und Cook 1990] spezifiziert ein *Mixin* eine Menge von Modifikationen in der Basisklasse, um eine neue abgeleitete Klasse zu erzeugen. Ein Mixin unterscheidet sich von einer abgeleiteten Klasse dahingehend, dass es von der Identität der Basisklasse abstrahieren kann. Als Operationen für die Komposition stehen `merge`, `override`, `copy·as` und `restrict` zur Verfügung.

[Myers 1996] bezeichnet als *Traits* ein Entwurfsmuster, mit dem Konfigurationsbelange einer Klasse separat in eine eigenen Klasse ausgelagert werden können. Er realisiert das mit C++ *Templates.* Als Beispiel demonstriert er einen Stream-Puffer, der für verschiedene Basistypen konfigurierbar ist. Die Konfigurationsklasse enthält dann sowohl den zu verwendenden Basistyp als auch die Datei-Ende-Markierung (EOF).

[Nierstrasz u. a. 2005; Schärli u. a. 2003] redefinieren den Begriff *Traits* für ein Konzept, das Brachas *Mixins* sehr ähnelt. *Traits* sind eine Gruppe von Methoden, die als Bausteine für Klassen verwendet werden und somit einfache Einheiten der Wiederverwendung darstellen. Konzeptuell sind sie zwischen Klassen und Methoden angesiedelt. Ein *Trait* besteht aus Methodendefinitionen und *benötigten Methoden. Traits* enthalten keine eigenen Attribute, zudem dürfen die Methodendefinitionen nicht auf Attribute der Klassen zugreifen, in denen sie verwendet werden. Der Zugriff muss daher indirekt über die *benötigten Methoden* erfolgen. Zusätzlich lässt sich auch bei *Traits* festlegen, wie mit Namenskonflikten umzugehen ist. Hierzu stehen die Operationen `sum`, `alias` und `exclude` zur Verfügung.

Eine andere *Traits*-Implementierung ist die von [Odersky und Zenger 2005] für die *Scala*-Programmiersprache. Sie unterstützt zwar typisierte *Traits*, aber es werden keine Operationen

Traits	**LOOM.NET**

<pre>public class MyCircle
{
  uses
  {
    Color; Circle;
  }
  ...
}</pre>
	<pre>[Color, Circle]
	public class MyCircle
	{
	  ...
	}</pre>

Listing 7.2: Vergleich zwischen Traits und Aspekten

unterstützt, um Konflikte aufzulösen. Damit entspricht diese Lösung eher den *Mixins* von [Moon 1986].

Introduktionen lösen exakt das Problem wie *Mixins* und *Traits*. Der Unterschied hier ist jedoch die Sichtweise bei der Auflösung von Konflikten. Bei *Mixins* und *Traits* hat derjenige, der die Komposition ausführt, die Verantwortung, den Konflikt zu lösen. Bei LOOM ist es hingegen der Aspektprogrammierer, indem er die Introduktion beispielsweise mit dem optionalen Modifizierer **override** versieht. Der Aspektprogrammierer ist damit in der Lage, seine Implementation an mögliche Konflikte bei der Komposition schon im Vorfeld anzupassen.

In Listing 7.2 ist ein direkter Vergleich zu einer Komposition einer Klasse (`MyCircle`) mit *Traits* für C# [nach Reichhart 2005, S 6.] und LOOMAspekten zu sehen. Während die Module `Color` und `Circle` über das Schlüsselwort `uses` hinzugefügt werden, erfolgt das bei der LOOM-Variante durch Annotation.

7.1.4 Erweiterungsmethoden

In der Version 3.0 der Programmiersprache C# gibt es das neue Konzept der *Erweiterungsmethoden* [Microsoft Corporation 2005b, §26.2], die es erlauben, Methoden zu Klassen hinzuzufügen. Dabei werden die Klassen selbst nicht geändert und müssen weder neu kompiliert oder muss eine Ableitung von ihnen erstellt werden. Erweiterungsmethoden sind eine spezielle Art statischer Methoden, können jedoch aufgerufen werden, als wären sie Methoden des Objekts.

Erweiterungsmethoden können nicht auf Klassenattribute und Methoden zugreifen, die nicht öffentlich sind. Somit wird die Modularisierung nicht aufgebrochen. Listing 7.3 zeigt eine beispielhafte Implementierung einer Erweiterungsmethode. Der obere Teil ist die Definition , während im unteren die Benutzung dargestellt ist. Der Namensraum, in dem sich eine Erweiterungsmethode befindet, ist nicht identisch mit dem der Klasse. Um eine Erweiterungsmethode zu nutzen, muss also zusätzlich der Namensraum der Erweiterungsmethode (hier `ExtensionMethods`) eingebunden werden.

Erweiterungsmethoden eignen sich weniger, um überschneidende Belage zu modellieren. Sie können jedoch gut verwendet werden, um Klassen zu erweitern, die nur in binärer Form zur Verfügung stehen. Hier gleichen sie einem LOOM-Aspekt mit Introduktionen, die auch in binäre Komponenten gewoben werden können.

7.2 Aspektorientierte Programmierung

Aspektorientierte Programmierung ist ein Oberbegriff für eine Vielzahl von Konzepten und Werkzeugen zur Softwareentwicklung. Der Begriff wurde maßgeblich von *AspectJ* und seinen Urhebern [Kiczales u. a. 1997] geprägt. Die Konzepte von AspectJ sind die Grundlage vieler

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this
            String str)
        {
            return str.Split(new char[] {
                ' ', '.', '?' },
                StringSplitOptions.
                RemoveEmptyEntries).Length;
        }
    }
}
```

```
using ExtensionMethods;

...

string s = "Hallo␣das␣ist␣eine␣
    Erweiterungsmethode";
int i = s.WordCount();
```

Listing 7.3: Erweiterungsmethoden in C#

aspektorierter Lösungen. Diejenigen mit der größten Überschneidung zu LOOM sollen in diesem Abschnitt vorgestellt werden sollen.

Abgrenzend zu allen in diesem Abschnitt vorgestellten Lösungen lässt sich für LOOM feststellen, dass nur in LOOM eine explizite Schnittstelle für beide Parteien - den Nutzer und den Anbieter eines Aspektes - besteht. Die zugrundeliegenden Konzepte der Annotation und Überdeckung sind ein Alleinstellungsmerkmal von LOOM.

7.2.1 AspectJ

AspectJ wurde von 1996-2002 von einer Gruppe um Georg J Kiczales am Xerox Palo Alto Research Center entwickelt [Kiczales u. a. 2001, 1997]. Seit Ende 2002 ist AspectJ Teil des Eclipse- Projektes [The Eclipse Foundation 2008a]. AspectJ enthält als Kommandozeilen Werkzeuge, einen Compiler *ajc*, einen Debugger *ajdb*, ein Werkzeug zum Erstellen von Dokumentationen *ajdoc* und als grafisches Werkzeug den *AspectJ Browser*.

Die Programmiersprache *AspectJ* ist eine Erweiterung von Java und voll kompatibel zum aktuellen Java-Sprachstandard. Auffällig ist die große Zahl neuer Schlüsselwörter, die mit AspectJ eingeführt wird. Diese verdoppelt sich fast im Vergleich zu Standard-Java. Der Grund ist die Vielzahl an Beschreibungsmöglichkeiten von Joinpoints in dieser Spracherweiterung. AspectJ erlaubt es, an nahezu jeden beliebigen Punkt im Basisprogramm zusätzlichen Quelltext einzuweben.

Konkret gibt es in AspectJ folgende Schlüsselwörter, um Joinpoints zu beschreiben:

- einen Aufruf oder eine Abwicklung einer bestimmten Methode (**call**, **execution**),
- den Zugriff auf Felder (**set**, **get**),
- eine Exemplarbildung (**initialization**, **preinitialization**),
- eine statische Initialisierung von Klassen **staticinitialization**),
- eine Ausnahmebehandlung (**handler**),
- den Aspektcode (**adviceexecution**),
- die Abwicklung von Quelltext im Kontext einer Klasse, eines Konstruktors oder einer Methode (**within**, **withincode**),
- das aktuell abgewickelte Objekt, das gerufene Objekt oder die Argumente eines bestimmten Typs (**this** , **target**, **args**),

- die Abwicklung von Quelltext, die aus einem bestimmten Kontrollfluss heraus erfolgt (**cflow**, **cflowbelow**).

Mit der Einführung der Annotationen gibt es für diese Schlüsselwörter weiterhin eine spezielle **@...**-Variante, um annotierte Joinpoints zu beschreiben. **@within** meint beispielsweise jeden Joinpoint, der in einer Klasse existiert, die mit der angegebenen Annotation markiert wurde.

Die Kombination dieser Joinpointbeschreibung bildet einen Pointcut. AspectJ kennt *primitive* Pointcuts und *Pointcutdefinitionen.* Die primitiven Pointcuts werden direkt mit einem Advice verwendet. Pointcutdefinitionen hingegen zeichnen sich durch einen Identifizierer aus und können vererbt und erweitert werden. Außerdem ist es möglich, von verschiedenen Advices auf eine Pointcutdefinition zu referenzieren.

Das Konzept der *Inter-Type-Deklarationen* ähnelt dem der Introduktionen, geht aber darüber hinaus. So ist es mit einer Inter-Type-Deklaration nicht nur möglich, neue Interfaces in eine Klasse einzuführen, sondern es kann die gesamte Vererbungshierarchie mit den Schlüsselwörtern **declare parents** verändert werden. Weiterhin ist es möglich, von einem Aspekt aus Methoden in Klassen einzuführen. Das entspricht im Wesentlichen dem Konzept der Erweiterungsmethoden aus Abschnitt 7.1.4, wobei es hier auch möglich ist, statische Methoden zu definieren. Konflikte bei der Einführung bereits vorhandener Methoden führen in AspectJ zu einem Fehler. LOOM bietet hier die Möglichkeit, Introduktionen so zu definieren, dass sie in diesem Fall wie ein Advice behandelt werden. Der Aspektprogrammierer kann so die einzuführende Schnittstelle unabhängig vom Programmierer der Basisklasse definieren.

Mit Inter-Type-Deklarationen lassen sich auch statische und nicht-statische Attribute zu Klassen hinzufügen. Dieses Konzept ähnelt den LOOM Joinpoint-Variablen. Auch Joinpoint-Variablen führen zu neuen Attributen in einer verwobenen Klasse. Das ist jedoch nur ein kleiner Teil des Konzepts der Joinpoint-Variablen, da diese vordergründig dazu dienen, den Zustand zwischen verwobener Klasse und deren Aspekten zu teilen. So können Attribute in LOOM auch virtuell eingeführt werden, um sie an bereits vorhandene gleichnamige Attributdefinitionen in der zu verwebenden Klasse zu binden. Weiterhin lassen sich Joinpoint-lokale Attribute definieren.

Die Exemplarbildung der Aspekte lässt sich in AspectJ durch sogenannte **per...**-Schlüsselwörter steuern. Die Möglichkeiten sind hier ähnlich wie bei LOOM: pro Exemplar der verwobenen Klasse, pro verwobene Klasse und als Einzelstück. Es ist in AspectJ auch möglich, die Exemplarbildung pro Methodenaufruf zu definieren. Da in LOOM Aspekte Module annotieren, existiert hier die Variante, pro Annotation ein Exemplar des Aspekts anzulegen.

Durch die Annotation können in LOOM den Aspekten im Gegensatz zu AspectJ parametriert werden. Das erfolgt über die Konstruktoren der LOOM-Aspekte und ist für jede Annotation separat möglich. Viele der vorgestellten Beispiele machen von dieser Möglichkeit Gebrauch.

Ab der im Jahr 2006 veröffentlichten AspectJ-Version 1.5 gibt es die *@AspectJ*-Erweiterung. Diese Erweiterung ermöglicht es, Aspekte ausschließlich mit Java-Anotationen zu definieren, wie das bei LOOM auch geschieht. Die Aspekte können mit einem Java 5 Kompiler übersetzt werden, müssen aber trotzdem anschließend mit einem AspectJ-Aspektweber verwoben werden.

In Abbildung 7.2 ist ein Vergleich wesentlicher Programmierkonzepte von AspectJ und LOOM dargestellt. Wenngleich die Joinpoint-Beschreibungssprache von AspectJ viel mehr Möglichkeiten bietet als LOOM, entstehen durch deren Mächtigkeit jedoch eine Vielzahl von Problemen, die in Kapitel 1 bereits ausführlich dargestellt wurden. Insbesondere das Einweben von Quelltext in private Teile fremden Quelltexts wird als Zerstörung des Modulkonzeptes angesehen [Constantinides u. a. 2004; Gabriel u. a. 2006; Steimann 2006; Tourwé u. a. 2003].

Zwar muss der Aspektprogrammierer mit dem Modifizierer **privileged** bei der Aspektdefinition explizit angeben, dass er auch Quelltext verweben möchte, der als geschützt oder

<table>
<tr><th></th><th>@AspectJ</th><th>LOOM.NET</th></tr>
<tr><td rowspan="2">Aspektdefinition und Exemplarbildung</td><td>@Aspect("pertarget")
public class MyAspect
{ ... }</td><td>[CreateAspect(Per.Class)]
public class MyAspect:AspectAttribute
{ ... }</td></tr>
<tr><td colspan="2">Definiert einen Aspekt, von dem zur Laufzeit genau ein Exemplar pro verwobenem Typ existiert.</td></tr>
<tr><td rowspan="2">1:n Advice</td><td>@Before("execution(*␣*.Target.*(..))")
public void Foo()
{ ... }</td><td>[Call(Advice.Before),IncludeAll]
public void Foo(params object[] args)
{ ... }</td></tr>
<tr><td colspan="2">Verwebt jede Methode der Klasse <code>Target</code> (AspectJ) oder des annotierten Moduls (LOOM) mit dem Advice <code>foo</code> ein.</td></tr>
<tr><td rowspan="2">1:1 Advice und Zugriff auf den Joinpoint</td><td>@around("execution(void␣org.Target.Foo(int))
    ␣&&␣args(i)")
public Object Foo(ProceedingJoinPoint jp,
    int i) {
return jp.proceed(new Object[]{i*2}); }</td><td>[Call(Advice.Before)]
public void Foo([JPContext] Context jp, int
    i) {
return jp.Call(i*2); }</td></tr>
<tr><td colspan="2">Ersetzt die Methode <code>Foo</code> der Klasse <code>Target</code> (AspectJ) oder des annotierten Ziels durch den Advice. Die Advice-Implementierung ruft die ersetzte Methode mit dem veränderten Parameter <code>i</code> auf.</td></tr>
<tr><td rowspan="2">Einführen neuer Interfaces und Attribute</td><td>@Aspect
public class MyAspect {
  public static class FooImpl implements
      IFoo {
    private Bar bar;
    public Bar Foo()
    {
      return bar;
    }
  }

  @DeclareParents(value="org.Target*",
      defaultImpl=FooImpl.class)
  private IFoo implementedInterface;
}</td><td>public class MyAspect:AspectAttribute
{
  [Introduce(typeof(IFoo))]
  public Foo([JPVariable] Bar bar)
  {
    return bar;
  }
}</td></tr>
<tr><td colspan="2">Fügt in die Klasse <code>Target</code> (AspectJ) oder dem annotierte Modul (LOOM) eine neue Variable <code>bar</code> und das Interface <code>IFoo</code> mit der Methode <code>Foo</code> ein.</td></tr>
<tr><td rowspan="2">Generieren von Verwebungsfehlern</td><td>@DeclareError("execution(*␣IFoo.*(..))␣&&␣!
    within(org.foo..*)")
static final String fehlermeldung = "
    Unerlaubte␣IFoo␣Implementation";</td><td>[Error("Unerlaubte␣IFoo␣Implementation")]
[Include(typeof(IFoo)),Exclude(typeof(Foo))]
public foo() {}</td></tr>
<tr><td colspan="2">Erzeugt beim Verweben die angegebene Fehlermeldung, wenn das Interface <code>IFoo</code> von anderen Klassen als der <code>Foo</code> Klasse implementiert wird. In LOOM muss zusätzlich die zu prüfende Assembly mit dem Aspekt annotiert werden.</td></tr>
</table>

Abbildung 7.2: Vergleich @AspectJ und LOOM.NET

privat deklariert wurde. Wenn es diese Möglichkeit gibt, wird er das auch tun. Doch auch nicht privilegierte Aspekte weben letztendlich Quelltext in private Teile des Quelltextes ein. Auch wenn diese Aspekte nur öffentliche Methoden von Klassen verändern dürfen, können einzelne Anweisungen der Methodenimplementation verwoben werden. Das betrifft zum Beispiel alle **call**-Pointcuts.

[Griswold u. a. 2006] begegnen diesem Problem mit dem Konzept der *überschneidenden Interfaces* (*crosscutting Interfaces*, kurz: XPI). Ein XPI ist eine zusätzliche Schnittstellendefinition, in der der Programmierer des Basisprogramms explizit festhält, welche Joinpoints durch einen Aspekt verwoben werden dürfen. Sie stellen somit einen Vertrag zwischen dem Programmierer des Basisprogramms und dem Aspektprogrammierer dar. Der Aspektprogrammierer darf sich in seinen Advices stets nur auf das XPI beziehen. Ähnlich wie beim *Design by Contract* Abschnitt 5.2 können zusätzlich auch Vor- und Nachbedingungen formuliert werden, die bei der Abwicklung an einem Joinpoint gelten müssen.

Ein XPI wird mit den sprachinhärenten Mitteln von AspectJ definiert und stellt eine Art Programmierrichtlinie dar. Das erfordert allerdings, dass sich alle Programmierer an diese Richtlinie halten. Letztendlich ist sie jedoch nur eine weitere Indirektion, da Joinpoints nicht direkt referenziert werden, sondern immer indirekt über das XPI. Der Programmierer des Basisprogramms ist somit gezwungen, für seine Module zusätzlich ein XPI zu definieren. Im Gegensatz dazu ist in LOOM durch die Konzepte der Annotation und Überdeckung das bereits automatisch enthalten.

Neben ajc existiert mit *aspectBench* (abc) eine weitere Implementierung für die AspectJ-Kompiler [Allan u. a. 2005]. Er wurde als Referenzcompiler zu ajc und als ein Framework zur Erweiterung von AspectJ und dessen Optimierung entwickelt. Abc unterstützt den kompletten Sprachumfang von AspectJ.

Eine sehr stark an die Sprachsyntax von AspectJ angelehnte Implementierung für die C++ Programmiersprache ist *AspectC++* [Spinczyk und Lohmann 2007].

7.2.2 JBoss-AOP

[JBoss AOP 2008] ist ein Framework, um aspektorientierte Lösungen für Java zu entwickeln. Es lässt sich nahtlos in den JBoss-Anwendungsserver integrieren, funktioniert aber auch ohne ihn als eigenständige Klassenbibliothek.

Aspekte des JBoss-AOP werden als Java-Klassen modelliert. Sie können Advices, Pointcuts und *Mixins* enthalten. Mixins sind bei JBoss-AOP ein Synonym für Introduktionen.

Grundsätzlich unterstützt JBoss-AOP zwei Varianten der Aspektbeschreibung. Bei der ersten und älteren Variante erfolgt ein Teil der Aspektbeschreibung in einer XML-Konfigurationsdatei. Dort wird ausgewiesen, welche Java-Klassen Aspekte und welche Methoden dieser Klassen Advices und Introduktionen sind. Des Weiteren erfolgt in der XML-Datei auch die Zuordnung der Joinpoints zu den Advices.

Die zweite Variante, welche ab der Java-JDK-Version 1.5 möglich ist, verwendet Annotationen für die Aspektbeschreibung. Java-Klassen können mit der Annotation `@Aspect` als Aspekt ausgewiesen werden. Pointcuts werden durch Annotation von Attributen mit `@PointcutDef` definiert. Das Attribut selbst dient in diesem Fall nur als Namensgeber, Advicemethoden können mit der Annotation `@Bind` auf den Pointcut verweisen (Listing 7.4).

Advices sind reguläre Methoden, die jedoch stets folgende Signatur haben müssen:

```
Object methodenname ( Invocation invocation ).
```

Ein `Invocation`-Exemplar repräsentiert hierbei jeweils den verwobenen Joinpoint. Das entspricht den `Context`-Objekten von LOOM.

Mixins sind Interface-Implementationen im Aspekt, die in einer oder mehreren anderen Klassen eingeführt werden sollen. Sie werden durch die Annotation `@Introduction` oder

```
1 @Aspect (scope = Scope.PER_VM)
2 public class MyAspect
3 {
4   @PointcutDef ("(execution(*␣org.blah.Foo->someMethod())␣OR␣execution(*␣org.blah.Foo
        ->otherMethod()))")
5   public static Pointcut fooMethods;
6   @Bind (pointcut="com.mypackage.MyAspect.fooMethods")
7   public Object myAdvice(Invocation invocation) {
8     ...
9   }
10 }
```

Listing 7.4: Eine Aspektdefinition in JBoss-AOP [nach JBoss AOP 2008, S. 28]

eine entsprechende Beschreibung in der XML-Konfigurationsdatei definiert. Das Konzept ähnelt dem der LOOM-Introduktionen. Allerdings muss bei JBoss-AOP die Zielklasse, in die das Interface eingeführt wird, direkt angegeben werden. Bei LOOM ergibt sich das automatisch durch die Überdeckung des Aspekts.

Es gibt bei JBoss-AOP drei verschiedene Wege, um die Aspekte mit dem Basisprogramm zu verweben. Die erste Variante ist der *JBoss-Präcompiler*. Er manipuliert während der Kompilationsphase die Java-*class*-Dateien und webt so die Aspekte ein. Das entspricht in etwa dem Verfahren von Gripper-Loom.Net.

Die zweite Variante ist das Einweben der Aspekte zur Ladezeit. Java bietet die Möglichkeit, sich in den Prozess des Ladens von Klassen in die virtuelle Maschine einzuhängen und den eigenen Kode abzuwickeln. JBoss-AOP nutzt diesen Mechanismus, um Klassen, die mit Apekten verwoben sind, entsprechend zu transformieren, bevor sie tatsächlich verwendet werden. Auch bei Rapier-Loom.Net werden die Klassen erst zur Laufzeit manipuliert. Da .Net im Gegensatz zu Java keinen entsprechenden Mechanismus kennt, bleibt für Rapier-Loom.Net nur die Verwendung einer weniger transparenten Fabrikmethode.

Die letzte Variante ist das *JBoss-Hotswap*. Die Java-Klassen werden hierzu im Vorfeld instrumentiert und können dann zur Laufzeit mit Aspekten verwoben werden. Ausgelöst wird die Verwebung durch sogenannte *dynamische AOP-Operationen* [JBoss AOP 2008, S.46 ff.]. Mit der in Kapitel 6 vorgestellten DSUP-Bibliothek lässt sich genau das auch in LOOM erreichen.

Der Vergleich von LOOM mit JBoss-AOP zeigt, dass insbesondere bei der seit Java JDK Version 1.5 verfügbaren Annotationsbasierenden Variante einige ähnliche Konzepte zu finden sind. Das Joinpoint-Modell von JBoss-AOP lehnt sich allerdings sehr stark an AspectJ an. Auch hier können private Implementationsdetails mit Aspekten verwoben werden. LOOM verfolgt hier einen mit dem Konzept der Überdeckung anderen Ansatz, da Aspekte selbst auch als Annotationen verwendet werden. Außerdem bietet LOOM noch weitere Konzepte, wie Joinpoint-Initialisierer und Joinpointvariablen.

7.2.3 Spring

Spring ist ein Applikationsframework und wurde entwickelt, um die Entwicklung mit Java/JavaEE zu vereinfachen. Entwickelt wurde es von [Johnson und Hoeller 2004] und stützt sich im Wesentlichen auf drei Architekturmerkmale:

- POJO's (*Plain Old Java Object*, normale Java-Objekte) -basierendes Programmiermodell,
- Verknüfungen durch *Dependency Injection* (automatisches Setzen abhängiger Objekte),
- Aspektorientierte Programmierung.

Die Implementierung von AOP in Spring verfolgt dabei das Ziel, einfache AOP-Features zu implementieren. In der ursprünglichen Version von Spring existierten vier verschiedene Interfaces, jeweils, um einen *Advice Before*, *Advice After* oder *Advice Around* sowie das Abfangen einer Ausnahme zu realisieren. Der Aspekt wurde durch eine Java-Klasse, die eines dieser Interfaces implementiert, definiert. Die Interfaces selbst haben jeweils nur eine Methode `invoke`, die die Implementation des Advices enthält. Somit kann pro Aspekt maximal ein Advice eines Advice-Typs erstellt werden. Die Definition, welcher Advice mit welchen Joinpoints verwoben wird, erfolgt durch zusätzliche XML-Konfigurationsdateien.

Inzwischen verwendet Spring jedoch das bereits vorgestellte *@AspectJ*. Die Spring-Variante für das Microsoft .NET-Framework [Spring.NET Group 2008] verwendete hingegen einige Zeit RAPIER-LOOM.NET [Seovic und Schult 2005].

7.2.4 AspectS

Neben den typischen Implementierungen für Java und .NET existieren auch Lösungen für andere Programmiersprachen. Eine von diesen ist *AspectS* von [Hirschfeld 2002] für die Smalltalk-Programmiersprache. In AspectS werden Aspekte als reguläre Smalltalk-Klassen implementiert. Die Verwebung erfolgt, indem eine *install*-Nachricht an eine Aspektinstanz gesendet wird. Dabei können potentiell alle Objekte eines Images verwoben werden. Joinpoints werden in AspectS durch die Benennung einer Zielklasse und eines Selektors (Methodenname) beschrieben. AspectS unterstützt die Verwebung von Aspektcode:

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

Advices können weiterhin durch Eigenschaften des Ruferkontexts einer Klasse/Instanz verfeinert werden. Das *Class-Specific-First*-Advice untersucht beim Test der Aspektaktivierung, ob auf dem Call-Stack einer gerufenen Methode dieselbe Klasse vorhanden ist. *Class-Specific-All-But-First* untersucht den Call-Stack auf das Vorhandensein mehr als eines Vorkommens derselben Klasse. Das entspricht dem **cflow** in AspectJ. AspectS unterstützt die Erweiterung existierender Klassen um neue Funktionalität durch Introduktionen.

Die Verwebung von Aspektcode mit dem Basisprogramm erfolgt in AspectS während der Laufzeit. AspectS bedient sich hierzu Konzepten von Smalltalk, um die Verarbeitung von Methodenaufrufen anzupassen.

7.2.5 Weitere Lösungen für die .Net-Plattform

Aspect.NET von [Safonov u. a. 2006] ist eine Sammlung von Werkzeugen zur Definition und Verwebung von Aspekten. Ähnlich wie GRIPPER-LOOM.NET benutzt diese Lösung auch das Microsoft Phoenix-Framework zum Einweben der Aspekte als nachgelagertem Kompilierungsschritt. Aspekte werden in Aspect.NET in der Sprache *Aspect.ML* definiert. Diese werden von einem Konverter in annotierte .NET-Klassen umgesetzt. Das Prinzip der Annotation ähnelt dem von LOOM.NET eingesetzten Verfahren.

Die Joinpoint-Beschreibungssprache von Aspect.NET setzt hingegen nur einige Basiskonzepte der Aspektorientierten Programmierung um. Es fehlt zum Beispiel gänzlich die Möglichkeit, Introduktionen definieren zu können. Ebenso ist es nicht ohne weiteres möglich, den Kontrollfluss aus einem Aspekt an das verwobene Ziel zurückzugeben. Dieses recht häufig benötigte Konzept lässt sich nur mit einigen Tricks über Reflection-Schnittstelle der .NET-Umgebung realisieren und ist im Vergleich zu anderen Lösungen um ein Vielfaches (ca. Faktor 200) langsamer [vgl. Forax u. a. 2005; Pobar 2005].

NAspect ist ein dynamischer Aspektweber und wird von [Helander und Johansson 2008] als Teil eines Frameworks für agile, domänenspezifische Softwareentwicklung entwickelt. Wie bei RAPIER-LOOM.NETist auch hier der explizite Aufruf einer Fabrikmethode notwendig, um verwobene Exemplare zu erzeugen. Die Joinpoint-Beschreibung erfolgt bei NAspekt in einer XML-Syntax separat von der Implementierung der zugehörigen .NET-Klasse. Die Aspektbeschreibung selbst ist also über verschiedene Stellen im Quelltext verstreut. Sie mutet auch kompliziert an. Um beispielsweise in einer Klasse beim Setzen eines Properties automatisch eine Ereignismethode aufzurufen, muss der Programmierer ein Interface, zwei Klassen und eine XML-Konfiguration erstellen (siehe ebenda: „Forums / Aspect Oriented Programming / AOP Newb"). In Summe benötigt die Modellierung dieses einfachen überschneidenden Belangs mehr als einhundert Zeilen Quelltext. In Kapitel 4 wurde hingegen das Beobachter-Muster vorgestellt, das hier mit einer einzigen Aspektklasse genau dasselbe leisten würde.

Ein weiterer Nachteil ist , dass auch hier der Kontrollfluss nur über die teuren Reflektionsschnittstellen an den verwobenen Quelltext weitergeleitet werden kann [vgl. eva Kühn u. a. 2006]. Im Gegensatz zu Aspekt.NET ist dies aber durch eine eigene Schnittstelle gekapselt.

[eva Kühn und Schmied 2005] und [eva Kühn u. a. 2006] sowie [Schmied 2006] stellen mit ihrem *XL-AOF* ein Aspektorientiertes Framework vor, das erlaubt, „leichtgewichtige Aspekte" zu definieren. Das *XL-AOF*-Framework ist im Anzatz RAPIER-LOOM.NET recht ähnlich, da es auch als dynamischer Weber in Form einer Bibliothek ausgeführt ist. Die Definition der Aspekte erfolgt auch hier vollständig mit sprachinhärenten Mitteln. Die Deklaration erfolgt zum Teil über .NET-Attribute, als auch über spezielle Interfaces. *XL-AOF* beschränkt sich jedoch darauf, ausschließlich eine kleine Teilmenge der AspectJ-Konzepte abzubilden und hat kein mit LOOM vergleichbares Schnittstellenkonzept.

Phx.Morph von [Eaddy 2006] ist ein Prototyp eines Aspektwebers auf der Basis des Phoenix-Kompilerframeworks von Microsoft. Er hat nur eine eingeschränkte Funktionalität; insbesondere unterstützt er ausschließlich Before- und After-Advices. Damit ist seine Einsatzmöglichkeit relativ beschränkt.

Eine Weiterentwicklung von Phx.Morph ist der *Wicca-Aspektweber* [Eaddy 2007; Eaddy u. a. 2007a]. Es handelt sich hierbei um einen Ansatz, der sowohl Quelltexttransformation, Änderung der Binärdateien und eine angepasste Laufzeitumgebung beinhaltet. Wicca unterstützt die Annotation von einzelnen Anweisungen, um explizit an diesen Stellen Aspektcode einzuweben. Das wird über die Quelltexttransformation realisiert. Außerdem verfügt Wicca über die Fähigkeit, Aspekte zur Laufzeit in das Basisprogramm einzuweben.

Wicca bedient sich spezieller Debug-Schnittstellen und dem *Edit and Continue* Mechanismus der Microsoft Laufzeitumgebung, um die Verwebung vorzunehmen. Diese Schnittstellen sind primär nur für die Verwendung eines Debuggers gedacht. Die Laufzeitumgebung muss entsprechend konfiguriert sein. Neben dem erhöhten Abwicklungsaufwand für ein verwobenes Programm ist eine solche Konfiguration nicht für Produktionsumgebungen geeignet. Auch unterstützt Wicca nur einfache Before- und After-Advices und kennt keine Introduktionen.

7.2.6 Weitere Lösungen für Java

CaesarJ [Aracic u. a. 2006] ist eine Java-basierte Programmiersprache für die Entwicklung variabler Komponenten. Komponenten sind hier grob granulare, in sich geschlossene Funktionseinheiten, in denen jeweils eine Eigenschaft implementiert wird. Eine umschließende Klasse - die Komponente - enthält nun virtuelle Klassen, die für diese Eigenschaft notwendig sind. Die umschließende Klasse kann durch Ableitung verfeinert werden. Dabei lassen sich die in ihr enthaltenen virtuellen Klassen überschreiben und erweitern.

Komponenten können auch aus mehreren Komponenten zusammengesetzt werden, um so verschiedene Eigenschaften miteinander zu kombinieren. Haben zwei virtuelle Klassen in den zu vereinigenden Komponenten den gleichen Namen, so werden sie in den Kompositionen zu einer virtuellen Klasse vereinigt.

```
1 static connector TracingConnector {
2   test.basic.Tracing.Simple temp = new test.basic.Tracing.Simple(* *.*(*));
3   temp.before();
4 }
```

Listing 7.5: Eine JasCo-Konnektor [nach Kemmer 2008]

Komponenten und die enthaltenen virtuellen Klassen werden jeweils durch das spezielle Schlüsselwort `cclass` deklariert. Neben dem Konzept der Ableitung können solche Klassen auch an andere gebunden werden. Das geschieht über das Schlüsselwort `wraps`. Die Semantik entspricht hierbei dem Adapter-Muster [Gamma u. a. 1995]. Durch dieses Konzept lassen sich Klassen in unterschiedlichen Kontexten in die Vererbungshierarchien einpassen.

Aspektklassen sind Caesar-Klassen, die zusätzlich Advices enthalten. Die zu verwebenden Joinpoints werden mit einer Pointcut-Sprache, ähnlich wie in AspectJ definiert. Eine Besonderheit bei CesarJ ist dabei, dass Aspekte in verschiedenen Kontexten aktiviert werden können. Aspekte können lokal, pro Thread oder pro Prozess aktiviert werden. Eine Deaktivierung zur Laufzeit ist ebenfalls möglich.

Die Aktivierung und Deaktivierung zur Laufzeit ist zwar eine interessante Eigenschaft von CeasarJ, stellt jedoch noch einmal erhöhte Anforderungen an den Programmierer. Lässt sich schon für das Einweben von Quelltext in private Implementationsdetails kein Koordinatensystem finden, an dem er sich orientieren könnte, so ist das erst recht nicht für Quelltext möglich, der einmal eingewoben ist und dann wieder nicht. Werden tatsächlich solche Eigenschaften benötigt, sind explizite Konzepte wie die Kontextorientierte Programmierung aus Abschnitt 5.1 vorzuziehen.

JasCO [Suvée u. a. 2003; Vanderperren u. a. 2005] wurde als aspektorientierte Lösung für das *Java Beans* Komponentenmodell. Über sogenannte Konnektoren lassen sich Aspekte in die Ausführung von Methoden einhängen. Diese Konnektoren definieren die Joinpoints, an denen sie wirken sollen. Zusätzlich kann das Verhalten an diesen Konnektoren definiert werden. Das entspricht den Advices; es können *Before*-, *After*- und *Around*-Advices definiert werden. In JasCo werden diese Advices auch als *Hooks* bezeichnet. Aspekte und Hooks müssen durch einen speziellen Kompiler kompiliert werden.

Konnektoren können - da sie separat vom Hook definiert werden - als Schnittstelle zwischen Aspektnutzer und Aspektanbieter fungieren. Im Vergleich zu den LOOM-Annotationen stellen sie jedoch weitere Indirektionsstufe dar. Ein Beispiel für einen Konnektor ist in Listing 7.5 dargestellt.

7.3 Aspektorientierte Modellierung

Mit LOOM.UML wurde eine UML2-Erweiterung vorgestellt, mit der die LOOM-Konzepte bereits bei der Modellierung verwendet werden können. Diese Modellerweiterung wurde entwickelt, um die LOOM-Konzepte schon in der Entwurfsphase anwenden zu können. Für die Modellierung von AspectJ-basierenden Systemen existieren bereits eine Vielzahl von Lösungen [Chavez und Lucena 2002; Han u. a. 2004; Stein u. a. 2002; Suzuki und Yamamoto 1999]. Ein Vergleich mit diesen Lösungen ist allerdings nur bedingt möglich, da LOOM eigene Kompositionskonzepte verwendet, die natürlich Einfluss auf die Modellierung haben. Es gibt jedoch auch Ansätze, die von sich behaupten, allgemeiner gehalten zu sein [Clarke und Walker 2001; Fuentes und Sánchez 2006; Schauerhuber u. a. 2006]. Hier existiert jedoch das Problem, dass wesentliche Konzepte entweder nicht darstellbar sind oder dass sie sich letztlich doch zu sehr an AspectJ orientieren.

Problematisch ist insbesondere die Modellierung von Joinpoints. Fast alle der genannten Lösungen erfordern, neben der Struktur auch das Verhalten modellieren zu müssen, da die Struktur aus dem Verhalten abgeleitet wird. Joinpoints im AspectJ-Sinne stellen Punkte

in der Ausführung des Programmes dar. Mit dem LOOM-Konzept der Überdeckung ist eine Modellierung des Verhaltens nicht notwendig.

Von den zahlreichen existierenden Lösungen sollen vier nachfolgend etwas detaillierter betrachtet werden. Eine umfassende Betrachtung von Aspektorientierten Analyse- und Entwurfsansätzen findet man in [Chitchyan u. a. 2005]. Hier werden 22 Ansätze vorgestellt, evaluiert und kategorisiert. Zusätzlich skizzieren [Chitchyan u. a. 2005] einen integrierten aspektorientierten Analyse- und Entwurfsprozess. Eine ähnliche, aber weniger umfangreiche Bestandsaufnahme erfolgt beispielsweise auch in [Reina u. a. 2004].

7.3.1 Der Ansatz von Suzuki und Yamamoto

Eine der ersten Forschungsarbeiten zur Erweiterung der UML mit Konzepten der Aspektorientierten Programmierung dürfte [Suzuki und Yamamoto 1999] sein. Dieser Ansatz erweitert das UML-Metamodell. Es wird eine neue UML-Meta-Klasse `Aspect` eingeführt, die direkt von `Classifier` ableitet. Letzterer ist auch die Basisklasse der UML-Meta-Klasse `Class`.

In [Suzuki und Yamamoto 1999] demonstrieren sie allerdings ausschließlich eine Notation für Introduktionen; unklar bleibt, wie Advices modelliert werden sollen. Insbesondere fehlt es an Regeln, die die Komposition von Klassen und Aspekten beschreiben.

Die Beziehung zwischen Aspekten und Klassen wird durch die «realize»-Beziehung ausgedrückt. Das ist insofern problematisch, da diese Beziehung nach [Object Management Group 2000] stets zwischen einem spezifizierenden Modellelement und einem Modellelement, das es implementiert, besteht. Aspekte sind jedoch beides: Spezifikation und Implementation.

7.3.2 Der Ansatz von Chavez und Lucena

Auch [Chavez und Lucena 2002] erweitern das UML-Metamodell mit verschiedenen Modellelementen. Der Kern ist dabei ein `CrosscuttingElement`, welches die Basis für alle Modellelemente ist, die eine *Crosscutting*-Beziehung eingehen können. Dabei handelt es sich um die Klassifizierer `Aspect` und `CrosscuttingInterface` (*überschneidende Schnittstellen*) sowie um `CrosscuttingFeature` (*überschneidende Eigenschaften*).

Ein Aspekt wird stets mit einer Menge von überschneidenden Schnittstellen und überschneidenden Eigenschaften parametriert. Eine überschneidende Schnittstelle stellt dabei den Teil einer Klasse (eines Objekts) dar, auf den der Aspekt wirken soll. Dies entspricht den Advices. Introduktionen werden hingegen als überschneidende Eigenschaft modelliert. Sie beschreiben diejenigen Dinge, die ein Aspekt in eine Klasse einbringen möchte.

[Chavez und Lucena 2002] beschränken sich auf die Erweiterung des Metamodells. Einen konkreten Vorschlag für die Darstellungsform der neuen Modellelemente unterbreiten sie hingegen nicht. Außerdem fehlen Regeln, die die konkreten Auswirkungen einer der Verwendung der Modellelemente exakt beschreiben.

7.3.3 Der Ansatz von Stein u.a.

Mit der *Aspect-Oriented Design Notation* (AODM) präsentieren [Stein u. a. 2002] eine Erweiterung der UML-Syntaxes für AspectJ. Diese Erweiterung unterstützt alle bekannten in AspectJ vorkommenden Sprachkonstrukte und spezifiziert eine UML-Implementation des AspectJ-Verwebungsmechanismus. Grundlage ist die UML-Spezifikation Version 1.3 [Object Management Group 2000].

Um Joinpoints sichtbar zu machen, schlagen [Stein u. a. 2002] vor, spezielle Stereotypen in Interaktionsdiagrammen zu verwenden. Die Art des Joinpoints wird als entsprechender Stereotyp einer Nachricht dargestellt. Als Stereotypen stehen beispielsweise `«execute»`, `«call»`, `«create»` oder auch `«initialize»` zur Verfügung. Diese korrelieren mit den entsprechenden Schlüsselwörtern zur Beschreibung von Joinpoints in AspectJ.

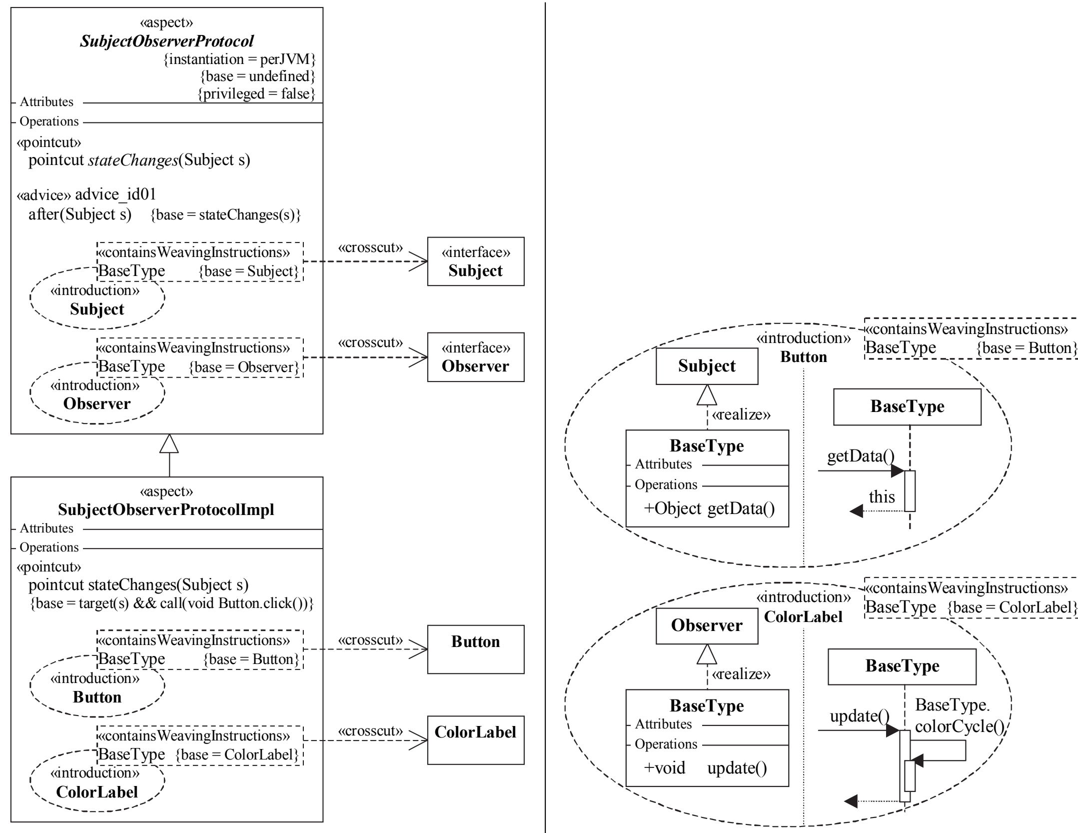


Abbildung 7.3: Modellierung in AODM [Quelle: Stein u. a. 2002, S. 109 f.]

Aspekte sind wie in L𝒪M.Uml als UML-Klasse des speziellen Stereotyps `«aspect»` repräsentiert. Eigenschaften des Aspekts, wie die Art der Exemplarbildung, werden, wie in Abbildung 7.3 dargestellt, in geschweiften Klammern als zusätzliche Eigenschaft der UML-Klasse notiert. Das Aspekt-Stereotyp erlaubt als neue Operationen die Stereotype `«pointcut»`, `«advice»` und `«introduction»`. Hier unterscheidet sich der Ansatz von dem des Autors, bei dem aus Gründen der Übersichtlichkeit diese Modellelemente in einem eigenen Abschnitt definiert werden.

Das Stereotyp `«pointcut»`) steht für eine Pointcutdefinition und setzt sich aus einer Signatur und einer Implementation zusammen. Beides entspricht im Wesentlichen der syntaktischen und semantischen Definition von Pointcuts in AspectJ. Die Implementation wird jedoch in einer zusätzlichen UML-Eigenschaft im Meta-Attribut `base` aufgeschrieben.

Introduktionen, dargestellt durch das Stereotyp `«pointcut»`, werden durch UML-Templates innerhalb des Abschnitts für Operationen in der UML-Klasse modelliert. Die Templateparameter enthalten die Verwebungsvorschrift, die beschreibt, auf welche Klassen die Introduktionen wirken. Aus der Darstellung der UML-Klasse ergibt sich jedoch nicht, welche Elemente in die Klasse eingeführt werden. Das muss in einem zusätzlichen Kollaborationsdiagramm [Object Management Group 2000] (in späteren UML-Versionen entspricht das in etwa dem Interaktionsdiagrammtyp) erläutert werden und entspricht der rechten Seite in Abbildung 7.3.

Dar letzte Stereotyp, das als Operation in einem Aspekt-Stereotypen vorkommen kann, ist `«advice»` und bezeichnet einen Advice. Dieser teilt sich - ähnlich wie bei den Pointcuts - in Signatur und Implementation. Die Signatur gibt dem Advice einen eindeutigen Identifizierer und enthält die Parameter, auf denen im Advice zugegriffen werden soll. Die Implementation hingegen verweist auf den zum Advice gehörenden Pointcut und wird auch hier als UML-Eigenschaft in dem Meta-Attribut `base` notiert.

Im Vergleich zu L𝒪M.Uml ist die Modellierung eines Aspekts mit AODM umfangreicher

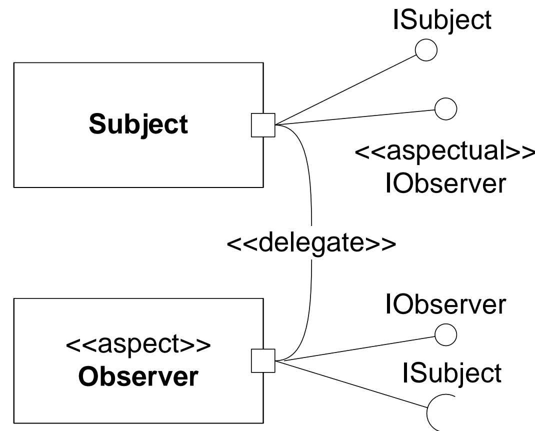


Abbildung 7.4: Introduktionen [nach Fuentes und Sánchez 2006]

und benötigt mehrere Diagramme zur Darstellung aller notwendigen Informationen. Darin ist zugleich auch der große Nachteil gegenüber LOOM.UML begründet, das eine weit kompaktere Darstellung innerhalb eines Diagramms erlaubt. Ein Grund für die Komplexität in AODM ist allerdings auch in der zugrunde liegenden Sprache AspectJ zu sehen.

7.3.4 Fuentes und Sánchez

In [Fuentes und Sánchez 2006] werden verschiedene Ansätze diskutiert, um ein UML-Profil für Aspektorientiertes Design zu definieren. Die Autoren schlagen hier einen generischen Ansatz vor, der für die jeweiligen Anforderungen stets neu angepasst werden soll. Ziel ist es dabei, bestehende UML-Elemente durch möglichst geringe Änderungen so einzusetzen, dass vorhandene Modellierungswerkzeuge ohne großen Aufwand weiterverwendet werden können.

Als Beispiel schlagen [Fuentes und Sánchez 2006] die Verwendung von Stereotypen für bestehende Modellelemente vor. Für Aspekte eignet sich hier das Stereotyp `«aspect»` aus einer UML-Komponente oder einer UML-Klasse. Für Advices und Introduduktionen schlagen sie die Stereotype `«aspectual»` und `«intertyp»` vor. Diese werden in dem Abschnitt zur Methodendefinition verwendet oder markieren die eingeführten Interfaces.

Um das Ziel einer Introduktion - also diejenige Klasse, in der das Interface oder die Methode eingeführt werden soll - zu definieren, schlagen [Fuentes und Sánchez 2006] die Verwendung von Ports (einem UML 2.0-Modellelement) vor, mit dem die Verbindung der Klasse zum jeweiligen Aspekt hergestellt wird (Abbildung 7.4). Ein anderer Vorschlag besteht darin, jeweils ein neues Modellelement mit dem Stereotyp `«weave»` zu definieren. Innerhalb des Rahmens dieses Modellelements werden die Klasse und der Aspekt dargestellt. Für jede Kombination aus Aspekt und Klasse muss dann jedoch ein solches Hilfskonstrukt modelliert werden.

Die Punkte, an denen Advices wirken sollen, werden durch *Hooks* beschrieben. Hooks sind Modellelemente, die ein Verhalten (Verhaltensdiagramme) mit dem Stereotyp `«hook»` beschreiben. An diese können nun Aspekte über eine `AspectBinding`-Metaklasse gebunden werden. Diese Metakasse ist eine UML-Assoziation, die zwischen Hooks und Aspekten hergestellt werden kann. Das Wirken der Hooks kann durch die in der UML definierten *Object Constraint Language* (OCL) [Object Management Group 2008] weiter eingeschränkt werden. Pointcuts können durch eine zusätzliche Metaklasse modelliert werden. Ein Beispiel für einen Advice mit einem Pointcut ist in Abbildung 7.5 zu sehen.

Im Vergleich zu LOOM.UML benötigt auch diese Lösung relativ viele Modellelemente zur Darstellung von Advices und Introduktionen. Ein Grund hierfür ist aber auch darin zu suchen, dass versucht wird, einen generischen Ansatz zu präsentieren.

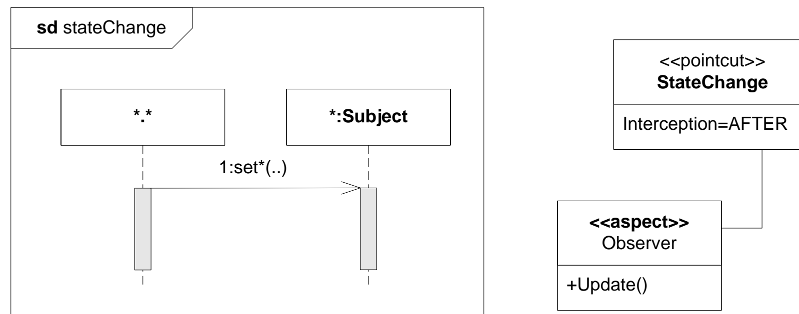


Abbildung 7.5: Advices [nach Fuentes und Sánchez 2006]

7.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Arbeiten zum aktuellen Stand der Forschung der in dieser Dissertation bearbeiteten Themen vorgestellt. Hierbei wurden Arbeiten betrachtet, die sich sowohl allgemein mit dem Problem der Trennung von Belangen beschäftigen als auch speziell Arbeiten zur aspektorientierten Programmierung. Neben den Programmierkonzepten wurden schließlich auch andere Modellierungskonzepte vorgestellt. Zusätzlich erfolgte eine Abgrenzung zu den Loom-Konzepten, den Loom.Net-Werkzeugen und der Loom.Uml-Erweiterung für die UML-Modellierungssprache.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein ganzheitliches Paradigma für die Entwicklung komponentenbasierender Softwaresysteme vorgestellt, das - angefangen beim Entwurf, über die Programmierung, dem Erstellen und Testen, bis zum Betrieb - in jeder Phase des Softwareentwicklungsprozesses einen Beitrag zur Effizienz und Qualitätssteigerung liefert. Mit diesem neuen Konzept können die Belange der Software - also Dinge, die für die Entwicklung, den Betrieb oder anderweitig von Interesse für die Software sind - separat in einzelnen Modulen implementiert werden. Das gelingt im Allgemeinen mit objektorientierten Methoden nicht [Tarr u. a. 1999]; hier überschneiden einzelne Belange oft mehrere Module.

Ausgangspunkt für das vom Autor entwickelte LOOM-Konzept waren die Ideen der aspektorientierten Programmierung nach [Kiczales u. a. 1997], einem Programmierkonzept, das bereits Lösungen für dieses Problem liefert. Insbesondere bei der Modularisierung von Software zeigt dieses Konzept aber auch Schwächen, die im Kapitel 1 ausführlich diskutiert wurden. Die nicht von jedem geteilte, aber auch nicht alleinige Meinung des Autors ist, dass mit den umfangreichen Joinpointsprachen vieler AOP- Lösungen den Programmierern ein Werkzeug in die Hand gegeben wird, das die von [Parnas 1972b] geforderte Modularisierung eher zerstört, anstatt sie zu verbessern.

Neu an dem vom Autor entwickelten und in dieser Dissertation vorgestellten LOOM-Konzept ist, dass der Wirkungsbereich der Aspekte (als Modularisierungseinheit für Belange), die mit objektorientierten Konzepten mehrere Module überschneiden würden, durch die neu entwickelten Konzepte der Überdeckung und Annotation explizit von demjenigen festgelegt werden kann, der auch von der Wirkung betroffen ist. Ein Programmierer eines Moduls kann die zusätzlichen Belange expliziten Referenzierens anderer Module (Aspekte) festlegen, ohne sie selbst implementieren zu müssen. Obwohl damit eine zentrale - wenn auch umstrittene - Forderung der aspektorientierten Programmierung, die Nichtsichtbarkeit von Aspekten [Filman und Friedman 2000], verletzt wird, zeigt die Arbeit, dass die Vorteile einer solchen expliziten Vorgehensweise überwiegen.

Die Grundlage jeder ordentlichen Softwareentwicklung ist der Entwurf. Daher wurde vom Autor in Kapitel 2 eine Erweiterung der Modellierungssprache UML vorgeschlagen, mit der sich alle LOOM-Konzepte schon in den frühen Phasen der Softwareentwicklung einsetzen lassen. Anhand dieser LOOM.UML Erweiterung wurden in Kapitel 2 alle neuen Konzepte ausführlich diskutiert.

Hervorzuheben sind - neben der Überdeckung und Annotation - weitere Konzepte, mit denen sich LOOM.NET von bekannten aspektorientierten Lösungen abhebt. Joinpoint-Initialisierer sind spezielle Advices, die es erlauben, für jeden von einem Advice selektierten Joinpoint separat Initialisierungen durchzuführen. Diese Initialisierungs-Advices werden immer dann abgewickelt, wenn ein neues Exemplar gebildet wird und somit der Kontrollfluss im Kontext dieses Exemplars die selektierten Joinpoints passieren kann. Anwendungsbeispiele für diese Initialisierer sind der *Ereignisabonnent* aus Kapitel 4, *Design-By-Contract* und das *Managementframework für die Windows Fernwartungsschnittstelle* aus dem Kapitel 6.

Neben den Joinpoint-Initialisiereren wurde mit den Joinpoint-Variablen ein weiteres neues AOP-Konzept vorgestellt, mit dem ein gemeinsamer Zugriff auf Attribute zwischen Klassen und Aspekten ermöglicht wird. Aspekte können mit diesem Konzept einfach öffentliche oder für die Vererbung sichtbare Attribute einer überdeckten Klasse in die eigene Implementa-

tion einblenden. Das Konzept erlaubt es, neue Attribute in überdeckte Klassen einzuführen, die wiederum auch von anderen Aspekten verwendet werden können. Schließlich können Joinpoint-Variablen auch Attribute direkt auf einem Joinpoint definieren. Das ermöglicht, Joinpoint-spezifische Daten zu verwalten.

Das Konzept der Nachrichtenmanipulation in LOOM.NET erlaubt es schließlich, die Objektinteraktion flexibel an einem Joinpoint anzupassen. Das beschränkt sich nicht nur auf die Veränderung von Nachrichtenparametern, sondern lässt auch einen Austausch des Adressaten zu. Schließlich kann sogar eine erneute Zuordnung der Nachrichten in Abhängigkeit aller Parametertypen veranlasst werden.

Einen weiteren Beitrag zur Unterstützung der Entwurfsphase liefert die Arbeit mit der Untersuchung verschiedener Entwurfsmuster für typische, immer wiederkehrende Programmieraufgaben, die nicht wiederverwendbar implementiert werden können [Gamma u. a. 1995, S. 3]. Hier wurden bereits bekannte Muster einer Revision unterzogen und diese mit den LOOM-Konzepten neu implementiert. Anhand einer Metrik konnte nachgewiesen werden, dass die neuen Konzepte eine effektivere Lösung erlauben, als das mit den objektorientierten Pendants der Fall ist.

Bezüglich der Eigenschaft der Wiederverwendbarkeit konnte für die bekannten Entwurfsmuster *Einzelstück*, *Beobachter*, und *Stellvertreter* gezeigt werden, dass diese mit den LOOM-Konzepten unabhängig in einer eigenen Komponente implementiert werden können. Die Möglichkeit, eine fertige Komponente wiederzuverwenden, bedeutet eine enorme Verbesserung gegenüber der steten, auf Vorlagen basierenden Neuimplementierung eines Belangs.

Neben den bekannten Mustern wurden auch LOOM.NET-typische Muster, wie das *Klassenkompositionsmuster* oder der *Ereignisabonnent* vorgestellt, die in den Beispielprojekten, aber auch bei der Implementation der LOOM.NET-Aspektweber selbst Anwendung fanden.

Um die LOOM-Konzepte in der Programmierung verwenden zu können, wurden in Kapitel 3 auf der Grundlage eines Industriestandards - der *Common Language Infrastructure* (CLI) [Microsoft Corporation u. a. 2006] - zwei vom Autor entwickelte Aspektweber und die auf CLI-Attributen basierende Spracherweiterung LOOM.NET vorgestellt. Mit LOOM.NET definierte Aspekte können als Komponenten verpackt und an Dritte weitergegeben und verwendet werden. Eine Erweiterung der Ausführungsumgebung oder des verwendeten Kompilers ist nicht erforderlich. Notwendig ist ausschließlich die Verwendung von einem der Aspektweber; RAPIER-LOOM.NET oder GRIPPER-LOOM.NET. Die Natur der CLI erlaubt es auch, die LOOM-Konzepte in allen verfügbaren CLI-Programmiersprachen zu definieren.

RAPIER-LOOM.NETist ein dynamischer Aspektweber, mit dem LOOM-Aspekte zur Laufzeit des Anwendungsprogrammes eingewoben werden. Er ermöglicht dem Programmierer, Belange von Klassen in Abhängigkeit von Laufzeitparametern bei der Exemplarbildung hinzuzufügen oder wegzulassen. Durch eine dynamische Verwebung entstehen jedoch zusätzliche Laufzeitkosten. Im Kapitel 3 wurden verschiedene Optimierungen vorgestellt und durch Messungen belegt, dass diese niedrig gehalten werden können.

Der zweite Aspektweber - GRIPPER-LOOM.NET- wurde als *Post-Compiler*, also als Werkzeug zur Nachkompilierung implementiert. Die Basis für diese Implementierung bildete das Microsoft *Phoenix-Compilerframework*, ein Rahmenwerk für die Kompilerentwicklung. Die Aspekte werden in dieser Lösung direkt in die binären Komponenten eingewoben, so dass die unter RAPIER-LOOM.NETnoch vorhandenen zusätzlichen Laufzeitkosten fast entfallen. Beide Lösungskonzepte wurden durch verschiedene Messungen miteinander verglichen.

Als weitere Unterstützung für die Phase der Programmierung einer Anwendung wurden in Kapitel 5 zwei weitere Programmierkonzepte - die *Kontextorientierten Programmierung* (COP) [Hirschfeld u. a. 2008] und *Design-by-Contract* [Meyer 1992] - auf die CLI unter Verwendung von LOOM.NET adaptiert. Hier konnten Erweiterungen an Programmiersprachen vorgenommen werden, ohne einen neuen Kompiler implementieren zu müssen. LOOM.NET ermöglichte das mit einem sehr geringen Aufwand; dieser betrug für COP weniger als 200 Zeilen

Quelltext und für DBC knapp über 500 Zeilen C#-Quelltext, inklusive der entsprechenden Aspekte.

Im Kapitel 6 werden schließlich verschiedene Projekte vorgestellt, die auf der Basis von LOOM.NET umgesetzt wurden. Hier konnte z. B. im Rahmen einer Industriekooperation mit der Deutschen Post IT-Services GmbH an einem realen Kundenprojekt mit 22.000 Nutzern gezeigt werden, wie die Softwarequalität mit LOOM verbessert werden kann. Insbesondere im Rahmen des Softwaretests konnte mit LOOM-Aspekte eine wesentliche Vereinfachung des Quelltextes erreicht werden, da über 4.300 Logging- und Tracingpunkte entfallen konnten.

Aber auch der Betrieb einer Anwendung kann mit LOOM wesentlich vereinfacht werden: Mit der *Dynamischen Softwareaktualisierung* (DSUP) können beispielsweise Programmierfehler lang laufender Anwendungen zur Laufzeit behoben werden. Hier wurde ein Modell entwickelt, auf dessen Basis aktualisierbare Anwendungen definiert und implementiert werden können. Eine Besonderheit dieser Lösung ist, dass sie keine Einschränkungen bezüglich nebenläufiger Threads hat. Es wurde gezeigt, dass sich bestehende Anwendungen mit erheblichem Quelltextumfang wie z. B. Paint.NET[Brewster 2008] leicht diesem Modell anpassen. Da DSUP ausschließlich auf einer Laufzeitbibliothek und den LOOM-Aspekten besteht, lassen sich die Anwendungen auf der Standardausführungsumgebung abwickeln.

Aber auch die Fernwartung von Software lässt sich durch den Einsatz von LOOM-Aspekten vereinfachen. Hier wurde vom Autor co-betreute Diplomarbeit vorgestellt, die auf Basis von LOOM.NET eine umfangreiche Bibliothek zur Verfügung stellt. Mit dieser Bibliothek lässt sich bestehende Software einfach an Standardfernwartungsschnittstellen anschließen.

Das Kapitel 7 beschäftigte sich schließlich mit aktueller Forschung aus dem Umfeld der Arbeit. Hier wurde eine Abgrenzung bestehender Lösungen im AOP-Umfeld vorgenommen.

Die vorliegende Arbeit konnte nicht auf alle Fragen eine Antwort geben. So gibt es einige Punkte, die Basis für weitere wissenschaftliche Forschung sein können.

Ein Metamodell für Loom.Uml

Die UML2 erlaubt es, durch die Erweiterung ihres Metamodels neue Konzepte in die Sprache aufzunehmen. Mit einer solchen Erweiterung könnte die LOOM.UML Spracherweiterung formalisiert werden. Daraus ergäben sich gleich mehrere Vorteile:

Viele der UML-Modellierungswerkzeuge erlauben die Verwendung eines eigenen Metamodells. Das bedeutet, dass die Werkzeuge die LOOM.NET-Konzepte unterstützen können, ohne selbst angepasst werden zu müssen.

Aber auch eine Validierung der LOOM.UML-Modelle auf ihre Korrektheit könnte durch ein gültiges Metamodell unterstützt werden. In Kapitel 2 wurden einige Fälle dargelegt, in denen ein syntaktisch korrektes Modell zu einem Fehler bei der Erstellung führen kann. So ist es z. B. nicht erlaubt, mit nicht-virtuellen Introduktionen eine neue Methode in eine Klasse einzuführen, die bereits eine Methode mit gleicher Signatur besitzt. Solche Fehler könnten mit einem Metamodell und entsprechender Werkzeugunterstützung schon frühzeitig erkannt werden.

Eine neue Programmiersprache für Loom

In Kapitel 3 wurde dargelegt, wie sich LOOM-Aspekte mit den Konzepten *Common Language Infrastructure* definieren lassen. Es ist ein großer Vorteil, dass LOOM-Aspekte mit jeder CLI-konformen Sprache definiert werden können. Verwirklichen lässt sich das durch intensive Verwendung von CLI-Attributen. Der Nachteil dieses Verfahrens ist allerdings, dass eine solche Aspektdefinition mitunter etwas umständlich aussieht und sich weniger intuitiv programmieren lässt. Mit einer eigenen Programmiersprache - z. B. auf Basis von C# - könnte dieser Mangel ausgeräumt werden. Eine nahtlose Integration würde dazu beitragen, die Konzepte für eine größere Zahl von Nutzern interessant zu machen.

Ein Erweiterung der Common Language Infrastructure

Die CLI ist in ihrer derzeitigen Spezifikation eine objektorientierte Programmierumgebung. Die LOOM-Konzepte wurden durch eine Klassenbibliothek aufgesetzt. Damit die LOOM-Konzepte funktionieren, muss zusätzlich eine der beiden LOOM.NET-Aspektweber verwendet werden.

Eine interessante Aufgabe wäre es, die Konzepte Aspekt, Advice, Introduktion und Joinpoint-Variable direkt in das Metamodell der CLI aufzunehmen und die Funktion des Aspektwebers in den *Intermediate Language Compiler* zu verlegen.

Zu erwarten ist, dass eine solche Vorgehensweise einige Vorteile im Hinblick auf den Laufzeitaufwand bringt. Insbesondere die in Kapitel 3 vorgestellten Hilfskonstrukte würden bei einer solchen direkten Implementation entfallen.

Weitere Entwurfsmuster für LOOM.Net

Das Kapitel 4 beschäftigte sich mit der Revision bekannter Entwurfsmustern; es wurden sechs Muster aus [Gamma u. a. 1995] ausgewählt und mit dem LOOM-Paradigma neu erstellt. Es gibt jedoch eine Vielzahl weiterer Muster, die noch nicht untersucht wurden. Die interessante Frage ist hier, ob es weitere Muster gibt, die sich mit LOOM effizienter erstellen lassen, bzw. bei denen sich die Musterlogik in eigene Module auslagern lässt.

Literaturverzeichnis

[Aksit und Bergmans 2001] AKSIT, Mehmet ; BERGMANS, Lodewijk: Composing Multible Concerns Using Composition Filters. In: *Communications of the ACM* 44, Issue 10 (2001), Oktober, S. 51–57

[Aksit und Tekinerdogan 1998a] AKSIT, Mehmet ; TEKINERDOGAN, Bedir: Aspect-Oriented Programming Using Composition-Filters. In: *ECOOP'98 Workshop Reader*, Springer Verlag, 1998

[Aksit und Tekinerdogan 1998b] AKSIT, Mehmet ; TEKINERDOGAN, Bedir: *Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters.* AOP'98 workshop position paper. 1998

[Aksit und Tripathi 1988] AKSIT, Mehmet ; TRIPATHI, Anand R.: Data Abstraction Mechanisms in SINA/ST. In: *OOPSLA*, 1988, S. 267–275

[Allan u. a. 2005] ALLAN, Chris ; AVGUSTINOV, Pavel ; CHRISTENSEN, Aske S. ; HENDREN, Laurie J. ; KUZINS, Sascha ; LHOTÁK, Jennifer ; LHOTÁK, Ondrej ; MOOR, Oege de ; SERENI, Damien ; SITTAMPALAM, Ganesh ; TIBBLE, Julian: abc: The AspectBench Compiler for AspectJ. In: GLÜCK, Robert (Hrsg.) ; LOWRY, Michael R. (Hrsg.): *GPCE* Bd. 3676, Springer, 2005, S. 10–16. – ISBN 3-540-29138-5

[AOSD Wiki 2007] AOSD WIKI: *Tools for Developers.* `http://www.aosd.net/wiki`. 2007

[Aracic u. a. 2006] ARACIC, Ivica ; GASIUNAS, Vaidas ; MEZINI, Mira ; OSTERMANN, Klaus: Overview of CaesarJ. In: *Transactions on AOSD I, LNCS* 3880 (2006), S. 135 – 173

[Arlat u. a. 1990] ARLAT, Jean ; KANOUN, Karama ; LAPRIE, Jean-Claude: Dependability Modeling and Evaluation of Software Fault-Tolerant Systems. In: *IEEE Trans. Computers* 39 (1990), Nr. 4, S. 504–513

[Arnout und Simon 2001] ARNOUT, Karine ; SIMON, Raphael: The .NET Contract Wizard: Adding Design by Contract to Languages Other than Eiffel. In: *TOOLS (39)*, IEEE Computer Society, 2001, S. 14–23. – ISBN 0-7695-1251-8

[Avizienis 1985] AVIZIENIS, Algirdas: The N-Version Approach to Fault-Tolerant Software. In: *IEEE Trans. Software Eng.* 11 (1985), Nr. 12, S. 1491–1501

[Barnett u. a. 2004a] BARNETT, Michael ; DELINE, Robert ; FÄHNDRICH, Manuel ; LEINO, K. Rustan M. ; SCHULTE, Wolfram: Verification of Object-Oriented Programs with Invariants. In: *Journal of Object Technology* 3 (2004), Nr. 6, S. 27–56

[Barnett u. a. 2004b] BARNETT, Mike ; LEINO, K. Rustan M. ; SCHULTE, Wolfram: The Spec# Programming System: An Overview. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France*, Springer Berlin / Heidelberg, Mar. 2004, S. 49–69

[Bartetzko u. a. 2004] BARTETZKO, Detlef ; FISCHER, Clemens ; MÖLLER, Michael ; WEHRHEIM, Heike: Jass – Java with Assertions. In: *Electronic Notes in Theoretical Computer Science* (2004), Jan.

[Beck 1999] BECK, Kent: *Extreme Programming Explained: Embrace Change*. 1st. Addison-Wesley Professional, October 1999. – 224 S. – ISBN 0201616416

[Ben-Ari 1982] BEN-ARI, Mordechai: *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982. – ISBN 0137010788

[van den Berg u. a. 2005] BERG, K. G. van den ; CONEJERO, J. M. ; CHITCHYAN, R.: AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation / AOSD-Europe. Enschede : AOSD-Europe, May 2005 (AOSD-Europe-UT-01 D9). – Technical Report. – S. 90. – ISBN not assigned

[Bidan u. a. 1998] BIDAN, Christophe ; ISSARNY, Valerie ; SARIDAKIS, Titos ; ZARRAS, Apostolos: A Dynamic Reconfiguration Service for CORBA. In: *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*. Washington, DC, USA : IEEE Computer Society, 1998, S. 35. – ISBN 0-8186-8451-8

[Bitter 2005] BITTER, Christian: *AOP with Rapier-LOOM.NET*. `http://bittis-blog.blogspot.com/2006/05/aop-with-rapier-loomnet.html`. 2005

[Born u. a. 2004] BORN, Marc ; HOLZ, Eckhardt ; KATH, Olaf: *Softwareentwicklung mit UML 2*. Addison-Wesley, 2004. – ISBN 3-8273-2086-0

[Bracha und Cook 1990] BRACHA, Gilad ; COOK, William: Mixin-based inheritance. In: *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM Press, 1990, S. 303–311. – ISBN 0-201-52430-X

[Brewster 2008] BREWSTER, Rick: *Paint.NET Homepage*. `http://www.getpaint.net`. 2008

[Campbell u. a. 1993] CAMPBELL, Roy H. ; ISLAM, Nayeem ; RAILA, David ; MADANY, Peter: Designing and implementing Choices: an object-oriented system in C++. In: *Commun. ACM* 36 (1993), Nr. 9, S. 117–126. – ISSN 0001-0782

[Carrillo-Castellón u. a. 1996] CARRILLO-CASTELLÓN, Manuela ; GARCÍA-MOLINA, Jesús ; PIMENTEL, Ernesto ; REPISO, Israel: Design by Contract in Smalltalk. In: *Journal of Object-Oriented Programming* 9 (1996), Nov.-Dec., Nr. 7, S. 23–28

[Chavez und Lucena 2002] CHAVEZ, Christina ; LUCENA, Carlos: *A Metamodel for Aspect-Oriented Modeling*. Workshop on Aspect-Oriented Modeling with UML (AOSD). 2002

[Chitchyan u. a. 2005] CHITCHYAN, R. ; RASHID, A. ; SAWYER, Pete ; GARCIA, A. ; BAKKER, J. ; ALARCON, M. P. ; TEKINERDOGAN., B. ; CLARKE, S. ; JACKSON, Andrew: Survey of Aspect-Oriented Analysis and Design / AOSD Europe. Mai 2005. – Forschungsbericht. Technical Report AOSD-Europe-ULANC-9

[Clarke und Walker 2001] CLARKE, Siobhán ; WALKER, Robert J.: Composition Patterns: An Approach to Designing Reusable Aspects. In: *Software Engineering, International Conference on* 0 (2001), S. 0005

[Clifton u. a. 2000] CLIFTON, Curtis ; LEAVENS, Gary T. ; CHAMBERS, Craig ; MILLSTEIN, Todd: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch fo Java. In: *Proceedings of the OOPSLA 2000, ACM SIGPLAN Notices*, 2000, S. 130–145. – ISBN 1-58113-200-X

[Constantinides u. a. 2004] CONSTANTINIDES, Constantinos ; SKOTINIOTIS, Therapon ; STOERZER, Maximilian: *AOP considered harmful*. 1st European Interactive Workshop on Aspect Systems (EIWAS). 2004

[Conway und Goebel 2001] CONWAY, Damian ; GOEBEL, Garrett C.: *Contract - Design-by-Contract OO in Perl*. `http://search.cpan.org/~mdupont/Introspector-0.04/lib/Class/Contract.pm`. Feb. 2001

[Coplien 1992] COPLIEN, J. O.: *Advanced C++, Programming Styles and Idioms*. Addison-Wesley, 1992

[Costanza und Hirschfeld 2005] COSTANZA, Pascal ; HIRSCHFELD, Robert: Language constructs for context-oriented programming: an overview of ContextL. In: *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*. New York, NY, USA : ACM, 2005, S. 1–10

[Costanza u. a. 2006] COSTANZA, Pascal ; HIRSCHFELD, Robert ; MEUTER, Wolfgang D.: Efficent Layer Activation for Switching Context-dependent Behavior. In: *Modular Programming Languages* Bd. 4228/2006, Springer Berlin / Heidelberg, 2006, S. 84–103

[Crispin 2003] CRISPIN, M.: *RFC 3501 - INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*. Internet RFC/STD/FYI/BCP Archives. 2003

[Cristian 1991] CRISTIAN, Flaviu: Understanding fault-tolerant distributed systems. In: *Communications of the ACM* 34 (1991), Nr. 2, S. 56–78. – URL `citeseer.ist.psu.edu/cristian93understanding.html`

[Detlefs und Agesen 1999] DETLEFS, David ; AGESEN, Ole: Inlining of Virtual Methods. In: GUERRAOUI, Rachid (Hrsg.): *ECOOP* Bd. 1628, Springer, 1999, S. 258–278. – ISBN 3-540-66156-5

[Dijkstra 1968] DIJKSTRA, Edsger W.: Letters to the editor: go to statement considered harmful. In: *Communications of the ACM archive* 11 (1968), Nr. 3, S. 147–148

[Distributed Management Task Force DMTF Homepage] DISTRIBUTED MANAGEMENT TASK FORCE: *2008*. `www.dmtf.org`. DMTF Homepage

[Dongarra u. a. 2003] DONGARRA, Jack ; LUSZCZEK, Piotr ; PETITET, Antoine: The LINPACK Benchmark: past, present and future. In: *Concurrency and Computation: Practice and Experience* 15 (2003), Nr. 9, S. 803–820

[Dutchyn u. a. 2001] DUTCHYN, Christopher ; LU, Paul ; SZAFRON, Duane ; BROMLING, Steven ; HOLST, Wade: Multi-Dispatch in the Java Virtual Machine: Design and Implementation. In: *COOTS*, USENIX, 2001, S. 77–92

[Eaddy 2006] EADDY, Marc: *Phx.Morph: Weaving using the Microsoft Phoenix compiler back-end*. Demonstration at Aspect-Oriented Software Development (AOSD 2006), Bonn, Germany. Mar. 2006

[Eaddy 2007] EADDY, Marc: *Wicca 2.0: Dynamic Weaving using the .NET 2.0 Debugging APIs*. Demonstration at Aspect-Oriented Software Development (AOSD 2007), Vancouver, British Columbia. Mar. 2007

[Eaddy u. a. 2007a] EADDY, Marc ; AHO, Alfred V. ; HU, Weiping ; MCDONALD, Paddy ; BURGER, Julian: Debugging Aspect-Enabled Programs. In: LUMPE, Markus (Hrsg.) ; VANDERPERREN, Wim (Hrsg.): *Software Composition* Bd. 4829, Springer, 2007, S. 200–215. – ISBN 978-3-540-77350-4

[Eaddy u. a. 2007b] EADDY, Marc ; CYMENT, Alan ; LAAR, Pierre van de ; SCHMIED, Fabian ; SCHULT, Wolfgang: Whitepaper: The Value of Improving the Separation of Concerns / Department of Computer Science, Columbia University. 2007 (CUCS-001-07). – Forschungsbericht

[Eaddy u. a. 2008] EADDY, Marc ; ZIMMERMANN, Thomas ; KAITLIN D. SHERWOOD, Vibhav G. ; MURPHY, Gail C. ; NAGAPPAN, Nachiappan ; AHO, Alfred V.: Do Crosscutting Concerns Cause Defects? In: *IEEE Transactions on Software Engineering*, IEEE Computer Society, May 2008

[Edelson 1992] EDELSON, Daniel R.: Smart Pointers: They're Smart, But They're Not Pointers. In: *C++ Conference*, Usenix Association, 1992, S. 1–20

[Eugster u. a. 2003] EUGSTER, Patrick T. ; FELBER, Pascal A. ; GUERRAOUI, Rachid ; KERMARREC, Anne-Marie: The many faces of publish/subscribe. In: *ACM Comput. Surv.* 35 (2003), Nr. 2, S. 114–131. – ISSN 0360-0300

[Fibonacci 1857] FIBONACCI, Leonardo ; BONCOMPAGNI, Baldassare (Hrsg.): *Scritti di Leonardo Pisano matematico del secolo decimoterzo.* Tipografia delle scienze matematiche e fisiche, 1857

[Filman und Friedman 2000] FILMAN, Robert E. ; FRIEDMAN, Daniel P.: Aspect-Oriented Programming is Quantification and Obliviousness. In: *Proceedings of Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Addison-Wesley, 2000, S. 21–35

[Filman und Friedman 2005] FILMAN, Robert E. ; FRIEDMAN, Daniel P.: Aspect-Oriented Programming Is Quantification and Obliviousness. In: FILMAN, Robert E. (Hrsg.) ; ELRAD, Tzilla (Hrsg.) ; CLARKE, Siobhán (Hrsg.) ; AKŞIT, Mehmet (Hrsg.): *Aspect-Oriented Software Development.* Boston : Addison-Wesley, 2005, S. 21–35. – ISBN 0-321-21976-7

[Forax u. a. 2005] FORAX, Rémi ; DURIS, Etienne ; ROUSSEL, Gilles: Reflection-based implementation of Java extensions: the double-dispatch use-case. In: *Journal of Object Technology, vol. 4, no. 10, Special Issue: OOPS Track at SAC 2005 SantaFe* (2005), Dec., S. 49–69. – URL `http://www.jot.fm/issues/issues 2005 12/article3`

[Fowler 1999] FOWLER, Martin: *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999. – ISBN 0-201-48567-2

[Fowler 2004] FOWLER, Martin: *Inversion of control containers and the dependency injection.* Januar 2004. – URL `http://martinfowler.com/articles/injection.html#InversionOfControl`

[Friedman und Wand 1984] FRIEDMAN, Daniel P. ; WAND, Mitchell: Reification: Reflection without Metaphysics. In: *LISP and Functional Programming*, ACM Press, 1984, S. 348–355. – URL `http://dblp.uni-trier.de/db/conf/lfp/lfp1984.html\#FriedmanW84`

[Fuentes und Sánchez 2006] FUENTES, Lidia ; SÁNCHEZ, Pablo: *Elaborating UML 2.0 Profiles for AO Design.* Workshop on Aspect-Oriented Modeling at AOSD. 2006

[FxCOP 2006] FXCOP: *FxCOP Team Page.* `http://www.gotdotnet.com/Team/FxCop/`. 2006

[Gabriel u. a. 2006] GABRIEL, Richard P. ; JR., Guy L. S. ; STEIMANN, Friedrich ; WALDO, Jim ; KICZALES, Gregor ; SULLIVAN, Kevin: Aspects and/versus modularity the grand debate. In: TARR, Peri L. (Hrsg.) ; COOK, William R. (Hrsg.): *OOPSLA Companion*, ACM, 2006, S. 935–936. – ISBN 1-59593-491-X

[Gamma u. a. 1995] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns*. Addison-Wesley Publishing Company, 1995. – ISBN 0-201-63361-2

[Google Trends 2008] GOOGLE TRENDS: *Häufigkeit von Suchanfragen für UML2*. http://www.google.de/trends. 2008. – [Online; zugegriffen am 1. August 2008]

[Gosling u. a. 2005] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *The Java Language Specification*. 3rd edition. Addison Wesley, 2005. – ISBN 0-321-24678-0

[Greenwood u. a. 2007] GREENWOOD, Phil ; BARTOLOMEI, Thiago T. ; FIGUEIREDO, Eduardo ; DÓSEA, Marcos ; GARCIA, Alessandro F. ; CACHO, Nélio ; SANT'ANNA, Cláudio ; SOARES, Sérgio ; BORBA, Paulo ; KULESZA, Uirá ; RASHID, Awais: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ERNST, Erik (Hrsg.): *ECOOP* Bd. 4609, Springer, 2007, S. 176–200. – ISBN 978-3-540-73588-5

[Griswold u. a. 2006] GRISWOLD, William G. ; SULLIVAN, Kevin ; SONG, Yuanyuan ; SHONLE, Macneil ; TEWARI, Nishit ; CAI, Yuanfang ; RAJAN, Hridesh: Modular Software Design with Crosscutting Interfaces. In: *IEEE Software* 23 (2006), Nr. 1, S. 51–60. – ISSN 0740-7459

[Guerreiro 2002] GUERREIRO, Pedro: Another Mediocre Assertion Mechanism for C++. In: *Proceedings of TOOLS 33*, 2002

[Hachani und Bardou 2002] HACHANI, Ouafa ; BARDOU, Daniel: *Using Aspect-Oriented Programming for Design Patterns Implementation*. OOIS 2002 Workshops, Montpellier, France. 2002

[Han u. a. 2004] HAN, YAN ; KNIESEL, Günter ; CREMERS, Armin B.: A Meta Model for AspectJ / Uni Bonn. URL http://www.cs.uni-bonn.de/~gk/papers/, 2004. – Forschungsbericht. – ISSN 0944-8535

[Hannemann 2005] HANNEMANN, Jan: *Role-Based Refactoring of Crosscutting Concerns*, University of British Columbia, Dissertation, 2005

[Hannemann und Kiczales 2002] HANNEMANN, Jan ; KICZALES, Gregor: Design pattern implementation in Java and AspectJ. In: *OOPSLA 2002*, 2002, S. 161–173

[Hannemann und Kiczales 2003] HANNEMANN, Jan ; KICZALES, Gregor: *Homepage: Aspect-Oriented Design Pattern Implementation*. http://www.cs.ubc.ca/labs/spl/projects/aodps.html. 2003

[Harrison und Ossher 1993] HARRISON, William ; OSSHER, Harold: Subject-oriented programming: a critique of pure objects. In: *SIGPLAN Not.* 28 (1993), Nr. 10, S. 411–428. – ISSN 0362-1340

[Haupt 2005] HAUPT, Michael: *Virtual Machine Support for Aspect-Oriented Programming Languages*, Technische Universität Darmstadt, Dissertation, Okt. 2005

[Helander und Johansson 2008] HELANDER, Mats ; JOHANSSON, Roger: *Puzzle.NET Homepage*. www.puzzleframework.net. 2008

[Hicks und Nettles 2005] HICKS, Michael ; NETTLES, Scott: Dynamic software updating. In: *ACM Trans. Program. Lang. Syst.* 27 (2005), Nr. 6, S. 1049–1096. – ISSN 0164-0925

[Hirschfeld 2002] HIRSCHFELD, Robert: AspectS - Aspect-Oriented Programming with Squeak. In: AKSIT, Mehmet (Hrsg.) ; MEZINI, Mira (Hrsg.) ; UNLAND, Rainer (Hrsg.): *NetObjectDays* Bd. 2591, Springer, 2002, S. 216–232. – ISBN 3-540-00737-7

[Hirschfeld u. a. 2006] HIRSCHFELD, Robert ; COSTANZA, Pascal ; NIERSTRASZ, Oscar: Extending advice activation in AspectS. In: *Proceedings of the AOSD Workshop on Open and Dynamic Aspect Languages*, 2006

[Hirschfeld u. a. 2008] HIRSCHFELD, Robert ; COSTANZA, Pascal ; NIERSTRASZ, Oscar: Context-oriented Programming. In: *Journal of Object Technology (JOT)* 7 (2008), Mar.-Apr., Nr. 3, S. 125–151. – URL `http://www.jot.fm/issues/issue_2008_03/article4.pdf`

[Hölzl] HÖLZL, Matthias: *Design by Contract in Common Lisp*. – URL {`http://www.muc.de/~hoelzl/tools/dbc/dbc.lisp`}

[Hugunin 2007] HUGUNIN, Jim: *Dynamic Languages on .NET - IronPython and Beyond*. `http://blogs.msdn.com/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx`. 2007

[Hunt 2002] HUNT, Andrew: *Design by Contract in Ruby*. Aug. 2002. – URL {`http://www.pragmaticprogrammer.com/ruby/downloads/dbc.html`}

[de Icaza und Andere 2008] ICAZA, Miguel de ; ANDERE: *Mono Homepage*. `http://www.mono-project.com`. 2008

[International Organization for Standardization 2001] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Information Technology — Syntactic Metalanguage — Extended BNF*. ISO/IEC 14977. 2001. – URL `http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26153`

[Iron Phyton Team 2008] IRON PHYTON TEAM: *IronPython Internetseite*. `http://codeplex.com/IronPython`. 2008

[JBoss AOP 2008] JBOSS AOP: *JBoss AOP - Aspect-Oriented Framework for Java (Reference Documentation)*. Version 1.5. http://www.jboss.org: , 2008

[Jeske u. a. 2006] JESKE, Janin ; BREHMER, Bastian ; MENGE, Falko ; HÜTTENRAUCH, Stefan ; ADAM, Christian ; SCHÜLER, Benjamin ; SCHULT, Wolfgang ; RASCHE, Andreas ; POLZE, Andreas: Aspektorientierte Programmierung - Überblick über Techniken und Werkzeuge / Hasso-Plattner-Institut. 2006. – Forschungsbericht. – ISSN 1613-5652

[Johnson und Hoeller 2004] JOHNSON, Rod ; HOELLER, Juergen: *Expert One-on-One J2EE Development without EJB*. Wrox, June 2004. – ISBN 0764558315

[Karaorman u. a. 1999] KARAORMAN, Murat ; HÖLZLE, Urs ; BRUNO, John L.: jContractor: A Reflective Java Library to Support Design by Contract. In: COINTE, Pierre (Hrsg.): *Reflection* Bd. 1616, Springer, 1999, S. 175–196. – ISBN 3-540-66280-4

[Kasten u. a. 2002] KASTEN, Eric P. ; MCKINLEY, Philip K. ; SADJADI, S. M. ; STIREWALT, Kurt: Separating Introspection and Intercession to Support Metamorphic Distributed Systems. In: *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. Washington, DC, USA : IEEE Computer Society, 2002, S. 465–472. – ISBN 0-7695-1588-6

[Kemmer 2008] KEMMER, Thomas: *Runtime AOP*. `http://lobmenschen.de/index.php/Runtime_AOP`. 2008

[eva Kühn u. a. 2006] KÜHN eva ; FESSL, Gerald ; SCHMIED, Fabian: Aspect-Oriented Programming with Runtime-Generated Subclass Proxies and .NET Dynamic Methods. In: *Journal of .NET Technologies* 4 (2006), S. 17–24. – ISBN 80-86943-13-5

[eva Kühn und Schmied 2005] KÜHN eva ; SCHMIED, Fabian: XL-AOF: lightweight aspects for space-based computing. In: *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*. New York, NY, USA : ACM, 2005. – ISBN 1-59593-265-8

[Köhne u. a. 2005] KÖHNE, Kai ; SCHULT, Wolfgang ; POLZE, Andreas: *Design by Contract in .NET Using Aspect Oriented Programming*. `http://www.dcl.hpi.uni-potsdam.de/research/loom/papers.htm`. 2005

[Kiczales u. a. 2001] KICZALES, G. ; HILSDALE, E. ; HUGUNIN, J. ; KERSTEN, M. ; PALM, J. ; GRISWOLD, W. G.: An overview of AspectJ. In: KNUDSEN, J. L. (Hrsg.): *Proc. ECOOP 2001, LNCS 2072*. Berlin : Springer-Verlag, June 2001, S. 327–353

[Kiczales u. a. 1997] KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Christina V. ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect Oriented Programming. In: *European Conference on Object-Oriented Programming (ECOOP)*. Finnland : Springer Verlag LNCS 1241, June 1997

[Kiczales und Mezini 2005] KICZALES, Gregor ; MEZINI, Mira: Aspect-oriented programming and modular reasoning. In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, S. 49–58. – ISBN 1-59593-963-2

[Kiczales und Voelter 2006] KICZALES, Gregor ; VOELTER, Markus: *Episode 11, Interview with Gregor Kiczales*. Podcast from Software Engeneering Radio, http://www.se-radio.net. 2006

[Kostian 2005] KOSTIAN, Andreas: *Vergleich der Performance-Auswirkungen generierter Adaptoren*, Carl von Ossietzky Universität Oldenburg, Masterarbeit, 2005

[Kramer und Magee 1990] KRAMER, J. ; MAGEE, J.: The Evolving Philosophers Problem: Dynamic Change Management. In: *IEEE Transactions on Software Engineering* 16 (1990), Nr. 11, S. 1293–1306

[Kramer 1998] KRAMER, Reto: iContract – The Java Design by Contract Tool. In: *Proceedings of TOOLS USA* IEEE Computer Society (Veranst.), 1998

[Kutner u. a. 2008] KUTNER, Joe ; PERKINS, Louise ; YENDURI, Sumanth ; ZAND, Farnaz ; ZHANG, Joe: AOP Maintains an Independent Coordinate System. In: *Fifth International Conference on Information Technology: New Generations (itng 2008)*, 2008, S. 1290–1291

[Laddad 2002] LADDAD, Ramnivas: I want my AOP! In: *JavaWorld.com* (2002)

[Laddad 2003] LADDAD, Ramnivas: *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, July 2003. – ISBN-10: 1930110936 ISBN-13: 978-1930110939

[Laprie 1985] LAPRIE, Jean-Claude: On Computer System Dependability: Faults, Errors, and Failures. In: *COMPCON*, IEEE Computer Society, 1985, S. 256–259. – ISBN 0-8186-0613-4

[Laprie 1992] LAPRIE, Jean-Claude: *Dependability: Basic Concepts and Terminology*. Springer-Verlag Wien, 1992

[Lidin 2002] LIDIN, Serge: *Inside Microsoft .NET IL Assembler*. 1st edition. Redmond, Washington : Microsoft Press, 2002. – ISBN 0-7356-1547-0

[Lippert und Lopes 2000] LIPPERT, Martin ; LOPES, Cristina V.: A study on exception detecton and handling using aspect-oriented programming. In: *Proceedings of the 22nd International Conference on Software Engineering*, ACM Press, 2000, S. 418–427. – ISBN 1-58113-206-9

[Luckham u. a. 1987] LUCKHAM, David C. ; HENKE, Friedrich W. von ; KRIEG-BRÜCKNER, Bernd ; OWE, Olaf: *Lecture Notes in Computer Science*. Bd. 260: *ANNA - A Language for Annotating Ada Programs, Reference Manual*. Springer, 1987. – ISBN 3-540-17980-1

[Lumi und Lumi 2008] LUMI, Willy ; LUMI, Ivar: *LumiSoft Homepage*. `http://www.lumisoft.ee`. 2008

[von Löwis u. a. 2007] LÖWIS, Martin von ; DENKER, Marcus ; NIERSTRASZ, Oscar: Context-oriented programming: beyond layers. In: DEMEYER, Serge (Hrsg.) ; PERROT, Jean-François (Hrsg.): *ICDL* Bd. 286, ACM, 2007, S. 143–156. – ISBN 978-1-60558-084-5

[Maes 1987] MAES, Pattie: Concepts and experiments in computational reflection. In: *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA : ACM Press, 1987, S. 147–155. – ISBN 0-89791-247-0

[Meemken] MEEMKEN, Dieter: *User Manual for the JaWA-Precompiler, Version 1.0*. `http://theoretica.informatik.uni-oldenburg.de/~jawa/doc.engl.html`

[Meyer 1992] MEYER, Bertrand: *Eiffel: the language*. 1st edition. New York, NY : Prentice Hall, 1992. – ISBN 0-13-247925-7

[Michie 1968] MICHIE, Donald: „Memo“ functions and machine learning. In: *Nature* Vol. 218 (1968), S. 19–22

[Microsoft 2008a] MICROSOFT: *Custom Serialization*. `http://msdn2.microsoft.com/en-us/library/ty01x675.aspx`. 2008

[Microsoft 2008b] MICROSOFT: *Using Hotpatching Technology to Reduce Servicing Reboots*. `http://technet2.microsoft.com/windowsserver/en/library/e55050fc-22c9-4984-9bae-b8b0527334721033.mspx`. 2008

[Microsoft Corporation 2003a] MICROSOFT CORPORATION: *C# Version 1.2 Specification*. Version 1.2. Redmond, USA: , 2003

[Microsoft Corporation 2003b] MICROSOFT CORPORATION: *Microsoft .Net Framework 1.1 Class Library Reference Volumes 1-4*. Microsoft Press, Feb. 2003. – ISBN 978-0735615557

[Microsoft Corporation 2005a] MICROSOFT CORPORATION: *C# Version 2.0 Specification*. Version 2.0. Redmond, USA: , July 2005

[Microsoft Corporation 2005b] MICROSOFT CORPORATION: *C# Version 3.0 Specification*. Version 3.0. Redmond, USA: , Sep. 2005

[Microsoft Corporation 2006] MICROSOFT CORPORATION: *Shared Source Common Language Infrastructure 2.0 Release*. `http://www.microsoft.com/downloads/details.aspx?FamilyID=8c09fd61-3f26-4555-ae17-3121b4f51d4d`. Mar. 2006. – Webseite

[Microsoft Corporation 2007a] MICROSOFT CORPORATION: *Microsoft BizTalk-Server*. `http://www.microsoft.com/biztalk/default.mspx`. 2007. – Webseite

[Microsoft Corporation 2007b] MICROSOFT CORPORATION: *.NET Framework Developer's Guide (.NET Compact Framework).* `http://msdn.microsoft.com/en-us/library/f44bbwa1.aspx`. Nov. 2007. – Webseite

[Microsoft Corporation 2008a] MICROSOFT CORPORATION: *Microsoft Visual Studio 2008.* `http://www.microsoft.com/germany/msdn/vstudio/default.mspx`. 2008. – Webseite

[Microsoft Corporation 2008b] MICROSOFT CORPORATION: *.NET Framework.* `http://msdn.microsoft.com/en-us/netframework/default.aspx`. 2008. – Webseite

[Microsoft Corporation 2008c] MICROSOFT CORPORATION: *Phoenix Compiler Framework.* `http://research.microsoft.com/phoenix/phoenixrdk.aspx`. 2008. – Webseite

[Microsoft Corporation u. a. 2006] MICROSOFT CORPORATION ; BORLAND ; FUJITSU SOFTWARE CORPORATION ; HEWLETT-PACKARD ; INTEL CORPORATION ; IBM CORPORATION ; ISE ; IT UNIVERSITY OF COPENHAGEN ; JAGGER SOFTWARE LTD. ; MONASH UNIVERSITY ; NETSCAPE ; NOVELL/XIMIAN ; PHONE.COM ; PLUM HALL ; SUN MICROSYSTEMS ; UNIVERSITY OF CANTERBURY (NZ): *Common Language Infrastructure (CLI).* ECMA-335 and ISO/IEC 23271. June 2006

[Miguel Alexandre Wermelinger 1999] MIGUEL ALEXANDRE WERMELINGER: *Specification of Software Architecture Reconfiguration*, Universidade Nova de Lisboa, Dissertation, 1999

[Milovanovic 2004] MILOVANOVIC, Igor: *Aspektorientierte Programmierung mit .NET.* `http://geekswithblogs.net/imilovanovic/articles/11592.aspx`. 2004

[Moazami-Goudarzi 1999] MOAZAMI-GOUDARZI, K.: *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*, Imperial College London, Dissertation, Mar. 1999. – URL `{ http://www-dse.doc.ic.ac.uk/~km2/phd/thesis.ps.gz}`

[Moazami-Goudarzi 1999] MOAZAMI-GOUDARZI, K.: *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*, Imperial College London, Dissertation, Mar. 1999. – URL `http://www-dse.doc.ic.ac.uk/~km2/phd/thesis.ps.gz`

[Moon 1986] MOON, David A.: Object-Oriented Programming with Flavors. In: *OOPSLA*, 1986, S. 1–8

[Myers 1996] MYERS, Nathan: A New and Useful Template Technique: „Traits“. In: LIPPMAN, Stanley B. (Hrsg.): *C++ Gems: Programming Pearls from the C++ Report*, Cambridge University Press, 1996, S. 451–457. – ISBN 9780135705810

[Nierstrasz u. a. 2005] NIERSTRASZ, Oscar ; DUCASSE, Stéphane ; REICHHART, Stefan ; SCHÄRLI, Nathanael: Adding Traits to (Statically Typed) Languages / Institut für Informatik. Universität Bern, Switzerland, Dec. 2005 (IAM-05-006). – Technical Report. – URL `http://www.iam.unibe.ch/~scg/Archive/Papers/Nier05gTraitsCSharp.pdf`

[Object Management Group 2000] Object Management Group (Veranst.): *Unified Modeling Language specification – Version 1.3.* Sep. 2000

[Object Management Group 2001] Object Management Group (Veranst.): *Unified Modeling Language specification – Version 1.4.* Sep. 2001

[Object Management Group 2007a] OBJECT MANAGEMENT GROUP: Unified Modeling Language 2.1.2 Infrastructure Specification / Object Management Group. URL `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF`, Nov. 2007 (Version 2.1.2). – Specification

[Object Management Group 2007b] Object Management Group: Unified Modeling Language 2.1.2 Super-Structure Specification / Object Management Group. URL http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF, Nov. 2007 (Version 2.1.2). – Specification

[Object Management Group 2008] Object Management Group: *OMG-Webseite*. http://www.omg.org/. 2008

[Odersky und Zenger 2005] Odersky, Martin ; Zenger, Matthias: Scalable component abstractions. In: Johnson, Ralph (Hrsg.) ; Gabriel, Richard P. (Hrsg.): *OOPSLA*, ACM, 2005, S. 41–57. – ISBN 1-59593-031-0

[Olejniczak 2007] Olejniczak, Michal: *Aspektorientiertes Managementframework für die Windows Fernwartungsschnittstelle (WMI)*, Universität Potsdam, Masterarbeit, 2007

[Opdyke 1992] Opdyke, William F.: *Refactoring Object-Oriented Frameworks*, University of Illinois at Urbana Champaign, Dissertation, 1992

[Ossher u. a. 1994] Ossher, H. ; Harrison, W. ; Budinsky, F. ; Simmonds, I.: Subject-oriented programming: Supporting decentralized development of objects. In: *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, URL citeseer.ist.psu.edu/ossher94subjectoriented.html, 1994

[Parnas 1972a] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In: *Commun. ACM* 15 (1972), Nr. 12, S. 1053–1058. – ISSN 0001-0782

[Parnas 1972b] Parnas, David L.: Information distribution aspects of design methology. In: *Proceedings of the IFIP Congress 1, North-Holland*, 1972, S. 339–344

[Pietrek 2002] Pietrek, Matt: Die Profilschnittstelle von .Net. In: *SYSTEM-Journal* 02 (2002), S. 42–50

[Piveta und Zancanella 2003] Piveta, Eduardo K. ; Zancanella, Luiz C.: Observer Pattern using Aspect-Oriented Programming. In: *Third Latin American Conference on Pattern Languages of Programming (SugarLoafPLOP 2003)*, URL http://www.cin.ufpe.br/~sugarloafplop/, 2003

[Plösch 1997] Plösch, Reinhold: Design by Contract for Python. In: *APSEC*, IEEE Computer Society, 1997, S. 213–219. – ISBN 0-8186-8271-X

[Pobar 2005] Pobar, Joel: Reflection: Dodge Common Performance Pitfalls to Craft Speedy Applications. In: *MSDN Magazine* (2005), July

[Polze und Schult 2004] Polze, Andreas ; Schult, Wolfgang: Neue Werkzeuge braucht das Land - Aspektorientierte Programmierung im .NET-Komponentenframework. In: *IT FOKUS* (2004), S. 31–38. – ISSN 0940-6352

[Puzovic 2005] Puzovic, Milos: *Extesible optimisation framework for .NET virtual machine*, Imperial College London, Masterarbeit, 2005

[Rasche 2002] Rasche, Andreas: *Dynamische (Re-)Konfiguration verteilter Systeme*, Humboldt Universität zu Berlin, Masterarbeit, Sep. 2002

[Rasche 2008] Rasche, Andreas: *Ausführung und Entwicklung adaptiver komponentenbasierter Anwendungen*, Fachbereich Informatik der Mathematisch-Naturwissenschaftlichen Fakultät der Universität Potsdam, Dissertation, 2008

[Rasche und Schult 2007] Rasche, Andreas ; Schult, Wolfgang: Dynamic Updates of Graphical Components in the .NET Framework. In: *Workshop on Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (GI/ITG-Tagung Kommunikation in Verteilten Systemen), Bern, Schweiz*, Mar. 2007

[Rasche u. a. 2005] Rasche, Andreas ; Schult, Wolfgang ; Polze, Andreas: Self-Adaptive Multithreaded Applications - A Case for Dynamic Aspect Weaving. In: *4th Workshop on Adaptive and Reflective Middleware (ARM 2005), Grenoble, France*, Nov. 2005

[red FIVE labs 2008] red FIVE labs: *red Five labs Homepage*. www.redfivelabs.com. 2008

[Reichhart 2005] Reichhart, Stefan: A Prototype of Traits for C# / University of Bern. URL http://www.iam.unibe.ch/~scg/Archive/Projects/Reic05a.pdf, 2005. – Informatikprojekt

[Reina u. a. 2004] Reina, A. M. ; Torres, J. ; Toro, M.: *Separating concerns by means of UML-profiles and metamodels in PIMs.* 5th Aspect-Oriented Modeling Workshop. 2004

[Richter 2007] Richter, Daniel: *Aspektorientierte Programmierung (mit Rapier-Loom.NET)*. http://www.mycsharp.de/wbb2/thread.php?threadid=32405. 2007

[Sadjadi u. a. 2004] Sadjadi, S. M. ; McKinley, Philip K. ; Cheng, Betty H. ; Stirewalt, R.E. K.: TRAP/J: Transparent Generation of Adaptable Java Programs. In: *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*. Agia Napa, Cyprus, Oct. 2004

[Safonov u. a. 2006] Safonov, Vladimir ; Gratchev, Mikhail ; Grigoryev, Dmitry ; Maslennikov, Alexander: Aspect.NET — aspect-oriented toolkit for Microsoft.NET based on Phoenix and Whidbey. In: *Proceedings of .NET Technologies 2006*, Science Press, Union Agency, Plzen, 2006, S. 19–29. – ISBN 80-86943-10-0

[Sakuda 2004] Sakuda, Julie A.: *A Comparative Journey Into 3 AOP Implementations with C#.NET*. http://www2.hawaii.edu/~jsakuda/AOP/aop_comparative.pdf. 2004

[Schärli u. a. 2003] Schärli, Nathanael ; Ducasse, Stéphane ; Nierstrasz, Oscar ; Black, Andrew: Traits: Composable Units of Behavior. In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)* Bd. 2743, Springer Verlag, Juli 2003, S. 248–274. – URL http://www.iam.unibe.ch/~scg/Archive/Papers/Scha03aTraits.pdf. – ISBN 978-3-540-40531-3

[Schauerhuber u. a. 2006] Schauerhuber, A. ; Schwinger, W. ; Kapsammer, E. ; Retschitzegger, W. ; Wimmer, M.: *Towards a Common Reference Architecture for Aspect-Oriented Modeling.* Workshop on Aspect-Oriented Modeling at AOSD. 2006

[Schöbel 2005] Schöbel, Michael: *Effiziente Implementierung des TupleSpace-Ansatzes für .NET*, Hasso-Plattner-Institut für Softwaresystemtechnik, Masterarbeit, 2005

[Schmidmeier u. a. 2003] Schmidmeier, Arno ; Hanenberg, Stefan ; Unland, Rainer: *Implementing Known Concepts in AspectJ.* 3. Workshop Aspekt-Orientierung der GI-Fachgruppe 2.1.9, Essen, Germany. Mar. 2003. – URL citeseer.ist.psu.edu/571060.html

[Schmied 2006] Schmied, Fabian: *AOF-Blog*. http://twoday.tuwien.ac.at/fcs/topics/AOF. 2006

[Schult 2005] SCHULT, Wolfgang: *Invited Talk: Rapier-LOOM.NET — A Dynamic Aspect Weaver for .NET*. AOP Goes .NET Workshop, Microsoft Research. Nov. 2005

[Schult 2007] SCHULT, Wolfgang: *Invited Talk: Design by Contract*. Prio Conference Baden-Baden. Nov. 2007

[Schult 2008] SCHULT, Wolfgang: *LOOM.NET Documentation*. Version 2.5. Hasso-Plattner-Institute, Potsdam: , 2008

[Schult und Polze 2002a] SCHULT, Wolfgang ; POLZE, Andreas: Aspect-Oriented Programming with C# and .NET. Los Alamitos, CA, USA : IEEE Computer Society, 2002, S. 241–248. – ISBN 0-7695-1558-4

[Schult und Polze 2002b] SCHULT, Wolfgang ; POLZE, Andreas: Dynamic Aspect-Weaving with .NET. In: *GI-Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen*. TU Berlin, Germany, Nov. 2002

[Schult und Polze 2003] SCHULT, Wolfgang ; POLZE, Andreas: Speed vs. Memory Usage - An Approach to Deal with Contrary Aspects. In: *The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at the International Conference on Aspect-Oriented Software Development*. Boston, Massachusetts, Mar 2003

[Schult u. a. 2003] SCHULT, Wolfgang ; TRÖGER, Peter ; POLZE, Andreas: *Component Programming with .NET - Exploiting .NET Metadata for Aspect-Oriented Programming*. Workshop on .NET: The Programmer's Perspective, ECOOP'03. 2003

[Seovic und Schult 2005] SEOVIC, Aleksandar ; SCHULT, Wolfgang: *Integration von LOOM.NET in Spring.NET*. Persönliche Kommunikation. 2005

[Shalit 1996] SHALIT, Andrew: *The Dylan reference manual : the definitive guide to the new object-oriented dynamic language*. 1st edition. Apple Computer, Inc., 1996. – ISBN 0-201-44211-6

[Shudo 2005] SHUDO, Kazuyuki: *Performance Comparison of Java/.NET Runtimes*. `http://www.shudo.net/jit/perf/`. 2005

[Simon und Stapf 2002] SIMON, Raphael ; STAPF, Emmanuel: Full Eiffel on the .NET Framework. In: *MSDN library* (2002), July. – URL `http://msdn2.microsoft.com/en-us/library/ms973898.aspx`

[Smith 1982] SMITH, B.C.: *Procedural Reflection in Programming Languages*, Mass. Inst. of Technology, Dissertation, Januar 1982

[Spinczyk und Lohmann 2007] SPINCZYK, Olaf ; LOHMANN, Daniel: The design and implementation of AspectC++. In: *Knowl.-Based Syst.* 20 (2007), Nr. 7, S. 636–651

[Spring.NET Group 2008] SPRING.NET GROUP: *Spring.NET Application Framework*. `http://springframework.net/`. 2008

[Stall 2005] STALL, Mike: *Debugging Support for AOP*. AOP Goes .NET Workshop, Microsoft Research. November 2005

[Steele 1990] STEELE, Guy L.: *Common Lisp the Language*. 2nd edition. Digital Press, 1990. – ISBN 1-55558-041-6

[Steimann 2006] Steimann, Friedrich: The paradoxical success of aspect-oriented programming. In: *SIGPLAN Not.* 41 (2006), Oct., Nr. 10, S. 481–497. – URL http://dx.doi.org/10.1145/1167515.1167514. – ISSN 0362-1340

[Stein u. a. 2002] Stein, Dominik ; Hanenberg, Stefan ; Unland, Rainer: A UML-based aspect-oriented design notation for AspectJ. In: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA : ACM Press, 2002, S. 106–112. – URL http://fparreiras/papers/UMLADOAspectJ.pdf. – ISBN 1-58113-469-X

[Stroustrup 1998] Stroustrup, Bjarne: *Die C++ Programmiersprache*. 3. Addison-Wesley-Longman, 1998. – ISBN 3-8273-1296-5

[Stutz u. a. 2003] Stutz, David ; Neward, Ted ; Shilling, Geoff: *Shared Source CLI - Exploring Microsoft's Rotor & the ECMA CLI*. O'Reilly & Associates, Inc., Mar. 2003. – ISBN 0-596-00351-X

[Sullivan u. a. 2005] Sullivan, K. ; Griswold, W.G. ; Song, Y. ; Chai, Y. ; Shonle, M. ; Tewari, N. ; Rajan, H.: On the Criteria to be Used in Decomposing Systems into Aspects. In: *Symp. Foundations of Software Engineering joint with the European Software Engineering Conf. (ESEC/FSE 2005)*, ACM Press, 2005

[Suvée u. a. 2003] Suvée, Davy ; Vanderperren, Wim ; Jonckers, Viviane: JAsCo: an aspect-oriented approach tailored for component based software development. In: *AOSD*, 2003, S. 21–29

[Suzuki und Yamamoto 1999] Suzuki, Junichi ; Yamamoto, Yoshikazu: Extending UML with Aspects: Aspect Support in the Design Phase. In: Moreira, Ana M. D. (Hrsg.) ; Demeyer, Serge (Hrsg.): *ECOOP Workshops* Bd. 1743, Springer, 1999, S. 299–300. – URL http://dblp.uni-trier.de/db/conf/ecoopw/ecoopw99.html#SuzukiY99. – ISBN 3-540-66954-X

[Szathmary 2002] Szathmary, Viktor: *Barter – beyond Design by Contract*. 2002. – URL {http://barter.sourceforge.net}

[Szyperski 2002] Szyperski, Clemens: *Component Software*. 2nd edition. Addison-Wesley (ACM-Press), 2002. – ISBN 0-201-74572-0

[Tarr u. a. 1999] Tarr, Peri ; Ossher, Harold ; Harrison, William ; Sutton, Jr. Stanley M.: N degrees of separation: multi-dimensional separation of concerns. In: *Proc. Int'l Conf. Software Engineering (ICSE)*, IEEE Computer Society Press, 1999, S. 107–119

[Teitelman 1966] Teitelman, W.: *PILOT: A STEP TOWARDS MAN-COMPUTER SYMBIOSIS*. Cambridge, MA, USA, Massachusetts Institute of Technology, Dissertation, 1966

[The Eclipse Foundation 2008a] The Eclipse Foundation: *AJDT: AspectJ Development Tools*. http://www.eclipse.org/ajdt/. 2008

[The Eclipse Foundation 2008b] The Eclipse Foundation: *Eclipse Homepage*. http://www.eclipse.org/. 2008

[Torgersen 2004] Torgersen, Mads: The Expression Problem Revisited. In: Odersky, Martin (Hrsg.): *ECOOP* Bd. 3086, Springer, 2004, S. 123–143. – ISBN 3-540-22159-X

[Tourwé u. a. 2003] Tourwé, Tom ; Brichau, Johan ; Gybels, Kris: On the Existence of the AOSD-Evolution Paradox. In: *Proceedings of the AOSD 2003 Workshop on Software Engeneering Properties of Languages for Aspect Technologies*, Mar. 2003

[Troeger 2002] Troeger, Peter: *Aspect-oriented object and component migration*, Humboldt Universität zu Berlin, Masterarbeit, Sep. 2002

[Tsang u. a. 2004] Tsang, Shiu L. ; Clarke, Siobhán ; Baniassad, Elisa L. A.: An Evaluation of Aspect-Oriented Programming for Java-Based Real-Time Systems Development. In: *ISORC*, IEEE Computer Society, 2004, S. 291–300. – ISBN 0-7695-2124-X

[Vanderperren u. a. 2005] Vanderperren, Wim ; Suvée, Davy ; Verheecke, Bart ; Cibrán, María A. ; Jonckers, Viviane: Adaptive programming in JAsCo. In: Mezini, Mira (Hrsg.) ; Tarr, Peri L. (Hrsg.): *AOSD*, ACM, 2005, S. 75–86. – ISBN 1-59593-042-6

[Vanegas u. a. 1998] Vanegas, Rodrigo ; Zinky, John A. ; Loyall, Joseph P. ; Karr, David ; Schantz, Richard E. ; Bakken, David E.: QuO's Runtime Support for Quality of Service in Distributed Objects. In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98).* The Lake District, England, Sep. 1998, S. 15–18

[Wadler 1998] Wadler, Philip: *The expression problem.* Eintrag in der „Java Genericity mailing list". 1998. – URL http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[Wagner 2003] Wagner, Jim: *The Push For Aspect-Oriented Programming.* http://www.internetnews.com/dev-news/print.php/3106021. Nov. 2003

[Weiser 1991] Weiser, M.: The Computer for the 21st Century. In: *Scientific American* 43 (1991), Sep., Nr. 3, S. 66–75. – URL {http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html}

[Wermelinger 1997] Wermelinger, Michel: A Hierarchic Architecture Model for Dynamic Reconfiguration. In: *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems.* 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA : IEEE, 1997, S. 243–254. – URL {citeseer.nj.nec.com/wermelinger97hierarchic.html}

[Wigley und Wheelwright 2003] Wigley, Andy ; Wheelwright, Stephen: *.NET Compact Framework.* Microsoft Press, 2003

[Wikipedia 2008a] Wikipedia: *Aspect-oriented programming — Wikipedia, The Free Encyclopedia.* 2008. – URL http://en.wikipedia.org/w/index.php?title=Aspect-oriented_programming&oldid=229729494. – [Online; zugegriffen am 6. August 2008]

[Wikipedia 2008b] Wikipedia: *List of CLI languages — Wikipedia, The Free Encyclopedia.* 2008. – URL http://en.wikipedia.org/w/index.php?title=List_of_CLI_languages&oldid=230209773. – [Online; zugegriffen am 10. August 2008]

[Xu 2007] Xu, Yan: Phoenix Academic Program Research Highlights / Microsoft Research. Juli 2007. – Forschungsbericht. http://research.microsoft.com/Phoenix/hlight/phoenix_report/phoenix_booklet_0625_ebook.pdf