Vergleichende Fallstudie über Techniken für wiederverwendbare Aspekte

Diplomarbeit

von
Henry Sudhof
Matrikelnummer: 194239
<hsudhof@cs.tu-berlin.de>

Berlin, den 03. März 2006

Diplomarbeit an
der Fakultät IV
der Technischen Universität Berlin
Institut für Softwaretechnik und
Theoretische Informatik

Betreuer:
Dr.-Ing. Stephan Herrmann
Gutachter:
Prof. Dr.-Ing. Stefan Jähnichen
Dr.-Ing. Stephan Herrmann

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 03. März 2006 _____

Unterschrift (Henry Sudhof)

Abstract

REUSE has always been a key factor in the case for new design and programming paradigms. *Aspect-Orientation* is no exception to that rule. However, so far there are very few approaches actually delivering on that promise.
This case study compares two leading approaches for *reusable* aspects, both based on *Aspectual Components*: ObjectTeams and JAsCo. This is done by implementing a set of session-management and authorization aspects for oblivious base programs. In that process, best-practices for design and implementation of *reusable aspects* are discussed.

The study explores the possibilities that aspectual components derived AOP approaches offer for the implementation of cooperating aspect designs for a-posteriori integration.
The findings indicate that aspect reusability is still dependent on considerable push on the base application and requires careful design of both aspects and base code. Furthermore, implementing aspect-integration code requires close familiarity with the base application's design and implementation details.
In conclusion, the work offers proposals to enhance aspect reuse in future versions of the examined languages.

Inhaltsverzeichnis

1 Einleitung

Wiederverwendbarkeit ist ein häufig anzutreffendes Ziel der Softwaretechnik, sowohl in der Praxis, als auch in der Theorie. Unabhängig von der Ebene der Betrachtung, sei es für Entwurfs-, oder Implementierungstechniken; gelungene Formulierungen oder ganze Systeme: Wiederverwendbarkeit ist ein zentrales Merkmal.
Dabei ist der Begriff eng mit *Modularisierung* verknüpft; Module sollen zugleich Einheiten der Wiederverwendung sein.
Auch auf dem vergleichsweise jungen Gebiet der *aspektorientierten Softwareentwicklung* stellt sich die Frage der Wiederverwendung. Diese Studie vergleicht zwei aspektorientierte Ansätze, die Wiederverwendung durch Modulkonzepte erleichtern sollen, in ihrer Eignung für die Implementierung von *wiederverwendbaren Aspekten*.

1.1 Geschichte

Modularisierung ist eines der ältesten Konzepte in der Softwaretechnik überhaupt: bereits Anfang der Siebziger Jahre formulierte David Parnas [Par72] Kriterien zur *Dekomposition* von Software in Module. Er zeigte in jenem Aufsatz, dass der Programmablauf nicht als zentrales Kriterium für die Aufteilung von großen Programmen in Module Anwendung finden sollte, sondern strukturelles Design erforderlich ist. Edsger Dijkstra prägte in [Dij74] den dafür häufig verwendeten Begriff "*Separation of Concerns*", um das Ideal einer Trennung unterschiedlicher *Aspekte* zu beschreiben.
Noch heute sucht die Softwaretechnik nach geeigneten Modulkonzepten, um mit den stetig wachsenden Anforderungen an Software Schritt zu halten.

Diese Entwicklung führte zur Durchsetzung der Objektorientierung, mit dem Versprechen, Wiederverwendbarkeit und Entwicklungsmethoden, zu verbessern. Zur Anerkennung der Objektorientierung führten dabei insbesondere aufbauende Arbeiten, nicht zuletzt hervorragend wiederverwendbare Bibliotheken kooperierender Klassen – *Frameworks* – für grafische Oberflächen[1].

Dabei sind die Schwächen der Objektorientierung nicht in Vergessenheit geraten. Die Modularisierung in Klassen neigt dazu, nur in einer Dimension zu trennen. Funktionen die sich nicht unbedingt einer Klasse zuordnen lassen, verteilen sich über mehrere Programmteile und vermischen sich mit anderen Funktionen (*Scattering* und *Tangling*). Diese Beobachtung wird auch als *Tyrannei der dominanten Dekomposition* (*Tyranny of the dominant decomposition*, [TOJH04]) bezeichnet und ließ den Ruf nach einem Modulkonzept für solche übergreifenden Funktionen – *Crosscutting Concerns* – aufkommen.

[1] Beginnend mit *Model-View-Controller* ([KP88]) basierenden Architekturen in Smalltalk-80.

1.2 Aspektorientierung

Gregor Kiczalez et al. schlugen die Aspektorientierung 1997 als neues Paradigma in ihrem Aufsatz "*Aspect Oriented Programming*" [KLM+97] vor, um solche *crosscutting concerns* in Module aufteilen zu können. In den darauf folgenden Jahren wurde dieser Ansatz zur Basis zahlreicher Untersuchungen und Vorschläge, wodurch sich *aspektorientierte Softwareentwicklung*, kurz *AOSD*[2], als eigene[3] Richtung mit dem Ziel eines neuen Modulkonzeptes, einer *"Advanced Separation of Concerns"* etablierte.
Eine rückblickende Definition der Aspektorientierung bieten Filman und Friedman in [FF04] wo sie *Obliviousness* und *Quantification* als die entscheidenden Kriterien für die Aspektorientierung anführen.
Obliviousness – Unbemerktheit – ist dabei das Ideal, dass Aspektmodule ohne explizite Bindung innerhalb des *Codes* eingebunden werden. Daraus folgt eine überlegende Modularisierung, da so zumindest in einer Richtung keine Abhängigkeiten bestehen.
Quantification[4] dagegen bezeichnet die Möglichkeit über Struktur und Zustand des Programmes hinweg zu quantifizieren. Also, nach Filman und Friedman, der Wunsch nach Ausdrücken der Form

> "In programs P, whenever condition C arises, perform action A"

. Dieses Konzept ist in der vorherrschenden AOP-Sprache „AspectJ" umgesetzt als Dreiklang von *Joinpoints*, *Pointcuts* und *Advice*. *Joinpoints* sind Stellen im Programmfluss einer *Basisanwendung*. *Pointcuts* wiederum sind Ausdrücke, die über *Joinpoints* quantifizieren.

Auf Basis von *Pointcuts*, bilden *Advices* den agierenden Part der Technik – die *Action A*: Erreicht der Programmfluss einen *Joinpoint*, der einen *Pointcut* erfüllt, so wird der zu dem Pointcut passende *Advice* ausgeführt.
Ein wenig aus diesem Schema heraus fallen die sogenannten *Introductions*: mit diesem Instrument erlauben viele AOP-Sprachen auch die Erweiterung der statischen Struktur eines Programmes, um z.B. neue Felder und Methoden zu bestehenden Klassen hinzuzufügen; *Advice*, *Pointcuts* und *Introductions*[5] sind dabei die üblicherweise als *Aspekt* oder Aspektcode bezeichneten Elemente. Eine weitergehende Einführung findet sich z.B. in [Lee02].

Diese grundlegenden Bausteine der Aspektorientierung werden durchaus kontrovers diskutiert. Die veränderte Lokalität des Codes wird so beispielsweise in [CSS04], in Anlehnung an Dijkstras "GoTo considered harmful" ([Dij68]), als *„schädlich"* – *„harmful"* – bezeichnet. Selbstverständlich war es den Autoren an der Überspitzung gelegen.

[2] **A**spect-**O**riented **S**oftware **D**evelopment
[3] Wobei andere, teils ältere, Ansätze in ihr aufgingen
[4] Quantifizierung
[5] In AspectJ *Inter-type Declarations*

1.3 Wiederverwendbarkeit

[IEE90] definiert die Wiederverwendbarkeit als den „*Grad zu dem ein Softwaremodul oder ein anderes Arbeitsergebnis, in mehr als einem Computerprogramm oder Softwaresystem genutzt werden kann*“.
Bei der Wiederverwendung wird üblicherweise zwischen „*White-Box Reuse*“ und „*Black-Box Reuse*“ unterschieden. Ersteres beschreibt Wiederverwendung, bei der die Kenntnis der Struktur des wiederverwendeten Moduls erforderlich ist; *Black Box Reuse* beschreibt Module, die direkt und ohne Kenntnis ihrer Interna für eine Wiederverwendung geeignet sind.

1.3.1 Wiederverwendung in der Objektorientierung

Es überrascht nicht, dass für dieses momentan am häufigsten anzutreffende Paradigma eine Vielzahl von Überlegungen für Wiederverwendung auf jeder erdenklichen Ebene existiert. Zwei besondere Teilgebiete sind aber für diese Studie von besonderem Interesse: Die Wiederverwendung von Entwurfsmustern (*Design Patterns*, [GHJV95]) und die von halbabstrakten Anwendungen: sogenannten *Application Frameworks*[6].

1.3.2 Komponenten

Komponenten sind eine in Anlehnung an Komponenten elektronischer Schaltungen, besonders auf Wiederverwendbarkeit ausgelegte Form von Modulen[7]. Anerkannte Eigenschaften für Komponenten sind das vollständige Abdecken einer Aufgabe, bei gleichzeitigen Vorhandensein expliziter Schnittstellen und Abhängigkeiten. Insbesondere aber auch das Zusammenfügen durch dritte – von den Entwicklern der Komponente verschiedenen – Personen. Komponenten sind üblicherweise *Black Box* Module, verbergen also ihre Details.

1.3.3 Aspekte

Die Aspektorientierung ist ein aktiver Schwerpunkt in der Forschung auf dem Gebiet der Softwaretechnik, zunehmend auch der Praxis. Oft wird aspektorientierter Software eine verbesserte Wiederverwendbarkeit attestiert, die aber bei näherer Betrachtung nicht uneingeschränkt auf die aspektorientierten Teile solcher Software zutrifft.
Es ist eine anerkannte Überlegung, dass eine bessere Modularisierung die Wartbarkeit von Software verbessert (z.B. [Par72]). Dies trifft auch auf die Aspektorientierung zu. Eine saubere Trennung von *crosscutting concerns* erleichtert Wartung, Erweiterung und Wiederverwendung bestehender Systeme. Dabei fällt aber eine konzeptionelle Lücke in vielen Ansätzen auf: Für Aspekte[8] selbst existieren wenige über den Stand von Überlegungen hinaus entwickelte Ansätze, um die Wiederverwendung zu erleichtern oder, um

[6] *Application Frameworks* sind nicht zwingend objektorientiert.
[7] Auch Komponenten sind zumeist auf Basis des objektorientierten Paradigmas verfasst. Der Komponentenbegriff stammt von McIlroy [MCI68] und ist damit ebenso alt, wie die Softwaretechnik selbst.
[8] Hiermit sind aspektimplementierende Module gemeint.

sie schlicht zu ermöglichen. Viele der bestehenden Überlegungen konzentrieren sich auf aspektorientierte Äquivalente der objektorientierten Design Patterns (etwa [HK02] und [Les05]), befassen sich daher eher mit der Wiederverwendung von Entwürfen und Entwurfselementen, als mit der Wiederverwendung von Aspekten selbst.

Das spezifische Problem der Wiederverwendung ist die Notwendigkeit Aspekte an die Typen der Basis zu binden[9], ohne Möglichkeiten wie Polymorphie und klar definierte Schnittstellen nutzen zu können. Die Ursache hierfür ist im Wesen von Aspekten und *Crosscutting Concerns* ebenso zu suchen, wie in den ersten Überlegungen zur aspektorientierten Programmierung.
Dies ist nicht als Nachteil zu bewerten, steht doch die erleichterte Wiederverwendung der Basisanwendungen außer Frage; eine Wiederverwendung von Aspekten und nicht nur durch Aspekte, bleibt aber offen. Dieses Ziel wahrhaft wiederverwendbarer Aspekte ist dabei aber weiter Ziel der Forschung und eine wichtige Etappe auf dem Weg zur Anerkennung von aspektorientierter Softwareentwicklung.

1.3.4 Aspektkomponenten

Die Kombination von Komponenten und Aspekten zu *Aspektkomponenten* ist eine der Überlegungen, um eine Wiederverwendung von Aspekten zu ermöglichen. Zentral ist dabei die Idee, durch neue Schnittstellen von der konkreten Basis zu abstrahieren und dadurch Aspekte generisch und wiederverwendbar gestalten zu können.
Dieser Gedanke von Aspektkomponenten („*Aspectual Components*“, [LLM99]) liegt auch den beiden für diese Arbeit als Untersuchungsgegenstände dienenden Ansätzen zu Grunde: *JAsCo* und *ObjectTeams*.

1.4 Untersuchungsziel

Die Studie verfolgt den Vergleich der beiden Techniken *JAsCo* und *ObjectTeams*, die als führende Ansätze für wiederverwendbare Aspekte zu betrachten sind. Die Untersuchung zielt auf die Erforschung der Ansätze, sowohl auf der Entwurfsebene als auch hinsichtlich der konzeptionellen und technischen Reife. Existierende Schwachpunkte, aber insbesondere auch Stärken, sollen hervorgehoben werden. Dabei soll Wiederverwendung in einem Vergleich der untersuchten Techniken als entscheidendes Merkmal in praxisnahen Szenarien dienen. Eine relative Bewertung der Sprachen wird hierbei nicht verfolgt, sind diese Ansätze doch noch sehr früh in ihrer Entwicklung. Abschließend sollen diese vergleichenden Beobachtungen Verwendung finden, um Vorschläge für die Weiterentwicklung der Sprachen zu formulieren.

[9] Dieser Zwang folgt nicht (nur) aus den Erfordernissen der Typsysteme, sondern schlicht aus der Natur der Sache.

1.5 Struktur

Zunächst wird Kapitel 2 kurz in JAsCo und ObjectTeams einführen.
Kapitel 3 soll einen Überblick über die grundlegenden Überlegungen im Zusammenhang mit dem Entwurf wiederverwendbarer Aspekte bieten.
Kapitel 4 stellt die Rahmenbedingungen der Fallstudie dar, auch die Szenarien und die gewählten Regeln.
Kapitel 5 ergänzt, vor der Fallstudie, die Vorstellung der Sprachen durch Überlegungen über den Umgang mit ObjectTeams und JAsCo.
Kapitel 6 stellt die Fallstudie und ihre Ergebnisse vor. Dabei wird auch eine knappe Beschreibung der Entwürfe gegeben, ergänzt durch ausgewählte Implementierungsdetails.
Kapitel 7 greift die Auswertung der Fallstudie auf, um auf Schwachstellen der untersuchten Konzepte und auf mögliche Lösungsansätze hinzuweisen.
Kapitel 8 summiert abschließend die Erkenntnisse.

2 Vorstellung der untersuchten Techniken

Dieser Abschnitt soll einen Überblick über die zu untersuchenden Techniken vermitteln und die grundlegenden Unterschiede und Gemeinsamkeiten der Ansätze verdeutlichen. Dabei wird eine gewisse Vertrautheit mit der Aspektorientierung im Allgemeinen und *AspectJ*[1] im Besonderen vorausgesetzt.

2.1 Aspectual Components

Beiden untersuchten Ansätzen ist gemein, dass sie auf dem Modell *Aspectual Components* beruhen. Diese *Aspectual Components* sind ein Ende der 90er Jahre von Mezini et al. vorgeschlagener ([LLM99]) Ansatz, der durch eine Verbindung von Aspekten und Komponenten eine wiederverwendbare Form aspektorientierter Module verspricht. Dabei ist ein *Aspectual Component* definiert als ein Konstrukt, bestehend aus einem Teilnehmergraph (PC, *Participant Graph*), eigenen *Features*, eingeschlossenen Klassen und komponentenweiten Definitionen. Der Teilnehmergraph ist hierbei die bemerkenswerteste Idee: Ein Teilnehmer ist ein abstrakter Platzhalter der Konstruktion, der durch zwei Schnittstellen, *expected* und *replace*, beschrieben wird und über lokale Features verfügen kann.
Mit *expect* markierte Features sind dabei Funktionen, die mögliche Basisklassen bereitzustellen haben – für die Kommunikation in die Basis. *Replace* bezeichnet Punkte zur Kommunikation aus der Basis in den *Aspectual Component*. Die Teilnehmer werden zu einem späteren Zeitpunkt durch dedizierte *Connectors* an Basisklassen gebunden, wobei in diesen beschrieben wird, wie die *expect* und *replace* Features auf die jeweilige Basisklasse anzuwenden sind.
Diese Grundidee wurde in ObjectTeams und JAsCo weitgehend unterschiedlich interpretiert, eine fundamentale Ähnlichkeit fällt aber bei der näheren Betrachtung auf.

2.2 JAsCo

JAsCo[2]([SVJ03]) ist ein der Tradition AspectJs verpflichteter Aspektkomponentenansatz, der durch Verzicht lexikalischer Bindungen in konkreten Aspekten die Wiederverwendbarkeit von Aspekten zu verbessern verspricht.
Dafür führt JAsCo das Konzept der *AspectBean* ein; ein an *JavaBeans* angelehntes Konstrukt, um Aspekte in Komponenten zu verwirklichen.

[1] `http://www.eclipse.org/aspectj/`
[2] `http://ssel.vub.ac.be/jasco/`

Weitere zugrundeliegende Konzepte sind *Composition Adapters* und der graphische Entwicklungsansatz *PacoSuite*[3]. Entwickelt wird er an dem *Software Engineering Lab* der Belgischen *Vrije Universiteit Brussel*.
Die Entwicklung von JAsCo zeichnet sich einerseits durch eine besondere Rücksicht auf Performanz aus, andererseits durch eine hohe Bereitschaft zur Berücksichtigung neuer Funktionen.
Der folgende Abschnitt soll einen Eindruck von JAsCo vermitteln und die sprachlichen und technischen Mittel detailliert vorstellen. Als Basis für diesen Abschnitt fungieren die auf Java 5 basierende Implementierung von JAsCo in Version 0.8.7 und die Sprachdefinition [Van05], welche informell und beispielsorientiert gehalten ist, weshalb, im Falle eines Widerspruchs von Sprachbeschreibung und Sprachimplementierung, der Implementierung der Vorrang eingeräumt wird[4].

2.2.1 Ansatz

Abstrakte Aspekte und dedizierte Verbindungen, so lässt sich die Idee von JAsCo formulieren. Die Sprache bietet Module zur Auslagerung basisabhängiger Teile aus dem Aspekt – der *AspectBean*. Dabei wird die Möglichkeit diese unverändert zu verwenden nicht aufgegeben.

2.2.1.1 Aspect Beans und Hooks

Die Grundidee JAsCos ist die Teilung der Aspekte in basisabhängige *Connectors* und unabhängige *Aspect Beans*. AspectBeans sind hierbei eine neue Art von *Java-Beans*, die *crosscutting concerns* modularisieren. Die eigentlichen Aspekte, *Pointcut* und *Advice* Paarungen, werden dabei in *Hooks*, eine spezielle Art *inner Classes* von *AspectBeans*, implementiert. Hierbei werden *Pointcuts* lediglich als abstrakte Schablonen in Form von Hookkonstruktoren angegeben; die Definition von passenden konkreten *Pointcuts* und basisabhängigen (*Advice*-) Methoden findet in Form von speziellen Konstruktoraufrufen in den *Connectors* statt, welche die einzige Möglichkeit darstellen, Instanzen von *Hooks* zu erzeugen.
Hooks können maximal einen Konstruktor definieren. Diese Konstruktoren können keinen Java-Code enthalten, sondern lediglich Verknüpfungen abstrakter Pointcuts.

2.2.1.2 Vererbung

Im Unterschied zu AspectJ, erlaubt JAsCo die Erweiterung konkreter Aspekte. Sowohl können *Aspect Beans* als auch einzelne *Hooks* kovariant vererbt bzw. redefiniert werden. Dabei existiert in gewisser Hinsicht *Late Binding of Advice*, was jedoch durch die ausschließliche Aspektinstantiierung in Connector-Dateien keine zentrale Rolle gewinnt.
Zusammenfassend sind die Möglichkeiten der Aspektvererbung hinsichtlich Polymorphie

[3] Beide Vrije Universiteit Brussel.
[4] Erkentnisse aus Korrespondenz mit den JAsCo Entwicklern wurden mitunter vorauseilend als „Realität“ angenommen.

eingeschränkt. *Hooks* in Sub-AspectBeans stehen in keiner Subtypbeziehung zu namensgleichen *Hooks* der ursprünglichen Implementierung.

Die einzige Möglichkeit, *Hooks* in eine Subtypbeziehung zu stellen, ist die explizite Erweiterung von Hooks über das Schlüsselwort `extends`. Dabei sind lediglich Hooks als Superhooks zulässig, welche bereits in einer Super-AspectBean definiert wurden; Vererbung innerhalb einer *AspectBean* ist nicht möglich. Die Semantik dieser Hookvererbung ist erweiternd und erlaubt lediglich die Ergänzung und Redefinition von *Advice* und Methoden; abstrakte Pointcuts können nicht redefiniert werden.

2.2.1.3 Instantiierung

AspectBeans können implizit oder explizit instantiiert werden. Üblicherweise erstellt die Laufzeitumgebung genau eine Instanz für jedes relevante *Connector*-Modul. Alternativ können auch Konstruktoren explizit in *Connector*-Dateien aufgerufen werden, wobei auch eine explizite Instantiierung von *AspectBeans* der Regel folgen muss, dass pro *Connector* nur eine Instanz einer gegebenen *AspectBean* erzeugt werden darf.
Flexibler gestaltet sich die Erzeugung der enthaltenen *Hook*-Instanzen. Sofern nicht anders deklariert, wird hier genau eine *Hookinstanz* pro *Hook-Konstruktoraufruf*[5] erstellt. Darüber hinausgehend bietet JAsCo *AspectFactories* [6]; Factory- Objekte, um dynamisch mehrere Hookinstanzen zu erzeugen.
Vordefinierte *Aspect Factories* implementieren z.B `perall`, `perclass`, `perobject`, `permethod`, `perthread` und `percflow`. JAsCo lässt auch die Definition neuer *Aspect Factory* Klassen zu, über welche die Instantiierung von *Hooks* gesteuert werden kann.
Dies wird durch die Möglichkeit des Einsatzes von Java-Code in den Connector unterstützt. Dieses Instrument erlaubt den Aufruf von (Konfigurations-)Methoden auf neu erzeugten *Hook*- und *AspectBean* Instanzen.

2.2.1.4 Joinpoints und Pointcuts

JAsCo verwendet ein an AspectJ angelehntes Joinpointmodell. Es verwendet den Aufruf/-die Ausführung von Methoden sowie das Auslösen von *Events* als mögliche *Joinpoints*. Die darauf aufbauende *Pointcutsprache* arbeitet lexikalisch mit '*' und '?' als *Wildcards*. Geprüft werden Annotations, Rückgabetypen, Package, Klasse[7] und Argumente.
Zusätzlich zu dem grundlegenden `execution` kennt die Sprache eine Reihe weiterer *Pointcutdesignatoren*: `cflow`, `call`[8] , `withincode` und `target`[9]. Diese Bedingungen können logisch verknüpft und negiert werden. Ferner existieren verteilte *Pointcuts*, auf die an

[5] Also der Zuweisung konkreter Werte an einen abstrakten *Pointcut*.
[6] In AspectJ Pointcut Designatoren
[7] inklusive implementierter Interfaces und Superklassen
[8] In der vorliegenden Version 0.8.7 zwar vom Compiler zugelassen, aber nicht implementiert. Die Dokumentation ist widersprüchlich in der Frage, ob `call` in einer zukünftigen Version von JAsCo unterstützt werden wird. Die Sprachbeschreibung [Van05] führt `call` nur am Rande auf. Die Kurzbeschreibung [Van06] kündigt die Implementierung für JAsCo 0.9 an.
[9] Auch `target` ist in der Version 0.8.7 nicht implementiert.

anderer Stelle eingegangen werden wird (Siehe: 2.2.1.14).
Auf der Pointcutebene setzt auch JAsCos zentraler Gedanke für wiederverwendbare Aspekte an. Da lexikalische Pointcutausdrücke naturgemäß abhängig von den jeweiligen Basisprogrammen formuliert werden müssen, ist die Angabe des lexikalischen Teils der *Pointcuts* bei JAsCo aus dem eigentlichen Aspektkomponenten ausgegliedert. So sollen konkrete Aspekte für eine Wiederverwendung hinreichend abstrakt gehalten werden können.
Genauer wird der lexikalische Teil von *Pointcuts* in die *Connectors* verlegt. Demgegenüber verbleibt der strukturelle Teil im Konstruktor des *Hooks* . Der lexikalische Teil wird dann als Parameter im Hookkonstruktor übergeben.

```
1 class MyAspectComponent {
    hook MyHook {
3     MyHook(method(int i)) {
        execution(method);
5     }
      ...
7   }
}
```

Listing 2.1: Einfacher Hookkonstruktor. In Zeile 4 wird der abstrakte *Pointcutparameter* `method` teilweise abstrakt gehalten.
Es erfolgt lediglich eine Festlegung auf Methoden, die ein Argument vom Typ `int` erwarten. `Method` muss in Connector-Dateien genauer angegeben werden.

```
static connector MyConnector {
2   MyAspectComponent.MyHook hook = new MyAspectComponent.MyHook(
      @myAnnotation * *.*.*(**));
}
```

Listing 2.2: Passend zu Listing 2.1: Instantiierung von `MyHook`. Alle Methoden mit der Annotation „`myAnnotation`“, die einen `int` als Argument erwarten, würden den Aspekt auslösen.

2.2.1.5 Advice

JAsCo erlaubt verschiedene Arten von *Advice*, die sich in dem Zeitpunkt ihres Eingreifens in den Programmfluss der Basisanwendung unterscheiden. Jeder *Hook* kann für die grundlegenden *Advice*-Arten – vor (`before`), nach (`after`) und an Stelle von (`around`)der Basismethode – maximal eine Definition angeben.
Über die drei grundlegenden *Advice*arten hinaus unterstützt JAsCo `after throwing`, `around throwing`, `after returning`, und `around returning` Advices. Die ersteren beiden erlauben es, auf geworfene *Exceptions* zu reagieren. Die letzteren erlauben es, den Rückgabewert der Basismethode abzuwandeln. Allen vieren ist gemein, dass sie die Angabe einer Signatur für die abgefangene *Exception* bzw. den Rückgabewert erfordern. Dies erlaubt *Overloading*, so dass mehrere solcher *Advices* in einem *Hook* angegeben werden können.

2.2.1.6 Refining Classes

JAsCos Werkzeug, um die Kommunikation in Richtung der Basis wiederverwendbar zu halten, sind *refinable* Methoden.

Dieserart deklarierte Methoden können in *Connector*-Dateien oder in dedizierten *Refining Classes*, an Basisprogramme angepasst, implementiert werden. *Inline* in Connector-Modulen implementierte *Refinements* haben hierbei die höchste Priorität.
Um die nötige Flexibilität zu bieten, existiert zusätzlich das Instrument spezieller *Refining Classes*. Diese werden an jeweils einen *Hook* und eine Basisklasse gebunden.
Zur Laufzeit werden solche Klassen von dem System automatisch herangezogen, wenn der jeweilige Aspekt auf einer Basisklasse des gegebenen Typs arbeitet.
Dabei wird jeweils die erste zu dem dynamischen Typ der Basis passende *Refining Class* für den jeweiligen Hook herangezogen. Dies ist in gewisser Hinsicht mit dem *Smart Lifting* in *ObjectTeams* vergleichbar (siehe 2.3.1.4). Anders als dieser Mechanismus wird aber nicht die am besten, sondern die erste passende Implementierung herangezogen.

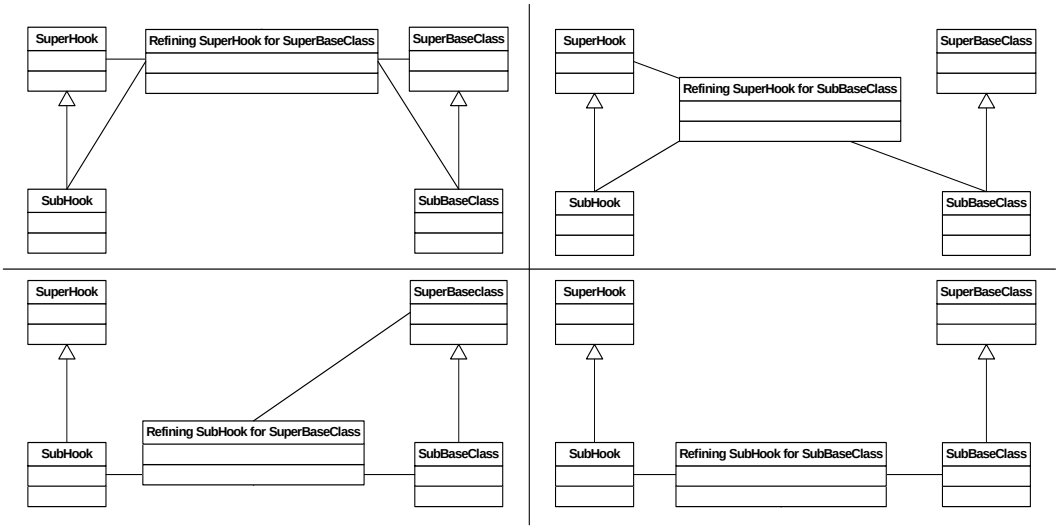

Abbildung 2.1: Auswahl von Refinement Classes in Abhängigkeit von *Hook* und Basistyp. Im Konfliktfall wird die erste passende *Refining Class* herangezogen.

Eine *Refining Class* verlangt nicht die Implementierung aller `refinable` Methoden eines *Hooks*. Die Implementationen können auf mehrere Refinements aufgeteilt werden.

2.2.1.7 Dynamische Bedingungen: `isApplicable`

Um dynamische Bedingungen in *Pointcuts* mit einzubeziehen, enthält JAsCo das besondere `isApplicable()` Konstrukt. Anders als der *if-pointcut* von AspectJ erlaubt *isApplicable* es, dynamische Bedingungen getrennt von lexikalischen Bedingungen zu formulieren. Grundsätzlich verhält das Konstrukt sich wie eine *Hookmethode* und erlaubt jeglichen Java-Code mit einem `boolean` als Rückgabewert.

Eine Besonderheit von *isApplicable* ist es, dass alle an einem gegebenen *Joinpoint* relevanten *isApplicable* Ausdrücke ausgewertet werden. Erst dann werden *Advicecode* und die Basismethode ausgeführt. Das ist in dem JAsCo Ansatz zur Verwaltung der gewobenen *Hooks*begründet. Der Ausdruck arbeitet logisch gesehen auf dem Zustand des Programms bei dem Erreichen des *Joinpoints*.

2.2.1.8 Rollen: Virtual Mixins

JAsCo bietet *Introductions* in Form so genannter *virtual Mixins*[10], mit denen Implementierungen von Java-*Interfaces* in *Hooks* an Basisobjekte gebunden werden können. Der Begriff „*Mixin*“ ist dabei vorsichtig zu behandeln, da es sich eher um Rollenobjekte, denn *Mixins* oder *Introductions* handelt. Ein *pseudo-Cast*[11] ist nötig, um die Funktionen zu nutzen.
Ein solcher *Cast* auf den eingeführten Typ ist dabei aber nur innerhalb von JAsCo Sprachkonstrukten möglich; ausserhalb dieser ist der Umweg über JAsCos *Reflection* Methoden erforderlich. Die Nützlichkeit dieser Funktion wird durch ihre Isoliertheit gemindert; so lassen sich z.B. *Virtual Mixins* nicht zur Einführung von reinen Marker-Interfaces nutzen, z.B. für *Refinement Classes*. Innerhalb enger Grenzen eignen sich die *JAsCo Mixins* aber zur Modellierung globaler Rollen und zur Kommunikation von Aspekten untereinander.
Konkrete *Virtual Mixins* werden als *Hooks* implementiert. Der Einsatz erfolgt genau wie der eines jeden anderen Hooks über den Aufruf eines Hookkonstruktors innerhalb eines *Connectors*. In der Tat unterscheiden sich Mixin-implementierende nur durch das Implementieren eines *Mixin-Interfaces* von anderen *Hooks*.
Eine tiefgreifende Einschränkung besteht allerdings darin, dass die Zuordnung der Basisinstanz zu dem Mixin-Objekt permanent und global ist: Ein Basisobjekt kann nur mit maximal einer Instanz eines gegebenen *Mixin-Typs* dekoriert werden – selbst wenn mehrere *Hooks* dieses Interface als *Mixin* anbieten.

2.2.1.9 Konfliktbeseitigung: Combination Strategies

Combination Strategies sind – erwartungskonform – eine Anwendung des Strategy Patterns ([GHJV95]), um bekannte Konflikte zwischen innerhalb eines *Connectors* instantierter Aspekte aufzulösen. Zu diesem Zweck können *Combination Strategies* in *Connector*-Modulen angegeben werden. Diese werden bei dem Erreichen eines gebundenen *Joinpoints* ausgeführt und erhalten Zugriff auf die *Hooklist*, der Liste aller gewobenen Hooks für den jeweiligen Joinpoint.
Dabei kann gesteuert werden, ob dies bei jedem Erreichen eines *Joinpoints* oder aus Performanzgründen nur einmal geschehen soll. *Combination Strategies* werden vor eventuellen `isApplicable` Ausdrücken ausgeführt. Eine Verwendung einzelner *Combination Strategy* Instanzen über mehrere *Connectors* hinweg ist nicht vorgesehen.

[10] Die Namenswahl ist irreführend, da sich der Mechanismus von *Mixins* im etablierten Sinne unterscheidet.
[11] Syntax eines *Casts*, aber mit einer gänzlich anderen Semantik, da das zurückgegebene Objekt nicht mit dem eingegebenen identisch ist.

2.2.1.10 Reflections und Introspections

Die API der Laufzeitumgebung bietet die in der AOP-Welt mittlerweile als Standard zu betrachtenden Möglichkeiten. Über das Schlüsselwort `thisJoinPoint` sind innerhalb von *Advices*und *Refinements* Informationen über den Aufrufkontext verfügbar, einschließlich der Parameter des zugrundeliegenden Joinpoints. Ferner ist über `thisJoinPointObject` der Zugriff auf das aktuelle Basisobjekt möglich.

Wie bei *Reflection*-APIs üblich, sind die zur Verfügung gestellten Methoden nicht statisch getypt. Ihre Verwendung birgt im Zusammenspiel mit unpassenden Connector-Argumenten ein beträchtliches Potenzial für Laufzeitfehler.

2.2.1.11 Jumping Aspects: Stateful Aspects

Als Ansatz zur Lösung für das Problem der *Jumping Aspects* (siehe [BMdV00]) wurde JAsCo um so genannte *stateful Aspects* nach [DFS04] erweitert. Diese ermöglichen das Verhalten von *Hooks*, sowohl bezüglich *Pointcuts* wie auch *Advice*, an Zustände zu knüpfen.
Dies erfolgt über endliche Automaten: Abstrakte *Pointcuts* dienen als Transitionen; an das Erreichen von Zuständen können *Advices* geknüpft werden. So können – innerhalb enger Grenzen – Protokolle überwacht oder erweitert werden ([VSCF05]).
Eine Besonderheit ist die Möglichkeit `isApplicable` Bedingungen an einzelne Zustände zu knüpfen. Diese beeinflussen dann die Ausführung der gewobenen *Advices*, nicht aber eventuelle Zustandswechsel.

Eine bemerkenswerte Einschränkung ist, dass der aktuelle Zustand nicht einfach programmatisch erfasst werden kann. Werden zusätzlich zustandsbezogene `isApplicable` Ausdrücke verwendet, so kann das Verhalten gänzlich unkontrollierbar werden.

2.2.1.12 Dynamische Aspekte

Gaben die bisherigen Beispiele einen Eindruck der eher statischen Möglichkeiten, so sollte doch nicht unerwähnt bleiben, dass JAsCo sehr wohl über vollständig dynamische Elemente verfügt. Zwar sind die bereits beschriebenen Merkmale wie `isApplicable` Methoden, *Combination Strategies*, *stateful Aspects* und sogar `cflow` Pointcuts durchaus dynamisch in ihrer Natur, aber ihre Wirkung ist nicht dafür vorgesehen, programmatisch gesteuert zu werden.
Es existiert darüber hinaus die Möglichkeit, zur Laufzeit Aspekte zu ergänzen/zu entfernen.
Wird bei einem *Connector* auf das `static` Schlüsselwort verzichtet, so wird er nicht automatisch von der Laufzeitumgebung verarbeitet. Es ist aber dann möglich, diesen *Connector* gezielt auf einzelne Instanzen anzuwenden. So können Aspekte punktgenau für einzelne Objekte gewoben werden.
Unabhängig von der Art ihrer Deklaration, können *Connectors* dynamisch aktiviert bzw. deaktiviert werden. Dies wirkt sich auf alle in dem jeweiligen *Connector* deklarierten

Hooks aus, wobei *Connectors* immer als *Singleton*s auftreten; die Auswirkungen sind daher immer global und *threadübergreifend*.

2.2.1.13 Adaptive Programming

Als Beweis [VSV+05] der Anwendbarkeit des JAsCo Conncetor-Konzeptes auf *adaptive Programming* ist eine weitgehend in JAsCo integrierte Schnittstelle zu DJ gedacht. Eine Einführung in die adaptive Programmierung würde den Horizont dieses Abschnittes übersteigen. Daher soll eine vereinfachte Darstellung genügen: Nach dem *Gesetz von Demeter* sollte eine Klasse nur auf unmittelbar referenzierte Objekte zugreifen. Dies ist jedoch nur schwer mit den Erfordernissen heutiger Datenstrukturen vereinbar, weshalb eine Abkehr von der expliziten Verwendung struktureller Informationen durch Klassen als erstrebenswert angesehen wird.
Adaptive Programming bietet die Möglichkeiten auf, im Referenzgraphen weit von der aufrufenden Klasse entfernt liegende, Daten zuzugreifen.

Dafür werden sogenannte *adaptive Vistors* eingeführt, die den Referenzgraphen traversieren, um die nötigen Operationen unter Einhaltung des *Gesetzes von Demeter* ausführen zu können.
In JAsCo finden sich zur Umsetzung dieses Konzeptes sogenannte *traversal connectors*, in welchen normale *Hooks* mit `visiting` Pointcuts instantiiert werden. Das entscheidende neue Merkmal in jenen Dateien ist die Angabe einer *Traversal Strategy*, welche einen Pfad durch den Referenzgraphen beschreibt. Wenn aktiviert, traversiert der Connector den Referenzgraphen entlang dieses Pfades und führt dabei den *Advice* der zuvor erzeugten *Hooks* aus.

2.2.1.14 Verteilte Aspekte: Distributed JAsCo

JAsCo bietet die Möglichkeit Joinpoints, in auf entfernten Virtual Machines laufenden Programmen, zu berücksichtigen und Advice auf solchen auszuführen. Dabei können sowohl *Pointcuts*, als auch *Advice* entfernt sein: Ausführung eines *Advices* als Reaktion auf das Erreichen eines *Joinpoints* in einem anderen Laufzeitkontext ist ebenso möglich[12] wie das Ausführen von *Advice* auf einer entfernten *virtual Machine* als Reaktion auf das Erreichen eines lokalen *Joinpoints*.
Dies unterscheidet sich von verteilten Java Anwendungen, etwa mittels *RMI*. Anders als bei RMI, wo entfernte Methodenaufrufe den Programmfluss wirklich verteilen, ist die Verteiltheit bei JAsCo über Kommunikation von JAsCo-Laufzeitumgebungen verwirklicht[13]. Daher erübrigen sich zwar Konstrukte, wie *Stubs* und *Skeletons*, jedoch fehlt aber auch eine typsichere Umgebung.

2.2.1.15 Weaver

Wie bereits angedeutet hat Performanz bei der Entwicklung von JAsCo einen sehr hohen Stellenwert. Für JAsCo existiert ein Hotswap/Javassist basierender *Runtime Wea-*

[12] Dies trifft nicht uneingeschränkt auf die vorliegende JAsCo Version 0.8.7 zu.
[13] Über *RMI* und *Reflections*.

ver([VS04b]), der dynamisch optimierten Aspektcode direkt in die Klassen einfügt. Bei manchen Aspekten führt dies zu einer hervorragenden Performanz, die AspectJ in vielen Belangen gleichkommt. Unterstützt wird der *Weaver* von einem System zum *Caching* von Ergebnissen und Kompilaten (JUTTA). Für die dynamischen Aspekte JAsCos fehlt momentan noch eine vergleichbare Untersuchung. Es ist von einem Ergebnis im normalen Rahmen dynamischer AOP-Ansätze auszugehen[14].

2.2.2 Verwandte Ansätze

JAsCo ist dicht an AspectJ angelehnt und kann als Anwendung von Aspectual Components auf AspectJ gesehen werden. Ähnliche Strategien wurden bei JAC, AspectWerkz und Caesar verfolgt. Insbesondere AspectWerkz ist JAsCo in seiner Ausrichtung auf AspectJ ähnlich, wobei es sich im Syntax weiter von AspectJ entfernt, in der Semantik aber näher an diesem bleibt. Mittlerweile ist das AspectWerkz Projekt AspectJ beigetreten.

Es existieren JAsCo Implementierungen für J2ME und .net, die jedoch weder an Reife noch an Funktionsumfang der untersuchten Java Version nahekommen.

2.2.2.1 JAC

JAC[15] ([PDF+02]) war einer der ersten POJO-AOP Ansätze, also jener AOP-Lösungen, die Aspekte in *Plain Old Java Objects* ermöglichen. JAC verwendet nicht XML Konfigurationsdateien oder Annotations[16], um die besondere Semantik anzudeuten, sondern lediglich die Mittel eines Frameworks im Zusammenspiel mit einer erweiterten Laufzeitumgebung. Dabei wird JAC von seinen Autoren als *Middleware* geführt und verfügt über innovative Ansätze für verteilte Aspekte. Darüber hinaus existiert für JAC eine Bibliothek mit generischen Aspekten, die viele Funktionen abdeckt und ihrerseits von der JAC Laufzeitumgebung verwendet wird.
Verantwortlich zeigt sich die Firma AOPSYS[17], in Zusammenarbeit mit den französischen Forschungseinrichtungen *LIP6*, *CEDRIC*, und *LIFL*.
Auch JAC verwendet zwei grundlegende Module: `AspectComponent` und `Wrapper`. Zugehörigkeit zu diesen entsteht jeweils durch Erweiterung entsprechender Klassen des JAC Frameworks. Grundsätzlich bieten `AspectComponents` Funktionen, um das *Wann und Wo* zu definieren, während die `Wrapper` Klassen das *Was* beantworten. Wie auch *AspectBeans* bei JAsCo, treten AspectComponents als *Singletons* auf.
JAC erlaubt die Konfiguration der Aspekte in dedizierten Konfigurationsdateien, die in einer deklarativen Sprache verfasst werden. Prinzipiell entsprechen Kommandos in jener Konfigurationsprache Methodenaufrufen auf dem jeweiligen `AspectComponent`. Dies ist sehr mit den *Connector* Dateien in JAsCo vergleichbar; auch dort ist die Java-

[14] Ein etwa um zwei Größenordnungen höherer *Overhead*.
[15] `http://jac.objectweb.org/`
[16] XML-Konfigurationsdateien sind optional möglich; Java5 und damit *Annotations* ist als Entwicklung zu jung, um in JAC Unterstützung zu finden.
[17] `http://www.aopsys.com`

Unterstützung auf Bean- und Hookmethoden beschränkt.

JAC bildet ferner den Grundstein für die *AOP-Alliance* ([Paw03]), einen Standard für AOP-Unterstützung in *Middleware*. Die Entwicklung von JAC ruht seit einiger Zeit, weshalb das System nicht mehr auf dem Stand der Technik ist und mit neueren Java-Erweiterungen nicht umgehen kann.

2.2.2.2 FUSE/J

Nicht unerwähnt bleiben sollte das Nachfolgeprojekt[SVWJ04] von JAsCo, *FuseJ*[18]. Wie JAsCo wird dieses am *System and Software Engineering Lab* der Vrije Universiteit Brussel entwickelt. FUSE/J stellt eine Abkehr von der geteilten Basis/Aspekt Welt dar und bietet AOP auf Basis von JavaBean-Komponenten.
Anstelle neuer Sprachkonstrukte, erfolgt die Umleitung des Kontrollflusses hier über deklarierte *Joinpoints* in Konfigurationsdateien. Dies beinhaltet den Verzicht auf jegliche neuen Sprachmerkmale zur ausschließlichen Beschreibung von *Advice*. Vielmehr verfügt jede Komponente über ein *Gate* Interface, über welches sowohl direkte, als auch aspektorientierte Komposition ermöglicht wird.
Durch die Forderung nach neuen, expliziten Schnittstellen wird die Verwendung von FUSE mit bestehenden Programmen erschwert. Die Zielsetzung und die in Kauf genommenen Einschränkungen unterscheiden daher diesen Ansatz von der klassischen Aspektorientierung.
Denn bei FUSE soll die Unterscheidung zwischen Aspekten und Komponenten vollständig verschwinden ([SFV05]), ohne die aspektorientierten Teile lediglich in externe Konfigurationsdateien zu verlagern, wie es etwa bei *AspectWerkz*[19] der Fall war.

2.3 Object Teams

ObjectTeams[20] ist ein im Rahmen des TOPPrax[21] Projektes entwickeltes Konzept, dessen Mittelpunkt die Programmiersprache ObjectTeams/Java darstellt. Im Weiteren werden „ObjectTeams“ und „ObjectTeams/Java“ synonym verwendet werden.
ObjectTeams ist ebenso ein direkter Abkömmling der *Aspectual Components*, die verfolgte Zielsetzung unterscheidet sich aber deutlich von JasCo.

ObjectTeams ist eine Antwort auf mehrere Schwachpunkte des rein objektorientierten Modulkonzeptes. So sind Klassen einerseits zu groß, da sie in der überwiegenden Zahl der Fälle Aufgaben von Klassen mit wahrnehmen, für die eigentlich ein eigenes, weiteres Modul erstrebenswert wäre. Das ist, als *Scattering & Tangling* oder *crosscutting concerns* bezeichnet, unter den schwergewichtigsten Argumenten für die Aspektorientierung.
Andererseits sind Klassen aber auch zu klein. Diese Beobachtung lässt sich im Kontext

[18] `http://ssel.vub.ac.be/fusej/`
[19] Das AspectWerkz Projekt ging Anfang 2005 in AspectJ auf. `http://aspectwerkz.codehaus.org/`
[20] `http://www.objectteams.org`
[21] `http://www.topprax.de/`

von Softwareevolution machen: Funktionalität wird nicht nur von einer einzigen Klasse implementiert[22], sondern entsteht durch die Zusammenarbeit verschiedener Klassen. Diese Zusammenarbeit aber verhindert die isolierte Erweiterung einzelner Klassen. Die gesamte Kollaboration muss im Regelfall angepasst werden, um der Erweiterung einzelner Teilnehmer Rechnung zu tragen.
ObjectTeams führt die *Teamvererbung* (siehe: [Her02b]) als Lösung für dieses Problem an: Ganze Kollaborationen – Teams – können kovariant erweitert werden.
ObjectTeams verspricht damit *„Teamgeist für Objekte“*. Der grundlegende Gedanke liegt in der Zusammenarbeit von (Rollen-) Objekten in *Collaborations*.
Crosscutting Concerns können so auf einem stabilen, typsicheren Boden in kooperierenden Rollen modularisiert werden, die in einem geschlossenen Kontext auftreten. ObjectTeams ist als Erweiterung für mehrere Programmiersprachen implementiert, für diesen Text wird die Java Version dienen, deren Entwicklung sehr weit fortgeschritten ist.

2.3.1 Ansatz

2.3.1.1 Teams und Rollen

Die beiden zentralen neuen Entitäten in *ObjectTeams* sind *Teams* und *Rollen*, respektive Teamklassen und Rollenklassen. Bei Teamklassen handelt es sich um eine neue Art von Klassen[23]. Rollenklassen sind *Inner Classes* von Teamklassen; sie können *gebunden* oder *ungebunden* auftreten.

Rollen können ihrerseits Teams sein, solche Schachtelungen sind ein grundlegendes Designmittel.
Rollen sind üblicherweise durch die umschließende Teaminstanz eingeschlossen, können aber *externalisiert*, d.h. ausserhalb ihres Teams referenziert und verwendet werden. Solche externalisierten Rollen sind für sich genommen nur auf Basis ihrer einschließenden Teaminstanz typisiert; Rollenklassen unterschiedlicher Teaminstanzen sind niemals kompatibel zueinander[24]. Über Schnittstellen oder gemeinsame Supertypen können sie aber ohne Kenntnis des Teams angesprochen werden.

2.3.1.2 Bindungen

Der Anspruch dieser Kombination *aspektorientiert* zu sein, kommt durch die Möglichkeit, Rollenklassen an andere Klassen zu binden, zum Vorschein: Werden Rollenklassen mit dem Schlüsselwort `playedBy` an andere Klassen gebunden, so dienen Instanzen der jeweiligen Rollenklasse als Rollenobjekte für Instanzen der gebundenen Klasse, im weiteren

[22] Hier ist ein feiner Unterschied zu *Scattering* hervorzuheben: Scattering bezeichnet die Verteilung einer Funktionalität über viele Klassen; dies ist zwar auch hier der Fall, aber im Sinne einer sauberen Aufgabenteilung auf verschiedene Teilnehmer.
[23] Teamklassen stehen in einer eigenen Typhierarchie; ihr generischer Supertyp ist `Team`, anstelle von `Object`
[24] Die Möglichkeit eines *Widenings* zu einer herkömmlichen Java-Klasse wird hiervon (noch) nicht tangiert.

Basisklasse genannt. Als Basisklassen kommen dabei alle regulären Klassen, inklusive anderer Teams und Rollen in Frage[25].
Diese Rolle-Basis Beziehung hat eine strikte 1:1 Multiplizität: eine Instanz einer gebundenen Rollenklasse hat innerhalb einer Teaminstanz genau ein Basisobjekt[26].

Der besondere Aspekt dieser Beziehung sind Methodenbindungen: Methoden der Basisklasse können als Auslöser für *callin* Methoden der Rollenklasse gebunden werden. Aufrufe der so gebundenen Methode auf einem Basisobjekt führen dann zur Ausführung der gebundenen *callin*-Methode auf einem Rollenobjekt. Dies kann vor, nach oder statt der Ausführung der ursprünglichen Basismethode passieren (`before`, `after`, `replace`). Der Operator hierfür ist „`<-`“, also „`<Rollenmethode> <- before/ after/ replace <Basismethode>`“, wobei `replace` lediglich für als „`callin`“ markierte Rollenmethoden zulässig ist – solche Methoden können und sollten über das Pseudoobjekt „`base`“ die jeweilige Basismethode aufrufen.
Jedenfalls stellt ObjectTeams sicher, dass die callin-Methode genau auf dem an die Basisinstanz gebundenem Rollenobjekt ausgeführt wird. In umgekehrter Richtung können Methoden und Felder der Basis über *Callouts* als Methoden der Rollenklasse zur Verfügung gestellt werden. Auch diese Methodenaufrufe folgen streng der Basis-Rollen Bindung und sind die einzige Möglichkeit, auf das Basisobjekt zuzugreifen. Der Operator hierfür ist genau umgekehrt, also “`<abstrakte Rollenmethode> -> <Basismethode>`”.
Um solche Methodenbindungen flexibel gestalten zu können, bietet ObjectTeams *Parametermappings*. Diese ermöglichen es, bei der Kommunikation mit der Basis die Parameter der Rollenmethoden an die der Basis anzupassen. Dies gilt für Bindungen in beiden Richtungen und ermöglicht die Anpassung der Parameter in Art und Reihenfolge, eingeschränkt auch in der Zahl.

2.3.1.3 Lifting und Lowering

Dabei sollte auch die Herstellung besagter Bindung nicht verschwiegen werden. Der dahinter stehende Mechanismus trägt den Namen *Lifting* und beschreibt den Prozess der dynamischen Zuordnung oder Erzeugung einer Rolle; schlicht den Weg von dem Basisobjekt zum Rollenobjekt. Dies findet üblicherweise implizit an den Übergängen von der Basis zum Team statt, namentlich den *Callin*-Methoden.
Darüber hinaus besteht aber auch die Möglichkeit, über *declared Lifting* Parameter von Teammethoden explizit zu liften. Dies wird durch das Schlüsselwort „`as`“ gesteuert.
Die Umkehrung dieses Vorgangs ist das *Lowering*, was aufgrund der festen und eindeutigen Rolle-Basis Beziehung einen ungleich einfacheren Vorgang darstellt: Eine Rolle kann nur ein Basisobjekt haben, ein komplexer Algorithmus entfällt. Lowering findet beispielsweise statt, wenn eine Situation, etwa ein Methodenaufruf, eine Instanz der Basisklasse erwartet, aber eine Rolle übergeben wurde.

[25] Technisch begründet gibt es aber einige Ausnahmen.
[26] Einem Basisobjekt können aber verschiedenen Rollen unterschiedlicher Typen zugeordnet sein.

2.3.1.4 Smart Lifting

Die Auswahl der bestpassendsten Rolleninstanz, beziehungsweise der richtigen Rollenklasse zur Erzeugung einer Rolleninstanz, basiert auf dem *Smart Lifting* ([HHM04b]) Algorithmus. Durch diesen wird anhand des dynamischen Typs des Basisobjekts die an der Stelle des Liftings bestpassende Rollenklasse auswählt, um dann von dieser eine Rolleninstanz zurückzuliefern. Entweder wird dabei eine passende Rolle erzeugt, oder, sollte bereits eine konforme an das Basisobjekt gebundene Rolleninstanz in dem Team existieren, eine bestehende Rolle nachgeschlagen.
Dies ist essentiell, um bei dem Vorhandensein von Vererbungshierarchien – auf Rollen- wie auf Basisseite – einen flexiblen und dennoch eindeutigen Liftingmechanismus zu ermöglichen.

2.3.1.5 Vererbung

Wurden Rollenklassen soeben noch als eine spezielle Form von *Java Inner Classes* eingeführt, so sei hier auf einen grundlegenden Unterschied hingewiesen. ObjectTeams bietet eine Erweiterung der objektorientierten Polymorphie von Java. Nicht nur einzelne Klassen können zueinander konform sein, sondern auch Gruppen - *Teams* - von Klassen. Dafür existiert eine neuartige Beziehung zwischen Rollenklassen von in einer Vererbungsbeziehung stehenden Teamklassen: *implizite Vererbung*.
Durch diese stehen Rollenklassen eines Subteams in einer impliziten Subtypbeziehung zu gleichnamigen Rollenklassen des Superteams und erben jeweils alle Features[27].

Ein Subteam *„erbt“* somit alle Rollen seines Superteams und ObjectTeams erlaubt deren *kovariante* Redefinition – Teams sind damit ein Modulkonzept mit Modulen, die über Klassen hinausgehen; eine konsistente Behandlung der Vererbung von Klassen und Methoden ist somit vorhanden.
Ein Subteam hat immer eine echte Subtypbeziehung zu seinem Superteam inne. Um diese Kompatibilität zu erhalten, gelten weitgehend die Java Regeln bezüglich Redefinition geerbter Felder und Methoden in *normalen* Klassen. Sie wurden aber erweitert, um den eingeführten Konzepten Rechnung zu tragen.

ObjectTeams erlaubt zusätzlich zu der impliziten Vererbung im Zusammenspiel mit der Vererbung von Teamklassen, auch die „klassische“, explizite Vererbung von Rollenklassen innerhalb eines Teams mittels des Schlüsselwortes `extends`. Die als Supertyp dienende Klasse darf entweder eine herkömmliche Java-Klasse oder eine Rollenklasse desselben Teams sein. Im letzteren Falle sind auch implizit geerbte Rollen erlaubt.

2.3.1.6 Dynamik

ObjectTeams ist ein dynamischer AOP Ansatz. Die callin-Bindungen, welche das zentrale Merkmal von ObjectTeams als AOP-Ansatz darstellen, beeinflussen den Programmablauf

[27] Es sei hier bereits erwähnt, dass implizit geerbte Rollenklassen **nicht** in einer Subtypbeziehung zu äquivalenten Rollenklassen des Superteams stehen.

nur, wenn eine Instanz eines Teams mit gebundenen Rollen *aktiv* ist. Ein Team kann explizit durch `within` Blöcke oder durch den Aufruf der `activate` Team-Methode aktiviert werden.
Within Blöcke sind ein Sprachmerkmal von ObjectTeams, welches den Aktivierungszustand eines bestimmten Teams garantiert, während Code innerhalb eines solchen Blockes ausgeführt wird. Darüber hinaus wird sichergestellt, dass das Team nach dem Verlassen des Blocks den gleichen Aktivierungszustand hat wie vor dem Eintreten in den Block. Dies kann in nebenläufigen Programmen zu einer Reihe unvorhergesehener Zustände führen, weshalb dieses Konstrukt in solchen nur sehr gezielt eingesetzt werden sollte.

Die häufiger anzuwendende Art der (De-)Aktivierung sind daher die globalen `activate` und `deactivate` Methoden der *Reflection-API*.

```
...
Team myTeam = new MyTeam();

// Team ist die generische Superklasse aller Teams
// Initial sind Teams deaktiviert

within{myTeam}{

  //Innerhalb des Blocks ist myTeam aktiv

  ...
}

//Nach Verlassen ist myTeam inaktiv
//Dabei ist es irrelevant, welche Operationen während
//des Aufenthalts in dem Block aufgerufen wurden.

...
```

Listing 2.3: ObjectTeams: Aktivierung über das *Within* Block Statement

Aktivierung kann auch implizit stattfinden. Während eine Rollen- oder Teammethode ausgeführt wird, ist das einschließende Team immer aktiv. Auch dies kann in nebenläufigen Programmen zu erheblichen Seiteneffekten führen, welche aber mit den Java-Synchronisierungsmethoden beherrscht werden können.

2.3.1.7 Guard Predicates

Die Dynamik von ObjectTeams ([HH] §5.4.) geht über die globale Aktivierung/Deaktivierung hinaus. Die Aktivität der Bindungen eines Teams hängt nicht nur von dem globalen Merkmal der Teamaktivierung ab. Zusätzlich können flexiblere *Guard Predicates* ([HHMW05]), boolsche Ausdrücke, für Teams, Rollen, callin Methoden und Bindungen platziert werden.

Guard Predicates auf Teamebene beeinflussen dabei die Aktivierung der Teaminstanz im Ganzen, wie es auch bei `activate`/`deactivate` Methoden der Fall ist. Sie sind aber abhängig vom Wahrheitswert des *Guard Predicate* Ausdrucks bei dem Erreichen der fraglichen Bindung.
Feiner sind *Predicates* auf Rollenebene. Diese regulieren die Aktivität aller Bindungen der verzierten Rollenklasse. Prinzipiell Analoges gilt für die *Predicates* auf Methoden und Bindungsebene: *Predicates* auf Methodenebene wirken auf alle Bindungen für die gegebene Methode; *Predicates* auf Bindungsebene nur auf die jeweilige Bindung. *Guard Predicates* unterhalb der Teamebene können in einer mächtigeren Form auftreten: `base`-*guards* beziehen sich nicht auf den Zustand des Teams bzw. der Rolleninstanz, sondern auf den Zustand des Basisobjektes. Sie sind ungleich mächtiger als *Guard Predicates* normaler Natur, da eine *Base Guard* auch die Erzeugung einer neuen Rolleninstanz verhindern kann. *Base Guards* werden vor dem Lifting bei dem Erreichen einer gebundenen Methode überprüft, andere *Predicates* erst nach dem Lifting – wobei dieses Lifting die Erzeugung der Rolleninstanz auslösen kann, wenn zu diesem Zeitpunkt noch keine solche existiert.

2.3.1.8 Decapsulation

Als Umkehrung der *Encapsulation* bietet ObjectTeams *Decapsulation* ([Her04c] und §4.6 in [HH]). Damit ist es möglich, die Zugangsbeschränkungen der Basisklasse zu umgehen und somit auch auf nichtöffentliche Felder und Methoden zuzugreifen, sowohl für *callin* als auch für *callout* Bindungen. Dies ähnelt dem Konzept privilegierter Aspekte aus *AspectJ*. Ein wichtiger Unterschied hierbei ist, dass AspectJ per *default* private Methoden als *Joinpoints* offenlegt, ohne privilegiert zu sein.

2.3.1.9 Confined und Opaque Roles

Die genau umgekehrte Richtung nimmt das Konzept opaker Rollen ([Her04b]). Die bereits angesprochene Möglichkeit, Rollen zu externalisieren, wird dadurch erweitert, dass der Zugriff auf solche externalisierten Rollen eingeschränkt wird. *Opaque Roles* können externalisiert werden. Sie erlauben aber keinerlei Zugriff auf ihre *Mitglieder*, nicht einmal über ein *Widening* zu `Object`. *Confined Roles* sind noch stärker eingeschränkt: sie können schlicht nicht externalisiert werden.

2.3.1.10 UML-Erweiterung: UFA

Aus dem Kontext von *ObjectTeams* stammt eine UML-Erweiterung: *UML for Aspects*; *UFA*. Für eine Einführung in diese Erweiterung sei auf [Her02a] verwiesen. UFA erweitert UML Klassendiagramme um eine – Teams sehr ähnliche – Art von *packages*. Diese sind als „*first Class Citizens*“ eine Mischung aus package und Klasse: Sie können sowohl Klassen – Rollen – als auch Klassenmitglieder – Methoden und Attribute – enthalten. Ferner findet sich eine Notation für *callin* und *callout* Methoden („←“, bzw. „→“) und die `adapt` Beziehung, welche die adaptierte Domäne kennzeichnet und so die Herkunft der Basisklassen verdeutlicht. Abbildung 2.2 zeigt eine einfache Teamhierarchie in der UFA

Darstellung.

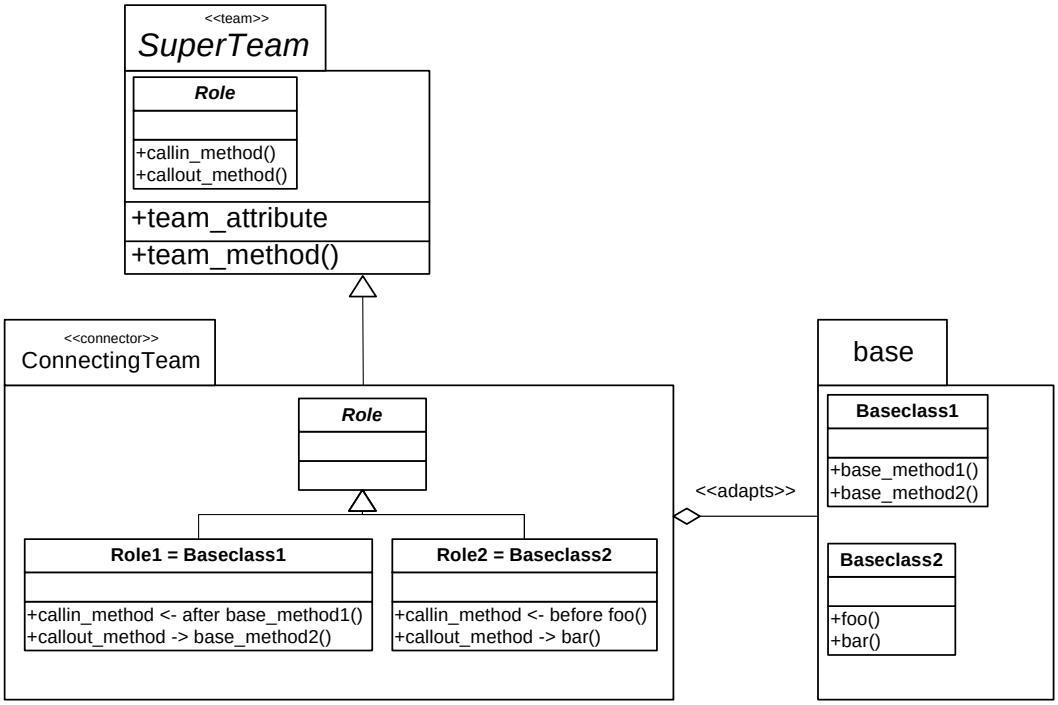

Abbildung 2.2: UFA-Darstellung von Teams. Die Gleichsetzung symbolisiert die „playedBy“ Relation, die im weiteren auch als UML-Abhängigkeit mit dem Stereotyp ≪playedBy≫ dargestellt werden wird.

2.3.2 Verwandte Ansätze

2.3.2.1 CAESAR

Caesar/Java ist eine stellenweise zu ObjectTeams sehr ähnliche Technik, die an der TU Darmstadt ([AGMO06], [MO03]) entwickelt wird. CAESAR bietet, wie ObjectTeams, ein Konzept für Module „größer als Klassen“ an. So verwendet Caesar als Teilnehmer in Kollaborationen *virtual Classes*, welche den Rollen von ObjectTeams ähnlich sind. Ein entscheidender Punkt von CAESAR ist die über Mixins verwirklichte parametrisierte Mehrfachvererbung. Diese erlaubt eine feingranulare Spezialisierung von Klassen.
Im Unterschied zu ObjectTeams verfügt CAESAR über ein, an AspectJ angelehntes, *Pointcutmodell*, threadlokale Aspekte, sowie Unterstützung für verteilte Anwendungen.

2.4 Ähnlichkeiten, Gemeinsamkeiten und Unterschiede

Wie eingangs geschildert, beruhen beide Techniken auf Anwendungen des Konzeptes „Aspectual Components“, zeigen aber erhebliche Unterschiede in Art und Ziel der Umset-

zung.
JAsCo spezialisiert sich in Richtung der *Komponentenbasierenden Softwareentwicklung* durch Einführung neuer Verbindungsmodule. Diese sollen eine weitreichende Abkopplung einzelner Aspektkomponenten sowohl von der Basis, als auch von anderen Aspekten bieten. Der Basisbezug wird dabei in beiden Richtungen schwach gehalten, Ursprung und Ziel von Nachrichten sollen für konkreten Aspektcode abstrakt gehalten werden.
ObjectTeams/Java verwendet aspektorientierte Methoden in einem objektorientierten

Aspectual Components		**JAsCo**	**ObjectTeams**
Aspectual Component		*AspectBean*	*Team*
Participant Graph	**Participant**	`Hook`	Rolle
	`expect`	`refinable` *Methods*	abstrakte Methoden für *Callouts*
	`replace`	*Advice*	Rollenmethoden für *Callins*
Definitionen	**lokale Klassen**	*inner Classes*	ungebundene Rollen
	Features	*AspectBean*-Mitglieder	Teammitglieder
Connector		`Connector` und *Refining Classes*	erbende Verbindungsteams

Tabelle 2.1: Apectual Components - vergleichbare Konstrukte

Kontext. Der Komponentenbegriff dient hier zur Beschreibung der Modularisierung von aufgabenbezogenen Kollaborationen innerhalb eines eigenen Kontextes. Er bezeichnet eher einen *white-Box* Ansatz, wobei beide Ansätze als Mittelweg zwischen *white-Box* und *black-Box* bezeichnet werden können.
Man kann also von zwei verschiedenen Philosophien der Erweiterung sprechen: Über definierte Schnittstellen für Konfigurationsdateien erfolgt Erweiterung bei JAsCo. Die Möglichkeiten zur Vererbung sind hier nicht immer für eine Erweiterung geeignet.

Über die Mächtigkeit der Vererbung erlaubt ObjectTeams Erweiterung: Einzelne Teilnehmer können abstrakt gehalten werden, bei der Redefintion in einem konkreten Umfeld bietet die Teamvererbung Konformität der bestehenden, kovariant geerbten, Bindungen und Abhängigkeiten.
Auch im praktischen Einsatz agieren *Hooks* und *AspectBeans* anders als *Rollen* und *Teams*. Sind Rollen immer auf ein Basisobjekt bezogen und meist hinter der Fassade der Teammethoden verborgen, so sind *Hooks* potentiell der Eintrittspunkt für eine Vielzahl von Basisobjekten unterschiedlicher Klassen. Dabei muss auch der gravierende Unterschied zwischen der Rollen-Instantiierung durch *Lifting* und der halbautomatischen Hook-Instantiierung durch *Factories* bedacht werden.

3 Wiederverwendung von Aspekten

Dieser Abschnitt verfolgt zwei Ziele: Erstens soll er Überlegungen und bestehende Erfolge bezüglich der Wiederverwendung von Aspekten zur Diskussion stellen. Zweitens wird die Domänenanalyse, genauer die *FODA* (*Feature-Oriented Domain Analysis*), als paradigmenunabhängige Methode zur vorbereitenden Analyse für den Entwurf wiederverwendbarer Software vorgestellt, die abgewandelt auch für Aspekte geeignet scheint. Dabei sind allerdings deutliche Einschränkungen in Umfang und Anspruch erforderlich.

3.1 Definition

Wiederverwendung und *Wiederverwendbarkeit* sind oft intuitiv verwendete Begriffe, um eine Übertragbarkeit von Programmen und insbesondere Quelltexten zu beschreiben. Genauere Definitionen finden sich z.B bei Charles Krueger in [Kru92], wo Wiederverwendung (*reuse*) definiert wird als:

> "[...]software reuse is using existing software artifacts during the construction of a new software system."

Eine Definition der *Wiederverwendbarkeit* (*reusability*) ist in [IEE90] zu finden:

> "The degree to which a software module or *other work product* can be used in more than one computer program or software system."

Die intuitiv angenommene Bedeutung stimmt mit diesen Definitionen durchaus überein. Hervorzuheben ist die ausdrückliche Betonung von „anderen Arbeitsergebnissen". Für diese Arbeit soll aber der Schwerpunkt auf der Wiederverwendung von Implementierungen verbleiben.
Eine genauere Aufschlüsselung der verschiedenen Möglichkeiten bietet Rubén Prieto-Díaz in [PD93]; für das untersuchte Feld ist dort die Unterscheidung zwischen *White-box Reuse* und *Black-box Reuse* besonders relevant.

> "*Black-box Reuse* is the reuse of software components without any modification. Black-box reuse is synonymous with reuse "as-is". Typically, reusable software components are packaged and their interactions defined by means of standard interfaces."

und

> "*White-box Reuse* is the reuse of components by modification and adaptation. This is by far the most common approach, mainly when reuse is conducted informally. The trend in white box reuse is towards parameterization and built-in adaptability."

3.2 Ziele und Widersprüche

Die vorhergegangene Definition wirft interessante Fragen bezüglich der Wiederverwendung von Aspekten auf: Die aspektorientierte Komposition verzichtet auf die klassischen expliziten Schnittstellen, *Black-Box Reuse* wird so in nicht trivialen Fällen zu einer besonderen Herausforderung. Aber auch die Frage *White-Box Reuse* ist nicht einfach zu beantworten. Bisher existieren nur wenige Ansätze für die Wiederverwendung von Aspektimplementierungen. Die Inanspruchnahme von existierenden Erfahrungen kann aber helfen, die Zielrichtung zu ergründen.

Das Vorbild für wiederverwendbare Aspekte sind hier hochgradig wiederverwendbare Systeme von Klassen, sogenannte *objektorientierte Frameworks*. Als solche werden üblicherweise halbabstrakte Anwendungen bezeichnet, die durch die Einbindung nutzerdefinierter Klassen mit neuen Systemen integriert werden und in diesen Funktionen bereitstellen. Je nach der Art der Funktionen wird zwischen *Application* und *Domain Frameworks* unterschieden. Erstere stellen Funktionen für Gruppen von Anwendungen bereit (horizontal), letztere Funktionen für ein begrenztes Problemfeld (vertikal). Beide Arten können als *Black-Box* wie auch als *White-Box* auftreten.

Dabei soll aber nicht der Eindruck erweckt werden, dass ähnlich komplexe Strukturen das Ziel seien: Der große Vorteil der Aspektorientierung liegt in schwächeren Bindungen für flexiblere und auch kleinere Strukturen.
Die Nähe zu dem Begriff *Framework* wird daher hier hergestellt, um anzudeuten, dass Konzepte und Lösungen aus der Objektorientierung – und davor – als Grundlage verwendet werden. Sie sind erforderlich, um generische Aspekte zu entwerfen, die in einem gewissen Umfang *zur Lösung von Aufgaben kooperieren*. *Framworkstatus* für die Ergebnisse wird hier noch nicht verfolgt.

Der Entwurf wiederverwendbarer Aspekte beinhaltet die Abwägung verschiedener Gegensätze. Tendenziell lässt sich beobachten: Je komplexer ein Aspekt wird, desto näher verbindet sich seine Funktionalität mit der Basis, desto konkreter müssen die Aussagen über die Eigenschaften der Basis sein. Umgekehrt verschlechtert die Nähe zur Basis die Wiederverwendbarkeit; zur Kompensation der festen Bindung wachsen die Anforderungen an eine Verbindungsschicht.
Wiederverwendung dieser stellt somit ein besonderes Problem während des Entwurfs dar und kann zum *Scattering* der Funktionalität bei gleichzeitigem Wachsen der Anforderungen führen.
Diese Eskalation folgt der aspektorientierten Komposition: Umkehrung der Einbindung

führt also auch zu einer Umkehrung der Anforderung; *Basisprogramme sind* (auch) *Frameworks für ihre Aspekte.*

3.3 Möglichkeiten wiederverwendbarer Aspekte

Der Entwurf wiederverwendbarer Software ist ein Gebiet, das seit Dekaden Forschung und Praxis beschäftigt. So gibt es eine Vielzahl erprobter Techniken, die sich mit dem Entwurf wiederverwendbarer Programme im Allgemeinen und Objekt-Orientierter *Frameworks* im Besonderen befassen. Bisher konnte aber kein anerkannter umfassender Prozess etabliert werden.
In Bezug auf die Aspektorientierung ist das Feld noch wesentlich offener. Da das Gebiet noch sehr jung ist, sind erprobte Entwurfsprozesse nicht in einem vergleichbaren Maße vorhanden.
Überlegungen bezüglich Wiederverwendung sind oftmals eher auf der Ebene von Entwurfsmustern und Idiomen, also nicht bei der Wiederverwendung von Implementierungen anzusiedeln. Die aufkommenden Sammlungen von *Library Aspects*([Isb06]) sind eine Entwicklung in Richtung der Wiederverwendung von Implementierungen. Sie bieten aber noch nicht die kooperierenden Elemente, die zur Lösung größer Probleme bzw. Abdeckung ganzer Domänen erforderlich wären.
Entwicklungen, die etablierten „*Frameworks*“ entsprächen, sind noch nicht Realität, es stellt sich daher die Frage nach *aspektorientierten Frameworks.*
Selbst der Begriff ist noch nicht fest besetzt. Er kann sowohl für Frameworks stehen, die aspektorientierte Funktionen bereitstellen, als auch für Frameworks, die aspektorientierte Funktionen nutzen (siehe: [VWD01], [HHUK04]).
Um dieser Unklarheit und den an den Begriff „*Framework*“ gekoppelten Erwartungen auszuweichen, wird im Folgenden der Begriff generische „*kooperierende Aspekte*“ verwendet.

3.3.1 Schnittstellen

Kooperierende Aspekte erfordern offensichtlich unterschiedliche Schnittstellen für die jeweilige Art der Inanspruchnahme. Zu nennen sind – angelehnt an [LLM99] – zunächst drei: In die Basis („*callouts*“, „*provided*“), aus der Basis („*callin*“, „*replace*“, *Pointcuts/-Advice*) und die *Features* der Aspekte selbst. Als mögliche vierte Schnittstelle sind noch durch den Aspekt selbst freigelegte *Joinpoints* zu ergänzen.
Die für die Aspektorientierung ausschlaggebende Schnittstelle ist hierbei die Kommunikation von der Basisanwendung in den Aspekt. Das Besondere dieser Schnittstelle ist die umgekehrte Richtung der Kommunikation: Der gewobene Aspekt enthält sowohl Implementierung als auch Aufruf der Schnittstelle. Für wiederverwendbare/generische Aspekte heißt dies, dass Mittel zur Anpassung an beliebige Basisprogramme vorhanden sein müssen.
Es kann auch für eine schwache Bindung der Basis an Aspekte argumentiert werden, etwa um Joinpoints explizit freizulegen.

So schlagen Gudmundson und Kiczales in [GK01] *Pointcut Interfaces* vor, öffentliche Schnittstellen für Klassen die *Pointcuts* bereit stellen, einzuführen. Damit obliegt es den Autoren der jeweiligen Klasse zu entscheiden, welche Joinpoints freigelegt werden, aber auch diese Schnittstellen gegebenenfalls an Änderungen der Implementierung anzupassen. Dafür wird zudem das Ausüben von „Druck“ auf die Implementation der Basis empfohlen, worunter Umwege in der Implementierung der Basis zur Erfüllung des *Pointcut Interfaces* zu verstehen sind.
Die jüngere Möglichkeit der Verwendung von Metadaten („Annotations“) ist als Nachfolger der Überlegung zu betrachten (siehe auch [KM05]). Hier findet die Markierung von *Joinpoints* nicht durch Umwege des Programmflusses statt, sondern durch sauberer zu trennende Metadaten. Zwar wird auch hier gleichermaßen das Prinzip der vollständigen *„Obliviousness“* in Frage gestellt, aber ohne negativen Auswirkungen auf die Codequalität der Basisanwendung und mit einer deutlich überlegenen Lesbarkeit.

Ob durch Umwege in der Implementierung oder durch *Annotations* verwirklicht, jedenfalls wird die vollständige Trennung von Aspekt und Basis in Frage gestellt. Aspekte sind jedoch in ihrer Wiederverwendbarkeit eingeschränkt, wenn sie von der Basis die Erfüllung und Bereitstellung von (Pointcut-)Schnittstellen erfordern.

Dies stellt eine deutlich schwächere *Separation of Concerns* dar als ursprünglich für AOP gefordert.
Umgekehrt aber ist der Realität Rechnung zu tragen. Die Einbindung neuer Funktionen a-posteriori ist ein ideales, nicht aber ein unverrückbares Ziel. Daher scheint eine moderate, nicht semantikverändernde Anpassung der Basis legitim, um Mächtigkeit und Verträglichkeit aspektorientierter Erweiterungen zu verbessern. Dabei muß die Bindung zwischen Aspekt und Basis hinreichend lose gehalten werden. Ideale bleiben hier *Obliviousness*, aber auch *Quantification*.

Grundsätzlich anders gelagert ist die Schnittstelle zur Kommunikation in der Gegenrichtung, also der Zugriff auf Eigenschaften der Basis durch Aspekte. Diese Kommunikation kann parallel zur vorgenannten laufen, dies ist aber keineswegs zwingend der Fall. Sollen Aspekte ohne eine feste Bindung funktionieren, so bedürfen sie einer losen Schnittstelle, um Informationen über die Basis zu erlangen. Auch hier kommt die Implementierung einer solchen Schnittstelle durch die Basis nicht in Frage. Es verbleiben demnach Vorkehrungen für entsprechende Adapter, um beliebige Basisklassen über eine generische Schnittstelle anzusprechen.

Schließlich wären noch die Schnittstellen zur direkten Ansprache der Aspekte zu nennen. Sollen Aspekte miteinander kommunizieren, so benötigen auch sie explizite Schnittstellen zum Austausch von Informationen. Dies entspricht weitgehend den Schnittstellen konventioneller Klassen und Komponenten. Nach Möglichkeit sollte eine aspektorientierte Komposition erlaubt werden: Aspekte sollten Joinpoints nicht nur nutzen, sondern auch – wenn technisch möglich und programmatisch sinnvoll – für andere Aspekte freilegen.
Dabei stellt sich die Frage: Gilt für Aspekte, was für Objekte selbstverständlich ist: *Diese sind keine Inseln*.

Soll dies auch für Aspekte bejaht werden, so ist auch das Anbieten von *Joinpoints* für andere Aspekte eine wichtige Erwägung. Idealerweise sollten Aspekte ihrerseits Joinpoints freilegen – hier durchaus in Form von *Pointcut Interfaces*. Für diese und die mit ihnen freigelegten Joinpoints lassen sich drei Kriterien formulieren:

- *Vollständig*: Vollständigkeit soll bezeichnen, dass alle relevanten Funktionen eines Aspektes auch als Joinpoints freigelegt werden. Perspektivisch handelt es sich hierbei um alle *Advices*.
- *Kontrollierend*: Die freigelegten Joinpoints sollen die Kontrolle über die Ausführung der zugrundeliegenden *Advices* ermöglichen. Genauer bedeutet dies, dass ein anderer Aspekt immer die Möglichkeit haben muss, die Wirkung des Freilegenden zu negieren.
- *Robust*: Die für absichtlich freigelegte Joinpoints selbstverständliche Eigenschaft der Robustheit: Die freigelegten Joinpoints sollen auch für alle abgeleiteten Aspekte freigelegt werden und – falls vorhanden – durch die gleichen *Pointcuts* ausgewählt werden.

In der gegenwärtigen Situation ist dies noch nicht immer zu erfüllen, als Ziel scheint diese Formulierung erstrebenswert.

3.3.1.1 Stabile Verbindungsschicht

Die Existenz von basisgewandten Schnittstellen erfordert den Entwurf einer stabilen Verbindungsschicht, um die Kommunikation in Richtung der Basis zu ermöglichen. Darunter ist eine generische Schnittstelle der Basis zu verstehen, vergleichbar mit der *expect* Schnittstelle von Aspectual Components.
Dies lässt den Nutzen von mächtigen lexikalischen *Pointcuts* fraglich scheinen, da diese unabänderlich lediglich die Kommunikation in Richtung des Aspektes beschreiben können. Aspekte müsssen aber oftmals[1] eine Annahme über die Schnittstelle der Basis tätigen, deren Details generisch gehalten werden sollten. Wiederverwendung von Aspekten kann daher von einer stabilen Verbindung profitieren, wenn diese als eigentliche Basis generischer Aspekte dient, um von Implementierungsdetails der Basis zu abstrahieren.

3.3.2 Hotspots

In objektorientierten *Frameworks* wird der Begriff *Hotspot* verwendet, um Bereiche großer Aktivität zu bezeichnen. Diese *Hotspots* sind die Orte, die besonders häufig *Extension Points* beinhalten, also Punkte zur Erweiterung durch *Clientcode*. Sollen Aspekte so betrachtet werden, dann existieren bei ihnen zwei verschiedene Arten von Orten der Anpassung.
Erstere sind die Orte um Aspekte und Aspektverhalten an das konkrete Umfeld anzupassen. Je nach dem betrachteten AOP-Ansatz kann dies sehr weitgehend der Anwendung

[1] Es lässt sich argumentieren, dass dies sogar immer der Fall ist. Es wird aber durch den Java-Typ `Object` verborgen.

von objektorientierten *Frameworks* entsprechen.
Die zweite Art sind die Orte, an denen der Kontrollfluss aus der Basis übernommen, bzw. an diese zurückgegeben wird, d.h. die bereits diskutierte Verbindungsschicht.
Es ist daher nötig, diese weitere Achse der Erweiterung zu beachten, um die Anpassung von Aspekten sowohl für die Basis als auch auf andere Aspekte zu ermöglichen, insbesondere, um Vorkehrungen für eine hinreichend mächtige Verbindungsschicht zu schaffen.

3.3.3 Verträge

Ein gern übersehenes Problem stellt die Einhaltung von Spezifikationen dar. Bei den bekannten OO-Frameworks obliegt die Aufgabe der Einhaltung von Verträgen(*Contracts*) dem Entwickler der Anwendung.
Übertragbar auf die Aspektorientierung ist dies aber nicht. Üblicherweise wird von Aspekten erwartet, die Spezifikationen der Basisprogramme einzuhalten. Dem kann in generischen Aspekten aber kaum entsprochen werden. Vielmehr erfordert dies die Möglichkeit, generische Aspekte an die Verträge der Basis anzupassen. Dies ist ein entscheidendes Kriterium an die Umsetzung von identifizierten *Hotspots* bezüglich des Entwurfs der Möglichkeiten einer Verbindung.

3.3.3.1 Verträge an die Basis

Der Schwerpunkt der etablierten aspektorientierten Ansätze liegt oftmals auf der Richtung Basis an Aspekt[2]. Also in der Bereitstellung von Methoden, um *Pointcuts* erschöpfend zu formulieren. So sollen *Obliviousness* und *Quantification* erreicht werden.
Dieser Schwerpunkt ist ein höchst bedeutsamer, aber für das vorliegende Problem nicht der Einzige. Zur Begründung, möchte ich eine nahezu triviale Beobachtung in das Feld führen: Bei den eingängigen objektorientierten *Frameworks* findet ein Anwendungsentwickler die Schnittstellen zur Kommunikation in das *Framework* vor. Es ist die Kommunikation in die Anwendung, welche noch zu implementieren verbleibt - eine Richtung, die das Korsett des *Frameworks* über die Anwendung verteilt[3].
Daher erfordern wiederverwendbare Aspekte, dass das Verhalten der Basis mit der Spezifikation des Aspektes in Übereinstimmung gebracht werden kann. Der beschränkende Faktor hierbei ist also die Anpassbarkeit der Verbindung, folglich der Entwurf der Schnittstellen. Die Mächtigkeit und Generizität der Verbindung ist daher ein entscheidender Faktor für die Anwendbarkeit von Aspekten.

3.4 Entwurf

Der Entwurf von wiederverwendbaren Systemen, insbesondere von Frameworks, ist eine besondere Herausforderung. Es würde Ziel und Sinn einer Fallstudie wie der vorliegenden

[2] *Inter-type declarations*, wie etwa bei AspectJ, sind bekannt. Das besondere Problem globaler *Introductions* ist ihre mangelnde Dynamik, insbesondere im Zusammenhang mit polymorphen Aspekten.
[3] In Form von festgelegten Datentypen

sprengen, einen allgemein gültigen Prozess zur Analyse und Entwicklung von wiederverwendbaren Aspekten vorzustellen.
Für Sinn und Ziel der Untersuchung wird es daher als hinreichend erachtet, grundlegende Anforderungen mittels wohl bekannter Techniken festzustellen und geeignet umzusetzen. Gerade in Bezug auf die anvisierten Aspektkomponenten mit sehr kleinen, isolierten Domänen ist dies als zielführend zu betrachten.
Der Entwurf kooperierender Aspekte wirft eine Reihe neuer Szenarien auf, da die erneut umgekehrte Richtung der Kommunikation von Bedeutung ist: Aspekte werden anders als Klassen objektorientierter *Frameworks* nicht explizit aus Basiscode aufgerufen. Die Auswirkungen nichtinvasiver Komposition auf Wiederverwendbarkeit müssen daher sorgfältig bedacht werden.
Als Entwurfsmethodik kommt eine Kombination angepasster, aber traditioneller Werkzeuge für den Entwurf von Frameworks zum Einsatz. Einen klaren Prozess zum Entwurf soll dies aber nur für den vorliegenden Fall bieten. Auch bei der Objektorientierung existiert kein durchgängig anerkannter Entwurfsprozess.
Zur Feststellung der Anforderungen fiel die Wahl auf eine stark verkürzte Fassung der Domänenanalysemethode FODA, zur weiteren Ausarbeitung wird in erster Linie auf die iterative Anwendung von Erfahrungen zurückgegriffen.

3.4.1 Domain Analysis - Domänenanalyse

Die Domänenanalyse ist eine Technik, die erstmals 1980 von James Neighbors [Nei80] benannt wurde. Es handelt sich um einen Ansatz, um Gemeinsamkeiten und Variabilitäten von Software und Arbeitsabläufen zu identifizieren. Ziel ist es, die Erfordernisse zur Entwicklung von besonders wiederverwendbarer Software festzustellen. Sie liegt einer Vielzahl von Methoden und Werkzeugen zu Grunde (DARE, FODA,...). Allen diesen Ansätzen gemein ist die Einbeziehung von Fachwissen, Experten und bestehenden Produkten zur Gewinnung der erforderlichen Erkenntnisse. Eine beispielhafte Definition ist in [PD90] zu finden:

> "We define domain analysis as a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems. During software development, information of several kinds is generated. From requirements analysis to specific designs to source code. Source code is at the lowest level of abstraction and is considered the most detailed representation of a software system. Complementary key information is also generated during software development. Code documentation, history of design decisions, testing plans, and user manuals are all essential to convey a better understanding of the total system."

Die Domänenanalyse wird dort definiert als Prozess zur Organisation vorhandenen Wissens in Form eines Domänenmodells für den Entwurf neuer, wiederverwendbarer Systeme. Für die vorliegende Untersuchung ist zunächst die Feststellung von Variabilitäten und

Konstanten in der betrachteten Domäne wichtig. Weder in Umfang noch in Zielrichtung wird eine vollständige Untersuchung nach einer Domänenanalysemethode verfolgt.

3.4.2 FODA

Die *Feature Oriented Domain Analysis* wurde Ende der 80er Jahre am *Software Engineering Institute* der *Carnegie-Mellon University* entwickelt[4]. Eingeführt wurde sie von Kang et al. in [KCH+90] in einer Machbarkeitsstudie anhand einer praktischen FODA-Analyse von *Windowmanagern*. Der besondere Schwerpunkt der FODA liegt in der Identifikation von zentralen *Features* in verwandten Softwaresystemen, um aus jenen *Features* die *Domäne* abzuleiten. Sie bietet dafür Methoden, um das Domänenmodell zu erstellen und Informationen für die Anwendungsentwicklung aufzubereiten. FODA geht davon aus, dass das generische Modell Applikationen beschreibt und Modelle konkreter Anwendungen aus dem Ergebnis der Analyse durch *Verfeinerungen* hergeleitet werden können. Gemeinsamkeiten werden also im Modell zusammengefasst, Unterschiede später auf *refinements*/Verfeinerungen zurückgeführt.

3.4.2.1 Phasen und Ergebnisse der FODA

Um auf ein solch hohes Niveau der Abstraktion zu gelangen, kommen in der FODA drei verschiedene Modellierungsoperationen zum Einsatz. Das sind: Generalisierung/Spezialisierung, Aggregation/Dekomposition und Parametrisierung. Die ersteren beiden Paare führen zur Abstraktion des Modells, dessen mögliche Verfeinerungen – Parameter – , der Gegenstand der letztgenannten Methode sind.
Die FODA deckt zwei Phasen der Domänenanalyse ab, die *Kontextanalyse* und die *Domänenmodellierung*. Der zur Domänenanalyse gehörige Schritt der *Architekturmodellierung* wird hier übersprungen. *FODA* sähe hierfür die *DARTS* Methode vor, die sehr laufzeit- und datenflussorientiert ist und daher als nicht geeignet angesehen wird.

Der initiale Schritt ist die *Kontextanalyse*, die dazu dient die Grenzen der zu untersuchenden Domäne festzustellen. Die Untersuchung wird mit dem Strukturdiagramm visualisiert, welches als Blockdiagramm die Beziehungen zu Sub- und Superdomänen darstellt. Ergänzend wird mit dem Kontextdiagramm der Datenfluss zwischen der betrachteten Domäne und Nachbardomänen dargestellt.
Das Ziel der Suche nach Kommunalitäten und Unterschieden wird in dem sogenannten *Feature* Modell deutlich; dessen prominentestes Merkmal stellen die *Featurediagramme* dar, welche als *und/oder* Bäume die Beziehungen und Abhängigkeiten zwischen *Standardfeatures* der Domäne abbilden.
Features werden dabei durch Knoten symbolisiert. *Und*-Knoten signalisieren eine Komposition, solche Knoten bestehen aus ihren Kindknoten. *Oder*-Knoten stehen für alternative *Features*, ihre Kinderknoten also für verschiedene mögliche Umsetzungen. Die wichtigste Unterscheidung ist die Bezeichnung eines *Features* entweder als *optional* oder *mandatory*

[4] Einen Zusammenhang zu *Star Wars* legt nicht nur die Namenswahl nahe.

– zwingend notwendig. Dies wird durch Kompositionsregeln ergänzt, die Abhängigkeiten und Ausschlusskriterien formal ausdrücken.

3.4.3 Modifikationen

FODA ist, wie nahezu alle Domänenanalysetechniken[5] eine Technik, die nicht für einzelne Analysten verfasst wurde. Ein Teil der Schritte kann aus praktischen Erwägungen nicht erfolgen, andere haben sich durch Entwicklungen der Softwaretechnik überlebt oder verschoben. Im Folgenden werde ich die getätigten Vereinfachungen und Veränderungen vorstellen. Eine Abkehr von der ursprünglichen Zielsetzung erzwingen nicht zuletzt die eingeschränkten Möglichkeiten, *Domänenwissen* einzubringen. Daher scheint zunächst

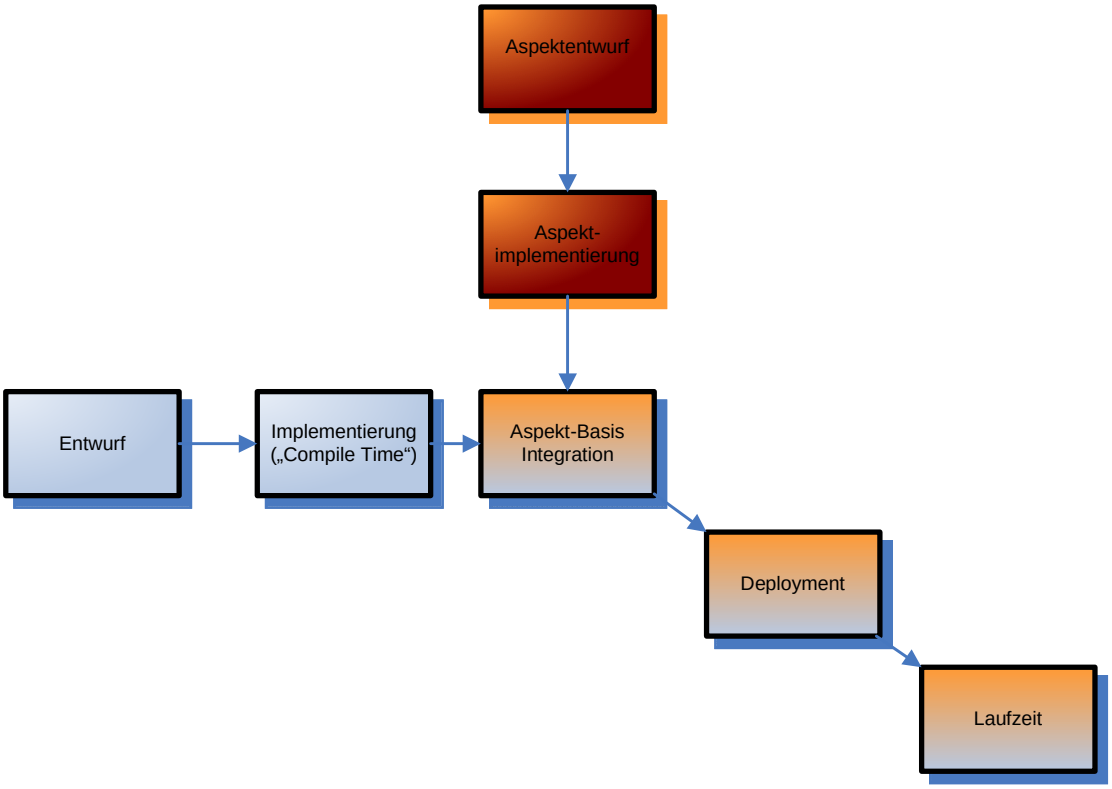

Abbildung 3.1: Vorgeschlagenes erweitertes Wasserfall Modell der Bindungsphasen. Für den untersuchten Fall ist die Entwicklung in im Bild vertikaler Richtung interessant; der Aufwand der Phasen ab der *Kreuzung* kann als Maß für die Wiederverwendbarkeit betrachtet werden.

die Einschränkung und Reduzierung der Sicht auf die Domäne gerechtfertigt. Die Erstellung eines Dömanenmodells, wie es auch die FODA liefern würde, soll als wichtigstes Ziel dieser Phase gelten. Darüber hinaus sind einige Modernisierungen zu treffen: FODA

[5] Genauer: wie die Domänenanalyse schlechthin.

„kennt“ die Bindungszeiten *Compile-Time*, *Load-Time* und *Runtime* zur Festlegung des Moments der Entwicklung, in dem die Entscheidung über ein Feature fallen muss.
Daher ist zumindest *Deployment-Time* noch als weitere Stufe der Entwicklung zu ergänzen. Aber selbst diese Erweiterung der Wasserfall-Modell ähnlichen Einteilung ist im Falle von a-posteriori integrierten aspektorientierten Ansätzen nicht hinreichend. Vielmehr ergeben sich in diesem Fall zwei orthogonale Achsen der Entwicklung.
Die Entwicklung des Aspekts soll weitgehend unabhängig von der Entwicklung der Basisanwendung stattfinden; die Entwicklung der Basisanwendung mindestens ebenso unabhängig von dem Aspekt. Es existieren also zwei Wege der Entwicklung, deren Integration miteinander einen weiteren Abschnitt der Entwicklung in einer neuen Richtung darstellt.
Als Folgerung schlage ich die Achse der Basis mit den Stufen Basisentwurf, Compilezeit und *Deployment*, sowie die Achse der Aspekte mit Aspektentwurf und Compilezeit vor. Diese finden in der gemeinsamen Achse Aspektintegration, Deployment und Laufzeit ihre Fortsetzung.

3.4.4 Umsetzung

Ein direkter Weg zur Umsetzung der Ergebnisse kann nicht angegeben werden. Die Kenntnis der identifizierten Features und ihrer Abhängigkeiten erlauben die Anwendung etablierter Entwurfsmethoden. Weitere abstrakte Analysemethoden können aber gegebenenfalls die folgenden Schritte verkleinern.
Beispielsweise ist der Ansatz von Riehle und Gross aus [RG98] zu nennen, ein rollenbasierender Modellierungsansatz zum Entwurf objektorientierter *Frameworks*. Dieser bietet einen solchen Ansatzpunkt, der tendenziell für den Entwurf der Basisanbindung von Aspekten geeignet scheint. Der praktische Nutzen bei Versuchen mit dieser Methode war aber beschränkt, was in der Art der Kommunikation in den untersuchten Domänen begründet sein mag. Hier sind gegebenenfalls noch weitere Untersuchungen nötig.

4 Methodik der Fallstudie

Dieser Abschnitt bietet einen kurzen Überblick über die Rahmenbedingungen der Fallstudie. Dazu zählen die gewählten Szenarien und die Regeln für den Umgang mit Basisanwendungen.

Die Fallstudie trägt dem Forschungscharakter der Ansätze Rechnung. In methodischer Hinsicht wurde eine Gruppe von zu implementierenden Aspekten ausgewählt, die in ihrer Komplexität überschaubar sind.
Wesentlich bedeutsamer ist die Zielsetzung: Ergebnis soll nicht eine eindeutige Metrik sein, sondern eine bedingte Beurteilung der Umsetzbarkeit.
Hierfür wurde die Entwicklung äquivalenter Aspekte unter Einsatz von *ObjectTeams* und *JAsCo* und ohne eine feststehende Bindung an eine Basisanwendung gewählt.
Im Anschluss erfolgt die Anpassung an eine Reihe von Basisanwendungen, welche nicht unter Kenntnis der Aspekte entstanden, um die Eignung zur a-posteriori Integration beurteilen zu können. Allerdings sind innerhalb dieser Anwendungen einige – von Aspekten nicht unmittelbar abhängige – Anpassungen erforderlich, um eine einheitliche Behandlung zu ermöglichen.

4.1 Ausgewählte Aspekte

Für die Fallstudie war also nach Aspekten zu suchen, die – gemäß dem Ideal der schwachen Bindung – nur mittelbar und optional voneinander abhängen, aber kooperativ in einer Domäne eingesetzt werden können. Dabei wird das Ziel der Möglichkeit einer a-posteriori Integration verfolgt.
Als unter diesen Gesichtspunkten aussagekräftige Beispiele, wenn auch nicht unbedingt von praktischer Relevanz, wurden daher *Sessionmanagement*, *Authentikation* und *Autorisation* ausgewählt. Diese Aspekte treten in bestehenden Ansätzen oft ineinander verwoben (*tangled*) auf, wenn auch zumeist in den hier nicht weiter betrachteten *Web-Anwendungen*.
Sessionmanagement soll dabei bedeuten, dass ein Nutzungskontext über aufeinanderfolgende Aktionen hinweg transportiert wird. Es zerteilt sich daher in zwei Unterprobleme: die Unterscheidung verschiedener – möglicherweise parallel stattfindener – Nutzungsvorgänge, sowie den Transport von kontextabhängigen Daten zu allen abhängigen Programmteilen.
Autorisation ist weniger aufwändig zu beschreiben: Einzelne Aktionen sollen mit Zugangsbeschränkungen belegt werden. Dies ist von einer vorhergehenden *Authentikation* abhängig, also der Feststellung der Identität eines Nutzers.

4.2 Vorstellung der Basisprogramme

Als Basisprogramme sollen mittelgroße Java-Anwendungen dienen, deren Implementierungen die ausgewählten Aspekte in geringer bis keiner Weise selbst verwirklichen. Diese Anwendungen wurden nicht gezielt für diese Studie implementiert. Insbesondere handelt es sich um einsatzfähige und vollständige Programme. Dieser Aufbau soll die Verwendung mit einer *oblivious* Basis praxisnah nachbilden.

4.2.1 Das Dispositionssystem

Das Dispositionssystem ist eine im Rahmen des Topprax Projektes entstandene Beispielsanwendung, die dort als objektorientierte Vergleichsprobe dient. Sie basiert auf dem Trend System der Firma Gebit und stellt eine realitätsnahe Plattform dar.
Die Anwendung bietet ein realistisches Umfeld, da sie ohne jede Absicht einer späteren Anpassung, insbesondere nicht einer aspektorientierten, entworfen und implementiert wurde. Ferner finden sich andere fortgeschrittene Techniken, etwa eine datenbankgestützte Persistenzschicht.
Konkret handelt es sich um ein *Frontend* zur Verwaltung von Warenbeständen in einem – virtuellen – Handelsunternehmen. Als Einzelplatzanwendung entworfen, sollen Nutzer Waren ordern, Lagerbestände kontrollieren, Bestellstrategien entwerfen und Algorithmen konfigurieren können. Umgesetzt ist es eine Anwendung, die in einem Fenster eine karteikartenähnliche Bedienlogik implementiert, wobei jede Karteikarte einen *Workflow* widerspiegelt. Ein solcher *Workflow* kann aus mehreren Schritten bestehen; mehrere *Workflows* – auch gleichartige – können parallel auftreten.

4.2.2 Der Logdateibetrachter

Der Logdateibetrachter ist eine Anwendung zum Betrachten von Logdateien, wie sie etwa von dem bereits genannten Trend-System erstellt wird. Es handelt sich um eine Anwendung mit ausschließlich lesendem Zugriff. Dieses Lesen kann aber lokal, über *http* oder über eine Client/Server Architektur stattfinden. Daher ist diese Anwendung sowohl als isolierter *Client*, als auch als (beliebiger)Teil einer Client/Server Architektur einsetzbar. Die Anwendung kann durch dieses vielseitige Auftreten sowohl als isolierte Anwendung betrachtet werden, wie auch als ein Client oder Server in einer verteilten Anordnung. Sie bietet also eine sehr vielseitige und aussagekräftige Basis für die anstehende Untersuchung.

4.3 Anpassung der Basisprogramme

Aspektorientierte Programmieransätze sind mitunter durch eine verschwindend schwache Bindung der Basisprogramme an ihre Aspekte definiert (*Obliviousness*). Die Schwäche dieser Bindung gehört zu den Eckpfeilern der Erwartung besserer Wiederverwendbarkeit. Dementsprechend bewertet diese Fallstudie die Anwendbarkeit von generischen Aspekten auf unabhängig von ihnen programmierten Basisanwendungen.

Allerdings sollte eine solche schwache Bindung nicht mit der Abwesenheit von Bedingungen gleichgesetzt werden. Schwache Bedingungen verletzen den Gedanken der *Obliviousness* nicht unbedingt, da sie für jegliche Aspekte auf jeder Basis zu finden sind. In starkem Maße sind sie technisch vorgegeben , obgleich auch eine nichtinvasive Bindung, wie durch *Weaving*, durchaus generelle Bedingungen an die Basis stellen kann.
Um einen Vergleich zu ermöglichen, wurden daher eine Reihe von Anpassungen an den Basisanwendungen vorgenommen, die zumeist im Bereich von *Refactorings* liegen. Die folgende Auflistung stellt diese vor.

- **Sparsamer Einsatz von inneren und eingebetteten Klassen**
  Keine aspektorientierte Technik erlaubt die Auswahl von Aufrufen in anonymen Klassen als Joinpoints im Kontext der einschließenden Klassen/Objekte[1]. Darüber könnte eine längere Diskussion geführt werden, da solche Klassen über Interfaces und Superklassen durchaus in Pointcuts eingesetzt werden könnten. Dies ist aber nicht hinreichend, da in einem Aspekt das Wesen als innere Klasse nicht mehr sichtbar wäre, der Aspekt also auf Basis der Inneren und nicht der umgebenden Klasse ausgeführt werden würde. In Ermangelung einer Alternative sollen daher Ausdrücke wie in Listing 4.1 Verwendung finden. Jede Aktion, die über die reine Verwaltung einer GUI o.Ä. hinausgeht, sollte in einer eigenen Methode der zugrundeliegenden Klasse stattfinden, nicht in eingebetteten oder gar anonymen Klassen. Diese Forderung ist durchaus unabhängig von AOP zu sehen, da es auch in reinen objektorientierten Programmen als besserer Stil anzusehen ist, auf Implementierungen in anonymen Klassen zu verzichten.

```
1  class Example{
2
3   void eventHandlingMethod(ActionEvent e){
4    ...
5   }
6
7   ... = new ...(){
8    public void actionPerformed(ActionEvent e){
9     eventHandlingMethod(e);
10   }
11  };
12 }
```

Listing 4.1: Zu bevorzugender *Inner Class* Stil

- **Konsistente Namensvergabe für Klassen und Methoden**
  Die strikte Einhaltung von *Naming Conventions* ist im Zusammenhang mit lexikalischen *Joinpoints* eine naheliegende Voraussetzung. Allerdings ist auch dies letztlich nur eine Forderung nach einem sauberen objektorientiertem Ansatz. Die Namensvergabe schließt auch die Ordnung der Parameter eines Methodenaufrufs ein. Dabei

[1] Eine Auswahl der Methoden ist mittels *Wildcards* durchaus möglich. *Advice* wird dann aber auf Basis der anonymen Klasse ausgeführt.

sollte die Reihenfolge von *Nadel und Heuhaufen* innerhalb der Anwendung, mindestens aber innerhalb einer Klasse, gleichbleibend sein.

- **Moderate *Encapsulation***
  Dies ist eine Anforderung, die normalen objektorientierten Gepflogenheiten widerspricht. Ein *Crosscutting Concern* kann mitunter zwingend auf Informationen angewiesen sein, die eigentlich durch die Klassengrenzen verborgen werden und möglicherweise auch verborgen werden sollten. AOP Sprachen wie *ObjectTeams* oder *AspectJ* bieten innerhalb gewisser Grenzen Mittel, um auch auf versteckte Informationen zuzugreifen. Dies zeigt aber, dass die Verwendung stilistisch irreführend ist – die Informationen werden keineswegs vor externem Zugriff geschützt. Daher sollten *Features*, die als Joinpoints in Frage kommen oder von Aspekten modifiziert werden, als `public` deklariert werden, selbst wenn unter normalen Kriterien eine Aufnahme dieser Methoden in die öffentliche Schnittstelle nicht in Betracht käme[2].

- **Subklassen**
  Momentan ermöglichen die üblichen *Weaver* nicht das Weben von Aspektcode in die Klassen der Java API. Für die Typhierarchie einer Anwendung relevante Klassen dieser API sollten daher in Form von Subklassen verwendet werden. Das macht ihre Schnittstellen als Joinpoints verfügbar. Diese Klassen benötigen keinerlei Logik, allenfalls `super`-Aufrufe, um *Joinpoints* freizulegen.

4.4 Vergleichskriterien

Die Fallstudie bietet eine vergleichende Auswertung von Entwürfen und Implementierungen in den verwendeten Sprachen. Dafür werden zunächst die vorgestellten Aspekte entworfen, implementiert und in die Basisanwendungen integriert. Diese Integration, insbesondere deren Aufwand und ihre Grenzen, soll dann als Grundlage für Vergleiche dienen, die aus verschiedenen Perspektiven getätigt werden.
Vergleichsgegenstand sind verschiedene Gesichtspunkte der Anwendung solcher Aspekte in dem konkreten Umfeld betrachtet werden, aber auch Möglichkeiten von besonderen *Features*.
Diese Vergleiche werden – soweit möglich – anhand ähnlicher Mittel in beiden Sprachen umgesetzt. In Abwesenheit von Äquivalenten wird eine nicht vergleichende Beurteilung vorgenommen.
Sollten einzelne *Features* nicht in den Umsetzungen berücksichtigt werden können, so ist eine gezielte Abänderung der anfangs entworfenen Aspekte ein Mittel, um die Studie an diesen Punkten zu erweitern. Erkenntnisse aus solchen spezialisierten Szenarien können aber kaum mit der gleichen Relevanz belegt werden.

Als konkrete Vergleichskriterien sind z.B. Wiederverwendbarkeit von Verbindungscode, Möglichkeiten von Advice, Möglichkeiten zur Abstraktion von der konkreten Basis und

[2] Der Sicherheitsgewinn durch strenge Schnittstellen ist umstritten, eine Verbesserung wird aber sicher nicht auftreten. Hier kann eine Abwägung erforderlich sein.

andere für eine Wiederverwendung von Aspekten relevante Punkte, gedacht. Der konkrete Nutzen der verschiedenen Spracheigenschaften soll so betrachtet werden können. Ein globaler Vergleich oder gar eine übergreifende Bewertung ist dabei aber ausdrücklich nicht Ziel der Studie.

5 Betrachtung der Ansätze

Um die Bausteine der Fallstudie vorzustellen, erfolgt zunächst eine Diskussion grundlegender Bausteine von Anwendungen in JAsCo und ObjectTeams. Diese Diskussion mischt sowohl bereits dokumentierte, als auch neue Grundstrukturen und erläutert Strategien, die in der Fallstudie Verwendung finden.

5.1 Grundtechniken wiederverwendbarer Aspekte in JAsCo

Anknüpfend an die Vorstellung der Sprache JAsCo, stellt dieser Abschnitt die grundlegenden Überlegungen im Umgang mit ihr dar. Diese basieren größtenteils auf Erfahrungen, die aus dem Umgang mit der Technik gewonnen werden konnten.

JAsCo verfügt über ein vergleichsweise schwaches statisches Typsystem, wie bei aspektorientierten Sprachen nicht unüblich. Dies ist weitgehend unkritisch, da in besonders basisabhängigen Teilen, wie etwa *Refining Classes*, die vorhandenen Typinformationen von JAsCo auch genutzt werden. Dennoch bleibt eine abstrakte Behandlung weitgehend möglich. Daraus folgt aber auch direkt ein erheblicher Dokumentationsaufwand für *Aspect-Beans*, um *Connector*-Module kompatibel gestalten zu können.

5.1.1 Pointcuts in JAsCo

5.1.1.1 Semantik des Pointcut-Modells

Über die Semantik der *Pointcutdesignatoren* zur Auswahl von `execution` *Joinpoints*, momentan die einzigen von JAsCo unterstützten, sagt die JAsCo Dokumentation wenig aus. Zu erfahren ist nur, dass das Konstrukt an das gleichnamige Pendant in AspectJ angelehnt ist. In dieser Tradition ist das Modell nicht formal spezifiziert. Selbst eine belastbare informelle Beschreibung der Semantik fehlt.

Tatsächlich ähnelt das Verhalten der `execution` *Pointcuts* der, von Barzilay et al. in [BFTY04] formulierten, Beschreibung des AspectJ Pointcutmodells. Wie auch in AspectJ werden geerbte Methoden einer Klasse nicht als *Joinpoints* für solche *Pointcuts* herangezogen. Allerdings gibt es einige Unterschiede. Inspiriert von [BFTY04], wird hier eine Formalisierung der durch Beobachtung gewonnenen Erkenntnisse geboten:

$$jp, \text{ ein Methodenaufruf } x.f() \text{ mit } x \in D$$
$$am = \texttt{* *.C.*(*)}$$
$$pc_e^{am} = \texttt{execution(am(..args))}$$

resultiert in:

$jp \in pc_e^{am} \Leftrightarrow D \subseteq C \wedge f$ ist definiert in $C \wedge f$ ist nicht redefiniert in einer Klasse E, mit $D \subseteq E \subset C$

beziehungsweise:

$$jp, \text{ ein Methodenaufruf } x.f() \text{ mit } x \in D$$
$$am^{+} = \texttt{* *.C.*+(*)}$$
$$pc_e^{am^{+}} = \texttt{execution(am(..args))}$$

ergibt[1]:

$$jp \in pc_e^{am^{+}} \Leftrightarrow f \text{ ist definiert in } E \text{ mit } D \subseteq E \subseteq C$$

Und der '++' Operator:

$$jp, \text{ ein Methodenaufruf } x.f() \text{ mit } x \in D$$
$$am^{++} = \texttt{* *.C.*++(*)}$$
$$pc_e^{am^{++}} = \texttt{execution(am(..args))}$$

führt zu dieser Semantik:

$$jp \in pc_e^{am^{++}} \Leftrightarrow D \subseteq C \wedge f \text{ ist deklariert in } C$$

Auch das Verhalten des `cflow` Pointcut Designators bleibt – sobald mit konkreten Parametern belegt – hinsichtlich Flexibilität hinter dem Vorbild AspectJ zurück. Anders als bei AspectJ, ist das Argument des `cflow` Ausdrucks von JAsCo nicht ein beliebiger Pointcut, sondern baut implizit auf `execution` auf. Listings 5.2 bis 5.4 illustrieren die Bedeutung des JAsCo `cflow` Designators praktisch verglichen mit jenem aus AspectJ. Es ist in JAsCo auch möglich einen abstrakten Pointcutparameter an mehrere Designatoren zu binden, um etwa Rekursion auszuschließen. Eine solche Festlegung ist aber bereits in der AspectBean nötig.
Die Parameter des als Anker für den `cflow` Ausdruck dienenden Joinpoints werden nicht gespeichert und stehen nicht für dynamische Bedingungen oder *Advicecode* zur Verfügung.

5.1.1.2 Einsatz von Designatoren

Robuste Pointcuts, also Pointcuts, die einer Evolution der Basis standhalten, sind eine wichtige Anforderung an den Entwurf aspektorientierter Programme[2]. In JAsCo ist dies nicht gänzlich anders, aber hier ist der Entwurf *robuster abstrakter Pointcuts* eine grundlegende Anforderung.

[1] Der JAsCo Compiler schränkt auch bei Angabe des '+' Operators die gültigen Pointcuts auf die in 'C' deklarierten Methoden ein. Das kann mit *Wildcards* umgangen werden, womit aber nicht immer äquivalente *Pointcuts* möglich sind.
[2] Zumindest für Sprachen mit Pointcut-Ausdrücken.

Die Rechtfertigung dieser Gewichtung liegt in der Unveränderbarkeit von Hookonstruktoren. Diese können weder redefiniert noch erweitert werden.
Daraus lassen sich zwei Ziele für den Entwurf von Hookkonstruktoren und damit abstrakten *Pointcuts* formulieren: Zum einen sollte jeder mögliche *Joinpoint* in einem konkreten *Pointcut* erfassbar sein, zum anderen sollte es möglich sein, jeden möglichen *Joinpoint* aus einem konkreten Pointcut auszuschließen.

Da es nicht möglich ist Reihenfolge, Typ oder Zahl der „festen“ Parameter in Hookkonstruktoren anzupassen, ist zur Erfüllung des ersten Ziels der Verzicht auf die Angabe von solchen Parametern in Hookkonstruktoren sinnvoll – ansonsten wären *Joinpoints* nicht mehr auswählbar. Sollten Parameterwerte von *Advicecode* benötigt werden, so bliebe immer noch der Rückgriff auf das ungetypte *Parameterarray* möglich, welches JAsCo zur Verfügung stellt.
Ergänzend ist ein Widerspruch zwischen diesem Ziel und dem `target` -Designator festzustellen[3]: Dieser bietet nicht die Möglichkeit, abstrakt gehalten zu werden und führt dadurch zur festen und endgültigen Bindung der *Hooks* an konkrete Basisklassen – hier kehrt JAsCo vom *Connector* Prinzip ab.

Das zweite Ziel, das genaue Auswählen von Joinpoints, erfordert die optimale Nutzung der angebotenen *Pointcutdesignatoren*. Die größte Mächtigkeit erreicht eine Konjunktion aus allen Designatoren – mit Ausnahme von `target` – sowohl negiert, als auch unnegiert. Der Einsatz einer solchen Konstruktion ist allerdings komplex: Im Connector müssen unbenötigte Teile mit neutralen Ausdrücken gefüllt werden, d.h. Dummy-Klassen[4] für die negierten und allgemeingültigen Wildcards (`* *.*(*)`) *Pointcuts* für die normalen Poincutdesignatoren.
Jedoch deckt auch ein solcher abstrakter *Pointcut* nicht alle Situationen ab. Insbesondere bei der Verwendung der '+' und '++' Ausdrücke werden auch `super`-Aufrufe ausgewählt, die sauber nur mit einem an den gleichen abstrakten Pointcutparameter, wie das einleitende `execution`[5], gebundenen `!withincode` ausgeschlossen werden können.

```
...
hook Example {

  Example(executionM(..args), notExecutionM(..args2),
          cflowM(..args3),notCflowM(..args4),
          withincodeM(..args4),notWithincodeM(..args5)
          )
  {
    execution(executionM) && !execution(notExecutionM) &&
```

[3] Wie bereits erwähnt, ist der `target` Pointcutdesignator in JAsCo 0.8 ohne Funktion. Die Kritik bezieht sich auf die Sprachdefinition.
[4] Eine unter Umständen wichtige Randbemerkung: Die Dummy-Klasse darf keinesfalls `Dummy` genannt werden, da der JAsCo Compiler diesen Namen intern nutzt.
[5] Den Variablennamen aus Listing 5.1 folgend, wäre ein „`&&!withincode(executionM)`“ erforderlich.

```
10        cflow(cflowM) && !cflow(notCflowM) &&
          withincode(withincodeM) && !withincode(notWithincodeM);
12    }
  ...
```

Listing 5.1: Robuster abstrakter Pointcut. Die Parameterarrays `args2` – `args5` werden von JAsCo nicht mit Werten gefüllt. Um die negierten abstrakten Designatoren mit neutralen Pointcuts belegen zu können, ist eine methodenfreie Dummy-Klasse erforderlich; `* *.*(*)` fungiert für die nicht negierten Designatoren als neutrales Element – wenn auch nicht gänzlich seiteneffektfrei.

```
1  ...
   hook MyCFlowHook {
3  MyCFlowHook(method1(..args1),method2(..args2)) {
     execution(method1) && cflow(method2);
5  }

7  before() {
     ...
9  }
   ...
```

Listing 5.2: Ein `cflow` Ausdruck in JAsCo.

```

2  static connector MyConnector MyConnector {
   MyAspectComponent.MyCFlowHook hook = new MyAspectComponent.
       MyCFlowHook( * *.ClassA.foo(*),* *.ClassB.bar(*));
4  }
```

Listing 5.3: Der zu Listing 5.2 gehörige Connector.

```
   public aspect TestAspect {
2
   pointcut method2PC() : execution(* ClassB.bar(..));
4  pointcut method1PC() : execution(* ClassA.foo(..)) && cflow(
       method2PC()) ;

6  before(): method1PC(){
       ...
8  }
   }
```

Listing 5.4: Zu Listing 5.2 äquivalenter cflow-Ausdruck in AspectJ.

5.1.2 Anwendung von Refinements

JAsCos Ansatz, um basistypabhängige Funktionen zu erlauben, sind `refinable` Methoden (siehe Abschnitt 2.2.1.6). Diese Methoden gestatten es, die *AspectBeans* unter Verzicht auf Vererbung vielseitig zu spezialisieren. Die Implementierungen für solche Methoden können in *Connector*-Dateien oder in *Refining Classes* angegeben werden, die jedoch

beide nicht vererbt werden können – Implementierungen von `refinable` Methoden sind daher kaum wiederverwendbar.
Erschwerend kommt hinzu, dass bei der Implementierung in *Refining Classes* mitunter Verhalten am Rande der Determiniertheit auftreten kann. Denn JAsCo zieht immer die erste, passende *Refining Class* heran. Dabei kann der Name einer solchen Klasse hinreichend sein, um einen Unterschied in der Auswahl zu bewirken.
Dieser Mangel an Determiniertheit verhindert auch die zuverlässige Verwendung einzelner *Refining Classes* durch mehrere *Hooks* – selbst bei in einer Vererbungshierarchie stehenden Hooks wird nicht zwingend das speziellste *Refinement* ausgewählt.
Im praktischen Umgang mit JAsCo erwies sich daher die Möglichkeit der Implementierung im *Hook*-Konstruktoraufruf oftmals als die zu bevorzugende.

5.1.2.1 Dynamische Bedingungen in Refinements

Diese `refinable` Methoden können eingesetzt werden, um das `isApplicable` Konstrukt – oder Teile davon – in den *Connector* zu verlegen. Dies ist eine vergleichsweise häufig einsetzbare Anwendung der Technik, um die Mächtigkeit des Connectors gegenüber dem jeweiligen Hook zu stärken, also den leicht redefinierbaren Teil des *Hooks* zu vergrößern. Am einfachsten ist dies zu erreichen, indem eine bool-wertige `refininable` Methode als Quelle des Rückgabewertes im `isApplicable` Ausdruck eingesetzt wird.

5.1.2.2 Virtual Mixins zur Nutzung von Refinements durch mehrere Hooks

Virtual Mixins eignen sich, um eine Refining Class durch mehrere Hooks nutzbar machen zu können. Dies gilt trotz der genannten Problematik bezüglich der Übertragbarkeit von *Refinements*.
Praktisch geschieht dies, durch Delegation der basisgewandten Methoden über eine Mixin-Instanz, deren Hook das Interface über Refinables implementiert. Eine solche Anordnung funktioniert auch über *AspectBean-* und *Connector*grenzen hinweg.
Dabei ist aber besondere Vorsicht bezüglich des *Pointcutausdrucks* dieses *Mixin-Hooks* nötig: Alle in Frage kommenden Basistypen müssen erfasst werden. Zudem müssen für jeden dieser Typen Definitionen der fraglichen `refinable` Methoden vorhanden sein.

5.1.3 Ausgewählte Anwendungen

Den JAsCo Teil abrundend, werden noch einige im Rahmen der Fallstudie entwickelte Lösungsansätze für häufig anzutreffende Implementierungsprobleme vorgestellt werden. Ob diese Ansätze *Idiomstatus* haben, ist hier nicht zu beantworten.

5.1.3.1 Target revisited

Als Ersatz für den `Target` Designator lässt sich eine `Refinable` Methode für das `isApplicable` Konstrukt einsetzen. Zwar ist das Vorgehen unter dem Gesichtspunkt der Performanz, unzweifelhaft fragwürdig, aber es bietet zwei Vorteile: Erstens die deutlich verbesserte Wiederverwendbarkeit: Eine Angabe spezifischer Klassen kann – sei es in

Form einer Methodendefinition oder als Argument einer Init-Methode – im Connector erfolgen. Das ist in der vorliegenden Sprachdefinition nicht für `target`-Pointcuts erlaubt. Der zweite Vorteil ist die Umsetzbarkeit: `Target` wird von JAsCo 0.8.x [6] nicht unterstützt.
Für die Umsetzung dieser Strategie bieten sich drei Vorgehensweisen an; sie stimmen in der Verwendung des `isApplicable` Konstrukts überein:
Erstere Möglichkeit ist die zuvor bereits beschriebene Verwendung einer `refinable` Methode, um `isApplicable` vollständig nach aussen zu öffnen.
Dies erlaubt einerseits eine sehr flexible Steuerung des Aspektes, sei es aus Connector-Modulen heraus, oder über eigene, typabhängige `Refining Classes`. Andererseits öffnet es aber auch eine breite Funktionalität ohne zwingenden Grund, daher kann die Modularisierung dieses Ansatzes als fragwürdig bezeichnet werden. Umgekehrt ist so eine feinstufige Verwendung dynamischer Bedingungen möglich.
Die zweite Strategie ist deutlich restriktiver aber ebenfalls auf `refinable` Methoden aufbauend. Anstelle der Frage nach einem boolwertigen Ergebnis kann eine einfache Abfrage nach einem `class` Objekt erfolgen. Die Zugehörigkeit von `thisJoinPointObject` zu der erfragten Klasse kann dann in `isApplicable` überprüft werden.
Ein dritter Weg ist durch das Verwenden von Initialisierungsmethoden anstelle von `refinables` gegeben. Eine Hook-Level Methode kann verwendet werden, um im Connector eine Klassenliste zu spezifizieren. Auch hier reicht dann eine Überprüfung in `isApplicable`, um `target` weitgehend zu ersetzen.

Allen diesen Ansätzen ist gemein, dass sie lediglich zur Umgehung von Eigenheiten und Lücken der vorliegenden Version JAsCos dienen: Unter Design- wie auch unter Performanzkriterien ist ein Einsatz in vielen, wenn auch nicht allen Szenarien eigentlich als wenig erstrebenswert zu sehen. Eine Neubewertung der Muster wird mit dem Erscheinen von JAsCo 0.9 nötig werden.

```
1  ...
   hook IsApplicableHook {
3    IsApplicableHook(...) {
       execution(...
5    }
     isApplicable(){
7      return connectorIsApplicable();
     }
9    public boolean refinable connectorIsApplicable();
   }
11 ...
```

Listing 5.5: Delegation der `isApplicable` Methode in den Connector.

```
1  ...
   aspects.MyBean.IsApplicableHook advice = new aspects.MyBean.
       IsApplicableHook(* *.*(*)){
```

[6] x < 8

```
3     public boolean connectorIsApplicable( ){
        return (thisJoinPointObject instanceof testbed.SubClass);
5     }
    }
7   ...
```

Listing 5.6: Connector zu Listing 5.5. Simuliert hier einen `target(testbed.SubClass)` Pointcutdesignator.

5.1.4 Parameterangaben im Hookkonstruktor

Ein Problem stellt sich bei *Advice-Code*, der auf Parameter zugrundeliegender *Joinpoints* angewiesen ist. JAsCo erlaubt die getypte Verwendung von solchen nur dann, wenn die Typen der Parameter bereits in der Definition des Hookkonstruktors fest angegeben werden. Da das die Zahl der auswählbaren Joinpoints einschränkt (siehe Abschnitt 5.1.1.2) und nicht redefiniert werden kann, ist die Anwendung in wiederverwendbaren Aspekten mitunter problematisch – insbesondere, da JAsCo kein Äquivalent von *Parametermappings* kennt (vergleiche Abschnitt 2.3.1.2). Für eine mögliche Wiederverwendung mag es angebracht sein, in den diesbezüglich relevanten *Hooks*, nur den `..args` Platzhalter für das statisch ungetypte *Array* der Methodenparameter zum Einsatz kommen zu lassen.
Um dennoch auf die Paramter zugreifen zu können, bieten `Refinable` Methoden zur Konvertierung der Methodenparameter einen Ausweg. Aufgerufen mit dem Parameterarray als Argument, können sie unter moderatem Aufwand *Parametermappings* realisieren.

5.2 Beispielhafte Ansätze und Idiome in ObjectTeams/Java

Im Anschluss an die Betrachtung von JAsCo, bietet dieser Abschnitt einige Beispiele der Idiomatik von ObjectTeams.

5.2.1 Gültige Basismethoden der Callin Bindung

Betrachtet man die Callin-Bindung im Zusammenhang mit der Bindung an eine Basisklasse, so fällt auf, dass diese Kombination als Auswahl eines Joinpoints gesehen werden kann.
Die Semantik ähnelt dabei einem auf eine Methode einer Klasse `execution` Ausdruck AspectJs, verhält sich aber bezüglich Vererbung eher wie `call` und damit deutlich intuitiver. Denn auch von Supertypen geerbte Methoden werden als gültige *Joinpoints* freigelegt. Gültig für *callins* sind alle für einen Basistyp deklarierten Methoden, nicht nur solche, die lexikalisch innerhalb der jeweiligen Basisklasse liegen . Ein Callin c einer Rolle R, gebunden an eine Basisklasse B, kann eine Methode m als Basis verwenden, wenn:

$$m \text{ deklariert ist in einer Klasse } K \wedge K \supseteq B$$

Dabei wird eine Bindung vererbt: Selbst wenn Methoden ohne `super` Aufruf redefiniert werden, bleibt die Bindung erhalten[7].

5.2.2 Automatisches Lifting vermeiden

Sind Aspekte nur für einzelne Basisinstanzen erforderlich, so bieten *Base Guards* die Möglichkeit, die automatische Erzeugung einer Rolle zu verhindern ([HHMW05]).
Sie sind eine hervorragende Grundlage, um Bindungen auf einzelne Instanzen der Basisklasse zu begrenzen oder um solchen Instanzen dynamisch verschiedene Rollenklassen zuzuordnen. Listing 5.7 zeigt eine Rollenklasse, für welche nicht automatisch Instanzen erzeugt werden. Mit Hilfe der ObjectTeams Reflection-Funktionen wird die Existenz einer Rolleninstanz überprüft. Die Bindungen werden nur aktiviert, wenn eine solche bereits existiert. So können Rollenklassen verfasst werden, deren Instanzen nur über Rollenkonstrukturen oder *declared Lifting* erzeugt werden können.

```
1  team class MyTeam {
2
3    public void createRole(BaseClass as Role newRole){}
4
5          public class Role playedBy BaseClass
6          base when(this.hasRole(base, Role.class))
7          {
8            ...
9          }
10         ...
11
   }
```

Listing 5.7: Automatische Erzeugung von Rollen verhindern. `createRole` fungiert als *Factory* Methode zur expliziten Rollenerzeugung.

Dieses Konzept kann verfeinert werden, um dynamisch die Rollenklasse der zu erzeugenden Rolleninstanz festzulegen. Dies kann als fortgeschrittene Version des *Factory Patterns* eingesetzt werden.

5.2.2.1 Konstanten mittels Parametermappings transportieren

Bei `before` und `after` Callins[8] kann ein *Parameter Mapping* verwendet werden, um bindungspezifische Konstanten in die Rollenmethoden zu transportieren. Dies ist eine sehr einfache Methode, um nicht unterstützte *Reflection*-Anweisungen zu simulieren, geht aber in der Mächtigkeit eher darüber hinaus. So können Parameter in Abhängigkeit von der aufgerufenen Methode ergänzt werden. Zudem wird ein sehr einfacher, deklarativer Programmierstil in Verbindungselementen ermöglicht.
Als Schattenseite ist zu erwähnen, dass diese Anwendung nicht gut geeignet ist, um mit `callin`-Methoden für `around` *Advice* zusammenzuarbeiten.

[7] Eine bemerkenswerte Ausnahme ist die implizite Vererbung (siehe: Abbildung 5.3).
[8] Die Anwendung bei `replace` Advice ist grundsätzlich möglich, leidet aber unter einigen Mängeln.

```
team class MyTeam {
void log( String str ) <- before void foo()
  with {
    str <- "foo"
  }
```

Listing 5.8: Parameter Mapping zur Weitergabe von bindungsabhängigen Konstanten.

5.2.3 Callins über mehrere Klassen/Objekte

Eine perspektivische Verwendung von geschachtelten Teams ist die Kommunikation „*über die Bande*“. Rollen können immer auf die *Features* ihrer einschließenden Teams zugreifen. Ist dieses Team selbst eine Rolle, so lassen sich dessen *callout* Methoden nutzen, um in die Basis zu kommunizieren.
Dies wird momentan noch durch die fehlende Unterstützung von *declared Lifting* für geschachtelte Teams behindert, verspricht aber sehr elegante *Observer* -Implementierungen (siehe Abbildung 5.1).
Aber auch heute schon ist so der Kunstgriff möglich, Callins auf mehr als ein Basisob-

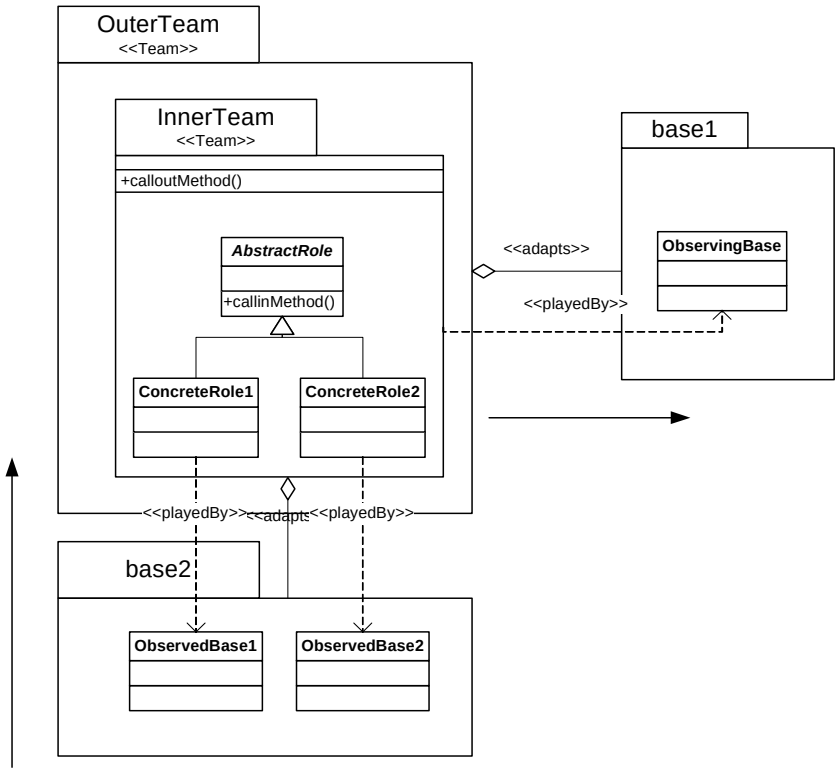

Abbildung 5.1: Über ein geschachteltes Team lassen sich *callins* auch von verschiedenen Basiobjekten aus verschiedenen Typen, bündeln. *Callouts* des Rollenteams können eine gänzlich andere Basis haben.

jekt und auf mehrere Basisklassen zu verteilen. Über gebundene Rollenklassen innerhalb des geschachtelten Teams können Instanzen dieses Teams – mittelbar – an verschiedene Basisklassen gebunden werden.

5.2.4 Aspekt-Basis Verbindungen

Ein besonderes Problem beim Entwurf wiederverwendbarer Aspekte ist das Ideal, die Basisanwendung möglichst vollständig von den Voraussetzungen der Aspekte zu trennen. Dafür liegt es nahe, eine hinreichend abstrakte Implementierung des Aspektes zu isolieren, die für sich genommen ohne jede Bindung an proprietäre Basisklassen auskommt, dann aber durch Verbindungselemente an ein konkretes Umfeld angepasst werden kann.

Allerdings hat solcher Verbindungscode auch eventuelle Verträge der Basis einzuhalten[9]. Umgekehrt verlangt dies nach genau spezifizierten Anforderungen für die erforderlichen Verbindungsstellen. Auch bei Object Teams ist es ein erstrebenswertes Designmerkmal, die Anbindung möglichst wiederverwendbarer Aspekte durch lose aber wohldefinierte Verbindungen zu gestalten.

5.2.5 Verbindung durch Vererbung

ObjectTeams sieht hierfür (z.B. [VH03]) die Verwendung konkreter Subteams von abstrakten Superteams vor. Dieser Ansatz ist nicht nur intuitiv einsetzbar. Er erlaubt auch eine saubere Trennung von Aspekt zu Basis, ohne auf Möglichkeiten der Anpassung zu verzichten. Die Mächtigkeit der kovarianten Vererbung von Rollenklassen kommt so voll zur Geltung.

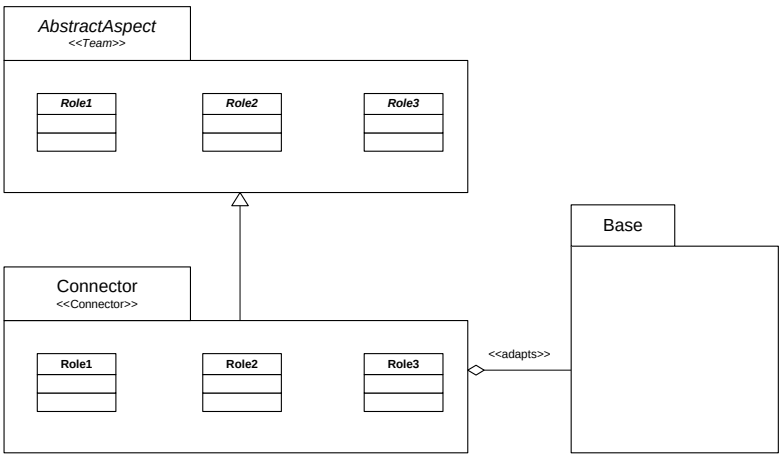

Abbildung 5.2: Die „klassische“ Verbindung in ObjectTeams: Ein abstraktes Team wird durch Vererbung an die Basis angepasst.

[9]Eine gerne übersehene Anforderung an wiederverwendbare Aspekte

Als fortgeschrittene Interpretation dieses Ansatzes kann ein Verbindungsteam mehrere Team-Rollenklassen enthalten, welche jeweils ein abstraktes Superteam konkretisieren. So kann diese Konstruktion verwendet werden, um mehrere Aspekte mit einer Basis und untereinander zu verbinden.

Allerdings hat der Ansatz einen, m.E. entscheidenden, Schwachpunkt: Teamvererbung führt laut OTJLD §1.3.1(d) [HH] *nicht* zu einer *Subtypbeziehung* der implizit geerbten Rollen. Daraus entsteht ein Problem, wenn Rollen des Superteams als Basisklassen eines anderen Teams verwendet werden: Diese Beziehung überträgt sich nicht auf konkrete Verbindungsteams. Daher schliesst dieser Verbindungsansatz direkte Aspekte auf Aspekte effektiv aus.

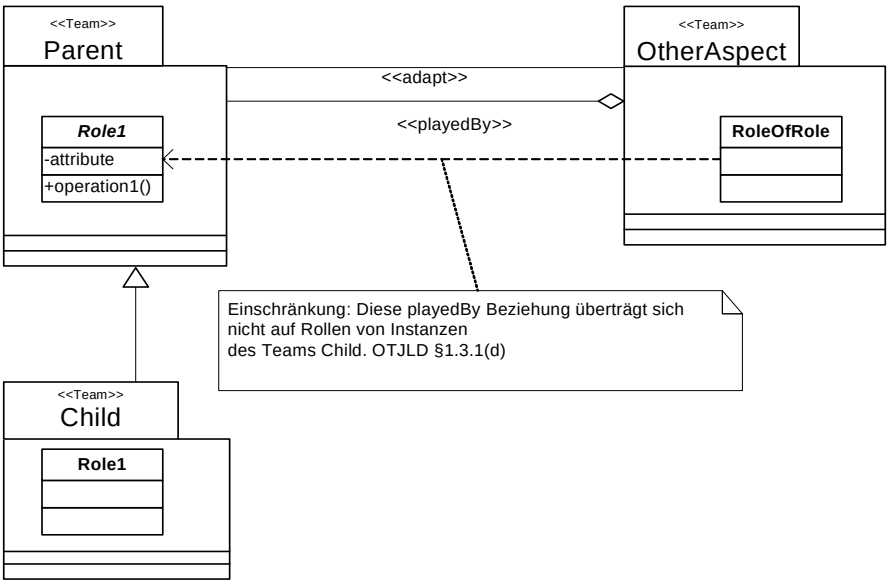

Abbildung 5.3: Grenzen der `playedBy` Bindung bei Rollen von Rollen.

5.2.6 Verbindung durch Marker

Eine Möglichkeit das Problem ins Leere laufender Bindungen zu umgehen, ist die Einführung von abstrakten Markerklassen mit Interface-Character – andere Aspekte können an diese generischen *Marker* gewoben werden. Diese Lösung ist inspiriert von der Idee der *Design Patterns*, durch welche die Erfahrung formuliert wurde, dass Probleme sich durch die Ergänzung einer Indirektion bewältigen lassen. Im Sinne dieser Überlegung können AOP-Bindungsmöglichkeiten eingesetzt werden, um mittels Indirektion einige Schwierigkeiten durch Markerklassen aus dem Weg zu räumen.

Markerklassen bezeichnet hierbei abstrakte Klassen, die als Supertyp für Rollenklassen

eingesetzt werden. An solche Markerklassen können Rollen problemlos gebunden werden. Auch ist eine solche Bindung an Markerklassen robust gegenüber Teamvererbung. Aber es schränkt die Teams dabei stark ein: der Zwang von einer Markerklasse zu erben, verstellt die Möglichkeit, explizit von anderen Klassen zu erben. Diese Einschränkung ist gegen die gewonnene Flexibilität abzuwägen.

Java Interfaces können diese Lücke augenblicklich noch nicht schließen, da die Verwendung von Interfaces als Basistyp in OT/J 0.8.17 noch nicht möglich ist. Sie ist aber für die nahe Zukunft angekündigt. Sobald diese Funktion implementiert ist, könnten Markerinterfaces eingesetzt werden, welche das Problem der in Java unmöglichen Mehrfachvererbung nicht teilen würden, das Hauptproblem bei der Verwendung von Markerklassen.

Eine mögliche Architektur auf Basis von Markerklassen sieht so aus, dass ein als abstrakte Verbindung fungierendes Team von Markerklassen erbende Rollenklassen enthält. Die generischen Aspektteams können konkret gehalten, ihre Rollenklassen an die Markerklassen gebunden werden. Konkret einsetzen lassen sich solche Konstruktionen durch konkrete Subteams der abstrakten Verbindungsteams.
Eine solcherart entworfene zweistufige Verbindung erlaubt die Verwendung von Aspekten auf Aspekte[10], ohne dass Mittel der Vererbung für die Rollen des konkreten Aspektes verhindert werden.

Jedenfalls ist auch die Einführung von Markerklassen an sich sinnvoll, würde ansonsten doch die Wartbarkeit von Verbindungselementen unter Umständen abnehmen. Dabei muss aber in Kauf genommen werden, dass abstrakte Markerklassen keine *callin*-Methoden enthalten können - dies hat die Begrenzung zur Folge, dass Aspekte nicht ohne weiteres an Callinmethoden der Verbindungsteams gebunden werden könnten; `Replace` *Advice* ist somit weitgehend ausgeschlossen.

Der Vorteil einer solchen zweistufigen Verbindung ist, dass eine stabile Verbindungsschicht zwischen Aspekten und Basis entstehen kann. Diese kann als stabile Schnittstelle zwischen Aspekt und Basis ausgestaltet werden. So können Verbindungen auch von verschiedenen – aber abhängig voneinander entworfenen – Aspekten genutzt werden und Evolution auf beiden Seiten standhalten. Dabei ist insbesondere zu bemerken, dass die eigentlichen Aspekte vollständig konkret und mit einem – zu einer abstrakten Markerklasse führenden – Basisbezug formuliert werden können. Dies öffnet Sprachmittel, wie etwa *Guard Predicates* oder *Declared Lifting* zur Verwendung in diesen Aspekten.

Die Konstruktion tauscht also den Vorteil der konkreten Arbeit auf der Aspektseite – mit Möglichkeiten wie *declared Lifting* – gegen den Verlust des direkten Basisbezugs.

Ob diese Vorteile allerdings den Verlust an Klarheit im Entwurf und die nicht unerheblichen technischen Begleiterscheinungen aufheben können, erscheint fraglich. Auch fällt

[10] Bindungen solcher Aspekte würden aber weiterhin bei Spezialisierungen des Basisaspekts brechen.

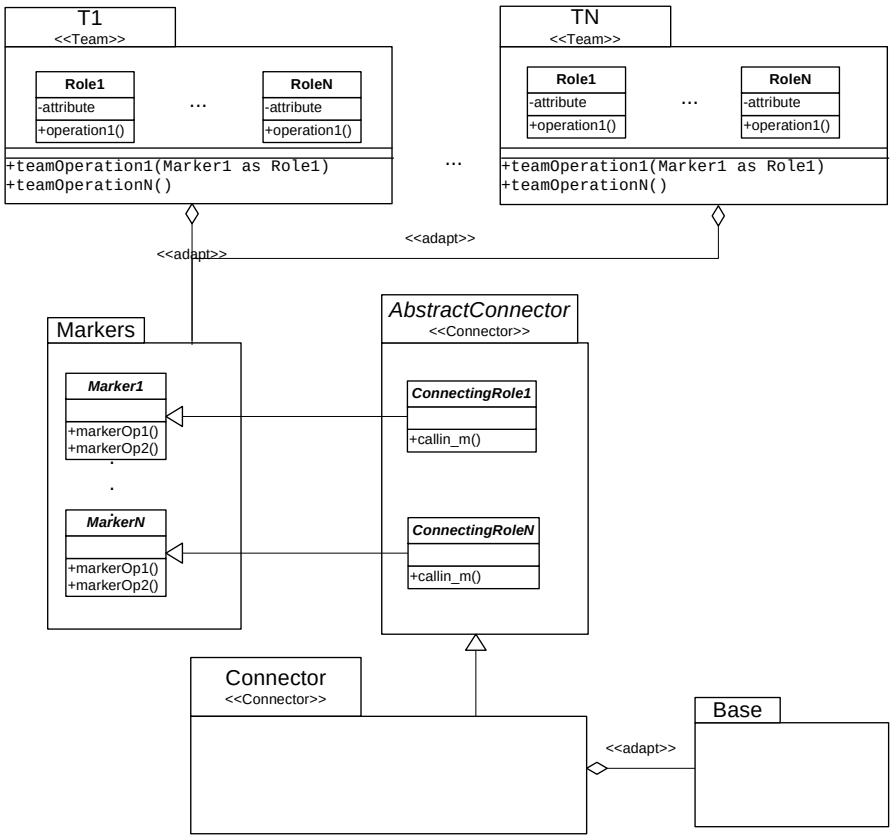

Abbildung 5.4: Ein komplexer zweistufiger Connector. Durch die Bindung über Marker, können wiederverwendbare Teams auf eine gleichbleibende Basis konkret aufgebaut werden. Die nötigen Abstraktionen verbleiben im Connector. Auch kann die basisabhängige Kommunikation der Aspekte durch die bekannten (Marker-) Basisklassen implementiert werden. Der Verlust an Klarheit und Basisbezug kann schwerer wiegen als die gewonnene Robustheit. Die Anpassung an einen konkreten Kontext entspricht weitestgehend derjenigen bei der Verwendung einfacher Teamvererbung.

die Typisierung der Rollen über ihr einschließendes Team weg. Das erschwert zielgerichtete Aspekte auf Rollen einzelner Teams.
Insofern stellt sich zudem die Frage, ob die Verwendung einer Verbindung durch mehrere Aspekte überhaupt als erstrebenswert aufgefasst werden kann, oder in sich bereits eine Vermischung verschiedener (Verbindungs-)Aspekte darstellt. Umgekehrt kann die Verteilung von Verbindungen auf ein Verbindungselement „pro-Team“ auch als *Scattering* angesehen werden.
Diese verteilte Verbindung wird erforderlich, da auch ObjectTeams keine Mehrfachvererbung kennt. Wird aber Vererbung zur Verbindung verwendet, so muss – offensichtlich – ein konkretes Team *erben*[11]. Das erbende Verbindungsglied kann prinzipiell auch eine Rolle mit Teamstatus sein, was aber noch mit technischen Beschränkungen einhergeht.

[11] Spitzfindig argumentiert sind durch abstrakte Vererbung zu verbindende Teams *white-box* Frameworks.

Zusammenfassend erfordert eine gemeinsame Nutzung von Markern durch verschiedene Aspekte, dass die Marker in Kenntnis all jener Aspekte entworfen wurden und die Aspekte unter der Kenntnis der Marker. Der Entwurfsaufwand steigt also überproportional und die Evolution könnte erschwert werden.
Daher scheint die Verbindung durch einfache Vererbung im Regelfall als die bessere Lösung, auch in der Inkarnation auf der Grundlage geschachtelter Teams.

6 Fallstudie

Der folgende Abschnitt stellt die eigentliche Fallstudie dar. Einleitend werden Ergebnisse der Analyse wieder gegeben. Diese beziehen sich auf den Funktionsumfangs der zu untersuchenden Aspekte und weiterführende Überlegungen. Dabei wird auch die Aufteilung der somit zu implementierenden Aspekte erläutert.
Anknüpfend werden die Entwurfs- und Implementierungsdetails der Fassungen für ObjectTeams und JAsCo knapp vorgestellt, allerdings geschieht dies nicht in einer direkt vergleichenden Weise.

Schließlich erfolgt eine Serie von Vergleichen, angelegt an architektonischen, wie auch an technischen Details. Besonderes Augenmerk wird unter anderem gelegt auf Koordination, Eignung zur weiteren Evolution, Anbindung der Basis, Dynamik und Nutzen experimenteller *Sprachfeatures*.

Inhaltsangabe

6.1 Vorstellung der Aspekte

Die ausgewählten Funktionen und Erläuterungen der aspektorientierten Interpretation werden in diesem Abschnitt vorgestellt. Die ausgewählten Aspekte umfassen dabei *Sessionmanagement*, Authentikation und Autorisation, wobei ein klarer Schwerpunkt auf dem gründlicher betrachteten *Sessionamanagement* liegt.
Für diese Aspekte wird das Ziel einer a-posteriori Integration verfolgt, was gerade im Zusammenhang mit Sicherheitsfunktionen nicht praxisfern ist.

6.1.1 „Sessionmanagement“

Der Begriff *Sessionmanagement* oder auch *SessionTracking* ist in heutigen Projekten sehr häufig anzutreffen, wenn auch selten mit einer klaren Beschreibung über sein Wesen. Oft verbirgt sich hinter dem Begriff eine proprietäre Technik, um eigentlich zustandslose Protokolle, wie HTTP, mit einem Zustand zu versehen. Es wird also eingesetzt, um Informationen bezogen auf einen gegenwärtigen Nutzer der Anwendung von einer Aktion zur nächsten zu transportieren.
Dafür bieten alle gebräuchlichen Frameworks zur Entwicklung von Webanwendungen mittlerweile weitreichende Unterstützung, um *Sessionstracking* in solchen Anwendungen zu implementieren. Obgleich auch das Gebiet von Webanwendungen sehr interessant für eine aspektorientierten Implementierung wäre, so bestehen doch technische Schranken im Weg der Umsetzung einer solchen Studie [1].

Eine durchaus aussagekräftige Teilmenge der Domäne ist die Ergänzung von Sessions in einer lokalen Anwendung, etwa um Rechtevergabe oder Überwachung umzusetzen. Perspektivisch wäre der Einsatz auch in verteilten Anwendungen mit paralleler Nutzung interessant. Für den Zweck dieser Studie wird die Domäne aber auf isolierte Anwendungen beschränkt.

Zunächst stellt sich die Frage nach dem Begriff *Session*, wie auch nach den damit assoziierten Funktionen.
Die existierenden *Sessionmanagement* Lösungen fassen üblicherweise unter diesen Begriff eine *Dictionary*-Datenstruktur, die Daten für den gegenwärtigen Nutzungskontext isoliert speichert und bereitstellt. Wichtig dabei ist die Möglichkeit, Daten im aktuellen Nutzungskontext zu speichern, wobei die Zuordnung des Kontextes zum Aufruf, das *Tracking*, in gewisser Weise teilautomatisiert stattfindet. Dafür wird üblicherweise ein Schlüsselwert zur Identifizierung implizit oder explizit durch den Kontrollfluss mittransportiert, um die Eingaben eines Nutzers zu verfolgen.

[1] Hier wäre zunächst die Lauffähigkeit in *Enterprise Application Containern* zu nennen. JAsCo erlaubt zwar unter Verzicht auf einige Funktionen die Ausführung in einem *Application Container*: Ein Vergleich wäre in dieser Umgebung aber nicht möglich. Ferner erfordert HTML/HTTP *Session Tracking* einige Aktionen, die momentan noch schlecht a-posteriori umsetzbar sind - etwa Link-Rewriting.

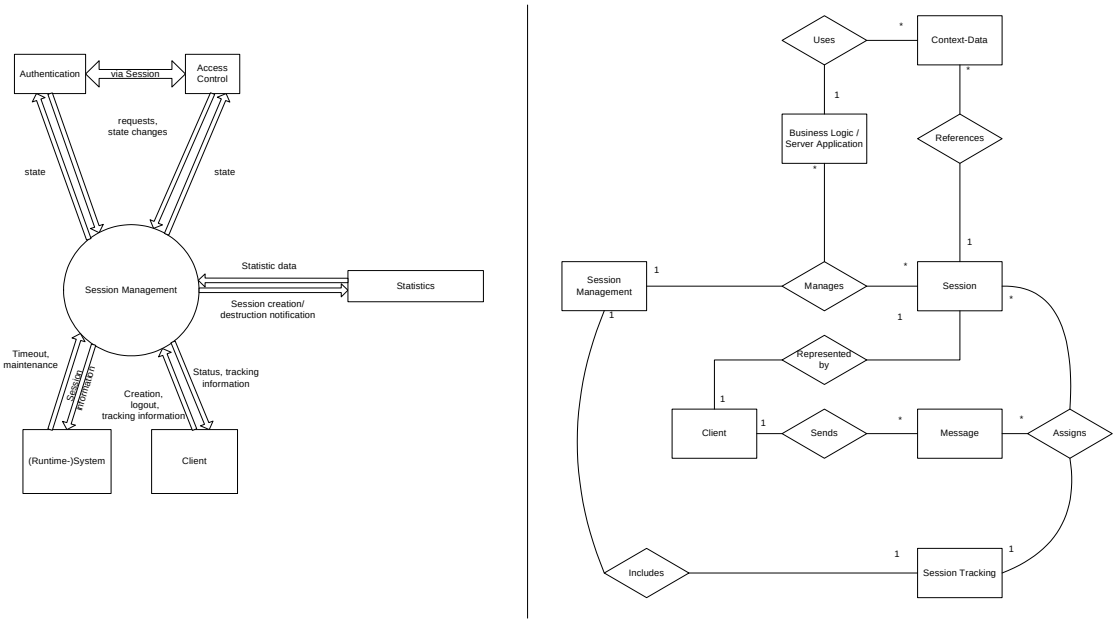

Abbildung 6.1: Beispielhafter Auszug aus den Ergebnissen der reduzierten FODA: Kommunikations- und E/R-Diagramm.

Es lassen sich also folgende *Features* für *Sessions* ohne Betrachtung von *Sessiontracking* aufzeigen:

- **Daten speichern** – Es sollte möglich sein, Daten in der aktuellen Session abzulegen, ohne im Vorhinein zu wissen, welche Session die aktuelle ist.
- **Daten abrufen** – Umgekehrt soll der Abruf dieser Daten wiederum möglich sein, allerdings nur aus derjenigen Session heraus, in welcher auch der ursprüngliche Eintrag geschah.

Dieses Doppel, das sich sehr an bestehenden Lösungen ausgerichtet, sollte aber für die Möglichkeiten der Aspektorientierung deutlich erweitert werden: Stellt die klassische Sessionperspektive nur den Zugriff auf sessionabhängige Datenstrukturen zur Verfügung, so können Aspekte die vorhandene Datenstruktur und das Programmverhalten der gegenwärtigen Session gemäß anpassen. Sie können also Einfluss auf das Programm ausüben:

- **Kontext verwalten** – Sessions sollten den abhängigen Teil des Programms in einer privaten Version verwalten können.
- **Kontext entfernen** – Folglich sollte sich auch der sessionabhängige Teil dynamisch aus dem Programm entfernen lassen.
- **Kontext „*ausrollen*“** – Und schließlich sollte der Kontext aktiv in das laufende Programm eingebracht werden können.

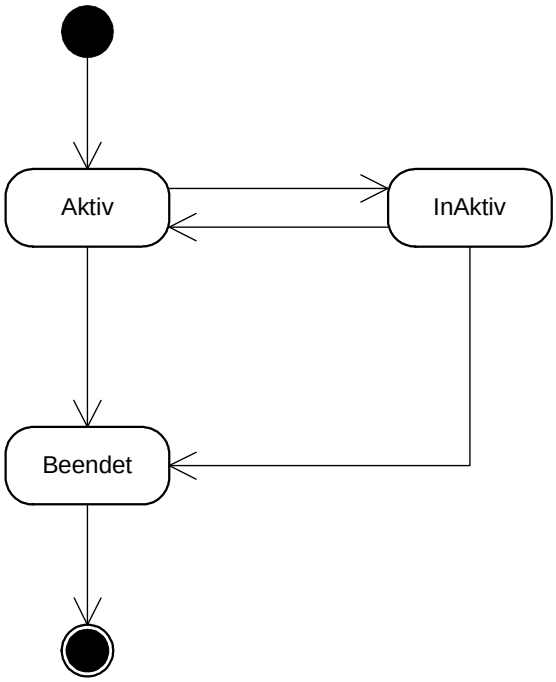

Abbildung 6.2: Zustände im Lifecycle von Session Instanzen.

Die ebenfalls häufig anzutreffende Funktion der Nutzer- und Nutzungsstatistiken wurde ignoriert, sollte aber als zusätzlicher Aspekt leicht umsetzbar sein.
Nutzungskontexte haben ferner einen *Lifecycle*; zumindest einen Anfang und ein Ende, was sie (noch) nicht von anderen Objekten differenzieren würde. Im spezifischen Fall scheinen vier Zustände die Möglichkeiten weitgehend zu beschreiben:

- **Nicht Existent** – Die Abwesenheit einer Instanz; sowohl Anfangs- als auch Endzustand.
- **Aktiv** – Die gegenwärtig aktive Session; in jedem Ablauf sollte nur maximal eine Session diesen Status tragen. Nur in diesem Zustand können Daten gelesen oder geschrieben werden; der gespeicherte Kontext ist identisch mit dem des Programms.
- **Inaktiv** – Die *Session* existiert, ist aber verborgen.
- **Beendet** – Finaler Zustand; Entfernung der *Session* aus dem Programm.

6.1.2 Authentikation und Autorisation

Ein nahe liegendes Anwendungsgebiet für *Sessions* ist das Gebiet der Zugangsbeschränkung und Nutzerauthentisierung. In Programmen ist nicht zuletzt die Rechteverwaltung oft der Anlass für tief greifende Erweiterungen. Wegen der weitgehenden Maßnahmen, die

derartige sicherheitsrelevante Systeme erfordern, ist eine grundsätzliche Neuimplementierung weit jenseits des Anspruchs dieser Arbeit. Das Wesen der Untersuchungsgegenstände würde sie auch überhaupt nicht erlauben.
Daher fiel der Blick auf ein bereits vorliegendes Grundgerüst: Java bietet seit Version 1.4 der J2SE den *Java Authentication and Authorization Service* – kurz *JAAS* – an, um Authentisierung und Autorisierung umzusetzen. Dieses *Framework* baut auf dem Java *Security Manager* auf und ist hochgradig anpassbar.
JAAS' zentrale Datenstruktur ist das `Subject`, in gewisser Hinsicht der Schlüsselring eines Nutzers. `Subject`s sammeln `Principals` , die ihnen vom System zugeteilt werden. Diese `Principals` wiederum können mit Genehmigungen verknüpft werden, was eine flexible Rechtestruktur ermöglicht.
Eine Einführung in die Verwendung und Erweiterung von JAAS findet sich etwa in [Cot05].

6.1.2.1 Authentikation

Authentikation in JAAS ist bereits hochgradig variabel. Im Idealfall kann durch einfaches Ändern der Konfiguration die Art der Authentikation verändert werden (Passwort, Nutzername, Smartcard, Kereberos,...). Der eigentliche Authentikationsvorgang wird zur Laufzeit durch einen Aufruf der Login Methode der Klasse `LoginContext` ausgelöst; das als Ergebnis erhaltene `Subject` muss dann programmatisch verwaltet werden.
Hier liegt die offensichtliche Schnittmenge zum *Sessionmanagement*: Eine *Session* scheint die naheliegende Struktur zu sein, um ein `Subject` zu verwalten. Einen direkten Vorteil bietet die Implementierung als Aspekt in ihrer Dynamik: Ort und Bedingung des Logins können dynamisch verschoben werden und sind nicht auf einen beliebigen Punkt festgelegt; eine bessere *Separation of Concerns* scheint somit realistisch. Als wichtige Funktionen wurden zunächst drei identifiziert;

- **Login** – Erzeugung einer `Subject` Instanz zu einem beliebigen Zeitpunkt im Programmablauf.
- **Login überprüfen** – Einen Nutzer zur erneuten Authentikation auffordern.
- **Einfacher Test** – Feststellung des Logins, ohne das Ergebnis zu speichern.

.

6.1.2.2 Autorisation

Die Autorisation über JAAS basiert auf *Permissions*, Genehmigungen auf eine Ressource zuzugreifen. Solche Genehmigungen können auf verschiedenen Kriterien basieren. Beispielsweise kommen sowohl dynamische Bedingungen wie die Rechte eines Nutzers, als auch strukturelle Bedingungen wie die Herkunft des aufrufenden Codes in Frage.
Hier ist in erster Linie die Überprüfung der `Principals` eines `Subjects` hinsichtlich

einer bestimmten Genehmigung, `Permission` von Belang. Java sieht bereits in der Standardkonfiguration eine Reihe von Überprüfungen vor; etwa bei dem Zugriff auf Dateien. Sollen aber noch weitere Überprüfungen eingefügt werden, erfordert dies den Aufruf der `checkPermission` Methode aus der Klasse `AccessController`. Diese verwaltet den aktuellen Sicherheitskontext, der jedoch zunächst nicht auf den Rechten eines programmatisch verwalteten Nutzers beruht. Um dies zu erreichen, dient die Ausführung von `PrivilegedActions`[2]: Dem *Java* Ansatz für *Threads* folgend, beinhaltet das *Interface* `PrivelegedAction` eine `run` Methode. In implementierenden Klassen wird der Code in dieser Methode mittels `doAsPriveleged` im Kontext eines dynamisch bestimmbaren `Subject`s ausgeführt.
So bieten sich drei *Concerns*, die bei JAAS-basierenden Programmen durch den normalen Code verteilt werden würden:

- **Ermittlung des richtigen Subjects** – Um die Genehmigungen eines `Subject`s zu überprüfen, muss dieses auch an den relevanten Stellen vorliegen. Daher ist es nötig, das *richtige* `Subject` referenzieren zu können. Hier liegt die offensichtliche Verbindung zu *Sessionmanagement*.
- **Ausführen in einem Kontext** – Wenn ein Aufruf genehmigungsrelevante Operationen enthält, so muss er in den passenden Sicherheitskontext verlegt werden. Dieses Auslagern von Methodenaufrufen zu `PrivelegedAction`-Klassen ist ein weiterer Aspekt.
- **Einfügen von Einschränkungen** – Soll der Zugriff auf eine Ressource beschränkt werden, die nicht von der normalen Java Architektur bereits beschränkt wird, so ist die Ergänzung von `checkPermission` Aufrufen erforderlich. Das Einfügen von solchen Schranken sollte sich sehr viel eleganter aspektorientiert verwirklichen lassen als durch direktes Einfügen in den zu beschränkenden Code.

6.2 Entwurf

Dieser Abschnitt stellt die detaillierten Entwürfe kurz dar, um als Grundlage für den nachfolgenden Vergleich zu dienen.

6.2.1 ObjectTeams

Als Notation für die ObjectTeams Klassenmodelle wurde eine leicht abgewandelte Form der UFA gewählt.

Der Entwurf von kooperativen aber unabhängigen Aspekten in *ObjectTeams* erfordert zunächst einige Überlegungen hinsichtlich der Grundstruktur der *Teams*. Nur innerhalb eines *Teams* liegende Rollen können wirklich gleichberechtigt miteinander kommunizieren. Die kovariante Vererbung garantiert die Aufrechterhaltung solcher Beziehungen in Spezialisierungen des Teams. Von dieser Seite aus gesehen, scheint der Einsatz besonders

[2] Für `Exception`-werfende Aktionen existiert zusätzlich `PrivilegedExceptionAction`.

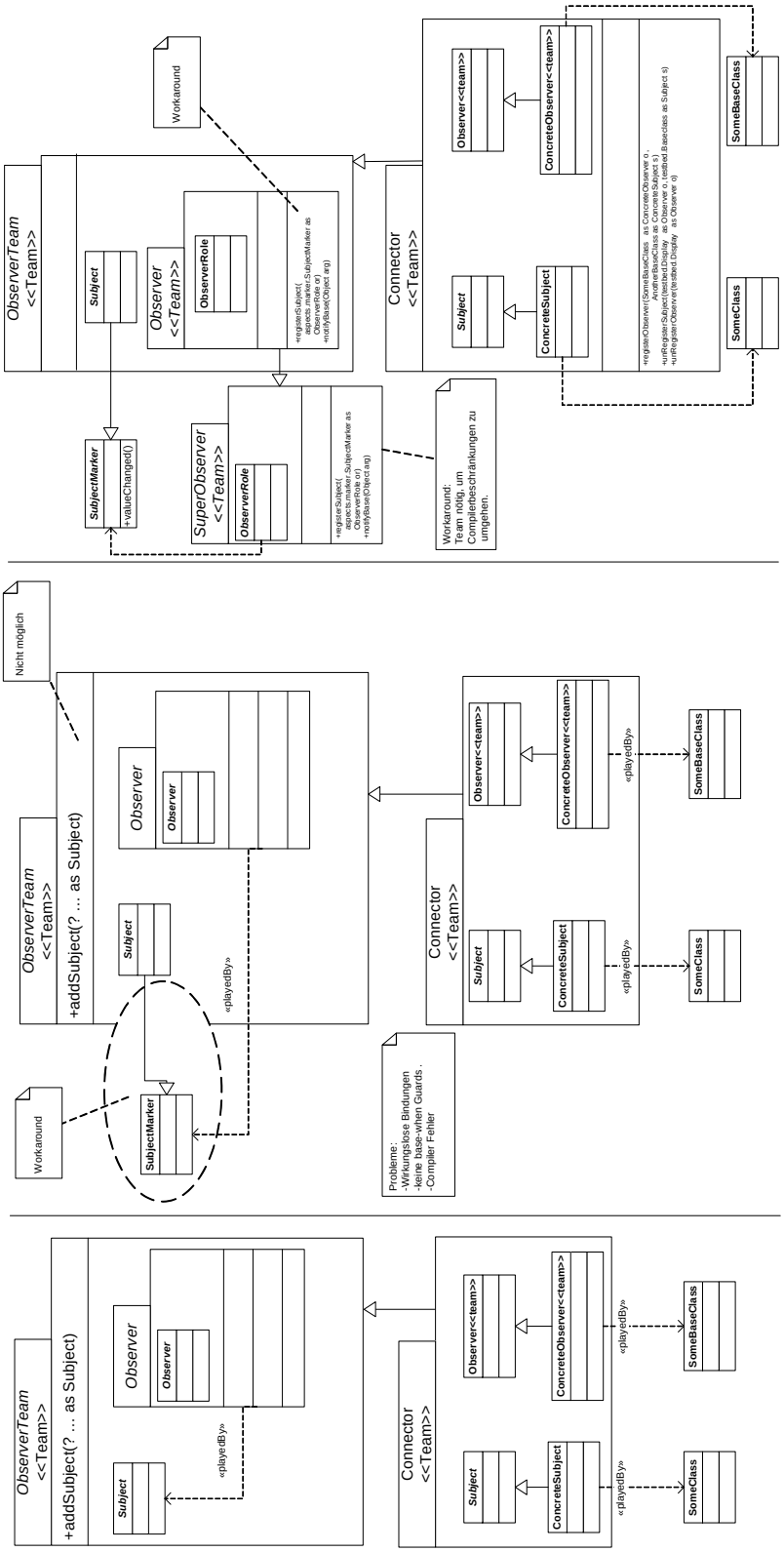

Abbildung 6.3: Beispiel hinsichtlich der Entwicklung von Entwürfen in ObjectTeams: Eine ungewöhnliche Variante des *Observer-patterns*. Der ursprüngliche Entwurf (links) musste weitreichend angepasst werden, um zu einer funktionsfähigen Implementierung (rechts) zu gelangen. Diese Architektur war zwar ursprünglich vorgesehen, wurde dann aber als zu schlecht umsetzbar abgetan.

großer, monolithischer Teams als Aspektkomponenten geboten. Dem entgegen steht die Unmöglichkeit, aus dem Team „*auszubrechen*“. Rollenklassen können nur mit ihrem Team spezialisiert werden, eine Spezialisierung orthogonal zu der des Teams wäre nicht möglich. Darüber hinaus würde die Aufrechterhaltung einer Schnittstelle für andere Aspekte problematisch werden.

Daher fiel die Entscheidung, die Aspekte einer Architektur mit einer expliziten Schnittstelle für andere Aspekte einzusetzen (siehe Abbildung 6.4). Diese Schnittstelle wird durch eine abstrakte Klasse vertreten, an welche eine kontrollierende, konkrete Rolle gebunden wird. *Kontrollierend* soll hierbei die Eigenschaften bezeichnen, dass zum Einen alle relevanten Funktionen des Teams lediglich durch Methoden der Rolle ausgelöst werden können, ohne jemals eine Instanz einer solchen Rolle zu externalisieren. Zum Anderen sollten diese Funktionen anderen Aspekten erlauben, die Ausführung des *Advices* des Teams zu unterbinden. Dies kombiniert zwei bereits formulierte Überlegungen: Die in Abschnitt 5.2.6 eingeführte Lösung, *Markerklassen* zur Verbindung von Rollen einzusetzen, wird hier verknüpft mit der Forderung nach aspektorientierten Interfaces für Aspekte aus 3.3.1.

Idealerweise bündeln diese Klassen alle Kommunikation *in* das Team hinein, erfüllen also auch die Aufgabe von *Pointcuts* in anderen Sprachen; insbesondere die Möglichkeit andere Aspekte abhängig zu weben.
Andererseits bieten sie aber auch die gesuchte Möglichkeit orthogonaler Spezialisierung: Die „arbeitenden“ Rollen, also jene die auf die Kommunikation nach Außen/zur Basis ausgerichtet sind, können kovariant spezialisiert werden, ohne dies für die oftmals gänzlich anders gelagerte Aufgabe der kontrollierenden Rolle zu erfordern.
Abrundend erlauben sie eine sehr flexible Anwendung des Aspektes: Die Funktionalität kann sowohl in der Manier eines *WhiteBox Frameworks* geerbt als auch durch aspektorientierte Bindung eingewoben werden. Dies führt zu einer interessanten Besonderheit: Der so entworfene Aspekt kann auch – scheinbar – objektorientiert eingesetzt werden.

6.2.1.1 Sessionmanagement

Der Entwurf des *Sessionmanagement* Aspektes ist das Produkt einer graduellen Entwicklung, bedingt durch technische Grenzen; daher wird die Genese schrittweise vorgestellt.

6.2.1.2 Ansatz: Sessions gleichberechtigt gegenüber ihrem Kontext

Der Überlegung folgend, dass *Sessiontracking* und *Sessions* abhängige aber verschiedene Teile von *Sessionmanagement* sind, wurde die Trennung dieser Belange in verschiedene Module gewählt. Gleichzeitig, begründet in den Abhängigkeiten beider Einheiten, sollten sie als Rollenklassen innerhalb eines Teams implementiert werden.
Genauer handelt es sich also um eine Rolle `Session` zur Verwaltung von sessionabhängigen Daten sowie eine Rolle `SessionTracker` zur Steuerung der globalen Funktionen wie Starten oder Beenden einer Session.

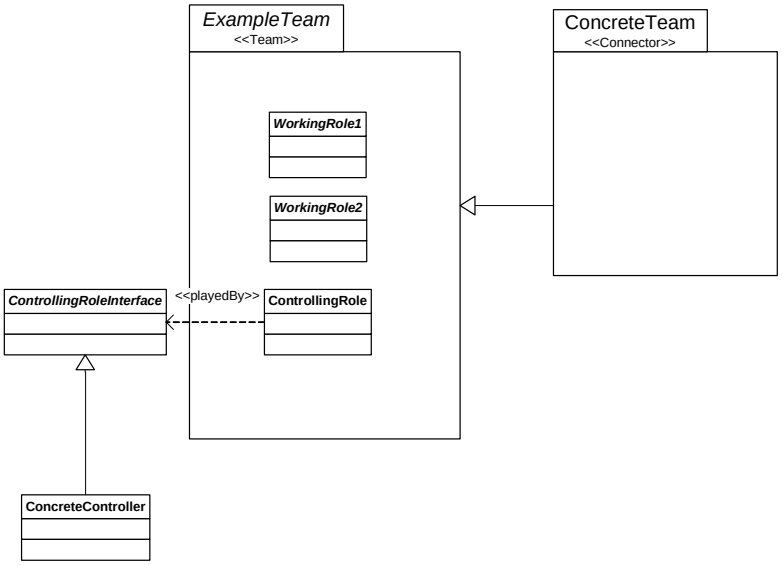

Abbildung 6.4: ObjectTeams: Beispiel für die gewählte Architektur. Durch die konkrete Bindung zwischen `ControllingRole` und `ControllingRoleInterface` verhalten sich Implementierungen von `ControllingRoleInterface` wie Spezialisierungen von `ControllingRole`. Dies ermöglicht eine orthogonale Vererbung und stellt einen Pointcut-ähnlichen Mechanismus zur Verfügung. Zusätzlich erlaubt es die Einbindung sowohl im Stile eines klassischen Frameworks als auch aspektorientiert.

Die grundlegende Idee der Sessionrolle soll dabei ein *Observerpattern* sein: Eine `Session` verwaltet eine Reihe von `ContextElements` und informiert sie über ihre Zustandsänderungen. Dabei wird die Sessionrollenklasse an eine leere `KeyMarker` Klasse gebunden, deren Instanzen durch die Verbindungsschicht dem *Session Tracking* Mechanismus übergeben werden. Über diese Aufteilung kann *Lifting* verwendet werden, um aus einem anonymen `Key`-Objekt die richtige `Session`-Instanz zu ermitteln.
`KeyMarker` ist hierbei ein Hilfselement, das als abstrakter Platzhalter für – in Verbindungscode festzulegenden – Klassen der Basisanwendung dienen soll. Mögliche konkrete Basisobjekte, mittelbar über Rollen eingebunden, sollen eine Session in der Basisanwendung vertreten. Sie dienen als Schlüssel zur Ermittlung der richtigen `Session` (Rollen-)Instanz. Mögliche Beispiele wären etwa *Tabs* einer `JTabbedPane`, Datenflussobjekte oder auch neu eingeführte triviale Strukturen.

Da *Session Tracking* unter Umständen einen gänzlich anderen Basisbezug haben kann als *Sessions* und deren Kontext, ist hier die Auslagerung entsprechend der zuvor beschriebenen Lösung mit einer *kontrollierenden* Rolle (siehe Abschnitt 6.2.1) angebracht. Diese *kontrollierende Rolle* enthält daher Funktionen zum Starten, Beenden, Wechseln, und Sperren der aktuellen *Session*. Andere Funktionen mit einer Abhängigkeit von der gegenwärtigen Session werden dem gegenüber nur als Team-Methoden berücksichtigt, insbesondere die Registrierung von `ContextElement` Instanzen. Die Rationale hierfür liegt im Wesen dieser Rollen: Erzeugung und Verwaltung von Kontext sind nicht Aufgabe von

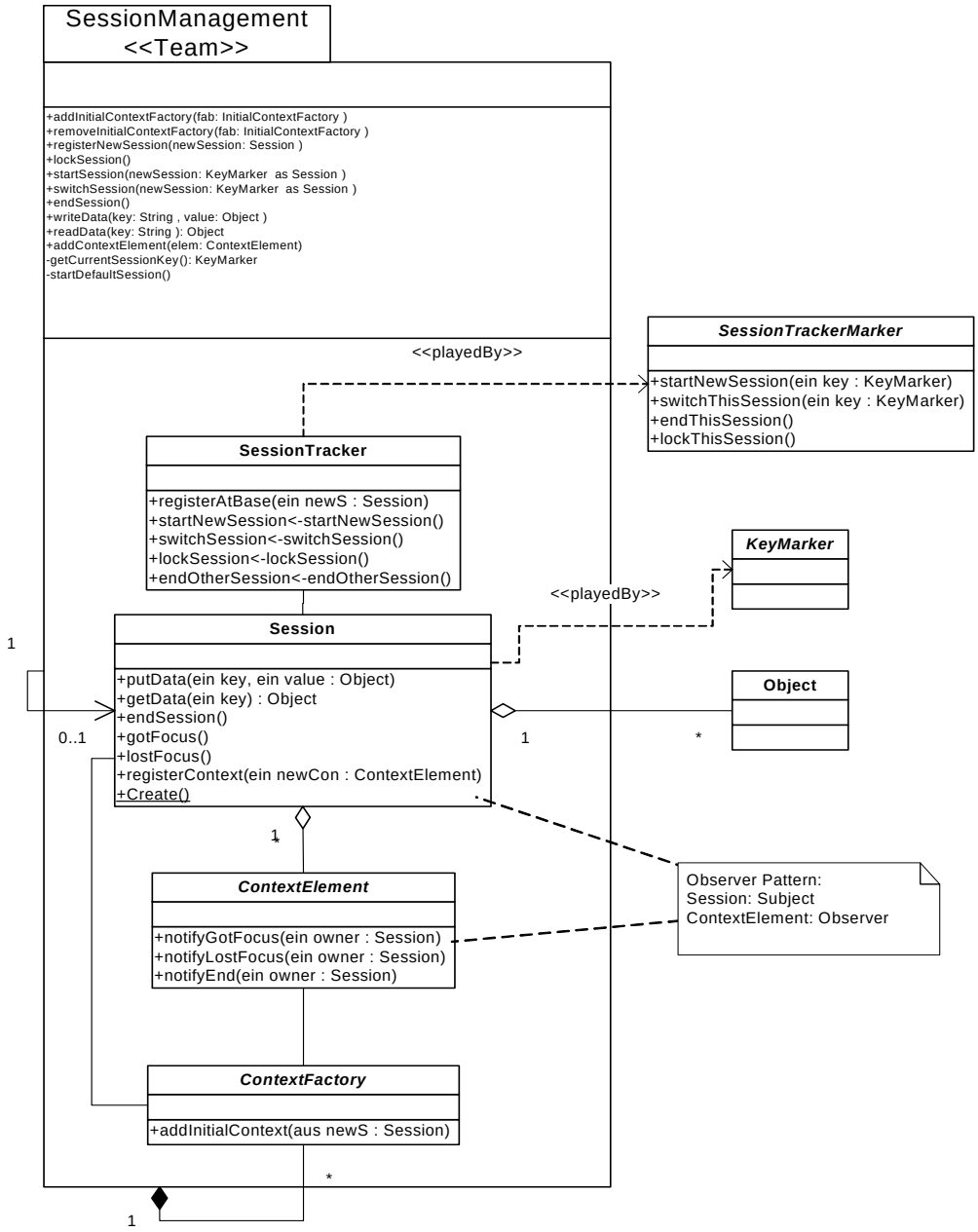

Abbildung 6.5: ObjectTeams: Erster Entwurf für Sessions in OT. Technisch gut umsetzbare, aber wenig befriedigende Modellierung von Sessions.

`SessionTracking`. Um diese Lücke zu füllen, dient eine weitere Rolle, `ContextFactory`, deren Implementierungen neu gestarteten `Session`s einen initialen Kontext zuordnen sollen. Art und Umfang dieser initialen Austattung von `Session` Instanzen obliegt der konkreten Verbindung.

Die Umsetzung dieser `ContextElement`s lässt jedoch Raum für Überlegungen: `Context-Element`-Instanzen sollen in der Lage sein, bei einem Wechsel der Session den Zustand ihrer Basisklasse datengemäß anzupassen. Damit sind sie jedenfalls basisabhängig und können erst in einem konkreten *Connectorteam* definiert werden, aber als Verbindung zu dem Kontext kann ein *Observer Pattern* bereits angegeben werden.

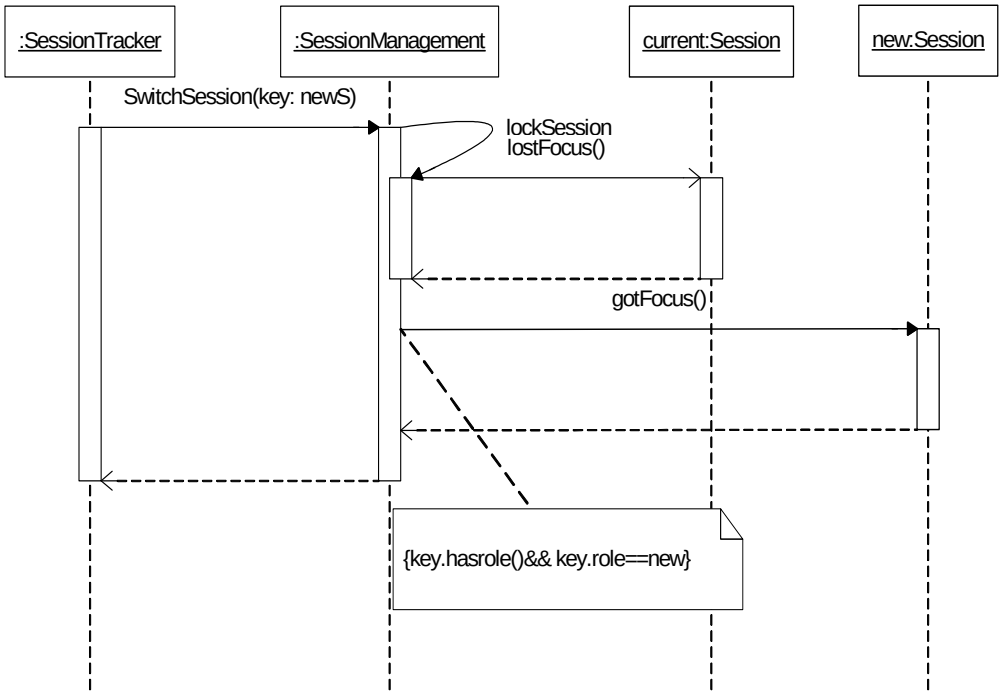

Abbildung 6.6: ObjectTeams: Der Grundgedanke beim Wechsel einer Session. Gültig für alle Entwürfe.

Ob jedoch `ContextElement` eine *Rollenklasse* auf der gleichen Ebene wie `Session` sein sollte, ist noch unbeantwortet. Für die Ansiedlung auf der gleichen Ebene spräche die einfache technische Umsetzung. Der entscheidende Schwachpunkt dieses Entwurfs liegt in der Gleichberechtigung von `Sessions` und `ContextElement`s: Da `ContextElement` auf der gleichen Ebene wie auch `Session` lägen, ist nur die Menge der registrierten `ContextElement` Instanzen *sessionabhängig*. Alle zustandsrelevanten Daten müssten aus der Session ausgelesen werden[3] (siehe: Abbildung 6.7), was die Möglichkeiten des Teamkontextes weitgehend ungenutzt lassen würde.

[3]Nicht unähnlich der Verwendung von Sessions bei J2EE Servlets oder PHP.

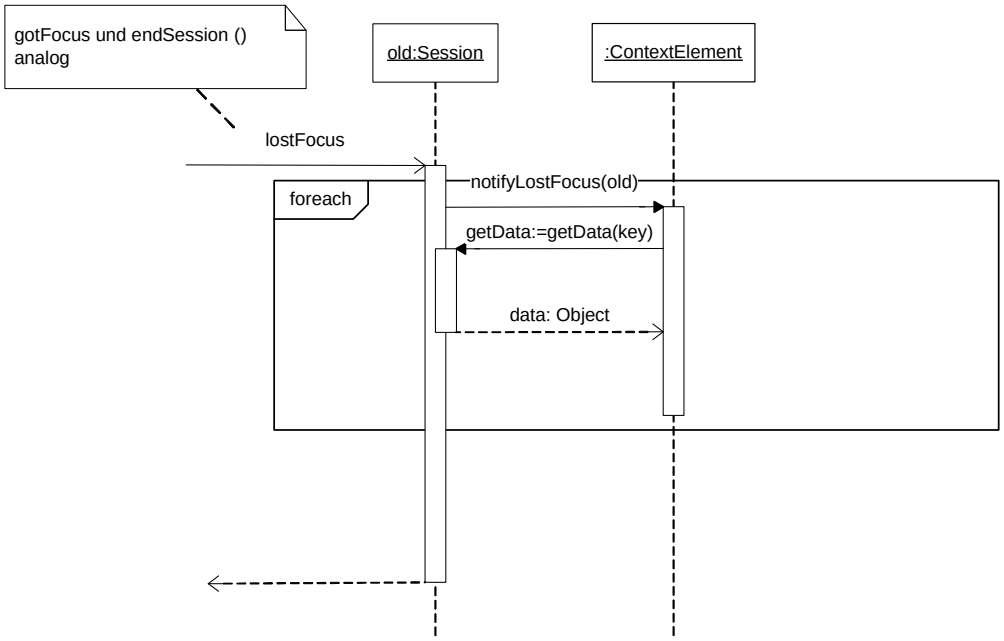

Abbildung 6.7: ObjectTeams: Da in diesem Entwurf `ContextElement` Rollen des gleichen Teams wie `Session` sind, müssten sessionabhängige Daten aus der Sessioninstanz ausgelesen werden.

6.2.1.3 Ansatz: Sessions als (nested) Team enthalten ihren Kontext

Als Folge dieser Überlegung ist ein eigenes Team für die Umsetzung von `Session` als die bessere Möglichkeit zu erachten.
Dies wird aber zu einem gewissen Anteil durch die Sprache *ObjectTeams* erschwert und eröffnet einige neue Probleme. Zunächst wäre die Implementierung von `ContextElement` als Rollenklasse des Teams `Session` zu nennen. Oberflächlich scheint dies nahe zu liegen, erlauben doch die Teaminstanzen eine geschlossene Perspektive auf die Basis – somit wären die sessionabhängigen Daten gut zu verwalten.
Auf der Kehrseite muss jedoch bedacht werden, dass ObjectTeams keine hinreichend flexible Möglichkeit bietet, um ein generisches *declared Lifting* zu formulieren. Also, keine Möglichkeit bereits in abstrakten Teams die Schnittstelle der Teams vollständig vorzugeben. Dies beruht auf der Notwendigkeit, sowohl Basisklasse, als auch Rollenklasse bereits bei der Deklaration von *Declared-Lifting* Methoden anzugeben. Ein Aufschieben in eine konkrete Implementierung ist ohne Aussage über die Basisklasse nicht möglich. Erschwerend kommt hinzu, dass für Rollen von geschachtelten Teams nur geerbte *declared Lifting* Ausdrücke zulässig sind.
Somit stellt sich bei der ausschließlichen Umsetzung von abstrakten Kontextdaten als Rollen eines *nested Teams* ein Zuordnungsproblem: Wie können einzelne Objekte der Basisanwendung in `Session`-Instanzen als Rollen registriert werden?

Als Lösung bleibt der Ausweg über eine Indirektion erwägenswert: Eine Schnittstelle zur Behandlung von Kontextelementen, eine Umsetzung als Rolle verbleibt als Aufgabe für die konkrete Verbindungsschicht. Das damit verbundene schlechtere *Confinement* der sessionabhängigen Daten ist in Kauf zu nehmen.

Letztlich stellte sich heraus, dass die Einschränkungen für *nested Teams* zu groß sind, um diesen Weg geschachtelter Teams direkt beschreiten zu können. Die daher schlussendlich gewählte Richtung verwirklicht `Session` als ein eigenes, abstraktes Team. Dies erlaubt es dem eigentlichen `SessionMangement` Team bereits auf der obersten Ebene, als konkrete Teamklasse zu existieren. Umgekehrt fällt allerdings die Möglichkeit weg, den *Lifting*-Mechanismus zur Ermittlung der aktuell aktiven `Session` zu verwenden.
Daher muss eine zusätzliche Struktur implementiert werden, die diese Zuordnung übernimmt, d.h. abhängig von einem übergebenen Schlüsselobjekt entweder eine bestehende `Session`-Instanz auswählt oder eine neue erzeugt.
Für den letzteren Fall besteht die Notwendigkeit weitere Details zu berücksichtigen. Denn der Entwurf sieht nunmehr zwei Teams vor, damit fällt die Möglichkeit, die Teamvererbung zur kovarianten Anpassung an die Basis zu verwenden, weg. Die nötige enge Kopplung erfordert aber die Möglichkeit `Session` spezialisieren zu können, ohne die Bindung der Teamklasse `SessionManagement` an `Session` zu zerbrechen.

Die gewählte Auflösung des Widerspruchs ist durch eine Verwendung des *Abstract-Factory Patterns* (siehe [GHJV95]) verwirklicht, was eine flexible Spezialisierung von `Session` ohne Auswirkungen auf `SessionManagement` erlaubt. So kann Session unter gänzlich anderen Gesichtspunkten an die Basis angepasst werden als die Kontrolllogik.

6.2.1.4 Authentikation

Die Umsetzung der Anforderungen für Authentikation ist vergleichsweise geradlinig. Ein zentrales Team, `LoginTeam`, setzt die geforderten Funktionen als Methoden um. Ferner enthält dieses Team eine konkrete Rolle, welche mit dem bereits in Abschnitt 6.2.1 vorgestelltem Muster an eine abstrakte (Marker-)Klasse gebunden wird. Diese abstrakte Klasse stellt die Schnittstelle sowohl für die Basis wie auch für andere Aspekte zur Verfügung.
Als klassische Schnittstelle umgesetzt ist `SubjectHandler` eine Struktur, über die `Subject` Instanzen verwaltet werden. Abrundend erlaubt die `LoginExceptionHandler` Schnittstelle, Fehlschläge bei Loginvorgängen zu behandeln.
Darüber hinaus wurden einige Hilfsklassen ergänzt, die jedoch mehr zur Vereinfachung des Einsatzes gedacht sind, als dass sie eine tragende Aufgabe wahrzunehmen hätten.

6.2.1.5 Autorisierung

Der Entwurf mit ObjectTeams ist in vielerlei Hinsicht von technischen Erwägungen geprägt, gelegentlich stärker als dies wünschenswert wäre. So ist es für den Teil der Autorisierung leider nicht möglich, in voller Mächtigkeit auf JAAS zurückzugreifen, da die Gültigkeit des „`base`“ Pseudobjekts[4] auf die jeweilige `callin` Methode beschränkt ist.

[4] Diesen Status hat `base` nicht wirklich; der Begriff soll den Vergleich ermöglichen.

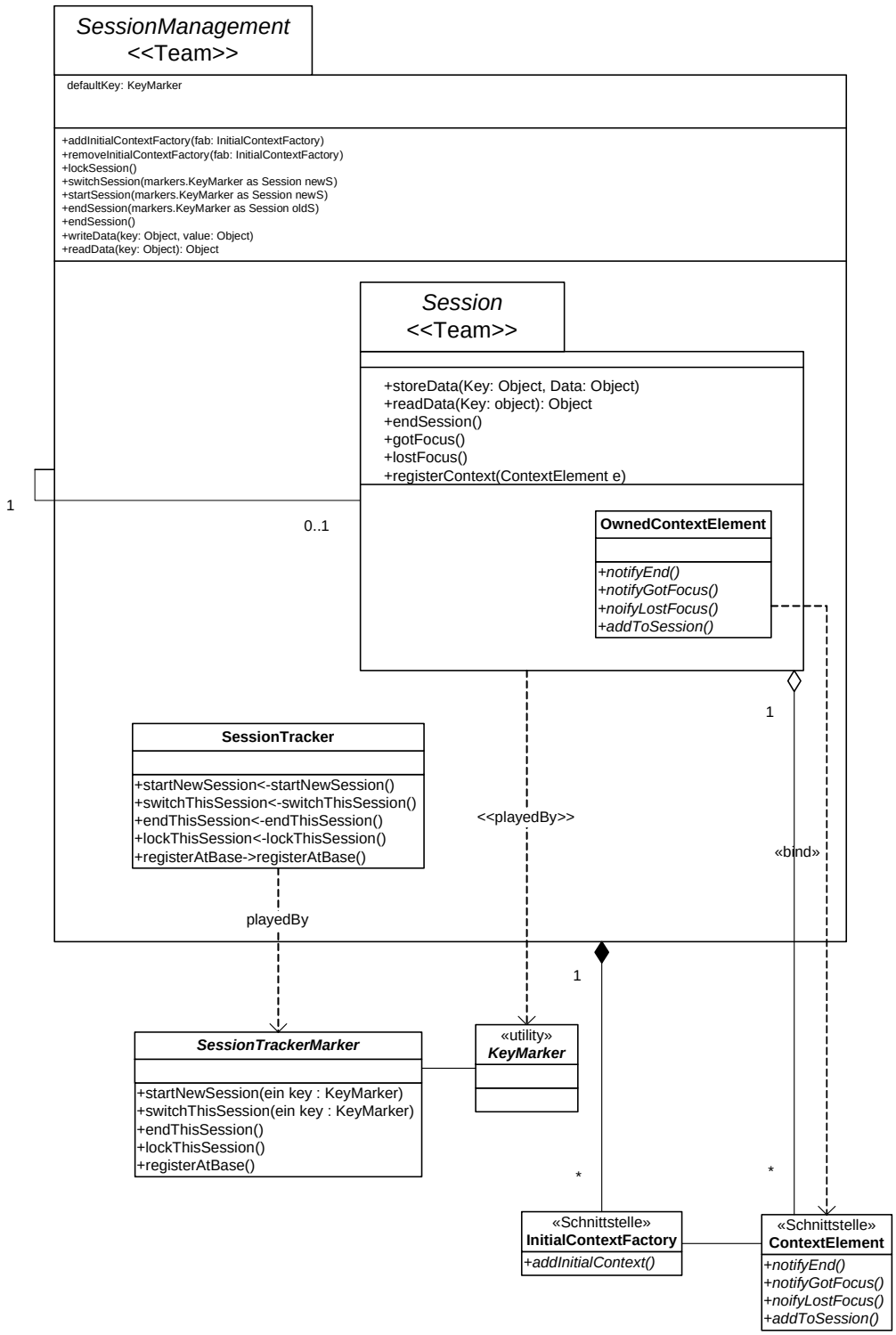

Abbildung 6.8: ObjectTeams: Nächste Stufe des Entwurfs. Session als nested Team.

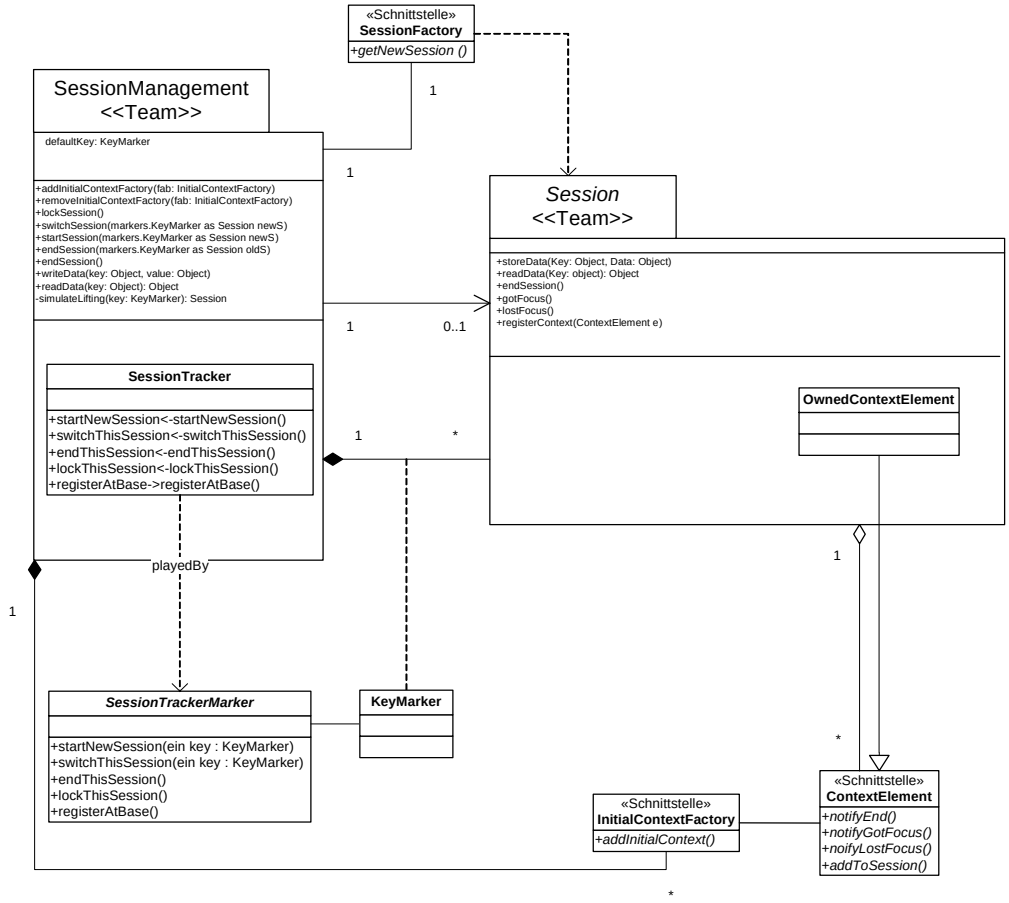

Abbildung 6.9: ObjectTeams: Der endgültige Entwurf.

Daher ist das Konzept der JAAS `doAs` Methoden über `action` Objekte nicht in vollem Umfang möglich. Es ist lediglich die Möglichkeit der Kontrolle der Rechte am Joinpoint gegeben. Für weiter unten im Kontrollfluss liegende Überprüfungen ist eine weitere Kontrolle erforderlich.
Ein weiteres Hindernis liegt in den eingeschränkten *Reflection* Methoden begründet, durch welche die Signatur der gerade ausgeführten Basismethode nicht während der Ausführung abgeleitet werden kann. Dies hätte die eigene Implementierung der nötigen Funktionalität gestattet.
Darüber hinaus ergeben sich weitere Probleme, insbesondere die Unmöglichkeit `super` Aufrufe in `callins` Methoden einzusetzen, weshalb eine Reduktion der Funktionen in der ObjectTeams basierenden Lösung erforderlich wurde.

Der gewählte Kompromiss sieht vor, dass Rechte lediglich punktuell überprüft werden. Im Fall eines Fehlschlags ergibt dies kaum einen Unterschied zur eigentlich präferierten Ausführung in einem anderen Kontext, da beide Fälle dann durch eine *Exception* den Kontrollfluss unterbrechen.

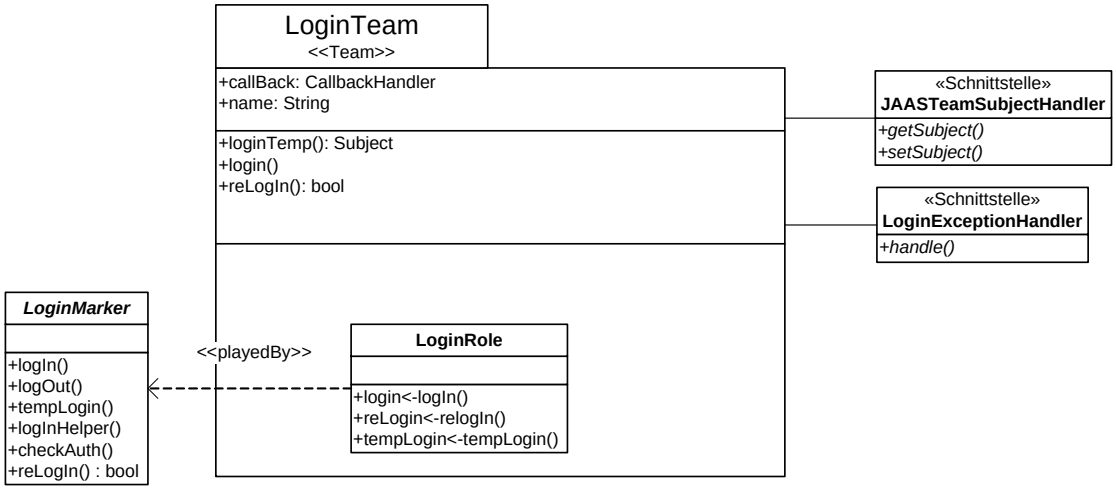

Abbildung 6.10: ObjectTeams: Das LoginTeam. Die Schnittstellen teilt sich dieses Modul mit dem Gegenstück zur Autorisierung. LoginMarker ist der vorherrschende neue *Extension Point*.

Die unter dieser Einschränkung formulierte Lösung ähnelt sehr dem Authentikationsteam, auch hier tritt wieder eine abstrakte Klasse als Stellvertreter des Teams auf. Diese Stellvertreterklasse gestattet es, über Callinbindungen von Rollen Zugangsbeschränkungen an Joinpoints einzufügen. Solche Rollen können in einem gänzlich anderen Team stehen, da die feste Bindung an die abstrakte Klasse beinahe automatisch zur Ergänzung der Funktionalität führt. Der `checkPermission` *Callin* erfordert eine `Permission` Instanz, die beispielsweise in einem Verbindungsteam mittels *Parameter Mapping* ergänzt werden kann.
Die Notwendigkeit, die Ausführung der Basismethode durch eine `Exception` zu verhindern, erfordert die Bindung in einem `replace` *Callin*. Dabei ist im Idealfall ein sehr deklarativer *Connectorstil* möglich, da über *Guard Predicates* dynamisch passende Bindungen ausgewählt werden können.

6.2.2 JAsCo

Als Notation für die statischen Teile der JAsCO Modelle kommt eine eigene Schreibweise zum Einsatz. *AspectBeans* werden ähnlich der *UFA* Notation für *Teams* sowohl als *Packages* und auch – je nach der Art ihrer Referenzierung – als Klassen dargestellt. *Hooks*, grundsätzlich zustandsbehaftete *Advice*/*Pointcut* Paarungen, werden als Klassen dargestellt, jedoch halb im Inneren, halb außerhalb ihrer *Bean*. Das soll ihren Schnittstellencharakter betonen, ohne ihre Instantiierbarkeit zu verstecken. Die Darstellung der *Beans* selbst orientiert sich in diesem Fall an der UFA und ist eine Kombination aus *Packages* und Klassen. Dabei werden einige neue Stereotypen eingeführt, zu nennen sind ≪Hook≫ und ≪AspectBean≫, welche Hooks, respektive AspektBeans kennzeichnen. Ferner ≪refinable≫, welches zur Markierung einer Methode als `refinable`, also zur

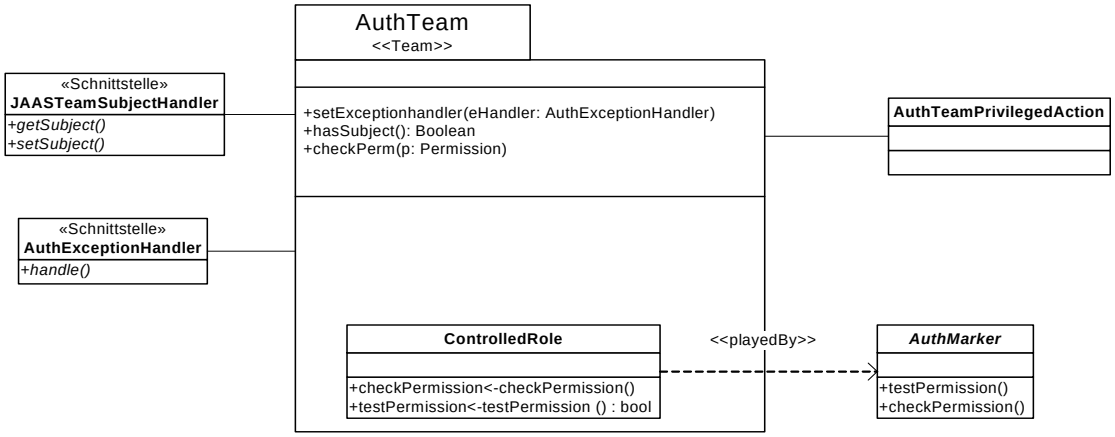

Abbildung 6.11: ObjectTeams: Der Entwurf für das Autorisierungsteam. AuthMarker fungiert als Schnittstelle; SubjectHandler entspricht jenem des Loginteams.

Laufzeit redefinierbar, dient. ≪Mixin≫ zeigt an, dass ein *Hook* ein Mixininterface (vgl. Abschnitt 2.2.1.8) implementiert und somit als Rollentyp auftritt. Schließlich wird noch ≪advice≫ verwendet, um – naheliegenderweise – Advice zu markieren.

Der Entwurf für die Implementation in JAsCo bedarf bereits zu Beginn die Notwendigkeit einer grundlegenden Abwägung: Beschränkung auf das statische Pointcut/Joinpointmodell oder Umgehung desselben durch `refinable` *Guard*-Methoden. Ersterer Ansatz wäre deutlich performanter, da dynamische Bedingungen in Aspekten JAsCo merklich verlangsamen; dies trifft noch deutlich stärker auf `refinable` Methoden zu. Andererseits ist das hauptsächlich statische Pointcutmodell momentan zu beschränkt, um Joinpoints zielgenau auswählen zu können, wobei insbesondere die Auswahl geerbter Methoden als *Joinpoints* kaum möglich ist.
Für alle im Rahmen des Entwurfs ausgestalteten Hooks ist daher die am flexibelsten zu beurteilende Lösung zu wählen, welche in der Kombination der in Abschnitt 5.1 diskutierten maximalen abstrakten *Pointcuts* mit teilweise in den *Connector* verlagerten `isApplicable` (siehe Abschnitt 5.1.2.1) Konstrukten zu sehen ist.

JAsCo erzwingt eine deutlich starrere Herangehensweise, um der vergleichsweise beschränkten Vererbungssemantik gerecht zu werden. Dies wird noch weiter durch den monolithischen Charakter von *JAsCos AspectBeans* bedingt. Daher wurden die *Hooks* in erster Linie als *Advice* in einem konfigurierbaren Kontext entworfen. Eine auf Stellvertreterobjekten basierende Komposition wie bei ObjectTeams ist in JAsCo aus technischen Gründen nicht möglich. Verkettung von verschiedenen Hooks kann daher nur durch eine geeignete Auswahl von Pointcuts stattfinden; die direkte Kommunikation zwischen Aspekten ist nicht ohne weiteres innerhalb des statischen Typsystems möglich.
Folglich haben *AspectBeans* eine noch größere Konzentrationswirkung als *Teams*.

6.2.2.1 Sessionmanagement

Die Anforderungen des Aspektes gliedern sich in zwei Gebiete. Eines liegt in der Infrastruktur zur Verwaltung von Sessions und ihrer Daten, das andere im Entwurf der Anbindung. Das gilt bezüglich der abstrakten *Pointcuts* und deren *Advice*, insbesondere aber hinsichtlich Anpassbarkeit auf Basisprogramme.
Für die Infrastruktur findet sich auch bei der auf JAsCo basierenden Interpretation des Konzeptes die Idee, durch – unter Umständen nicht im Programmfluss der Basis liegende – Ereignisse, den Zustand der Basisanwendung zu beeinflussen.
Bei JAsCo sind zunächst zwei Instrumente geeignet, eine solche Funktion zu implementieren. Das erste solche Instrument ist die Möglichkeit, Nachrichten zu anderen Instanzen der gleichen Klasse umzuleiten. So können Objekte faktisch ausgetauscht werden, ohne eine Anpassung der Referenzen in anderen Objekten der Basis zu erfordern. Dieser Ansatz hat den Nachteil, dass nur Nachrichten umgeleitet werden können, welche auch durch das Pointcutmodell erfassbar sind – dies ist jedoch nur eine eher kleine Teilmenge.
Die zweite Möglichkeit stellen *Virtual Mixins*, JAsCos Instrument, um Basisobjekte mit weiteren Funktionen zu dekorieren, welche in – als Rollenklasse dienenden – *Hooks* implementiert werden. Ähnlich wie in dem auf ObjectTeams aufbauenden Entwurf, können somit Zustände der Basis gespeichert und wieder hergestellt werden. Anders aber als bei ObjectTeams, erlaubt JAsCo lediglich eine Rolleninstanz eines gegebenen Typs für ein Objekt der Basis – eine globale Einschränkung, die auch *Connector*- und *Bean*grenzen überschreitet.
Da die Einschränkungen der ersteren Lösung wesentlich größere Anpassungen der Basisanwendung erfordern, wurde die letztere Variation gewählt.

Die Einschränkung auf eine Rolleninstanz pro Basisobjekt erfordert dabei die Speicherung der sessionabhängigen Daten außerhalb des als Rollenklasse dienenden Hooks.
Dadurch wird es erforderlich, die Rollen der sessionabhängigen Objekte global in der *AspectBean* zu speichern, eine Speicherung im Kontext nur einer Instanz einer Sessionklasse ist nicht zielführend und kann im Bedarfsfall einfach emuliert werden.
Umgekehrt sind hierdurch die Anforderungen an eine Klasse `Session` bereits weitgehend festgelegt: Speicherung der sessionsabhängigen Daten der Rollen. Für die Datenstruktur zur Speicherung der Daten einer *Session* wurde eine dedizierte Klasse – `Session` – gewählt. Ein Basisbezug existiert nicht, weshalb ein *Hook* nicht in Frage kommt. Der Singletonstatus der *Bean* macht diese selbst ungeeignet für die Aufgabe. Eine Implementierung der Sessionklasse als *Inner Class* der *AspectBean* selbst ist ebenfalls nicht empfehlenswert, da eine solche Klasse bei Vererbung nicht redefiniert werden könnte.

Zwei verschiedene Arten der Speicherung von Daten scheinen notwendig: Speicherung nur für die speichernde Instanz und globale, aber sessionslokale Speicherung.

Der Zugriff auf solche Daten für die Rollenobjekte, von nun an klassifiziert als `Session-Dependent`, wird über entsprechende Hookmethoden berücksichtigt. Dabei ist der Zugriff nur möglich, wenn der Aufruf aus einer Session-Instanz – etwa bei einem Zustandswechsel – erfolgt.

Durch dieses konsumierende Wesen des `SessionDependent` *Hooks* ist also die Benachrichtigung durch die jeweils aktive `Session` nötig. Für diese Benachrichtigung kommt eine Variante das *Observer* Patterns zum Einsatz, bei welchem die Liste der *Observer* nicht in dem *Subject* `Session` gehalten wird.

Auch der Entwurf des *Hooks* für die Mixins `SessionDependent` ist somit bereits weitgehend festgelegt. Es fehlen lediglich die *Extension Points* für basisabhängigen Code. Diese sind über `refinable` Methoden einfach und vielseitig umsetzbar. Für die Ereignisse `gotFocus`, `LostFocus` und `endSession`, also für Wechsel der aktiven *Session* bzw. deren Ende ist dies naheliegend. Darüber hinaus werden mit `init` und `dynamicCondition` noch Möglichkeiten vorgesehen, die Erzeugung zu beeinflussen. `DynamicCondition` erlaubt es, die Registrierung von neuen `SessionDependent` Instanzen zu unterbinden; `init` ermöglicht Aktionen während der Erzeugung. Über die durch den Hook implementierte (Mixin-) Schnittstelle können innerhalb dieser Methoden Daten in der aktuellen Session hinterlegt bzw. aus dieser gelesen werden.

Die Auswahl der Menge der „herkömmlichen“ *Hooks* lädt zu der nahe liegenden Auflistung der geforderten Operationen ein:

- StartSession
- EndSession
- LockSession
- SwitchSession

Hierbei setzt jeder Hook die Auslösung der namensgebenden Operation in `around`-Advices um. Die Implementierung der eigentlichen Funktionalitäten liegt in der SessionBean selbst, um unnötige Redundanzen zu vermeiden. Allen diesen Hooks sind zwei `refinable` Methoden gemein: `informBase` und `dynamicCondition`: Erstere dient zur Ergänzung von basisabhängigem Code während der Ausführung der *Advices*, letztere ermöglicht einen dynamischen Umgang mit dem `isApplicable` Konstrukt, wie in Abschnitt 5.1.2.1 vorgestellt. Diese Methode fungiert als *Guard* über die Ausführung des *Advices*.

Die Hookkonstruktoren werden gemäß den Überlegungen aus 5.1.1.2 maximal ausgestaltet, um eine weitgehende Unabhängigkeit von Basisprogrammen zu ermöglichen.

Da die Sessioninstanzen zuordenbar sein müssen, erfordern die steuernden Hooks – `SwitchSession`, `EndSession` und `StartSession` – einen Mechanismus, um einen Schlüsselwert zu erhalten. Hierfür ist mit `getKey` jeweils eine weitere `refinable` Methode vorgesehen, deren Implementierung jedoch nicht optional ist.

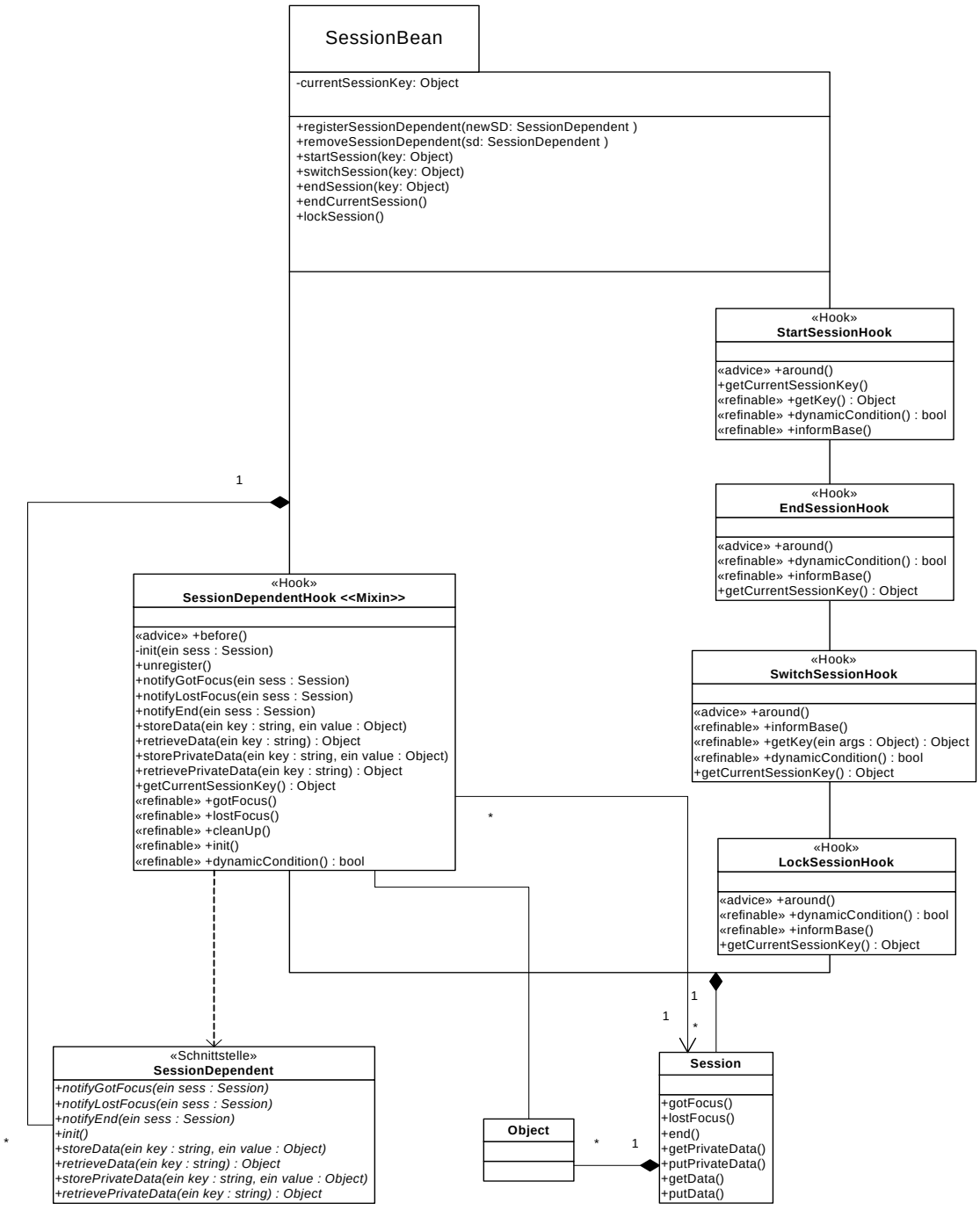

Abbildung 6.12: JAsCo: Die vollständige Sessionbean. Eigene Notation: Die *Hooks* sind halbwegs außerhalb dargestellt, um ihren Schnittstellencharakter zu verdeutlichen.

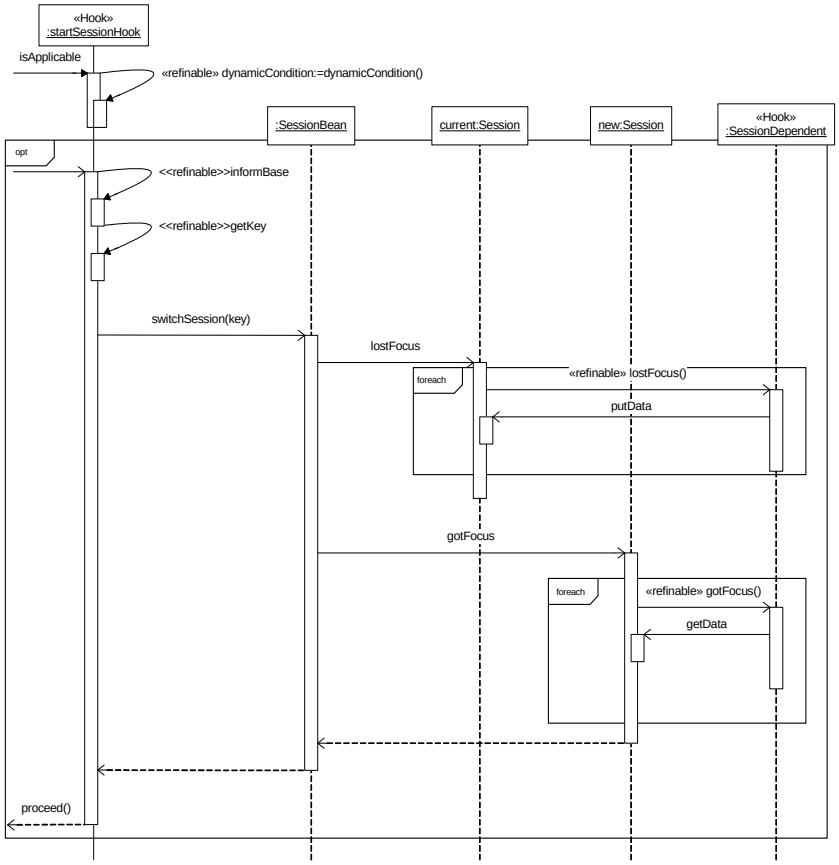

Abbildung 6.13: JAsCo: Ablauf der Kommunikation bei einem Zustandswechsel. „`lostFocus`“ ist beispielhaft gewählt. Auf die Darstellung der, hier noch gezeigten, Abfrage der `isApplicable` Bedingung wird in folgenden Sequenzdiagrammen verzichtet. Das Verhalten ist jeweils analog.

Die `SessionBean` selbst übernimmt die Funktion der Verknüpfung beider Richtungen: Sie implementiert die Methoden, die passend zu den *Hooks* das gewählte Sessionsystem aktualisieren.

6.2.2.2 Autorisation und Authentikation

JAsCos Konzept der Aspektkommunikation lässt die Umsetzung beider *Concerns* in einer einzigen *AspectBean* praktikabler erscheinen als eine Aufteilung auf mehrere Module. Dennoch gliedern sich die Funktionen der zu modellierenden `LoginBean` in zwei Gruppen: Der Bereich der Authentikation erfordert die Funktionen *Login* und *Logout* sowie die schwächere Form der Überprüfung bzw. Aktualisierung.
Autorisation wiederum ist mit JAsCos Mitteln sehr mächtig umsetzbar und erlaubt eine weitgehende Nutzung der Java `security` Infrastruktur: Nicht nur können Sicherheitsbeschränkungen (`Permission`) in das Programm eingefügt werden, es ist auch möglich, den Sicherheitskontext zu verändern und bestehende Beschränkungen, etwa für Dateien, auf

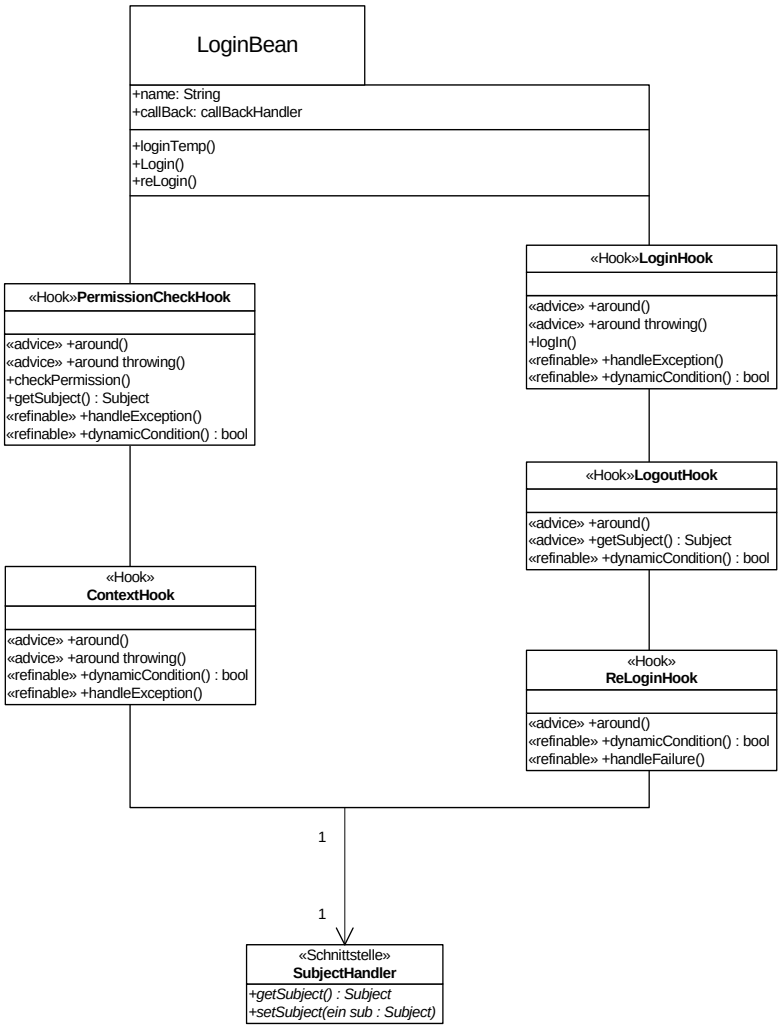

Abbildung 6.14: JAsCo LoginBean: Insbesondere die Kommunikation im Fehlerfall erfordert die Umsetzung in einer *Bean*. Lediglich das Einfügen neuer `Permission`s bleibt getrennt.

Basis des gewobenen Logins zu überprüfen.

Übersetzt in einen Entwurf bedeutet dies eine *Bean* mit drei *Hooks* für Authentikation und zwei für Autorisation, wobei die umschließende *Bean* die Anbindung an *JAAS* und die Verwaltung des `Subject`s wahrnimmt.

Die drei *Hooks* für die Anbindung der Authentikation sind als `LoginHook`, `LogoutHook` und `ReLoginHook` bezeichnet. Ihre Aufgabenzugehörigkeit gliedert sich analog zu ihren Namen. Wie auch schon bei den *Hooks* der `SessionBean` ist die dynamische *Guard* als `refinable dynamicCondition` vorgesehen, um nicht auf statische *Pointcuts* beschränkt zu sein. Die *Hooks* `LoginHook` und `ReLoginHook` ermöglichen ferner die Konfiguration des Verhaltens im Falle eines Fehlschlags über `handleException` bzw. `handleFailure` Methoden, die einen erneuten Versuch ebenso ermöglichen, wie einen Abbruch.

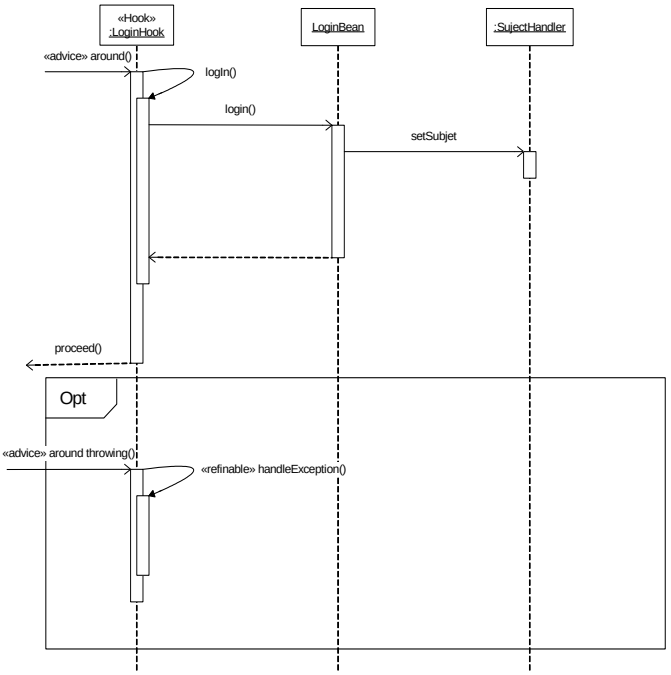

Abbildung 6.15: JAsCo LoginBean: Ein Loginvorgang, andere Authentikationsoperationen verlaufen analog.

Die zwei für die Autorisation vorgesehenen *Hooks* sind etwas anders gelagert: Zwar ist auch bei ihnen eine `refinable dynamicCondition` vorgesehen, jedoch besteht der *Advice* nicht aus konfigurierbaren Operationen. Vielmehr verlagert ihr *Advice* die Ausführung der abgefangenen Basisoperation in den Sicherheitskontext des gerade eingeloggten `Subjects`, was sich auf alle Java Sicherheitsbeschränkungen auswirkt. `ContextHook` hat hierbei keine weitere Wirkung, während der sehr ähnliche `PermissionCheckHook` eine zusätzliche Beschränkung einfügt. Letztere wird über die `getPermission refinable` Me-

thode konfiguriert.
Mittels `around throwing` Advice ist bei beiden *Hooks* auch jeweils eine `handleException` Methode vorgesehen.

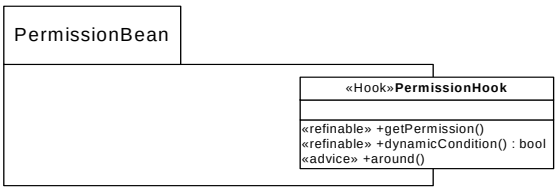

Abbildung 6.16: JAsCo PermissionBean: Eine triviale Bean, deren Hook neue Permissions in das Programm webt. Siehe `PermissionCheckHook`.

Zusätzlich zu dieser kombinierten Authentisierungs-/Autorisierungsbean ist eine sehr einfache `PermissionBean` vorgesehen, die das Einfügen von Zugangsperren in existierende Programme erlaubt. Diese *Bean* verfügt nur über einen einzigen Hook, der weitgehend mit dem `PermissionCheckHook` der Kombinierten übereinstimmt. Die Idee hinter diesem Zusatz ist die Möglichkeit der Nutzung von Autorisiation ohne Authentikation.

Die *Bean* selbst referenziert ein `SubjectHandler` Objekt, über welches das zugrundeliegende `Subject` verwaltet wird. Hier liegt auch die Schnittstelle für andere Aspekte/-Module.

6.3 Vorstellung der Resultate

6.3.1 Integration

Den eigentlichen Kern der Studie stellen die Möglichkeiten der a-posteriori Integration in die unter Abschnitt 4.2 beschriebenen Anwendungen dar.

Dazu soll zunächst beispielhaft das beschriebene Dispositionssystem dienen. Als zweite Stufe erfolgt dann die weitere Anpassung an den Betrachter für Logdateien. Zusätzlich wurden auch einige Sonderfälle mit einbezogen, etwa der Einsatz in verteilten Anwendungen.

Für die Integration der unabhängigen Aspekte in die Anwendungen wurde folgende Strategie gewählt:

- Sessions werden durch eine ergänzte grafische Oberfläche kontrolliert.
- Ohne aktive Session wird die Basisanwendung gesperrt.
- Einloggen wird bei dem Start einer Session erforderlich.
- Ein Fehlschlag beim Einloggen beendet die Session.
- Wechseln der Session führt zur Überprüfung der Authentikation.

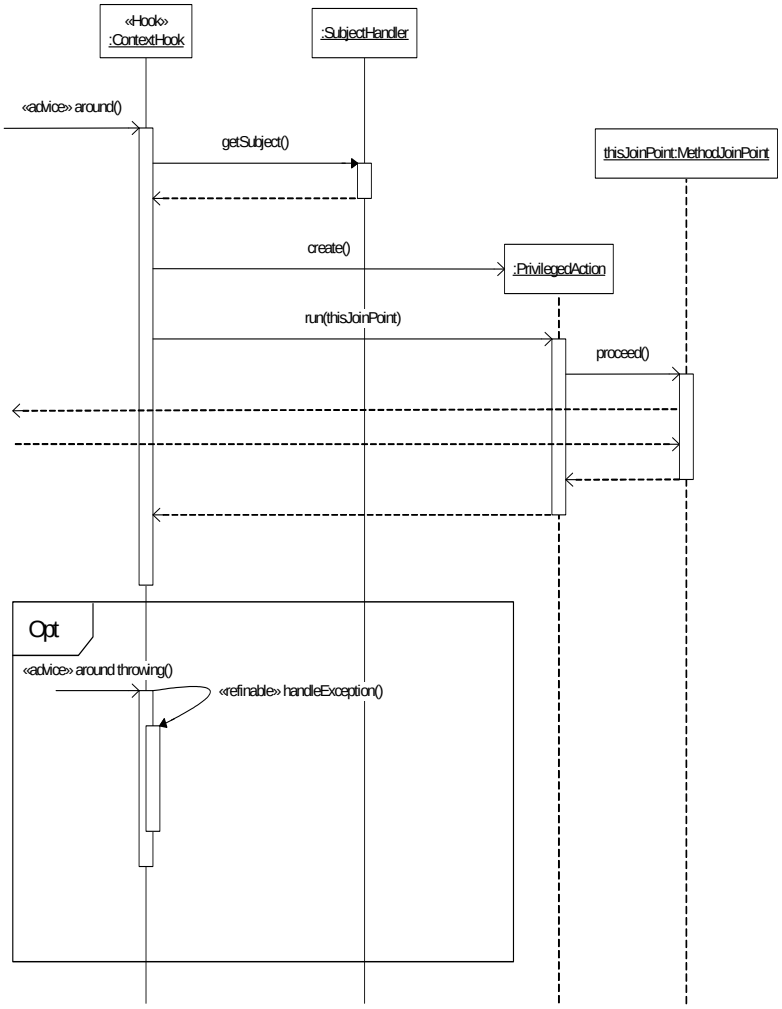

Abbildung 6.17: JAsCo: Die Verlagerung in einen neuen Sicherheitskontext.

- Ein Fehlschlag sperrt die Session.
- Der Versuch des Zugriffs auf gesperrte Ressourcen führt zur Frage nach Autorisierung durch einen anderen Nutzer.

6.3.1.1 Object Teams

Der auf ObjectTeams basierende Ansatz ist über zwei verschiedene Wege an Basisprogramme anzupassen. Zum einen wird über die Anbindung kontrollierender Rollen gearbeitet, welche auf Rollen/Klassenebene vererbt werden, zum anderen erfolgt Spezialisierung von Teams, also Teamvererbung. Dabei ermöglicht die Teamvererbung sowohl das Zusammenfassen mehrerer Teilaspekte als auch deren getrennte Spezialisierung.

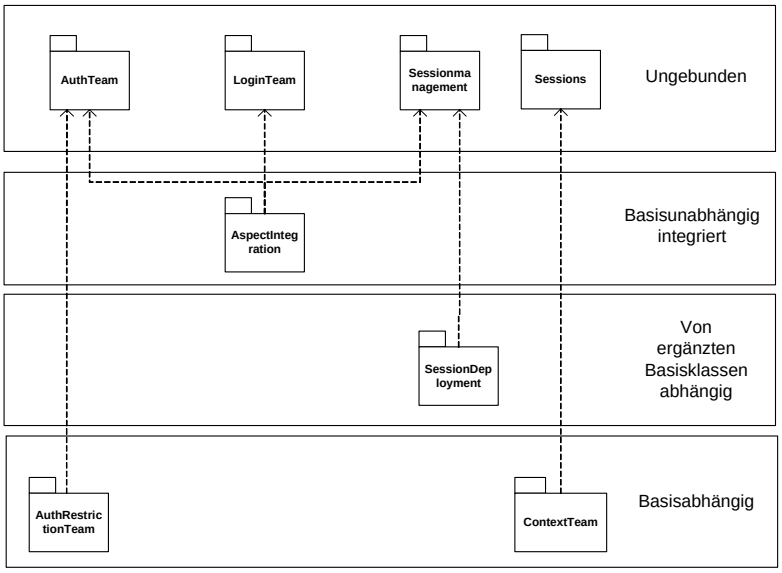

Abbildung 6.18: ObjectTeams: Informelle Darstellung der Abhängigkeiten der verschiedenen Integrationsstufen in der ObjectTeams Lösung: Lediglich die in der untersten Ebene liegenden Teams sind von der eigentlichen Basisanwendung abhängig.

So ist es möglich, durch Kombination von Implementierungen der im Entwurf vorgesehenen *Stellvertreterklassen*, weite Teile des Protokolls ohne jede feste Basisbindung umzusetzen, also mit erneut hervorragenden Voraussetzungen für eine weitere Wiederverwendung. Konkret war es möglich, den Sessionaspekt mit dem Authentikationsaspekt bereits abstrakt zu verbinden.

Als inhärent basisabhängig stellte sich – wohl bemerkt entwurfskonform – die Anbindung von `Session` an die konkrete Basis heraus. Da der Anwendungszustand vergleichsweise schwer zu erfassen ist, wurde der dynamische Austausch weiter Teile der Oberfläche als geeignete Methode zur Verwirklichung paralleler Sitzungen gewählt. *ObjectTeams* ist im Moment noch nicht in der Lage, mit Basisklassen aus der Java-Api umzugehen. Um dem zu begegnen, wurde die Einführung einer Dummy-Klasse erforderlich, die jedoch für den Entwurf nicht von Belang ist.

Die Einbindung des Autorisationsaspektes ist ebenfalls vollständig basisabhängig. Hier stellten sich einige Probleme, da sich der ursprüngliche Entwurf (siehe Abschnitt 6.2.1.5), diesen Schritt über *Parameter Mappings* deklarativ auszuführen, als nicht umsetzbar erwies. Das Ergebnis erzwang zwar keine Änderungen an Entwurf oder Implementierung, aber erheblich komplexere Verbindungsteams. Dies ergab sich aus einer Reihe von Beschränkungen bezüglich callin-Bindungen. Ausschlaggebend war die Unmöglichkeit mehrere `base` *Guards* einzusetzen und die Natur von *Parameter Tunneling* bei `replace` Callins[5].

[5]Eine Ergänzung von Parametern erfordert die Deklaration vieler zusätzlicher `callin`-Methoden.

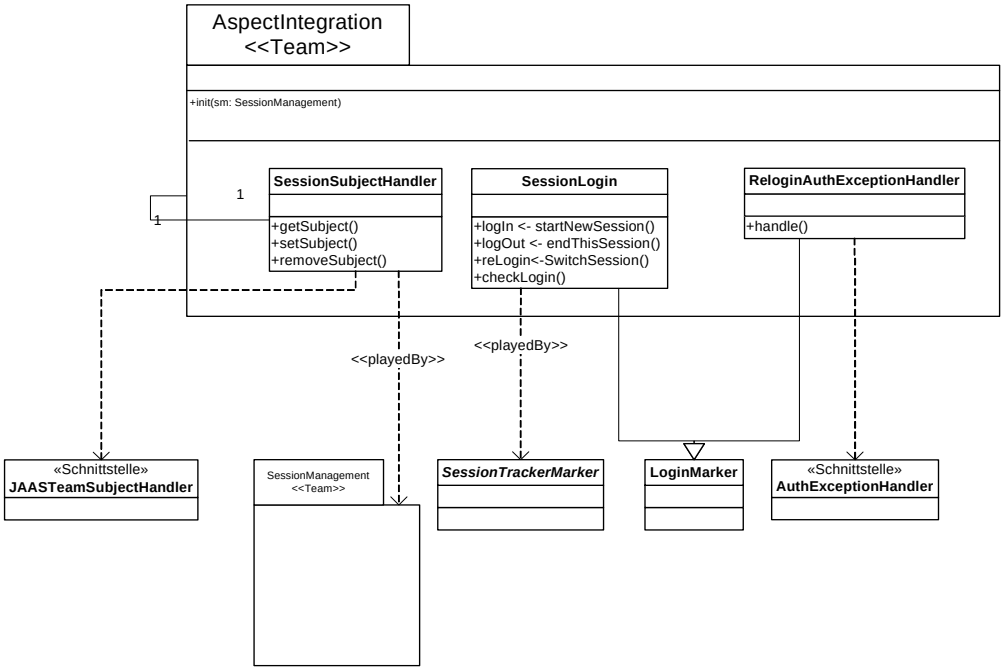

Abbildung 6.19: ObjectTeams: Dieses Team integriert Login und Sessions - ohne eine Basisbindung zu erfordern.

Die weitere Anpassung an den Logbetrachter wurde vergleichbar umgesetzt. Dabei konnte die Verbindung der Teams unverändert weiterverwendet werden. Für `Session` war dabei ein neues Verbindungsteam, nebst einer dazugehörigen `Factory` Klasse, erforderlich. Diese Anpassungen waren bereits im Entwurf vorgesehen. Die Integration der Login- und Sessionaspekte stellte sich als vollständig wiederverwendbar heraus.

6.3.1.2 JAsCo

JAsCo erfordert die Instantiierung abhängiger Aspekte innerhalb eines einzigen *Connectors*. Nur dann ist die Abstimmung mittels *Precedence*- und *Combination* Strategien möglich. Dies bedeutet im vorliegenden Fall, dass eine wiederverwendbare Teilintegration von Aspekten miteinander kaum möglich ist. Vielmehr ist eine Verkettung von Aspekten mittels Weben an den gleichen *Joinpoints*, also durch Angabe von entsprechenden Pointcuts möglich. Die Reihenfolge ist dann über Präzedenzdeklarationen zu klären. Dies ist nicht direkt als Nachteil zu verstehen, da die *Connector* - Dateien durchaus die Aufgabe haben, derartige Abstimmungen mittelbar abhängiger Aspekte zu ermöglichen. Dabei muss in Kauf genommen werden, sehr viele verschiedene Aspekte in einer einzigen *Connector*-Datei zu kombinieren, was auch die Dynamik zur Laufzeit empfindlich reduziert.

Eine teilweise Implementierung des Connectors, die eine bessere Wiederverwendbarkeit in Aussicht stellen würde, ist nicht möglich, da der nötige monolithische *Connector*

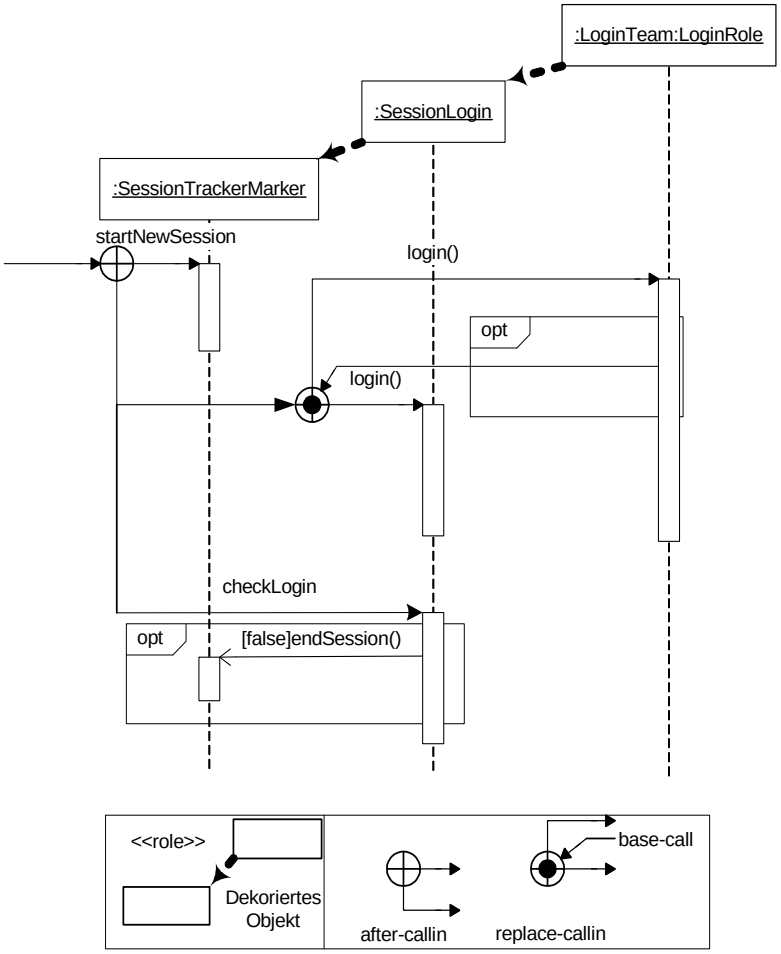

Abbildung 6.20: ObjectTeams: Ablauf der Integration des Sessionaspektes mit dem Loginaspekt. Der möglicherweise fehlende Base-call ist unkritisch, da auf einem `after` Callin basierend. Notation nach [HHM04a].

basisabhängige Teile ebenso konfigurieren muss, wie auch unabhängige. Die fehlende Unterstützung von Vererbung und abstrakten *Connectors* schließt daher eine direkte Wiederverwendung nichttrivialer[6] Verbindungselemente aus.

An einigen weiteren Punkten ist der Spielraum unter Umständen kleiner als es eigentlich wünschenswert wäre: Als *Joinpoints* kommen lediglich öffentliche Methoden der Basisanwendung in Frage[7]. Auch bei der Kommunikation in Richtung Basis, insbesondere in *Refinements*, sind nur gemäß der Java-Sprachdefinition sichtbare Methoden gültig. Daher mag die Anbindung an eine vollständig *oblivious* Basis mitunter nicht möglich sein.

[6] *Logging* ist z.B. vollständig wiederverwendbar.
[7] Es können auch private Methoden ausgewählt werden, wenn die Laufzeitumgebung entsprechend konfiguriert wird.

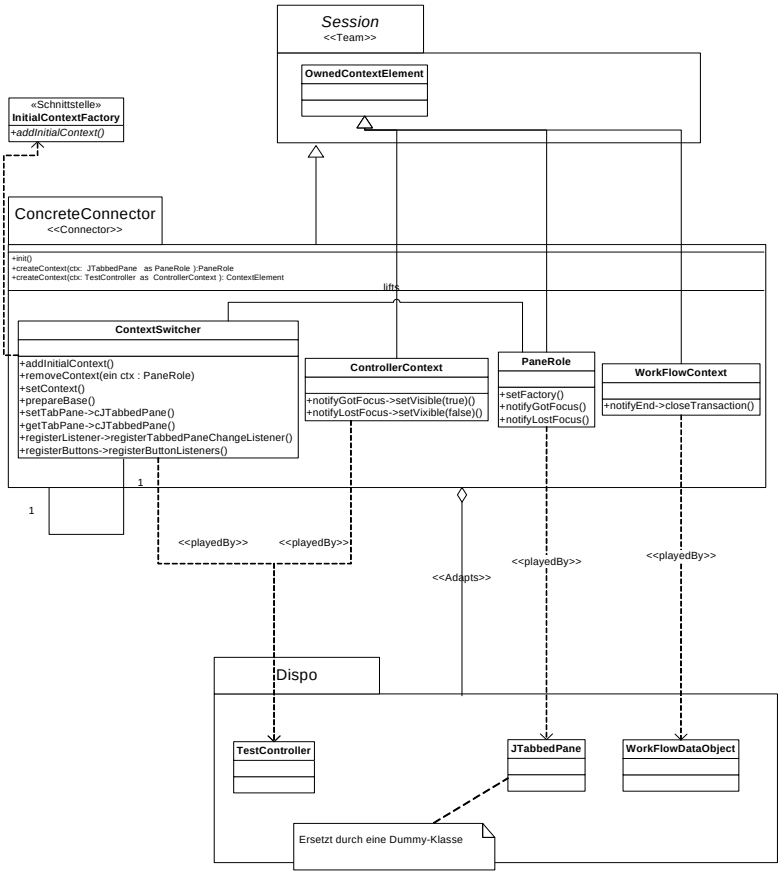

Abbildung 6.21: ObjectTeams: Anpassung von `Session` an das Dispositionssystem.

Dennoch war die Anpassung an das zunächst als Beispiel dienende Dispositionssystem ohne weitere Probleme möglich, gleich ob teilweise in dedizierten *Refining Classes* oder vollständig in einen *Connector* umgesetzt. Aufgrund des bereits beschriebenen Mangels eines Wiederverwendungsskonzeptes für *Connector*-Dateien, war zur Anpassung an den Logbetrachter ein neuer *Connnector* nötig. Dieser unterschied sich in weiten Teilen nur in den Klassennamen in *Hookkonstruktoren* von jenem für das Dispositionssystem.

6.4 Gegenüberstellung der Resultate

Wie bereits zu Beginn des Kapitels festgestellt, soll keine absolute Bewertung vorgenommen werden. Vielmehr werden an isolierten Situationen beispielhaft Vorteile und Unterschiede verdeutlicht und Grundlagen für Verbesserungsvorschläge gelegt. Eine direkte Gegenüberstellung von *JAsCos Hooks*, welche letztlich zustandsbehaftete *Pointcut/Advice* Paarungen sind, und Rollen in *ObjectTeams* ist dabei nur auf Basis vergleichbarer Bedeutung in der Implementation möglich.

6.4.1 Einweben neuer Funktionen

Als Einstieg ist die Pflichtdisziplin der Aspektorientierung geeignet: Das Einweben neuer Funktionen mit *Obliviousness* und *Quantification*. *Quantification* spielt für die vorliegende Untersuchung eine vergleichsweise kleine Rolle, da die untersuchten Aspekte einen eher punktuellen Charakter haben. Als Ausnahme von dieser Regel muss allerdings die Autorisation angeführt werden, die in einer Vielzahl von Szenarien von mächtigen *Quantification* Ausdrücken, i.e. *Pointcuts*, unterstützt werden kann.
Erwartungsgemäß gelingt dies mit *JAsCos Pointcut* Ausdrücken um ein Vielfaches einfacher, als mit den direkten Bindungen von ObjectTeams. So müssen bei letzterer Sprache alle mit Sicherheitsüberprüfungen zu belegenden Methoden explizit in `callins` gebunden werden, was wiederum für ihre jeweiligen Klassen eigene Rollenklassen erfordert – sofern die zu beschränkenden Methoden nicht in einer gemeinsamen Superklasse deklariert wurden. Soll eine einzelne Bindung für verschiedene Objekte und Methoden halten, so erfordert dies entweder verschachtelte Teams oder die Einführung einer gemeinsamen Superklasse, jedenfalls schwerwiegende Eingriffe in die Struktur.
ObjectTeams ist dabei aber nicht eindeutig weniger mächtig als JAsCo; JAsCos Pointcuts erlauben nicht uneingeschränkt das Auswählen geerbter Methoden als Joinpoints. Diese können aber von ObjectTeams sehr wohl als Basis für `callin` Bindungen verwendet werden. Im konkreten Fall bedeutet dies, dass manche Methoden nur unter Inkaufnahme von Umwegen als Webpunkte für Sicherheitsbeschränkungen ausgewählt werden können. Dies ist dann mit einem erheblichen „*Overhead*“, sowohl in Hinsicht auf Performanz als auch auf Anpassungsaufwand, verbunden. *ObjectTeams* unterliegt in dieser Situation keinen vergleichbaren Einschränkungen.

6.4.2 Koordination von Aspekten

Dieser Vergleichspunkt gilt nicht primär den unterschiedlichen Möglichkeiten, Konflikte zwischen Aspekten aufzulösen, sondern vielmehr den Formen der angewendeten und anwendbaren Aspektkommunikation. Beide Sprachen sind in ihrem jeweiligen momentanen Entwicklungsstand bezüglich der Auswahl von *Joinpoints* in anderen Aspekten eingeschränkt. Insbesondere in JAsCo ist es nicht sinnvoll möglich, Aspekte aspektorientiert aneinander zu binden. Auch ist die Referenzierung von Objekten aus einer *AspectBean* heraus exklusiv: Direkt referenzierte Klassen kommen nicht als Basis für Aspekte in Frage. Daher müssen die Versuchsaspekte grundsätzlich unterschiedlich koordiniert werden. Mit ObjectTeams ist eine Kombination aus direkter und aspektorientierter Bindung möglich, um einzelne Aspekte miteinander zu kombinieren. In JAsCo hingegen verbietet die nötige Distanz der Aspekte eine direkte Kombination. Dies manifestiert sich in Umwegen, etwa identisch zu wählenden *Pointcuts* für abhängige Aspekte, die so – an die gleichen Joinpoints gewoben – über *Precedence*- und *Combination Strategies* aufeinander abgestimmt werden können.

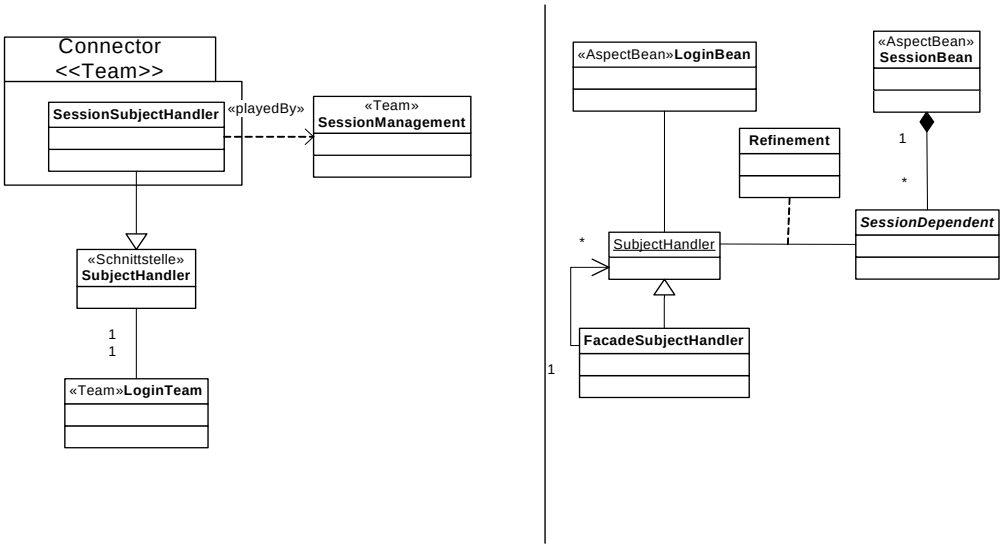

Abbildung 6.22: Verbindung des Loginaspektes mit dem Sessionmanagement Aspekt. Auf der linken Seite ObjectTeams, auf der rechten JAsCo. Im Falle von JAsCo erhöht sich die Komplexität dadurch, dass nur Objekte, die nach der Bearbeitung des Connectors erzeugt wurden, als Basisobjekte in Frage kommen. Daher ist die Einführung einer Indirektion nötig, um die Erzeugung der `SubjectHandler` Instanz erst zur Laufzeit des Basisprogramms stattfinden zu lassen. Nur dann können die Daten durch die `SessionBean` auf den Stand der gerade aktiven Session gesetzt werden.

6.4.3 Anpassbarkeit von Bindungen

Die Mächtigkeit und Flexibilität der Verbindung ist ein grundlegendes Merkmal für wiederverwendbare Aspekte. Soll der Aspekt a-posteriori in eine Basisanwendung integriert werden, so ist eine beträchtliche Anpassung erforderlich.

Aspekte über Bindungen möglichst im Stile eines *BlackBox Frameworks* anpassen zu können war ein verfolgtes Ziel für beide Ansätze. Die Idee liegt in der bloßen Anpassung der Bindung, was besonders bei JAsCo als wichtig erachtet wurde, da eine Anpassung von AspectBeans problematisch ist.
In ObjectTeams, wo nahezu alle Konstrukte redefiniert werden können, war dies weniger entscheidend, aber immer noch relevant. Diese Relevanz fußte auf dem Ziel, möglichst einfache Bindungen zu eröffnen.

Parameter Mappings in ObjectTeams schienen geeignet, um mit Methodenbindungen die Basis einfach und flexibel anzubinden. Der Plan war es, geeignet formulierte `callin` Methoden durch Parameter anzupassen. Die konkreten Parameter sollten dann unabhängig – über *Parameter Tunneling* – von den Parametern der Originalmethode in einem *Mapping* definiert werden.

Dies erfüllte sich jedoch nur für die Bindungen, für die keine `replace` Callins erforderlich waren. Für andere Methoden scheiterte die Anwendung an einer Reihe von

Grenzen. So ist ein Zugriff auf die ursprünglichen Parameter eines `callin`s nicht möglich. `Parameter Tunneling` ist nicht mächtig genug, um ergänzte Parameter der Callin-Methode von jenen der Basis zu unterscheiden. Die Anwendung erfordert vielmehr die Definition zusätzlicher `callin` Methoden, um die nötige Vielfalt an Signaturen bereitzustellen.
Diese Ergänzung zusätzlicher Parameter ist kaum flexibel zu gestalten, da eine Delegation auf andere Callin Methoden nicht möglich ist. Daher blieb die ursprünglich angedachte Verbindungsstrategie nur für parameterfreie Basismethoden unangepasst anwendbar.
Darüber hinaus führte die Beschränkung auf ein `base` *Guard Predicate* je Basismethode zu noch weitergehenden Problemen, die mitunter mehrere Rollen erforderlich machten. War eigentlich (siehe 6.2.1.5) die Verwendung von *Guard Predicates* für die Auswahl der zu überprüfenden `Permission` vorgesehen, so wurde dies durch diese Einschränkung teilweise verhindert.

```
1 ...

3 aspects.LoginBean.isolatedPermissionCheck permissionHook = new ...
  {
5                 public    java.security.Permission getPermission(){
                            return new login.util.SimplePermission("
                                action",2);
7                 }
  }
9 ...
```

Listing 6.1: JAsCo: Anpassung einer Methodenbindung in einem Connector. Der deskriptive Connectorstil muss durch die Definition einer `refinable` Methode unterbrochen werden. Erst zur Laufzeit erfolgt die Einbindung. Dafür kann die Verbindung so abhängig von dynamischen Bedingungen formuliert werden und ermöglicht eine nahezu ideale Ausnutzung der Mächtigkeit von *Pointcuts*.

Daher ist in beiden Sprachen die konkrete Implementierung der Verbindung dieses Aspektes sehr ähnlich: Eine `abstract`, beziehungsweise `refinable` Methode liefert die eigentliche `Permission` Instanz, die durch den Aspekt zwecks Überprüfung in das Programm gewoben wird. Bei ObjectTeams war die Implementierung dieser Methode in Rollenklassen erforderlich. Das waren immer konkrete Rollen des Verbindungsteams.
Auch bei JAsCo war die Implementierung jener Methoden direkt im Verbindungselement erforderlich, da die Auswahl über *Refining Classes* nicht zuverlässig funktionierte.
Eine weitere grundlegende Unterscheidung ist der Zugriff auf Features der Basisinstanzen: Selbst mit *Refining Classes* und allen verfügbaren Optionen der Laufzeitumgebung erlaubt JAsCo nicht den Zugriff auf nichtöffentliche Mitglieder der Basisprogramme. Dies führte dazu, dass ein moderater Eingriff in die Basisanwendung, wie in Abschnitt 4.3 angekündigt, unvermeidbar wurde. Dies wäre für ObjectTeams nicht in vergleichbaren Maße nötig geworden, hier waren lediglich Probleme mit `Listener` Implementierungen zu bemerken. *Decapsulation* erlaubte den Zugriff auf sonstige relevante Funtionen und Felder, ohne eine Anpassung zu erfordern.

6.4.4 Wechseln des Nutzungskontextes

Java verfügt über eine Reihe spezieller Schnittstellen, die das Einbinden von Code unter besonderen, von dem sonstigen Programm verschiedenen, Laufzeitkonditionen erlauben. Die wohl Wichtigste dieser Möglichkeiten ist die Ausführung in einem anderen `Thread`, erreichbar über die `run` Methode von `Thread` bzw. `Runnable`. Die zweite wichtige Konstruktion dieser Art in Java ist `PrivelegedAction`, welches die Ausführung von Code mit einem anderen Sichterheitskontext erlaubt – etwa abhängig vom angemeldeten Nutzer.
In JAsCo bietet das spezielle Objekt `thisJoinPoint` hinreichende Methoden, um den Aufruf der Basismethode in `around` Methoden in einen anderen Kontext zu „*verlegen*“. So kann durch einen Aspekt die Ausführung von Methoden in einen anderen `Thread` ausgeführt oder – wie im untersuchten Fall – mit anderen Zugriffsprivilegien ausgestattet werden.

Vergleichbares ist mit dem Ausdruck `base` in *ObjectTeams* `callin` Methoden nicht möglich, da dieser kein vollwertiges Objekt referenziert. Eine ganze Gruppe mächtiger Aspekte kann daher in ObjectTeams nicht oder nur mit Abstrichen implementiert werden. Hier traf dies auf die übergreifende Überprüfung von Zugriffsprivilegien auf Basis variabler `Subject` Instanzen zu.
So musste schon der Entwurf des Autorisierungsteams unter reduzierten Anforderungen stattfinden (Siehe Abschnitt 6.4.4). Eine Anwendung der Authentisierungsinformationen auf bereits in der Basisanwendung bestehende Beschränkungen war unmöglich.

6.4.5 Vererbung

Ein weiterer deutlicher Unterschied zwischen den beiden Systemen ist der Einsatz der Vererbung als Mittel der Anpassung und Evolution. Die kovariante Vererbung stellt in ObjectTeams ein höchst mächtiges Merkmal dar, das für die Anbindung des `Session`-team ebenso zum Einsatz kommt wie für den Autorisationsaspekt. Darüber hinaus erlaubt es eine teilweise Wiederverwendung von Verbindungscode und dessen Evolution.

Dagegen kommt in der JAsCo Umsetzung keinerlei Vererbung zum Einsatz, obgleich sich *JAsCo* bereits im Verlauf der Studie in dieser Hinsicht entwickelt[8] hat. *JAsCo* bietet weder für *Refining Classes* noch für *Connector*-Module überhaupt Vererbung an. Auch die Vererbung für `Hooks` und *AspectBeans* ist sehr eingeschränkt. AspectBeans können zwar von Java-Klassen und von anderen *AspectBeans* erben, im letzteren Fall werden auch *Hooks* geerbt. Diese sind jedoch im Falle eine Redefinition niemals konform zu gleichnamigen *Hooks* in *SuperBeans*. Eine konforme Redefinition von Hooks ist also nicht möglich. *Hooks* können tendenziell schlechter spezialisiert werden als *Java Inner Classes*[9].

[8] Konkret zerbrachen anfangs Bindungen an *Refinements*, welche nun vererbt werden können
[9] Diese können innerhalb einer Klasse definiert und spezialisiert werden.

6.4.6 Wiederverwendbarkeit von Verbindungen

In direktem Anschluss an die Vererbung, auch von Verbindungscode, ist die Wiederverwendbarkeit von demselben zu vergleichen. Verbindungscode in ObjectTeams besteht in erster Linie aus Teams, in denen über Vererbung und Rollenbindung die Anpassungen von Aspekten an konkrete Programme vorgenommen wird. In JAsCo besteht der Verbindungscode aus *Refining Classes* und *Connector* Dateien.

In der strukturierten Wiederverwendung von Verbindungscode waren klare Vorteile bei ObjectTeams festzustellen. Durch die in dem Entwurf getrennten *Extension Points* für Rechtevergabe, Sessionssteuerung und Basisanpassungen war eine weitreichende Wiederverwendung möglich. So konnte die Abstimmung der Authentikation mit der Sessionsteuerung unverändert von dem Dispositionssystem zum Logdateibetrachter als Einzelplatzanwendung übernommen werden.

JAsCo bietet kein strukuriertes Konstrukt zur Wiederverwendung von Verbindungselementen. Aufgrund ihrer Sonderstellung verfügen diese auch nicht über die Mittel von Java.
Dieser Mangel wird durch das Fehlen jeder Möglichkeit, `Connector`s auf mehrere Module aufzuteilen, bzw. zur Kommunikation mehrerer `Connector`-Dateien untereinander, noch wesentlich verschärft. Nicht einmal eine „`include`“ Anweisung existiert.
Wegen der bereits erwähnten Beschränkungen von *Precedence-* und *Combination Strategies*, erfordert Aspektkoordination so die vollständige Deklaration abhängiger Aspekte in einer einzigen Datei. Daher war eine „echte“ Wiederverwendung nicht möglich; Entwurf und weite Teile des Dispositionssystem-`Connector`s konnten aber übertragen werden. Textuell[10] gesehen war der Grad der Wiederverwendung vergleichbar.

6.4.7 Kommunikation in die Basis

Das möglicherweise ähnlichste Merkmal beider Ansätze ist das Konzept typabhängiger Kommunikation in die Basisanwendung hinein. ObjectTeams verwirklicht dies über gebundene Rollenklassen, die über wohl definierte *Callout* Bindungen und in eng gefassten *Schleusenbereichen* den Zugriff auf die Basis erlauben. Dies bedeutet, dass für jede Klasse, deren Instanzen durch den Aspekt verwendet werden sollen, (mindestens) eine eigene Rollenklasse erforderlich wird, um diese Richtung der Kommunikation ordentlich zu trennen. Mitunter verlangte es auch aufwändige und basisnahe Kommunikation, um benötigte Rolleninstanzen zu erzeugen.
JAsCos Werkzeuge, einschließlich der *Refining Classes*, sind wesentlich weniger formalisiert gestaltet. Das Basisobjekt ist innerhalb von Advice erstmal lediglich als `Object` verfügbar. Diese schwache Bindung verbietet aber auch eine Erfolgsgarantie auf Aufrufe des Basisobjekts. Trotz der Auswahl geeigneter `AspectFactories` und `Mixin` Interfaces

[10] *“Copy'n Paste”*- Wiederverwendung

wurde in der Fallstudie eine zusätzliche, selbstimplementierte Basisbindung notwendig[11]. Der eigentlich interessante Teilaspekt der *Refining Classes* ist die Auswahl einer *passenden Refining Class* in Abhängigkeit der Typen von *Hook* und *Basis*. *Refining Classes* sind zustandslose Klassen, die es so erlauben, basisabhängigen Code in explizit basisabhängigen Modulen zu implementieren.

Praktisch war die Anwendung beider Mechanismen hinreichend, um die im Rahmen der Studie jeweils auftretenden Anforderungen an die Basis umzusetzen. Es handelte sich insbesondere um den Eingriff in die Basisanwendung zur Anpassung des aktuellen Zustands. Die Umsetzung in ObjectTeams gelang weitgehend ohne explizites *Widening*; in JAsCo war ein solches nicht gänzlich vermeidbar.
Grund hierfür ist, dass JAsCos *Refining Classes* in ihrer Mächtigkeit vergleichsweise eingeschränkt sind. Bei JAsCo wird die erste „passende“ Implementierung herangezogen, in ObjectTeams, die bestpassendste.

Die Unsicherheit, welche Implementierung bei verschiedenen, in Subtypbeziehungen stehenden Basisklassen ausgewählt wird, machte es notwendig, in einigen Fällen auf *Refining Classes* (siehe Abschnitt 2.2.1.6) zugunsten von *inline*-Implementierungen im *Connector* zu verzichten. Dadurch wurden diese Verbindungselemente sehr groß. Da aber *Refining Classes* ebenso schlecht wiederverwendbar sind, ist dies nicht als Verschlechterung zu bewerten. Allerdings erschwert dies das Vorhaben, eine Hook-Instanz für verschiedene Basisobjekte aus unterschiedlichen Typen zu nutzen. Dafür wäre der *inline*-Ansatz, einschließlich der Behandlung verschiedener Typen in einer Methode erforderlich, was ohne *Widening* schwerlich umgesetzt werden kann.

6.4.8 Umgang mit Rollen

Die erheblichen Unterschiede zwischen Rollen in *ObjectTeams* und *Mixins* in *JAsCo* machen den Vergleich in dieser Disziplin schwierig. Während *ObjectTeams* Rollen als ein zentrales Element der Sprache anbietet, werden sie in JAsCo als eine besondere Ausprägung von *Hooks* umgesetzt. Auch die Art der Rollen ist grundsätzlich verschieden: So ist es in JAsCo lediglich möglich, jeweils ein *Mixin* eines gegebenen Typs für jeweils ein Basisobjekt zu erzeugen, während ObjectTeams eine Rolleninstanz pro Team erlaubt. Praktisch bedeutet dies, dass ObjectTeams Rollen einen lokalen, eingegrenzten Charakter innerhalb eines Aspektes haben, während JAsCo Mixins ausdrücklich Daten über verschiedene Aspekte verteilen.
Jedoch kann hier ein Vorteil von JAsCo zum Tragen kommen, der auch mit dem JoinPointModell zusammenhängt: Mixininstanzen werden durch die `perobject` *Aspectfactory* bei dem ersten erreichten *Joinpoint* erzeugt, der von einem '`*`'Pointcut ausgewählt wird. Dies ist bei ObjectTeams nicht anders: Auch hier erfolgt die Erzeugung bei der Ausführung einer gebundenen Basismethode. Mangels *Pointcuts* ist dies aber nur durch explizite Bindungen möglich, was hier eine deutlich geringere Flexibilität bedingt.

[11] Bei dem Problem handelt es sich um einen „Bug“ JAsCos. Werden die *Mixins* angesprochen, so schlägt die Zuordnung der Basisinstanz gelegentlich fehl.

Noch gewichtiger ist der Unterschied beim Zugriff auf *Mixin*instanzen, respektive Rollen. JAsCo redefiniert den *Cast* Operator, um zu einem beliebigen Basisobjekt die passende *Mixin*-Instanz zu liefern beziehungsweise zu erzeugen[12]. Ein solches „Nachschlagen" der Mixininstanz erfolgt dynamisch, erfordert also nur die Angabe des Mixintyps, nicht aber des Basistyps. Dies ist in der untersuchten Situation sehr effektiv einsetzbar, um neue Elemente in einer *Session* zu registrieren. Der Typ der Basisobjekte ist zum Zeitpunkt der Implementierung in der Regel noch unbekannt, weshalb Methoden zur Registrierung von Basisobjekten sich nicht auf solche Typen beziehen können.
ObjectTeams erfordert aber genau dies: Das äquivalente Konstrukt zum Nachschlagen inklusive Erzeugen von Rolleninstanzen ist in ObjectTeams *declared Lifting*. Dieses bedarf der beidseitigen Festlegung: Sowohl der Typ der Rolle, als auch der Typ der Basis müssen zur *Compilezeit* feststehen, was zumindest eine *BlackBox*-Wiederverwendung behindert. Die Methode `getRole` der *Reflection-API* kann die Lücke nicht füllen, da hierdurch keine Erzeugung von Rolleninstanzen ausgelöst wird.
Zwar gibt es durchaus Vorgehensweisen, um auch in ObjectTeams einen vergleichbaren Mechanismus zu emulieren, aber diese erfordern mindestens eine `callin` Bindung auf eine (seiten-)effektfreie Methode der Basisklasse, die obendrein nicht durch andere `callins` gebunden werden sollte und auch möglichst nicht Teil der Schnittstelle sein sollte: Idealerweise also eine gänzlich leere Methode. Daher eignet sich ein solcher Ansatz nicht für *oblivious* Basisklassen. Durch die erforderliche Bindung in jeder Rollenklasse, entspricht der Aufwand dem von *declared Lifting*. Ein solcher Zug ist eher als Notlösung zu betrachten[13].

Dieses besondere Teilproblem unterstreicht einen der grundlegenden Unterschiede zwischen den starr typisierten Aspekten von *ObjectTeams* und den weitgehend abstrakt formulierbaren in *JAsCo*. Dabei sollte aber nicht vergessen werden, dass auch *JAsCos* Aspekte strenger typisiert werden können, was ihre Wiederverwendbarkeit jedoch noch wesentlich stärker reduzieren würde.

6.4.9 Dynamische Aspekte

ObjectTeams und JAsCo verstehen sich als *dynamische* Aspektansätze, was sich jedoch sehr unterschiedlich manifestiert. Um einen Vergleich zu ermöglichen, wurde die Anforderung um einen trivialen Hilfsaspekt erweitert: Einzelne Sessions sollen ein automatisches *Logging* auslösen, andere nicht.

In ObjectTeams war dies sehr zufriedenstellend umsetzbar, allerdings mit der Einschränkung, dass die Abwesenheit von *Pointcuts* die Umsetzung im Umfang vergleichsweise groß ausfallen ließ. Ferner wäre, zumindest in diesem Anwendungsfall, die Möglichkeit wünschenswert, dynamisch Informationen über die Basismethode erhalten zu können. Die praktische Lösung wurde entworfen, indem ein als `ContextElement` fungierendes Team

[12] Zusätzlich existieren entsprechende Funktionen in der API.
[13] Konzeptionell eignen sich z.B. auch Methoden wie `equals` - diese Methoden werden aber unter Umständen sehr oft aufgerufen und können auch zu Endlosschleifen führen. Daher ist dieser Umweg über die Methoden von `Object` nur in Ausnahmefällen sinnvoll.

in der Sessioninstanz registriert wurde. Über die `lost/gotFocus` Methoden wurde dieses gegebenenfalls aktiviert bzw. deaktiviert.

Bei JAsCo lag das Problem weniger in der Beschreibung der Methoden: Das Poincutmodell, mit den verfügbaren Informationen über den der Adviceausführung zu Grunde liegenden *Joinpoint*, verbessert die Generizität dieses Aspekts erheblich. Um eine De-/Aktivierung zu ermöglichen, war die Verwendung eines neuen *Connectors* mit lediglich dem konkreten *Pointcut/ Hookkonstruktor* für die Loggingfunktion nötig.
Für diesen wurde eine neue Klasse entworfen, um – unter Verwendung von *Reflection* Methoden aus Javas, wie auch aus JAsCos Fundus – die Aktivierung zu steuern. Diese wiederum wurde über ein `SessionDependent` *Mixin* an die *Sessionbean* angebunden. Diese Umsetzung verhielt sich erwartungsgemäß, ermöglichte aber keinen sehr feingranularen Umgang. Insbesondere blieb eine Abstimmung und Koordination mit den anderen Aspekten aufgrund der Notwendigkeit eines eigenen *Connectors* weitgehend unmöglich.

6.4.10 Einsatz in einer verteilten und parallelen Umgebung

Eine parallele Nutzung in einer nur zur Einzelplatznutzung entworfenen Basisanwendung ist mit keinem der beiden Ansätze uneingeschränkt möglich, da die zur Synchronisation nötigen Anpassungen die Möglichkeiten schnell übersteigen. Zwar wurde deshalb schon eingangs auf die Formulierung einer dahingehenden Anforderung verzichtet. Zu Untersuchungszwecken wurden aber für parallele Nutzung vorgesehene Versionen der Sessionaspekte kreiert.

Dabei waren zunächst die Möglichkeiten der Nutzung im Zusammenspiel mit dem Logdateibetrachter als einer über RMI kommunizierenden Client/Server Umgebung näher zu untersuchen. Diese Anwendung ist für parallelen und verteilten Zugriff entworfen, die Datenstrukturen der Basisanwendung müssen daher nicht nachträglich parallelisiert werden. Bei Untersuchungen an ObjectTeams wurde offensichtlich, dass eine wirklich verteilte, also auf beiden Seiten der Kommunikation koordiniert stattfindende Nutzung, nicht in Frage kommt. Dies liegt in verschiedenen Beschränkungen der Sprache begründet, die sicherlich in ihrer Gesamtheit der noch unfertigen Implementierung geschuldet sind:

- Die aktuelle ObjectTeams Version erlaubt nicht die Verwendung von *Interfaces* als Basistypen. Die *Stubs* von entfernten Objekten sind aber nur über die jeweiligen *remote-Interfaces* typisiert. Daher ist es nicht immer möglich, mit Aspekten das Kommunikationsverhalten auf der Clientseite zu beeinflussen.

- Rollen, auch ungebundene, können ihrerseits nicht direkt als entfernte Objekte verwendet werden. Zunächst liegt das daran, dass ObjectTeams keine[14] *Rollenkonstruktoren* mit `throws` Deklaration zulässt. Dies verhindert den Einsatz von `UnicastRemoteObject`, kann aber durch manuelles Exportieren umgangen werden.

[14] Gleich, ob es sich um *Lifting* oder herkömmliche Konstruktoren handelt.

Allerdings ist RMI selbst dann nicht in der Lage, *Stubs* und *Skeletons* für Rollenklassen zu erzeugen. Daher ist ein Einsatz von Rollen als RMIServer Objekte nicht möglich.

- Rollen/Basis Zuordnung: ObjectTeams verlässt sich auf `WeakHashMap` und damit mittelbar auf `hashCode` und `equals`, um Rollen ihren Basisinstanzen *zuzuordnen*. Dies führt beim Umgang mit Referenzen, etwa `RemoteReference` Instanzen mitunter zu nicht richtigen Ergebnissen.

- Bezüglich gewobener Aspekte besteht eine *Symmetriepflicht*: Damit RMI den Austausch der Daten mittels Serialisierung zulässt, besteht die Notwendigkeit, bei allen möglicherweise als Parameter- oder Rückgabetyp für entfernte Methodenaufrufen dienenden Klassen, genau die gleichen Aspekte auf beiden Seiten zu weben. Dies gilt auch für verteilte Funktionen der Basisanwendung.

Erfolgreicher war der auf eine Seite beschränkte Einsatz, insbesondere der serverseitige. Dies folgt daraus, dass Implementierungen – anders als *Remote Interfaces* – gültige Basisklassen darstellen.

Als Untersuchungsszenario wurde daher der Logdateibetrachter in seiner Inkarnation als *Serveranwendung* gewählt. Die Anforderungen entsprechen weitgehend derer im normalen Einsatz: *Authentikation* bei der Anforderung eines *Logs* und Bewahren der Authentisierungsinformation über die folgenden Zugriffe hinweg.
Die für diesen Einsatz nötigen Anpassungen waren durch Vererbung ohne Eingriffe in das Gerüst der Aspekte zu lösen. Insbesondere umfassten sie den Wechsel auf Referenzierung der aktuellen *Session* durch `ThreadLocal` Instanzen, also der Einführung einer aktiven *Session* für jeden *Thread*. Da ObjectTeams keine threadlokale Aktivierung von Teams erlaubt, war es nötig, eine solche Funktionalität als *Guard Predicate* zu implementieren, was als funktionierende, aber nicht uneingeschränkt als befriedigende Lösung, angesehen werden kann.
Ein weiteres Problem tritt durch die Implementierung von RMI selbst auf, die keine Zuordnung von *Threads* zu *Clients* erlaubt und nur sehr eingeschränkt Informationen über die Quelle von Aufrufen propagiert. Das Einhalten der Konsistenz des Aspektzustandes ist somit komplizierter als es sein müsste.

Der Aufwand an Verbindungscode war vergleichbar hoch und erforderte darüber hinaus noch die Implementierung eines neuen `CallbackHandlers` nebst zugehöriger Infrastruktur, um die Eingabe von Authentisierungsinformationen durch den Nutzer des *Clients* zu ermöglichen. IP-basierende Authentikation war auch ohne solche – nicht invasive – Erweiterungen der Clientanwendung möglich. Dieser Zusatzaufwand wäre teilweise durch entfernt verfügbare Rollen vermeidbar gewesen, was aber an den genannten technischen Gründen scheiterte. Zusammenfassend war ein Einsatz der ObjectTeams Lösung in der Client/Server Umgebung möglich, aber sowohl hinsichtlich der Parallelität als auch der Verteiltheit beschränkt.

Übertragen auf *JAsCo* ergaben sich ähnliche Hürden. Probleme mit der Synchronisierung waren hierbei nicht ausschlaggebend, da der ursprüngliche Entwurf bereits eine pseudoparallele Struktur von Sessions mit weitgehenden zustandslosen *Hooks* erforderte. Wie bei ObjectTeams scheitert der Datenaustausch bei ungleichen Aspekten. Diese Pflicht zu symmetrischen Aspekten tritt sogar noch ausgeprägter auf, da JAsCo Basisklassen auch ohne explizit gewobenen Aspekt verändert [15]. Zusätzlich wird die Herstellung der Symmetrie durch das *Weaver*-Konzept und dessen dynamisches Verhalten erschwert. Daher erfordert der Einsatz sehr gezielt entworfene *Pointcuts*, die keine *Joinpoints* in der Clientanwendung erfassen, aber dennoch zum Einweben der Aspekte führen.
Ergänzend ist noch zu erwähnen, dass keiner der Ansätze eine einfache Lösung für den angesprochenen Sessions-lokalen *Logging* Aspekt in einer parallelen Umgebung bot.

6.4.11 Einsatz allein stellender Features

Beide Ansätze verfügen über vollkommen neuartige *Features*. Bei ObjectTeams wären dies etwa *Confined Roles*, bei JAsCo *Distributed Aspects*, *Stateful Aspects* und *Adaptive Programming*. Diese Features wurden untersucht, werden hier jedoch wegen der Unmöglichkeit eines Vergleichs außer Konkurrenz gestellt.

Confined Roles ermöglichen es in ObjectTeams den Zugriff auf externalisierte Rollen zu verhindern, um eine stärkere *Encapsulation* zu erreichen. In dem gesetzten Szenario wäre dies eine sehr interessante Option für `Session` gewesen, um den benachrichtigten *Observern* nur eine eingeschränkte Sicht auf das System zu erlauben.
Dieses Vorgehen versprach eine saubere Trennung der Interfaces und eine stärkere Trennung der verschiedenen Operationen. Da die Unterstützung für *nested Teams* aber noch nicht hinreichend fortgeschritten ist, musste dieses Detail aus späteren Entwürfen und Implementierungen entfernt werden.

Ein anderes wertvolles ObjectTeams Feature ist die *Decapsulation*: Die Möglichkeit, die Sichtbarkeit von Klassenmitgliedern zu übergehen, um den Anforderungen von *Crosscutting Concerns* in einem *oblivious* Umfeld gerecht zu werden. Konkret wurde damit in ObjectTeams basisgewandte Kommunikation möglich wo JAsCo einen Eingriff in die Quellen der Basisprogramme zwingend erforderte.

Bei JAsCo findet sich eine ganze Reihe besonderer Features, die eingangs für das gestellte Problem als sehr hilfreich angesehen wurden, obgleich sich diese Erwartung nicht uneingeschränkt umsetzen ließ.
Zunächst sei hier *Distributed JAsCo* angeführt, welches mächtige verteilte Aspekte bieten soll. Die vorliegende Version ist aber weder technisch noch konzeptionell weit genug fortgeschritten, um Verwendung zu finden. Daher wurde zwar untersucht, ob sich die Möglichkeit *Advice* auf anderen *Virtual Machines* auszuführen, nutzbringend einsetzen lässt; es wurde aber darauf verzichtet, dieses Feature fest in den Entwurf einzubinden.

[15] Konkret reicht bereits die Verwendung in einem `cflow` Pointcut.

Als Grundlage wurde erneut der Logdateibetrachter als Client/Server Anwendung gewählt, wobei der *LoginAspekt* als Reaktion auf einen serverseitigen *Pointcuts* ausgeführt werden sollte.
Die gehegte Hoffnung war es, so die verteilte Authentikation, die in der Lösung ohne distributed JAsCo als aufwändig zu bezeichnen ist, klarer und knapper ausdrücken zu können. So wäre z.B. die sonst komplizierte Operation des Referenztauschs zwecks Aufrufs des `Callbackhandler`s entfallen; explizit clientseitige Aspekte wären nicht notwendig geworden.
Wie bereits angedeutet, waren die praktischen Ergebnisse anders gelagert. Schon die Wiederverwendbarkeit des Konstrukts ist in der gegenwärtigen Reife praktisch nicht vorhanden. Der Einsatz erfordert die Designatoren `joinpointhost(<ip>)` bzw. `executionhost-(<ip>)`, welche jeweils zur *Compilezeit* der *AspectBean* – nicht des *Connector*s – die IP-Adresse der kommunikativen Gegenstücke fixieren. Jenseits dieser Einschränkung fällt die unspezifizierte Semantik verteilter *Advices* auf, die auch nur sehr eingeschränkt aus dem Laufzeitverhalten erschlossen werden konnte. Die Ausführung von *Advice* auf einer anderen *Virtual Machine* findet auch im Kontext dieser – anderen – *Virtual Machine* statt. Der Zustand der AspectBean auf der Seite des *Connectors* ändert sich nicht.
Somit eignet sich das Konzept für den gewünschten Zweck in keiner Weise, da gewonnene Authentikationsinformationen nicht zurück zum Server gelangen. Schließlich lässt auch das Fehlen jeglicher Sicherheitsüberlegungen die Unreife erkennen.

Stateful Aspects wurden während der Untersuchung nur eingeschränkt berücksichtigt. Bereits bei einer oberflächlichen Begutachtung stellte sich das Konzept als nicht hinreichend flexibel heraus, um Protokollen in Anwendungen zu begegnen. Insbesondere die fehlende Möglichkeit, Transitionen zu unterbinden – etwa im Falle eines gescheiterten Login-Vorgangs – verhinderten einen sinnvollen Einsatz. Darüber hinaus neigt die Menge der nötigen Zustände zu exponentiellem Wachstum, wenn Protokolle auf einer Menge von Ereignissen basieren. Die Idee, beispielsweise das Starten/Wechseln/Beenden von Sessions über *Stateful Aspects* zu verwirklichen, scheiterte an der Unmöglichkeit einen Zustandswechsel dynamisch zu unterbinden.

Das in dem vorliegenden Aufbau wohl am besten einsetzbare *JAsCo Feature* war die Spracherweiterung für *Adaptive Programming*, die eine auf *AspectBeans* basierende Anbindung von *DJ*[16] bietet. Die Umsetzung ist dergestalt konzipiert, dass *Hooks* normaler *AspectBeans* durch eine spezielle Art von *Connector* als adaptive *Visitors* eingesetzt werden können. Diese *Vistors* traversieren den Klassengraphen von einer beliebigen Wurzel aus und tätigen Operationen auf den besuchten Knoten. Dabei erlaubt JAsCo durch `refinable` Methoden und *Refining Classes* diese Operationen auf einer hohen Abstraktionsebene zu formulieren.
Konkret wurde die Eignung des Systems überprüft, Kontextänderungen – Sessionwechsel – „*auszurollen*“, also die Anpassung der GUI im Falle eines Sessionswechsels im Dispositionssystem umzusetzen. Dafür wurde eine entsprechende Hilfsklasse geschrieben, die über

[16] `http://www.ccs.neu.edu/research/demeter/DJ/`

die bestehenden Schnittstellen in das Sessionsystem eingefügt wurde. Praktisch zeigt auch diese Erweiterung die hinreichend bekannte Einschränkung hinsichtlich Klassen aus der Java-Api, was den Nutzen leider zu einem gewissen Grad reduziert. Perspektivisch gelang so das zur Anpassung des Basiszustandes nötige Verfolgen der Referenzen sehr viel besser, als es die Lösung ohne *Adaptive Programming* gestattete. Da eine solche „Verfolgung“ für den Logdateibetrachter nicht nötig war, fehlt die Möglichkeit einer qualifizierten Aussage bezüglich der Wiederverwendbarkeit; die erstellte *Bean* konnte aber hochgradig abstrakt gehalten werden.
Insbesondere gelang so das serielle Anpassen referenzierter Objekte wesentlich einfacher und sauberer modularisiert. In ObjectTeams war für vergleichbare Situationen der Rückgriff auf – oft mehrere – *callout* Methoden ohne weitere Funktion nötig. Allerdings scheint das Laufzeitverhalten der gegenwärtigen *Adaptive Programming* Implementierung noch nicht hinreichend ausgereift für eine konsequente Verwendung, da der Aufruf die Ausführung deutlich verzögert.

6.5 Versuch einer Bewertung

Die erzielten Resultate bleiben teilweise hinter den anfangs gehegten Erwartungen zurück. Allerdings kann dies nicht völlig den Ansätzen der untersuchten Techniken zugeschrieben werden. Denn die Wiederverwendung von Aspekten ist ein bewegtes Feld, für welches bislang noch kein so weitreichender Erfahrungsschatz vorliegt, wie etwa bei der Objektorientierung. Selbst bei der Objektorientierung wurde noch kein Prozess gefunden, der Wiederverwendung garantieren könnte, einen solchen für die Aspektorientierung einzufordern wäre demnach übereilt.

Darüber hinaus muss wiederholt werden, dass bei beiden Sprachen fehlende Implementierungen die einsetzbaren Fähigkeiten der Konzepte beschränken. Hier sind insbesondere die Bindung von Rollen an *Interfaces* in *ObjectTeams* und der `call` *Pointcutdesignator* in *JAsCo* zu nennen.

Auf der positiven Seite ist zu bemerken, dass durchaus funktionierende und weitgehend generische Implementierungen für die Mehrheit der betrachteten Szenarien erstellt werden konnten. Dabei waren im Endeffekt zwar einige Entwürfe nicht umsetzbar, funktionsgleiche Auswege konnten aber gefunden werden. Einzelne Ansätze, etwa die Verwirklichung über einen zweistufigen *Connector*, wie in Abschnitt 5.2.6 diskutiert, waren zwar nicht zuverlässig möglich[17], auch aus jenen waren aber Erkenntnisse für die vorgestellten Versionen zu ziehen.

Weitere erwähnenswerte Ausnahmen wurden dabei insbesondere im Umfeld *RMI* gefunden, welches mit keinem Ansatz wirklich befriedigend einsetzbar war. Einzig serverseitig war eine sinnvolle Umsetzung möglich.
Auch die JAsCo Funktionen für verteilte Aspekte stellten sich als nicht hinreichend entwickelt heraus, um in den gewählten Szenarien verwendet zu werden. Die extrem lange

[17] Die Implementierung funktionierte nur in trivialen Einsatzszenarien; ein Einsatz auf Basis des Dispositionssystems scheiterte.

Ausführungszeit lässt *Adaptive Programming* als praktisch nicht einsetzbar erscheinen.

Die Wiederverwendbarkeit der Implementierungen war im Rahmen der Fallstudie als gegeben zu bewerten. Funktionsfähige Implementierungen waren mit wenig Aufwand zwischen den verschiedenen Basisanwendungen übertragbar. Einschränkend ist hier einzuwenden, dass der Einsatz von guten Kenntnissen der internen Strukturen dieser Anwendungen abhing. Für unbekannte *third-party* Anwendungen ist ein Einsatz unrealistisch, da die erforderlichen Verbindungsschichten explizite Abhängigkeiten von Implementierungs- und Strukturdetails der Basisanwendungen erfordern. Die in Kapitel 3 formulierte Feststellung, *dass Basisprogramme Frameworks für ihre Aspekte sind*, lässt sich in der Nachbetrachtung für die untersuchten Techniken enger fassen: „*Basisprogramme sind White-Box Frameworks für ihre Aspekte*“.

Eine interessante Beobachtung, die im Fortgang der Fallstudie gemacht werden konnte, ist die Möglichkeit der Verwendung der Aspekte in einem objektorientierten Kontext. Wird in den Verbindungselementen die Flexibilität eingeschränkt, so lassen sich beide Implementierungen objektorientiert einsetzen. Eine Bindung an einen abstrakten Typ, der dann mittels Vererbung eingebunden werden kann – und schon entspricht der Einsatz weitgehend dem Klassischen. Im Falle von JAsCo ist so sogar möglich, durch die Bindung an *Interfaces* Strukturen zu kombinieren[18].

[18] Dies ähnelt sehr der Mehrfachvererbung, hat aber wegen der basisseitigen Vererbungssemantik der ’+’ bzw. ’++’ Operatoren ein anderes Verhalten.

7 Folgerungen

Im Laufe der Fallstudie wurde eine Reihe von Problemen und kontraintuitiven Situationen beobachtet. Die untersuchten Techniken werden alle noch erforscht. Deshalb erscheint ein kritischer Umgang gerechtfertigt und kann die Entwicklung unterstützen.
Ohne einen Anspruch auf Vollständigkeit zu verfolgen, wird eine Auswahl von solchen Beobachtungen angeführt, oft mit – möglicherweise nicht umsetzbaren – Lösungsvorschlägen. Diese beruhen zum Teil auf dem direkten Vergleich von ObjectTeams mit JAsCo; in anderen Situationen wurden AspectJ und JAC (siehe Abschnitt 2.2.2.1) als Vorbilder herangezogen.
In einigen Fällen sind auch Ideen gänzlich ohne Vorbild aufgeführt. Hierbei wird das Ziel verfolgt, Diskussionsansätze für zukünftige Entwicklungen zu bieten.

7.1 JAsCo

Die bei JAsCo zu bemängelnden Details liegen zu einem großen Teil im Gebiet der Pointcutsprache und in deren Aufteilung auf *Connector*s und *Hook*s. Darüber hinaus ergaben sich einige Unstimmigkeiten in der Dokumentation und die fehlende Ausdrucksmächtigkeit in einigen Situationen.

7.1.1 Aufteilung Bean/Connector konsequenter gestalten

In diesem Zusammenhang ist die konsequentere Umsetzung von `target` hinsichtlich abstrakter *Pointcut* Designatoren zu fordern: Wenn der lexikalische Teil mittels abstrakter Pointcutparameter in Connector-Dateien stattfindet, so stellt sich die Frage, warum `target` die Klasse unabänderlich in der `AspectBean` festlegen soll.
Da die Umgehung dieses Unterschiedes mit dynamischen Mitteln umständlich und performanzintensiv ist, schlage ich die Einführung von abstrakten Klassenpointcuts (i.e. `target`) analog zu den integralen abstrakten Methodenpointcuts vor.

7.1.2 Flexiblere Pointcuts I : „Echter“ `cflow`

Der in JAsCo implementierte `cflow` Pointcutdesignator basiert auf einem[1] `execution` Pointcut. Damit ist es – durch die an AspectJ angelehnte `execution`-Semantik – nicht möglich, alle Methoden einer Klasse als *Joinpoints* für `cflow` auszuwählen. Auch ist es nicht einfach möglich dynamische Bedingungen in die Gültigkeit mit einzubeziehen[2].

[1] Auch verodert mehrere.
[2] Mögliche Lösungen benötigen Aspekte auf Aspekten und mindestens einen Hilfsaspekt. Dafür sind Einstellungen an der Laufzeitumgebung erforderlich.

Daher sollte zumindest ein zusätzlicher auf `call`[3] aufbauender `cflow` Ausdruck in die Sprache aufgenommen werden.
Aber auch ein solcher Ausdruck könnte das Problem mangelnder Dynamik nicht lösen; eine wirkliche Erweiterung, um `cflow` in der Mächtigkeit von *AspectJ* zu erhalten, würde weitreichende Änderungen für das *Pointcut/ Connector* Modell erfordern.

In diesem Kontext ist noch ein weiteres Problem mit den vorhandenen `cflow` und `withincode` *Pointcutddesignatoren* zu nennen. Genauer gibt es das direkt zu dem in Abschnitt 7.2.7 beschriebenen, umgekehrte Problem bei JAsCo zu beobachten: Wird ein *Pointcut* mit einem '`+`' oder '`++`' versehen, so führt jeder `super` Aufruf, gleich ob Konstruktor oder Methode, in einer Unterklasse des ursprünglich angegebenen Typen zu einer erneuten Ausführung des *Advices*. Um dieses Verhalten zu unterbinden, ist – abgesehen von dynamischen *Workarounds* – entweder die Einführung einer *Annotation* in der Basisanwendung oder der Einsatz von `!cflow` bzw. `!withincode` *Pointcuts* nötig. Da nur der Weg über solche zusätzliche *Pointcuts* ohne invasive Änderungen möglich ist, bedeutet dies gleichzeitig, dass bereits im Hookkonstruktor eine Entscheidung bezüglich des Verhaltens bei Rekursion und `super`-Aufrufen getroffen werden muss. Das reduziert die Wiederverwendbarkeit von *Hooks*. Es wäre daher wünschenswert, *nachgeordnete* Pointcutdesignatoren, wie `withincode` auch nachträglich ergänzen zu können, beispielsweise durch Vererbung oder durch mächtigere *Connector* Elemente.

7.1.3 Pointcuts II: Semantik

JAsCos Pointcutmodell ist intentionell sehr dicht an AspectJs durchaus diskussionswürdigen Ansatz gehalten. Es ist für eine dynamische Sprache zwar eigentlich nicht notwendig die Eigenheiten aus AspectJs statischem Modell zu übernehmen, aber sicherlich ist das Modell von AspectJ das bekannteste.
Es wäre daher für JAsCo wünschenswert, mehr *Joinpoints* freizulegen, da die momentane Einschränkung vieles erschwert, insbesondere mit der gegenwärtigen Implementierung, die nur `execution` *Pointcuts* und abgeleitete Ausdrücke kennt. Da eine Abkehr von dem momentanen Pointcut Ansatz aber unrealistisch wäre, möchte ich doch zumindest drei Veränderungen vorschlagen:
Erstens wäre für die Pointcutsprache eine klare formelle Spezifikation dringend erforderlich. Die momentane Situation erschwert den Umgang erheblich.
Zweitens fehlt ein klares Bekenntnis für oder gegen die AspectJ Pointcutsprache. Auf den ersten Blick entspricht JAsCo AspectJ, jedoch ergeben sich bei einem Vergleich erhebliche Unterschiede. (Vgl. [BFTY04] und 5.1.1.1) Diese haben zur Folge, dass manche *Joinpoints* in JAsCo nicht direkt ausgewählt werden können.
Drittens und weniger relevant sollte die statische Prüfung im `Connector`-*Compiler* im Falle eines leeren, also – scheinbar – niemals zutreffenden, Pointcuts eine Warnung anstelle eines Fehlers ausgeben, da dieses Verhalten die Ausdrucksmächtigkeit empfindlich reduziert. Um die gegenwärtige Situation zu illustrieren: Ein Ausdruck `* *.A.f+(*)` wird

[3] Sobald implementiert

als Fehler gemeldet, wenn eine Klasse`A` keine Methode `f` definiert. Sollte eine – möglicherweise dynamisch nachgeladene – Unterklasse von `A` eine Methode `f` definieren, so könnte diese – in der vorliegenden Version von JAsCo – nicht ohne *Wildcard* in dem *Pointcut* erfasst werden.
Ein weiteres Problem ist die fehlende Unterscheidung zwischen Konstruktoren, statischen und nichtstatischen Methoden. Während die ersten beiden nicht auf der Basis eines Objekts ausgeführt werden, ist dies bei letzteren der Fall. Die dynamisch nötige Unterscheidung nach `thisJoinPointObject != null` löst das Problem, aber vordefinierte *Pointcuts* nach dem Vorbild JACs (siehe 2.2.2.1)[4], z.B. `CONSTRUCTORS`, `STATIC`, etc. wären durch die Möglichkeit der Angabe im *Connector* eine erwägenswerte Erweiterung.

7.1.4 Klare Sprachdefinition

Die Sprache JAsCo wird momentan in einer Reihe von Aufsätzen und Webseiten definiert[5]. Es fehlt an einer klaren Sprachdefinition hinsichtlich Syntax und Semantik. Einige der publizierten Aufsätze sind widersprüchlich, was sicherlich der laufenden Fortentwicklung der Sprache anzulasten ist.
Eine klare formale Spezifikation der Sprache, ergänzend zu einem vollständigen und konsistenten informellen Teil, würde den Umgang mit JAsCo, insbesondere bei der Fehlersuche, erheblich vereinfachen. Auch wäre so öfter eine klare Antwort auf die wiederkehrende Frage: *„Bug oder Feature“* ohne Konsultation möglich.

7.1.5 Stateful Aspects für komplexere Protokolle

Stateful Aspects (siehe 2.2.1.11) sind ein sehr vielversprechender Mechanismus zur Handhabung von *Jumping Aspects*. Ihre praktische Nutzbarkeit wird jedoch dadurch reduziert, dass die Reihenfolge der Zustände festgelegt ist und nur statische Pointcuts erlaubt. Dies lässt sich kaum mit dynamischen Mitteln beeinflussen, da reflektiver Zugriff auf den Zustand nicht möglich ist.

So war ihr Einsatz in der Fallstudie nicht sinnvoll möglich, insbesondere die fehlende Steuerbarkeit über dynamische Bedingungen ließen einen Einsatz wenig sinnvoll erscheinen (siehe 6.4.11).

Daher schlage ich zunächst drei Erweiterungen dieser Technik vor:
Erstens sollten komponierte *Pointcuts* möglich sein, also Transitionen, die ihrerseits zustandsbehaftet sind. Die Begründung für diese Forderung liegt in dem Wesen vieler Protokolle, mehrere voneinander unabhängige Aktionen als Grundlage für einen Zustandswechsel zu verwenden. Als Beispiel sei folgendes Szenario angeführt: Eine Komponente soll erst aktiviert (Transition: `inaktiv` → `aktiv`) werden, wenn alle ihre Konfigurationsmethoden aufgerufen wurden; dabei ist die Reihenfolge dieser Aufrufe irrelevant. Ein derartiges Protokoll ist momentan jenseits der Möglichkeiten von *Stateful Aspects*, da

[4] JACs Auswahl an vordefinierten Pointcuts wurde im Rahmen der AOPAlliance reduziert.
[5] [FVSB05],[SVJ03],[Van06],[Van02],[Van04],[VS04a],[VS04b],[Van05]...

der zugrundeliegende endliche Automat sich exponentiell mit steigender Zahl der Konfigurationsmethoden vergrößern würde. Zur Auflösung dieses Problems schlage ich die Einführung eines Operators vor, um Kombinationen als Bedingungen für Transitionen angeben zu können.
Zweitens schlage ich vor, reflektiven Zugriff auf den aktuellen Zustand zu erlauben; sowohl um den gegenwärtigen Zustand zu erfragen als auch mit der Möglichkeit, programmatisch den Zustand zu ändern.
Drittens ist es momentan zwar möglich, `isApplicable` Konstrukte für jeden Zustand anzugeben, nicht aber für Transitionen. Dynamische Bedingungen sind aber meines Erachtens für Transitionen interessanterer Protokolle zwingend erforderlich.

7.1.6 Zugriff auf Felder

Zugriff auf Felder und Freilegung von Feldzugriffen als *Joinpoints* für *Pointcuts* mögen dem *JavaBean* Gedanken widersprechen, sind aber notwendige Mittel für die Anpassung von Aspekten an verschiedene Basisanwendungen. Daher sollte zumindest über die Einführung von `get` und `set` Pointcut Designatoren nachgedacht werden, um das an AspectJ angelehnte Pointcutmodell abzurunden. Es liegt im Wesen von *Crosscutting Concerns*, dass sie nicht unbedingt entlang der öffentlichen Schnittstelle von Klassen verlaufen, allzu oft nicht einmal entlang den privaten Schnittstellen.

So wurde es in der Fallstudie erforderlich, die Basisanwendungen zu verändern, um den Zugriff auf benötigte Informationen zu erlauben (siehe Abschnitt 6.4.3).
Daher schlage ich die Erweiterung von JAsCo um eine Möglichkeit vor, die Klassenstruktur aufzubrechen, mindestens in der Auswahl der freigelegten Joinpoints, idealerweise aber deutlich darüber hinaus: Zugriff auf Felder der Basisobjekte, wie etwa *privileged Aspects* in *AspectJ* oder *Decapsulation* in *ObjectTeams*, durch Methoden von *Refining Classes*, wäre beispielsweise eine sehr hilfreiche Erweiterung.

7.1.7 Connectors stärker formalisieren

Der JAsCo Connector-Compiler überwacht nur Syntax, Typen und Zahl der abstrakten Pointcutparameter. Wünschenswert wäre aber eine statische Möglichkeit, den Aufruf von Initialisierungsmethoden ebenso zu überwachen wie auch die Auswahl von *Aspect Factories*. Ich bin nicht der Ansicht, dass die Wahl der AspectFactories ihren Platz ausschließlich im Connector haben sollte, da Aussagen über Multiplizitäten der Aspekte/AspectBean/Basis Bindungen so zur Zeit der Aspektimplementierung kaum möglich sind. Dies kann Wiederverwendung mitunter erheblich erschweren, mindestens aber den Dokumentationsaufwand unnötig erhöhen.
AspectBeans sollten die für einen *Hook* zulässigen *AspectFactories* angeben können, wobei auch eine nicht durch den Compiler überprüfte Dokumentationsnotation als hinreichend zu erachten wäre.

Zusätzlich schlage ich die Einführung eines Schlüsselwortes, z.B. `essential`, vor, um

erforderliche Initialisierungsmethoden für Hooks und AspectBeans zu kennzeichnen. Dies ist in der Abwesenheit von Java-Konstruktoren für Hooks begründet.

Die aber wohl wichtigste Forderung aus diesem Zusammenhang ist die nach einem Vererbungskonzept für *Connector*-Dateien. Wie in den Abschnitten 6.4.2 und 6.4.6 festgestellt, erschwert das fehlende Wiederverwendungskonzept für *Connector*-Dateien die Integration verschiedener Aspekte ebenso, wie die – hypothetische – Wiederverwendung solcher Teilintegrationen. Möglicherweise wäre eine AspectJ-ähnliche Spezialisierung abstrakter Verbindungen ein gangbarer Weg, hier eine Verbesserung herbeizuführen.

7.1.8 „Smarte“ Auswahl von Refining Classes

Der Algorithmus zur Auswahl der jeweils passenden *Refining Class* (siehe Abschnitt 2.2.1.6) für einen beliebigen *Hook* wählt die erste zur Basis passende Implementierung aus. Dies bedeutet z.B., dass es in Anwesenheit eines *Refinements* für eine in der Klassenhierarchie der Basis weit oben liegende Klasse nicht zuverlässig möglich ist, in *Refining Classes* speziellere *Refinements* für speziellere Klassen anzugeben. Als Folge war im Rahmen der Fallstudie der Ausweg über die Implementierung in Connector-Dateien nötig, was die Lokalität des Codes mitunter erheblich verschlechterte. (siehe 6.4.7)
Ein wirklich zu *Smart Lifting* (siehe Abschnitt 2.3.1.4) vergleichbarer Mechanismus würde die Mächtigkeit dieses Instruments erheblich verbessern.

7.2 ObjectTeams

Bei der Arbeit mit ObjectTeams, ensteht häufig der Wunsch nach Möglichkeiten, auf einer höheren Abstraktionsebene mit Rollen arbeiten zu können. Die gegenwärtig in der Sprache vorhandenen Mittel, Rollen praktisch zu beherrschen, verlangen oftmals nach konkreten, gebundenen Rollen. Eine solche Bindung kann aber ohne Kenntnis der späteren Basisklassen nicht formuliert werden. So gibt es empfindliche Einschränkungen beim Umgang mit *declared Lifting*, *(base) Guard Predicates* und auch mit *callin* Methoden.
Vereinfacht formuliert fehlt ein Äquivalent abstrakter Methoden auf Teamebene für die Übergänge von der Basis in den Aspekt.

7.2.1 Flexibles explizites Lifting

Der Mechanismus des *declared Liftings* sollte meines Erachtens stärkere Unterstützung für Polymorphie bieten. Momentan ist der Ausdruck

```
someMethod(Baseclass as Roleclass identifier)
```

auf beiden Seiten des „`as`“ festgelegt: Auf der linken Seite sind nur einzelne Klassen[6] als Basis gültig; gerade ObjectTeams verspricht aber die Abkehr von der Struktur der Basis, um eine eigene Struktur parallel zu ermöglichen. Nicht in einer Vererbungshierarchie stehende Typen können hier nicht polymorph angegeben werden; auch *Overloading* genügt

[6] Sowie deren Unterklassen.

hier nicht, da dieses auf dem statischen Typ basiert.
Zur rechten Seite des Ausdrucks steht zu bemerken, dass nicht die Mächtigkeit von *Smart Lifting* zum Tragen kommen kann, da sonst *Overloading* problematisch werden könnte.

7.2.2 Polymorphie auf der Rollenseite

Die Erweiterung der Rollenseite des *declared Liftings* um ein polymorphes Mittel abwärts der Vererbungshierarchie sollte erwogen werden. In der aktuellen Definition erwartet die rechte Seite eines '`as`' Ausdrucks eine Rollenklasse, die an die linkstehende Klasse gebunden ist.

Es würde aber mit einigen Einschränkungen – den Regeln für *Smart Lifting* folgend – reichen, wenn eine Subrollenklasse im gleichen Team an die, auf der linken Seite des Ausdrucks stehende, Basisklasse gebunden wäre.

Mit der gewonnenen Flexibilität wäre eine dynamische Auswahl des eigentlichen Rollentyps möglich, was eine Vielzahl vielversprechender Entwurfsmuster erlauben würde.

7.2.3 Polymorphie auf der Basisseite

Eine besondere Stärke im Design mit ObjectTeams ist die Möglichkeit, nicht in einer Vererbungs/Typbeziehung stehende Klassen auf eine Rollenhierarchie abzubilden. Über den überstehenden Punkt hinaus könnte diese Eignung noch um ein Vielfaches verstärkt werden.
Der Operator für *declared Lifting*, `as`, erlaubt nur Ausdrücke, welche ein polymorphes *Lifting* für Basisklassen mit einem gemeinsamen Supertyp auslösen. In generischen Teams kann aber zum Zeitpunkt der Implementierung noch keine Aussage über eine solche Beziehung gemacht werden; Wiederverwendbarkeit wird demnach empfindlich erschwert. So ist es momentan kaum möglich, auf hoher Abstraktionsebene Funktionen mit einem (abstraktem) Basisbezug vorzugeben; solche Konstruktionen bedürfen immer eines expliziten Adapters in einer gebundenen Verfeinerung (siehe 6.2.1.3). Ein solcher wiederum kann auch nur auf der Verbindungsebene Verwendung finden. Ein Äquivalent abstrakter Methoden, um Teile der basisbezogenen inter-Team Kommunikation auf einer hohen Abstraktionsebene zu formulieren, fehlt momentan.

```
1  public abstract team class MyTeam {

3   public void shareBase( Role){
            ...
5   }

7   public abstract class Role {...}
    ...
9   }
  }
11
  public team class ConcreteTeam extends MyTeam {
```

```
13
   public void shareRole(Base as Role b){
15  shareBase(b);
   }
17
   public class Role playedBy Base{...}
19 ...
   }
21 }
```

Listing 7.1: Momentan nötiger Umweg, um eine Rolle in Abhängigkeit einer Rolle zu erzeugen (z.B. zum Austausch von Basisreferenzen zwischen zwei Teams). Dies ist in manchen Situationen gänzlich unmöglich.

Eine vorgeschlagene Lösung ist die Erweiterung von ObjectTeams, um Aufzählungen wie `someMethod(Baseclass1, Baseclass2, Baseclass3,... as Roleclass role)` zu erlauben. Dies würde in der Tat die Umsetzung spezifischer Entwürfe in einer wesentlich eleganteren Weise ermöglichen, greift aber hinsichtlich der Wiederverwendbarkeit zu kurz. Ein Ausdruck dieser Art kann in einem statischen Typsystem kaum erweiterbar gestaltet werden.
Dies liegt in der nicht abgeschlossenen Menge der Rollenklassen begründet: Die Menge der Rollenklassen steht erst nach der Kompilation fest. In erbenden Teams können mehr Basisklassen möglich sein als zum Zeitpunkt der ursprünglichen Definition bekannt. (Listings 7.2 und 7.3) Eine endliche Aufzählung von gültigen Basistypen wäre aber geschlossen, müsste hinter der Evolution des Systems zurückbleiben. Auch wäre eine Aufzählung expliziter Basistypen sehr vom expliziten Kontext des Einsatzes abhängig und damit kaum wiederverwendbar. Letztlich wäre also gegenüber *Overloading* durch einen solchen Aufzählungsausdruck nichts gewonnen.

```
public team class MyTeam {
2
  public void doSomething(Base1, Base2 as Role){
4         ...
  }
6
  public class Role {...}
8
  public class Role1 extends Role playedBy Base1 {...}
10 public class Role2 extends Role playedBy Base2 {...}

12 ...
  }
14 }
```

Listing 7.2: Eine Aufzählung ist nur auf den ersten Blick ein guter Ausweg. Nur für ein einzelnes Team wäre eine Aufzählung hinreichend.

```
public team class SubTeam extends MyTeam {
2
```

```
  public class   Role3 extends Role playedBy Base3 {...}
4
  ...
6
}
```

Listing 7.3: Aber was nun? `doSomething` müsste redefiniert werden, um die neue Rollenklasse einzuschließen – das würde aber erhebliche Fragen hinsichtlich der Konformität zur ursprünglichen Methode aufwerfen.

Die Möglichkeit, zu einem spezifischen Rollentyp *geliftet* werden zu können, ist eine grundlegende Eigenschaft von Klassen in ObjectTeams, über die es unbedingt möglich sein sollte zu quantifizieren.
Die Bedeutung der *Liftbarkeit* zu einem gegebenen Rollentyp ist vergleichbar wichtig wie die Zugehörigkeit zu einem als *Extension Point* dienenden Typ in objektorientierten Frameworks.

Ich schlage daher die Einführung einer Gruppe von generischen Pseudotypen vor, die für jede Rollenklasse automatisch erzeugt werden sollten: Z.B. `LiftableTo<RoleClass>`. Basisklassen, an welche eine Rolle x gebunden ist, sollten die leere pseudo-Schnittstelle LiftableTo<x> implementieren. Implizit existiert bereits ein entsprechender Ausdruck, in Form der zulässigen Rollentypen für die rechte Seite einer *Callout* Bindung.

So wäre

```
someMethod(LiftableTo<RoleClass> as Roleclass identifier)
```

ein typsicherer, robuster und erweiterbarer Ausdruck, um das umrissene Problem zu lösen.

7.2.4 Abstrakt gebundene Rollen

Gelegentlich stellt sich die Situation, dass eine Rollenklasse nur als Supertyp für andere Rollenklassen dienen soll, um etwa eine Klassenhierarchie der Basis auf eine Hierarchie von Rollen abzubilden.

Gebundene abstrakte Rollenklassen werden – in Abwesenheit einer an die gleiche Basis gebundenen Subrollenklasse – von ObjectTeams als „relevant“ eingestuft, was ihre konkrete Implementierung erzwingt. Wird aber eine Rollenklasse nur als Supertyp für eine (noch) unbekannte Zahl von konkreten Rollen benötigt, so bleibt nur der Ausweg über eine ungebundene, abstrakte Rollenklasse. Würde eine (nicht abstrakte) Rollenklasse hierfür gewählt, so würden somit gegebenenfalls auch (ungewollte) Rolleninstanzen durch *Lifting* erzeugt. Ein *Guard Predicate* kann dies nicht verhindern, da es sich auf die intentional konkreten Rollen vererben würde. Dieses Problem hat eine gewisse Verwandtschaft zu dem in 7.2.7 beschriebenen “Callin without `super`” Problem, basiert aber auf der statischen Struktur und nicht dem Programmfluss.

Die Notwendigkeit, auf eine gebundene Rollenklasse in solchen Situationen zu verzichten schließt die Angabe von `base` *Guard Predicates* ebenso aus wie die Angabe von Callin-Methoden und Bindungen. Eine ungebundene Rollenklasse ist nicht wesentlich mächtiger als eine abstrakte Java-Klasse.
Ich schlage daher vor, dass ObjectTeams um eine Möglichkeit erweitert wird, die *Absicht* einer Bindung anzugeben.
In solcherart markierten Rollenklassen sollte es möglich sein `(base) when` Bedingungen und callin-Bindungen anzugeben, ohne dass eine solche Rolle jemals als relevant markiert oder instantiiert wird.

7.2.5 Unterstützung paralleler Programme

Sprachen mit dynamischen Aspekten erfordern neue Konzepte zur Synchronisation nebenläufiger Anwendungen. Dies trifft insbesondere bei unabhängig einsetzbaren Aspekten zu, bei denen im Vorhinein kaum eine Aussage über die Parallelität der Basisanwendung gemacht werden kann, nicht einmal gemacht werden sollte. Dabei ist insbesondere zu bedenken, dass eine spätere Anpassung der Aspekte, ob durch Connector-Module oder Erweiterung, auch eventuell nötige Synchronisationsanpassungen bieten können muss.

Ferner sollte bei ObjectTeams/Java die Semantik von `within` überdacht oder um einen threadgebundenen Ausdruck erweitert werden. Es gibt keine einfache Möglichkeit die Auswirkungen von `within` in einer parallelen Anwendung wirklich auf den eingeschlossenen Code zu beschränken. Die Teamaktivierung hat immer – möglicherweise vom Anwendungsentwickler ungewollte – Auswirkungen auf das gesamte Programm und alle Threads. Selbst eine Synchronisierung kann ungewollte Auswirkungen hinsichtlich der Teamaktivierung nicht gänzlich ausschließen.
Daher sollte ein Sprachfeature eingeführt werden – z.B. in Form eines Markerinterfaces oder einer Erweiterung der API um ein *Konstrukt* ähnlich `ThreadLocal` – , um die Aktivierung Teams auf einzelne Threads zu beschränken. Idealerweise aber ein `within`, welches wirklich nur den eingeschlossenen Code beeinflusst. Nicht zuletzt wäre somit auch ein `cflow` leicht zu ersetzen.

Darüber hinaus (vgl. 6.4.10) ergaben sich auch bei dem Versuch des Einsatzes in einer verteilten Umgebung einige Schwierigkeiten. Die bessere Unterstützung von RMI würde einige interessante Anwendungsfälle eröffnen.

7.2.6 Reflektionen auf Objektbasis

ObjectTeams erlaubt den Aufruf von Basiscode nur in eng definierten Schleusenbereichen und verbirgt das Basisobjekt weitestgehend außerhalb dieser. Darin liegt eine ausgesprochene Stärke, die sehr zur Eleganz dieser Sprache beiträgt.
Allerdings existieren Szenarien, in denen die Grenze zur Basis meines Erachtens aufgebrochen werden sollte. Ein solches ist zum Beispiel gegeben, wenn ein Aspekt die asynchrone Ausführung der Basismethode eines `replace`-*Advices* bewirken soll, analog

die Ausführung in einem anderen Sicherheitskontext (Vergleiche Abschnitt 6.4.4). In solchen Fällen erfordert Java die Einführung eines *Strategy* Objektes (z.B. eine `Runnable`-Implementierung für einen neuen Thread.), üblicherweise in einer anonymen Klasse[7].

```
1  team class AsynchronousTeam  {

3   public class  AsynchronousRole playedBy BaseClass {

5    callin void continueAsync(){
      Runnable runner = new Runnable(
7           public void run(){
             base.continueAsync() //nicht moeglich
9           }
      };
11    Thread asyncExec= new Thread(runner);
      asyncExec.start();
13   }

15  ...

17 }
```

Listing 7.4: Asynchrone Fortsetzung der Ausführung: In ObjectTeams/Java nicht möglich.

Unabhängige Aspekte erfordern die Bereitstellung eines Pseudo-Objektes, um z.B die Ausführung des Advice-Codes in einer anonymen inneren Klasse fortzusetzen; `base` kann und soll dieses nicht leisten.

7.2.7 Sichtbarkeit der Methodenbindung

Die Callin-Bindung hat eine Besonderheit, die sie von allen objektorientierten Mechanismen unterscheidet: Die Bindung gilt auch für alle Unterklassen der ursprünglichen Basisklasse. Selbst wenn eine redefinierte Methode keinen `super`-Aufruf enthält, bleibt die `callin`-Bindung bestehen. Anders formuliert: Es gibt keine Möglichkeit, für Subklassen innerhalb einer Bindungshierarchie weniger Methoden zu binden, ohne dafür neue Rollenklassen zu implementieren.
Christine Hering bezeichnete in [Her04a] dieses Verhalten als *"callin without super call"-Problem.*
Während JAsCos *Pointcut* Modell zwar nicht uneingeschränkt als eindeutig aufgefasst werden kann, so ist doch zu bemerken, dass dieses Problem dort durch die – teilweise aus AspectJ übernommenen – '+' und '++' Operatoren gelöst wird. In AspectJ schlicht als *Subtype Pattern* bezeichnet, bewirkt dieser Ausdruck erheblich mehr: Er verhält sich als aspektorientierte Variante der Sichtbarkeit in der Objektorientierung, da er die (basisseitige) Vererbung von Methodenbindungen steuert.
Ich schlage daher die Einführung eines Operators für Bindungen in ObjectTeams vor, der

[7] Wobei die Art der neuen Klasse keine Auswirkung auf die Forderung hat.

optional das Verhalten der Bindungen bezüglich redefinierter Methoden der Basisklasse an das *default*-Verhalten von JAsCo anpasst.

7.2.8 Parameter Tunneling bei replace-Callins

Parameter Mappings erlauben eines der nützlichsten ObjectTeams Idiome: Die Angabe von Konstanten in Methodenbindungen ermöglicht eine feine bindungsabhängige Konfiguration von *Callin*-Methoden. Eine typische Anwendung ist die Angabe von Logging Informationen, Testwerten oder – aus dem Kontext dieser Arbeit – Zugangseinschränkungen.
ObjectTeams bietet durch *Parameter Tunneling* bei `before` und `after` Bindungen hervorragende Unterstützung für dieses Idiom an, da die ursprünglichen Parameter unabhängig von denen des *Mappings* sein können, nicht aber so im Falle von `replace` *Callins*.
Die Diskrepanz liegt im unterschiedlichen Wesen der Advices begründet: `before`- und `after`-*Advices* enthalten keinen Aufruf einer Basismethode[8], bei `replace callin`s ist ein solcher aber essentiell.

Dies wäre für sich genommen ebenfalls kein Problem, wäre nicht in ObjectTeams eine Beschränkung gegeben, welche die Durchgabe aller ursprünglichen Parameter erforderlich macht(vgl. 6.3.1.1), wenn ein `base` Aufruf stattfinden soll.

JAsCo erlaubt beispielsweise den parameterfreien `proceed` Aufruf, der stets die ursprünglichen Parameter zur Fortsetzung verwendet.

Die Verwendung von `replace` *Callins* mit zusätzlichen Parametern erfordert daher immer `callin` Methoden mit

(|ergänzte Parameter| + |ursprüngliche Parameter|) Parametern

. Wünschenswert wäre eine automatische Durchleitung der ursprünglichen Parameter, wie es in JAsCo möglich ist.

```
1 ...
  void saySomething(Object myAddedParam, Object original1)
3       <- replace void foo(Integer param)
  with
5 {
    myAddedParam <-"test",
7       original1 <-param

9  }
   ...
```

Listing 7.5: Die Grenzen von Parametermappings I. Dieses funktioniert...

```
...
2  void saySomething(Object myAddedParam)
```

[8] Ein solcher wäre bei `after` advices ohnehin offensichtlich wirkungslos.

```
      <- replace void foo(Integer param)
with
{
  myAddedParam <- param
}
  ...
```

Listing 7.6: Die Grenzen von Parametermappings II. ...dieses auch...

```
...
void saySomething(Object myAddedParam)
      <- replace void foo(Integer param)
with
{
  myAddedParam <-"test"
}
  ...
```

Listing 7.7: Die Grenzen von Parametermappings III. ...aber dieses funktioniert nicht.

7.2.9 Delegation von `callin` Methoden

Ein zusätzliches Problem mit `replace` *Callins* ist die Unmöglichkeit, als `callin` markierte Methoden aufzurufen. Zumindest aus anderen Callin-Methoden heraus kann dies aber durchaus erforderlich sein.

Sind Callin-Methoden mit unterschiedlichen Parameterzahlen erforderlich, so müssen in der gegenwärtigen Konstellation *Callin*-Methoden mit allen denkbaren Parametern definiert werden – selbst wenn der Funktionsrumpf für jede dieser Methoden praktisch identisch ist. Ein *Refactoring* zur Extraktion dieser Rümpfe in eigene – nicht als `callin` markierte – Methoden ist oft möglich und naheliegend, jedoch umständlich.

Die Unmöglichkeit der Delegation trifft auch auf geerbte `callin` Methoden zu. Diese können zwar redefiniert werden, wobei dies bei impliziter und expliziter Vererbung gleichermaßen gut funktioniert. Aber es kann innerhalb von redefinierten Callinmethoden nur auf die `(t)super`-Implementierung mit genau der passenden Signatur zugegriffen werden.

In der Fallstudie ergab sich die Situation, dass für die Anbindung des Autorisationsaspektes, die Ergänzung von Parametern für callin-Methoden nötig wurde[9]. Da diese Funktionalität eine `replace`-Bindung zwingend erfordert, war es somit unmöglich, den gleichbleibenden Teil bereits in einer abstrakten Rolle zu formulieren. Dies erhöhte den Aufwand für die konkreten Verbindungen erheblich, zumal eine vielfache Implementierung sehr ähnlicher Methoden kaum als erstrebenswert angesehen werden kann.
Daher ist zumindest der Wunsch nach einer Möglichkeit, auch `callin` Methoden delegieren zu können, also andere `callin` Methoden anstelle der Basismethode aufzurufen, als ein legitimes Anliegen zu bezeichnen.

[9]Die Notwendigkeit für diese Ergänzung wurde in Abschnitt 7.2.8 behandelt

7.2.10 Guard-basierende „Switch“ Ausdrücke

Eine weitere wünschenswerte Erweiterung der Methodenbindungen wäre die Möglichkeit, mehrere Bindungen der gleichen Basismethode mit Guard Predicates zu versehen. In der Kombination mit der zuvor genannten Unmöglichkeit der Delegation ist sogar von „dringend anzuratend“ zu sprechen.
In der gegenwärtigen Situation erlaubt ObjectTeams lediglich ein `base Guard Predicate`, was auch durch Vererbung nicht umgangen werden kann[10]. Dies ist unzureichend, da eine Delegation von `replace` *Callins* nicht möglich ist. Soll die Auswahl der richtigen Callin-Methode und deren Parameter von dynamischen Bedingungen der Basis abhängen, so bleibt nur der Ausweg über normale *Guard Predicates*, aufbauend auf *Callout*-Bindungen[11] (siehe: Abschnitt 6.4.3).

```
...
bind1: void checkPermission( String permissionRequired )
          <- replace void foo()
      base when (base.value >= 10000)
      with
      {
          permissionRequired <- "admin"
      }
bind2: void checkPermission( String permissionRequired )
          <- replace void foo()
      base when (base.value < 10000)
      with
      {
         permissionRequired <- "normal_user"
      }
  ...
```

Listing 7.8: ObjectTeams erlaubt eine Konstellation wie die obige nicht. Die Auswahl und Behandlung von Methodenparametern bei `callins` hängt aber oft von dynamischen Bedingungen ab, so dass mitunter mehr als nur eine guardbewehrte Methodenbindung für eine gegebene Basismethode innerhalb einer Rollenklasse erforderlich ist. Insbesondere wird so die Nutzung von Parametermappings für einen deklarativen Connector-stil praktisch unmöglich, da zusätzliche Methoden und *Callouts* erforderlich werden.

7.3 Allgemein

7.3.1 Objektkommunikation zugänglich machen

Have more Smalltalk, könnte die Forderung heißen. Diese Forderung ist unter Performanzgesichtspunkten mitunter unrealistisch, aber das Wesen einer Forderung erlaubt meines

[10] Hier ist von einer Beschränkung der Implementation auszugehen.
[11] Natürlich sind auch `if`-Ausdrücke in den callin-Methoden möglich, wenn auch unschön.

Erachtens ein gewisses Ausblenden solcher Aspekte[12]. Eine Nachricht hat in der Regel nicht nur einen Empfänger, sondern auch einen Absender. Die Bedingungen für die Ausführung von Aspektcode, sei es `isApplicable` oder `when`, sollten auch den Aufrufer in Bezug nehmen können. Dabei ist `cflow` ein Anfang. Diese Pointcuts können aber nicht auf den Zustand des Aufrufers zugreifen und sind daher meines Erachtens unzureichend. Die Bereitstellung von `call` *Pointcuts* könnte dies mitunter abmildern. Ich fordere aber den *gleichzeitigen Zugriff auf Aufrufer und Aufgerufenen.* Dies ist z.B. bei der Implementierung von Zugriffsbeschränkungen praktisch wünschenswert; vielmehr aber ist es eine grundlegende Eigenschaft von *Crosscutting Concerns*, eben nicht im Kontext von nur einer Instanz einer Klasse zu liegen.

7.3.2 Webepunkte zugänglich machen

Obgleich sich Advicemethoden aus Aspekten in beiden Sprachen ihrerseits als Joinpoints auswählen lassen[13], gibt es dabei einen erheblichen Mangel: An diese Methoden gewobene Aspekte haben keinen Bezug zu dem ursprünglichen Joinpoint in der Basisanwendung, insbesondere kann das Ergebnis des Basisaufrufes ebensowenig modifiziert werden wie der Programmfluss. Die mögliche Lösung lediglich `around`/`replace` advice zu verwenden, ist dabei keinesfalls hinreichend[14], da Advice dann lediglich Advice sein kann: Ein Doppelspiel als normale Methode wäre ebenso unmöglich, wie eine wohldefinierte Schnittstelle. In der Studie ergab sich mehrfach die Situation von Aspekten, die stets an genau die gleichen *Joinpoints* zu binden waren. Diese Auswahl der gleichen *Joinpoints* ist aber statisch nicht überprüfbar. Daher wären Ausdrücke der Form :

> Wenn Aspekt a gewoben, so webe auch Aspekt b

wünschenswert[15]. Benannte unabhängige *Pointcuts*, wie etwa in *AspectJ* zu finden, würden dieses Problem lösen.
Ein weiteres Problem aus diesem Kontext findet sich beim Entwurf kooperativer Aspekte. Aufgrund der *oblivious* Basis können sich Probleme ergeben, wenn an einen Joinpoint gewobene Aspekte aufeinander aufbauen. Scheitert die Ausführung eines solchen Aspektes, werden Maßnahmen nötig, um das Programm in einem konsistenten Zustand zu halten. Dies impliziert einen gewissen Transaktionscharakter für gemeinsame Bindungen kooperativer Aspekte, der spätestens auf der Entwurfsebene gesondert berücksichtigt werden muss.

[12] Es fällt auf: Performanz als Beispiel eines realitätsbezogenen Einwandes ist nach wie vor ein sehr schlecht modularisierter Concern.
[13] Zumindest grundsätzlich
[14] Auch Performanz wäre hier zu nennen.
[15] Solches Verhalten lässt sich durch*JAsCo Combination Strategies* in gewissen Grenzen simulieren: Die Entfernung eines Aspektes aufgrund von Existenz/Nichtexistenz eines anderen an einem gegebenen *Joinpoint* ist möglich, nicht aber seine Ergänzung.

7.3.3 Unterstützung der Java Kern-API

Beiden Ansätzenist gemein, dass die Klassen der Java-API nicht als gültige Basisklassen in Frage kommen; *Joinpoints* in ihnen werden nie freigelegt[16].
In JAsCo kann diese Einschränkung zwar nicht wirklich ignoriert werden. Durch die Möglichkeiten der *Pointcuts* ist aber eine weitgehende Umgehung durch veroderte Ausdrücke möglich.
In ObjectTeams wiegen die Auswirkungen ungleich schwerer, da ObjectTeams 0.8.17 die Klassen der Java-Bibliothek nicht als gültige Basisklassen für gebundene Rollen unterstützt. Aufgrund des Konzepts von ObjectTeams kann dort nur über den Typ quantifiziert werden. Somit wird, durch die Unmöglichkeit die Java-API zu verwenden, die Konstruktion abstrakter Kollaborationen – eine grundlegende Anforderung für eine Wiederverwendung – erschwert. Die momentan noch vorhandene Unmöglichkeit, *Interfaces* als Basistyp zu verwenden verschärft dies noch, weil eigentlich in einer Hierarchie stehende Klassen nicht immer auf eine Rollenklassenhierarchie abgebildet werden können.

[16] Wie bei sehr vielen anderen Ansätzen auch.

8 Abschließende Bemerkungen

Dieses Kapitel bildet den Abschluß der Arbeit, zunächst in Form einer kurzen Zusammenfassung. Darüber hinaus, werden weiterführende und verwandte Arbeiten vorgestellt. Schließlich folgt ein Ausblick auf mögliche zukünftige Entwicklungen.

8.1 Zusammenfassung

Kapitel 3 enthält weitreichende Überlegungen zu dem Umgang mit Aspektkomponenten und wiederverwendbaren Aspekten. Eine vorsichtige Erweiterung der FODA wurde dabei angerissen.
Kapitel 5 stellt in Betrachtung der untersuchten Sprachen grundlegende Erfahrungen über den Umgang mit ihnen vor. Beide Ansätze sind noch auf dem Weg zu einer etablierten Idiomatik, so dass auch naheliegende Betrachtungen eine erhebliche Auswirkung auf die praktische Nutzung haben können.
Die Fallstudie stellte in Kapitel 6 eine prinzipielle Eignung beider untersuchten Ansätze für die Implementierung wiederverwendbarer Aspekte fest.
Entwürfe und die anvisierten Funktionalitäten konnten in beiden Sprachen dabei weitgehend implementiert werden; es gelang jeweils die Erstellung funktionierender Implementationen. Auch war eine weitgehende a-posteriori Integration dieser Aspekte in verschiedene Basisprogramme möglich. Allerdings wurden im Rahmen dieser Implementierungen viele Grenzen, Fehler und Unzulänglichkeiten aufgedeckt.

Insgesamt gesehen wurden grundsätzliche Erfahrungen über den Umgang mit den Sprachen gewonnen und formuliert, einige davon dienen als Grundlage für die in Kapitel 7 gesammelten Felder, die eine weitere Betrachtung rechtfertigen.
Darüber hinaus wurden Erkenntnisse über die Implementierung und Wesen wiederverwendbarer Aspekte in JAsCo und ObjectTeams aber auch im Allgemeinen gewonnen. Die vorgestellte Idee der kooperierenden Aspekte sind eine mögliche Vorstufe von Aspektbibliotheken für ganze Domänen, auch wenn dieser Schritt hier noch nicht gegangen werden konnte.

8.2 Verwandte Arbeiten

Hanenberg et al. veröffentlichten in [HHUK04] eine Fallstudie über aspektorientierte Frameworkentwicklung. Darin verwendeten sie ein *Framework* zum Traversieren von Objektgraphen (vgl. *Adaptive Programming*) als Gegenstand für einen Vergleich von Objektorientierung und Aspektorientierung. Die untersuchten AOP Sprachen waren hier *AspectJ*

und *AspectS*.

Bart De Win führte in seiner Dissertation [Win04] eine vergleichende Fallstudie über Sicherheitsaspekte in *Weaving* und *Interception* basierenden *AOSD* Ansätzen durch. Als Vertreter für *Weaving* diente *AspectJ*, für *Interception* eine Eigenentwicklung.
Die Studie umfasste Authentikation und Autorisation in einem FTP Server sowie in einem *Personal Information Manager* (PIM) und kam zu einem bedingt positiven Ergebnis, insbesondere auch hinsichtlich der Wiederverwendbarkeit. Auch in dieser Studie wurden JAAS basierende Sicherheitsaspekte entworfen und eingesetzt. Ähnlich der vorliegenden Arbeit warf das Ergebnis Fragen bezüglich der technischen Weiterentwicklung auf; konzeptionell wurde ein positives, wenn auch konstruktives Ergebnis festgestellt. Hierbei wurde nicht der Anspruch verfolgt, mit (C)OTS[1], also *oblivious*, Basisanwendungen eine a-posteriori Integration zu erreichen.

Christine Hering untersuchte in ihrer Diplomarbeit [Her04a] die Möglichkeiten einer *ObjectTeams* Implementierung des objektorientierten Frameworks *JHotDraw*. Die auswertende Fallstudie verglich isolierte Teile der ursprünglichen, objektorientierten Implementierung mit verschiedenen mit *ObjectTeams* entworfenen Varianten. Die dabei gewonnen Erkentnisse haben die jüngere Entwicklung von *ObjectTeams* merklich beeinflusst und bieten so die Basis einzelner Teile der vorliegenden Arbeit.

Eine Reihe von Studien untersucht die Möglichkeiten, objektorientierte *Design Patterns* ([GHJV95]) wiederverwendbar mit AOP-Sprachen auszudrücken und neue aspektorientierte Muster zu identifizieren. Hannemann und Kiczales implementierten in [HK02] eine Reihe der *GoF* Patterns in AspectJ, eine weitere Betrachtung dieser Untersuchung findet sich bei Garcia et al. ([GSF+05]).
Hanenberg und Constanza diskutieren in [HC02] neue Vorgehensweisen für häufig anzutreffende Szenarien in AspectJ, wobei der *Pattern*-Status dieser Vorschläge zur Diskussion gestellt wird. [HU03] kann als Fortsetzung davon betrachtet werden.

8.3 Ausblick

Die Studie belegt eine grundsätzliche Eignung zur Implementierung wiederverwendbarer Aspekte bei beiden Techniken. Sie zeigt aber auch, dass keine der beiden untersuchten Programmiersprachen als vollständig ausgereift betrachtet werden kann.
Daher ist eine Neubetrachtung vieler Einzelheiten dieser Studie nach dem Erscheinen finaler Versionen angezeigt.
Ein anderer Punkt, der noch wesentliche weitere Arbeit rechtfertigt, ist der Entwurf und die Implementierung aspektorientierter Bibliotheken. Der angerissene Prozess zeigte erste Ansätze, bedarf aber noch einer erheblichen Verfeinerung. Auch passte er nicht gleichermaßen gut auf beide untersuchten Programmiermodelle.

[1] (**C**ommercial) **O**f **T**he **S**helf

Die Überlegungen zu wiederverwendbaren Aspekten waren in erster Linie geeignet, das schiere Ausmaß der möglichen Richtungen anzudeuten. Um das Ziel zuverlässiger, wiederverwendbarer Aspekte erreichen zu können, scheint es erforderlich zu sein, einen fundierten Entwurfsprozess für generische Aspekte zu entwickeln. Dafür stellt sich auch die Frage nach einem Domänenanalyseverfahren, welches für solch besonders gelagerte Domänen geeignet ist.

Der Ansatz kooperierender wiederverwendbarer Aspekte bietet dann die Perspektive, auch ganze Domänen abzudecken. Diese Entwicklung könnte in der Zukunft zu Aspektbibliotheken führen, die dem Vergleich mit objektorientierten Frameworks standhalten. Die vorliegende Arbeit konnte und sollte diesem Anspruch nicht gerecht werden. Hier bietet sich ein höchst lebendiges Gebiet für weitere Forschung.

8.4 Fazit

Beide untersuchten Ansätze bieten Grundlagen für wiederverwendbare Aspekte. Allerdings waren zum Zeitpunkt dieser Untersuchung beide Sprachen noch nicht hinreichend entwickelt. Ein abschließendes Urteil wird erst in zukünftigen Arbeiten möglich sein.
Es gelang aber schon jetzt die Möglichkeiten wiederverwendbarer Aspekte zu beleuchten, und Methoden hinsichtlich Entwurf und Implementierung solcher Aspekte vorzustellen. Dabei konnten die Möglichkeiten der beiden untersuchten Sprachen herausgearbeitet und in Perspektive zueinander gebracht werden.
Das Konzept von ObjectTeams kann schon jetzt als gute Voraussetzung für die Implementierung und Anbindung – insbesondere rollenbasierender – Frameworks gesehen werden. Weitere Forschung bezüglich der Formulierung abstrakter Teams und Rollen scheint dennoch gerechtfertigt, um Aspektverhalten besser wiederverwendbar implementieren zu können. Für die Implementierung Framework-ähnlicher Ansätze, insbesondere in Richtung von *Black Box Frameworks*, fehlt noch die Ausdrucksmächtigkeit, um einige Funktionen mit der gebotenen Abstraktion von Basisanwendungen zu implementieren. Es wurden Vorschläge formuliert, die möglicherweise hier helfen könnten. Unabhängig von diesen ist aber die gute Wiederverwendbarkeit von geeignet entworfenen *White-Box* Implementierungen hervorzuheben.

JAsCo bietet eine bemerkenswerte Eignung zur Implementierung von Black-Box Aspekten, ohne diese in ihrer Anpassbarkeit zu beschränken. Gelegentlich erscheint dabei die Formalisierung der Grenze zwischen Aspekt und Verbindung als zu schwach. Zusätzlich führen technische Einschränken zu einer Reduktion der Ausdrucksmächtigkeit, die voraussichtlich in zukünftigen Versionen deutlich besser ausfallen wird. Auch die Unterstützung von objektorientierten Konzepten – etwa Vererbung – ist noch nicht ausgereift.
Dessen ungeachtet verspricht auch diese Sprache viele anwendbare Ansätze für wiederverwendbare Aspekte. Eine konsistentere Dokumentation würde helfen, solche Ansätze schneller zu identifizieren.

Dank möchte ich an dieser Stelle den hinter JAsCo und ObjectTeams stehenden Perso-

nen aussprechen, die immer zeitnah auf gefundene Probleme und „*Bugs*“ reagiert haben. Dank auch an die Teilnehmer der Diskussionen auf den Mailinglisten der Ansätze.

Literaturverzeichnis

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *Transactions on Aspect-Oriented Software Development*, 2006.

[BFTY04] O. Barzilay, Y. Feldman, S. Tyszberowicz, and A. Yehudai. Call and execution semantics in aspectj. In C. Clifton, R. Lammel, and G. T. Leavens, editors, *Languages Workshop at AOSD 2004*, 2004.

[BMdV00] Johan Brichau, Wolfgang De Meuter, and Kris de Volder. Jumping aspects. In *14th European Conference on Object-Oriented Programming*, 2000.

[Cot05] Michael Coté. Jaas in action. `http://www.jaasbook.com`, 10 2005.

[CSS04] Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Stoerzer. Aop considered harmful, 2004.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD04*, 2004.

[Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11:147–148, 1968.

[Dij74] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer Verlag New York, 1982 (1974).

[FF04] Robert E. Filman and Daniel P. Friedman. Aspect oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clark, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*, chapter 2, pages 21–35. Addison-Wesley, 2004.

[FVSB05] Bruno De Fraine, Wim Vanderperren, Davy Suvée, and Johan Brichau. Jumping aspects revisited. In *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*, pages 77–86, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented-Software*. Addison-Wesley Professional, 1995.

[GK01] Stephan Gudmundson and Gregor Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. In *ECOOP 2001*, 2001.

[GSF+05] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. In *AOSD2005*, 2005.

[HC02] Stefan Hanenberg and Pascal Costanza. Connecting aspects in aspectj: Strategies vs. patterns. In Yvonne Coady, editor, *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 40–45, 2002.

[Her02a] Stephan Herrmann. Composable designs with ufa. In *Workshop on Aspect-Oriented Modeling with UML*, 2002.

[Her02b] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *NOD*, 2002.

[Her04a] Christine Hering. Evaluierung der aspektorientierten sprache objectteams/java zur strukturverbesserung des frameworks jhotdraw. Diplomarbeit, Technische Universität Berlin, 2004.

[Her04b] Stephan Herrmann. Confinement and representation encapsulation in object teams. Technical report, Technical University Berlin, 2004.

[Her04c] Stephan Herrmann. Sustainable architectures by combining flexibility and strictness in object teams. In *IEE Proceedings - Software Engineering Special Issue on Unanticipated Software Evolution*, 2004.

[HH] Stephan Herrmann and Christine Hundt. *ObjectTeams/Java Language Definition (OTJLD) (Version 0.8).* Online Guide: `http://www.objectteams.org/def/0.8/index.html`.

[HHM04a] Stephan Herrmann, Christine Hundt, and Katharina Mehner. Mapping use case level aspects to objectteams/java. In *Workshop on Early Aspects – OOPSLA 2004.*, 2004.

[HHM04b] Stephan Herrmann, Christine Hundt, and Katharina Mehner. Translation polymorphism in object teams. Technical Report 2004/05, Technical University Berlin, 2004.

[HHMW05] Stephan Herrmann, Christine Hundt, Katharina Mehner, and Jan Wloka. Using guard predicates for generalized control of aspect instantiation and activation. In *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*, pages 93–101, 2005.

[HHUK04] Stefan Hanenberg, Robert Hirschfeld, Rainer Unland, and Katsuya Kawamura. Applying aspect-oriented composition to framework development - a case study. *FUSE 2004*, 2004.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA*, 2002.

[HU03] Stefan Hanenberg and Rainer Unland. Aspectj idioms for aspect-oriented software construction. In *EuroPLOP*, 2003.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Standards Board, leee std 610.12–1990 edition, 1990.

[Isb06] Wes Isberg. Check out library aspects with aspectj 5. *AOP@Work*, 14:N/A, 2006.

[KCH+90] Kyo Kang, Sholom Choen, James Hess, William Novak, and A. Spencer Peterson. Feature oriented domain analysis(foda) feasibility study. Technical report, Software Engineering Institute Carnegie Mellon University, 1990.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect - oriented programming. In *ECOOP*, 1997.

[KM05] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP2005*, 2005.

[KP88] Glenn E. Krasner and Stephen T. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):132–183, 1992.

[Lee02] Ken Wing Kuen Lee. An introduction to aspect-oriented programming. Reading Assignment, The Hong Kong University of Science and Technology, 2002.

[Les05] Nicholas Lesiecki. Enhance design patterns with aspectj, part 1+2. *AOP@Work*, N/A:N/A, 2005. `http://www-128.ibm.com/developerworks/java/library/j-aopwork5/` `http://www-128.ibm.com/developerworks/java/library/j-aopwork6/`.

[LLM99] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical report, College of Computer Science, Northeastern University, Boston, 1999.

[MCI68] M.D. MCILROY. Mass produced software components. In Peter Naur and Brian Randell, editors, *Software Engineering: Report on a conference sponsored by the Nato Science Comittee, Garmisch, Germany*, pages 79–87. NATO Science Committee, 1968.

[MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD03*, 2003.

[Nei80] James M. Neighbors. *Software Construction Using Components*. PhD thesis, Department of Information and Computer Science University of California, Irvine, 1980.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Paw03] Renaud Pawlak. The aop alliance: Why did we get in? Draft, 2003.

[PD90] Rubén Prieto-Díaz. Domain analysis: An introduction. *ACM SIGSoft Software Engineering Notes*, (15)2:47–54, 1990.

[PD93] Rubén Prieto-Díaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, May 1993.

[PDF+02] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. Jac: An aspect-based distributed dynamic framework. *N/A*, N/A:N/A, 2002.

[RG98] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, 1998.

[SFV05] Davy Suvée, Bruno De Fraine, and Wim Vanderperren. Fusej: An architectural description language for unifying aspects and components. In *SPLAT 2005*, 2005.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD-2003*, 2003.

[SVWJ04] Davy Suvée, Wim Vanderperren, Dennis Wagelaar, and Viviane Jonckers. There are no aspects. In *SC 2004*, 2004.

[TOJH04] Peri Tarr, Harold Ossher, Stanley M. Sutton Jr., and William Harrison. N degrees of seperation: Multi-dimensional separation of concerns. In Robert E. Filman, Tzilla Elrad, Siobhán Clark, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*, chapter 3, pages 37–61. Addison-Wesley, 2004. Original: Proceedings of ICSE99.

[Van02] Wim Vanderperren. A pattern based approach to separate tangled concerns in component based development. In Yvonne Coady, editor, *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 77–80, 2002.

[Van04] Wim Vanderperren. Aspect oriented programming in jasco. Course Slides, 04 2004.

[Van05] Wim Vanderperren. *JAsCo Language Reference 0.8*, 2005.

[Van06] Wim Vanderperren. Jasco quick reference. Online Guide: `http://ssel.vub.ac.be/jasco/documentation:quick`, 2006. Version vom 27.01.2006.

[VH03] Matthias Veit and Stephan Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In *2nd International Conference on Aspect-Oriented Software Development, March 2003, Boston.*, 2003.

[VS04a] Wim Vanderperren and Davy Suvée. Jascoap: Adaptive programming for component-based software engineering. In *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 77–78, 2004.

[VS04b] Wim Vanderperren and Davy Suvée. Optimizing jasco dynamic aop through hotswap and jutta. In *Dynamic Aspects Workshop*, 2004.

[VSCF05] Wim Vanderperren, Davy Suvée, Maria Agustina Cibrán, and Bruno De Fraine. Stateful aspects in jasco. In F. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Proceedings of SC 2005*, pages 167–181. Springer-Verlag Berlin, Heidelberg, 2005.

[VSV+05] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in jasco. In *AOSD2005*, 2005.

[VWD01] Bart Vanhaute, Bart De Win, and Bart De Decker. Building frameworks in aspectj. In *ECOOP 2001 Workshop on Advanced Separation of Concerns, Budapest, 2001*, 2001.

[Win04] Bart De Win. *Engineering Application-level Security through Aspect-Oriented Software Development*. PhD thesis, Katholieke Universiteit Leuven, 2004.

Listings

Abbildungsverzeichnis