Diplomarbeit zum Thema

Aspektorientierte Softwareentwicklung mit .NET

von

Peter Zukowski

Zur Erlangung des akademischen Grades Diplom-Informatiker (FH),
vorgelegt am Fachbereich Mathematik, Naturwissenschaften
und Informatik der Fachhochschule Gießen-
Friedberg im September 2004

Betreuung:

Professor Dr. Wolfgang Henrich, Referent
Professor Dr. Wolfgang Schmitt, Korreferent

We think of the current state of AOP research
as analogous to that of OOP 20 years ago.

Gregor Kiczales et al., 1997

AOP quite possibly represents the next big step
in the evolution of programming methodologies.

Ramnivas Laddad, 2002

Vorwort

Die objektorientierte Softwareentwicklung genießt heutzutage einen sehr angesehenen Status und ist in den Augen der meisten Entwickler sicherlich ein selbstverständliches sowie unentbehrliches Werkzeug ihrer Kunst. Gegenwärtig werden Softwaresysteme immer komplexer und ihre Entwicklung ist in vielen Fällen trotz Anwendung objektorientierter Methodik immer noch ein mühsamer und fehleranfälliger Prozess. Zahlreiche Forschungsgruppen sind deshalb damit beschäftigt, Verbesserungen oder sogar komplett neue Ansätze zu entwickeln, die zu einer noch klareren Dekomposition komplexer Sachverhalte und damit einer erfolgreicheren Softwareentwicklung führen. Viele dieser Ansätze bauen auf der Objektorientierung auf, indem sie diese in deren Schwachstellen aufbessern. Ein solcher Ansatz ist die aspektorientierte Softwareentwicklung.

Obwohl selbst ihre Urväter betonen, dass die aspektorientierte Softwareentwicklung die objektorientierte keinesfalls ersetzen soll, ist jetzt schon von einem neuen Paradigma die Rede. Die aspektorientierte Softwareentwicklung sowie zahlreiche Forschungen und Entwicklungen auf verwandten Gebieten führten zu neuen Techniken, die sich sogar völlig von der objektorientierten Methodik distanzieren. Denn diese ist trotz ihrer massiven Verwendung letztendlich nur eine Betrachtungsweise, mit der man sehr viele, aber längst nicht alle Probleme zufrieden stellend lösen kann.

Wir müssen uns deshalb ins Bewusstsein rufen, dass wir mit einer Weltanschauung leben, die nicht zwangsweise die ultimative sein muss…

Inhaltsverzeichnis

Kapitel 1: Zum Dokument

Dieses Kapitel befasst sich noch nicht mit der eigentlichen Thematik, sondern dient als Einführung in dieses Dokument. Nachdem die Ziele dieser Diplomarbeit bestimmt wurden, wird ein Umriss über ihren Verlauf gegeben sowie Vereinbarungen über Formulierungen getroffen.

1.1 Motivation und Ziele

Obwohl die Anzahl der Publikationen über aspektorientierte Softwareentwicklung mittlerweile die Zahl 1000 überschreiten dürfte (Das Bibliografie-Verzeichnis in [Film03] enthält 800 Einträge.), lässt die Menge der Veröffentlichungen, die das Thema von allen Seiten forschend und hinterfragend durchleuchten, eher zu wünschen übrig. Die bisher veröffentlichten Bücher wie [Ladd03] konzentrieren sich vorwiegend auf die Java-Implementierung AspectJ. Dieses Dokument soll diesem kleinen Notstand ein wenig abhelfen, indem es den umgekehrten Weg geht. Es soll sich dem Phänomen aspektorientierte Softwareentwicklung von der theoretischen Seite nähern und schließlich der hinter ihm steckenden Technik sorgfältig auf den Grund gehen. Dabei soll das .NET-Framework als Zielplattform evaluiert werden. Die aus dieser Analyse gewonnen Erkenntnisse werden in Form einer Software-Implementierung für die .NET-Plattform umgesetzt.

Insbesondere wird Folgendes angestrebt:

- Darlegung des Zwecks und der Ziele aspektorientierter Softwareentwicklung
- Vorstellung technischer Mittel der aspektorientierten Softwareentwicklung
- Untersuchung verwandter Projekte und existierender Implementierungen
- Erforschung der durch die .NET-Plattform gegebenen Mittel auf Lösungen für die Problematik der aspektorientierten Softwareentwicklung
- Entwurf und Implementierung einer Software zur Realisierung aspektorientierter Funktionalität unter .NET
- Dokumentation der Implementierung und Erstellung von Beispielprojekten
- Bewertung der Implementierung und Schlussfolgerungen

Trotz der Komplexität des Themas habe ich mich bemüht, dieses Dokument so zu verfassen, dass es auch für den Leser mit wenigen Vorkenntnissen verständlich ist. Aus eigener Erfahrung weiß ich wie wertvoll und wohltuend es ist, auf ein Dokument zurückgreifen zu können, dass die Problematik eines Themas sorgfältig und intuitiv vermittelt. Der Leser sollte jedoch mit der .NET-Plattform und der Programmiersprache C# vertraut sein.

1.2 Überblick über die Kapitel

Kapitel 1: Zum Dokument
Die Einleitung dient dem Überblick über die Ziele und den Aufbau dieser Diplomarbeit sowie der Vereinbarung von Konventionen.

Kapitel 2: Die Problematik
In Kapitel 2 werden Sinn und Zweck der aspektorientierten Softwareentwicklung dargelegt. Verschiedene Modelle sowie Anwendungsfälle zur Veranschaulichung und Verdeutlichung werden präsentiert.

Kapitel 3: Die Technologien
Anhand der gewonnenen Erkenntnisse über die Problematik werden Anforderungen und Ziele für deren Lösung erarbeitet. Verschiedene Lösungsansätze und Techniken werden diskutiert und anhand der zuvor entworfenen Richtlinien evaluiert.

Kapitel 4: Die Implementierungen
Kapitel 4 gibt einen Überblick über den heutigen Stand der aspektorientierten Softwareentwicklung, indem derzeit relevante Softwarelösungen untersucht werden.

Kapitel 5: Realisierung
Der Entwurf einer Software zur Realisierung aspektorientierter Softwareentwicklung unter .NET wird beschrieben, nachdem zuvor die Ziele dieser Implementierung definiert worden sind. Anhand des entworfenen Modells wird die Implementierung der Software durchgeführt und in diesem Kapitel geschildert.

Kapitel 6: Fazit
Eine Schlussfolgerung aus der Analyse des Themas sowie der Realisierung der Eigenimplementierung schließt dieses Dokument ab.

1.3 Konventionen

Der Begriff „Aspektorientierte Softwareentwicklung“ wird im Folgenden mit „AOSE“ abgekürzt. Dementsprechend wird für den Begriff „Aspektorientierte Programmierung“ die Abkürzung „AOP“ benutzt. Alle Abkürzungen in diesem Dokument werden in Anhang 1 erläutert.

Der größte Teil der Literatur zum Thema AOSE liegt in englischer Sprache vor. Daher wird für fachliche Begriffe zunächst eine sinnvolle Übersetzung gesucht und der Originalbegriff in Klammern dahinter geschrieben. Wenn eine Übersetzung nicht sinnvoll erscheint, wird die originale Formulierung verwendet. Code und Diagramme, die mit Code in Verbindung stehen, sind in englischer Sprache verfasst.

Alle Ausführungen und Codefragmente im Zusammenhang mit .NET beziehen sich auf die .NET-Framework-Version 1.1.

Alle UML-Diagramme in diesem Dokument halten sich an das deutsche UML-Standardwerk [Oest01].

Die in den Listings verwendete Sprache ist C#.

Kapitel 2: Die Problematik

Dieses Kapitel dient als Einführung in die Problematik der AOSE. Es soll eine Antwort darauf geben, was die AOSE überhaupt ist, welchem Zweck sie dient, welche Möglichkeiten durch sie eröffnet werden und wo sie angewendet werden kann. Zum besseren Verständnis sowie zur Verdeutlichung der Bedeutung des Themas werden Modellvorstellungen und Fallbeispiele präsentiert.

2.1 Was ist AOSE?

2.1.1 Dekomposition

„Programmieren bedeutet das Lösen von Problemen der realen Welt mit Hilfe des Computers“ [Page94]. Die reale Welt besteht aus hochkomplexen, für den normalen Menschen in aller Vollständigkeit nur schwer fassbaren Systemen [Booc94]. Deshalb bedient sich der Mensch in jeder nur vorstellbaren technischen Disziplin verschiedener Denkmuster, welche die Komplexität reduzieren oder zumindest verbergen. Bereits in den Anfängen der Programmierung war die Vereinfachung von Komplexität ein wichtiger Bestandteil; sie bestand jedoch bestenfalls aus der funktionalen Zerlegung eines Programms. Die Erfindung des Transistors in den Jahren 1948/49 und der integrierten Schaltkreise im Jahr 1958 führte zu einem schlagartigen Technologiesprung, der die Größe und Komplexität von Software enorm ansteigen ließ. Dies zog notwendigerweise die Entstehung von Software-Engineering als Profession nach sich, welche einerseits den Softwareentwicklungsprozess in Projektphasen zerlegte und andererseits neue Konzepte der Datenmodellierung erschuf. In den frühen 70gern entstand schließlich die Idee, die reale Welt durch abstrakte Objekte nachzubilden. Die objektorientierte Softwareentwicklung ermöglicht eine Betrachtungsweise, welche die Komplexität von Gegebenheiten mit Hilfe von Mitteln der Abstraktion, Kapselung, Modularität und Hierarchie zerlegt [Booc94].

Die Dekomposition eines Problems in einfachere, greifbare logische Bestandteile begleitet die Informatik seit ihrer Geburt durch jedes Programmierparadigma. E.W. Dijkstra hat dieses Leitprinzip der Softwaretechnik mit dem Schlagwort „*Trennung von Sachverhalten*“

(„separation of concerns“) bezeichnet [Dijk76]. Den Begriff des Sachverhalts („concern“)[1] definiert R. Laddad folgendermaßen: “A concern is a particular goal, concept, or area of interest” [Ladd02].

Die Objektorientierung realisiert das Prinzip der Trennung von Sachverhalten durch Dekomposition in Objekte. Dies ist gewiss eine vernünftige und nachvollziehbare Denkweise, besteht die reale Welt doch schließlich bis ins kleinste Detail aus Objekten. Jedes Objekt hat eine wohldefinierte Schnittstelle und kapselt eine bestimmte Funktionalität zur Erreichung eines konkreten Ziels [Booc94]. Der Code eines Objekts erreicht idealerweise eine *hohe Kohäsion* (Die Funktionalität der Methoden und Felder ist innerhalb der Klasse stark verwandt.) sowie eine *Entkopplung* von anderen Objekten [Booc94].

2.1.2 Verstreute Sachverhalte

Es gibt jedoch Sachverhalte, die man mit dem objektbasierten Paradigma nicht darstellen kann. Dieses Problem soll im Folgenden an einem Beispiel anschaulich gemacht werden.

Gegeben sei ein vereinfachtes Softwaremodell mit der folgenden Klassenhierarchie:

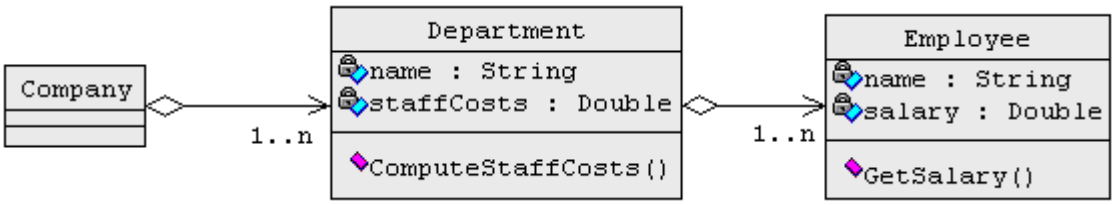

Bild 2.1 Klassendiagramm für eine Firma mit Abteilungen und Mitarbeitern

Eine Firma besteht aus einer oder mehreren Abteilungen und diese wiederum bestehen aus je einem oder mehreren Mitarbeitern. Die Abteilung kann ihre Personalkosten in der Methode `Department.ComputeStaffCosts` berechnen, indem sie mit Hilfe der Methode `Employee.GetSalary` die Gehälter aller Mitarbeiter aus den Feldern `Employee.salary` ermittelt. Die Summe aller Gehälter wird in das Feld `Department.staffCosts` geschrieben.

In diesem Modell ist jeder Sachverhalt in einer Klasse gekapselt. Er ist von allen anderen Sachverhalten eindeutig entkoppelt und dient zur Erfüllung einer wohldefinierten Aufgabe. Diese Klassenhierarchie ist ebenso trivial wie wohlgeformt und mit den Werkzeugen der objektorientierten Softwaremodellierung intuitiv darstellbar.

[1] „Concern“ wird in diesem Dokument mit „Sachverhalt“ übersetzt. In anderen deutschsprachigen Quellen findet man auch „Angelegenheit“, “Interesse“ oder „Belang“.

Nehmen wir nun an, diese Klassenhierarchie sei Teil eines Produktes, z.B einer Applikation zur Personalverwaltung. Man würde die Applikation veranlassen wollen, ihren aktuellen Stand in einer Binärdatei zu speichern. Dazu würde man die Belegung der Felder aller Objekte in einen seriellen Datenstrom übertragen wollen. Jede Klasse würde somit eine entsprechende Methode zur Serialisierung bekommen:

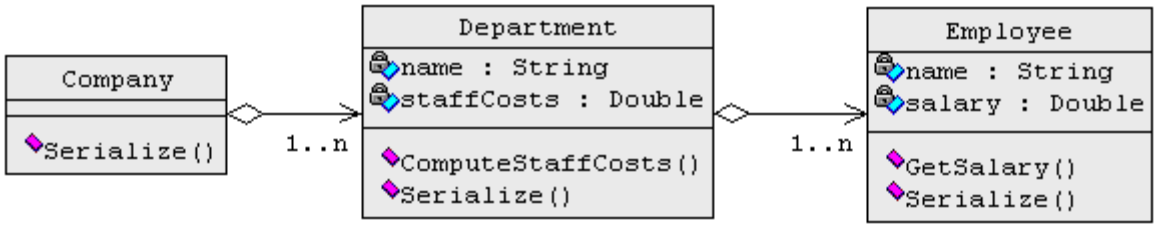

Bild 2.2 Klassendiagramm von Bild 2.1 erweitert um die Funktionalität der Serialisierung

Diese Lösung ist zwar in einem Klassendiagramm darstellbar. Es wird jedoch schnell klar, dass sie das für die Softwaremodellierung so wichtige Prinzip der Trennung der Sachverhalte verletzt, und zwar an folgenden Stellen:

- **Mangelnde Kohäsion:** Die Klassen in Bild 2.2 implementieren Sachverhalte, die konzeptionell gesehen nichts miteinander zu tun haben. Sie kapseln Funktionalitäten zweier unterschiedlicher *Problemfelder* („problem domains“, „domains of interest“), nämlich das der Personalverwaltung und das der Verwaltung der Applikation.

- **Streuung:** Die Serialisierung taucht an mehreren Stellen im Softwaremodell auf, denn sie ist auf die Klassen `Company`, `Department` und `Employee` verstreut.

Gleichzeitig ist jedoch folgende Eigenschaft für die Serialisierung gegeben:

- **Sie bildet logisch gesehen eine Einheit:** Obwohl ihre Methoden an verschiedenen Stellen lokalisiert sind, dienen sie einem Zweck, kapseln also zusammen eine bestimmte Funktionalität zur Erreichung eines konkreten Ziels.

Die Serialisierung bildet somit einen wohl definierten *Sachverhalt* und sollte deshalb als modulare Einheit entworfen werden können. Dass dies nicht geht liegt nur daran, dass sie sich zum restlichen Modell *orthogonal* verhält und man mit der objektorientierten Methodik diese Orthogonalität nicht ausdrücken kann. Der Sachverhalt der Serialisierung *durchkreuzt* geradezu das Softwaremodell und wird deshalb von G. Kiczales el al. als *„crosscutting concern“* bezeichnet [Kicz97]. In diesem Dokument soll hierfür der Begriff des *verstreuten Sachverhalts* verwendet werden.

2.1.3 „Zerstreuung“ und „Verfilzung“ von Code

In der heutigen Softwareentwicklung ist es eine Selbstverständlichkeit, dass es Sachverhalte gibt, die über viele Stellen zerstreut sind. Es sind Sachverhalte, auf die man nicht verzichten kann, die aber durch ihre Anwesenheit die Leitprinzipien der Modularisierung und Kapselung ernsthaft verletzen und dadurch die Software unübersichtlich und fehleranfällig machen. Verstreute Sachverhalte führen zu vielerlei Schwierigkeiten:

- **Mangelnde Übersicht:** Es wird immer schwieriger, Sachverhalte auseinander zuhalten.
- **Verschmutzung von Code:** Der Code für die eigentliche Geschäftslogik wird durch den verstreuten Code verschmutzt.
- **Aufblähung von Code:** Dadurch wird der Geschäftscode unter Umständen sogar um ein Vielfaches aufgebläht.
- **Redundanz im Code:** Code des verstreuten Sachverhaltes kommt in gleicher oder ähnlicher Form an verschiedenen Stellen mehrfach vor.
- **Mangelnde Wartbarkeit:** Der Code eines verstreuten Sachverhaltes ist nicht gekapselt und daher nur sehr schwer zu warten.
- **Mangelnde Wiederverwendbarkeit:** Gleiches gilt für seine Wiederverwendung. Aber auch die Wiederverwendbarkeit des Geschäftscodes wird durch seine Verschmutzung stark beeinträchtigt.

Möchte man Änderungen an einem verstreuten Sachverhalt vornehmen, z.B. in unserem Beispiel die Serialisierung in eine XML-Datei statt wie bisher in eine Binärdatei[1], so müsste man Code an vielen Stellen mehrfach ändern. Dies führt schnell zu Unordnung und ist besonders bei großen und ständiger Dynamik unterliegenden Projekten eine häufige Fehlerquelle. Solcher Code ist weder gut lesbar, noch besonders geeignet zur Wiederverwendung. Mehr noch, er ist redundant, denn er kommt an verschiedenen Stellen in gleicher oder ähnlicher Form mehrfach vor. Kiczales et al. bezeichnen dieses Phänomen als *Zerstreuen von Code* („code scattering“) [Kicz97]. Das Durcheinanderbringen von Code mehrerer Sachverhalte nennen sie *Verfilzung von Code* („code tangling“) [Kicz97].

2.1.4 Aspekte

Es wäre viel logischer, konzeptionell schöner, aber auch viel weniger fehleranfällig, wenn man verstreute Sachverhalte von der Geschäftslogik entkoppeln und für sich alleine gekapselt modellieren könnte. Dies wird durch die Tatsache erleichtert, dass meistens

[1] XML-Dateien bieten gegenüber Binärdateien den Vorteil der Interoperabilität. In der .NET-Welt lässt sich mit XML, SOAP und WDSL Software über Rechnersystem-, Betriebssystem- und Sprachgrenzen hinweg einsetzen [West01].

einzelne verstreute Sachverhalte von anderen verstreuten Sachverhalten bereits entkoppelt sind.

Die AOSE ist eine Technik, die eine Lösung für genau dieses Problem bietet. Sachverhalte, die an mehreren Stellen verstreut im Softwaremodell des Problemfeldes liegen, werden aus diesem herausgenommen und für sich gekapselt an zentraler Stelle entworfen, wie Bild 2.3 zeigt:

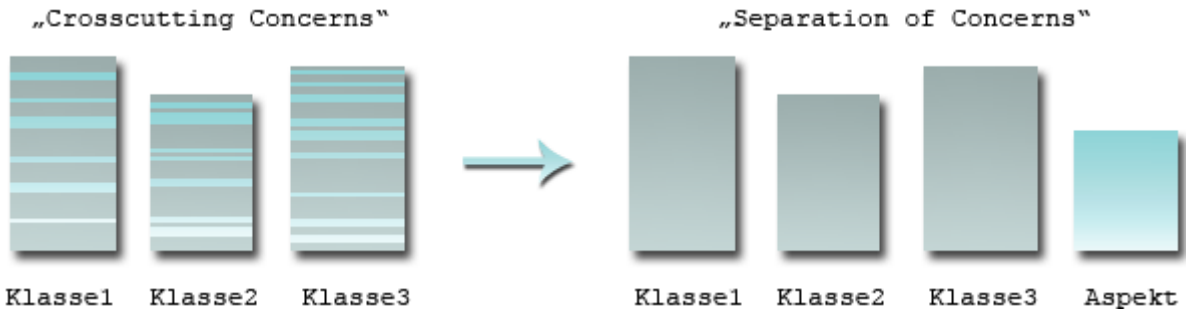

Bild 2.3 Zweck der AOSE

Die modulare Einheit, in der diese verstreuten Sachverhalte gekapselt werden, bezeichnet man als *Aspekt* („aspect"). Aspekte stehen im gewissen Sinne mit Klassen auf einer Ebene, denn beide bilden zusammenhängende, wohl definierte und wieder verwendbare modulare Einheiten. Während die Objektorientierung jedoch versucht, Zusammenhänge zwischen Klassen zu finden und diese *vertikal* in einer höheren Ebene der Vererbungshierarchie anzusiedeln, versucht die AOSE, zusammenhängende Sachverhalte aus Objekten *horizontal* auszulagern und in Form von Aspekten zu formulieren [Cacm01a]. Bild 2.4 verdeutlicht den Begriff des Aspekts:

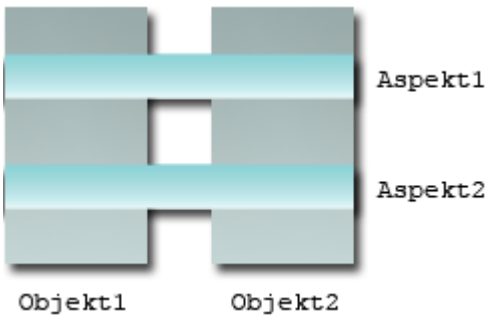

Bild 2.4 Objekte als modulare Einheiten zusammenhängender und Aspekte als modulare Einheiten verstreuter Sachverhalte

2.1.5 Geschichtliches

Die AOSE entstand aus einem von Gregor Kiczales geleiteten Forschungsprojekt am Palo Alto Research Center [Parc04]. Kiczales veröffentlichte seine Ergebnisse 1996 und 1997 in

[Kicz96] und [Kicz97]. Nach der Entwicklung einiger prototypischer AOP-Sprachen begann 1998 die Arbeit an der heute sehr verbreiteten Java-AOP-Sprache AspectJ. AspectJ verkörpert die von Kiczales geprägten Ideen und gilt heute als Muster einer AOSE-Implementierung. Seit 2002 findet jährlich die „Aspect-Oriented Software Development Conference“ statt. Die mit ihr verbundenen Ausarbeitungen, Workshops und Tutorials werden auf der Website [AOSD04] kommentiert, die in der heutigen Literatur als Einstiegspunkt in die AOSE gilt.

2.2 Modelle

Modelle dienen zur Veranschaulichung von komplexen Sachverhalten. Dieses Kapitel stellt einige Modelle vor, um den Begriff der AOSE zu verdeutlichen. Darüber hinaus basieren viele der im Laufe dieser Arbeit behandelten Techniken und Implementierungen auf den hier vorgestellten Modellen.

2.2.1 Das Prismenmodell

Nach R. Laddad lässt sich ein Softwaresystem als eine kombinierte Implementierung von diversen Sachverhalten betrachten [Ladd02]. So ist ein wichtiger Schritt in der Entwicklung eines Systems, ein Gemisch von Anforderungen in ein Spektrum von Sachverhalten umzuwandeln. Alle Sachverhalte sollten getrennt implementiert und anschließend wieder zum Gesamtsystem komponiert werden. Dieser Vorgang lässt sich gut mit zwei Prismen beschreiben [Ladd02], wie Bild 2.5 zeigt:

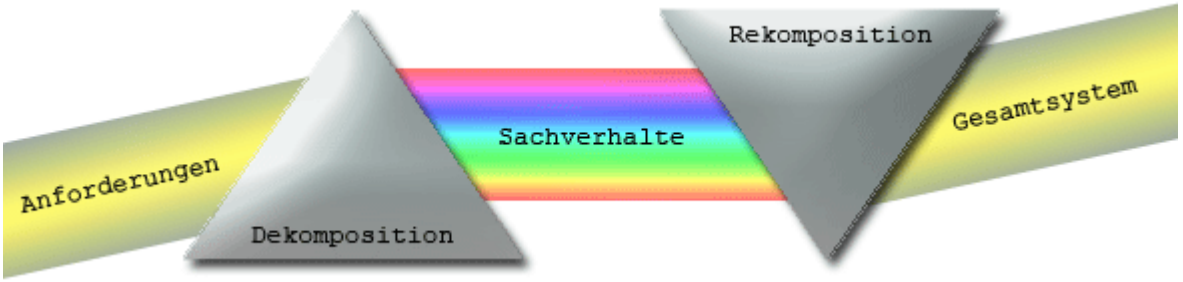

Bild 2.5 Dekomposition und Rekomposition mit dem Prismenmodell

2.2.2 Das Dimensionsmodell

Maschinen- und Quellcode bestehen aus einer sequentiellen Folge von Bits bzw. Zeichen. Bezogen auf Dimensionen, ist der Code einer Applikation immer eindimensional. Ein Raum mit nur einer Dimension hat jedoch seine Grenzen: Man kann die Komplexität mancher Probleme nicht mit nur einer Dimension darstellen, besteht unsere Welt doch aus mehreren von ihnen. Die in der heutigen Zeit etablierte Rechnerarchitektur sowie die herkömmlichen

Programmierparadigmen zwingen uns, alle Anforderungen und alle Probleme auf eine Dimension zu projizieren und in dieser einen Dimension zu implementieren. [Tarr99] bezeichnet diese Dimension als *„dominante Dimension"* („dominant dimension") und den Mangel an Mitteln, Sachverhalte voneinander zu trennen, mit *„Tyrannei der dominanten Dekomposition"* („the tyranny of the dominant decomposition"). R. Laddad macht deutlich, dass die Menge der Anforderungen an eine Software ein n-Dimensionaler Raum ist [Ladd02]. Jede weitere Anforderung bzw. jeder weitere Sachverhalt, der implementiert werden soll, ist eine weitere Dimension. Eine bekannte Umsetzung des Dimensionsmodells ist das Softwarewerkzeug Hyper/J, das später in Abschnitt 4.5.1 behandelt wird.

2.2.3 Das Kontextmodell

Mit dem Zuordnen von Aspekten zu einem Objekt erstellt man eine spezifische Umgebung für das Objekt [Shuk02]. Das Objekt muss diese Umgebung nicht kennen; sein Verhalten wird jedoch durch diese beeinflusst. Man sagt auch, dass das Objekt in einem bestimmten *Kontext* eingebettet ist, der durch die Bereitstellung von *Diensten* bestimmt ist [Shuk02]. Verschiedene Kontexte entstehen durch Anbringen von verschiedenen Aspekten. Bild 2.6 zeigt ein Objekt, das in einen Kontext eingebettet ist, der aus den Diensten „Synchronisation", „Fehlerprotokollierung" und „Persistenz" besteht:

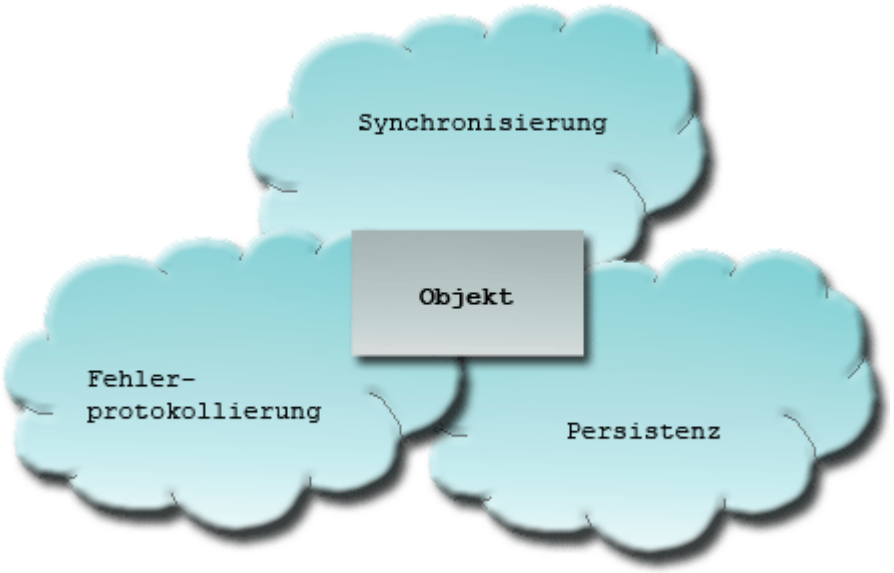

Bild 2.6 In einen bestimmten Kontext eingebettetes Objekt

Eine Umsetzung des Kontextmodells ist aus der Welt von COM+ und .NET als *attributierte Programmierung* („attributed programming" oder „attribute-oriented programming") bekannt [Shuk02]. Attribute und ihr Zusammenhang mit der AOSE werden im Abschnitt 3.4.3 genauer behandelt.

2.3. Fallbeispiel: Tracing

2.3.1 Schritt 1

Anhand eines für die AOSE typischen Fallbeispiels soll durch systematische programmiertechnische Herangehensweise die Problematik der verstreuten Sachverhalte deutlich gemacht werden. Dazu betrachte man die folgende Spezifikation für einen Anwendungsfall: „Für jede Methode in der Applikation soll der Name der Methode ausgegeben werden, wenn diese ausgeführt wird.“ Dies stellt einen trivialen und nützlichen Anwendungsfall dar, in dem aber hinsichtlich OOP-Paradigma einige Probleme wurzeln. Mit Reflection kann man unter .NET sehr leicht den Methodennamen der gerade ausgeführten Methode bestimmen. Allerdings müsste man in jeder Methode der Applikation diesen Code eintragen, was bei größeren und sich ständig verändernden Applikationen praktisch nahezu unmöglich wäre:

```
class SomeClass
{
	public void SomeMethod()
	{
		MethodBase meth = MethodBase.GetCurrentMethod();
		Console.WriteLine(meth.Name);
	}
	public void SomeMethod(int i, double d)
	{
		MethodBase meth = MethodBase.GetCurrentMethod();
		Console.WriteLine(meth.Name);
	}
}
```

Listing 2.1 Tracing mit Reflection

2.3.2 Schritt 2

Man stelle sich nun vor, dass die Spezifikation für den Anwendungsfall geändert wird: „Für jede Methode in der Applikation soll der Name der Methode ausgegeben werden, wenn diese ausgeführt wird, sowie der Name der Klasse, in der sie definiert ist.“ Schnell wird klar, dass derart verstreuter Code zu einer riesigen Last für das gesamte Projekt werden kann. Es liegt nahe, den Sachverhalt des Tracing in eine separate Klasse auszulagern, damit nur diese von eventuellen Änderungen des Anwendungsfalls betroffen wird. Dazu erstellt man die Klasse `Tracer`, deren statische Methoden von überall her aufrufbar sind. Eine davon wäre `TraceMethod` und könnte anhand des übergebenen `MethodBase`-Objekts mit Reflection den Methodennamen sowie den sie definierenden Typ ermitteln:

```
class Tracer
{
	public static void TraceMethod(MethodBase meth)
	{
		Console.WriteLine(meth.DeclaringType + "." + meth.Name);
	}
}

class SomeClass
{
	public void SomeMethod()
	{
		Tracer.TraceMethod(MethodBase.GetCurrentMethod());
	}
	public void SomeMethod(int i, double d)
	{
		Tracer.TraceMethod(MethodBase.GetCurrentMethod());
	}
}
```

Listing 2.2 Kapselung des Tracing-Sachverhaltes in separater Klasse

2.3.3 Schritt 3

Listing 2.2 macht es dank verbesserter Modularisierung möglich, dass die Ausgabe der Methodeninformation an zentraler Stelle geregelt werden kann. Jedoch muss man erkennen, dass auch dieses Vorgehen schnell an seine Grenzen gerät. Denn man stelle sich nun folgende Situation vor:

- Wiederum wird die Spezifikation geändert: Nicht jede Methode der Applikation soll betroffen werden, sondern nur alle öffentlichen Methoden.
- Es kommen neue Komponenten, neue Klassen und neue Methoden hinzu.

Spätestens jetzt wird deutlich, dass das Problem so nicht lösbar ist. Das liegt daran, dass in Listing 2.2 der Sachverhalt des Tracing in Wirklichkeit überhaupt nicht von der restlichen Applikation entkoppelt ist. Denn nach wie vor muss in jeder Methode, die für das Tracing ausgesucht wurde, ein Aufruf zu `TraceMethod` erfolgen. Der Sachverhalt ist zwar zentral gekapselt, hinterlässt im Applikationscode aber immer noch seine Spuren. Es wird nun bewusst, dass das Definieren und Warten von verstreuten Sachverhalten mit dem OOP-Paradigma nur sehr schwer realisierbar ist und sehr schnell zu Instabilität im Code führen kann.

2.3.4 Ausblick

Mit der aspektorientierten Java-Erweiterung AspectJ lässt sich das Tracing-Problem mit Hilfe weniger Codezeilen lösen:

```
aspect TraceAspect
{
	pointcut allPublicMethods():
		execution(public * *(..));

	before():
	allPublicMethods()
	{
		System.out.println("----------------------------");
		System.out.println(thisJoinPoint.getSignature());
	}
}
```

Listing 2.3 Tracing mit AspectJ

Listing 2.3 besagt, dass die Signatur jeder öffentlichen Methode der Applikation ausgegeben werden soll, bevor diese ausgeführt wird. Auf genauere Erklärung soll vorerst verzichtet werden.[1] Hier soll nur deutlich gemacht werden, wie kurz und bündig eine Lösung des Tracing-Problems mit Mitteln der AOSE formuliert werden kann. Dabei bleibt die Existenz des Aspekts für den Applikationscode unbemerkt. Zudem kann der Aspekt sehr schnell und unkompliziert aus der Applikation entfernt werden, ohne an einer unüberschaubaren Anzahl von Stellen einzelne Codezeilen auskommentieren zu müssen.

2.4 Weitere Anwendungsbeispiele

2.4.1 Überblick

In einem Softwaremodell geraten häufig Sachverhalte aus verschiedenen Problembereichen aneinander. In den häufigsten Fällen treffen Sachverhalte der Geschäftslogik mit nicht fachlichen verstreuten Sachverhalten zusammen. Doch auch Probleme mit verstreuten Sachverhalten, die selbst Teil des Fachmodells sind, sind mit der AOSE gut lösbar. Dieser Abschnitt soll einige sehr typische Rollen für Aspekte präsentieren, um die Wichtigkeit und Tragweite der AOSE zu verdeutlichen. Ferner dient eine sorgfältige Analyse von Anwendungsfällen zur Erstellung des Anforderungsprofils an die geplante Eigenimplementierung. Anwendungsfälle für Aspekte können in folgende Kategorien unterteilt werden [AspJ03]:

[1] AspectJ und andere Implementierungen der AOSE werden in Kapitel 4 behandelt.

- **Entwicklungs-Aspekte:** Aspekte, die als notwendige Hilfsmittel im Softwareentwicklungsprozess eingesetzt werden: Tracing, Profiling, Prüfung von Vor- und Nachbedingungen usw.
- **Produktions-Aspekte:** Aspekte, die im fertigen Softwareprodukt systemtechnische Aufgaben übernehmen und nicht dem Fachmodell entspringen: Sicherheits- und Transaktionsmanagement, Synchronisation, Zugriffskontrolle usw.
- **Fachliche Sachverhalte:** Benutzerprofile, Konfigurationsmanagement, Realisierung von Produktionslinien usw.

2.4.2 Entwicklungs-Aspekte

Wie bereits an dem Tracing-Beispiel in Abschnitt 2.3 demonstriert wurde, bieten Aspekte eine nützliche Hilfe für den Entwickler. Mit der Enkopplung des Tracing vom Komponentencode kann auf das mühsame Ein- und Auskommentieren von Meldungen oder auf das Abfragen von Debug-Variablen verzichtet werden. Des Weiteren können Aspekte für *Profiling* eingesetzt werden, indem sie z.B. die Ausführungszeit von Codezeilen messen oder Aufrufe zu Methoden zählen.

Auch der durch B. Meyer in [Meye97] populär gewordene Programmierstil *Design by Contract* kann durch die AOSE entscheidend verbessert werden [AspJ03]. F. Diotalevi identifiziert die drei Zusicherungsarten „Vorbedingung“, „Nachbedingung“ und „Invariante“ als verstreute Sachverhalte. In [Diot04] stellt er eine Softwarelösung mit AspectJ vor, mit der diese Zusicherungen auf einfache Weise in ein Programm eingebracht werden können. R. Laddad beschreibt den Einsatz von Aspekten im *Refactoring*. In [Ladd03a] gibt er einen Überblick über die Anwendung von AOP zur Restrukturierung von Code. In [Ladd03b] beschreibt er, wie man verschiedene Extrations-Techniken des Refactoring als Aspekte implementieren kann.

2.4.3 Produktions-Aspekte

Aspekte bieten eine Lösung für *Synchronisationsaufgaben* (z.B. die Verwaltung von kritischen Abschnitten), da solche Mechanismen häufig an verschiedenen Stellen eines Softwaresystems benötigt werden. D. Lohmann et al. benutzen in [Lohm04] die AOSE zur Implementierung eines Mutex. Y. Coady et al. fassen in [Cacm02g] essenzielle Elemente von Betriebssystemen als verstreute Sachverhalte auf, da ihre Implementierung mit vielen Ebenen des Systems gekoppelt ist. Sie untersuchen die AOP als Mittel zur Verbesserung der Struktur von Betriebssystemen am Beispiel *Prefetching*. Auch *sicherheitstechnische* Sachverhalte können gut mit Aspekten realisiert werden, nämlich überall wo Verschlüsselungen, Entschlüsselungen, Rechte- und Authentizitätsprüfungen usw. an mehreren Stellen eingesetzt werden müssen.

Wie in Abschnitt 2.1.2 an der Serialisierung demonstriert wurde, sind Probleme der *Persistenz* klare Anwendungsfälle für Aspekte. Gleiches gilt für die *Replikation* von Objekten und die Implementierung von Objektpools. stellen in [Schu03] eine Implementierung vor, die mit einem Aspekt erstellte Objektreplikate zur Tolerierung von Objektabstürzen verwendet. Die Objekt-Replikation lässt sich auch für die Verteilung von Objekten auf Threads oder die Delegation an andere Rechner verwenden. W. Schult und A. Polze beschreiben in [Schu04] die Implementierung eines Aspekts, mit dem das *Zeitverhalten* eines Algorithmen zur Laufzeit gesteuert werden kann, indem seine Berechnung auf mehrere Rechner verteilt wird.

2.4.4 Fachliche Aspekte

Der Einzugsbereich von Aspekten ist nicht nur auf systemtechnische Aufgaben begrenzt. Eine bedeutende Form von verstreuten Sachverhalten ist die Variabilität einer Applikation. Ist eine Software für verschiedene Konfigurationen oder Benutzerprofile ausgelegt, so wird sie von vielen Punkten der Variabilität durchkreuzt [Sach03]. Statt den Code jeder Konfiguration permanent in der Applikation zu verwalten, kann man jede Konfiguration als getrennten Aspekt modellieren. Mit dieser Methodik eröffnen sich auch elegante Möglichkeiten zur Realisierung von Produktionslinien, bei denen die Software sich lediglich in bestimmten Schichten unterscheidet [Sach03]. P. Dolog et al. präsentieren in [Dolo01] eine Strategie, mit der sogar Veränderungen im Softwareentwicklungsprozess durch Aspekte dargestellt werden können.

Auch auf der Ebene elementarer Programmieraufgaben lassen sich verstreute Sachverhalte in Aspekten kapseln. In ihrem viel zitierten Pionier-Dokument [Kicz97] sehen G. Kiczales et al. den Einsatz von Aspekten auch für den Zusammenschluss von Schleifen oder die Matrizendarstellung in mathematischen Algorithmen vor. Das Team von Kiczales verwendet zur Erklärung der AOP in [Cacm01b] und [AspJ03] ein Beispiel mit geometrischen Objekten. In einigen Methoden dieser Objekte befinden sich Aufrufe zu einer bestimmten Methode eines Figureditors, welche die Aktualisierung der Ansicht implementiert. Hier wird schnell deutlich, dass selbst Methodenaufrufe einen verstreuten Sachverhalt darstellen und als Aspekt modelliert werden kann. Die Granularität kann sogar noch verkleinert werden. Wenn man einen Sachverhalt als eine Eigenschaft auffasst, die eine Klasse besitzt, sind auch Felder, oder etwa Schnittstellen- oder Basisklassenangaben als verstreute Sachverhalte zu betrachten.

Kapitel 3: Die Technik

Dieses Kapitel stellt die verschiedenen Techniken vor, mit denen AOSE-Implementierungen realisiert werden können. Zuerst soll die Technik der Verwebung vorgestellt werden, die ein fundamentales Element der AOSE ist. Anschließend sollen weitere Techniken sowie softwaretechnische Konstrukte untersucht werden, die zur Lösung des Problems der verstreuten Sachverhalte beitragen können. Ferner sollen die verschiedenen Methoden analysiert werden, die vom .NET-Framework bereitgestellt werden. Die Untersuchung der gegebenen Techniken ist eine notwendige Voraussetzung, um die Funktionsweise bestehender Implementierungen zu verstehen sowie eine eigene Implementierung entwerfen zu können. Vor dieser Analyse soll ein Anforderungsprofil entworfen werden, das die Basis für die Evaluation der verschiedenen Techniken bilden wird.

Bei den Betrachtungen in diesem und den nächsten Kapiteln sind folgende Begriffe zu unterscheiden:

- **Aspekt / Aspektcode:** Aspekte kapseln Code verstreuter Sachverhalte.
- **Zielklasse / Zielcode:** Klasse, auf die Aspekte angewendet werden sollen.
- **Client / Clientcode:** Clients benutzen Zielklassen, indem sie Instanzen der Zielklassen erstellen oder ihre Methoden aufrufen.
- **Komponente / Komponentencode:** Komponenten kapseln den Code der Geschäftslogik einer Applikation und bestehen aus Zielklassen und Clients.
- **Applikation /Applikationscode:** Eine Applikation ist ein kompilier- und lauffähiges Softwareprogramm, das aus Komponenten besteht und Aspekte nutzt.

3.1 Theoretische Anforderungen

3.1.1 Trennung von Sachverhalten

Eine Software oder Technik zur Realisierung der AOSE sollte ihrem eigentlichen Zweck, nämlich der Trennung der Sachverhalte, erfolgreich dienen. Daher sollte der Aspektcode vom Komponentencode klar getrennt sein. C. Flach und C. de Lucena fordern in [Flac03] eine strenge Unterscheidung von Komponenten und Aspekten in Softwaresystemen und bezeichnen sie als *Zweiteilung von Aspekt und Basis* („aspect-base dichotomy“).

3.1.2 Wiederverwendbarkeit und Quantifikation

Aspektcode sollte von konkreten Softwareprodukten abstrahierbar sein und als unabhängige Komponente wieder verwendet werden können. Zur Erfüllung dieser Bedingung sind Ausdrucksmöglichkeiten erforderlich, die es erlauben, Aspekte als eine Art Schablonen für bestimmte verstreute Sachverhalte zu entwerfen. Damit verbunden ist der Begriff *Quantifikation* („quantification"). Quantifikation ist die Fähigkeit zur Formulierung einheitlicher Ausdrücke, die sich auf möglichst viele Stellen eines Programms auswirken [Film00]. Quantifikation ermöglicht Ausdrücke wie: „Wenn eine Voraussetzung V in einem Programm P eintritt, führe die Aktion A aus" [Film00]. [Cacm01a] verwendet auch den Ausdruck „globality versus locality" für die Eigenschaft der Quantifikation.

3.1.3 Transparenz und Unbewusstheit

Damit keine Verschmutzung des Komponentencodes durch Aspektcode erfolgt, sollte die getrennte Entwicklung von Komponenten und Aspekten möglich sein.[1] Der Komponentencode sollte so entwickelt werden können, dass er sowohl ohne Aspekte als auch mit Aspekten kompilier- und ausführbar ist. Das Vorhandensein von Aspekten sollte dem Entwickler des Komponentencodes gegenüber *transparent* sein. In [Cacm01b], [Film00] u.a. wird diese Anforderung auch als *Unbewusstheit* („obliviousness") bezeichnet. Im Folgenden wird hierfür der Begriff der *Transparenz* verwendet, da er in der Informatik für ähnliche Gegebenheiten steht.

3.1.4 Flexibilität

Angewendet auf eine Zielklasse, sollte ein Aspekt Folgendes bewerkstelligen können:

- Änderung des Verhaltens der Zielklasse.
- Erweiterung des Funktionsumfangs der Zielklasse.

Die Änderung des **Verhaltens** einer Klasse spricht ihre **dynamische** Seite an. Die Dynamik einer Klasse äußert sich in Zustandsänderungen des Objekts. Dies kann an folgenden Stellen der Fall sein:

- Ausführung einer **Methode**.
- **Konstruktion** oder Destruktion des Objekts.
- **Setzen** eines Feldes.

Es muss also eine Möglichkeit geboten werden, an diesen Stellen in den Zielcode eingreifen zu können und ihn verändern oder erweitern zu können.

[1] Dieses Prinzip ist z.B. aus den *Enterprise Java Beans* und ihrem Einsatz in Applikationsservern bekannt, wo strikt zwischen dem Entwickler der Geschäftslogik und dem Entwickler der Systemlogik unterschieden wird.

Die Änderung des **Funktionsumfangs** einer Klasse spricht ihre **statische** Seite an. Das Ziel ist nicht die Änderung von Anweisungen, sondern der Struktur der Klasse. Dies kann durch folgende Maßnahmen realisiert werden:

- Erweiterung der Klasse um **Felder**, **Methoden** oder **Konstruktoren**.
- Ableitung der Klasse von weiteren **Basisklassen**[1] oder Implementierung weiterer **Schnittstellen**.

Die Erweiterung von Klassen durch Einführung von Membern und Beziehungen wird in [AspJ03] als *„inter-type declaration“*, in [Urba04] und anderer Literatur als *„introduction“* oder *„member introduction“* bezeichnet. Die Erweiterung durch Basisklassen wird als *umgekehrte Vererbung* („reverse inheritance“) bezeichnet, da die Klasse die Vererbung nicht anfordert, sondern vom Aspekt „aufgezwungen“ bekommt [AspJ03].

Des Weiteren sollte es möglich sein, **mehrere Aspekte** an einer Stelle des Komponentencode einsetzen zu können.

3.1.5 Präzision

Ferner wird deutlich, dass Aspekte nicht nur Klassen als Ganzes zum Ziel haben können sollten, sondern auch ihre Member. In Abschnitt 2.4.5 wurden sogar einzelne Funktionsaufrufe als verstreuter Sachverhalt identifiziert. Idealerweise sollten Aspekte auf allen Ebenen des Komponentencodes operieren können, wie z.B. auf Klassen, Klassenmembern, einzelne Anweisungen, Modulen usw. In diesem Zusammenhang wird der Begriff der *Präzision* eingeführt. Eine hohe Präzision erlaubt die Anwendung von Aspekten auf Konstrukte kleiner Granularität.

3.1.6 Anwendbarkeit

Der Aspektcode sollte an den Komponentencode keine Anforderungen stellen. Insbesondere sollte weder Syntax noch Semantik des Komponentencodes verändert werden müssen, um Aspekte anwenden zu können. Häufig wird die Anwendbarkeit eingeschränkt, wenn bei den Entwurfszielen die Transparenz im Vordergrund steht [Flac03]. Bei fehlender Anwendbarkeit muss der Komponentencode für den Einsatz von Aspekten vorbereitet werden. Diesen Vorgang bezeichnen T. Elrad et al. als *Intimität* („intimacy“) [Cacm01a].

[1] Im Falle des .NET-Framework ist die Ableitung einer Klasse von insgesamt nur einer Klasse möglich.

3.2 Verwebung

3.2.1 Aspekt-Weaver

Eine logische Konsequenz aus dem Grundprinzip der Trennung der Sachverhalte, sowie dem im letzten Kapitel erstellten Anforderungsprofil ist die physische Trennung von Komponenten und Aspekten. Auf diese Weise können beide separat entwickelt, gewartet und wieder verwendet werden. Um jedoch eine lauffähige Applikation zu erstellen, die den Aspektcode nutzt, müssen der fertige Komponenten- und Aspektcode miteinander *verwoben* werden, wie Bild 3.1. zeigt:

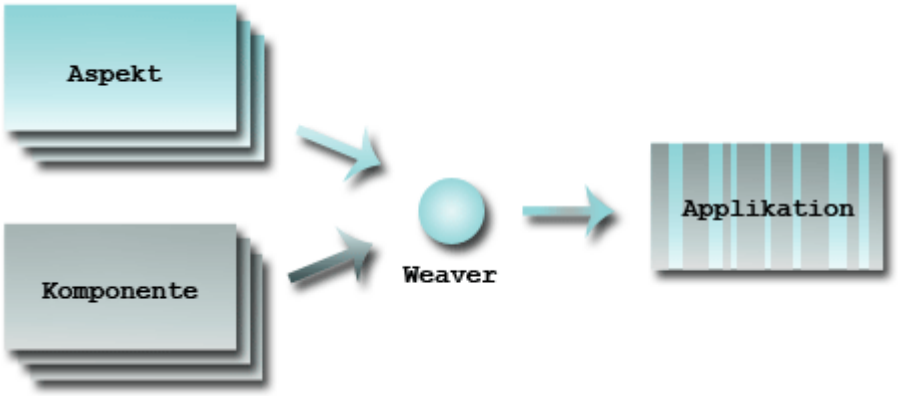

Bild 3.1 Prinzip der Verwebung

Eine Applikation oder Softwarekomponente zur Verwebung von Komponentencode mit Aspektcode wird *Aspekt-Weaver*[1] („aspect weaver“) genannt [Kicz97].

3.2.2 Verwebungsarten

Je nach Aspekt-Weaver kann die Verwebung zu verschiedenen Zeitpunkten erfolgen:

- Bei *statischer Verwebung* werden Aspekt- und Komponentencode *vor* der Ausführung der Applikation verwoben.
- Bei *dynamischer Verwebung* werden Aspekt- und Komponentencode *während* der Ausführung der Applikation verwoben.

Des Weiteren unterscheidet [Flac03] zwischen *destruktiver Verwebung* („in-place modification“) und *nichtdestruktiver Verwebung* („client migration“). Nach der destruktiven Verwebung ist der Komponentencode nicht mehr vorhanden, während nach der nichtdestruktiven Verwebung der Komponentencode erhalten bleibt.

[1] „to weave“ = verweben

3.2.3 Statische Verwebung

Da bei statischer Verwebung der Code schon spätestens vor seiner Ausführung verwoben werden muss, erfordert die statische Verwebung eine genauere Analyse des Codes. Im Idealfall muss der Weaver über einen Parser verfügen, der die gesamte Syntax der Quellsprache sowie der Aspektdefinition beherrscht. Dabei kann die Aspektdefinition in manchen Fällen auch vollständig durch die Grammatik der Quellsprache ausgedrückt werden. Wenn die Aspektdefinition auf der Quellsprache basiert und eigene Schlüsselwörter einführt, spricht man von *Spracherweiterung* [AspC]. In [Kicz97], [Ladd02], [Flac03] u.a. wird sogar der Begriff der *AOP-Sprache* benutzt. Bei der Quellsprache kann es sich entweder um eine höhere Programmiersprache oder um eine virtuelle Maschinensprache (z.B. bei .NET oder Java) handeln. Der Zeitpunkt der Verwebung lässt sich bei statischer Verwebung also noch genauer spezifizieren:

- Bei *Verwebung auf Quellcodeebene* werden Aspekt- und Komponentencode *vor* der Kompilierung[1] der Applikation verwoben.
- Bei *Verwebung auf Zwischencodeebene* werden Aspekt- und Komponentencode *nach* der Kompilierung der Applikation verwoben (z.B. bei Java oder .NET).

Bild 3.2 gibt einen Überblick über die verschiedenen Arten der Verwebung:

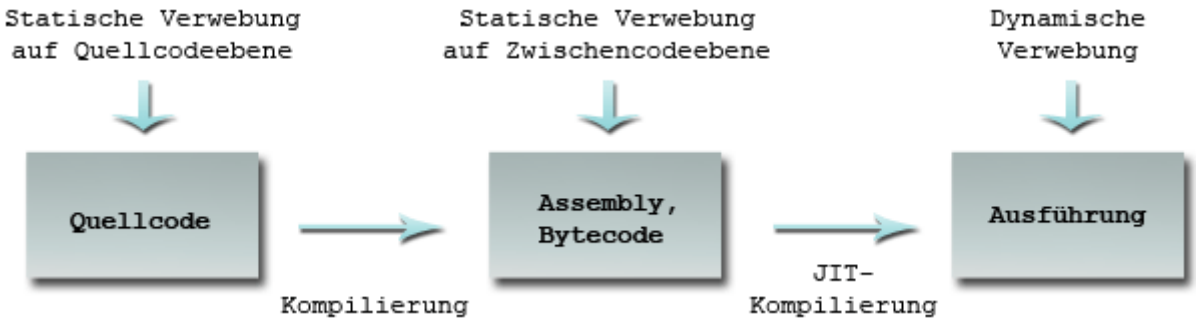

Bild 3.2 Verwebungsarten

Die Verwebung auf Zwischencodeebene hat einige wesentliche Vorteile: Sie ermöglicht präzisere Eingriffe in den Code, kann sich Reflektionsmechanismen zu nutze machen und ermöglicht im Falle .NET den Einsatz mehrerer Programmiersprachen, ohne dass für jede von ihnen ein spezieller Parser vorhanden sein muss. Die Nachteile der Zwischencodeverwebung liegen vor allem in der Unübersichtlichkeit und der mühsamen Implementierung. Die Manipulation mit Zwischencode erfordert genauste Kenntnis über die physische Struktur von .NET-PE-Dateien bzw. Java-Bytecode-Dateien sowie über die Arbeitsweise der virtuellen Maschinen. Als Hilfe kann hier z.B. ein Disassembler oder die

[1] Hiermit ist nur die Kompilierung von Quellcode gemeint, nicht die JIT-Kompilierung von Zwischencode.

Analyse der Metadaten mit Reflection dienen, wobei Reflection nur Auskunft über die statische Struktur einer Anwendung gibt. Dagegen hat die Verwebung auf Quellcodeebene den Vorteil, dass das Resultat der Verwebung für den Programmierer des Quellcodes lesbar ist und gegebenenfalls noch von ihm optimiert werden kann.

Ferner sprechen C. Flach und C. de Lucena in [Flac03] von der *Verwebung zur Ladezeit* („load-time weaving“) als spezielle Form der statischen Verwebung, gehen aber nicht näher darauf ein. Laut C. Laffra und M. Lippert werden dabei Klassen mit Aspektcode zu einem Zeitpunkt verwoben, zu dem sie in die virtuelle Maschine geladen werden. Diese Technik wird durch speziell dafür entwickelte Klassenlader realisiert [Laff03].

3.2.4 Dynamische Verwebung

Die dynamische Verwebung hat gegenüber der statischen einige maßgebliche Vorteile: Sie ermöglicht es zum einen, die Zuweisung von Aspekten zu den Komponenten auf die Laufzeit zu verschieben. In [Schu04] wird die Nutzung von dynamischer Verwebung beschrieben, um die Entscheidung zwischen speicher- oder rechenzeitoptimierter Durchführung eines Mandelbrot-Algorithmus zur Laufzeit zu treffen. Des Weiteren erlaubt es die dynamische Verwebung, auf das Parsen des Quellcodes zu verzichten, was ein entscheidendes Kriterium für die Durchführung einer Implementierung sein kann.[1] Laufzeitverwebung macht es möglich, Aspekt-Weaver zu implementieren, die lediglich aus einer Klassenbibliothek bestehen und ohne zusätzliche Software auskommen. Solche Implementierungen können in die Applikation eingebunden und von ihr wie eine API benutzt werden. Ein weiterer entscheidender Vorteil der Laufzeitverwebung ist im Falle .NET die Unabhängigkeit von der Programmiersprache.

Allerdings unterliegt die dynamische Verwebung einigen Einschränkungen, denn im Gegensatz zur statischen Verwebung kann hier keine beliebige Manipulation von Code erfolgen. Zwar ermöglicht .NET die Generierung von Code zur Laufzeit mit `Reflection.Emit`. Die `Reflection.Emit`-API erlaubt allerdings nur die Generierung von Code in neu erstellten Assemblies [Robi02].[2] Ferner setzt die dynamische Verwebung dynamische Bindung voraus, da der Weaver nur mit solchen Objekten und Methoden manipulieren kann, die erst zur Laufzeit gebunden werden. Konsequenterweise müssen alle Ziel-Member als virtuell oder in Schnittstellen deklariert sein [RaDo04]. Oder aber man bedient sich der Reflection, was jedoch Geschwindigkeitseinbußen mit sich führt. Des Weiteren ist aufgrund der dynamischen Bindung keine Behandlung statischer Member möglich.

[1] Es ist trotzdem denkbar, dass ein Parser zur Prüfung des Codes oder zur Intimität (vgl. Abschnitt 3.1.6) eingesetzt werden kann.
[2] `Reflection.Emit` wird in Abschnitt 3.4.2 genauer behandelt.

Um solche Einschränkungen zu umgehen, gibt es Ideen und Projekte, AOSE-Unterstützung in Laufzeitumgebungen zu implementieren. [Hils04] erwähnt ein Konzept namens „just-in-time aspect weaving“, mit dem in AspectJ die Verwebung zur Ladezeit ermöglicht werden soll. Auch Werkzeuge zur Bearbeitung von Bytecode zur Laufzeit werden gerne als Techniken für die AOSE verwendet. [Jass04] stellt eine Implementierung für Java vor, mit der sich geladener Bytecode zur Laufzeit manipulieren lässt, ohne beim Programmierer Kenntnisse über die Bytecodesyntax voraussetzen zu müssen. [BCEL02] ist eine weit verbreitete Klassenbibliothek zur statischen Modifikation von Bytecode.

3.2.5 Joinpoints

Verwebung erfolgt, indem Aspektcode in bestehenden Komponentencode eingefügt wird. Dazu muss definiert werden, an welchen Stellen Aspektcode eingewoben werden soll. Diese Stellen werden *“Joinpoints“* (d.h. Verbindungspunkte) genannt [Kicz97]. In der AOSE gibt es verschiedene *Joinpoint-Modelle* („join point models“), die den Begriff des Joinpoint genauer spezifizieren [Flac03]:

- **Statisches Joinpoint-Modell:** Joinpoints sind wohldefinierte Stellen im Quellcode.
- **Dynamisches Joinpoint-Modell:** Joinpoints sind wohldefinierte Stellen in der Ausführung eines Programms [Kicz01].

Erreicht beim dynamischen Joinpoint-Modell der Kontrollfluss des Programms zu verschiedenen Zeitpunkten die gleiche Stelle im Quellcode, so handelt es sich trotzdem um verschiedene Joinpoints. Man kann sich Joinpoints im dynamischen Joinpoint-Modell als Knoten in einem Programmflussgrafen vorstellen [Kicz01], wie Bild 3.3 erläutert:

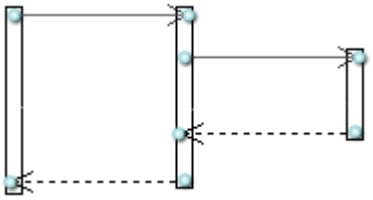

Bild 3.3 Joinpoints (dargestellt als blaue Punkte) im dynamischen Joinpoint-Modell

Für Joinpoints können Codebereiche beliebiger Präzision (vgl. Abschnitt 3.1.5) in Frage kommen. Je nach Implementierung sind einzelne Codezeilen, Methodenaufrufe, ganze Codebereiche wie z.B. Methoden, Klassen, Module usw. als Zielpunkte im Komponentencode denkbar.

3.2.6 Pointcuts

Verstreute Sachverhalte sind über viele Stellen einer Applikation verstreut. Deswegen sollte es möglich sein, mehrere Joinpoints zu einem Konstrukt zusammenzufassen. Dieses Konstrukt wird *Pointcut* genannt. Pointcuts stellen ein wesentliches Element der Modularisierung in Aspekten dar, denn mit einem Pointcut kann eine Gesamtheit von Zielstellen für einen Aspekt definiert werden. Beispiele für Pointcuts sind: Alle öffentlichen Methoden, alle Methoden mit einem bestimmten Rückgabetyp, alle Methodenaufrufe zu einem Objekt eines bestimmten Typs, alle Member einer bestimmten Klasse, alle Klassen in einem bestimmten Modul, alle Methoden einer Klasse, deren Name mit einer bestimmten Zeichenkette beginnt usw. In Implementierungen wie AspectJ und AspectC++ lassen sich Pointcuts durch die logische Verknüpfung von Ausdrücken definieren, die aus Zeichenketten bestehen, die wiederum Wildcards enthalten können. Im dynamischen Joinpoint-Modell können Pointcuts auch Programmabschnitte darstellen, die anhand des Kontrollflusses der Applikation bestimmt werden [AspJ03].

3.2.7 Advice

Den auszuführenden Aspektcode, der in den Komponentencode eingewoben wird, nennt man *Advice*. Advice[1] ist eine Menge von Aktionen, die an einen Pointcut gebunden sind und für einen Aspekt ausgeführt werden sollen [AspJ03]. In den meisten Fällen ist Advice ähnlich zu gewöhnlichen Methoden und verfügt über Namen, Parameterlisten sowie Rückgabewerte. In vielen Implementierungen (z.B. Rapier-Loom.Net) werden anhand dieser Signaturelemente Zielmethoden im Komponentencode gesucht. Der Advice-Code wird so hineingewoben, dass er zu verschiedenen Zeitpunkten bzgl. der Zielmethode ausgeführt wird, z.B. *davor*, *danach* oder *anstelle* der Zielmethode. Im letzten Fall ist häufig die Rückkehr des Programmflusses in die Zielmethode möglich [AspJ03].

3.2.8 Introduction

Häufig möchte man die Funktionalität der Aspekte nicht nur auf Aktionen begrenzen, die an bestimmten Stellen ausgeführt werden sollen. Eine weitere mächtige Eigenschaft vieler Aspekt-Weaver ist deswegen die Fähigkeit, statische Veränderungen im Komponentencode durchzuführen. Wie in Abschnitt 3.1.4 bereits erwähnt wurde, kann dazu die Einführung von Membern in Klassen oder auch das Hinzufügen von Ableitungs- oder Schnittstellen-implementierungsangaben gehören. Advice- und Zielcode können dann auf die eingeführten Member zugreifen, als wären sie bereits Bestandteil des Zielcodes.

[1] Manchen Leser mag die etwas gewöhnungsbedürftige grammatikalische Verwendung des Begriffs „advice“ wundern. Es handelt sich dabei um ein „uncountable noun“, also ein unzählbares Nomen wie „information“ oder „furniture“, das im Englischen für das Singular und Plural nur eine Form hat.

3.2.9 Zusammenfassung und Ausblick

Verwebung ist die Basistechnik der AOSE. Sie dient der Rekomposition eines Systems, das zuvor in Sachverhalte dekomponiert wurde.[1] Aspekte enthalten Advice zur Veränderung des dynamischen sowie unter Umständen auch Introduction zur Veränderung des statischen Verhaltens des Komponentencodes. Die Verwebestellen werden Joinpoints genannt und in den Aspekten durch Pointcuts spezifiziert. In Bild 3.4 sind all diese Eigenschaften nochmals zusammenfassend dargestellt:

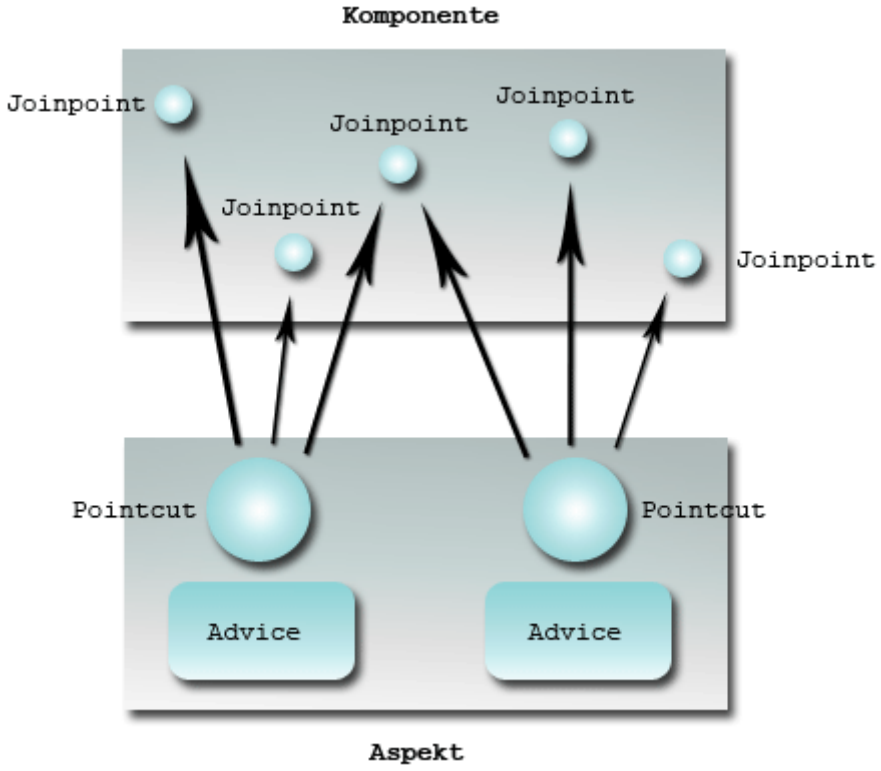

Bild 3.4 Aspekte, Advice, Pointcuts und Joinpoints als Grundelemente der Verwebetechnik

Implementierungen, die auf statischer Verwebung beruhen, ermöglichen nahezu beliebige Modifikation des Codes und somit präzise Eingriffe in den Komponentencode sowie eine hohe Flexibilität in den Ausdrucksmöglichkeiten. Dynamische Verwebung erlaubt dagegen Implementierungen ohne zusätzliche Sprachmittel und Software. Die Definition von Aspekten ist mit reinen objektorientierten Sprachkonstrukten realisierbar. Aspekte haben Ähnlichkeit mit Klassen und können unter Umständen als solche definiert werden. Gleiches gilt für Advice und Methoden.

Die Technik der Verwebung wurde maßgeblich von AspectJ inspiriert. Viele der hier vorgestellten Begriffe werden deshalb im Kapitel 4.1 im Zusammenhang mit AspectJ neu aufgegriffen und genauer beschrieben.

[1] Dies ist eine Analogie zu dem in Abschnitt 2.2.1 vorgestellten Prismenmodell.

3.3 Proxy-Techniken

3.3.1 Statischer Proxy

Eine mit einem Aspekt versehene Klasse ist eine Klasse mit erweitertem Funktionsumfang. Die trivialste Möglichkeit, eine Klasse in ihrem Funktionsumfang zu erweitern, ist die *Vererbung*. Eine abgeleitete Klasse kann als *Proxy* (d.h. Stellvertreter) für die Basisklasse agieren, indem sie Methoden der Basisklasse überschreibt. Die abgeleitete Klasse kann dann, mit zusätzlicher Funktionalität ausgestattet, stellvertretend für die Basisklasse benutzt werden. Die Proxy-Methoden, welche die Methoden der Zielklasse überschreiben, übernehmen dann aspektspezifische Aufgaben. Bei Bedarf können sie *zu einem beliebigen Zeitpunkt* die Basismethode (also die Zielmethode) aufrufen. Dieses Verhalten wurde in Abschnitt 3.2.7 als wichtige Eigenschaft von Advice erwähnt. Listing 3.1 zeigt ein Beispiel für die Realisierung eines Proxy mit Vererbung:

```
namespace TargetNS
{
	class Target
	{
		public void SomeMethod()
		{
			Console.WriteLine("SomeMethod() called!");
		}
	}
}
```

```
namespace ProxyNS
{
	class Target : TargetNS.Target
	{
		static int nrCalls;

		public new void SomeMethod()
		{
			Console.WriteLine("Nr. of calls: "+(++nrCalls));
			base.SomeMethod();
		}
	}
}
```

```
// using TargetNS;
using ProxyNS;

namespace ClientCodeNS
{
	class Client
	{
		public void ClientCode()
		{
			Target t = new Target();
			t.SomeMethod();
		}
	}
}
```

Listing 3.1 Beispiel für statischen Proxy

In Listing 3.1 implementiert die Klasse `ProxyNS.Target` einen Profiling-Aspekt, der die Aufrufe zur Methode `TargetNS.Target.SomeMethod` zählt. Der Aufruf der Basismethode in der Methode `ProxyNS.Target.SomeMethod` bewirkt, dass der Advice-Code *vor* dem Komponentencode ausgeführt wird. Um im Clientcode keine Referenzen ändern zu müssen, gibt man der Aspektklasse den Namen der Zielklasse und platziert sie in einem separaten Namensraum. Zur Anwendung der Aspektklasse anstelle der Zielklasse muss im Clientcode lediglich die `using`-Anweisung geändert werden.

3.3.2 Dynamischer Proxy

E. Gamma et al. beschreiben zwei Techniken zur Wiederverwendung von Funktionalität: *Klassenvererbung* und *Objektkomposition*. Die Objektkomposition ist das Zusammenfügen von Objekten mit einheitlichen Schnittstellen über Referenzen [Gamm96]. Ferner sagen E. Gamma et al. Folgendes über die Verwendung von Vererbung und Komposition aus: „Ziehe Objektkomposition der Klassenvererbung vor“ [Gamm96].

Während die Vererbung der Software eine vordefinierte Statik aufzwingt, ermöglicht die Komposition Dynamik zur Laufzeit. Dies macht die Komposition als Technik für die dynamische Verwebung interessant. Darüber hinaus sind Vererbungsbeziehungen in vielen Fällen hinderlich, z.B. wenn die Erstellung des Objekts der Basisklasse hinausgezögert werden soll. Dies kann der Fall sein, wenn sie eine speicher- oder prozessorlastige Operation ist oder etwa eine Transaktion oder Deserialisierung erfordert [Gamm96]. Die Komposition ist somit geeignet für die Verwendung von Aspekten der Persistenz, Replikation und Verteilung (vgl. Abschnitt 2.4.4 über Produktionsaspekte).

Das für die AOSE so wichtige Proxy-Modell kann mit Komposition implementiert werden, indem das *Entwurfsmuster „Proxy“* aus [Gamm96] verwendet wird. Bild 3.5 zeigt dazu das Klassendiagramm:

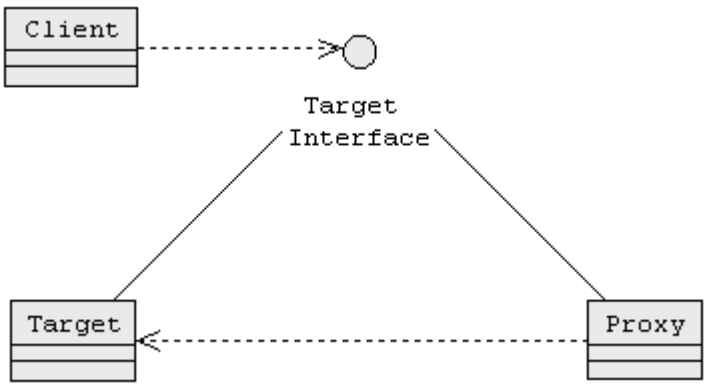

Bild 3.5 Klassendiagramm für das Proxy-Entwurfsmuster

Das Proxy-Muster erlaubt es, beide Akteure von ihrer Vererbungsbeziehung zu lösen. Damit sie aber gegeneinander austauschbar bleiben, implementieren sie die gleiche Schnittstelle. E. Gamma et al. sagen über die Anwendbarkeit des Proxy-Musters Folgendes aus: „Das Proxy-Muster ist anwendbar, sobald es den Bedarf nach einer anpassungsfähigeren und intelligenteren Referenz auf ein Objekt als einen einfachen Zeiger gibt“ [Gamm96].

In C++ kann ein solcher „smart pointer“ durch Überladung des Member-Zugriffsoperators implementiert werden. So kann bei jedem Zugriff auf die Proxyklasse der Zugriff sofort an die Zielklasse weitergeleitet werden. In C# ist diese Art der Überladung nicht möglich, daher muss man alle Methoden der Zielklasse im Proxy aufführen und dort die Umleitung explizit implementieren. Dies ist aber kein Problem, wenn alle öffentlichen Methoden in einer Schnittstelle deklariert sind, die sich Zielklasse und Proxy miteinander teilen. Listing 3.2 zeigt die Implementierung des Proxy-Musters in C#:

```
public interface ITarget
{
	void MethodA();
	void MethodB();
}
class Target : ITarget
{
	public void MethodA(){Console.WriteLine("MethodA() called!");}
	public void MethodB(){Console.WriteLine("MethodB() called!");}
}
class ProfilingAspect : ITarget
{
	ITarget t = null;
	static int nrCalls;

	void Profiling()
	{
		if(t==null)
			t = new Target() as ITarget;
		Console.WriteLine("Nr. of calls: "+(++nrCalls));
	}
	public void MethodA()
	{
		Profiling();
		t.MethodA();
	}
	public void MethodB()
	{
		Profiling();
		t.MethodB();
	}
}
class Client
{
	public void ClientCode()
	{
		ITarget target = new ProfilingAspect() as ITarget;
		target.MethodA();
		target.MethodB();
	}
}
```

Listing 3.2 Beispiel für dynamischen Proxy

Der Proxy in Listing 3.2 implementiert wiederum einen Profiling-Aspekt, der die Anzahl der Zugriffe auf Methoden der Zielklasse zählt. Durch die Indirektion wird die Klasse `Target` um Profiling-Funktionalität erweitert, ohne dass Ziel- und Proxyklasse eine Vererbungsbeziehung miteinander eingehen. Ferner wird aus Listing 3.2 deutlich, dass der Variable `target` Referenzen zur Laufzeit zugewiesen werden können. Dies ist ein entscheidender Vorteil gegenüber dem statischen Proxy.

3.3.3 Bewertung

Der statische Proxy mit Vererbung bietet eine einfach zu realisierende Möglichkeit für den Einsatz von Aspekten. Der eigentliche Zweck der Trennung der Sachverhalte wird erfüllt, denn das Verhalten der Zielklasse wird beeinflusst, ohne dass sie selbst verändert wird. Einzig die Angabe der `using`-Direktive verletzt die in Abschnitt 3.1.3 formulierten Anforderungen an die Transparenz. Die Zielklasse kann um Methoden, Felder, Ableitungsbeziehungen usw. erweitert werden. Insofern ist auch Introduction möglich.

Einige gravierende Nachteile erschweren allerdings die Anwendbarkeit des Proxy-Modells als Technik für die AOSE: Zum einen ist im Proxy-Modell die Zuweisung mehrerer Aspekte zu einer Zielklasse nicht direkt möglich. Dieses Problem könnte man mit Entwurfsmustern lösen, z.B. mit einer Verkettung von Proxy-Klassen, wie sie z.B. das *Entwurfsmuster „Dekorierer“* bietet [Gamm96]. Zum anderen erzwingt das Proxy-Modell, dass ein Aspekt ausschließlich auf eine konkrete Zielklasse zugeschnitten ist bzw. auf alle Klassen, welche die Schnittstelle der Zielklasse implementieren. Dies ist ein Mangel hinsichtlich der in Abschnitt 3.1.2 geforderten Quantifikation, sodass die Wiederverwendung von Aspekten schwierig ist. Ferner können nur öffentliche und geschützte Member um Funktionalität erweitert werden, was die Flexibilität beeinträchtigt (vgl. Abschnitt 3.1.4). Die Präzision (vgl. Abschnitt 3.1.5) ist auf die Ebene von Klassen beschränkt.

Trotz der vielen Nachteile sind Proxy-Klassen in zahlreichen AOSE-Implementierungen vertreten. Dank der Technik der Verwebung können sämtliche Vorgänge der Intimität (vgl. Abschnitt 3.1.6), die im Client- und Komponentencode vorgenommen werden müssen, automatisiert werden. In Abschnitt 4.2 wird ein Aspekt-Weaver vorgestellt, der mithilfe von *Schablonen* Proxies für beliebige Zielklassen nach dem Muster von Listing 3.1 generiert. Der dynamische Proxy hingegen ist eine häufig verwendete Technik in Weaver-Implementierungen mit Laufzeitverwebung. Dort werden Proxy-Objekte zur Laufzeit erstellt, um, mit erweiteter Funktionalität angereichert, stellvertretend für Zielobjekte zu agieren. Dynamische Proxies werden auch in der Interception-Technik von .NET verwendet, die in den Abschnitten 3.4.4 und 3.4.5 behandelt wird. Ein praktisches Beispiel für eine AOSE-Implementierung mit dynamischen Proxies wird in Abschnitt 4.3 vorgestellt.

3.4 .NET-Techniken

3.4.1 Überblick

Eine sorgfältige Evaluation der .NET-Plattform seitens AOSE ist wichtig für den Entwurf der Eigenimplementierung sowie das Verständnis der in Kapitel 4 diskutierten Software. .NET verfügt über eine mächtige Klassenbibliothek, die an zahlreichen Stellen Möglichkeiten für die Entwicklung einer AOSE-Implementierung bietet:

- Zugriff auf den Compiler vieler .NET-Sprachen über die Klassenbibliothek.
- Erstellung eines sprachunabhängigen abstrakten Syntaxbaums und Übersetzung in Quellcode.
- Reguläre Ausdrücke und diverse andere Werkzeuge für Compilerbauer.
- Reflection.
- Erzeugung von MSIL-Code zur Laufzeit.
- Attribute.
- Interception.

Folgend sollen dynamische Codeerzeugung, Attribute sowie Interception auf ihre Relevanz für die AOSE untersucht werden.

3.4.2 Dynamische Codeerzeugung

Der Namensraum `Reflection.Emit` ermöglicht die Generierung von MSIL-Code zur Laufzeit. Diese Eigenschaft präsentiert sich zunächst als ein sehr hilfreiches Mittel für die Implementierung eines Weavers auf Zwischencodeebene, hat jedoch eine Einschränkung: Mit `Reflection.Emit` lassen sich keine bestehenden Assemblies importieren und mit Code ergänzen. `Reflection.Emit` ist lediglich auf die Möglichkeit beschränkt, eine neue Assembly anzulegen und ihren Inhalt Zeile für Zeile aufzubauen [Robi02]. Um den physikalischen Aufbau der Assembly muss man sich dabei nicht kümmern, kann sich also auf den Code konzentrieren. Mit `Reflection.Emit` arbeitende Implementierungen der Laufzeitverwebung erstellen Kopien des Assembly-Codes einer Zielklasse, reichern ihn mit Aspektcode an und emittieren ihn in eine neu erzeugte temporäre Assembly, auf welche die laufende Applikation zugreift. Abschnitt 4.3 zeigt eine Implementierung, die mit dieser Technik arbeitet. In der Java-Welt existieren dagegen einige Projekte, die das Einfügen und Modifizieren von Code *zur Laufzeit* ermöglichen. Ein Beispiel dafür ist „Javassist“ [JAss04], mit dem auch die AOP-Erweiterung von JBoss arbeitet [JBos04]. Letztere wird in Abschnitt 4.4.3 vorgestellt.

Eine andere Art von dynamischer Codeerzeugung bietet der ***CodeDOM***-Namensraum. Mit einem Mechanismus namens ***D****ocument* ***O****bject* ***M****odel (DOM)* lässt sich ein sprachunabhängiger abstrakter Syntaxbaum aus sprachunabhängigen Programmierbausteinen erstellen [Robi02]. .NET bietet eine Schnittstelle, die für die Umwandlung eines solchen DOM-Syntaxbaums in Quellcode gedacht ist. In der .NET-Klassenbibliothek werden drei Implementierungen dieser Schnittstelle mitgeliefert: Für C#, Visual Basic und JScript [Robi02]. Diese internen Codegeneratoren erstellen Quellcode einer der oben genannten Sprachen und können diesen durch einen internen Compiler in MSIL-Code übersetzen lassen. Bild 3.6 gibt einen Überblick über die Möglichkeiten von `CodeDOM`:

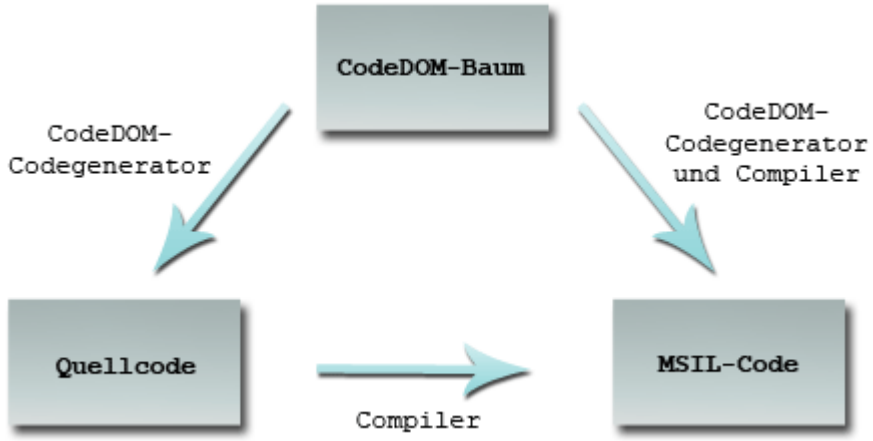

Bild 3.6 Dynamische Codeerzeugung mit dem Document Object Model

Die automatische Generierung eines DOM-Syntaxbaums aus Quellcode wird von der .NET-Klassenbibliothek nicht unterstützt [Robi02]. Es wird also ein Parser benötigt, der die Symbole des Quellcodes in DOM-Elemente umwandelt. Ferner sind nur CLS-konforme Elemente im DOM benutzbar. Nicht konformer Code kann, in Zeichenketten gespeichert, im DOM-Syntaxbaum beibehalten werden [MSDN03]. Dies macht selbstverständlich jedoch nur Sinn, wenn die Zielsprache der Quellsprache entspricht. Trotzdem ist das DOM eine nützliche Technik für Verwebung, denn ein DOM-Syntaxbaum lässt sich modifizieren und erweitern. Zudem lässt sich so Code verweben, der verschiedenen Programmiersprachen entstammt. Abschnitt 4.4.4 stellt eine Implementierung vor, die `CodeDOM` als fundamentale Technik nutzt.

3.4.3 Attribute

Durch Attribute erlaubt .NET dem Entwickler, Metainformationen an Klassen, Klassenmember und andere Konstrukte anzufügen. Schon in [Kicz97] sah man in Reflection und Metasprachen ein mächtiges Werkzeug für die AOP. Attribute sind deklarative Mittel, um Dienste bereitzustellen, die an einer Stelle definiert sind, möglicherweise aber an mehreren Stellen einer Applikation benötigt werden. Diese Idee ist stark verwandt mit Aspekten. In [Pies03] werden Attribute sogar direkt mit Verwebung in Verbindung gebracht, da sie in vielen Fällen die CLR zum Generieren und Einweben von Code veranlassen. Trotzdem ist die attributierte Programmierung mit der AOP nicht zu verwechseln [Nolt04]. Attribute können zwar eine nützliche Hilfe in der Implementierung einer AOP-Software sein, stellen selbst jedoch keine Funktionalität zur Verfügung. In gewisser Hinsicht wirken Attribute wie Markierungen im Komponentencode, die optional Parameter beinhalten können. Diese Idee ist vergleichbar mit den in der AOSE verwendeten Joinpoints.

Erst im Zusammenhang mit einem Benutzer bekommen Attribute eine Bedeutung. Der Benutzer kann einer der folgenden Akteure sein:

- Der **Compiler** bei Compilerattributen wie z.B. `Conditional`, `Obsolete` usw.
- Die **CLR** bei Attributen, die von .NET zur Laufzeit ausgewertet werden.
- Eine **Applikation**, welche die Metadaten der Assembly untersucht.
- Klassen und Methoden **innerhalb der Applikation**.

Für eine AOSE-Implementierung ist der erste Fall nur dann interessant, wenn ein eigener Compiler geschrieben werden soll. Der zweite Fall ermöglicht den Einsatz von *benutzerdefinierten Kontextattributen*. Diese bieten einen für die AOSE interessanten Ansatz und werden in Abschnitt 3.4.4 im Zusammenhang mit Interception diskutiert. Im dritten und vierten Fall kann es sich bei dem Benutzer um einen Aspekt-Weaver handeln. Ein Aspekt-Weaver kann die mithilfe der Attribute spezifizierten Metadaten auslesen und auf deren Basis zusätzlichen Code generieren. Im vierten Fall ist der Aspekt-Weaver Teil der Applikation oder Teil einer Klassenbibliothek, die von der Applikation verwendet wird. Dies erfordert, dass die Verwebung zur Laufzeit der Applikation stattfindet. Die CLR unterstützt dies, indem sie Instanzen von Attributen zur Initialisierungszeit von Objekten erstellen kann. Auch für die *Definition von Aspekten, Pointcuts und Advice* bieten Attribute viel versprechende Ausdrucksmöglichkeiten, denn sie können als *Ersatz für Spracherweiterungen* dienen (vgl. Abschnitt 3.2.3 über die statische Verwebung).

3.4.4 Interception

.NET bietet eine Umsetzung für das in Abschnitt 2.2.3 vorgestellte Kontextmodell. Durch Attribute können *Komponentendienste* („component services“) angefordert und durch *Interception* (d.h. Abfangen) von Methodenaufrufen implementiert werden. Den Abfangmechanismus realisiert die CLR, indem sie zur Laufzeit einen *Proxy* zwischen dem Aufrufenden und dem Aufgerufenen installiert [McLe02]. Es handelt sich also um eine Form von *Laufzeitverwebung*. .NET erlaubt es dem Programmierer, in diesen Prozess einzugreifen und an der Abfangstelle weitere Funktionalität zu implementieren. Diese Tatsache macht die Interception interessant für die AOSE, da das Abfangen von Methoden unter Ausführung zusätzlicher Aktionen mit dem Prinzip der Verwebung vergleichbar ist [Shuk02]. Interception wurde jedoch nicht mit dem Ziel entwickelt, AOSE in das .NET-Framework einzuführen, denn sie stammt aus dem Bereich des *Remoting* und der verteilten Applikationen [McLe02].

Ein Prozess kann in .NET in viele logische Prozesse, die sog. *Applikationsdomänen* („application domains“), aufgeteilt werden. Damit ist es möglich, viele Applikationen in einem Prozess laufen lassen, ohne dass sie sich gegenseitig durch Schutzverletzungen stören [McLe02]. Dies funktioniert dank Verifizierung des Codes von .NET-Applikationen durch die CLR [Robi02]. Die Grundidee des Remoting in .NET ist es, jegliche Funktionalität, die für die Kommunikation zwischen verteilten Anwendungen benötigt wird, vor dem Komponentencode zu verschleiern. Für Kommunikationspartner, die sich in verschiedenen Applikationsdomänen befinden, also in verschiedenen Applikationen oder auf verschiedenen Rechnern, stellt .NET die Kommunikationsinfrastruktur automatisch zur Verfügung. Ruft in solchen Fällen ein Clientobjekt ein Serverobjekt auf, wird der Aufruf durch einen Proxy unterbrochen und durch eine Reihe von *Nachrichtensenken* („message sinks“) geleitet. Nachrichtensenken sind auf Client- und Serverseite in Ketten angeordnet und übernehmen einzelne Teilaufgaben, um z.B. stapelbasierte Methodenaufrufe in serielle Ströme zu wandeln und umgekehrt. Auf der Serverseite bildet eine Senke zur Wiederherstellung des Aufrufstapels das Gegenstück zum Proxy. Bild 3.7 zeigt den Grundriss der Remoting-Infrastruktur von .NET:

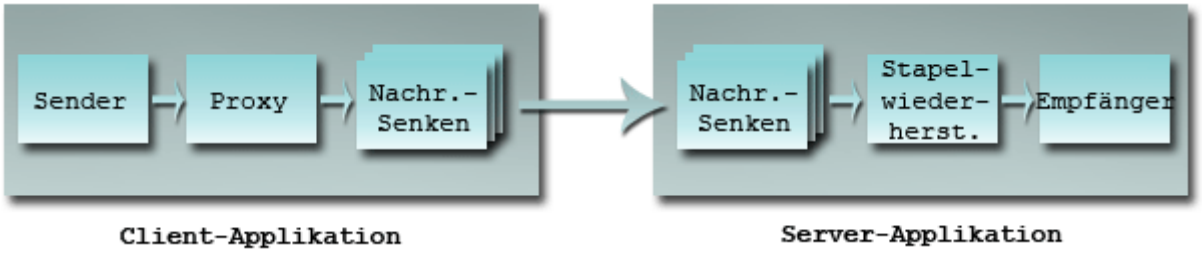

Bild 3.7 Kette von Nachrichtensenken in der Remoting-Infrastruktur von .NET

[Shuk02] präsentiert eine Methode, wie Interception für die AOSE nützlich gemacht werden kann. [Lowy03] und [Siev03] vertiefen diesen Ansatz. Interception wird für die AOSE erst dadurch besonders interessant, dass sie nicht nur an Grenzen von Applikationsdomänen zum Zuge kommt, sondern auch *innerhalb* einer Applikation eingesetzt werden kann. Applikationsdomänen lassen sich nämlich optional in *Kontexte* unterteilen. Es ist möglich, für bestimmte Objekte einen Kontext zu erzeugen, indem man ihnen *Kontextattribute* zufügt. Das einzige vorgefertigte .NET-Kontextattribut ist `Serializable` [Arch02]; es lassen sich jedoch auch benutzerdefinierte Kontextattrbiute erstellen. Damit ein Objekt an seinen Kontext gebunden wird, muss es zusätzlich von `ContextBoundObject` abgeleitet sein. Sind diese Voraussetzungen gegeben, so wird der ganze Mechanismus der Interception mit den zugehörigen Proxies und Ketten von Nachrichtensenken für das Objekt eingerichtet. Bild 3.8 zeigt den Zusammenhang zwischen Applikationsdomäne, Kontext und Interception:

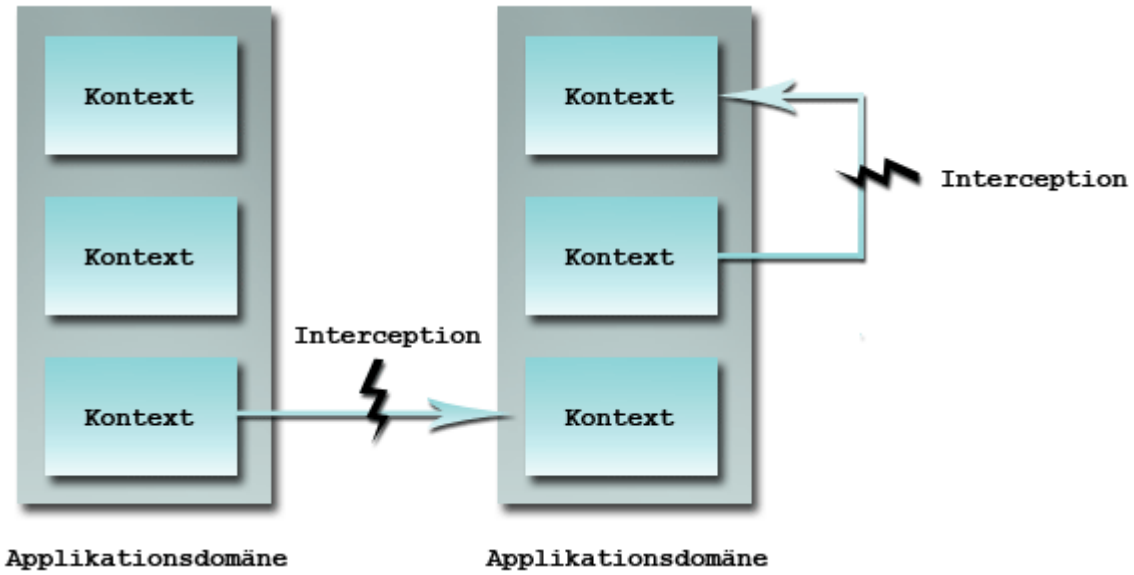

Bild 3.8 Eintritt von Interception bei Überschreitung von Applikationsdomänen- oder Kontextgrenzen

3.4.5 Anwendung der Interception

Zur Realisierung der Interception in einer Applikation müssen benutzerdefinierte Komponentendienste implementiert werden. Dazu gehört die Definition eines *Kontextattributs*, einer *Kontexteigenschaft* („context property“), sowie einer eigenen *Nachrichtensenke*, die in die Kette der vorhandenen Nachrichtensenken integriert wird und die Geschäftslogik des Dienstes enthält. Für diese Vorgänge stellt das .NET-Framework verschiedene Schnittstellen bereit. Listing 3.3 zeigt einen Auszug aus der Nachrichtensenke eines Tracing-Dienstes:

```
public IMessage SyncProcessMessage(IMessage msg)
{
	IMethodMessage methMsg = (IMethodMessage)msg;

	// Do some pre-processing here:
	Console.WriteLine("Tracing method: " +
	methMsg.MethodBase.DeclaringType + "." +
		methMsg.MethodBase.Name);

	// Pass message forward to next message sink:
	IMethodReturnMessage retMsg =
	(IMethodReturnMessage)this.nextSink.SyncProcessMessage(msg);

	// Do some post-processing here:
	if(retMsg.Exception != null)
		Console.WriteLine(retMsg.Exception+" in target method");

	// Return returned message to previous message sink:
	return retMsg;
}
```

Listing 3.3 Auszug aus einer benutzerdefinierten Nachrichtensenke

Die Remoting-Infrastruktur von .NET fängt Methodenaufrufe ab und leitet sie durch verschiedene Nachrichtensenken. Unterwegs wird die Methode in ein Objekt vom Typ `IMessage` verpackt, damit an ihr in den Nachrichtensenken zusätzliche Operationen durchgeführt werden können wie z.B. Verschlüsselung. In der benutzerdefinierten Nachrichtensenke von Listing 3.3 wird auf das `IMessage`-Objekt zugegriffen und eine bestimmte *Vorverarbeitung* durchgeführt. Anschließend wird das `IMessage`-Objekt an die nächste Senke weitergegeben. Dieser Vorgang setzt sich fort, bis die Nachricht schließlich an die Server-Seite gegeben wird, wo entsprechende Verarbeitungen in umgekehrter Reihenfolge stattfinden. Gelangt die Nachricht als Methodenaufruf an den Empfänger, wird der Aufruf durchgeführt. Die Rückmeldung wird nach demselben Prinzip in ein `IMessage`-Objekt verpackt, übertragen und gelangt schließlich an die Nachrichtensenke von Listing 3.3 zurück. Hier können eventuelle *Nachverarbeitungen* durchgeführt werden, bevor die Nachricht an die letzte Nachrichtensenke in Richtung Senderobjekt zurückgegeben wird. Aus Listing 3.3 wird klar, dass in Nachrichtensenken Advice-Code platziert werden kann, der zu verschiedenen Zeitpunkten bzgl. des Zielcodes ausgeführt werden kann.

Eine andere Möglichkeit für das Abfangen von Methoden bietet die Ableitung von der Proxy-Klasse, die am Anfang der Kette der Nachrichtensenken steht. Die Implementierung von benutzerdefinierten Proxies erfordert jedoch die manuelle Durchführung vieler Aktionen, die unter Verwendung der in Listing 3.3 gezeigten Methode von .NET automatisch bewältigt werden [McLe02]. Kontextattribute und Interception sind undokumentierte Elemente von .NET, sodass die MSDN-Dokumentation keine Auskunft über die Verwendung der damit verbundenen Klassen und Member gibt [MSDN03]. Eine genaue Behandlung der Interception lässt sich jedoch in [McLe02] finden. Einen Überblick über Kontextattribute gibt auch [Arch02].

Schlussfolgernd lässt sich sagen, dass die Interception eine hilfreiche Methode darstellt, um AOSE in .NET zu implementieren. Wenn auch die Interception kein hinreichendes und an vielen Stellen verbesserungswürdiges [Pies03] Mittel für eine AOSE-Implementierung ist, kann sie durchaus als Fundament für die dynamische Verwebung eingesetzt werden. Mit der in Listing 3.3 gezeigten Vorgehensweise lässt sich die Idee der Aspekte und Advice umsetzen. Die Joinpoints werden durch das Anbringen von Kontextattributen direkt im Komponentencode spezifiziert. Dies ist hinsichtlich der Transparenz (vgl. Abschnitt 3.1.3) ein Nachteil, kann aber für manchen Benutzer eine komfortablere Methode darstellen, als die Spezifizierung von Pointcuts mit Hilfe der logischen Verknüpfung von Ausdrücken (vgl. Abschnitt 3.2.6 über Pointcuts).

Kapitel 4: Die Implementierungen

Nach einer ausgiebigen Analyse der Basistechniken zur Realisierung der AOSE sowie einer Untersuchung verschiedener .NET-Techniken, sind die Grundlagen für das Verständnis der heute verbreiteten Implementierungen gegeben. Eine genaue Analyse von bestehenden Softwarewerkzeugen und Forschungsprojekten ist eine notwendige Voraussetzung für den Entwurf der Eigenimplementierung.

Zuerst soll AspectJ vorgestellt werden, das in der AOP-Welt als Muster für jegliche AOP-Implementierungen gilt und derzeit das AOP-Werkzeug mit der größten Verbreitung und kommerziellen Bedeutung ist. Eine genaue Analyse der Benutzerschnittstelle *sowie* der Arbeitsweise des Weavers soll Erkenntnisse über *Anforderungen und Implementierung* eines Aspekt-Weavers liefern. Anschließend sollen im Hinblick auf die geplante Eigen-implementierung zwei .NET-Implementierungen genauer untersucht werden, von denen die eine statische und die andere dynamische Verwebung als Grundtechnik nutzt: LOOM.NET und Rapier-Loom.Net.

In Abschnitt 4.4 werden weitere Implementierungen kurz vorgestellt. Abschnitt 4.5 geht auf Software und Projekte ein, die wie die AOSE die Realisierung des Prinzips der Trennung der Sachverhalte anstreben, aber nicht unter den Begriff der AOSE fallen. Auf [AOSD04] findet man ein umfangreiches Verzeichnis mit AOSE-Projekten.

4.1 AspectJ

4.1.1 Überblick

AspectJ [AspJ04] ist eine AOSE-Implementierung für Java. Von einem Xerox-PARC-Team unter der Leitung von Gregor Kiczales entwickelt, gilt es als Referenzimplementierung für die AOP (vgl. Abschnitt 2.1.5). Es ist sehr schwierig, über AOSE zu sprechen ohne sich mit AspectJ befasst zu haben. Zu dem Thema gibt es zahlreiche Veröffentlichungen, darunter auch sehr aufschlussreiche von den Entwicklern selbst (z.B. [Kicz01], [Cacm01f] und [AspJ03]). Heutzutage ist AspectJ das am meisten verbreitete AOSE-Werkzeug und erfreut sich einer beachtlichen Community. Seit Dezember 2002 ist es unter der Obhut des Eclipse-Projekts etabliert und unterliegt daher ständigen Testreihen sowie Verbesserungsvorschlägen

seitens der AOP-Praktiker selbst [Parc04]. Die Benutzerschnittstelle, die Implementierung sowie der Funktionsumfang von AspectJ sind von entscheidender Bedeutung für dieses Projekt und sollen im Folgenden näher untersucht werden. Die wesentlichen Eigenschaften von AspectJ sind:[1]

- **Sprachprozessor mit Java-ähnlicher Syntax:** Aspekte, Joinpoints, Pointcuts und Advice lassen sich durch eine mit Java verwandte Syntax beschreiben. Ein spezieller Compiler wandelt den Aspektcode in Bytecode um.
- **Dynamisches Joinpoint-Modell:** Der Begriff des Joinpoint bezieht sich auf die Ausführung eines Programms, nicht auf die statische Stelle im Code (vgl. Abschnitt 3.2.5).
- **Weaving auf Bytecodeebene:** Aspektcode wird zu Bytecode kompiliert, um dann mit dem Bytecode der Komponentenklassen verwoben zu werden.
- **Flexible und mächtige Ausdrucksmöglichkeiten zur Formulierung von Pointcuts und Advice:** Die Vielfältigkeit der Arten von Pointcuts und Advice bietet eine hohe Flexibilität in der Integration von Aspektcode in Komponentencode.
- **Context exposing:** Elemente, die im Komponentencode an Stellen der Joinpoints sichtbar sind (z.B. das Zielobjekt oder Methodenparameter) können im Advicecode benutzt werden.
- **Zugriff auf Attribute von Joinpoints:** Während der Laufzeit kann Information über Joinpoints und ihre Umgebung (z.B. die Signatur der Zielmethode) eingeholt werden.
- **Inter-type declarations:** AspectJ erfüllt die in Abschnitt 3.1.4 geforderten Fähigkeiten der Introduction in vollem Umfang.
- **IDE-Unterstützung:** Zahlreiche Projektgruppen entwickeln Plug-Ins und Module für Eclipse, Borland JBuilder, Emacs und Entwicklungsumgebungen von SUN.

4.1.2 Die Benutzerschnittstelle

Aspekte in AspectJ sind ähnlich wie Klassen in Java aufgebaut. Sie enthalten Definitionen für Pointcuts und Advice, können aber auch gewöhnliche Java-Konstrukte wie Felder und Methoden kapseln. Listing 4.1 zeigt ein Beispiel für eine Aspektdefinition in AspectJ:

[1] Alle Ausführungen, ausgenommen über die Technik, beziehen sich auf die AspectJ-Version 1.2.0.

```
aspect SomeAspect
{
       public void someMethod()
       {
            methodCode();
      }

       pointcut pc():
       execution(public * *(..)) && call(* SomeClass.*(..));

       after():
       pc() || call(void SomeOtherClass.someMethod(int, int));
       {
             adviceCode();
       }
}
```

Listing 4.1 Benutzerschnittstelle von AspectJ

Der Aspekt in Listing 4.1 definiert neben einer gewöhnlichen Java-Methode `SomeMethod` einen Pointcut mit dem Namen `pc`, der alle Joinpoints im Komponentencode heraussucht, die Teil der Ausführung einer öffentlichen Methode sind *und auch* Aufrufe einer beliebigen Methode der Klasse `SomeClass` darstellen. Darunter ist Advice angegeben, der sich auf die durch `pc` spezifizierten Stellen auswirkt *sowie* auf Aufrufe aller Methoden mit dem Namen `SomeOtherClass.SomeMethod`, die zwei Integer als Parameter nehmen und nichts zurückgeben. Der Advice enthält puren Java-Code, der ausgeführt wird, *nachdem* der Code der jeweiligen durch `pc` spezifizierten Methode abgearbeitet wurde.

In Listing 4.1 wird deutlich, dass zur Angabe von Methodensignaturen eine spezielle Syntax benötigt wird, welche die Benutzung von Wildcards vorsieht. Ferner offenbart sich eine wichtige Eigenschaft der Syntax von AspectJ, nämlich die Verknüpfung von Pointcut-Ausdrücken mit den Operatoren `&&`, `||` und `!`. Darüber hinaus demonstriert Listing 4.1 die Verwendung von benannten, wieder verwendbaren Pointcuts, aber auch die Deklaration anonymer Pointcuts inline im Kopf einer Advice-Methode. In AspectJ existiert eine enorme Anzahl vieler verschiedener Arten von Pointcuts und Advice, die in den folgenden Abschnitten näher erläutert werden.

4.1.3 Pointcut-Typen

Dank dynamischem Joinpoint-Modell sowie Codegenerierung und –Manipulation auf Bytecodeebene bietet AspectJ eine fast schon verwirrend große Anzahl von Arten, mit denen Joinpoints herausgesucht werden können. Hier ein Überblick über die Wichtigsten:

- **Aufruf:** Sucht *Aufrufe* einer oder mehrerer Methoden oder Konstruktoren im Komponentencode.
- **Ausführung:** Bezieht sich auf die *Ausführung* von Methoden bzw. Konstruktoren.

- **Initialisierung:** Bezieht sich auf besondere Stellen in der Initialisierung von Objekten (z.B. *vor dem Aufruf des Basisklassenkonstruktors* oder *nach der Initialisierung des gesamten Objektes*)
- **Get / Set:** Wenn Felder gelesen oder beschrieben werden.
- **Ausnahmen:** Bezieht sich auf die Ausführung einer Ausnahme im Catch-Block.
- **Code-Inhalt:** Bezieht sich auf alle Joinpoints (Aufrufe von Methoden bzw. Konstruktoren, Lesen und Schreiben von Feldern usw.), die innerhalb der angegebenen Methode im Code vorkommen.
- **Kontrollfluss:** Sucht Joinpoints *im* oder *nach* dem Kontrollfluss einer bestimmten Methode und ihrer Unteraufrufe.
- **Objekttyp:** Spezifiziert Joinpoints anhand des Objekttyps, in dem die betroffene Methode definiert ist.

4.1.4 Advice-Typen

Durch verschiedene Advice-Typen kann bestimmt werden, zu welchem Zeitpunkt bezüglich des betroffenen Joinpoint der Advice-Code ausgeführt werden soll.

- **Before**: Der Advice-Code wird *vor* dem Methodenaufruf (bzw. Konstruktoraufruf, Methodenausführung, Feldbeschreibung usw.) ausgeführt.
- **After returning**: Der Advice-Code wird ausgeführt, *nachdem* die spezifizierte Methode ausgeführt und normal beendet wurde.
- **After throwing**: Der Advice-Code wird ausgeführt, nachdem die spezifizierte Methode *mit einer Ausnahme* beendet wurde.
- **After**: Der Advice-Code wird ausgeführt, nachdem die spezifizierte Methode *entweder normal oder mit einer Ausnahme* beendet wurde.
- **Around**: Der Advice-Code wird *anstelle* der spezifizierten Methode ausgeführt. Die Weiterausführung der ursprünglichen Methode ist mit Hilfe einer speziellen Anweisung namens `Proceed` möglich.

4.1.5 Context exposing

AspectJ ermöglicht den Zugriff auf Elemente im Kontext von Joinpoints („*context exposing*“). Joinpoints werden auf diese Weise zur Schnittstelle mit dem Komponentencode. In der Pointcut-Definition kann angegeben werden, welche Objekte im Kontext des Joinpoint dem Aspekt zur Verfügung gestellt werden. Es werden folgende Möglichkeiten geboten:

- **Zielobjekt:** Zugriff auf die Objektinstanz, zu der die vom Joinpoint betroffene Methode gehört.

- **Quellobjekt:** Zugriff auf die Objektinstanz, aus der die vom Joinpoint betroffene Methode aufgerufen wird (z.B. bei Aufruf-Pointcuts).
- **Argumente:** Zugriff auf ein oder mehrere Argumente der betroffenen Methode.
- **Rückgabewert:** Zugriff auf den Rückgabewert der betroffenen Methode.

Listing 4.2 gibt ein einfaches Beispiel zur Verdeutlichung:

```
pointcut pc(SomeClass sc, int x, int y):
      call(void SomeClass.SomeMethod(int, int))
      && target(sc)
      && args(x, y);
```

Listing 4.2 Beispiel für „context exposing“ in einem Pointcut

Mit dem Pointcut in Listing 4.2 werden alle Methodenaufrufe zu `void SomeClass.SomeMethod(int, int)` herausgesucht. Im Kopf des Pointcut werden die Objekte angegeben, auf welche zugegriffen werden kann: eine Instanz `sc` von `SomeClass` sowie zwei Integer `x` und `y`. Das darunter aufgeführte Schlüsselwort `target` gibt an, dass `sc` das Zielobjekt ist, zu dem die am Joinpoint aufgerufene Methode gehört. Das Schlüsselwort `args` gibt an, dass `x` und `y` Argumente dieser Methode sind.

Solche Angaben laufen zwangsweise auf eine genauere Spezifikation des Pointcut und somit eine *Einschränkung seiner Wertemenge* hinaus. Denn diese muss notwendigerweise auf solche Joinpoints reduziert werden, die sich auf Methoden mit tatsächlich nur zwei Integer-Argumenten beziehen und zum Objekttyp `SomeClass` gehören. In [Kicz97] und [Cacm01f] wird das Heraussuchen von Joinpoints durch Pointcuts sogar als *Filterung* bezeichnet.

4.1.6 Zugriff auf Attribute eines Joinpoint

Es ist möglich, aus dem Advice-Code heraus auf Jointpoint-Eigenschaften zuzugreifen. Dazu wird eine Schnittstelle verwendet, die Methoden deklariert, die Informationen zu den Joinpoint-Eigenschaften liefert. Auf Methoden dieser Schnittstelle kann mit dem Schlüsselwort `thisJoinPoint` aus dem Advice-Code heraus zugegriffen werden. Mit `thisJoinPoint` ist es möglich, statische Eigenschaften, aber auch dynamische Information über Objekte im Kontext des Joinpoint heranzuholen. Hier die wichtigsten Eigenschaften, die abgefragt werden können:

- **Art:** Art des Joinpoint (Aufruf, Ausführung usw.) als Zeichenkette.
- **Signatur:** Signatur der vom Joinpoint betroffenen Methode als Zeichenkette.
- **Argumente:** Methodenparameter als Array von Instanzen des Typs `Object`.
- **Zielobjekt** und **Quellobjekt:** Aufrufquelle und Aufrufziel.

- **Lage im Quellcode:** Zeile, Spalte, Klasse und Dateiname.
- **Kurzinformation:** Verschiedene Zeichenketten zur Kurzbeschreibung des Joinpoint.

4.1.7 Inter-type declarations

AspectJ ermöglicht die Einführung von Feldern, Methoden, Konstruktoren, Vererbungs- und Schnittstellenimplementierungsangeben in Klassen des Zielcodes. Die hinzugefügten Konstrukte sind aus dem Aspektcode *und* dem Komponentencode heraus zugreifbar, wobei sie noch mit Sichtbarkeitsregeln versehen werden können. Beispiele für eingefügte Deklarationen (in AspectJ *„inter-type declarations“* genannt) zeigt Listing 4.3:

```
aspect SomeAspect
{
       private int SomeClass.i = 5;
       public void SomeClass.setI(int i){this.i = i;}
      public SomeClass.new(int i){this.i = i;}

       declare parents: SomeClass extends SomeBaseClass;
       declare parents: SomeClass implements SomeInterface;
}
```

Listing 4.3 Beispiele für „inter-type declarations“ in AspectJ

In Listing 4.3 wird die Klasse `SomeClass` um ein Feld, eine Methode, und einen Konstruktor erweitert. Ferner wird deklariert, dass `SomeClass` von der Basisklasse `SomeBaseClass` erbt und die Schnittstelle `SomeInterface` implementiert.

4.1.8 Die Technik

Eine genaue Betrachtung der Implementierung von AspectJ ist wichtig für den späteren Entwurf der Eigenimplementierung. Ein Vorteil hierbei ist, dass es sich um ähnliche Zielsysteme handelt, da die Java-Plattform und das .NET-Framework in vielen Hinsichten miteinander verwandt sind: Virtuelle Maschine, Zwischencode, virtuelle Laufzeitumgebung usw. Laut [AspJ03] ist zurzeit [Hils04] das einzige Papier, das Details der Implementierung von AspectJ diskutiert.[1]

Nach [Hils04] besteht der Compiler von AspectJ aus zwei Hauptmodulen: einem *Front-End* und einem *Back-End*. Das Front-End stellt einen eigens entwickelten Compiler dar, der den in den Aspekten enthaltenen Java- und AspectJ-Code zu Bytecode mit Zusatzinformationen über Advice und Pointcuts kompiliert. Das Back-End findet die Stellen, an welche Advice eingefügt werden soll und verwebt den Advice-Code mit dem Komponentencode. Bild 4.1 zeigt die grundlegende Arbeitsweise des Front-End:

[1] Die in [Hils04] beschriebenen Implementierungsdetails beziehen sich auf die Version 1.1 von AspectJ.

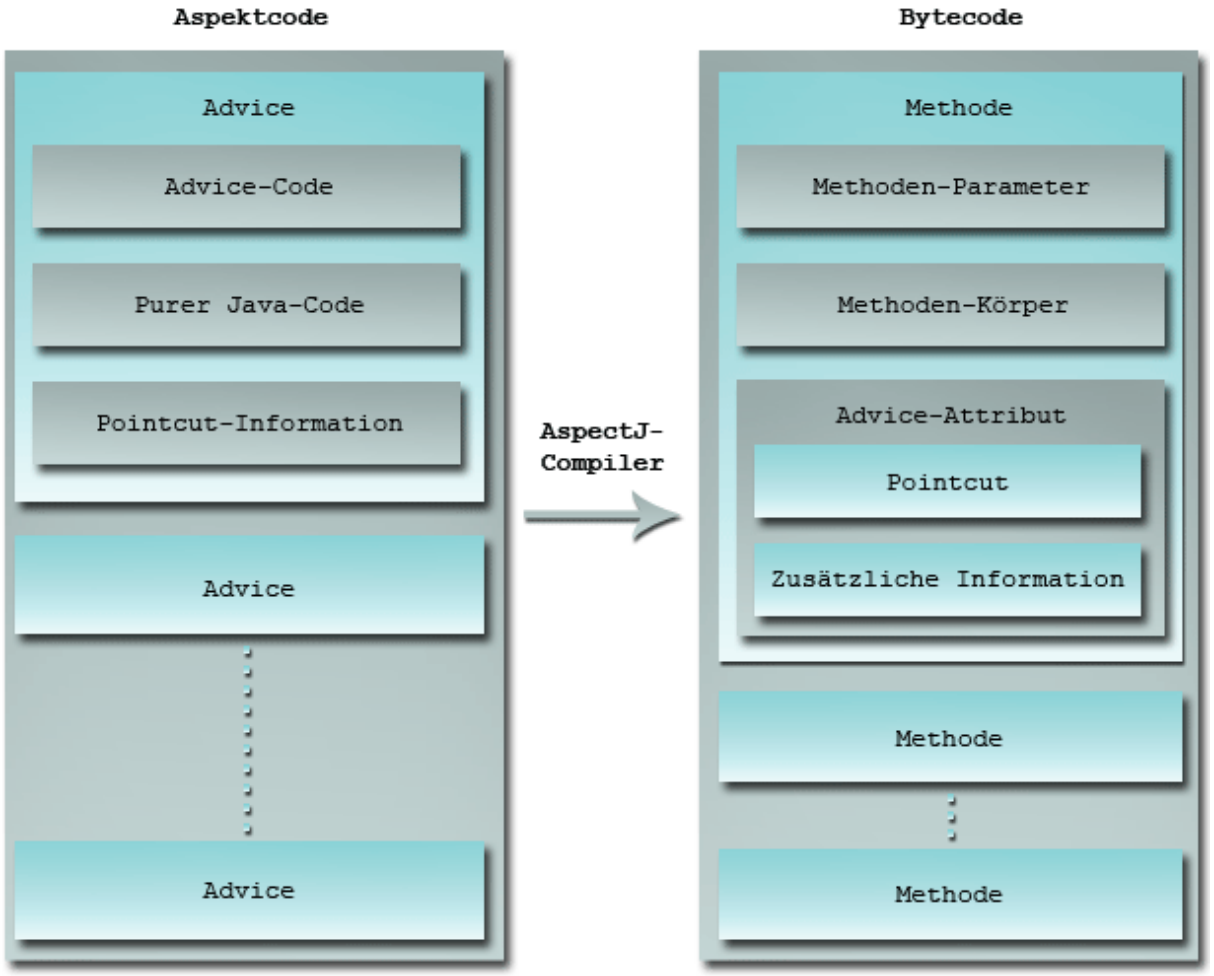

Bild 4.1 Front-End des AspectJ-Compilers

Jeder Advice wird zu einer gewöhnlichen Java-Methode in Bytecode kompiliert. Die Parameter dieser Methode sind die in dem Advice angegebenen Parameter. Unter Umständen werden diese noch um reflexive Information für die Funktionalität von `thisJoinPoint` erweitert, falls ein Aufruf einer `thisJoinPoint`-Methode im Advice-Code gefunden wird. Zur Optimierung der Performanz werden nicht verwendete Parameter entfernt.

Das in Bild 4.1 aufgeführte Advice-Attribut ist ein Standard-Java-Bytecode-Attribut und enthält drei Arten von Information:

- Kennzeichnung der Methode als Advice-Methode.
- Speicherung des Pointcut, der zum Advice gehört.
- Optional die Speicherung von Zusatzinformation zur Realisierung der `thisJoinPoint`- und `Proceed`-Funktionalität.

Die vorkompilierten Advice-Methoden müssen nun in den Komponentencode eingewoben werden. Außer bei Around-Advice, wo der Code inline eingewoben wird, geschieht dies durch den *Aufruf* der vorkompilierten Advice-Methode. Bevor jedoch Code verwoben werden kann, müssen intelligente Verfahren eingesetzt werden, die das Finden von Joinpoints vornehmen, die auf die Pointcut-Information im Advice-Attribut passen. Dieses sog. *Matching* und die anschließende Verwebung führt das Back-End des AspectJ-Compilers durch. Das Flussdiagramm in Bild 4.2 verdeutlicht die grundlegende Arbeitsweise des Back-End-Compilers:

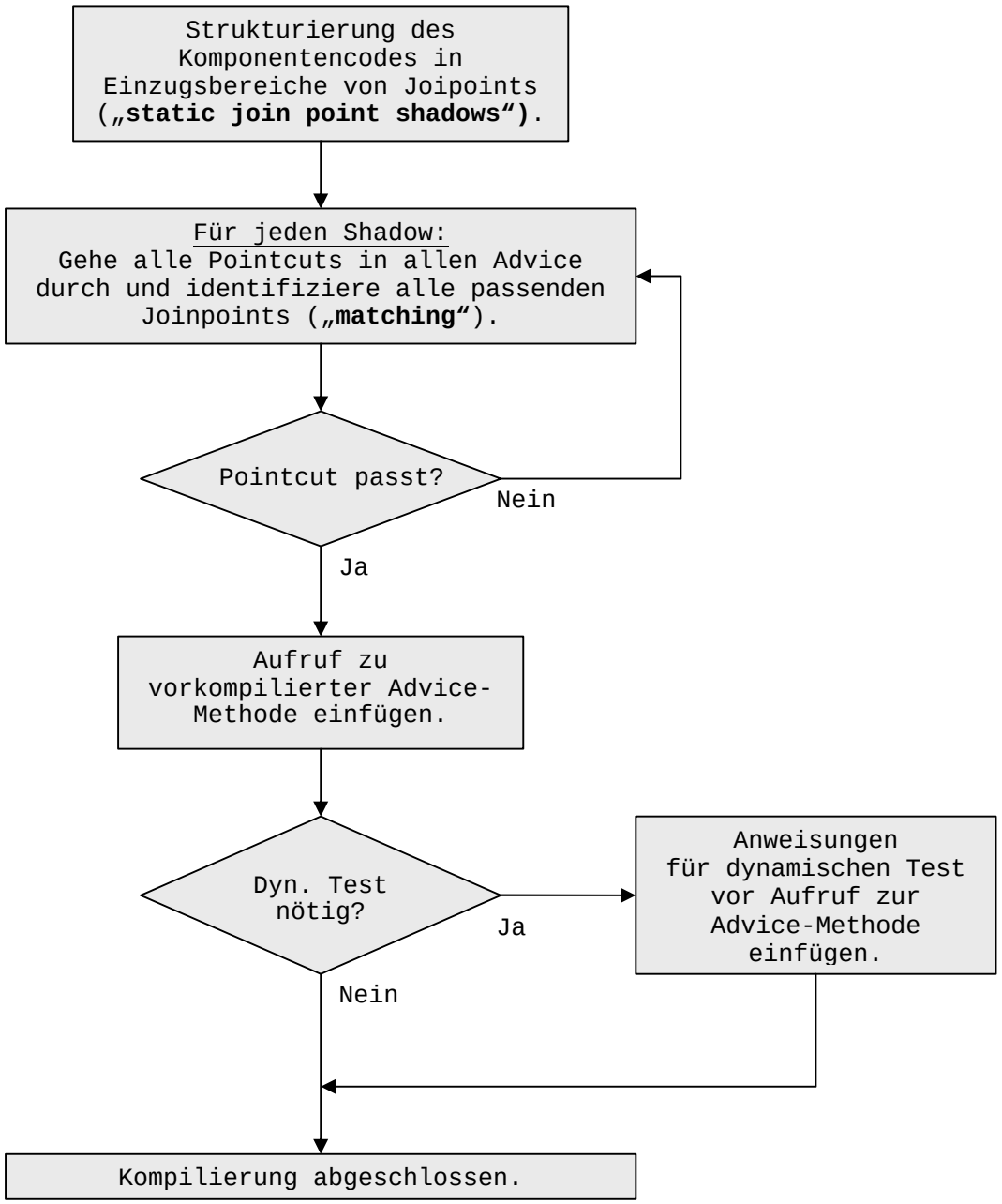

Bild 4.2 Back-End des AspectJ-Compilers

Das Back-End-Modul des Compilers geht den gesamten Zielcode durch und teilt ihn in sog. *Joinpoint-Schatten* („join point shadows“) ein. Es handelt sich dabei um statische Codebereiche, die potenziell zu einem Joinpoint gehören können (z.B. ein ganzer Methodenkörper). Joinpoint-Schatten sind Datenstrukturen, die folgende Informationen enthalten:

- **Art:** Art des Joinpoint, z.B. Ausführung, Aufruf usw.
- **Signatur:** Signatur der zugehörigen Methode bzw. des Konstruktors oder Feldes
- **Bytecodesegment:** Definiert die Region im Bytecode
- **Lage im Quellcode:** Angabe über Zeile, Dateinamen usw.
- **Quellobjekt und Zielobjekt:** Aufrufquelle und Aufrufziel
- **Argumente:** Parameter der betroffenen Methode bzw. des Konstruktors

Man beachte, dass durch dieses Verfahren ein erheblicher Overhead entsteht, da wirklich der gesamte Quellcode in Bereiche für potenzielle Advice-Methodenaufrufe eingeteilt wird und für jeden dieser Bereiche eine Datenstruktur für einen Schatten erstellt werden muss. Bisher sind laut [Hils04] an diesem Verfahren nur an einigen Stellen Optimierungen durch Heranziehung von Metainformationen vorgenommen worden.

Jeder Joinpoint-Schatten wird nun mit sämtlichen Pointcuts verglichen, die in allen vorkompilierten Advice-Methoden verwendet werden. Dieser Matching-Prozess berücksichtigt zunächst nur Vergleiche von statischer Information. Damit zur Laufzeit die Advice-Methode nicht mit falschen Parametern aufgerufen wird, werden unmittelbar vor dem eingewobenen Methodenaufruf Anweisungen für einen dynamischen Test eingefügt, der eine Typprüfung vornimmt und nur dann den Aufruf der Methode gestattet, wenn der Typ übereinstimmt.

4.1.9 Bewertung

AspectJ basiert auf dynamischen Joinpoints sowie Matching und statischer Verwebung auf Bytecodeebene. Damit ermöglicht AspectJ höchste Transparenz und Anwendbarkeit für den Komponentencode. Die Auffassung von Joinpoints als dynamische Stellen im Flussgrafen des Programms bringt zahlreiche Vorteile in der Flexibilität und den Ausdrucksmöglichkeiten der Sprache. Diese gewinnen nicht zuletzt dank vieler aktiver Diskussionsforen von Version zu Version stark an Umfang, so dass man durch die vielen gebotenen Möglichkeiten, die sich teilweise sehr ähnlich sind, sehr leicht in Verwirrung geraten kann. Die Verlagerung der Verwebung auf den Bytecode erlaubt präzise Eingriffe in den Komponentencode. Besonders Sprünge werden dadurch einfacher realisierbar, da Java-Bytecode Sprünge zu konkreten Zeilennummern erlaubt [Hils04].

4.1.10 Fazit und Relevanz für das Projekt

Eine eingehende Untersuchung von AspectJ lieferte sehr aufschlussreiche Erkenntnisse über die Anforderungen und Möglichkeiten einer AOSE-Implementierung. Darunter fallen z.B. flexible Ausdrucksmöglichkeiten zur Beschreibung von Pointcuts, verschiedene Typen von Advice, Zugriff auf Objektinstanzen im Kontext von Joinpoints und dessen Metainformationen, sowie Möglichkeiten zur statischen Erweiterung des Klassenmodells. Diese Eigenschaften sollten maßgebend für den geplanten Entwurf der Eigenimplementierung sein, wenn auch sicher nicht der enorme Funktionsumfang zu aller Vollständigkeit berücksichtigt werden kann.

4.2 Das HPI und LOOM .NET

4.2.1 Überblick

Das **H**asso-**P**lattner-**I**nstitut (HPI) an der Universität Potsdam ist durch hohe Aktivität im Bereich AOSE bekannt. Begonnen hat die Arbeit auf dem Gebiet mit Softwareprojekten und Publikationen rund um das Team von Andreas Polze. Mittlerweile gibt es am HPI sogar Vorlesungen zu AOSE. Das HPI hat darüber hinaus zwei bedeutende Implementierungen veröffentlicht [Loom04]:

- **LOOM .NET:** Ein statischer Weaver für .NET [Loom03].
- **Rapier-Loom.NET:** Ein dynamischer Weaver für .NET [RaDo04].

4.2.2 Die Technik

LOOM[1] .NET unterstützt den Programmierer beim Erstellen von statischen Proxy-Klassen (vgl. Abschnitt 3.3.1). Der Benutzer erstellt *Schablonen*, die mit beliebigen Klassen und Klassenmembern verwoben werden können. Bei der Verwebung unterstützt ihn eine grafische Oberfläche. Die Verwebung vollzieht sich auf Reflection-Ebene. Das Ergebnis sind Proxy-Klassen nach dem Muster von Listing 3.1 auf Seite 32. Die Schablonen bestehen aus C#-Code und *Platzhaltersymbolen*. Durch geeignete Textersetzung und Codegenerierung verknüpft LOOM.NET die Schablonen zu kompilierfähigen Einheiten.

4.2.3 Die Benutzerschnittstelle

Pro Aspekt kann der Benutzer je eine Schablone für Namensräume, Klassen, Methoden, Konstruktoren oder Felder erstellen. Jede Schablone wird in einer separaten Datei gespeichert. Alle Aspekte werden in einem zentralen Verzeichnis archiviert und können so

[1] „loom“ = Webstuhl

auf einfache Weise wieder verwendet werden. Die Zuordnung von Schablonen zu Aspekten regelt eine XML-Datei, wie Listing 4.4 zeigt:

```
<Aspect Name="Tests.TRACING">
	<Tag Type="file" Name="NAMESPACE">namespace_TRACING.tpl</Tag>
	<Tag Type="file" Name="METHOD">method_TRACING.tpl</Tag>
	<Tag Type="text" Name="BASECLASS">
	/*[BASENAMESPACE]*/./*[CLASSNAME]*/</Tag>
<Aspect/>
```

Listing 4.4 Beispiel für die Repräsentation eines Aspekts in der XML-Konfigurationsdatei von LOOM .NET

Der Auszug aus der XML-Konfigurationsdatei in Listing 4.4 definiert, dass der Aspekt `TRACING` aus einer Namensraum-Schablone sowie einer Methoden-Schablone besteht, die in den jeweiligen tpl-Dateien beschrieben sind. Im letzten Tag-Parameter wird der vollständige Name der Basisklasse spezifiziert. Die zwischen den Symbolen `/*[` und `]*/` angegebenen Zeichenketten sind die oben erwähnten Platzhaltersymbole. Listing 4.5 zeigt die in den tpl-Dateien definierten Schablonen für Namensräume und Methoden:

```
namespace /*[ASPECTNAME]*/Proxy
{
	class Tracer
	{
		public static string Params2Str(params object[] args)
		{
			string ret = "";
			foreach(object arg in args)
				ret += arg + " ";
			return ret;
		}
	}
	/*[CLASSDEFINITION]*/
}
```

```
public /*[MODIFIER]*/ /*[RESULTTYPE]*/ /*[METHODNAME]*/
(/*[PARAMDECLARATION]*/)
{
	Console.WriteLine("Before: /*[METHODNAME]*/ ",
		Tracer.Params2Str(/*[PARAMARRAY]*/));
	/*[RETVALINIT]*/
	/*[RETVALASSIGN]*/base./*[METHODNAME]*/(/*[PARAMLIST]*/);
	Console.WriteLine("After: /*[METHODNAME]*/");
	/*[RETVALRETURN]*/
}
```

Listing 4.5 Beispiele für Schablonen in LOOM .NET

Wie Listing 4.5 zeigt, enthält der Namensraum des `TRACING`-Aspekts eine Hilfsklasse `Tracer` sowie einen Platzhalter für die Klassen-Schablone. An dieser Stelle würde LOOM.NET den Text der Klassen-Schablone einfügen. Da jedoch keine Klassen-Schablone definiert ist, generiert LOOM.NET den Code selbst. Es erstellt eine Klasse, in welche die Methoden-Schablone von Listing 4.5 eingefügt wird. Diese Methoden-Schablone benutzt die in der Namensraum-Schablone definierte Klassenmethode `Tracer.Params2Str`. Auf das

Zielobjekt im Komponentencode wird mit `base` zugegriffen. Auf diese Weise erreicht man die Implementierung von Before-, After- und Around-Advice. Des Weiteren wird in Listing 4.5 deutlich, dass vordefinierte *Makros* durch Platzhaltersymbole aktiviert werden können. Auf diese Weise kann z.B. ein Array von Methodenparametern angefordert werden.

4.2.4 Beurteilung

Da LOOM .NET auf dem statischen Proxy basiert, ist es auch von dessen Schwachstellen betroffen, die bereits in Abschnitt 3.3.3 dokumentiert wurden. Vor allem ist LOOM .NET nicht dazu konzipiert worden, mit komplexen Matching-Regeln Ziel-Joinpoints zu bestimmen, wie dies etwa bei AspectJ der Fall ist. Für viele Applikationen mag dies aber durchaus genügen. Die generierten Proxy-Klassen müssen von Hand in den Komponentencode integriert werden. Ferner können Aspekte nur auf öffentliche Member angewendet werden.

LOOM .NET ist jedoch dank seiner konzeptionellen Einfachheit sehr kompakt und leicht zu erlernen. Die logische Aufteilung von Aspekten in hierarchische Ebenen (Namensraum, Klasse, Klassenmember) erlaubt eine übersichtliche Strukturierung der Problemlösung. Die Benutzung von Schablonen sowie die Angabe der Kerninformation zentral in einer XML-Datei unterstützen die Wiederverwendbarkeit von Aspekten. Durch Weaving auf MSIL-Ebene ist die Sprache des Komponentencodes nicht vorbestimmt. Einzig der Code in den Schablonen ist auf C# begrenzt.

4.2.5 Fazit und Relevanz für das Projekt

Zusammenfassend lassen sich aus der Untersuchung von LOOM .NET folgende wichtigen Erkenntnisse ziehen, die auch für die geplante Eigenimplementierung von Bedeutung sein könnten:

- Die Implementierung von AOSE-Logik ist durch statische Proxy-Komponenten einfach realisierbar, muss aber mit einigen Einschränkungen erkauft werden.
- Die Darstellung von grundlegenden Zusammenhängen kann durch Benutzung deklarativer Mittel wie XML oder Zeichenketten in Kommentaren einfach realisiert werden.
- In der Beschreibung von AOSE-Sachverhalten wie z.B. Verwebungsregeln treten wiederholt bestimmte Schemata auf, was durch den Einsatz von Entwurfsmustern wie z.B. Schablonen oder Platzhaltern zur Textersetzung ausgenutzt werden kann.
- Bei Weaving und Matching auf MSIL-Ebene stellt Reflection ein hilfreiches Mittel dar.

4.3. Rapier-Loom.Net

4.3.1 Überblick

Anders als sein Vorgänger LOOM .NET arbeitet Rapier[1]-Loom.Net [Loom04] mit Laufzeitverwebung und dynamischen Proxies (vgl. Abschnitt 3.3.2). Als Benutzerschnittstelle dient eine API zur Definition und Zuweisung von Aspekten. Weitere Softwarekomponenten werden nicht benötigt. Zur Spezifikation von Pointcuts dienen Attribute. Auf diese Weise hat Rapier-Loom einen wesentlichen Schritt in Richtung Integrierung in die .NET-Entwicklungsumgebung, Flexibilität sowie Funktionsumfang getan.

4.3.2 Die Technik

Rapier-Loom verwebt Komponenten- und Aspektcode auf Klassenebene, indem es zur Laufzeit Proxy-Klassen in temporäre Assemblies emittiert (vgl. Abschnitt 3.4.2). Damit die Proxy-Klassen anstelle der Zielklassen verwendet werden können, müssen alle Zielklassen mit `Weaver.CreateInstance` erstellt werden. Dieser Fabrikmethode gibt man Instanzen von zuvor erstellten Aspekt-Objekten mit. Aspekt-Objekte sind Instanzen von Klassen, die von `Loom.Aspect` abgeleitet sind. Die Alternative zur Verwendung von `Weaver.CreateInstance` ist das Versehen der Zielklasse mit einem Attribut, das dem Klassennamen der Aspekt-Klasse entspricht. Dies ist möglich, da `Loom.Aspect` von `System.Attribute` abgeleitet ist. Diese Lösung legt die Zuweisung von Aspekten allerdings auf die Kompilierzeit fest.

4.3.3 Die Benutzerschnittstelle

Von `Loom.Aspect` abgeleitete Aspekt-Klassen können Advice-Methoden und Pointcut-Deklarationen, aber auch beliebig viele andere Konstrukte enthalten. Advice-Methoden sind gewöhnliche Methoden, die mit Attributen zur Deklaration von Pointcuts versehen sind. Pointcuts können anhand folgender Möglichkeiten bestimmt werden:

- **Name:** Methoden werden anhand von Methodennamen einbezogen oder ausgeschlossen.
- **Typ:** In einem bestimmten Typ deklarierte Methoden werden einbezogen oder ausgeschlossen.
- **Attribut:** Mit einem bestimmten Attribut versehene Methoden werden einbezogen oder ausgeschlossen.

[1] Ein „rapier loom“ ist laut [RaDo04] eine bestimmte Art von Webstuhl, der das Weben einfach, aber nicht immer schnell macht.

- **Ausschluss in vererbten Klassen:** Gibt an, dass vererbte Methoden der Zielklasse ausgeschlossen werden.
- **Alle:** Alle Methoden werden einbezogen oder ausgeschlossen.

Bei der Angabe von Ziel-Membern ist eine Auswahl zwischen Methoden und Konstruktoren möglich. Pointcut-Attribute lassen sich mehrfach anbringen, sodass ähnliche Ausdrücke formuliert werden können wie mit den aus AspectJ bekannten Operatoren `&&`, `||` und `!`.

Als Parameter von Pointcut-Attributen gibt man den Advice-Typ an. Rapier unterstützt die gleichen Advice-Typen wie AspectJ bis auf den Typ „after“ [RaDo04]. Des Weiteren enthalten Aspektklassen Datenstrukturen zur Kapselung des Aufrufkontextes, bieten also mit `thisJoinPoint` verwandte Funktionalität (vgl. Abschnitt 4.1.6). Auch die von AspectJ bekannte Spezifikation von Pointcuts anhand von Wildcards ist mit Rapier realisierbar, indem in den Signaturen der Advice-Methoden Typen von Basisklassen angegeben werden, z.B. `object` für den Rückgabetyp oder `params object[]` für die Parametertypen. Listing 4.6 zeigt ein Beispiel für die Verwendung der API von Rapier-Loom.NET:

```
using System;
using Loom;
using Loom.ConnectionPoint;

namespace TracingExample
{
	public class TraceAspect : Aspect
	{
		[Include(typeof(TargetClass))]
		[Call(Invoke.Instead)]
		public void Trace(object[] args)
		{
			object instance = Context.DeclaringType;
			string method = Context.MethodName;
			Console.WriteLine(instance.ToString()+
				"." + method);
		}
	}
}
```

```
using System;

namespace TracingExample
{
	public interface ITargetCode
	{
		void Quad(int x);
	}

	public class TargetClass : ITargetCode
	{
		public void Quad(int x)
		{
			Console.WriteLine(x*x);
		}
	}
}
```

Listing 4.6 Tracing-Beispiel in Rapier-Loom.Net (Fortsetzung auf nächster Seite)

```
using System;
using Loom;

namespace TracingExample
{
	class TracingExampleClient
	{
		static void Main(string[] args)
		{
			TraceAspect ta = new TraceAspect();
			ITargetCode tc =(ITargetCode)Weaver.CreateInstance
					(typeof(TargetClass), null, ta);
			tc.Quad(2);
		}
	}
}
```

Listing 4.6 Tracing-Beispiel in Rapier-Loom.Net (Fortsetzung)

Der Aspekt von Listing 4.6 definiert Around-Advice, der auf alle Member der Klasse `TargetClass` angewendet wird, die `void` als Rückgabewert und eine beliebige Anzahl von Parametern haben. Der Advice-Code ermittelt Kontextinformation zum aktuellen Joinpoint. Die Methode der Zielklasse ist aus den im kommenden Abschnitt genannten Gründen in einer Schnittstelle deklariert. Im Clientcode muss die Aspekt-Klasse instanziert und die Zielklasse mit `Weaver.CreateInstance` angelegt werden.

4.3.4 Beurteilung

Rapier-Loom.Net ist eine gelungene AOSE-Implementierung mit vielen Vorteilen, die AspectJ-ähnliche Funktionalität bietet, ohne dass Spracherweiterungen oder externe Softwarewerkzeuge benutzt werden müssen. Durch die Verwebung auf MSIL-Ebene ist die Codierung sprachunabhängig. Gleichzeitig erlaubt es Rapier, durch dynamische Verwebung Aspekte erst zur Laufzeit an Objekte zuordnen zu müssen. Die in Abschnitt 3.2.4 beschriebenen Eigenschaften der dynamischen Bindung führen allerdings zu Einschränkungen in der Anwendbarkeit. Zudem beeinträchtigt die Anwendung von dynamischen Proxy-Klassen die Transparenz für den Clientcode. Ferner führt die dynamische Verwebung Performanceeinbußen mit sich. Nach [RaDo04] kann die Objekterstellung mit `Weaver.CreateInstance` um einem Faktor von bis zu 100 langsamer sein als mit dem `new`-Operator.

4.3.5 Fazit und Relevanz für das Projekt

Eine eingehende Untersuchung von Rapier-Loom.Net demonstrierte den Einsatz vieler interessanter Techniken:

- Laufzeitverwebung mit dynamischen Proxies.
- Einsatz einer Klassenbibliothek zur Definition von Aspektcode statt wie bei AspectJ einer Spracherweiterung und eines Compilers.

- Benutzung von `params object[]` in der Parameterliste der Advice-Methode als Platzhalter für beliebige Parameter.
- Einsatz von Attributen als deklaratives Mittel zur Beschreibung von Pointcuts und Advice.
- Verwendung von Attributen zur Kennzeichnung von Joinpoints.
- Mehrfache Angabe von Attributen zur Verkettung von Pointcut-Deklarationen.

4.4 Weitere Projekte

4.4.1 AspectC++

AspectC++ ist eine überwiegend deutsche Entwicklung und entstand aus der Zusammenarbeit zwischen der Universität Erlangen-Nürnberg und einer Magdeburger Softwarefirma [AspC04]. AspectC++ orientiert sich im Wesentlichen an AspectJ und zeichnet sich durch eine vergleichbare Syntax sowie einen ähnlichen, wenn auch geringeren, Funktionsumfang aus [Urba04]. Größere Unterschiede gibt es in der Implementierung, da Matching und Weaving nicht auf Bytecode-, sondern auf Quellcodeebene durchgeführt werden. Trotz großer Ähnlichkeit mit seinem Java-Pandant gibt es einige erwähnenswerte Unterschiede und Ergänzungen. Hier die Wichtigsten von ihnen:

- **Rein virtuelle Pointcuts:** Dies ermöglicht abstrakte, wieder verwendbare Aspekte.
- **Introduction:** Im Gegensatz zu AspectJ spezifiziert AspectC++ Zielklassen für eingeführte Elemente *anhand von Pointcuts*.
- **Advice-Reihenfolge steuerbar:** Falls mehrere Aspekte oder mehrere Advice um einen Joinpoint konkurrieren, können Präzedenzen definiert werden.
- **Nichtdestruktive Verwebung:** Der Komponentencode bleibt nach der Verwebung erhalten (vgl. Abschnitt 3.2.2).

Der Compiler basiert auf statischem Verweben von Aspekt- und C++-Code zu reinem C++-Code [Spin04]. Es handelt sich also um einen Codetransformator, der ganze Verzeichnisbäume mit Code in andere Verzeichnisbäume transformiert. Ein weiterer Modus erlaubt es, nur einzelne Dateien zu transformieren. Der verwobene Code kann mit den gängigen Compilern (GNU g++, Borland C++ oder Microsoft VisualC++) zu ausführbarem Code kompiliert werden [Spin04]. Dies hat den Nachteil einer etwas umständlichen Benutzung sowie eines nicht ganz trivialen Konfigurationsaufwandes, bedeutet aber, dass man den fertig verwobenen Code genau untersuchen und gegebenenfalls noch optimieren kann.

4.4.2 AspectWerkz

AspectWerkz [AWer04] ist eine Java-Implementierung, die in letzter Zeit viel an Popularität gewinnt. AspectWerkz stellt dem Benutzer zwei Verwebungs-Modi zur Auswahl. Der „Online“-Modus verwebt Aspekt- und Komponentencode zur Klassenladezeit und lässt somit den Quellcode unangetastet. Der „Offline“-Modus hingegen ist destruktiv [AWHW04]. Darüber hinaus erlaubt AspectWerkz durch dynamische Bytecodemanipulation das Hinzufügen und Entfernen von Advice zur Laufzeit. AspectWerkz verzichtet im Gegensatz zu AspectJ auf Spracherweiterungen. Lediglich eine `import`-Anweisung wird benötigt. AspectWerkz benutzt eine XML-Datei zur Zuordnung von Pointcuts und Advice zu Aspekten, wie Listing 4.7 zeigt:

```
<aspectwerkz>
 <system id="AspectWerkzExample">
  <package name="testAOP">
   <aspect class="MyAspect">
    <pointcut name="greetMethod"
              expression="execution(*testAOP.HelloWorld.greet(..))"/>
    <advice name="beforeGreeting" type="before" bind-to="greetMethod"/>
    <advice name="afterGreeting" type="after" bind-to="greetMethod"/>
   </aspect>
  </package>
 </system>
</aspectwerkz>
```

Listing 4.7 Zuordnungsdatei von AspectWerkz. Quelle: [AWHW04]

Die Definitionen von Pointcuts und Advice lassen sich optional aus der XML-Datei herausnehmen und stattdessen als JavaDoc-Elemente im Quellcode angeben. Für diese Variante muss jedoch auf ein zusätzliches Tool zugegriffen werden, das die JavaDoc-Angaben durchgeht [AWHW04]. Listing 4.8 zeigt einen Aspekt mit JavaDoc-Angaben:

```
package testAOP;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspectWithAnnotations
{
	/**
	 * @Before execution(* testAOP.HelloWorld.greet(..))
	 */
	public void beforeGreeting(JoinPoint joinPoint)
	{
		System.out.println("before greeting...");
	}

	/**
	 * @After execution(* testAOP.HelloWorld.greet(..))
	 */
	public void afterGreeting(JoinPoint joinPoint)
	{
		System.out.println("after greeting...");
	}
}
```

Listing 4.8 Aspekt-Definition in AspectWerkz. Quelle: [AWHW04]

4.4.3 JBoss-AOP

Große AOP-Aktivität ist auf dem Gebiet der Middleware zu verzeichnen. Bekannte Beispiele hierfür sind die AOP-Komponentenarchitektur Java Aspect Components [JAC04] sowie die Integration von AOP-Funktionalität in den verbreiteten Applikationsserver JBoss [JBos04]. Seit der Version 4.0 DR3 verfügt JBoss über eine eingebaute AOP-Architektur. Sie stellt Dienste wie Persistenz, Caching, Replikation usw. zur Verfügung und ermöglicht darüber hinaus die Programmierung benutzerdefinierter Aspektkomponenten. Als fundamentale Techniken benutzt JBoss-AOP Interception, dynamische Codeerzeugung durch Javassist [JAss04] und dynamische Proxies. Interceptors (d.h. Advice) können zur Laufzeit eingespielt werden, sodass Klassen ihre Dienste dynamisch beziehen können. Zum Funktionsumfang gehört ein flexibles Pointcut-Modell, mit dem sich beispielsweise Aufrufstelle und Kontrollfluss berücksichtigen lassen. Zu den Zielcodeelementen gehören Felder, Methoden und Konstruktoren ohne Beschränkung auf Zugriffsmodifizierer. Dabei verzichtet der Benutzer auf Spracherweiterungen und spezifiziert Pointcuts und Advice in einer XML-Datei.

4.4.4 AspectC#

AspectC# entstand im Rahmen einer Master-Dissertation von Howard Kim [Kim02] an der Distribited Systems Group des Trinity College Dublin [ACSh03]. AspectC# macht sich für die Verwebung das Document Object Model (vgl. Abschnitt 3.4.2) zunutze. AspectC# geht Aspekt-, sowie Komponentencode getrennt mit einem eigens geschriebenen Parser durch, der anfangs für eine Untermenge der C#-Sprachsyntax entwickelt wurde. AspectC# erstellt aus den gewonnenen Tokens einen DOM-Syntaxbaum. Aspekt- und Komponentencode werden anschließend auf Ebene des Syntaxbaums verwoben. Der CodeDOM-Codegenerator erstellt aus dem verwobenen Syntaxbaum C#-Code. Als weitere interessante Eigenschaft verwendet AspectC# eine XML-Datei für das Mapping von Aspektcode zu Komponentencode [Kim02].

4.5 Mit AOSE verwandte Projekte und Software

4.5.1 Subjektorintierte Programmierung und Hyper/J

Hyper/J [HypJ03] führt den Grundgedanken der AOSE entscheidend weiter. Hyper/J benutzt eine Technik, die von ihren Autoren als ***M**ulti-**D**imensional **S**eparation **o**f **C**oncerns* (MDSoC) bezeichnet wird [Cacm01d]. In Abschnitt 2.2.2 wurde erwähnt, dass sich jeder Sachverhalt als Dimension im Problemfeld auffassen lässt. Diese Dimension wird in der MDSoC-Technik als *Hyperslice* bezeichnet. Jede Hyperslice repräsentiert ein separates unabhängiges Softwaremodell, sodass viele Dekompositionen eines Softwaresystems nebeneinander existieren können. Mit Hyper/J kann man diese Dekompositionen beliebig miteinander kombinieren. MDSoC entwickelte sich aus der in [Harr93] publizierten *subjektorientierten Programmierung* („Subject-Oriented Programming"), in der Hyperslices als Subjekte bezeichnet werden.

4.5.2 Adaptive Methoden und die DJ library

Adaptive Methoden („Adaptive Methods") kapseln das Verhalten von Operationen, die sich über viele Objekte erstrecken, an einer Stelle und abstrahieren dabei von der Klassenstruktur [Cacm01c]. Der Programmierer definiert zweierlei Elemente: Eine Strategie zum Durchqueren von bestimmten Teilen des Zielcodes („traversal strategy") und Aktionen, die bei Erreichung einer Zielklasse ausgeführt werden sollen („adaptive visitor"). Eine Implementierung dieser Technik ist die DJ library [DJlb01]. Die DJ library erlaubt die Angabe einer „traversal strategy" durch Zeichenketten mit speziellen Schlüsselwörtern. Mit Reflection wird ein Graf erstellt, der die Klassenstruktur der gesamten Applikation darstellt. Den „adaptive visitor" implementiert man in Form des Besuchermusters, sodass der Graf anhand der „traversal strategy" durchquert wird und dabei bestimmten Code ausführt.

4.5.3 Kompositionsfilter und ComposeJ

Kompositionsfilter („Composition Filters") [CoFi01] sind eine Programmiertechnik, die von Objekten gesendete und empfangene Nachrichten analysiert und bearbeitet [Cacm01e]. Hierzu werden sog. Kompositions-Filter-Klassen erstellt, welche unter anderem die Klassen der Geschäftslogik als innere Klassen enthalten. Kompositions-Filter-Klassen implementieren eine Reihe von Eingangs- und Ausgangsfiltern, die jede ein- und ausgehende Nachricht passieren muss. Insofern zeigen sich Ähnlichkeiten mit der in Abschnitt 3.4.4 vorgestellten .NET-Interception. ComposeJ ist ein Compiler, der das Einweben von Kompositions-Filter-Klassen ermöglicht [ComJ01].

Kapitel 5: Realisierung

Nach umfangreicher Analyse der Problematik, Techniken und Implementierungen der AOSE soll nun der Entwurf und die Implementierung eines Softwarewerkzeuges durchgeführt werden, das AOSE unter .NET ermöglicht. Zunächst soll in Abschnitt 5.1 die Wahl der grundlegenden Techniken getroffen werden. Anschließend werden in Abschnitt 5.2 im Grobentwurf Anforderungen an die Software spezifiziert sowie Systemarchitektur und Anwendungsfälle beschrieben. Der Feinentwurf in Abschnitt 5.3 dokumentiert den Entwurf der einzelnen Komponenten mit Hilfe von Klassen- und Sequenzdiagrammen. Abschnitt 5.4 beschreibt die Implementierung anhand wichtiger Kernpunkte.

Zum besseren Verständnis des Entwurfs und der Implementierung möge man auf die beiliegende CD zurückgreifen. Sie enthält das Quellprojekt der Software sowie eine systematische Einleitung in ihre Benutzung. Weitere Informationen zum Inhalt der CD gibt Anhang 2.

5.1 Systemidee

5.1.1 Allgemeine Richtlinien

Für die Entwicklung der Systemidee sollen die folgenden Richtlinien maßgebend sein:

1. Größtmögliche Einhaltung der in Abschnitt 3.1 entworfenen Anforderungsprofils
2. Als Vorbild für Funktionsumfang und Ausdrucksmöglichkeiten für den Benutzer soll AspectJ gelten.
3. Die Benutzerschnittstelle sollte kompakt und ihre Handhabung intuitiv sowie einfach erlernbar sein.
4. Ausnutzung der von .NET zur Verfügung gestellten Mechanismen wie Reflection und Attribute
5. Durchführung des Projekts in einem Zeitraum von 8 Wochen. In dieser Zeit muss die Software entworfen, implementiert, getestet und dokumentiert werden.

5.1.2 Entscheidungsfindung

Als Name der Implementierung wird *Spider.NET* gewählt, da er sowohl treffend[1] als auch einprägsam und nach Wissen des Autors noch nicht belegt ist. Die Implementierung wird in der Sprache C# durchgeführt.

Die Entscheidung für die grundlegende Technik fällt auf .NET-Interception. Dadurch kann auf die intensive Auseinandersetzung mit Quell- bzw. Zwischencode verzichtet werden, was sicherlich eine mühselige und etwas eintönige Implementierung mit sich führen würde. Besonders hinsichtlich Punkt 5 der obigen Aufzählung ist dieses Argument von entscheidender Bedeutung. Interception arbeitet mit dynamischer Verwebung, eröffnet also eine Vielzahl von Möglichkeiten wie z.B. Zuweisung von Aspekten zur Laufzeit, Sprachunabhängigkeit und einfache Benutzung der Software in Form einer Klassenbibliothek. Dies erfüllt die Richtlinie von Punkt 3. Der Verzicht auf Spracherweiterungen lässt sich mit Attributen kompensieren, was wiederum die Richtlinie von Punkt 4 erfüllt.

Es wird oft kritisiert, dass die reine Interception kein hinreichendes Mittel für die AOSE ist [Nolt04], [Pies03]. Vor allem bietet sie keine Möglichkeit, Pointcuts nach dem Vorbild von AspectJ zu definieren. Um die Richtlinien von Punkt 1 und 2 zu erfüllen, sind also noch weitere Schritte nötig. Gebraucht wird eine Benutzerschnittstelle, mit der man Aspekte, Pointcuts und Advice auf einer von der restlichen Applikation abstrahierenden Ebene spezifizieren kann. Eine Verwaltungseinheit muss die vom Benutzer vorgenommenen Deklarationen mit Reflection abfragen und in Anweisungen für die Interception-Infrastruktur umsetzen.

5.1.3 Verwebung

Die Verwebung in Spider.NET wird zur Laufzeit von der CLR vorgenommen. Mit Kontextattributen gekennzeichnete Objekte bekommen Proxies und Ketten von Nachrichtensenken zugewiesen (vgl. Abschnitt 3.4.4). Wenn Aspekte von `ContextAttribute` abgeleitete Klassen sind, können sie als Attribute an Zielklassen angebracht werden und implementieren für jedes Zielobjekt eine benutzerdefinierte Nachrichtensenke. Gleichzeitig können in diesen Aspekt-Klassen Advice-Methoden und Pointcut-Deklarationen auf eine Art platziert werden, die in Abschnitt 4.3 im Zusammenhang mit Rapier-Loom.Net demonstriert wurde. Bild 5.1 verdeutlicht diese Idee:

[1] Spinnen sind Weber.

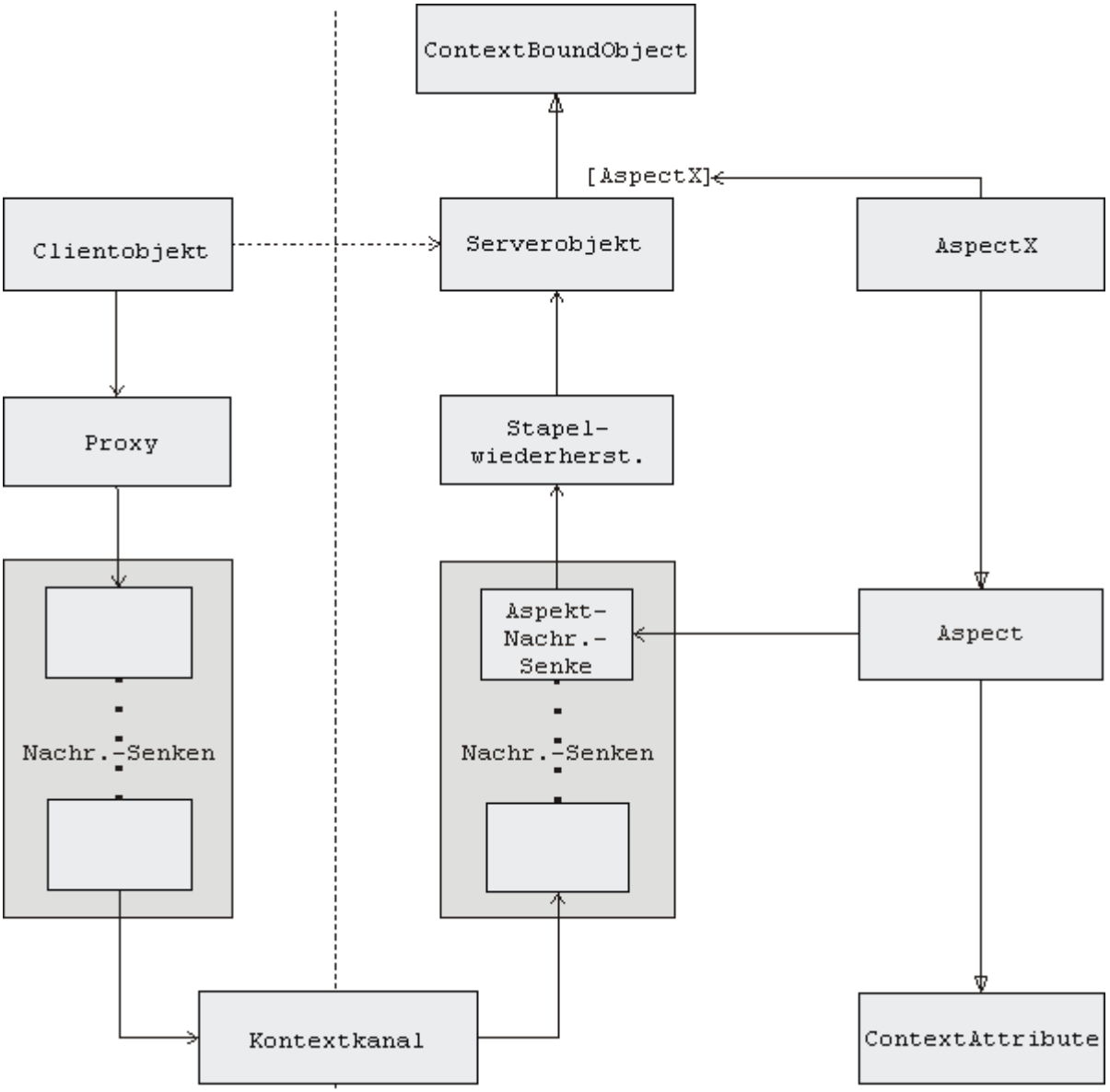

Bild 5.1 Aspekte und Interception in Spider.NET

Wie Bild 5.1 demonstriert, werden Aufrufe von Clientobjekten zu Serverobjekten von der Interception-Infrastruktur abgefangen. Der Client adressiert völlig transparent einen Proxy, der sich für den Client wie das Zielobjekt verhält. Der Aufruf wird in eine Nachricht umgewandelt, die dann von zahlreichen Nachrichtensenken bearbeitet wird. Anschließend wird die Nachricht durch den Kontextkanal geschickt, der sich nach außen wie ein Remoting-Kanal[1] verhält, aber lediglich zwei Kontexte miteinander verbindet. Serverseitig gelangt die Nachricht wieder durch eine Kette von Nachrichtensenken und wird schließlich durch den Stackbuilder in einen Aufruf umgewandelt. Die Methode des Serverobjekts kann nun aufgerufen werden.

[1] Ein Remoting-Kanal könnte z.B. ein Kanal sein, der das TCP-Protokoll implementiert und damit zwei Rechner verbindet.

Die Basisklasse für alle Aspekte ist `Aspect` und wird wie oben erwähnt von `ContextAttribute` abgeleitet. Für Aspekte zuständige Nachrichtensenken werden auf der *Serverseite* installiert, da Aspekte Zielobjekte adressieren sollen. In diesen Nachrichtensenken können die in den Aspekten definierten Advice-Methoden aktiviert werden. Dies kann *vor*, *nach* oder *anstelle* der Weiterleitung der Nachricht erfolgen. Ferner werden in den zurückkehrenden Nachrichten Ausnahmen signalisiert, sodass auch eine Implementierung des von AspectJ bekannten Advice-Typs „after throwing" möglich ist. Der Abfangmechanismus funktioniert im Zusammenhang mit Methoden, Konstruktoren, Feldern, Properties und Events.

Die Nachteile dieses Modells sind:

1. Jedes Zielobjekt muss von `ContextBoundObject` abgeleitet sein, damit es in dem eigens für das Objekt vorgesehenen Kontext erstellt wird.
2. Alle Zielobjekte müssen im Komponentencode mit Aspekt-Attributen versehen sein.
3. Nur Objekte als Ganze können ausgewählt werden und nicht z.B. einzelne Member.
4. Interception setzt nur bei kontextübergreifenden Aufrufen ein. Die Kommunikation zwischen zwei Methoden desselben Objekts wird somit nicht abgefangen.
5. Aus Punkt 4 folgt, dass nur öffentliche Member an der Interception beteiligt sein können.
6. Interception macht die Applikation langsamer. [Lost04] gibt die Geschwindigkeit mit 3000 Objekterstellungen oder 30000 Methodenaufrufen pro Sekunde an.

An den Punkten 4 bis 6 kann man ohne Veränderungen am .NET-Framework nichts ausrichten. Man kann höchstens versuchen, die Geschwindigkeit nicht noch wesentlich herabzusetzen. Die Punkte 1 bis 3 stellen wesentliche Einschränkungen hinsichtlich Transparenz, Flexibilität und Anwendbarkeit dar. Punkt 2 erzwingt zudem, dass die Entscheidung über die Zuweisung von Aspekten zu den Zielklassen zur Kompilierzeit erfolgt und nicht zur Laufzeit, was einen großen Vorteil der dynamischen Verbebung wieder zunichte macht. Für die in den Punkten 1 bis 3 aufgezeigten Probleme ist somit eine Lösung zu suchen.

5.1.4 Pointcut-Modell

Kontextattribute lassen sich bei Bedarf vererben, ebenso wie die Eigenschaften von `ContextBoundObject`. Folglich müssen nur die obersten Klassen der Ableitungshierarchie der Applikation mit dem Kontextattribut versehen und von `ContextBoundObject` abgeleitet sein. Dies ist die einzige Stelle im Komponentencode, an der Intimität (vgl. Abschnitt 3.1.6) erforderlich ist. Dadurch wird ein Maß an Transparenz erreicht, das für die

dynamische Verwebung verhältnismäßig hoch ist. Nachteilig sind die dadurch entstehenden Geschwindigkeitseinbußen. Der Benutzer hat somit die Wahl zwischen Transparenz und Performanz, je nachdem an welcher Stelle er die Aspekt-Attribute anbringt. Durch geeignetes Anbringen von Aspekt-Attributen bestimmt er eine Menge von potenziellen Zielklassen für Aspekte. Innerhalb dieser Menge kann er dann mit Pointcut-Deklarationen Mengen von Joinpoints bestimmen. Diese Pointcut-Deklarationen formuliert er entweder *statisch* mit Attributen im Aspektcode oder *dynamisch* mit Methodenaufrufen im Aspekt- oder Komponentencode. Mit dieser Technik ist ein Pointcut-Modell nach Vorbild von AspectJ oder Rapier-Loom.Net realisierbar und die in den Punkten 1 bis 3 formulierten Probleme lösbar.

Zur Realisierung des Pointcut-Modells muss Spider.NET folgendermaßen vorgehen:

1. Suche aller definierten Aspekt-Klassen und Advice-Methoden und Archivierung in einer Datenbasis (im Folgenden „*Applikationsbasis*“ genannt).
2. Suche aller Aspekt-Attribute und dadurch Bestimmung der Menge potenzieller Zielklassen (im Folgenden „*Domainmenge*“ genannt).
3. Suche von Pointcut-Deklarationen und dadurch Bestimmung der Menge von Joinpoints (im Folgenden „*Pointcutmenge*“ genannt) für jede Advice-Methode.
4. Archivierung der Pointcutmengen in der Applikationsbasis.
5. Beim Abfangen eines Aufrufs Suche der Zielmethode in den Pointcutmengen in der Applikationsbasis.
6. Falls ein für die Zielmethode entsprechender Joinpoint in der Applikationsbasis gefunden wurde, Aktivierung der für den Joinpoint definierten Advice-Methode.

Aus dieser Vorgehensweise lassen sich zwei wesentliche Entwurfsaufgaben erschließen:

- Für die Applikationsbasis muss eine Datenstruktur entworfen werden, in der Aspekte, Advice und Pointcuts in geeigneter Form repräsentiert werden.
- Für das Bestimmen der Pointcutmenge muss eine geeignete Matching-Strategie entworfen werden.

5.1.5 Matching

Folgend aus der obigen Aufzählung bzgl. Realisierung des Pointcut-Modells ist die Pointcutmenge eine Untermenge der Domainmenge. Die Pointcutmenge wird somit schrittweise aus der Filterung (vgl. [Kicz97] und [Cacm01f]) von Mengen auf verschiedenen Ebenen bestimmt. Tabelle 5.1 zeigt, welche Mengen dabei eine Rolle spielen und wie sie bestimmt werden:

Name	Symbol	Beschreibung
Grundmenge	G	Entspricht der Menge aller Klassen in der gesamten Applikation.
Domainmenge	D	Menge von Klassen, ausgewählt durch Angabe des Aspekt-Attributs. Für alle Objekte in D installiert die CLR die Interception-Infrastruktur. Das Kontextattribut kann am obersten Hierarchieelement angebracht sein. In dem Fall gilt D = G. Der kleinstmögliche Wert von D ist eine Klasse. Es gilt also K $\subseteq$ D $\subseteq$ G, wenn K die Menge ist, die einer Klasse entspricht.
Membermenge	M	Menge von Membern, die anhand von Pointcut-Deklarationen in Form von Attributen in der Aspekt-Klasse bestimmt werden. Herausgesucht werden alle Member, auf die eine Kombination von *Klassenname* und *Membername* passt, wobei auch Wildcards verwendet werden können.
Pointcutmenge	P	Menge von Membern, die anhand der Signatur der Advice-Methode bestimmt werden.
Joinpoint	J	Unteilbare Elementarmenge, die jeweils einen Ziel-Member beschreibt.
Leere Menge	Ø	Leere Menge. Falls Pointcut-Angaben gemacht werden, die auf keine Menge zutreffen, ist die Ergebnismenge leer.

Tabelle 5.1 Bestimmung von Joinpoints durch Mengeneinschränkung.

Es gilt also für die Mengen G, D, M, P, J und Ø:

$$Ø \subseteq J \subseteq P \subseteq M \subseteq D \subseteq G$$

Der Benutzer soll Member- und Pointcutmenge auf ähnliche Weise bestimmen können wie in Rapier-Loom.Net, d.h. durch Versehen der Advice-Methode mit Pointcut-Attributen und einer besonderen Signatur. Spider.NET muss diese Deklarationen durch geeignetes Matching in Joinpoints umwandeln, um sie dann in der Applikationsbasis zu speichern. Zwei Arten von Matching müssen implementiert werden:

- **High-Level-Matching:** Sucht eine Menge von Membern anhand einem Muster von Klassennamen und Membernamen (falls vorhanden) heraus. Dazu werden alle Klassen und Member der Domainmenge mit Angaben in Pointcut-Attributen verglichen. Das Resultat ist die Membermenge M.
- **Low-Level-Matching:** Sucht eine Menge von Membern anhand der Parametertypen sowie des Rückgabetyps (falls vorhanden) der Advice-Methode heraus. Dazu werden alle Signaturen der Membermenge mit der Signatur der Advice-Methode verglichen. Das Resultat ist die Pointcutmenge P.

5.2 Grobentwurf

5.2.1 Spezifikation der Anforderungen

Dieser Abschnitt gibt eine informale Spezifikation von Pflichten, welche die fertige Software zu erfüllen hat.

Das gesamte AOSE-System sollte durch eine klare Trennung von Komponentencode, Aspektcode und Weaver[1] gekennzeichnet sein. Der Weaver sollte in Form einer DLL benutzbar sein, auf die vom Komponenten- und Aspektcode aus verwiesen wird. Um den Einsatz von wieder verwendbaren Aspektkomponenten zu gewährleisten, sollen auch in externen DLLs befindliche Aspekte einbezogen werden können.

Die einzigen notwendigen und hinreichenden Anforderungen an den Komponentencode für die Anwendung von Aspekten sind:

- Hinzufügen eines Verweises auf die DLL des Weavers.
- Angabe der Ableitung von `ContextBoundObject` in der obersten Klasse der Klassenhierarchie.
- Angabe des Aspekts im Komponentencode als Klassenattribut in der obersten Klasse der Klassenhierarchie.
- Falls sich Aspekte in weiteren DLLs befinden, muss auch auf diese verwiesen werden. Dann muss auch Spider.NET mitgeteilt werden, in welchen Assemblies Zielklassen und Aspektklassen zu finden sind.

An keiner weiteren Stelle darf im Komponentencode das Prinzip der Transparenz verletzt sein. Damit der Benutzer nicht unnötig durch den Namen `ContextBoundObject` verwirrt wird, sollte er z.B. durch `AspectTarget` kaschiert werden.

Aspekt- und Komponentencode können in jeder beliebigen .NET-Sprache programmiert sein.[2] Jegliche Information, die über die Grammatik der .NET-Sprache hinausgeht, wird mit Hilfe von Attributen angegeben. Aus ergonomischen Gründen sollen keine externen Konfigurationsdateien benutzt werden müssen.

Die Benutzerschnittstelle sollte von klarem und kompaktem Umfang sein. Sie soll einfach benutzbar und erlernbar sein sowie die hinter ihr verborgene Technik verdecken. Spider.NET soll die Advice-Typen „before“, „around“, „after“ und „after throwing“ unterstützen.

[1] Mit Weaver ist im Folgenden Spider.NET gemeint.
[2] Spider.NET soll jedoch nur im Zusammenhang mit C# getestet werden.

Innerhalb eines Aspekts soll es nur eine Advice-Methode von jedem Typ geben können. Ferner sollen die aus AspectJ bekannten Funktionen `proceed` und `thisJoinPoint` zur Verfügung stehen. „thisJoinPoint“ soll ferner auch die Funktion des „context exposing“ übernehmen, d.h. Kontextinformation wie z.B den Rückgabewert zur Verfügung stellen.

Die Deklaration von Pointcuts soll an drei Stellen vollzogen werden können:

- Durch Anfügen einer oder mehrerer **Pointcut-Attribute** an die Advice-Methode.
- Durch **Parametertypen** und Rückgabetyp der Advice-Methode.
- In Form von speziellen **Methodenaufrufen** für die Zuweisung von Pointctuts zur Laufzeit.

Pointcut-Attribute sollen Angaben über folgende Eigenschaften machen können:

- **Memberart**: Methode oder Konstruktor.
- **Klassenname:** Name oder Namensmuster der Zielklasse.
- **Membername:** Name oder Namensmuster des Zielmembers.

Bei der Memberart soll vorerst nur zwischen Methode und Konstruktor unterschieden werden. Die Unterstützung für Felder, Properties und Events kann zu einem späteren Zeitpunkt ergänzt werden. Durch Wildcards (z.B. „*“) sollen Klassen- und Membername (Membername nur bei Methoden) optional als Muster formuliert werden können. In der Signatur der Advice-Methode sollen Wildcards verwendet werden können, indem Typen von Basisklassen und variable Argumentlisten (in C# mit dem Schlüsselwort `params`) angegeben werden.

Einen Eindruck von den grundlegenden Fähigkeiten der Benutzerschnittstelle soll Listing 5.1 vermitteln:

```
public class Tracing : Aspect
{
	[Advice(AdviceTypes.Around)]
	[IncludeMethod("*", "SomeMethod")]
	[ExcludeMethod("SomeClass", "*")]
	object AroundAdviceMethod(params object[] args)
	{
		Console.WriteLine(
		  ThisJoinPoint.GetDeclaringType()+ThisJoinPoint.GetMethod());
		Proceed();
		return null;
	}
}
```

Listing 5.1 Konzept der Benutzerschnittstelle von Spider.NET

In Listing 5.1 wird Around-Advice spezifiziert, der auf alle *Methoden* mit dem Namen `SomeMethod` *in allen Klassen* angewendet wird, *außer* in der Klasse `SomeClass`. Parameter und Rückgabetyp passen auf *jede* Methode, da sie variabel gehalten sind. Die Advice-Methode gibt den Namen der Zielklasse sowie der Zielmethode aus. Durch die `Proceed`-Anweisung wird die Abarbeitung aller weiteren Advice-Methoden zugelassen.

5.2.2 Komponentenarchitektur

Folgernd aus den Betrachtungen der letzen Abschnitte wird Spider.NET in folgende Komponenten eingeteilt:

- **User Interface:** Definiert eine Schnittstelle zwischen Benutzer und dem System.
- **Management:** Erstellt und verwaltet die Applikationsbasis mit Informationen über die in der Applikation vorhandenen Aspekte, Advice, Pointcuts und Joinpoints.
- **Interception:** Kapselt die zur Realisierung der .NET-Interception-Infrastruktur benötigte Funktionalität und führt die Aktivierung von Advice durch.
- **Error Treatment:** Definiert Richtlinien zur Fehlerbehandlung, indem sie benutzerdefinierte Ausnahmen einführt.

In Bild 5.2 sind diese Komponenten mit ihren Abhängigkeitsbeziehungen in einem Komponentendiagramm dargestellt:

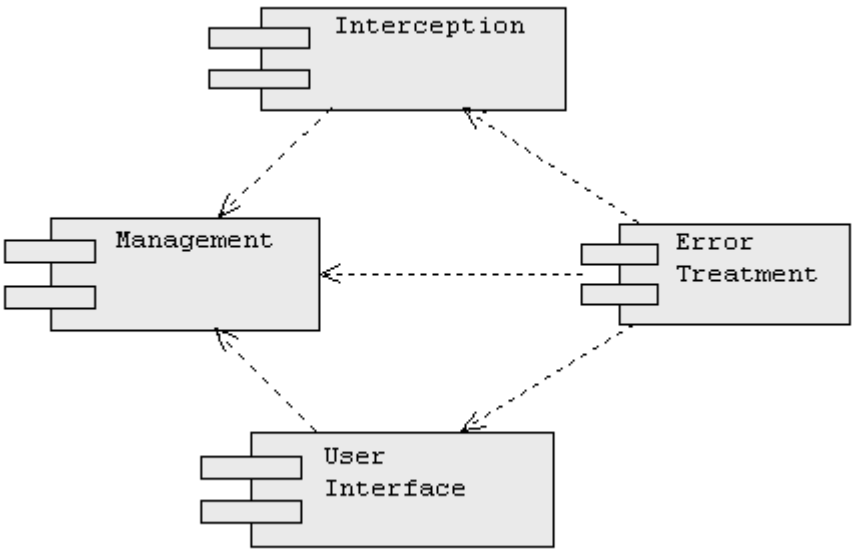

Bild 5.2 Komponentenarchitektur von Spider.NET

Es handelt sich hierbei um logische Komponenten, die eine logische Trennung von Zuständigkeiten realisieren sollen. Die Komponenten werden lediglich in ihren Namensräumen voneinander getrennt und zusammen untrennbar als *eine* DLL ausgeliefert.

5.2.3 Geschäftsvorfälle

Aus grober Sicht spielen zwei Geschäftsvorfälle eine Rolle:

- **Realisierung des Pointcut-Modells:** Besteht aus den Anwendungsfällen „Deklariere Pointcuts“ und „Erstelle Applikationsbasis“.
- **Advice-Aktivierung:** Besteht aus den Anwendungsfällen „Spezifiziere Advice“, „Fange Aufruf ab“, „Finde Joinpoint in Applikationsbasis“ und „Aktiviere Advice“.

Der Geschäftsvorfall „Realisierung des Pointcut-Modells“ wird in der Komponente „Management“ implementiert. Der Geschäftsvorfall „Advice-Aktivierung“ wird in der Komponente „Interception“ implementiert. Die Anwendungsfälle „Deklariere Pointcuts“ und „Spezifiziere Advice“ realisiert der Benutzer. Bild 5.3 zeigt alle Anwendungsfälle:

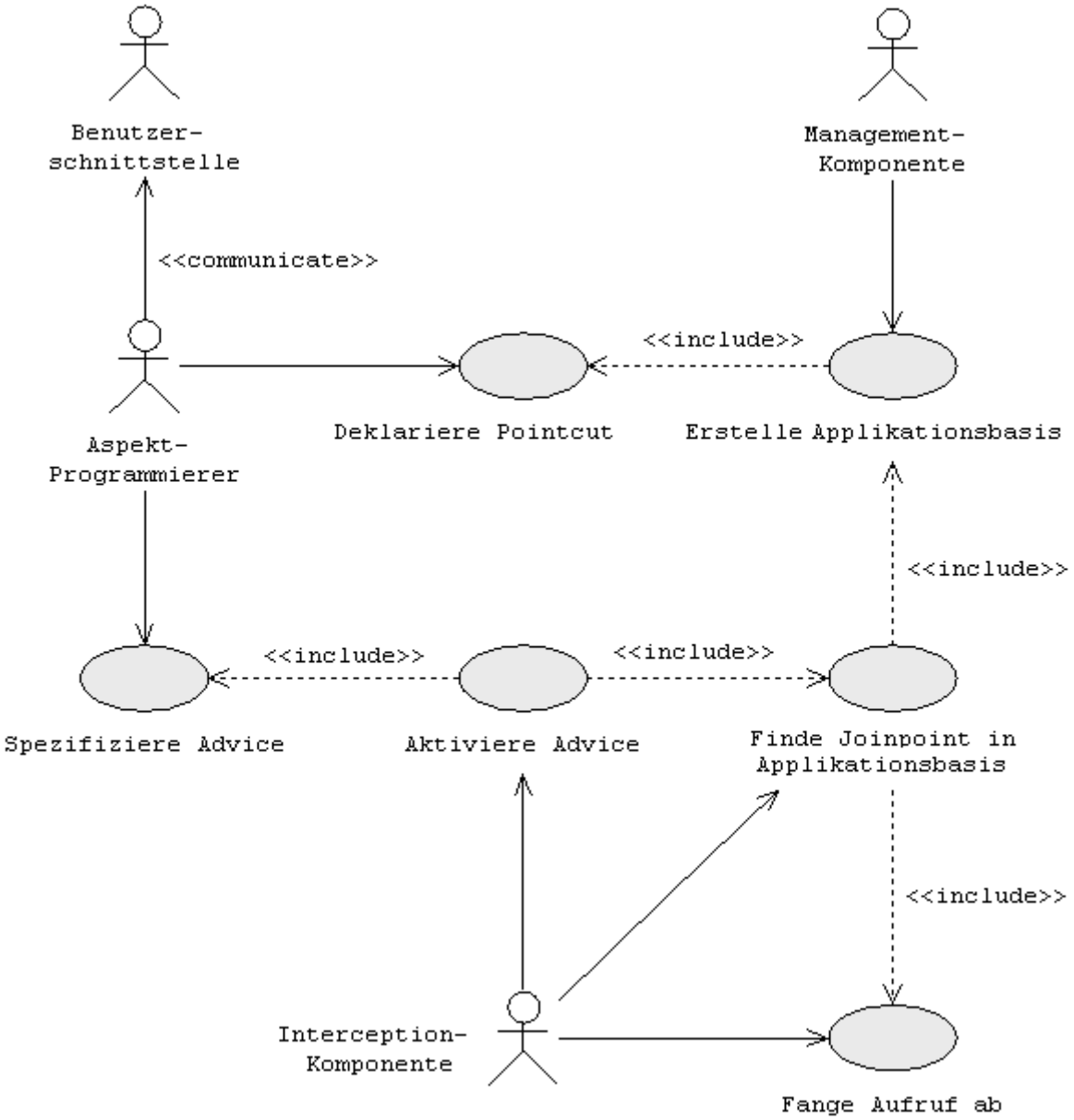

Bild 5.3 Use Cases in Spider.NET

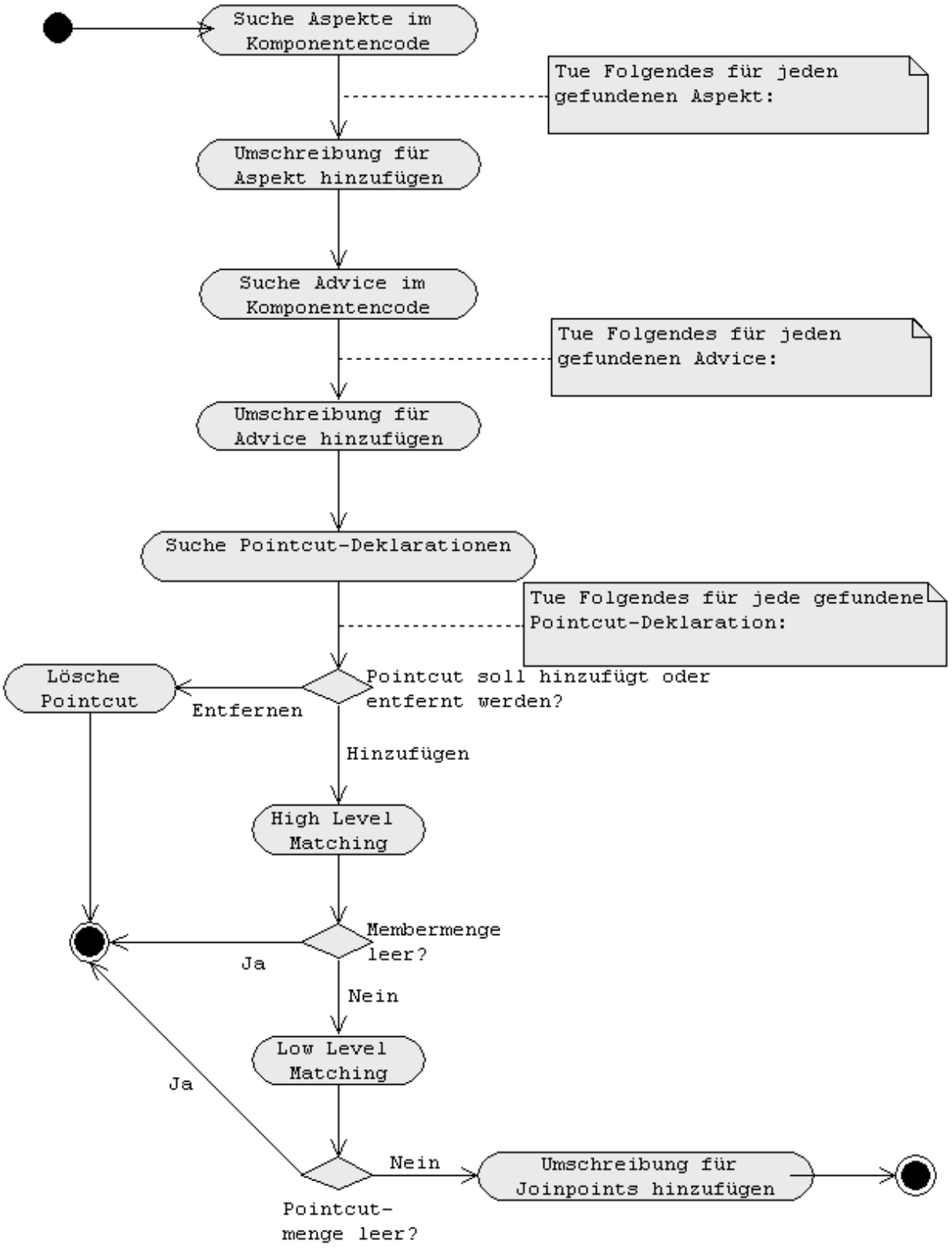

Bild 5.4 Aktivitätsdiagramm zum Geschäftsvorfall „Realisierung des Pointcut-Modells“

Wie in Bild 5.3 auf Seite 71 zu sehen ist, spezifiziert der Aspektprogrammierer Advice und Pointcuts über die Benutzerschnittstelle. Die Management-Komponente wertet die Pointcut-

Deklarationen aus und erstellt die Applikationsbasis. Die Interception-Komponente prüft nach Abfangen eines Aufrufs, ob ein Joinpoint in der Applikationsbasis auf die Zielmethode passt. Ist dies der Fall, so kann sie die zugehörige Advice-Methode aktivieren.

Das Aktivitätsdiagramm von Bild 5.4 auf Seite 72 verdeutlicht die Arbeitsweise des Geschäftsvorfalls *Realisierung des Pointcut-Modells*. Nach Bild 5.4 wird noch *vor dem Eintritt der ersten Interception* anhand der durch den Aspektprogrammierer definierten Aspekte, Advice und Pointcuts die Applikationsbasis aufgebaut. Dort wird für jeden Aspekt, Advice und Pointcut eine *Umschreibung* angelegt, die diese Elemente eindeutig identifiziert. Eine Umschreibung kann sich z.B. aus einem oder mehreren Objekten vom Typ `Type` oder `String` zusammensetzen. Dies soll konkret jedoch erst im Feinentwurf bestimmt werden. Pointcuts können der Applikationsbasis hinzugefügt oder aus ihr entfernt werden. Sollen sie hinzugefügt werden, so werden durch High- und Low-Level-Matching Member im Komponentencode bestimmt, die auf die Beschreibung der Pointcuts zutreffen. Für jeden zutreffenden Member wird eine Umschreibung in Form eines Joinpoint in die Datenbasis eingefügt. Die Datenbasis wird zum Zeitpunkt der Interception durch den Geschäftsvorfall *Advice-Aktivierung* abgefragt.

Der Geschäftsvorfall *Advice-Aktivierung* ist in dem Aktivitätsdiagramm von Bild 5.5 auf Seite 74 dargestellt. Er tritt ein, sobald eine Nachricht an ein Zielobjekt in die Nachrichtensenke des Aspekts gelangt. Die Aktivitäten für die vier Advice-Arten „before“, „around“, „after“ und „after throwing“ sind miteinander verwandt. Für jede Advice-Art wird geprüft, ob Advice für den Aspekt definiert ist. Dann wird anhand des Inhalts der Nachricht ein passender Joinpoint in der Applikationsbasis gesucht. Wird er gefunden, so kann die Advice-Methode mit Reflection aktiviert werden.

Begonnen wird mit Around-Advice, da Around-Advice die Ausführung aller anderen Advice ersetzen kann. Wird Around-Advice regulär beendet, muss der Rückgabewert in eine neue Nachricht verpackt und an die letzte Nachrichtensenke in der Kette zurückgeben werden. Wird Around-Advice dagegen mit einer `proceed`-Anweisung beendet, so wird die Ausführung unterbrochen und übriger Advice kann bearbeitet werden. Nach der Ausführung von Before-Advice wird die Nachricht an die nächste Nachrichtensenke weitergeleitet. Nach der Rückkehr der Nachricht vom Zielobjekt wird zuerst After-Throwing-Advice und danach After-Advice aktiviert. After-Throwing-Advice soll *nur dann* ausgeführt werden, wenn die Zielmethode mit einer Ausnahme beendet wurde. After-Advice wird ausgeführt, wenn die Zielmethode *entweder* regulär *oder* mit einer Ausnahme beendet wurde.

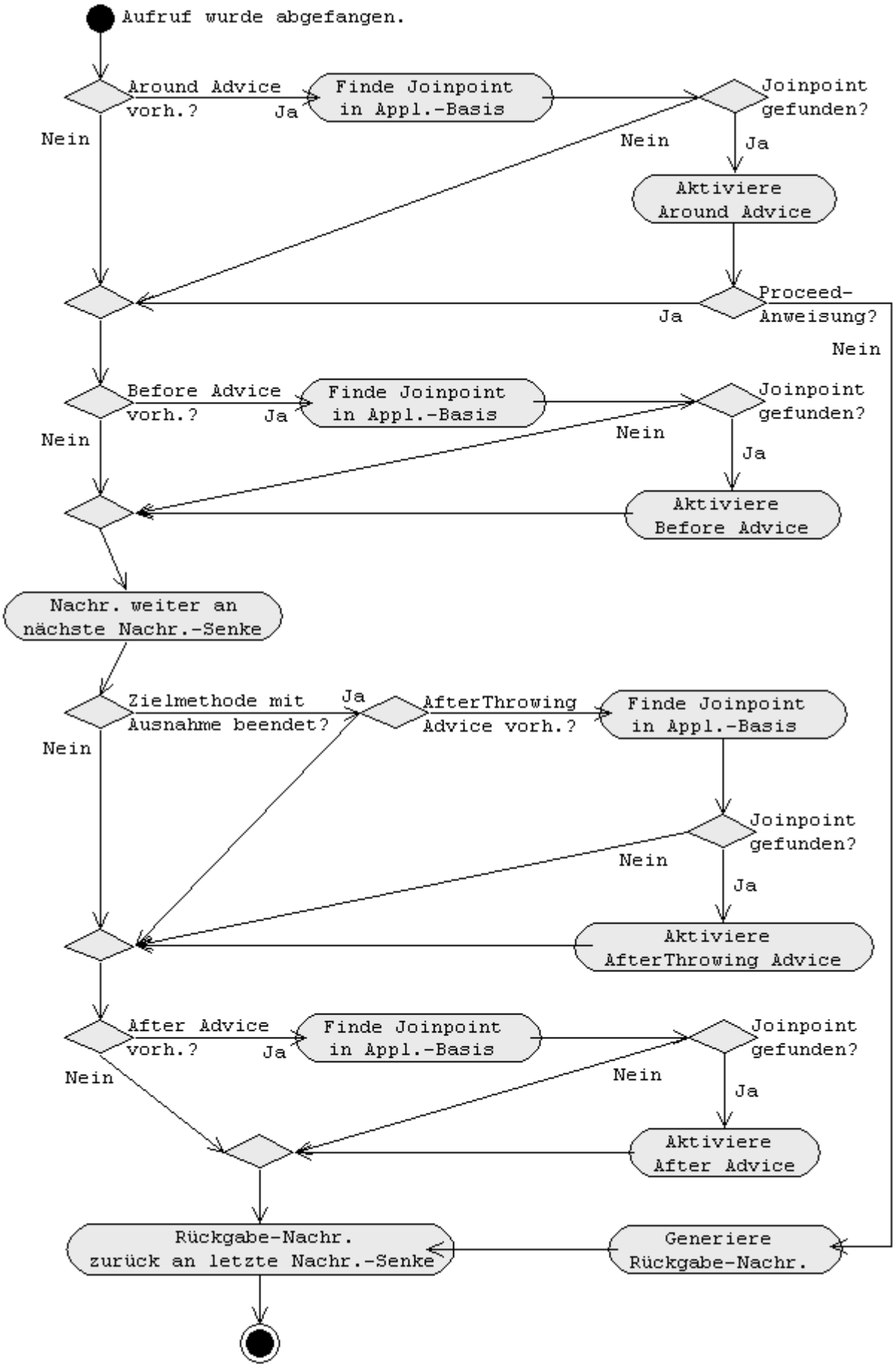

Bild 5.5 Aktivitätsdiagramm zum Geschäftsvorfall „Advice-Aktivierung“

5.3 Feinentwurf

5.3.1 Die Komponente „User Interface“

Das *User Interface* kapselt die Implementierung folgender Funktionalitäten:

- Spezifizierung von Aspekten.
- Spezifizierung von Advice.
- Spezifizierung von Pointcuts.
- Abfrage des Aufrufkontexts in den Advice mit `thisJoinPoint`.
- Abbruch der Around-Advice-Ausführung mit `proceed`.
- Anweisungen für die dynamische Zuweisung von Pointcuts zur Laufzeit.

Wie bereits in Bild 5.1 auf Seite 64 aufgeführt, sind Aspekte von `Aspect` abgeleitete Klassen. Aspect ist wiederum von `ContextAttribute` abgeleitet. Ein Objekt vom Basistyp `Aspect` muss durch einen eindeutigen Namen identifizierbar sein. Bild 5.6 zeigt das Klassendiagramm. Die Methoden `IsContextOK` und `GetPropertiesForNewContext` sind von .NET vorgeschrieben, um die Funktionalität des Kontextattributs zu realisieren.

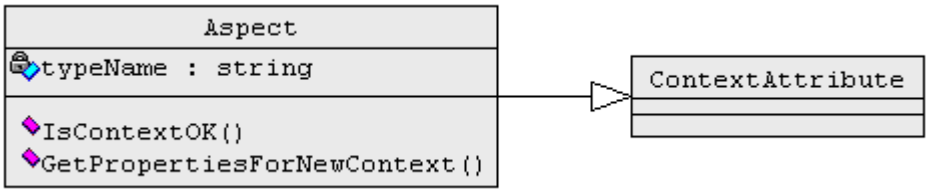

Bild 5.6 Klassendiagramm für die Spezifizierung von Aspekten

Aspekte können in zwei Modi betrieben werden. Im „Singleton“-Modus existiert nur eine Instanz des Aspekts in der Applikation. Im „Multiple“-Modus hingegen wird bei jeder Interception die Aspektinstanz neu angelegt. Die Unterscheidung zwischen Singleton- und „Multiple“-Modus ist aus AspectJ bekannt. Den Modus kann der Benutzer mit dem Attribut `AspectModeAttribute` angeben, welches das Klassendiagramm in Bild 5.7 zeigt:

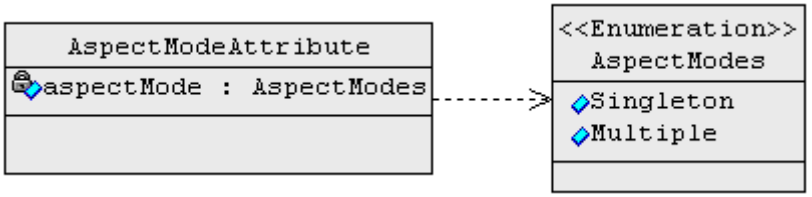

Bild 5.7 Klassendiagramm für die Angabe des Aspekt-Modus

Advice sind Methoden, die mit einem Attribut vom Typ `AdviceAttribute` gekennzeichnet sind. Als Argument für den Konstruktor von `AdviceAttribute` wird der Typ des Advice angegeben. Der Advice-Typ wird durch die Enumeration `AdviceTypes` angegeben. Bild 5.8 zeigt das Klassendiagramm für die Spezifizierung von Advice:

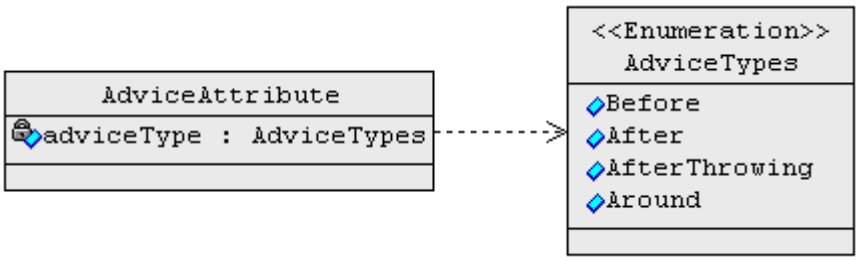

Bild 5.8 Klassendiagramm für die Spezifizierung von Advice

Pointcuts werden mit Attributen spezifiziert, die von `PointcutAttribute` abgeleitet sind. Je nach Art des Zielmembers (Konstruktor, Methode usw.) werden verschiedene Informationen benötigt. Es liegt nahe, für jeden Membertyp ein separates Attribut zu modellieren. Die Benutzung von benannten Parametern[1] ist hier nicht angebracht, da die Benutzerschnittstelle so nicht mehr klar und frei von Redundanz wäre. Die verschiedenen Arten von Membertypen werden in der internen Enumeration `TargetMemberTypes` festgehalten. Bild 5.9 zeigt die aus diesen Anforderungen resultierende Klassenhierarchie:

[1] Benannte Attribut-Parameter in .NET sind Parameter, die optional an die im Attribut-Konstruktor explizit vorgegebenen Parameter angefügt werden können. Dadurch muss der Programmierer nicht für jede Kombination von Parametern einen eigenen Konstruktor definieren [Arch02].

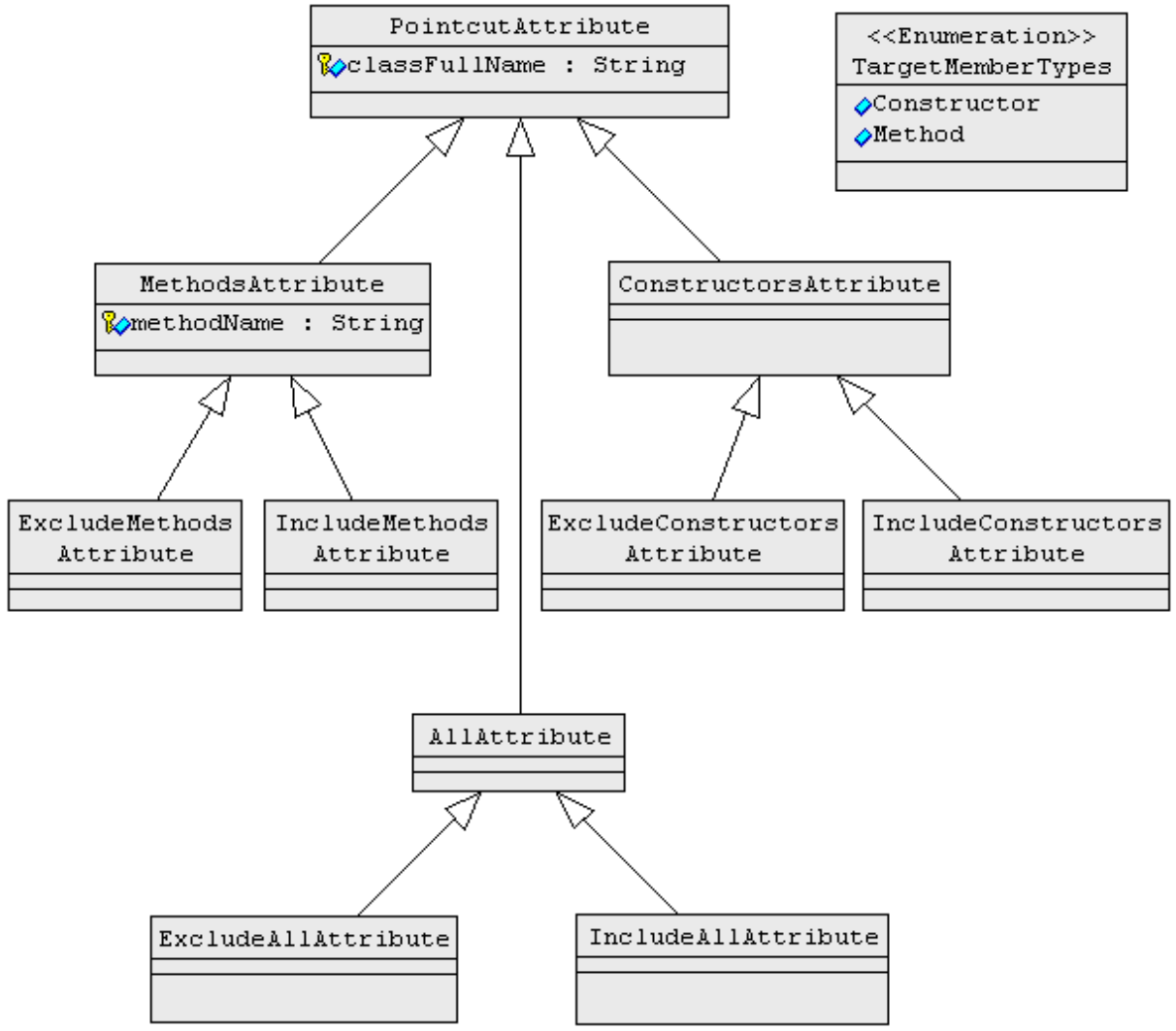

Bild 5.9 Klassendiagramm für die Spezifizierung von Pointcuts

Zwischen den Funktionalitäten von `thisJoinPoint` und `proceed` lassen sich Analogien finden: Beide sind Dienste, die aus dem Advice-Code heraus in Anspruch genommen werden, und beide müssen mit der Interception-Komponente kommunizieren können, obwohl diese zum Zeitpunkt der Ausführung des Advice-Codes nicht erreichbar ist. Die Lösung besteht darin, Informationen, die für solche Advice-Dienste gebraucht werden, vorübergehend in der `Aspect`-Instanz zu speichern. Advice-Dienste sind Klassen, die das Interface `IAdviceService` implementieren. Die Klasse `Aspect` muss somit neu entworfen werden. Sie bekommt Verweise auf Advice-Dienste und implementiert öffentliche Methoden, welche die Benutzung der Advice-Dienste einfach machen. Bild 5.10 zeigt das Klassendiagramm:

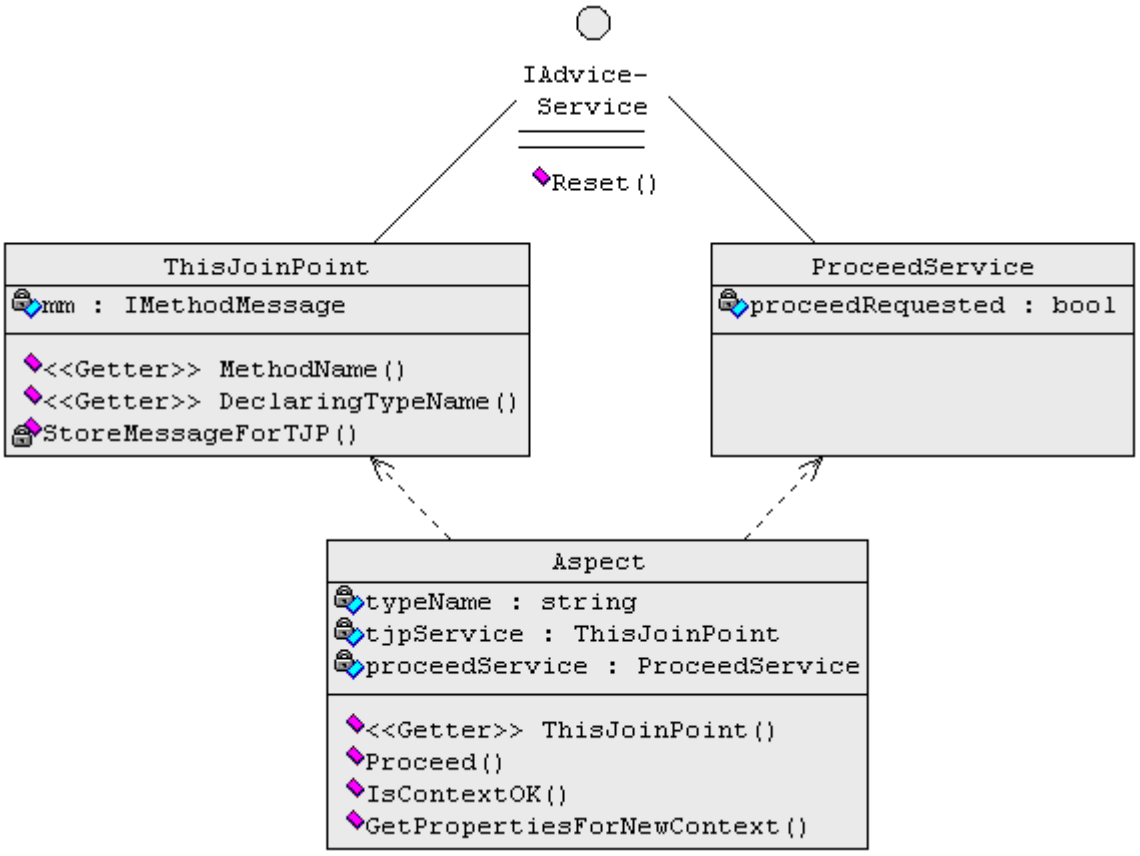

Bild 5.10 Klassendiagramm für die Realisierung der Advice-Dienste `ThisJoinPoint` und `Proceed`

Die Advice-Service-Klassen `ThisJoinPoint` und `ProceedService` dienen als Vermittler zwischen Interception-Komponente und Advice-Code. Sie sind Member der Klasse `Aspect`. Die Interception-Komponente speichert die empfangene Nachricht in einem Feld der `ThisJoinPoint`-Instanz, damit der Advice-Code die in der Nachricht enthaltenen Informationen abrufen kann. Umgekehrt signalisiert der Advice-Code in einer `ProceedService`-Instanz das Auftreten einer Proceed-Anfrage, damit diese von der Interception-Komponente bearbeitet werden kann. Mit der Methode `Reset` kann die Information für Advice-Dienste nach Gebrauch wieder gelöscht werden.

Für die Zuweisung von Pointcuts zur Laufzeit wird die Klasse `Weaver` verwendet. Obwohl er aus technischer Sicht nicht ganz zutrifft[1], ist dieser Name für die Benutzerseite gut geeignet, da er für den Haupteinstiegspunkt ins Weaver-Tool Spider.NET steht (vgl. Klasse `Weaver` in Rapier-Loom.Net). Die Befehle für die dynamische Pointcut-Zuweisung sollen die gleiche Wirkung haben wie Pointcut-Deklarationen mit Hilfe von Attributen und werden demnach ähnlich strukturiert (vgl. Bild 5.9 auf Seite 77). Ferner soll der Benutzer mit der

[1] Die Verwebung bei Spider.NET wird nicht von der Klasse `Weaver`, sondern von der CLR eingeleitet.

Klasse `Weaver` Spider.NET über das Vorhandensein mehrerer Assemblies informieren können (vgl. Abschnitt 5.2.1). Dazu bekommt `Weaver` eine Methode zur Registrierung von Assemblies mit Aspekten sowie eine Methode zur Registrierung von Assemblies mit Zielklassen. Des Weiteren soll `Weaver` auch der Einstiegspunkt für das Erstellen der Applikationsbasis sein. Die Erstellung der Applikationsbasis muss noch vor Eintritt der ersten Interception erfolgt sein. Um größte Transparenz zu wahren und den Aspektprogrammierer nicht mit dem expliziten Auslösen dieser Aktion zu belasten, sollte sie zum Zeitpunkt der ersten Instanzierung eines `Aspect`-Objekts eingeleitet werden. Dazu soll die interne statische Methode `Weaver.Weave` aufgerufen werden. Somit ergibt sich für die Klasse `Weaver` das in Bild 5.11 dargestellte Klassendiagramm:

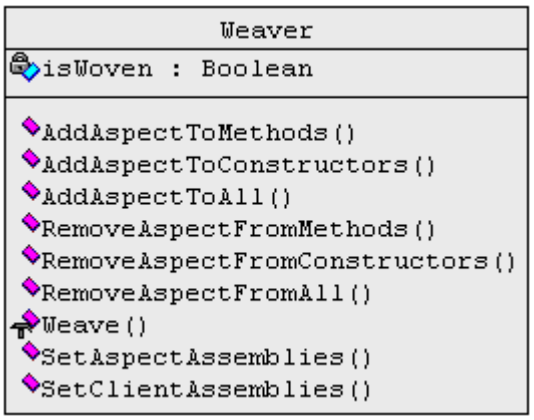

Bild 5.11 Klassendiagramm für die Klasse `Weaver`

5.3.2 Die Komponente „Management“

Die Management-Komponente spielt eine ähnliche Rolle wie das Back-End des AspectJ-Compilers (vgl. Abschnitt 4.1.8), da sie das Matching durchführt und die Verwaltung der Applikationsbasis regelt. Zum Zeitpunkt der Interception muss die Applikationsbasis darüber Auskunft geben, ob ein passender Joinpoint für den abgefangenen Aufruf vorhanden ist. Da die Interception eine performanzlastige Operation ist, muss diese Information so schnell wie möglich zur Verfügung stehen. Ein grundlegendes Entwurfsziel der Management-Komponente ist es somit, die Applikationsbasis für den schnellen Zugriff durch die Interception-Komponente zu optimieren. Dies erfordert den Entwurf einer geeigneten Datenstruktur sowie geeigneter Umschreibungen für Aspekte, Advice und Joinpoints (vgl. Bild 5.4 auf Seite 72). Die schnellsten Ergebnisse können erzielt werden, wenn eine Baumstruktur verwendet wird, wie sie in Bild 5.12 dargestellt ist:

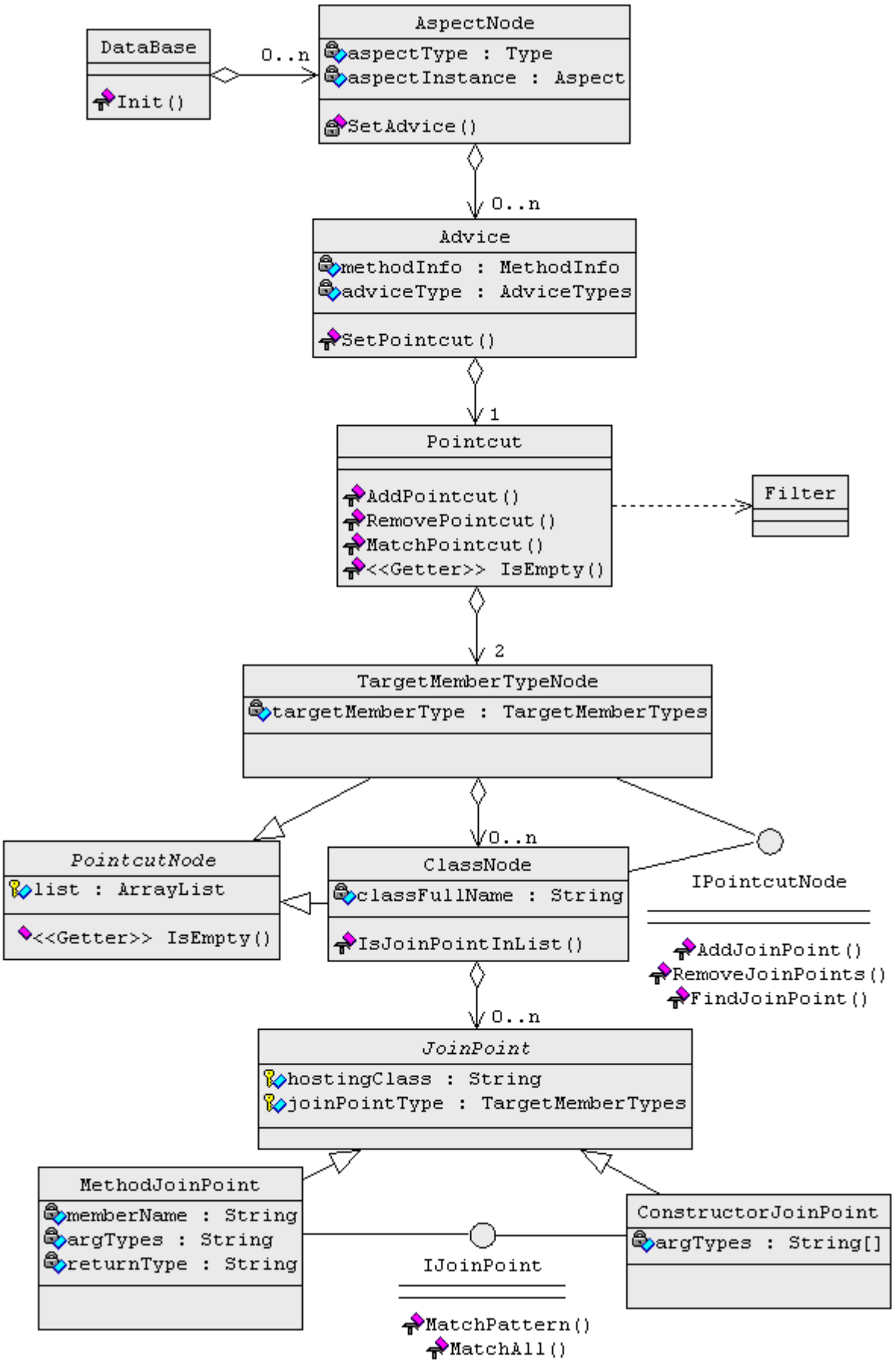

Bild 5.12 Struktur der Applikationsbasis

Die Klasse `DataBase` ist die Wurzel des Applikationsbasis-Baums. Sie enthält eine Methode `Init`, mit der das Aufbauen der Applikationsbasis eingeleitet wird. Im „Multiple“-Modus werden Aspekte in der Klasse `AspectNode` durch einen Member vom Typ `Type` repräsentiert, damit sie später mit geringstem Aufwand instanziert werden können. Im „Singleton“-Modus wird die Aspektinstanz schon beim Aufbau der Applikationsbasis angelegt und im Member `aspectInstance` gespeichert. Advice wird durch den Advice-Typ sowie durch ein `MethodInfo`-Objekt repräsentiert, damit auch hier schnellstmögliche Aktivierung vollzogen werden kann. Die Methode SetPointcut sucht Pointcut-Deklarationen im Aspektcode und leitet ihre Bearbeitung in der Klasse `Pointcut` ein. Diese enthält eine Methode zum Hinzufügen (`AddPointcut`) sowie eine Methode zum Herausnehmen (`RemovePointcut`) eines Pointcut. `AddPointcut` verwendet Methoden der Klasse `Filter`, um durch High-Level- und Low-Level-Matching die endgültige Pointcutmenge zu bestimmen.

Die Pointcutmenge wird durch eine Menge von Objekten des Basistyps `JoinPoint` dargestellt. Da es mehrere Arten von Joinpoints gibt (`ConstructorJoinPoint` und `MethodJoinPoint`), wird für ihre Erstellung eine abstrakte Fabrik verwendet. Joinpoints werden in den Blättern des Applikationsbasis-Baums gespeichert. Damit sie schneller gefunden werden, werden sie anhand der Kategorien Membertyp und Hostklasse in verschiedenen Zweigen gespeichert. Die Baumknoten, welche die Information zu den Kategorien enthalten, sind Objekte vom Typ `PointcutNode` und implementieren die Schnittstelle `IPointcutNode`. Joinpoints können anhand der Methoden `Pointcut.MatchPointcut` und `IPointcutNode.FindJoinPoint` gefunden werden. Die Getter-Properties `IsEmpty` dienen der Beschleunigung der Anfragen, falls bestimmte Zweige des Baums leer sind.

Das Sequenzdiagramm von Bild 5.13 zeigt den Entwurf der Durchführung des High- und Low-Level-Matching:

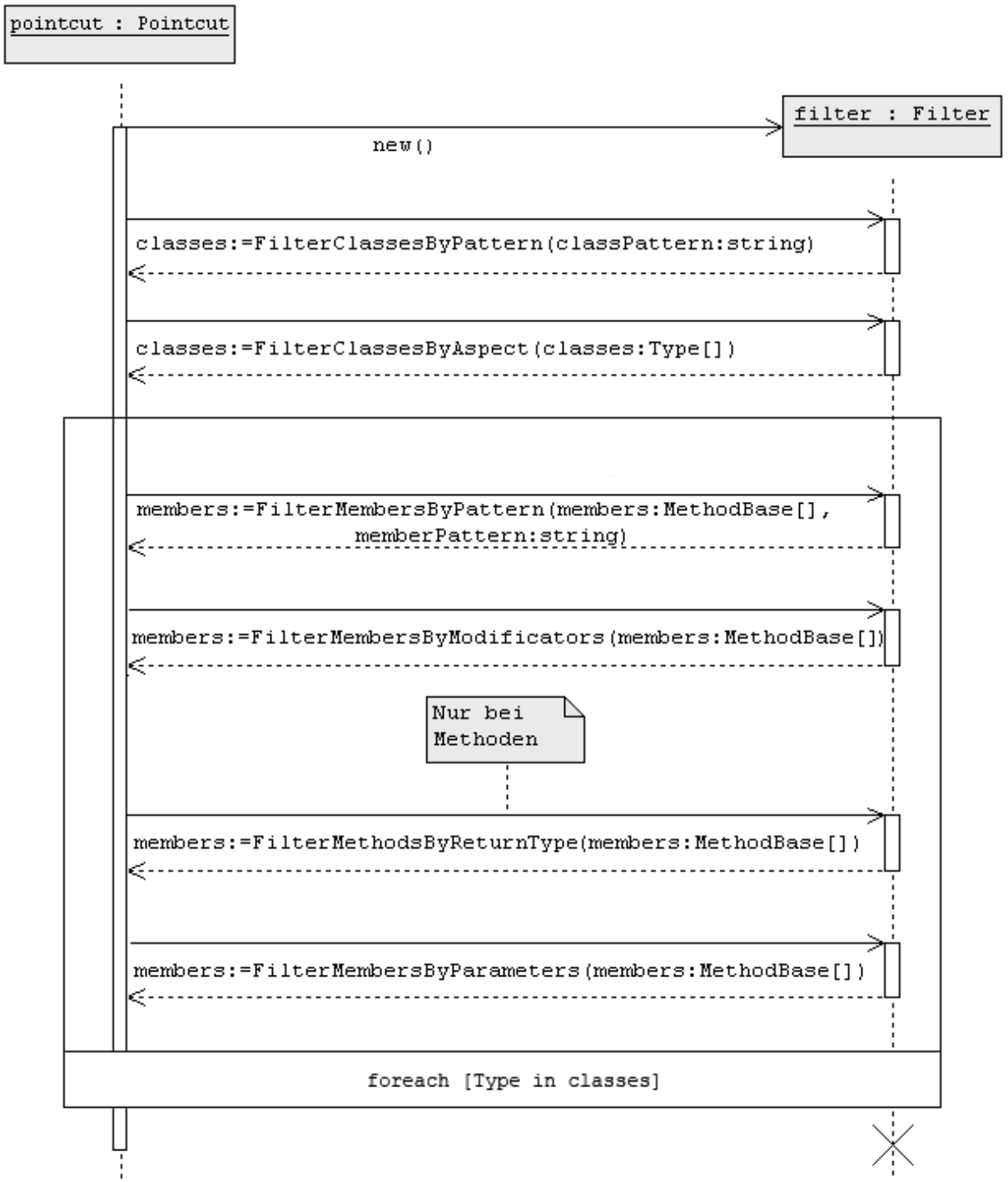

Bild 5.13 Sequenzdiagramm für das Matching

Wie oben erwähnt, benutzt die Methode `Pointcut.AddPointcut` zur Bestimmung der Pointcutmenge verschiedene Methoden der Klasse `Filter`. Die Methoden der Klasse `Filter` arbeiten alle nach demselben Prinzip: Sie nehmen eine Menge von Elementen entgegen und nehmen solche heraus, die bestimmten Kriterien nicht entsprechen. Die Ausgabe ist wiederum eine Menge von Elementen, die kleiner oder gleich der Eingangsmenge sein kann. In Bild 5.13 ist beschrieben, wie ein Array von potenziellen Zielklassen gefiltert wird. Von den übrig gebliebenen Klassen werden Member gefiltert. Die

Methoden `FilterMethodsByReturnValue` und `FilterMembersByParameters` führen das Low-Level-Matching durch.

5.3.3 Die Komponente „Interception“

Die Klassenstruktur der Interception-Komponente ist weitgehend durch die Remoting-Infrastruktur von .NET vorbestimmt. Es wird eine Klasse benötigt, welche die Schnittstelle `IContextProperty` implementiert. Diese Klasse muss eine Nachrichtensenke instanzieren, und zwar im Fall von Spider.NET eine serverseitige (vgl. Abschnitt 5.1.3). Hierfür muss die Schnittstelle `IContributeServerContextSink` implementiert werden. Des Weiteren wird eine Klasse benötigt, die eine Nachrichtensenke realisiert, indem sie die Schnittstelle `IMessageSink` implementiert. Bild 5.14 zeigt dazu das Klassendiagramm:

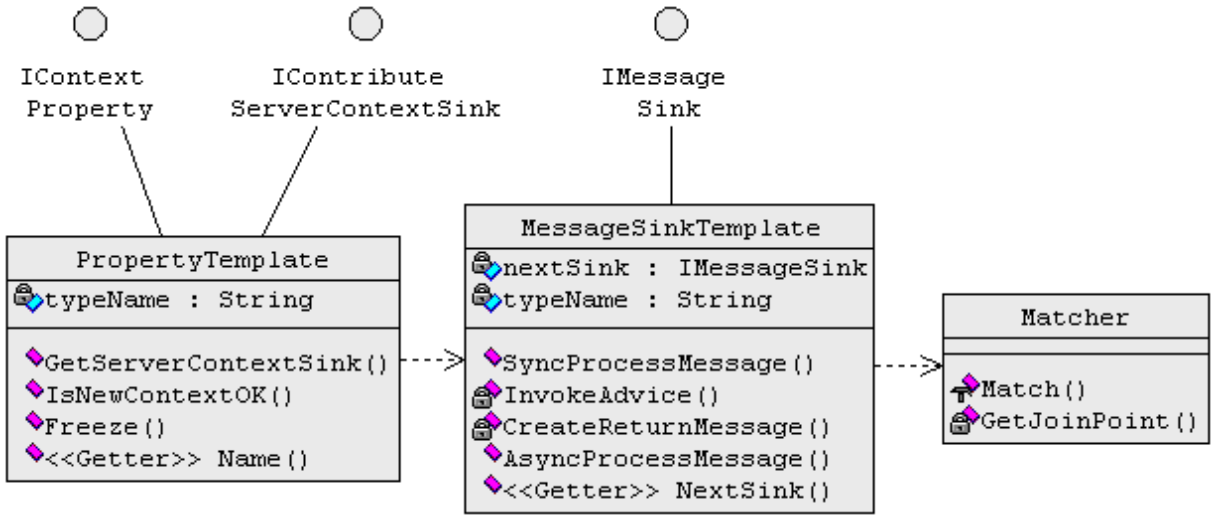

Bild 5.14 Klassendiagramm für die Komponente „Interception“

Jeder Aspekt bekommt für jedes Objekt, für welches das Kontextattribut des Aspekts angegeben wurde, eine Instanz der Klasse `PropertyTemplate` sowie eine Instanz der Klasse `MessageSinkTemplate` zugewiesen. Der zugehörige Aspekt wird durch das Feld `typeName` identifiziert. Nach dem Abfangen eines Aufrufs durch die CLR gelangt der Kontrollfluss in die Methode `MessageSinkTemplate.SyncProcessMessage`, wo die Aktivierung der Advice eingeleitet werden kann (vgl. Listing 3.3 auf Seite 41). Das Sequenzdiagramm in Bild 5.15 zeigt den Entwurf des Vorgangs der Advice-Aktivierung, und zwar beispielhaft am Fall Aspekt-Modus „Singleton“ und Before-Advice. Vorausgesetzt wird, dass Aspekt und Advice für den gegebenen Ziel-Joinpoint existieren und keinerlei ausnahmen auftreten:

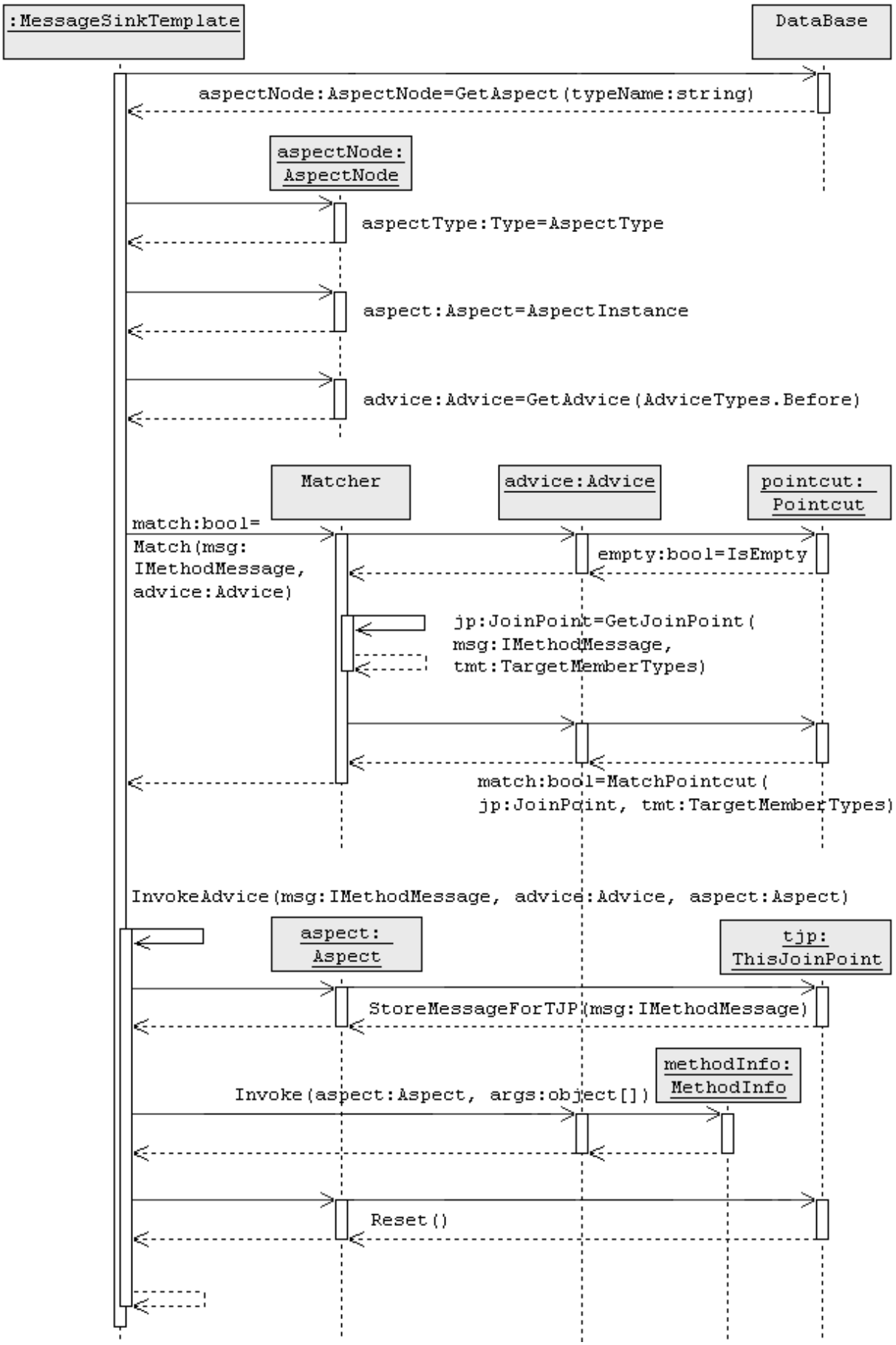

Bild 5.15 Sequenzdiagramm für die Advice-Aktivierung

Wie Bild 5.15 demonstriert, beschafft sich die Methode `SyncProcessMessage` als erstes eine Referenz auf die zur Nachrichtensenke gehörende `AspectNode`-Instanz. Diese enthält die Aspekt-Instanz sowie alle auf ihr definierten Advice-Methoden. Die Methode `Matcher.Match` leitet die Anfrage nach einem zum Zielmember passenden Joinpoint in der Applikationsbasis ein. Dazu erstellt die Methode `Matcher.GetJoinPoint` aus Parametern der Nachricht eine Instanz des Typs `JoinPoint`, um sie anschließend durch den Applikationsbasis-Baum zu führen und dort mit all den anderen Instanzen von `JoinPoint` zu vergleichen. `Pointcut.MatchPointcut` leitet diese Suche ein. Im Erfolgsfall der Suche aktiviert `MessageSinkTemplate.InvokeAdvice` die Advice-Methode. Bevor jedoch `MethodInfo.Invoke` aufgerufen werden kann, muss der `ThisJoinPoint`-Service eine Referenz auf die Nachricht bekommen (vgl. Bild 5.10 auf Seite 78). Dementsprechend muss die Referenz nach der Rückkehr der Advice-Methode wieder gelöscht werden.

Die in dem Klassendiagramm von Bild 5.14 auf Seite 83 abgebildete Methode `CreateReturnMessage` erstellt Rückmeldungen, um Rückgabewerte von Around-Advice an Aufrufobjekte leiten zu können. Die Methoden `AsyncProcessMessage` und `NextSink` sind von .NET vorgegebene Schnittstellenimplementierungen.

5.3.4 Die Komponente „Error Treatment“

Zur Fehlerbehandlung werden benutzerdefinierte Ausnahmeklassen benötigt:

- **InternalException:** Interne Ausnahme, die bei falscher Benutzung von internen Methoden ausgelöst wird. Soll nur in der Entwicklung verwendet werden.
- **SpecViolationException:** Öffentliche Ausnahme, die ausgelöst wird, wenn der Aspektprogrammierer bestimmte Vereinbarungen nicht einhält.
- **AspectNotFoundException:** Wird ausgelöst, wenn ein Aspekt anhand seines Namens nicht gefunden wird. Ist nur für das frühe Entwicklungsstadium gedacht.

Bild 5.16 zeigt das Klassendiagramm für die Komponente „Error Treatment“:

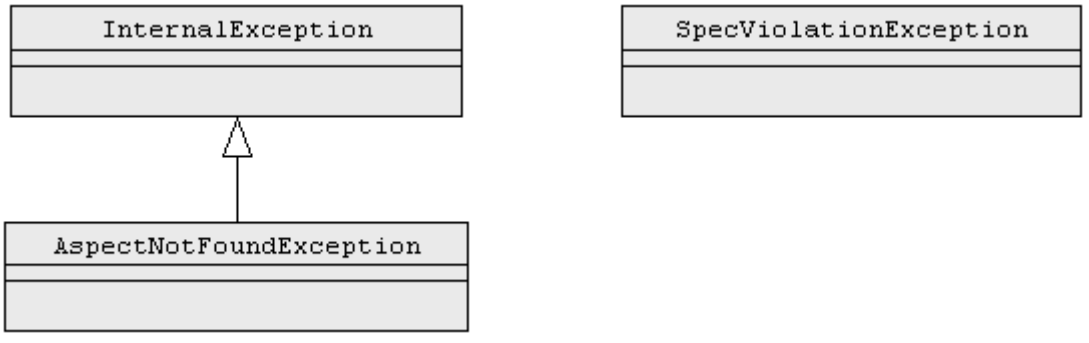

Bild 5.16 Klassendiagramm für die Komponente „Error Treatment“

5.4 Implementierung

5.4.1 High-Level-Matching und Unterstützung größerer Projekte

Durch **H**igh-**L**evel-**M**atching (HLM) wird aus der in einem Pointcut-Attribut angegebenen Information (d.h. Zeichenkette für Klassen- und Methodenname oder Muster) eine Menge von Ziel-Membern (d.h. Objekten vom Typ `MethodBase`) herausgesucht. Dazu muss das Pointcut-Attribut mit allen Klassen im gesamten Applikationscode verglichen werden. Durch Filterung werden Klassen und Member eliminiert, die mit den Angaben im Pointcut-Attribut nicht übereinstimmen. Die Methode `Pointcut.AddPointcut` enthält die Geschäftslogik für das HLM und benutzt Methoden der Klasse `Filter` zur Filterung (vgl. Bild 5.13 auf Seite 82).

Ein Kernpunkt bei der Implementierung des HLM ist, dass Zielklassen einer Applikation unter Umständen auf mehrere Assemblies verteilt sein können. Gleiches gilt für Aspekte. Um Unterstützung auch für größere Projekte sowie den Einsatz wiederverwendbarer Aspektkomponenten zu gewährleisten, muss das HLM alle Assemblies einer Applikation nach potenziellen Zielklassen durchsuchen können. Dazu werden zwei Methoden implementiert, mit denen der Benutzer externe Assemblies angeben kann (vgl. Abschnitt 5.3.1):

- `Weaver.SetClientAssemblies(params Assembly[] clientAssemblies)`
- `Weaver.SetAspectAssemblies(params Assembly[] aspectAssemblies)`

Eine Alternative wäre eine XML-Konfigurationsdatei, wie sie z.B. in AspectC# oder LOOM.NET verwendet wird. Die Entscheidung fiel jedoch auf die Lösung mit den `Weaver`-Methoden, da sie den Einsatz der Reflection-API zur Bestimmung von Assemblies ermöglicht.

Falls die Angaben im Pointcut-Attribut Muster enthalten, führt das HLM die Analyse dieser Muster durch. Vorerst wurde nur die Berücksichtigung des Wildcard-Symbols „*“ implementiert, das für alle Klassennamen bzw. alle Methodennamen steht. Die Implementierung der Analyse komplexerer Ausdrücke wie in AspectJ ist jedoch möglich.

Listing 5.2 beschreibt die Arbeitsweise der HLM-Methoden anhand der Methode `Filter.FilterClassesByAspect`. Alle in Bild 5.13 auf Seite 82 aufgeführten Filter-Methoden arbeiten nach diesem Muster:

```
internal Type[] FilterClassesByAspect(Type[] classes)
{
	ArrayList al = new ArrayList();

	// Filter classes that don't belong to the aspect.
	foreach(Type t in classes)
	{
		object[] attribs = t.GetCustomAttributes(
			advice.MethodInfo.DeclaringType, true);
		if(attribs.Length>0)
			al.Add(t);
	}
	return al.ToArray(typeof(Type)) as Type[];
}
```

Listing 5.2 High-Level-Matching

Wie Listing 5.2 demonstriert, nehmen Methoden der Klasse `Filter` all die Elemente, die auf bestimmte Kriterien zutreffen, in einer Liste auf und geben diese Liste als Array zurück.

5.4.2 Low-Level-Matching und variable Argumente

Das **L**ow-**L**evel-**M**atching (LLM) filtert anhand des Methodenkopfes der Advice-Methode eine Menge von `MethodBase`-Instanzen zu einer Menge von Instanzen der Typs `JoinPoint`. Dazu werden drei Eigenschaften zum Vergleich herangezogen:

- Rückgabetyp
- Anzahl der Parameter
- Parametertypen

Ferner hat das LLM zum Ziel, ein hohes Maß an Quantifikation zu gewährleisten (vgl. Abschnitt 3.1.2). Zum einen sollen durch Angabe eines Basistyps alle Typen akzeptiert werden, die an den Basistyp zugewiesen werden können. Dieses Verhalten wird von .NET durch die Methode `Type.IsAssignableFrom(Type)` erbracht. Zum anderen soll Variabilität in der Parameteranzahl unterstützt werden, indem das Schlüsselwort `params` verwendet wird. Der LLM-Algorithmus muss dazu mit Reflection ermitteln können, ob eine Parameterliste variable Argumente hat. Dabei spielen folgende Eigenschaften eine Rolle:

- Variable Argumente müssen die letzen Argumente der Parameterliste sein.
- Variable Argumente werden zur Laufzeit in einem gewöhnlichen Array übergeben.
- Variable Parameter-Arrays werden mit dem Parameterattribut `ParamArrayAttribute` identifiziert.

Mit dieser Information kann eine Methode implementiert werden, die das Vorhandensein variabler Argumente ermittelt. Listing 5.3 zeigt diese Methode:

```
internal static bool HasVarArgs(ParameterInfo[] pi)
{
	if(pi.Length==0)
		return false;
	ParameterInfo last = pi[pi.Length-1];
	object[] paAttr = last.GetCustomAttributes(
		typeof(System.ParamArrayAttribute), false);
	if(paAttr.Length>0)
		return true;
	return false;
}
```

Listing 5.3 Methode, die ermittelt, ob eine Parameterliste über variable Argumente verfügt

Ferner muss der LLM-Algorithmus beim Vergleich der Methodenköpfe drei mögliche Fälle unterscheiden:

- Weder Ziel-Member noch Advice-Methode haben variable Argumente.
- Nur die Advice-Methode hat variable Argumente, um mehrere Ziel-Member anzusprechen.
- Der Ziel-Member hat variable Argumente. In solchen Fällen muss auch die Advice-Methode variable Argumente haben.

Hierbei ergibt sich eine Vielzahl von Kombinationen, die der LLM-Algorithmus berücksichtigt. Tabelle 5.2 zeigt einen repräsentativen Auszug von Advice- und Zielmethoden, die aufeinander passen:

Parameterliste der Advice-Methode	**Parameterliste der Zielmethode**
`Advice(params int[])`	`Ziel(int)`
`Advice(params object[])`	`Ziel(int)`
`Advice(params int[])`	`Ziel()`
`Advice(params object[])`	`Ziel(params int[])`
`Advice(params object[])`	`Ziel(string, params Type[])`

Tabelle 5.2 Low-Level-Matching mit variablen Argumenten

.NET bietet für solche Probleme keine Unterstützung, obwohl .NET-Compiler ähnliche Prüfungen durchführen müssen. Beim Vergleich von Array-Typen (z.B. `int[]` und `object[]`) lässt sich `Type.IsAssignableFrom(Type)` nicht mehr anwenden, da der C#-Compiler Arrays von Untertypen Arrays von Basistypen nicht zuweisen kann. Daher wird eine Methode benötigt, die den Typ der Arrayelemente aus dem Array-Typ bestimmt (z.B. `int` aus `int[]`). Der Autor hat hierzu keinen anderen Weg gefunden als die Benutzung von Zeichenkettenoperationen, wie Listing 5.4 zeigt:

```
internal static Type ArrayType2ElementType(Type arrayType)
{
       // Remove brackets:
       string arrayTypeStr = arrayType.FullName;
       string elemTypeStr  = arrayTypeStr.Remove(
                                        arrayTypeStr.Length-2, 2);
       return Type.GetType(elemTypeStr);
}
```

Listing 5.4 Bestimmung des Elementtyps aus dem Array-Typ

Auch im Zusammenhang mit der Aktivierung von Advice-Methoden mit `MethodInfo.Invoke` stellen variable Argumente die Hauptschwierigkeit dar. Methodenargumente liegen in abgefangenen Nachrichten in serieller Form vor. Die `MethodInfo.Invoke`-Methode hingegen nimmt Argumente nur in Form eines object[]-Arrays entgegen. Daher muss man die seriell angeordneten variablen Argumente extrahieren, in ein mit Reflection erstelltes Array konvertieren und dieses an die letzte Position der Argumentliste der `Invoke`-Methode übergeben. Insofern wird das Verhalten der CLR bei stapelbasierten Methodenaufrufen mit variablen Argumenten emuliert. Dabei ist eine ähnliche Kombination von Fällen zu berücksichtigen wie beim LLM-Algorithmus (s.o.).

Aufgrund der Vielzahl der möglichen Kombinationen sind LLM und die Formatierung von Advice-Argumenten umfangreiche Operationen, die in diesem Dokument nicht wiedergegeben werden können. Hierzu sei auf die Methoden `Filter.FilterMembersByParameters` und `MessageSinkTemplate.InvokeAdvice` verwiesen.

5.4.3 Nachritenverarbeitung und Advice-Kette

Bild 5.5 auf Seite 74 zeigte bereits die Reihenfolge der Ausführung verschiedener Advice-Arten. Bisher wurde jedoch nicht über den *Einfluss* von Advice auf das Verhalten der Applikation gesprochen. Beispielsweise soll folgendes Szenario möglich sein: In einem Aspekt sind Around-, Before und After-Advice definiert, von denen alle die Methode `int SomeMethod(int i)` zum Ziel haben. `SomeMethod` gibt lediglich das ihr übergebene Argument zurück. `SomeMethod` wird mit dem Integer-Wert 0 aufgerufen. Tabelle 5.3 zeigt die Parameter- und Rückgabewerte an bestimmten Stellen der Ausführung unter Durchführung bestimmter Aktionen:

Stellen im Programmfluss	Wert des Arguments (int i) vor Ausf. d. Aktionen	Wert der Rückgabe vor Ausf. d. Aktionen	Aktionen
Aufruf an Aufrufstelle	0	Nicht vorh.	

Around-Advice	0	`null`	`++i;`
			`Proceed();`
			`return 2;`

Before-Advice	1	2	`i = Rückgabewert;` `return 0;`
Zielmethode	2	Nicht vorh.	`return i;`
After-Advice	2	2	`return Rückgabewert + i;`
Rückkehr zur Aufrufstelle	Nicht vorh.	4	

Tabelle 5.3 Manipulation der Ergebnisse von Zielmethoden durch Verkettung von Aktionen

Um die in Tabelle 5.3 demonstrierte Logik zu realisieren, müssen beim Abfangen einer Nachricht alle Advice-Methoden eines Aspekts zu einer Verarbeitungskette verbunden sein. Dazu müssen Advice-Methoden *Parameter- und Rückgabewerte* ermitteln, modifizieren und nach außen weitergeben können. Dazu werden in der `ThisJoinPoint`-Klasse folgende Properties definiert:

- `ReturnValue`: Liefert Rückgabewert des vorherigen Glieds der Verarbeitungskette.
- `Exception`: Liefert Ausnahme-Objekt für After-Throwing-Advice.

Die Modifizierung von Parametern ist realisierbar, indem der Benutzer Parameter als Referenzparameter (C#-Schlüsselwörter `ref` oder `out`) deklariert. Als Voraussetzung müssen Referenzparameter vom LLM unterstützt werden. Die Methode von Listing 5.5 prüft dazu die Zuweisung von Referenzparametern an normale Parameter, indem sie mit Zeichenkettenoperationen den normalen Parameter aus dem Referenzparameter extrahiert:

```
static bool CheckTypeRef(Type clientType, Type adviceType)
{
	string adviceTypeStr = adviceType.ToString();
	if(adviceTypeStr.EndsWith("&"))  // adviceType is a ref parameter!
	{
		// Remove the '&' char and check if assignable.
		string stripRef = adviceTypeStr.Remove(
			adviceTypeStr.Length-1, 1);
		Type adviceTypeNoRef = Type.GetType(stripRef);
		return adviceTypeNoRef.IsAssignableFrom(clientType);
	}
	return false;
}
```

Listing 5.5 Bestimmung des Typs aus einem Referenztyp und Prüfung auf Zuweisung

Leider lassen sich abgefangene Nachrichten nicht direkt verändern. Die einzige Möglichkeit, um Argumente einer Nachricht zu verändern, ist das „Verpacken“ der Nachricht in eine neue Nachricht mit Hilfe der Klasse `MethodCallMessageWrapper`. Nachrichten vom Typ `MethodCallMessageWrapper` lassen sich bearbeiten.

Kapitel 6: Fazit

Dieses Kapitel fasst die Ergebnisse dieser Arbeit zusammen, indem es allgemeine sowie technische Schlussfolgerungen präsentiert. Dabei werden auch Verbesserungsvorschläge gemacht und zu erwartende Entwicklungen geschildert.

6.1 AOSE als neues Programmierparadigma

Die AOSE ist eine Technik zur Modularisierung verstreuter Sachverhalte. Die Kapselung verstreuter Sachverhalte in Aspekten bietet der Softwareentwicklung viele Erleichterungen. Aspekte reduzieren nicht nur die Redundanz und Komplexität im Geschäftsmodell, sondern sind ein elegantes Mittel zur Lösung von systemtechnischen Problemen getrennt von der Geschäftslogik. Dadurch eröffnet sich die Möglichkeit, Software nur im Hinblick auf die nötigsten Eigenschaften zu entwerfen und dann nachträglich zu verschiedenen Zeitpunkten der Entwicklung mit Aspekten anzureichern. Nach [Ladd02] ist dies die Lösung für das „under/overdesign dilemma“. Darüber hinaus bilden Aspekte eine Alternative für diverse Entwickler-Tools wie Debugger und Profiler. Trotzdem löst die AOSE die objektorientierte Softwareentwicklung in keinster Weise ab[1], sondern vollstreckt ihre Ideologie: Getrennte Sachverhalte werden an verschiedenen Stellen getrennt voneinander modelliert. Zudem machen Kiczales et al. in [Kicz97] deutlich, dass das Anwendungsgebiet der AOSE nicht auf die OOP beschränkt ist.

Die Verwebung von Aspekt- und Komponentencode ist die Grundtechnik der AOSE. Mit Hilfe von Pointcuts werden Verwebestellen spezifiziert, an denen der Zielcode mit Funktionalität in Form von Advice (dynamisch) oder Introduction (statisch) angereichert wird. Eine Methode ohne direkte Manipulation des Quellcodes ist der Einsatz von Proxy-Klassen. Während die Erstellung von Proxies durch Vererbung vorwiegend in statischer Verwebung eingesetzt wird, sind Proxies nach dem Proxy-Entwurfsmuster gut für die dynamische Verwebung geeignet. Diese erlaubt die Zuweisung von Aspekten zur Laufzeit sowie die Verwendung von Light-Weight-Weavern in Form von Klassenbibliotheken. Um jedoch völlige Transparenz und Freiheit für den Komponentencode sowie größte Vielfalt in

[1] Genauso wenig wie die OOP Prozeduren abgelöst hat, nur dass diese jetzt in Klassen verpackt sind und den Namen „Methode“ tragen.

der Angabe von Aspekten, Advice, Pointcuts und Joinpoints zu erlangen, ist ein Aspekt-Compiler die bessere Lösung. Um Spracherweiterungen zu umgehen, bedienen sich viele Tools deklarativer Mittel wie XML-Dateien [Kim02], Attribute [RaDo04], oder Angaben in Dokumentationskommentaren [AWHW04].

Trotz der vielen Vorteile merkt man der AOSE noch ihre Unreife an. Zahlreiche auf der viel zitierten zentralen AOSE-Website [AOSD04] angegebene Projekte haben den Status von Forschungsprojekten und finden nur geringe Verbreitung. Viele von ihnen sind völlig undokumentiert und die in ihren Bezugsquellen vorhandenen Diskussionsforen enthalten seit Jahren keine Beiträge. Vor allem fehlen Anstrengungen zur Integration der AOSE in den Softwareprojektzyklus. Über die Verbreitung von Ansätzen zur UML-Unterstützung der AOSE wie z.B. ThemeUML [Them03] ist nichts bekannt. Zudem fehlen Konzepte zur Vereinheitlichung von AOSE-Systemen. Einerseits ist die heutige AOSE stark durch AspectJ geprägt; andererseits existiert eine Vielzahl von verwandten Ansätzen wie MDSoC oder Kompositionsfilter, die sich unter dem Oberbegriff „***A**dvanced **S**eparation **o**f **C**oncerns*“ (ASoC) etabliert haben [Flac03]. Leider fehlt die Konvergenz dieser Ansätze sowie Definitionen für einheitliche Frameworks. C. Flach und C. de Lucena machen einen Schritt in diese Richtung, indem sie ein Modell für AOP-Implementierungen ausarbeiten [Flac03].

6.2 .NET als Plattform für die AOSE

Die mächtige und reichhaltige Reflection-Klassenbibliothek von .NET hat sich als hilfreiches und zudem einfach zu handhabendes Werkzeug herausgestellt. Besonders nützlich sind das unkomplizierte Durchgehen von Assemblies sowie die Ermittlung vieler compilerspezifischer Eigenschaften, wie z.B. ob ein Typ einem anderen Typ zugewiesen werden kann. Doch finden sich auch zahlreiche Unstimmigkeiten in der Logik. Beispielsweise gibt es in der Klasse `ParameterInfo` eine Property, die ermittelt, ob ein Parameter ein `out`-Parameter ist; eine entsprechende Prüfung für `ref`-Parameter gibt es dagegen nicht. Die Property `IMethodMessage.HasVarArgs` liefert nicht das gewünschte Ergebnis. Für AOSE-Implementierungen dürfte die Methode `Type.GetMembers` sowie die Enumeration `BindingFlags` eine große Hilfe sein, mit denen man bestimmte Member einholen kann, deren Name einem angegebenen Muster entspricht. Aufgrund in vielen Fällen unerwarteter Ergebnisse kommen sie jedoch nicht in Spider.NET zum Einsatz. Trotz umfangreicher und flexibler Klassenbibliothek, mussten nützliche Funktionen in vielen Fällen selbst implementiert werden.

Eine große Erleichterung für die AOSE sind Attribute. Sie erlauben die Verwendung von benutzerdefinierten Ausdrucksmitteln und dienen zudem zur Kennzeichnung von Codestellen. Positiv ist, dass sich Attribute vererben lassen und dass man mit Kontextattributen implizit Dienste anfordern kann. Leider kann man Attributen nur Konstanten als Argumente übergeben. Ein großes Hindernis für eine dynamische AOSE-Implementierung ist die Statik der Attribute. Die Zuweisung von Attributen zur Laufzeit wäre eine entscheidende Verbesserung. Dies würde z.B. das Geschwindigkeitsproblem von Spider.NET lösen. Da Attribute ohnehin zur Objekterstellungszeit gebunden werden können, wäre dies realisierbar. Nach F. Piessens et al. verfügt .NET bereits über die Low-Level-Strukturen für eine solche Erweiterung [Pies03]. Sie gehen sogar noch weiter und schlagen die Zuweisung von Attributen zu Aufrufen und Beziehungen vor.

Lobenswert ist der Einsatz von Kontexten und damit die Möglichkeit, Interception in lokalen Anwendungen einsetzen zu können. Allerdings ist der Umgang mit Interception sehr schwierig. In vielen Fällen (z.B. bei Kontextattributen) verweigert die MSDN jegliche Hilfe mit der Formulierung: „Dieser Typ unterstützt die .NET Framework-Infrastruktur und ist nicht für die direkte Verwendung aus Ihrem Code gedacht“ [MSDN03].

Dies macht die Suche nach Funktionen und die Fehleranalyse problematisch. Zudem ist die Funktionalität zur Benutzung der Interception nicht zentral gekapselt, sondern verteilt auf

viele Klassen, statische Methoden und Namensräume und daher kaum überschaubar. Ihre Dokumentation muss in vielen Fällen in Online-Artikeln und Remoting-Büchern gesucht werden. Der Zweck der Interception scheint sich vorwiegend auf verteilte Applikationen zu konzentrieren und erlaubt an zahlreichen Stellen keine Eingriffsmöglichkeiten. Zwar gibt es eine verwirrend große Anzahl von Nachrichtentypen; Methoden lassen sich von Feldern oder Properties jedoch nur durch ihren Namen unterscheiden. Für zukünftige .NET-Versionen wäre es besser, die Interception vom Remoting ganz zu lösen und in einem speziell für sie vorgesehenen Namensraum zu kapseln.

Die Anwendung von Proxies für die AOSE ist zwar möglich, erfordert jedoch auch eine intensive Auseinandersetzung mit Remoting und zudem mit URLs. Eigenentwicklungen wie das in [Hamm04] vorgestellte „DynamicProxy“-Projekt [Aval04] sind eine logische Konsequenz aus diesem Mangel. Dagegen erlaubt Java die Erstellung eines dynamischen Proxy (`java.lang.reflect.Proxy`) mit Hilfe weniger Codezeilen [Java03]. Grundsätzlich ist Java aufgrund des zeitlichen Vorsprungs, aber wohl vor allem dank großem Stellenwert der Community um viele Hilfsmittel für AOSE-Implementierungen reicher. Der Einsatz von benutzerdefinierten Klassenladern ermöglicht Implementierungen mit Ladezeitverwebung. Bytecode-Modifikatoren wie die „Byte Code Engineering Library“ [BCEL02] und Javassist [JAss04] sind hervorragende Mittel für die statische und dynamische Verwebung.

Ein hilfreiches Mittel für die Wahrung der Transparenz in Implementierungen mit Proxies wäre die Überladung des `new`- sowie des Member-Zugriffsoperators (vgl. Abschnitt 3.3.2), die in C# nicht möglich ist. Auch der Einsatz von Schablonen wäre von großem Nutzen, da Schablonen Aspekte hoher Quantifikation unterstützen können. [Lohm04] zeigt, dass man mit Schablonen Aspekte durchaus emulieren kann. Für das .NET-Framework 2.0 sind Schablonen (dort „Generics“ genannt) vorgesehen [MSCS03].

Während der Analyse der Fähigkeiten von .NET und der Entwicklung von Spider.NET wurde oft der Anschein erweckt, dass Microsoft den Entwicklern zwar große Möglichkeiten geben möchte, gleichzeitig aber Angst hat, ihnen zu viel zu offenbaren und zu erlauben. Um die Infrastruktur für AOSE-Hilfsmittel wie MSIL-Code-Manipulatoren oder Klassenlader zu schaffen, sollte Microsoft nicht in Versuchung geraten, das Potenzial der Entwicklergemeinschaft zu vernachlässigen. Ferner wäre auch ein Tutorial für .NET sehr nützlich, wie es aus der Java-Welt bekannt ist. Lobenswert ist die Implementierung des Kontextmodells; zur vollen Unterstützung der AOSE fehlen jedoch noch wesentliche Schritte.

6.3 Spider.NET als Implementierung für die AOSE

Es ist gelungen, die Anforderungen der AOSE mit der Technik der .NET-Interception zu erfüllen. Vergleichbare Projekte wie [Shuk02], [Lowy03], [Lost04] und [Shak03] benutzen zwar die Interception als Ansatz, implementieren jedoch keine Benutzerschnittstelle und kein Pointcut-Modell nach dem Muster von AspectJ. Spider.NET hingegen bietet eine Schnittstelle, die den mit den klassischen AOP-Werkzeugen vertrauten Aspektprogrammierer anspricht und dabei weitgehend von der Technik abstrahiert. Spider.NET geht sogar noch weiter als AspectJ, indem es Aspekte, Pointcuts und Advice zur Laufzeit zuweisen lässt. Wie AspectJ bietet Spider.NET eine Integration in die Entwicklungsumgebung, jedoch völlig ohne zusätzliche Werkzeuge.

Das Ziel, den Funktionsumfang von AspectJ so weit wie möglich zu realisieren, wurde an folgenden Stellen erreicht:

- Verknüpfung von Pointcut-Angaben zu komplexen Pointcuts
- Unterstützung der Advice-Typen „before“, „after“, „after throwing“ und „around“
- `proceed`-Anweisung und `thisJoinPoint`-API
- „Context exposing“ wird realisiert durch die übergebenen Argumente sowie den Zugriff auf Properties der `ThisJoinPoint`-Klasse.
- Unterstützung der Aspekt-Modi „Singleton“ und „Multiple“

Nicht realisiert wurde das dynamische Joinpoint-Modell, die große Anzahl an Pointcut-Arten sowie die Introduction, was hauptsächlich an der verwendeten Technik liegt. Die Implementierung für die AOSE essenzieller Anwendungen ist jedoch möglich, wie Listing 6.1 zeigt: Ein Kryptografie-Aspekt benutzt Before-Advice, um Text zu verschlüsseln, bevor er in eine Datei geschrieben wird. Dementsprechend definiert er After-Advice, um Text zu entschlüsseln, der aus der Datei gelesen wurde:

```
public class Cryptography : Aspect
{
	[IncludeMethods("TargetApp.FileOperator", "Write")]
	[Advice(AdviceTypes.Before)]
	public void EncryptBeforeWritingFile(ref string str)
	{
		str = Encrypt(str);
	}

	[IncludeMethods("TargetApp.FileOperator", "Read")]
	[Advice(AdviceTypes.After)]
	public string DecryptAfterReadingFile()
	{
		return Decrypt(ThisJoinPoint.ReturnValue as string);
	}

	string Encrypt(string str){ … }
	string Decrypt(string str){ … }
}
```

Listing 6.1 Kryptografie-Aspekt unter Spider.NET

Auch Aspekte mit hohem Quantifikationsgrad lassen sich auf einfache Weise erstellen. Listing 6.2. zeigt einen Tracing-Aspekt, der vor Aufruf aller öffentlichen Methoden und Konstruktoren die Zielklasse, den Methodennamen und alle Argumente auf der Konsole ausgibt. Der Vergleich mit Listing 2.3 auf Seite 20 zeigt, dass die Lösung des Tracing-Problems mit Spider.NET weder komplizierter noch umfangreicher ist als die mit AspectJ.

```
public class Tracing : Aspect
{
	[Advice(AdviceTypes.Before)]
	[IncludeAll("*")]
	public object Trace(params object[] args)
	{
		Console.WriteLine(ThisJoinPoint.DeclaringTypeName+"."+
			ThisJoinPoint.MethodName);
		foreach(object o in args)
			Console.WriteLine(" "+o);
		return null;
	}
}
```

Listing 6.2 Tracing-Aspekt unter Spider.NET

Eine Schwachstelle von Spider.NET ist das durch Reflection und Interception beeinträchtigte Zeitverhalten. Tests ergaben einen Wert von 13000 Methodenaufrufen pro Sekunde mit Interception und 8000 mit Interception und Advice-Aktivierung. Bei einzelnen Methodenaufrufen konnten jedoch keine Geschwindigkeitseinbußen festgestellt werden. Zwar nimmt die Erstellung der Applikationsbasis je nach Größe der Applikation mindestens 0,04 Sekunden in Anspruch; sie wird jedoch nur einmal durchgeführt. Zudem bestätigte die Analyse der Technik von AspectJ, dass die Durchsuchung der Applikation nach Joinpoints und der Matching-Prozess zwangsweise langwierige Aktionen sind.

Die Architektur von Spider.NET ist skalierbar, sodass für zukünftige Implementierungen folgende Verbesserungen vorgeschlagen werden können:

- Erweiterung der unterstützten Ziel-Member-Typen auf Felder, Properties, Indexer und Delegaten.
- Erweiterung der Ausdrucksmöglichkeiten für die Spezifizierung von Pointcuts (z.B. Heraussuchen von Joinpoints anhand von Modifizierern).
- Komplexe Wildcard-Ausdrücke (z.B. unter Benutzung regulärer Ausdrücke).
- Verbesserung der Wiederverwendung durch Unterstützung abstrakter Aspekte.

In zukünftigen Versionen sollten auch die Eignung der Methoden `Type.GetMembers`, und `Type.InvokeMember`, sowie der Binding-Flags und Parameter-Attribute [MSDN03] ausgiebig untersucht werden.

Anhang 1: Abkürzungsverzeichnis

CLR	Common Language Runtime (Laufzeitumgebung von .NET)
CLS	Common Language Specification (Spezifikation zur Vereinheitlichung der .NET-Sprachen)
DLL	Dynamic Link Library
DOM	Document Object Model (Modell für abstrakte Syntaxbäume unter .NET)
ECOOP	European Conference on Object-Oriented Programming
ICAOSD	International Conference on Aspect-Oriented Software Development
ICSE	International Conference on Software Engineering
IDE	Integrated Development Environment
HLM	High-Level-Matching
HPI	Hasso-Plattner-Institut an der Universität Potsdam
JIT	Just In Time (Laufzeitkompilierung von Zwischencode in Maschinencode)
JVM	Java Virtual Machine
LLM	Low-Level-Matching
MSIL	Microsoft Intermediate Language (virtuelle Maschinensprache von .NET)
OOP	Objektorientierte Programmierung
OOPSLA	Conference On Object-Oriented Programming, Systems, Languages & Applications
PARC	Palo Alto Research Center
PE	Portable Executable (EXE- oder DLL-Datei, die ein .NET-Modul enthält)

Anhang 2: Informationen zur beiliegenden CD

Inhalt der CD

Dieser Diplomarbeit ist eine CD beigelegt. Sie enthält folgende Materialien:

- Diese Diplomarbeit im PDF-Format
- Im PDF- und HTML-Format zugängliche Literaturdokumente
- Quellprojekt von Spider.NET mit Dokumentation des Codes
- Testprojekte und Testprotokolle
- Handbuch und Installationsanleitung für den Benutzer
- Beispielprojekte und Tutorial
- Dokumentation der Spider.NET-Klassenbibliothek

Sämtliche Dokumentationen und Protokolle sind in englischer Sprache verfasst.

Empfohlene Vorgehensweise

Legen Sie die CD ein und führen Sie die Datei `index.html` aus. So gelangen Sie zu einer Auswahl der Materialien.

Zum Kennenlernen der Verwendung von Spider.NET werden die Beispielprojekte „Hello World“ und „Cryptography“ empfohlen, die im „Spider.NET Quick Start Tutorial“ dokumentiert werden.

Anhang 3: Bilderverzeichnis

Anhang 4: Tabellenverzeichnis

Anhang 5: Listingverzeichnis

Anhang 6: Literatur

Hinweis: Alle Online-Quellen sind auf dem Stand vom 21.09.2004.

[ACSh03] **Distributed Systems Group, Trinity College Dublin**
AspectC#: Intro
www.dsg.cs.tcd.ie/index.php?category_id=169

[AJFQ04] **Xerox Corporation, Palo Alto Research Center**
Frequently Asked Questions about AspectJ
dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/faq.html

[AOSD04] **AOSD Steering Committee**
Aspect-Oriented Software Development
aosd.net

[Arch02] **Tom Archer, Andrew Whitechapel**
Inside C#
2. Aufl. Microsoft Press, 2002

[AspC04] **pure-systems GmbH**
The Home of AspectC++
www.aspectc.org

[AspJ03] **Xerox Corporation, Palo Alto Research Center**
The AspectJ™ Programming Guide
dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html

[AspJ04] **AspectJ Project**
AspectJ Project
www.aspectj.org

[Aval04] **Apache Software Foundation**
Avalon Castle – Dynamic Proxy
avalon.apache.org/central/laboratory/castle/dynamicproxy

[AWer04] **Jonas Bonér, Alexandre Vasseur**
AspectWerkz – Dynamic AOP for Java
aspectwerkz.codehaus.org

[AWHW04] **Jonas Bonér**
AspectWerkz Hello World
docs.codehaus.org/display/AW/Hello+World

[BCEL02] **Markus Dahm**
Byte Code Engineering Library
bcel.sourceforge.net

[Booc94] **Grady Booch**
Objektorientierte Analyse und Design
Addison-Wesley, 1994

[Cacm01a] **Tzilla Elrad, Robert E. Filman, Alef Bader**
Aspect-Oriented Programming
In: Communications of the ACM. Band 44 (Oktober 2001), Nr. 10

[Cacm01b] **Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, Harold Ossher**
Discussing Aspects of AOP
In: Communications of the ACM. Band 44 (Oktober 2001), Nr. 10

[Cacm01c] **Karl Lieberherr, Doug Orleans, Johan Ovlinger**
Aspect-Oriented Programming with Adaptive Methods
In: Communications of the ACM. Band 44 (Oktober 2001), Nr. 10

[Cacm01d] **Harold Ossher, Peri Tarr**
Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software
In: Communications of the ACM. Band 44 (Oktober 2001), Nr. 10

[Cacm01e] **Lodewijk Bergmans, Mehmet Aksit**
Composing Crosscutting Concerns using Composition Filters
In: Communications of the ACM. Band 44 (Oktober 2001), Nr. 10

[Cacm01f] **Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold**
Getting Started with AspectJ
In: Communications of the ACM. Band 44 (Oktober 2001), Nr. 10

[Cacm01g] **Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, Joon Suan Ong**
Structuring System Aspects
In: Communications of the ACM. Band 44 (Oktober 2001), Nr. 10

[CoFi01] **Trese Research & Education on Software Engineering**
Aspect-Oriented Research on Composition Filters homepage
trese.cs.utwente.nl/oldhtml/composition_filters

[ComJ01] **Trese Research & Education on Software Engineering**
ComposeJ
trese.cs.utwente.nl/oldhtml/prototypes/composeJ/

[Dijk76] **E. W. Dijkstra**
A Discipline of Programming
Prentice Hall, 1976

[Diot04] **IBM developerWorks: Filippo Diotalevi**
Apply Design by Contract to Java software development with AspectJ
www-106.ibm.com/developerworks/library/j-ceaop/

[DJlb01] **Demeter Research Group, College of Computer Science, Boston**
DJ: Dynamic Structure-Shy Traversals and Visitors in Pure Java
www.ccs.neu.edu/research/demeter/DJ

[Dolo01] **Peter Dolog, Valentino Vranic, and Mária Bieliková**
Representing Change by Aspect
In: ACM SIGPLAN Notices. Band 36 (Dezember 2001), Nr. 1

[Film00] **Robert E. Filman, Daniel P. Friedman**
Aspect-Oriented Programming is Quantification and Obliviousness
Addison-Wesley, 2000

[Film03] **Robert E. Filman**
A Bibliography of Aspect-Oriented Software Development
Version 1.22, Research Institute for Advanced Computer Science, NASA Ames Research Center, 2003

[Flac03] **Christina von Flach G. Chavez, Carlos J. P. de Lucena**
A Theory of Aspects for Aspect-Oriented Software Development
In: Simpósio Brasileiro de Engenharia de Software 2003. Manaus. 2003

[Gamm96] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**
Entwurfsmuster
Addison-Wesley, 1996

[Hamm04] **Hamilton Verissimo**
Java-like proxies in .Net
jroller.com/page/hammett/Weblog/java_like_proxies_in_net

[Harr93] **William Harrison, Harold Ossher**
Subject-oriented programming: a critique of pure objects
In: OOPSLA 1993. Washington, USA. ACM press, 1993

[Hils04] **Erik Hilsdale, Jim Hugunin**
Advice Weaving in AspectJ
In: ICAOSD 2004. Lancaster, Großbritannien. ACM press, 2004

[HypJ03] **IBM Alphaworks**
HyperJ
www.alphaworks.ibm.com/tech/hyperj

[JAC04] **ObjectWeb Consortium**
The JAC Project
jac.objectweb.org

[JAss04] **Shigeru Chiba**
Javassist
www.javassist.org

[Java03] **Sun Microsystems**
Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification: Proxy
java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/Proxy.html

[JBos04] **Bill Burke**
JBoss Aspect Oriented Programming
www.jboss.org/products/aop

[Kicz96] **Gregor Kiczales, John Irwin**, **John Lamping**, **Jean-Marc Loingtier**, **Cristina Videria Lopes**, **Chris Maeda**, **Anurag Mendhekar**
Aspect-Oriented Programming
In: ACM Computing Surveys. Band 28 (Dezember 1996), Nr. 4

[Kicz97] **Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin**
Aspect-Oriented Programming
In: ECOOP 1997. Jyväskylä, Finnland. Springer-Verlag, 1997

[Kicz01] **Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold**
An Overview of AspectJ
In: ECOOP 2001, Budapest, Ungarn. Springer-Verlag, 2001

[Kim02] **Howard Kim**
AspectC#: An AOSD implementation for C#
Diss. Trinity College Dublin, 2002

[Ladd02] **Java World: Ramnivas Laddad**
I want my AOP!, Part 1
www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html

[Ladd03] **Ramnivas Laddad**
AspectJ in Action
Manning, 2003

[Ladd03a] **TheServerSide: Ramnivas Laddad**
Aspect-Oriented Refactoring Series: Part 1 – Overwiev and Process
www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1

[Ladd03b] **TheServerSide: Ramnivas Laddad:**
Aspect-Oriented Refactoring Series: Part 2 – The Techniques of the Trade
www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart2

[Laff03] **Chris Laffra, Martin Lippert**
Visualizing and AspectJ-enabling Eclipse Plugins using Bytecode Instrumentation
In: OOPSLA 2003. Anaheim, USA. ACM press, 2003

[Lohm04] **Daniel Lohmann, Olaf Spinczyk, Andreas Gal**
Aspect-Oriented Programming with C++ and AspectC++
In: ICAOSD 2004. Lancaster, Großbritannien. ACM press, 2004

[Loom03] **Hasso-Plattner-Institut an der Universität Potsdam**
LOOM .NET Reference
Hasso-Plattner-Institut an der Universität Potsdam, 2003

[Loom04] **Operating Systems and Middleware Group at HPI**
Welcome to the LOOM .NET Project!
www.rapier-loom.net

[Lost04] **CodeProject: Lostinet**
Build an AOP.NET Extensible Business Component using ContextBoundModel
www.codeproject.com/dotnet/ContextBoundModel.asp

[Lowy03] **Microsoft Corporation: Juval Lowy**
Decouple Components by Injecting Custom Services into Your Object's Interception Chain
MSDN Magazine, März 2003
msdn.microsoft.com/msdnmag/issues/03/03/ContextsinNET/default.aspx

[McLe02] **Scott McLean, James Naftel, Kim Williams**
Microsoft .NET Remoting
Microsoft Press, 2002

[Meye97] **Bertrand Meyer**
Object-Oriented Software Construction
Prentice Hall, 1997

[MSCS03] **Microsoft Corporation**
A Sneak Preview of Visual C# Whidbey, November 2003
MSDN Developer Center, November 2003
msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/whidbey_csharp_preview.asp

[MSDN03] **Microsoft Corporation**
MSDN Library für Visual Studio .NET 2003
Microsoft Corporation, 2003

[Nolt04] **Mathew Nolton**
Attribute Oriented Programming != Aspect Oriented Programming
weblogs.asp.net/mnolton/archive/2004/04/23/119181.aspx

[Oest01] **Bernd Oestereich**
Objektorientierte Softwareentwicklung
Oldenbourg, 2001

[Page94] **Bernd-Uwe Pagel, Hans-Werner Six**
Software Engineering
Band 1, Addison-Wesley, 1994

[Parc04] **Palo Alto Research Center Incorporated**
AspectJ
www.parc.com/research/csl/projects/aspectj/default.html

[Pies03] **Frank Piessens, Bart Jacobs, Eddy Truyen, Wouter Joosen**
Support for Metadata-driven Selection of Run-time Services in .NET is Promising but Immature
In: Journal of Object Technology. Band 3, Nr. 2. ETH Zurich, 2004

[RaDo04] **Hasso-Plattner-Institut an der Universität Potsdam**
Rapier-Loom.Net Documentation
www.dcl.hpi.uni-potsdam.de/RAPIER-LOOM

[Robi02] **Simon Robinson**
Advanced .NET Programming
Wrox Press, 2002

[Sach03] **Dominik Sacher**
Orthogonale Komponenten in Fahrerinformationssystemen
Diplomarb. Fachhochschule Gießen-Friedberg, 2003

[Schu02] **Wolfgang Schult, Andreas Polze**
Aspect-Oriented Programming with C# and .NET
Hasso-Plattner-Institut an der Universität Potsdam, 2002

[Schu03] **Wolfgang Schult, Andreas Polze**
Aspect-Oriented Programming with C# and .NET
Hasso-Plattner-Institut an der Universität Potsdam, 2003

[Schu04] **Wolfgang Schult, Andreas Polze**
Dynamic Aspect-Weaving with .NET
Hasso-Plattner-Institut an der Universität Potsdam, 2004

[Shak03] **CodeProject: Motti Shaked**
.NET Remoting Customization Made Easy: Custom Sinks
www.codeproject.com/csharp/customsinks.asp

[Shuk02] **Microsoft Corporation: Dharma Shukla, Simon Fell, Chris Sells:**
Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse
MSDN Magazine, März 2002
msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx

[Siev03] **Microsoft Corporation: Jim Sievert**
Create a Custom Marshaling Implementation Using .NET Remoting and COM Interop
MSDN Magazine, September 2003
msdn.microsoft.com/msdnmag/issues/03/09/CustomMarshaling/default.aspx

[Spin04] **Olaf Spinczyk**
AC++ Compiler Manual
pure-systems GmbH, 2004

[Tarr99] **Peri Tarr**
N Degrees of Separation: Multi-Dimensional Separation of Concerns
In: ICSE 1999. Los Angeles, USA. ACM press, 1999

[Them03] **Distributed Systems Group, Trinity College Dublin**
ThemeUML: Intro
www.dsg.cs.tcd.ie/index.php?category_id=355

[Urba04] **Matthias Urban, Olaf Spinczyk**
AspectC++ Language Reference
pure-systems GmbH, 2004

[West01] **Ralf Westphal**
.NET kompakt
Spektrum Akademischer Verlag, 2001

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, den 23. September 2004 ……