U N I V E R S I T Ä T
KOBLENZ • LANDAU

Fachbereich Informatik

DIPLOMARBEIT

für die Prüfung zum Diplom-Informatiker
vorgelegt von:

Daniel Lohmann

Thema:

Multidimensionales Trennen der Belange im Softwareentwurf

Betreuer:	Prof. Dr. Jürgen Ebert, Universität Koblenz
Eingereicht am:	30. September 2002

Vorwort

Aspektorientierte Softwareentwicklung und multidimensionales Trennen der Belange ist ein hochaktuelles Thema in der Forschung und erfreut sich als solches stetig steigender Beliebtheit. In diesem Jahr fand in den Niederlanden die *1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*[1] statt. Die Konferenz war ausgesprochen gut besucht und wird zukünftig jährlich an jeweils wechselnden Orten ausgerichtet. Es zeigte sich, dass neben den Wissenschaftlern sich auch zunehmend Praktiker für Aspekttechniken interessieren und diese auch bereits mit beachtlichem Erfolg einsetzen.

Die aspektorientierte Softwareentwicklung hat das Potential in den nächsten Jahren zu großen Fortschritten in der Softwaretechnik zu führen – sowohl in der Theorie als auch in der Praxis. Über ein derartig aktuelles und zukunftsweisendes Thema eine Diplomarbeit zu schreiben war eine große Herausforderung. Und es hat wirklich Spaß gemacht.

In den letzten Jahren ist es üblich geworden, im Vorwort einige Worte über die immer wieder diskutierte Frage der männlichen/weiblichen Formulierungen zu verlieren. Ich habe sie für mich gelöst, indem ich die handlichere männliche Form verwende und bitte darum diese als neutrale Generalisierung zu verstehen.

Die Führungsebenen von Gesellschaft, Wirtschaft und gerade auch der IT-Branche sind dringend auf eine stärkere Durchdringung von Frauen angewiesen. Ich hoffe, dass dieses Ziel in nicht allzu ferner Zukunft erreicht wird und sich die wichtige Diskussion um Gleichberechtigung dann nicht mehr auf vermeintlichen Nebenkriegsschauplätzen wie der Sprache bewegt.

Bei der Erstellung dieser Arbeit haben mir einige Menschen sehr geholfen. Ohne sie hätte es diese Arbeit nicht gegeben:

Zunächst möchte ich mich ganz besonders bei meinem Betreuer, Herrn Prof. Dr. Jürgen Ebert, bedanken. Nun findet sich ein Satz wie dieser – verdient oder unverdient – wohl an erster Stelle in den Danksagungen einer jedern Diplomarbeit. Ich erlaube mir daher ihn hier noch ein wenig weiter auszuführen: Herr Ebert hat mir bei der Bearbeitung des für uns beide neuen Themas viele Freiheiten gewährt – ohne mich damit alleine zu lassen. Die vielen Stunden, die wir mit intensiven und fruchtbaren Diskussionen verbracht haben, werden mir in guter Erinnerung bleiben. Er hat mir außerdem die Fahrt zur AOSD 2002 und zu weiteren Workshops ermöglicht. Die Eindrücke, die ich davon mitnehmen konnte, waren mir eine große Hilfe.

Ebenfalls bedanken möchte ich mich bei meinem Mitbewohner Bernhard Gabler. Bernhard war immer ansprechbar, wenn ich mal jemanden brauchte, um den einen oder an-

[1] [Kic02]

deren Punkt zu diskutieren. Er hat mir dadurch sehr geholfen meine Gedanken und Ideen in Worte zu gießen. Ebenfalls unvergessen bleiben seine Hilfe beim Korrekturlesen und seine nachdrücklichen Bemühungen um eine präzise Sprache.

Schließlich möchte ich mich bei meinen Eltern bedanken, die mir dieses Studium ermöglicht haben. Sie haben mich immer meinen eigenen Weg gehen lassen und mir dabei alle nur denkbare Unterstützung gewährt.

Koblenz, im September 2002

Daniel Lohmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Das Prinzip der Trennung der Belange

Das *Prinzip der Trennung der Belange (Principle of Separation of Concerns)* ist das fundamentale Prinzip jeglicher Ingenieurstätigkeit. Es basiert auf der einfachen Erkenntnis, dass es uns als Menschen mental nicht möglich ist, beliebig viele Angelegenheiten gleichzeitig im Auge zu haben. Kognitionswissenschaftler sprechen hier gerne von der „magischen Zahl 7“ und meinen damit, dass der durchschnittliche Mensch nicht in der Lage ist mehr als sieben verschiedene Dinge gleichzeitig zu erfassen.

Wie immer man derartiges Zahlenwerk auch beurteilen mag, so steht doch unweigerlich fest, dass unser mentales Auffassungsvermögen leidlich begrenzt ist. Um nicht-triviale Probleme lösen zu können, sind wir folglich gezwungen, die damit verbundenen Angelegenheiten aufzuteilen, zu gliedern und einzeln zu betrachten. Wir befolgen das Prinzip der Trennung der Belange.

1.1. Belange in der Softwaretechnik

1.1.1. Das Prinzip der Trennung der Belange – eine treibende Kraft

In der Softwaretechnik wird das Prinzip der Trennung der Belange heute vor allem auf die Arbeiten von DAVID L. PARNAS [Par72] und EDSGAR W. DIJKSTRA [Dij76] zurückgeführt, die seine Verwendung im Rahmen der Softwareentwicklung geprägt haben. Sie legen nahe, die Elemente eines (Software-) Systems intentional und lokal (Lokalitätsprinzip) zu beschreiben. Eine konsequente Trennung der Belange führt zu besser verständlicher, wieder verwendbarer und wartbarer Software.

Das Identifizieren und Trennen der verschiedenen Belange eines Zielsystems erfolgt dabei in der Praxis mit Hilfe von *Dekompositionstechniken*, wie modularer und objektorientierter Dekomposition. Traditionell geht es dabei vor allem um eine hierarchische Zerlegung des Systems in Module, also in wohl definierte, funktional abgeschlossene und überschaubare Einheiten mit minimaler Kopplung und Redundanz.

Man kann wohl sagen, dass das Prinzip der Trennung der Belange eine treibende Kraft für die Entwicklung von Paradigmen, Methoden, Sprachen und Werkzeugen in der

Softwaretechnik ist. Neue Vorgehensweisen haben vor allem dann Chancen sich nachhaltig durchzusetzen, wenn sie einfachere und natürlichere (intentionalere) Abstraktionen anbieten, also gleichsam versprechen Dekomposition und Komposition (und damit die Anwendung des Prinzips der Trennung der Belange) zu vereinfachen. Dieses führte im Bereich der Programmiersprachen über strukturierte Programmierung und Modulkonzept bis zu den objektorientierten Sprachen. Für spezielle Domänen wurden eigene Programmiersprachen entwickelt wie z.B. die Abfragesprache SQL, die besondere Abstraktionen anbietet zur Bearbeitung genau der Belange, die beim Umgang mit relationalen Datenbanken auftauchen. Und auch auf der Analyse- und Entwurfsebene lässt sich eine derartige Entwicklung beobachten. Neue Methoden brachten weitere Techniken und Sprachen mit, von Datenflussmodellen über strukturierte Analyse bis zu objektorientierter Analyse und Design mit der Unified Modelling Language (UML).

1.1.2. Der (Miss-)Erfolg der Objektorientierung

Das objektorientierte Paradigma hat demnach seinen durchschlagenden Erfolg in den 90er Jahren sicherlich der Tatsache zu Verdanken, dass es die Trennung der Belange besser unterstützt als die vorangegangenen Ansätze. Tatsächlich scheinen gerade die von der OOP angebotenen Abstraktionen wie *Klasse*, *Objekt* und *Beziehung* besonders geeignet, um die im Rahmen der funktionalen Dekomposition ermittelten Konzepte auf eigenständige Software-Einheiten abzubilden.

Nichtsdestotrotz hat auch die Objektorientierung nur einen Bruchteil des Versprechens gehalten, komplexe Softwaresysteme und deren Bausteine so zu realisieren, dass sie *verständlich*, *wieder verwendbar*, *einfach erweiterbar* und *leicht zu warten* sind. Stattdessen tendieren die Bausteine realer Softwaresysteme dazu eine hohe Kopplung und Abhängigkeit aufzuweisen, in denen die Implementierung verschiedener Belange oftmals *vermischt* und *ineinander verwoben* ist. Die Realisierung einzelner Belange lässt sich dann nur schwerlich wieder verwenden und erweitern, Wartungsaufgaben sind aufwändig und risikoreich. Das Hinzufügen oder Entfernen von Belangen führt oftmals zu umfangreichen Modifikationen am bestehenden System. Offensichtlich reichen die vorhandenen Dekompositionstechniken der Objektorientierung nicht aus, um alle auftretenden Belange zu separieren und getrennt zu behandeln.

1.2. Über diese Arbeit

Diese Arbeit handelt von neueren Ansätzen zur Trennung der Belange in der Softwaretechnik. Mit neueren Ansätzen sind hier Paradigmen, Methoden und Werkzeuge gemeint, die in den letzten Jahren entstanden sind und versuchen die Limitationen der Objektorientierung zu überwinden. All diesen Ansätzen ist gemein, dass sie nicht versuchen die Objektorientierung zu ersetzen, sondern sie vielmehr *ergänzen* wollen.

In Kapitel 2, werden dazu zunächst ein paar Grundlagen vermittelt. Dabei wird der Belangbegriff erörtert, dargestellt wie Belange entstehen und warum es so schwierig ist,

das Prinzip der Trennung der Belange durchzuhalten. Des Weiteren wird ein kurzer Überblick auf bekannte Methoden zum Trennen der Belange gegeben.

Die *Aspektorientierte Programmierung* (Kapitel 3) ist der in der aktuellen Forschung wohl meistdiskutierte Ansatz für eine weitergehende Trennung der Belange und dementsprechend auch einen Schwerpunkt dieser Arbeit. Sie nimmt sich gezielt dem Problem der so genannten „quer schneidenden" Belange an. In Kapitel 3 wird zunächst an einem Beispiel gezeigt, was quer schneidende Belange sind und warum sie in der objektorientierten Programmierung so große Schwierigkeiten bereiten. Anschließend wird der Aspektbegriff eingeführt und es erfolgt eine Vorstellung der bekanntesten Ansätze zur aspektorientierten Programmierung.

Der *Hyperspace-Ansatz* (Kapitel 4) wird im Allgemeinen ebenfalls zur aspektorientierten Programmierung gezählt, geht jedoch konzeptuell darüber hinaus. Während die aspektorientierte Programmierung das Problem der quer schneidenden Belange überwiegend auf der Implementierungsebene sieht und behandelt, favorisiert der Hyperspace-Ansatz ein „multidimensionales Trennen der Belange" über alle Phasen der Softwareentwicklung. Der Hyperspace-Ansatz ist der Schwerpunkt dieser Arbeit, er wird in Kapitel 4 ausführlich vorgestellt. Neben dem theoretischen Modell wird dabei auch eine Einführung in Hyper/J gegeben, ein Werkzeug für multidimensionales Trennen der Belange mit Java.

In Kapitel 5, *Erweiterung des Hyperspace-Modells*, wird der Hyperspace-Ansatz auf seine Eignung speziell für die frühen Phasen der Softwareentwicklung untersucht. Dabei wird sich zeigen, dass das bestehende Modell ungeeignet ist für eine Verwendung mit den in Analyse und Entwurf gebräuchlichen Artefaktsprachen. Anhand dieser Erkenntnisse wird das erweiterte Modell entwickelt und mit einem realen Beispiel überprüft. Das erweiterte Modell ist das Ergebnis dieser Arbeit. Es wird zum Abschluss ausführlich diskutiert und weitere Verwendungsmöglichkeiten werden aufgezeigt.

Kapitel 6 fasst die Ergebnisse dieser Arbeit noch einmal zusammen und gibt einen Ausblick auf zukünftige Forschungs- und Entwicklungsaktivitäten, die sich aus dem erweiterten Modell ableiten lassen.

Kapitel 2

Belange und wie sie getrennt werden

In der Softwaretechnik handelt das Prinzip der Trennung der Belange von „Belangen eines Softwaresystems“ und wie man diese durch „Trennen“ beherrschbar macht. Doch was sind hier eigentlich „Belange“? Und wie werden sie „getrennt“? Warum bereitet das Trennen Probleme?

Um diese Fragen zu beantworten, soll im Folgenden zunächst der Belangbegriff mit seinen Ausprägungen im Kontext der Softwareentwicklung näher beleuchtet werden. Außerdem wird in diesem Kapitel noch ein kurzer Überblick zu klassischen und moderneren Methoden für eine Trennung der Belange gegeben.

2.1. Was einen Belang ausmacht

2.1.1. Belange sind Ausdruck einer Intention

Laut RICH HILARD [Hil99] drückt ein Belang ein „spezifisches Interesse in einer Sache, bezogen auf ein System oder eine Angelegenheit“ aus: *„A concern expresses a specific interest in some topic pertaining to a particular system of interest (or other subject matter).“* Ein Belang ist also Ausdruck einer bestimmten Intention. Belange kommen nicht aus dem Abstrakten, sondern stehen in Relation zu *Interessen* von beteiligten Menschen (Anwender, Benutzer, Entwickler, Verwalter, …). Sie erscheinen demnach naturgemäß aus *verschiedensten Quellen* und zu *jedem Zeitpunkt* des Softwarelebenszyklus, angefangen mit Konzeption über Anforderungsanalyse, Entwurf, Implementierung, Wartung und Weiterentwicklung [Hil99]. Bezogen auf Beteiligte und Phasen des Softwarelebenszyklus werden Belange in unterschiedlicher *Art* oder *Gestalt* dargestellt.

So ist z.B. die vorherrschende Gestalt der Belange im objektorientierten Entwurf die *Klasse*. Bei der Anforderungsanalyse werden Belange hingegen häufig in Gestalt von *Funktionalitäten* oder *Features* (wie z.B. *Drucken*, *Sicherheit* oder *Persistenz*) ausgedrückt. Auf der Implementierungsebene finden wir sie schließlich in Gestalt von *Quelltext*.

2.1.2. Dimensionen von Belangen

Die unterschiedlichen Gestalten von Belangen bezeichnen HAROLD OSSHER und PERI TARR als *Dimensionen* der Belange [OT00]. Dabei ist festzuhalten, dass typischerweise sowohl ein Belang in mehreren Dimensionen liegt als auch eine Dimension mehrere verschiedene Belange betrifft und sowohl Belange als auch Dimensionen dabei überlappen können.[2] Vereinfacht kann man sich eine Dimension als eine *Sicht* auf ein Problem, entstanden durch eine *Dekompositionstechnik* vorstellen. Verschiedenen Sichten entsprechen dabei häufig bestimmten Artefaktsprachen, wie z.B. Klassendiagramme (statische Sicht) und Kollaborationsdiagramme (dynamische Sicht) in der UML.

Betrachtet man ein Belang anhand der einzelnen Dimensionen, so ergeben sich jeweils unterschiedliche Ansichten aus denen sich Aktivitäten, Ziele und Gestaltungsgrundsätze ableiten lassen. Die meistens bekannten Dekompositionstechniken (als Vorgehensweisen zum Trennen der Belange) favorisieren jedoch eine „dominante" Dimension, anhand der die Zergliederung zu erfolgen hat. So *muss* beim objektorientierten Entwurf eine Zergliederung in Klassen erfolgen, bei der funktionalen Programmierung *muss* das Problem in Form von Funktionen separiert werden, beim Domain-Engineering *muss* in Features zergliedert werden. Wie OSSHER und TARR weiter ausführen [OT00, TOHS99] führt das in jedem nichttrivialen Beispiel dazu, dass es Belange gibt, die sich über mehrere der damit separierten Einheiten verteilen und den Entwickler dann zwingen das Prinzip der Trennung der Belange zu verletzen. Und das obwohl sich genau dieselben Belange oft in einer anderen Dimension (und damit durch eine andere Dekompositionstechnik) sauber voneinander trennen ließen. Sie bezeichnen dieses Phänomen folgerichtig als die *„Tyrannei der dominanten Dekomposition"* und sehen es als die Hauptursache dafür, dass eine klare Trennung der Belange in der Praxis kaum erreicht wird.

2.1.3. Belange sind abhängig vom Kontext

Die Menge der betrachteten Belange und Dimensionen variiert dramatisch über den Betrachtungspunkt, den Betrachter und seinen Intentionen. Anders ausgedrückt: Die Relevanz von Belangen und Dimensionen ist *abhängig vom Kontext*. Anwender eines Systems sprechen nun mal nicht von Klassen und Methoden, sondern von Funktionalitäten, Features und Eigenschaften. Architekten betrachten ein System nicht auf der Ebene von Quellcode, Systemintegratoren interessieren sich hauptsächlich für Pakete und deren Abhängigkeiten, usw.

Folgt man diesem Gedanken weiter, so kann es in der Softwareentwicklung gar nicht „die" Dimension geben, anhand der eine saubere Trennung der Belange möglich ist. *Jede* Art der Dekomposition und Komposition ist für einen bestimmten Kontext geeignet, jede unterstützt bestimmte Aktivitäten und hebt bestimmte Eigenschaften hervor – während sie andere unterdrückt [OT00]. Was wir also brauchen sind Methoden und Werkzeuge,

---

[2] Der Begriff der „Dimension" ist also nicht im mathematischen Sinne zu sehen, wo verschiedene Dimensionen üblicherweise als orthogonal (linear unabhängig) angenommen werden. Ebenso sind Belange in der Praxis selten unabhängig voneinander.

die ein *simultanes multidimensionales Trennen der Belange* anhand *mehrerer Dimensionen* ermöglichen.

2.1.4. Multidimensionales Trennen der Belange

OSSHER und TARR definieren den Begriff *multidimensionales Trennen der Belange* als Trennung der Belange [OT00]

- mit beliebig vielen Dimensionen von Belangen,
- *simultanem* Trennen der Belange entlang dieser Dimensionen,
- der Möglichkeit, neue Belange und Dimensionen *dynamisch* zu handhaben, so wie sie im Softwareentwicklungsprozess erscheinen, und
- überlappenden und interagierenden Belangen. (Es ist verlockend sich Belange als unabhängig oder „orthogonal“ vorzustellen, aber derartige Belange kommen in der Praxis nur selten vor.)

In der Softwaretechnik wurde in den letzten Jahren eine Reihe von Methoden und Werkzeugen entwickelt, die eine Dekomposition anhand mehrerer Dimensionen unterstützen – wenn auch nicht anhand beliebig vieler Dimensionen, wie oben gefordert. Ich möchte diese Ansätze daher als im Folgenden als Methoden für *mehrdimensionales Trennen der Belange* bezeichnen.

2.2. Mehrdimensionales Trennen der Belange

2.2.1. Softwareentwicklung erfolgt mehrdimensional

Schon seit den frühen Tagen der Softwareentwicklung boten neuere Sprachen jeweils zusätzliche Konstrukte für Dekomposition: Makros, Unterprogramme, Module, generische Einheiten, Klassen – um nur einige der bekanntesten zu nennen. Diese Konzepte werden heute so selbstverständlich verwendet, dass man sich ihres Vorhandenseins kaum noch bewusst ist. Neuere Dekompositionskonstrukte bauen dabei auf den älteren auf, erweitern also Sprachen um neue Dimensionen. Das Modulkonzept wäre ohne Unterprogramme wohl genauso wenig denkbar gewesen wie das Klassenkonzept ohne die Idee der Modularisierung.

Auch auf den höheren Ebenen der Softwareentwicklung wurde und wird ganz selbstverständlich mehrdimensional gearbeitet. So entsprechen die verschiedenen Diagrammtypen der UML verschiedenen Dimensionen, anhand derer das Softwaresystem beschrieben wird. Die zentrale Rolle (und damit die dominante Dimension) stellt in der UML dabei das Klassendiagramm dar. Dessen statische Sicht wird ergänzt durch Diagramme für das dynamische Verhalten des Systems wie Interaktionsdiagramme oder Statecharts.

Die oben angesprochenen Erwartungen und Enttäuschungen bzgl. der Objektorientierung sind jedoch ein deutliches Beispiel dafür, dass die vorhandenen Dimensionen nicht ausreichend sind um eine klare Trennung der Belange zu erzielen.

2.2.2. Neuere Ansätze für mehrdimensionales Trennen der Belange

Dementsprechend bilden neuere Ansätze, die anstreben die Grenzen der Objektorientierung zu überwinden und in [EFB01] als *post-object programming (POP) mechanims* bezeichnet werden, ein aktives Feld in Forschung und Entwicklung. Bei der überwiegenden Anzahl der POP Ansätze geht es vor allem darum, die Ausdrucksstärke der Sprachen in den späten Phasen der Softwareentwicklung zu erweitern (Entwurf und Implementierung), um zusätzliche Dimensionen zur Dekomposition bereit zu stellen. Bekannte Beispiele sind *Generisches Programmieren*, *Metaprogrammierung, Reflexion* und *Entwurfsmuster*. Andere Ansätze wie *Viewpoints* richten ihren Fokus stärker auf die frühen Phasen des Softwarelebenszyklus. Neuere Felder an denen aktiv geforscht wird sind *Generatives Programmieren* und *Aspektorientiertes Programmieren*. Der *Hyperspace Ansatz* schließlich versucht ein Modell zur phasenübergreifenden Integration beliebig vieler Dimensionen und Artefakte zu realisieren, also echtes multidimensionales Trennen der Belange zu ermöglichen.

2.2.2.1. Generisches Programmieren

Sprachen wie C++ oder Ada verfügen über das Konzept der *generischen Einheiten* oder *Templates*. Generische Einheiten stellen in typisierten Sprachen eine zusätzliche Dimension zur Dekomposition dar, indem sie es ermöglichen Programmcode typunabhängig zu beschreiben und zu verwenden. Insbesondere in Sprachen wie C++, die außerdem das Überladen von Funktionen und Operatoren erlauben, lassen sich damit viele typische Programmiertätigkeiten separieren und wieder verwenden. Ein schönes Beispiel für diese unter dem Begriff *Generisches Programmieren* bekannt gewordene Technik, ist die C++ Standardbibliothek (standard template library, STL), die es erlaubt eine große Anzahl von Algorithmen, Datenstrukturen, Zugriffsmethoden und Datentypen beliebig miteinander zu kombinieren. Durch die strikte Trennung dieser Konzepte bietet die STL eine enorm hohe Flexibilität und eine sehr gute Erweiterbarkeit bei vergleichsweise kleiner Quellcode-Größe. Ein weiteres Beispiel für die Mächtigkeit dieses Ansatzes ist die an die STL angelehnte Boost Graph Library (früher Generic Graph Component Library) [Boost, SLL99], eine generische Bibliothek zur Bearbeitung von graphartigen Datenstrukturen, die Flexibilität mit sehr guter Ausführungsgeschwindigkeit verbindet.

2.2.2.2. Statische Metaprogrammierung

In C++ eignet sich das Template-Konzept außerdem zur Metaprogrammierung. Die C++ Template-Sprache stellt eine (Turing-mächtige) funktionale Sprache dar, deren Programm zur Übersetzungszeit vom Compiler abgearbeitet wird und umfangreiche Möglichkeiten zur Beeinflussung des resultierenden Codes bietet. So gibt es entsprechende Idiome um Bindungszeiten (inline, statisch, dynamisch) zu parametrisieren, Methoden in Klassen einzumischen (Mixins) und vieles mehr. In der Praxis wird C++ Metaprogrammierung vor allem für generative Techniken verwendet (siehe unten). Bekanntes Beispiel dafür ist die Numerikbibliothek Blitz++ [Vel96], die zur Überset-

zungszeit umfangreiche Optimierungen durchführt und sehr performanten Code erzeugt.

2.2.2.3. Reflexion / dynamische Metaprogrammierung

In reflexiven Sprachen mit dynamischem Typkonzept wie CLOS und Smalltalk ist es möglich, durch *Metaobjektprotokolle (MOP)* die Verteilung der Nachrichten (Dispatching) zu beeinflussen und so Nachrichten abzufangen und zu verändern. Durch diese umfangreichen und nichtinvasiven Einflussmöglichkeiten auf das eigentliche Programm, lassen sich mit MOP umfangreiche Spracherweiterungen realisieren. MOP bieten damit zusätzliche Dimensionen zur Dekomposition.

2.2.2.4. Entwurfsmuster

Entwurfsmuster [GoF94] sind eine Methode um typische Entwurfsprobleme zu identifizieren, zu benennen und ihre Lösung und Realisierung wieder zu verwenden. Entwurfsmuster erlauben es auf der Entwurfsebene mit höheren Abstraktionen zu arbeiten und bieten eine Anleitung, wie sich diese höheren Abstraktionen in üblicherweise objektorientierten Sprachen realisieren lassen. Sie sind inzwischen ein gebräuchliches und bewährtes Hilfsmittel bei der Dekomposition von Softwaresystemen. Neuere Sprachen (z.B. C# und die weiteren .NET-Sprachen) integrieren bewährte Muster wie Singleton oder Beobachter bereits in den nativen Sprachumfang.

2.2.2.5. Viewpoints

Das *Viewpoints* Modell [BKF94, FS96] ist ein Ansatz für mehrdimensionales Trennen der Belange in den frühen Phasen. Obwohl prinzipiell nicht darauf beschränkt, liegt der bisherige Schwerpunkt auf dem Bereich der Anforderungsanalyse. Die Idee ist, sowohl Sichten als auch Vorgehensweisen (*views*) der einzelnen Beteiligen durch *Viewpoints* zu kapseln. In [BKF94] werden Viewpoints definiert als *schwach gekoppelte*, *lokal verwaltete*, *verteilbare Objekte*, die *partielles Wissen* über *Spezifikation*, *Repräsentation* und *Entwicklungsprozess* eines Systems kapseln. Verschiedene Viewpoints können dabei dieselben Belange in unterschiedlichen Notationen beschreiben, wobei zunächst auch widersprüchlicher Beschreibungen (z.B. Varianten) zugelassen werden. Im Rahmen eines Integrationsprozesses müssen derartige Inkonsistenzen dann aufgelöst werden. Die eigentliche Integration erfolgt manuell, die kritischen Punkte können jedoch vom System (mit Hilfe von Theorembeweisern) gefunden werden.

2.2.2.6. Generatives Programmieren

Die *Generative Programmierung* [CE00] verfolgt die Idee, eine bestimmte Art von Softwareprogrammen automatisch zu erstellen. Dazu wählt ein *Generator* anhand bestimmter Regeln und Wünsche (Konfiguration) aus einer Menge wohl definierter Komponenten die „Richtigen“ aus und verbindet sie geeignet. Das ist vergleichbar mit Produktionsstraßen aus der Automobilindustrie: Ein Auto besteht aus einer Anzahl verschiedener Komponenten (von Motor und Getriebe bis zur Innenraumausstattung), die nach Kundenwunsch in unterschiedlichsten Kombinationen zusammengestellt werden. Natürlich

kommt dabei immer ein Auto heraus, aber die genaue Gestalt des Autos fällt abhängig vom Kundenwunsch sehr unterschiedlich aus. Entsprechend spricht man bei der Generativen Softwarentwicklung von *Software product-lines* oder *product families*, in denen ein domänenspezifisches Softwareprodukt (z.B. eine SAP R3-Anwendung) aus vorhandenen Komponenten (SAP R3-Modulen) *automatisch* nach Kundenwunsch ausgewählt und aggregiert wird. „Automatisch" bildet hier die Abgrenzung zur Generischen Programmierung, bei der man zwar ebenfalls flexibel kombinierbare Komponenten anstrebt, diese jedoch manuell auswählen und kombinieren muss.

2.2.2.7. Aspektorientierte Programmierung

Die *Aspektorientierte Programmierung (AOP)* adressiert das Problem der *quer schneidenden Belange*. Es hat sich herausgestellt, dass sich eine Reihe von vorwiegend technischen Belangen, wie Synchronisation, Fehlerbehandlung oder Sicherheit, mit der klassischen objektorientierten Dekomposition nicht separieren lassen. Die Realisierung derartiger Belange ist vielmehr über eine oft große Anzahl von Artefakten verteilt, sie *berührt* oder *schneidet* diese Artefakte. Ziel der AOP ist es, Konstrukte bereitzustellen, die es erlauben derartige Belange als eigenständige Einheiten (first-class entities) zu behandeln und damit eine zusätzliche Dimension für die Dekomposition bereit zu stellen. Ursprünglich von der XEROX PARC Arbeitsgruppe um GREGOR KICZALES im Rahmen der Arbeiten für *AspectJ* eingeführt, dient AOP heute als Sammelbegriff für sämtliche Ansätze zur Behandlung quer schneidender Belange. Beispiele sind, neben dem bereits erwähnten AspectJ, *Adaptive Programming (AP)*, *Subject-Oriented Programming (SOP)* und *Composition Filters (CF)*.

Die Aspektorientierte Programmierung stellt einen der Schwerpunkte dieser Arbeit dar und wird in Kapitel 3 genauer vorgestellt.

2.2.2.8. Hyperspaces

Der *Hyperspace-Ansatz* hat zum Ziel multidimensionales Trennen der Belange über sämtliche Phasen des Softwareentwicklungszyklus zu ermöglichen. Die Elemente der eine Software beschreibenden Artefakte (*Units*) werden dabei in einem mehrdimensionalen Raum (*Hyperspace*) angeordnet. Jede Achse dieses Raumes entspricht einer *Beschreibungsdimension* von Belangen, jeder Punkt auf einer Achse einem *Belang* in dieser Dimension. Belange erstrecken sich typischerweise simultan über mehrere Dimensionen und werden durch mehrere Units beschrieben. Eine einzelne Dimension kann als eine bestimmte Dekompositionsrichtung gesehen werden, mit allen in dieser Dimension relevanten Belangen.

Neben der *Identifikation* und *Kapselung* von Belangen verfolgt der Hyperspace-Ansatz insbesondere die *Integration* der verschiedenen Beschreibungsdimensionen in so genannte *Hypermodules*. Damit kommt von den hier angesprochenen Techniken der Hyperspace-Ansatz der Vision einer echten multidimensionalen Trennung der Belange am nächsten. Es handelt sich um einen noch recht jungen Ansatz, mit Hyper/J steht aber bereits ein entsprechendes Tool für Java zur Verfügung.

Der Hyperspace-Ansatz bildet den Schwerpunkt dieser Arbeit. Er wird in Kapitel 4 detailliert vorgestellt. Dabei werden zunächst das theoretische Modell in Abschnitt 4.1 und die Funktionsweise von Hyper/J in Abschnitt 4.2 erörtert, um diese dann in Abschnitt 4.3 an einem konkreten Beispiel zu verwenden. Anschließend wird in Kapitel 5 diskutiert, inwiefern die Ideen von Hyper/J auch auf die früheren Phasen der Softwareentwicklung übertragen werden können, welche Probleme dabei auftauchen und wie man das Modell modifizieren könnte.

Kapitel 3

Aspektorientierte Programmierung

Das objektorientierte Paradigma hat sich erfolgreich als Methode zur Trennung der Belange in der Softwareentwicklung durchgesetzt. Es hat sich jedoch gezeigt, dass es eine Reihe von Eigenschaften gibt, die sich nur schlecht mit den Mitteln der OOP ausdrücken lassen. Dieses trifft insbesondere auf spezielle, eher technische Anforderungen zu, wie z.B. Parallelverarbeitung, Echtzeitfähigkeit, verteilte Systeme, Performance-Optimierung, Sicherheit oder Fehlerbehandlung. Die *Aspektorientierte Programmierung (AOP)* tritt an, dieses Problem zu überwinden.

In diesem Kapitel sollen nun die bekanntesten Ansätze für Aspektorientierte Programmierung vorgestellt werden. Dabei wird zunächst das Problem des *code-tangling* dargestellt, das die Ausgangsmotivation für die Entwicklung neuerer Techniken und des Aspektbegriffs bildete. Anschließend wird der (etwas überladene) Begriff „Aspekt“ genauer definiert, um daraus grundsätzliche Anforderungen an ein System zur Programmierung mit Aspekten zu formulieren.

Den umfangreichsten Teil des Kapitels nimmt die Vorstellung von konkreten Ansätzen zur Aspektorientierten Programmierung in Anspruch. Dabei werden zunächst kurz die *Traditionellen Vorgehensweisen* (Entwurfsmuster und Metaprogrammierung) angesprochen, mit denen auch in den klassischen OOP-Sprachen eine teilweise Separierung von Aspekten möglich ist. Anschließend werden mit *Demeter / Adaptive Programming, Subject-Oriented Programming* und *Composition Filters* die drei „großen Säulen“ der AOP beschrieben, die seit den frühen 90er Jahren entwickelt werden. All diese Ansätze hatten Einfluss auf die Entwicklung der *Aspektorientierten Programmierung mit AspectJ*. Diese heute wohl bekannteste Sprache zur AOP wird ebenfalls vorgestellt.

Den Abschluss bildet eine Beschreibung des *Intentional Programming* Projekts, das mit ausgesprochen hochgesteckten Zielen versuchte den Umgang mit Entwicklungssprachen zu revolutionieren. Das Projekt wurde inzwischen gestoppt, es erwies sich wohl doch als ein wenig zu ehrgeizig. Ich stelle es hier vor, da es sehr schön zeigt, wie weit man zumindestens in Gedanken die Idee des Separierens von Belangen (Intentionen) treiben kann.

3.1. Die Grenzen der Objektorientierung

Das objektorientierte Paradigma favorisiert die Modellierung der Eigenschaften und Funktionalitäten eines Softwaresystems mit Hilfe von *Klassen* und *Objekten*. Es gibt jedoch eine Reihe von Eigenschaften, die sich nur schlecht in Form von Klassen und Objekten ausdrücken lassen. Gerade moderne Software hat heute neben der eigentlichen Funktionalität eine große Anzahl eher technischer Eigenschaften zu erfüllen, wie z.B. Parallelverarbeitung, Echtzeitfähigkeit, Funktionsfähigkeit in verteilten Systemumgebungen, Sicherheitsanforderungen oder Fehlertoleranz.

Wir müssen daher trennen zwischen *grundlegenden Belangen (basic concerns)*, welche die eigentliche Applikationslogik darstellen, und *speziellen Belangen (special concerns),* die weitergehende Anforderungen an die Applikation formulieren [HL95]. Während man erstere gut mit den Mitteln der OOP in den Griff bekommt, versagt die reine Objektorientierung bei der Modularisierung der speziellen Belange. Mit dem klassischen Vererbungsansatz der OOP führt das zu einer Explosion der Klassenzahl. Wenn neue Eigenschaften durch Vererbung in die Klassenhierarchie eingebracht werden, verdoppelt sich im Extremfall die Anzahl der Klassen pro Eigenschaft. Als Lösung für dieses Problem wird gerne die Mehrfachvererbung angeführt, mit der sich zusätzliche Eigenschaften durch Mixin-Klassen einfach zu bestehenden Klassen „beimischen“ ließen. Die Anreicherung durch Mixin-Klassen funktioniert in den gängigen Sprachen (wie etwa C++) jedoch nur, wenn es keine Abhängigkeiten zwischen dem einzufügenden Konzept und der bestehenden Klasse gibt. In der Praxis tritt eine derartige Orthogonalität eher selten auf, bei quer schneidenden Belangen wohl gar nicht. Für derartige Belange muss man dann, genau wie bei der Einfachvererbung, auch bei der Mehrfachvererbung eine große Anzahl von Methoden überschreiben. Das folgende Beispiel verdeutlicht dieses für die Einfachvererbung, der Aufwand wäre jedoch auch mit mehrfacher Vererbung nicht geringer.

3.1.1. Das Problem der Vererbungsanomalien

Das Problem, das man für das Hinzufügen einer vergleichsweise einfachen Eigenschaft zu einer Klasse mittels Vererbung nahezu alle Methoden überschreiben muss, wird als *Vererbungsanomalie* bezeichnet [MWY90]. Angenommen wir haben eine Klasse `Stack`, die einen generischen Stapel implementiert. Diese könnte beispielsweise folgendermaßen aussehen:

```
template < class TYPE >
class Stack
{
public:
  enum error_t { errEmpty, errFull };

  Stack( int _size = 100 )
     : sp( -1 ), size( _size )
  {
    // allocate as raw buffer to prevent constructor calls
    elements = (TYPE*) new char[ _size * sizeof TYPE ];
  }
```

```
  virtual ~stack()
  {
    // call destructor explicitly only for existing elements
    while( sp >= 0 ) elements[ sp-- ].~TYPE();
    delete (void*) elements;
  }
  virtual void push( const TYPE& e )
  {
    if( isFull() ) throw errFull;
    // Increment SP and copy-construct element at this position
    new ( &elements[ ++sp ] ) TYPE( e );
  }
  virtual void pop()
  {
    if( isEmpty() ) throw errEmpty;
    // Destroy element and decrement SP
    elements[ sp-- ].~TYPE();
  }
  virtual const TYPE& top() const
  {
    if( isEmpty() ) throw errEmpty;
    return elements[ sp ];
  }
  virtual bool isEmpty() const
  {
    return sp == -1;
  }
  virtual bool isFull() const
  {
    return sp == MAX_SIZE - 1;
  }
private:
  int   size;                 // max number of elements
  int   sp;                   // Stackpointer
  TYPE  elements[ MAX_SIZE]; // Buffer
};
```

Die Klasse `stack` ist in dieser Form nicht für parallele Zugriffe von mehreren Threads geeignet. Da eine derartige Verwendung eher die Ausnahme als die Regel darstellt und jede Art von Synchronisation einen gewissen Laufzeitaufwand verursacht, ist das jedoch prinzipiell erstmal eine gute Designentscheidung.

Nun tritt aber der Fall ein, dass eine threadfeste Variante der Klasse `stack` benötigt wird. Nach guter alter OOP-Manier würde man dazu eine neue Klasse `syncStack` von `stack` ableiten. Für die Threadsynchronisation stehe uns dazu von Seiten des Betriebssystems eine Klasse `mutex` mit den typischen Operationen `mutex::lock()` und `mutex::unlock()` zur Verfügung. Ferner verwenden wir eine Hilfsklasse `auto_lock`, die in ihrem Konstruktor einen Mutex sperrt und diesen in ihrem Destruktor wieder freigibt:

```
class auto_lock
public:
  auto_lock( mutex& m )
    : mut( m )
  {
    mut.lock();
  }
  ~auto_lock()
  {
    mut.unlock();
  }
  mutex& mut;
};
```

Legt man zu Beginn eines Codeblocks eine Instanz von `auto_lock` an, so wird der übergebene Mutex automatisch gesperrt. Wird dieser Codeblock irgendwann mal wieder verlassen, so wird der Mutex automatisch wieder freigegeben. Da C++ garantiert, dass selbst

beim Auftreten einer Ausnahme die Destruktoren aller lokalen Variablen aufgerufen werden, gilt dieses sogar bei einem Laufzeitfehler.[3]

Damit können wir nun die Klasse `SyncStack` definieren. Jede Instanz von `SyncStack` erhält einen Mutex, über den der gegenseitige Ausschluss unter Zuhilfenahme von `auto_lock` realisiert wird:

```
template < class TYPE >
class SyncStack : public Stack< TYPE >
{
public:
  SyncStack( int _size = 100 )
     : Stack( _size )
  {
  }
  virtual void push( const TYPE& e )
  {
    auto_lock lock( mut );
    Stack::push( e );
  }
  virtual void pop()
  {
    auto_lock lock( mut );
    Stack::pop();
  }
  virtual const TYPE& top() const
  {
    auto_lock lock( mut );
    return Stack::top();
  }
  virtual bool isEmpty() const
  {
    auto_lock lock( mut );
    return Stack::isEmpty();
  }
  virtual bool isFull() const
  {
    auto_lock lock( mut );
    return Stack::isFull();
  }
private:
  mutex mut;
};
```

Um die Klasse `SyncStack` zu definieren, mussten wir (mit Ausnahme des Destruktors) *alle Methoden* überschreiben. Weiterhin fällt auf, dass die Methoden immer gleich aussehen: Sie sperren den Mutex und rufen die entsprechende Funktion der Superklasse auf. Ein hoher Aufwand für eine so kleine Ergänzung.

Zusätzlich droht noch eine weitere Gefahr. Angenommen, die Klasse `Stack` wird im Rahmen der Programmevolution um eine Methode `Stack::clear()` erweitert, die es ermöglicht, den gesamten Stapel mit nur einem Aufruf zu leeren. Diese Methode würde dann auch an die Klasse `SyncStack` vererbt und stände den Klienten von `SyncStack` ebenfalls zur Verfügung. Damit wäre die Klasse `SyncStack` jedoch nicht mehr threadfest, da beim Aufruf von `clear()` der Mutex nicht gesperrt wird! Dazu wäre es nämlich erforderlich die Methode ebenfalls in `SyncStack` zu überschreiben – und das wird in der Praxis allzu leicht übersehen. Faktisch bedeutet dieses, dass bei einer Erweiterung der Oberklasse *sämtliche* Unterklassen zu prüfen sind und eine neue Methode der Oberklasse im

[3] Die Verwendung einer Hilfsklasse, um bestimmten Code automatisch beim Betreten oder Verlassen eines Blocks auszuführen, ist ein unter dem Namen *Konstruktor-Destruktor-Klammer* bekanntes C++ Idiom.

Extremfall zusätzlich in jeder Unterklasse überschrieben werden muss. Anstatt sich durch die Zusammenfassung des gemeinsamen Verhaltens in einer Oberklasse Arbeit zu sparen, tritt das Gegenteil ein – eine Vererbungsanomalie.

3.1.2. Das Problem des code tangling

Der Grund für die oben geschilderten Anomalien ist, dass Synchronisation ein klassischer quer schneidender Belang ist. Um die Threadsicherheit zu erreichen, ist es erforderlich in *jede Methode* etwas Code einzufügen. Dieses Problem wird als *code tangling* bezeichnet: Zur Realisierung einer speziellen Eigenschaft muss an vielen Stellen eine kleine Menge Code in die eigentliche Applikationslogik hineingemischt werden – mit großen Nachteilen für deren Verständlichkeit, Wartbarkeit und Wiederverwendbarkeit.

Derartige Belange werden daher als Querschnittsbelange oder *Aspekte* bezeichnet. Die verschiedenen Ansätze, die das Problem des *code tangling* adressieren indem sie die Trennung von Aspekten und Fachkonzepten bis auf die Implementierungsebene unterstützen, werden unter dem Oberbegriff *Aspektorientierte Programmierung (AOP)* zusammengefasst.[4]

3.2. Aspektorientierte Programmierung

Das Ziel der Aspektorientierten Programmierung (AOP) liegt in der Bereitstellung von Methoden und Techniken zur Dekomposition eines Problems *sowohl* in seine funktionalen Anteile (Fachkonzepte, Geschäftslogik) *als auch* in eine Reihe von Aspekten – Konzepte, die als Querschnittseigenschaften die Fachkonzepte überdecken [CE00 S.251ff].

3.2.1. Zum Aspektbegriff

Der Begriff „Aspekt“ wird in der Softwareentwicklung auf unterschiedlichste Weise verwendet. Die große und weiter steigende Popularität der AOP in der Forschungslandschaft führt außerdem zu der (etwas unglücklichen) Tendenz, dass immer mehr Ideen und Ansätze unter den Aspektbegriff und den Mantel der AOP gestellt werden. Um Missverständnisse zu verhindern, möchte ich deshalb den Aspektbegriff und seine Bedeutungen – so wie ich sie hier verstehe und verwende – etwas schärfer herausarbeiten.

Meiner Beobachtung nach wird der Begriff „Aspekt“ im Rahmen der Softwareentwicklung vorwiegend auf dreierlei Weise verwendet: *umgangssprachlich* (Aspekt im wörtlichen Sinn), *technisch* (Aspekt im Sinne von AOP als quer schneidendes Konzept oder Model-

[4] Die Begriffe *Aspect* und *Aspect Oriented Programming (AOP)* wurden Mitte der 90er Jahre von der XEROX PARC Arbeitsgruppe um GREGOR KICZALES im Rahmen der Arbeiten für AspectJ erfunden. In der Literatur wird daher häufig AOP synonym mit AspectJ verwendet – ungeachtet der Tatsache, dass ältere Arbeiten z.B. von KARL LIEBERHERR und CHRISTINA LOPES (Adaptive Programming) bzw. MEHMET AKSIT und LODEWIJK BERGMANS (Composition Filter) bereits gleiche oder ähnliche Ziele verfolgen. Ich verstehe unter AOP hier daher *alle* Ansätze zur Programmierung mit Aspekten.

lierungskonstrukt) und *in der Begriffswelt von AspectJ* (Aspekt als syntaktisches Sprachkonstrukt in AspectJ):

Der umgangssprachliche Aspektbegriff

Die Bedeutung des Wortes „Aspekt" leitet sich aus dem lateinischen (*„das Hinsehen"*) ab, seine wörtliche Bedeutung wird im DUDEN mit *Blickwinkel, Blickpunkt, Betrachtungsweise, Gesichtspunkt* angegeben.

In der Umgangssprache wird „Aspekt" jedoch als Synonym für Eigenschaften und Anforderungen jeglicher Art gebraucht. Im diesem weiten Sinn kann nahezu jede Aussage über ein System als „Aspekt des Systems" bezeichnet werden. Dementsprechend möchte ich umgangssprachliche Aspekte als *Aspekte im weiteren Sinne (i.w.S.)* bezeichnen.

Der technische Aspektbegriff

Unter „Aspekten im technischen Sinn" verstehe ich identifizierbare und benennbare Konstrukte der Modellierung (first-class entities), die andere Modellierungskonstrukte quer schneiden. AOP handelt von Aspekten im technischen Sinn, entscheidend für die Anwendung des technischen Aspektbegriffs ist die Querschnittseigenschaft. (Die Abgrenzung zwischen Aspekten und Fachkonzepten wird in Abschnitt 3.2.2 noch genauer ausgeführt.) Aspekte als Modellierungskonstrukt möchte ich als *Aspekte im technischen Sinne (i.t.S.)* oder, da sie hier den Schwerpunkt bilden, einfach kurz als *Aspekte* ohne weitere Qualifizierung bezeichnen.

Der Aspektbegriff von AspectJ

Die Sprache AspectJ, die in Abschnitt 3.7 vorgestellt wird, verwendet den Begriff `aspect` als ein Schlüsselwort zur Definition quer schneidender Code-Abschnitte. Ein Aspekt im Sinne von AspectJ ist, ähnlich wie eine Klasse, ein eigenständiges Artefakt. Diese Art von Aspekten wird im weiteren Verlauf als *Aspekte im Sinne von AspectJ (i.S.v. AspectJ)* oder, entsprechend dem AspectJ-Schlüsselwort, als `aspect` bezeichnet.

Natürlich gibt es einen Zusammenhang zwischen den verschiedenen Aspektbegriffen. So ist ein Aspekt im Sinne von AspectJ auch ein Aspekt im technischen Sinne und jeder Aspekt im technischen Sinne kann auch umgangssprachlich als Aspekt im weiteren Sinne gesehen werden. Der Umkehrschluss jedoch gilt nicht! Gerade aus diesem Grund halte ich es für wichtig jeweils klarzustellen, welcher Aspektbegriff gemeint ist. Sonst besteht die Gefahr, dass ein AspectJ Entwickler immer, wenn von Aspekten die Rede ist, sie sich geistig bereits als `aspect` vorstellt.

3.2.2. Was einen Aspekt ausmacht

Ein Konzept ist ein modellierter Belang eines Softwaresystems. Konzepte treten auf als Fachkonzepte[5] (funktionale Anteile der Software) oder als Aspekte (i.t.S.). Der entscheidende Unterschied zwischen Fachkonzepten und Aspekten (i.t.S) ist die Querschnittseigenschaft. Ein Konzept ist genau dann ein Aspekt, wenn es eine Reihe von Fachkonzepten berührt. Die Aspekteigenschaft ist somit *relativ zum eigentlichen Modell* zu sehen [CE00 S.265]. Ein und dasselbe Konzept kann sowohl ein Aspekt als auch ein Fachkonzept sein. Eine typische Datenbankanwendung muss z.B. große Teile des Fachcodes in Transaktionen einbetten, was das Transaktionskonzept zu einem Aspekt der Anwendung macht. Der Entwickler eines Datenbankkerns würde denselben Transaktionsbegriff hingegen als ein Fachkonzept der Datenbank auffassen.

Des Weiteren ist festzuhalten, dass es Aspekte auf allen Ebenen des Softwareentwicklungsprozesses gibt. Auf der Implementierungsebene ist ein Konzept genau dann ein Aspekt, wenn es mehrere separate Code-Module berührt. Auf der Entwurfsebene ist ein Aspekt ein Entwurfsteil, der die Struktur anderer Entwurfsteile berührt usw. Ein Konzept kann demnach beim Übergang von einer Ebene auf die nächste (z.B. vom Entwurfsmodell zum Implementierungsmodell) „plötzlich“ zum Aspekt werden. Das passiert genau dann, wenn es sich in der auf dieser Ebene verwendeten Sprache nicht mehr als first-class Entität ausdrücken lässt, sondern sich verteilt auf andere Entitäten wieder findet.

Ein schönes Beispiel dafür ist die Anwendung von Entwurfsmustern: Eine Entscheidung im Entwurfsmodell, wie „A ist ein Beobachter von B“, muss im Implementierungsmodell durch zusätzliche Methoden in A und B, Änderung von Oberklassen oder Erstellung von Hilfsklassen usw. umgesetzt werden. Dabei ist *Beobachter* mit nur wenigen beteiligten Klassen noch vergleichsweise harmlos. Andere Muster wie *Abstrakte Fabrik* oder *Besucher* berühren typischerweise eine sehr große Anzahl von Klassen und sind nicht zu unrecht wegen ihres Wartungsaufwandes in der Praxis umstritten. Es scheint eine inhärente Eigenschaft von Entwurfsmustern zu sein, dass sie quer schneidend sind und deshalb zu umfangreichem „code tangling“ im Implementierungsmodell führen.

3.3. Realisierungstechniken für AOP

3.3.1. Grundsätzliche Anforderungen

Um aspektorientiert programmieren zu können bedarf es einer *Sprache* um Aspekte *definieren* zu können. Des Weiteren brauchen wir geeignete *Realisierungstechniken*, die Fachkonzepte und Aspekte miteinander verbinden. Spätestens beim Einsatz in der Praxis be-

[5] Mit dem Begriff *Fachkonzept* wird häufig auch das vollständige Analysemodell einer Anwendung bezeichnet. Hier und im Folgenden ist unter *Fachkonzept* hingegen ein einzelner, funktionaler Belang des Softwaresystems zu verstehen.

steht außerdem der Wunsch nach geeigneten *Werkzeugen*, die uns den Umgang mit Aspekten erleichtern. Im Einzelnen:

- Wir brauchen eine geeignete *Sprache*, in der sich Aspekte formulieren und ausdrücken lassen. Die Sprache sollte generisch genug sein, um Aspekte aus unterschiedlichsten Bereichen zu unterstützen. Um eine klare Trennung der Belange zu erreichen, sollten Aspekte außerdem als eigenständige Einheiten getrennt von den Fachkonzepten definiert werden können.
- Ferner werden *Realisierungstechniken* benötigt, um Aspekte zu kombinieren, zu verändern, sie den Fachkonzepten hinzuzufügen oder wieder zu entfernen. Nach [CE00 S.267ff] stellen sich die grundlegenden Anforderungen daran wie folgt dar:
  - Die Kompositionstechnik sollte eine *minimale Kopplung* zwischen Aspekten und Fachkonzepten ermöglichen. Eine vollständige Separierung ist, außer in trivialen Fällen, nicht möglich, da die wenigsten Aspekte unabhängig vom Kontext sind und beliebig kombiniert werden können.
  - Bei der Komposition sollten verschiedene *Bindungszeiten und -modi* unterstützt werden. Es sollte möglich sein, Aspekte sowohl statisch (zur Übersetzungszeit) als auch dynamisch (zur Laufzeit) an die Fachkonzepte zu binden.
  - Außerdem sollte *nicht-invasives Hinzufügen von Aspekten zu existierendem Code* unterstützt werden. Erst damit ist es möglich ein hohes Maß an Adaptierbarkeit der Fachkonzepte zu erreichen.
- Zudem ist eine umfangreiche *Toolunterstützung* wichtig. Für einen produktiven Einsatz in der Praxis sollten sich die Konzepte möglichst nahtlos in die Entwicklungsumgebung integrieren lassen und die Arbeiten höchstmöglich automatisierbar sein. Eine große Herausforderung stellt hierbei die Integration in Debugger und Wartungswerkzeuge dar, schließlich will man auch bei der Fehlersuche und bei der Programmevolution in der Lage sein, zwischen Fachcode und Aspektcode zu trennen.

3.3.2. AOP mit den Mitteln herkömmlicher Sprachen

Unter *Aspektorientierter Programmierung in herkömmlichen Sprachen* verstehe ich jene Ansätze, die versuchen das Problem der quer schneidenden Konzepte ohne spezielle Erweiterung von Compiler, Sprache und Laufzeitsystem zu lösen. Ansätze dazu kommen zum einen aus der Pattern-Community, wo mit Hilfe von Entwurfsmustern versucht wird quer schneidende Konzepte zu *objektifizieren*, also in eigene Objekte auszulagern. Zum anderen sind hier die Ansätze zur Metaprogrammierung zu nennen, von der statischen C++ Template-Metaprogrammierung bis zur dynamischen Metaprogrammierung in reflexiven Sprachen wie Smalltalk oder CLOS.

3.3.2.1. Entwurfsmuster

Entwurfsmuster [GoF94] bieten konkrete Lösungsansätze und Techniken um verschiedene Bestandteile eines Entwurfs zu entkoppeln. Ziel ist es *Variationspunkte* zu realisieren, um zukünftige und erwartete Erweiterungen des Programms ohne umfangreiche Änderungen des Entwurfs durchführen zu können. Entwurfsmuster basieren auf den klassischen Techniken der OOP, allen voran Schnittstellenvererbung (im Sinne von Substituierbarkeit) und Objektkomposition.

Einige quer schneidender Konzepte lassen sich durch die Anwendung von Entwurfsmustern gut separieren. Genannt seien hier beispielhaft das Iteratormuster, das eine lose Kopplung von Algorithmen und Datenstrukturen erlaubt oder das Dekorierermuster, mit dem Funktionalität dynamisch zu Komponenten hinzugefügt werden kann. Der Einsatz von Entwurfsmustern hat jedoch oft Folgeerscheinungen, wie z.B. zusätzliche Indirektionen, Verlust der Identitätsbeziehung und vor allem viel „manuelle Tipparbeit“. Die Anwendung eines Entwurfsmusters fügt häufig zusätzliche Klassen, Methoden und Schnittstellen ein, wodurch der Überblick und das Verständnis des Codes leiden. Für einige klassische Querschnittsaspekte wie Synchronisierung oder Tracing, die insbesondere einzelne Methoden quer schneiden, bieten Entwurfsmuster keine gute Lösung.

Nichtsdestotrotz sind Entwurfsmuster eine brauchbare Realisierungstechnik für Aspekte. So zeigen CONSTANTINIDES und ELRAD in [CE00a], wie sich mit Entwurfsmustern ein generisches Framework zur AOP realisieren lässt. Auch die „höheren“ AOP-Sprachen verwenden intern Muster wie Proxy, Dekorierer oder Besucher zur Realisierung.

3.3.2.2. Metaobjektprotokolle (MOP)

In reflexiven Sprachen mit dynamischem Typkonzept, allen voran Smalltalk, ist es üblich Querschnittseigenschaften auf die Metaebene zu verlagern. In diesen Sprachen ist es möglich durch *Metaobjektprotokolle (MOP)* die Verteilung der Nachrichten (Dispatching) zu beeinflussen und so Nachrichten abzufangen und zu verändern. Prinzipiell lassen sich auf diese Weise beliebige Aspekte einfügen und zur Laufzeit anpassen [Sul01].

Insbesondere in Smalltalk hat die Verwendung von MOP eine große Tradition. Smalltalk ist von seiner Struktur her so angelegt, dass es nur sehr wenige syntaktische Sprachkonstrukte gibt und nahezu der gesamte Funktionsumfang der Sprache, einschließlich so elementarer Dinge wie Schleifen und Bedingungen, durch die Laufzeitbibliothek realisiert wird. An Stelle syntaktischer Regeln treten Sprachidiome, die regeln was „gültige“ Konstrukte sind. Durch das kaum vorhandene syntaktische Korsett und das umfangreiche MOP lässt sich der Sprachumfang (in Form von Idiomen) beliebig erweitern. Somit ist die Verwendung von MOP eine nahe liegende Wahl zur Realisierung von AOP in Smalltalk. Das von ROBERT HIRSCHFELD betriebene *AspectS Projekt* [AspectS] verwendet diesen Ansatz.

Ein grundsätzlicher Nachteil bei der Verwendung von MOP ist neben der Tatsache, dass alles erst zur Laufzeit geschieht und entsprechend langsam ist, das Problem mehrere derartige Erweiterungen zu kombinieren. Das Problem der Kombinierbarkeit und des nicht-invasiven Hinzufügens von Konzepten stellt sich auf der Meta-Ebene erneut.

3.3.2.3. C++ Template Metaprogrammierung

Die Sprache C++ bietet durch Konzepte wie Templates und Overloading umfangreiche Möglichkeiten zur statischen Metaprogrammierung. Die Verwendung dieser (eigentlich nur zur Realisierung generischer Bibliotheken in den Sprachumfang aufgenommenen) Sprachmittel zur Metaprogrammierung und für generative Techniken ist ein vergleichsweise junges Feld in Forschung und Praxis.[6] Die C++ Community hat dabei in den letzten Jahren eine ganze Reihe von Idiomen und Techniken entwickelt, die in hohem Maße an aspektorientierte Programmierung erinnern, hier nur zwei Beispiele:

- In [Ale01] stellt ANDREI ALEXANDRESCU die Utilitybibliothek *Loki* vor, die auf dem Konzept so genannter *Policies* und des *Policy-Based Class Design* basiert. Policies sind dabei kleinste Bausteine, die einen ganz bestimmten Belang isoliert kapseln. Er verwendet Policies dann, um typische quer schneidende Belange wie Synchronisation zu realisieren. Das erinnert sehr stark an Aspekte.
- In [Str00] zeigt BJARNE STROUSTRUP, wie man mit Hilfe von speziellen Smart Pointern automatisch Code vor und nach dem Betreten einer Memberfunktion ausführen lassen kann. Derartiges *Method-Wrapping* ist ein wesentlicher Bestandteil von AOP Techniken.

Interessanterweise gibt es bislang jedoch noch keine Arbeit, in der die Möglichkeiten von Standard C++ gezielt auf ihre Verwendbarkeit für AOP untersucht werden und z.B. mit den Sprachmitteln von AspectJ verglichen werden.[7]

Ein Problem der C++ Template-Sprache ist, dass sie eigentlich nicht zur Metaprogrammierung in derartigem Umfang gedacht war. Als Turing-mächtige Sprache ist sie zwar theoretisch für alles geeignet, aber die zugrunde liegenden „Sprachmittel" sind hässlich und undurchschaubar und die Fehlersuche ist weit mehr als nur diffizil. Ein weiteres Problem ist, dass viele bekannte Compiler nicht in der Lage sind, alle im C++ Standard definierten Template-Konstrukte zu übersetzen. Hier zeichnet sich aktuell jedoch Besserung ab: Sogar die nicht gerade für ihre Standardtreue bekannte Firma Microsoft hat sich vom Saulus zum Paulus gewandelt und angekündigt, den sehr weit verbreiteten Visual C++ Compiler in der für nächstes Jahr angekündigten Version 8.0 zum „standardkonformsten C++ Compiler überhaupt" zu machen.

---

[6] *„What's left to say about C++ that hasn't already been said? Plenty, it turns out."*

JOHN VLISSIDES im Vorwort zu [Ale01].

[7] Man könnte den Eindruck bekommen, C++ würde von AOP Community nahezu vollständig ignoriert. So gab es auf der ersten internationalen Konferenz zum Thema (AOSD 2002 [Kic02]) eine Vielzahl von Beiträgen im Zusammenhang mit Java, jedoch *nicht einen*, der sich mit C++ beschäftigte. Mit AspectC [CKFS01] und AspectC++ [SGS02, AspectC++] gibt es zwar erste Ansätze für AOP mit C/C++, diese verwenden jedoch nicht die oben genannten eingebauten Sprachmittel, sondern erweitern die Sprache nach dem Vorbild von AspectJ.

3.3.3. AOP mit speziellen Spracherweiterungen

Die im Folgenden beschriebenen Ansätze zur AOP haben gemein, dass sie spezielle Sprachen oder Spracherweiterungen benutzen, um Aspekte zu definieren und anzuwenden. Dazu werden eine *Aspektdefinitionssprache* zur Beschreibung der Aspekte, eine *Aspektanwendungssprache* die beschreibt, wo sie eingefügt werden sollen sowie eine *Weaver*, der den Aspektcode in den Fachcode „hineinwebt", benötigt.

Eine *Aspektdefinitionssprache* (Aspektsprache, action language) ist eine (Programmier-) Sprache, in der die quer schneidenden Konzepte als first-class Entitäten definiert und implementiert werden. Hier ist ein Trend von domänenspezifischen Aspektsprachen, wie Adaptive Programming, COOL oder RIDL [Lop97], zu general-purpose Aspektsprachen, wie Java im Falle von AspectJ, festzustellen.

In der *Aspektanwendungssprache* (cross-cut language) wird beschrieben, auf welche Weise die Aspekte mit den Fachkonzepten kombiniert werden sollen. Hier überwiegen deklarative Ansätze, bei denen die Anwendung von Aspekten durch Prädikate festgelegt wird. Unterschiede finden sich, neben der konkreten Sprache, vor allem darin wo diese Beschreibungen stehen: im Fachcode, bei der Aspektdefinition oder in einem unabhängigen, dritten Artefakt. Letzteres kann auch ein Konfigurations-Repository sein, z.B. wenn der Ansatz es erlaubt neue Aspekte zur Laufzeit hinzuzufügen.

Mit dem *Weaver* wird schließlich der Aspektcode in den Fachcode „hineinwebt". Dazu ist ein Transformationsprozess notwendig, der den entsprechenden Fachcode und die Aspektdefinitionen anhand der Regeln zur Aspektanwendung integriert und ablauffähigen Code erzeugt. Für diesen Transformationsvorgang hat sich die Bezeichnung *weaving* durchgesetzt, die ausführenden Tools werden entsprechend *Weaver* genannt.

Der eigentliche Transformationsprozess kann zu unterschiedlichsten Zeitpunkten geschehen. Die meisten Ansätze verwenden einen Precompiler, der aus Aspekten und Fachcode übersetzbaren Code in der entsprechenden Programmiersprache erzeugt. Dieser Ansatz ist vor allem aus pragmatischen Gründen beliebt, da man so herkömmliche Compiler und Laufzeitsysteme für die eigentliche Programmgenerierung und –ausführung verwenden kann. Die Integration der Aspekte zur Übersetzungszeit erlaubt es außerdem, Aspekte ohne größere Laufzeiteinbußen zu verwenden. Um Aspekte auch dynamisch hinzuzufügen oder entfernen zu können, ist hingegen ein *Runtime-Weaver* erforderlich, der es ermöglicht Aspektcode zur Laufzeit einzufügen. Natürlich sind auch Mischformen denkbar.

3.4. Adaptive Programming (AP)[8]

3.4.1. Einleitung

Die Bezeichnung *Adaptive Programming* (AP) wurde Anfang der 90er Jahre durch KARL LIEBERHERR eingeführt. Die ursprüngliche Idee war die Trennung von Verhalten (Algo-

[8] [Lie96], [Lie97], [CE00 S.259ff], [LL95]

rithmen) und Struktur (Klassengraph) durch die Einführung des Aspektes der *Traversierung* (*traversal strategies*, kurz: *traversals*). Später wurde AP noch um weitere Aspekte (i.t.S.) wie Synchronisierung und entfernte Aufrufe erweitert.

AP gilt heute als bedeutende Wurzel von AOP und versteht sich inzwischen eher als eine Untermenge von AOP, in der Aspekte in Form von Strategien über Graphen beschrieben werden [Lie97].

Demeter ist eine Methode zum Adaptive Programming und stellt eine Sprache zur Definition von Graphen und Traversierungen bereit. Demeter wurde in diverse OO Sprachen integriert, wie Demeter/C++ [Lie96], Demeter/Java [Lie97] oder AP/S++ [LL95]. Klassenstruktur und Traversierungen werden darin nach der Demeter-Methode beschrieben, während das eigentliche Verhalten der Programme (die Algorithmen) in z.B. C++ definiert wird.

3.4.2. Motivation: Strukturanomalitäten

Demeter / Adaptive Programming bezieht seine Motivation aus der Beobachtung, dass in OO Programmen oft eine sehr enge Kopplung zwischen Algorithmen und der Klassenstruktur besteht. Diese enge Kopplung entsteht, weil die Kommunikation zwischen zwei verschiedenen Objekten üblicherweise über eine Gruppe von weiteren Objekten verläuft. In Abbildung 1 ist ein solcher Fall dargestellt am Beispiel eines einfachen Texteditors. Eine Methode `Document::Search( String aWord)`, die alle Vorkommen eines angegebenen Wortes im Text markieren soll, könnte folgendermaßen aussehen (Pseudocode):

```
void Document::Search( String aWord )
{
  foreach Paragraph p in paragraphs {
    foreach Line l in p.lines {
      foreach Word w in l.words {
        if( w.value == aWord )
          w.Highlight();
      }
    }
  }
}
```

Derartiger Code gilt jedoch als unbefriedigend, weil er zu einer hohen Kopplung zwischen den Klassen führt. Im Beispiel hängt die Klasse `Document` von drei weiteren Klassen ab (`Paragraph, Line, Word`). Enge Kopplung führt jedoch zu aufwändigen Compilerläufen, geringer Wiederverwendbarkeit und schlechter Wartbarkeit, da durch die Implementierung des Verhaltens die Klassenstruktur hart codiert wird. Ändert sich die Klassenstruktur, so müssen sämtliche Algorithmen angepasst werden.

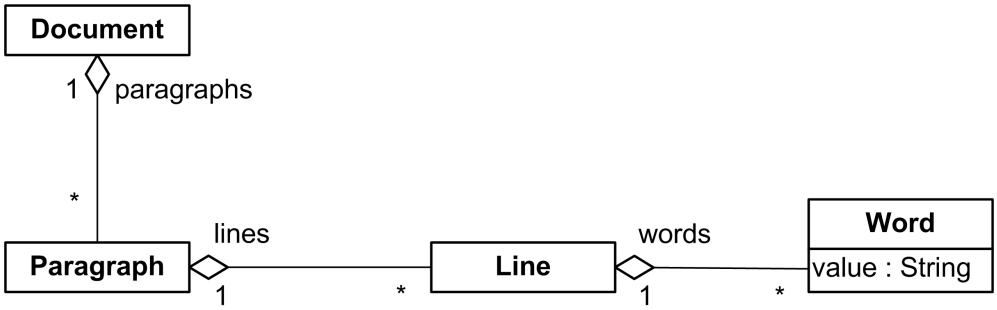

Abbildung 1 Klassenstruktur eines einfachen Editors
Ein Dokument besteht aus mehreren Paragraphen, die sich jeweils aus Zeilen zusammensetzen. Die Zeilen bestehen wiederum aus einzelnen Wörtern. (Abbildung verändert nach [LL95])

Aus diesem Grund gibt es in der OO-Gemeinde das weithin akzeptierte Prinzip des *Law of Demeter* [Lie96]. Dieses Prinzip sagt: Eine Methode soll Nachrichten ausschließlich an Objekte schicken, die im lokalen Sichtbarkeitsbereich liegen (`this`, Instanzvariablen, lokale Variablen, globale Variablen und Klassenvariablen). Die konsequente Anwendung des Law of Demeter löst daher die enge Kopplung der Klassen:

```
void Document::Search( String aWord )
{
  foreach Paragraph p in paragraphs
    p.Search( String aWord );
}
void Paragraph::Search( String aWord )
{
  foreach Line l in lines
    l.Search( aWord );
}
void Line::Search( String aWord )
{
  foreach Word w in l.words
    w.Search( aWord );
}
void Word::Search( String aWord )
{
  if( value = aWord )
    Highlight();
}
```

Bei dieser Implementierung ist jede Klasse nur noch von maximal einer weiteren Klasse abhängig. Der Preis dafür ist jedoch das, was LIEBERHERR und LOPES in [LL95] als *Strukturanomalität* bezeichen: Der eigentliche Algorithmus ist nun über die gesamte Klassenstruktur verteilt. Dabei haben die meisten Methoden rein „transportierenden" Charakter – sie tun nichts weiter, als die empfangene Nachricht an alle Kindobjekte weiter zu leiten. Darunter leidet natürlich die Übersichtlichkeit („Wo wird eigentlich die Arbeit geleistet?") und bei einer Veränderung der Signatur muss an vielen verschiedenen Stellen Code angepasst werden.

3.4.3. Lösung: Trennung von Algorithmen und Klassenstruktur

Eine Lösung für dieses Problem besteht darin, die Algorithmen (das Verhalten) nicht mehr an einen konkreten Klassengraphen zu binden, sondern an einen partiellen. Dazu wird das Konzept der Traversierungsstrategien eingeführt. Formal handelt es sich bei einer Traversierungsstrategie um ein Tupel (*source*, *Target*, *Constraints*), das eine Menge

von Pfaden über einen Klassengraphen beschreibt, ausgehend vom Knoten *source* zur Knotenmenge *Target* unter Beachtung optionaler Regeln *Constraints*, durch die bestimmte Teilpfade explizit ausgeschlossen werden können. Der Algorithmus kann nun, ausgehend von einer Instanz von *source*, über alle erreichbaren Instanzen aus *Target* iterieren, ohne die dazwischen liegenden Klassen und Beziehungen kennen zu müssen. Dafür stehen spezielle Sprachkonstrukte zur Verfügung (`from Document to Word`).

Durch dieses Konzept ist ein großes Maß an Unabhängigkeit zwischen dem Algorithmus und der konkreten Klassenstruktur gegeben. Der Suchalgorithmus im Beispiel fordert nur, dass es einen Weg geben muss um ausgehend von einer `Document` Instanz eine Nachricht an alle zugehörigen `Word` Instanzen zu senden. Diese partielle Klassenstruktur kann nun mit verschiedenen konkreten Klassengraphen instantiiert werden. Dabei wird der entsprechende Code zum Weiterreichen der Argumente durch die konkreten Klassen automatisch generiert. Wird unser Editor z.B. um Tabellen erweitert (Abbildung 2), so hat dieses zwar Modifikationen an der Klassenstruktur zur Folge, der Suchalgorithmus kann jedoch unverändert bleiben.

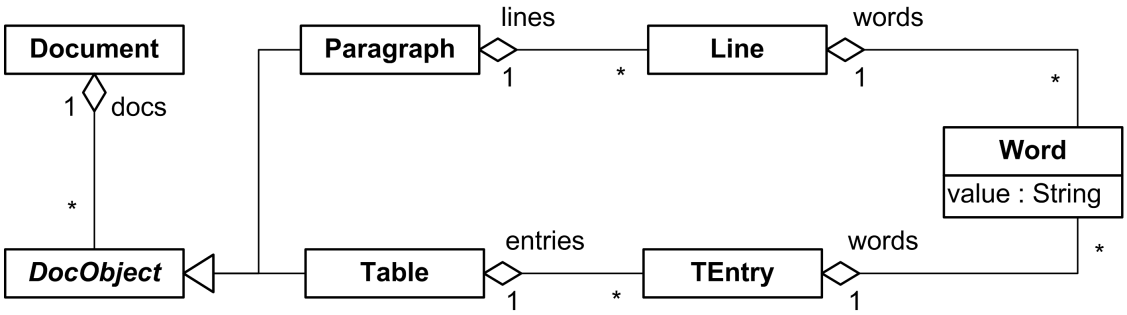

Abbildung 2 Klassenstruktur des Editors nach der Einführung von Tabellen
Ein Dokument kann jetzt sowohl Paragraphen als auch Tabellen beinhalten, womit es zwei Wege zu den einzelnen Wörtern des Dokumentes gibt. Dank der Trennung von Algorithmen und Klassenstruktur muss der Suchalgorithmus dafür nicht angepasst werden – er findet ohne Modifikation nun auch Wörter in Tabellen. (Abbildung verändert nach [LL95])

3.4.4. Bewertung

Adaptive Programming zählt zu den Ansätzen, die sich vor allem zur Separierung von *Strukturaspekten* eignen. Das Konzept der Traversierungsstrategien zeigt sehr schön, wie eine minimale Kopplung von Programmverhalten und Programmstruktur erreicht werden kann. Es löst damit Probleme, für die ansonsten häufig Entwurfsmuster wie Strategie, Iterator, Besucher usw. verwendet werden ohne deren Nachteile (viel „Handarbeit“, Übersicht im Code geht verloren, etc.) in Kauf nehmen zu müssen.

Adaptive Programming unterstützt nur statische Bindung, was jedoch kein Nachteil im Sinne der in Abschnitt 3.3.1 beschriebenen Anforderungen ist, da in nahezu allen Sprachen die Programmstruktur zur Übersetzungszeit feststeht. Da die Traversals in einer eigenen Sprache definiert werden und beim Übersetzen zusätzliche Methoden einfügen, findet der Programmierer sich beim Debuggen jedoch auf einem anderen Abstraktionsniveau wieder als bei der Entwicklung.

LIEBERHERR hat sich inzwischen stark der Aspect/J-Idee verschrieben. Neurer Ansätze streben daher eine Integration von Adaptive Programming in das Aspect/J-System an. [LOO01]

3.5. Subject-Oriented Programming (SOP)[9]

3.5.1. Einleitung

Die Idee des *Subject-Oriented Programming (SOP)* wurde Anfang der 90er Jahre von WILLIAM HARRISON und HAROLD OSSHER am T.J. Watson IBM Research Center als Erweiterung der objektorientierten Programmierung entwickelt. Im Fokus von SOP liegt das Problem der verschiedenen *subjektiven Perspektiven* auf ein einzelnes Objekt. Eine Katzenbesitzerin würde z.B. ein Objekt, das eine *Katze* repräsentieren soll, mit anderen Attributen und Operationen modellieren, als eine Tierärztin. Die Art und Weise, wie eine Katze betrachtet wird, ist *abhängig vom Betrachter* (Abbildung 3).

Ziel des SOP ist es nun, die verschiedenen Sichtweisen auf dasselbe Objekt unter Beibehaltung der semantischen Integrität, insbesondere der Identitätsbeziehung, geeignet zusammenzuführen. Realisiert wird dieses durch Spracherweiterungen zur Beschreibung und Komposition von *subjects*. Es stehen entsprechende Erweiterungen unter anderem für C++, Java und Smalltalk zur Verfügung.

Das SOP Projekt ist inzwischen aufgegangen in den Hyperspace-Ansatz, in dem die grundlegenden SOP Ideen (Integration verschiedener Sichtweisen) nochmals deutlich erweitert und auf den gesamten Softwareentwicklungsprozess übertragen werden. Der Hyperspace-Ansatz wird in Kapitel 4 im Detail vorgestellt.

3.5.2. Zentrale Begriffe

Ein SOP Programm besteht aus zwei oder mehreren Objektmodellen, genannt *subjects*, die durch *composition rules* zu einem Komposit vereint werden.

3.5.2.1. Subjects

Ein *Subject* ist eine Menge von Klassen oder Klassenfragmenten (Mixins), die miteinander durch Aggregation, Vererbung usw. in Beziehung stehen. Ein subject ist somit ein partielles Objektmodell. In SOP/C++ entspricht dieses einem Namensraum. Die Katzenbesitzerin aus Abbildung 3 würde z.B. folgendes Objektmodell (der Einfachheit halber mit nur einer Klasse) spezifizieren:

[9] [SOP], [HO93], [CE00 S.254ff], [Vlis98]

```
namespace Frauchen {
  class Katze {
  public:
    void Fressen( char* was );
    void Schnurren( int lautstaerke );
    void Spielen();

  protected:
    char* Name;
    int   Alter;
    char* Lieblingsfutter;
  };
}
```

Das Objektmodell der Tierärztin könnte folgendermaßen aussehen:[10]

```
namespace Tierarzt {
  class Katze {
  public:
    void NimmMedizin( char* was );

  protected:
    char* Rufname;
    int Alter;
    int Blutdruck;
  };
}
```

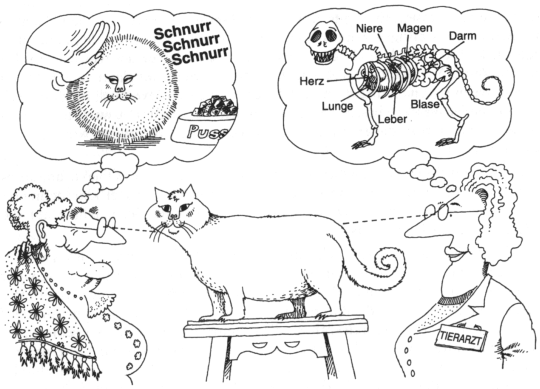

Abbildung 3 Subjektive Sichten auf ein Objekt (Abbildung aus [Boo94 S.62])

Solange die Katzenbesitzerin und die Tierärztin nicht auf denselben Instanzen arbeiten, stellt die unterschiedliche Sichtweise kein Problem dar. Ansonsten sind beide Darstellungen zusammen zu führen. Dieses wird in der SOP durch *Composition Rules* geleistet.

3.5.2.2. Composition Rules

Es gibt drei verschiedene Typen von Composition Rules: *Correspondence Rules, Combination Rules* und *Correspondence-and-Combination Rules*. (Letztere stellen jedoch nur eine Schreibabkürzung dar.)

Mit *Correspondence Rules* wird das Mapping auf Deklarationsebene beschrieben. Dabei werden gleichnamige Klassen, Methoden und Datenelemente zusammengefasst, Ausnahmen (falls Elemente mit unterschiedlichen Bezeichnern zusammengefasst werden sollen oder welche mit gleichem Bezeichner nicht zusammen gehören) lassen sich explizit formulieren.

Combination Rules dienen dem Zusammenführen der Methodendefinitionen. Im Konfliktfall (zwei gleichnamige Methoden werden zusammengeführt) kann entweder eine Definition bevorzugt werden oder der Code verschiedener Definitionen unter Angabe einer Reihenfolge kombiniert werden.

Mit *Correspondence-and-Combination Rules* kann man schließlich beide Schritte auch in einer einzelnen Regel ausrücken. Die verschiedenen Subjects des Katzenbeispiels lassen sich z.B. wie folgt zusammenfassen:

---

[10] Die deutlich andere Sicht der Tierärztin auf Katzen deutet sich auch dadurch an, dass sie für ihre Methodenbezeichner die Imperativform wählt.

```
MergeByName( kombiniert, (Frauchen, Tierarzt ) )
Equate( variable kombiniert.Katze.Name (Frauchen.Katze.Name,
Tierarzt.Katze.Rufname) )
```

`MergeByName` ist eine solche Korrespondenz-und-Kombinationsregel, die auf Deklarationsebene alle Elemente mit identischer Signatur zusammenfasst und die Definitionen in der angegebenen Reihenfolge kombiniert. Dieser Fall tritt im Beispiel nicht auf, da es keine Methoden mit identischer Signatur gibt. Zusätzlich wird mit der Korrespondenzregel `Equate` festgelegt, dass die Datenelemente `Katze::Name` und `Katze::Rufname` aus den eingehenden Subjects identisch sind und im Ergebnis als `Katze::Name` zusammengefasst werden.

3.5.3. Bewertung

SOP unterstützt insbesondere die verteilte und unabhängige Entwicklung von Applikationen, die erst später zusammengefügt werden. Durch das Konzept der Subjects wird das Prinzip der Kapselung und des Information Hiding sehr gut unterstützt. Im Gegensatz zur reinen OOP, wo immer die gesamte Schnittstelle eines Objektes zur Verfügung steht, greifen Clients bei der SOP über minimale, auf den jeweiligen Kontext reduzierte Schnittstellen zu.

SOP ist ein eher strukturorientierter Ansatz. Die unterstützten Bindungsmodi differieren zwischen den Implementierungen für verschiedene Sprachen. SOP/C++ unterstützt nur statische Bindung zur Übersetzungszeit, das neuere SOP für Java (Hyper/J) hingegen auch die Erweiterung der Objekte zur Laufzeit ohne Neukompilierung.

3.6. Composition Filters (CF)[11]

3.6.1. Einleitung

Das Konzept der *Composition Filters (CF)* wurde Anfang der 90er Jahre von MEHMET AKSIT, LODEWIJK BERGMANS u. a. entwickelt. AKSIT und BERGMANS sehen das Problem des *code tangling* zentral in der Tatsache, dass es in den gängigen OO-Sprachen nicht möglich ist den Nachrichtenaustausch zwischen mehreren Objekten bereits auf *Schnittstellenebene* zu koordinieren. Stattdessen ist es erforderlich den Koordinations-Code (z.B. für Wächter oder Synchronisation) explizit in jeder Methode einzufügen – die Koordination erfolgt also real auf der *Implementierungsebene*. Der CF-Ansatz versucht diesen Nachteil auszugleichen.

3.6.2. Grundidee: Einfügen einer zusätzlichen Schnittstellenebene

Die Grundidee ist, jedes Objekt mit einem zusätzlichen *interface-layer* zu umgeben (Abbildung 4). Alle eingehenden und ausgehenden Nachrichten müssen diese zusätzliche Ebene passieren. Auf dem interface-layer lassen sich nun *Filter* installieren, durch die

[11] [CF], [Ber94], [Aks96], [AT98], [BA01], [CE00 S.256ff]

eingehende und ausgehende Nachrichten z.B. modifiziert, verzögert oder unterdrückt werden können.

Der CF Ansatz eignet sich somit für eine Separierung derartiger Aspekte, die sich in Form von Kommunikations- und Kollaborationsmustern zwischen Objekten finden. Da im objektorientierten Paradigma Verhalten und Zustand eines Objektes *ausschließlich* über Nachrichten gesteuert werden, können damit theoretisch sämtliche Aspekte (i.t.S.) des Verhaltens modelliert werden. Dementsprechend wurden bereits Filter für eine große Anzahl typischer Aspekte vorgestellt, wie Synchronization, Real-time, Multiple views, Logging und vieles mehr [AT98].

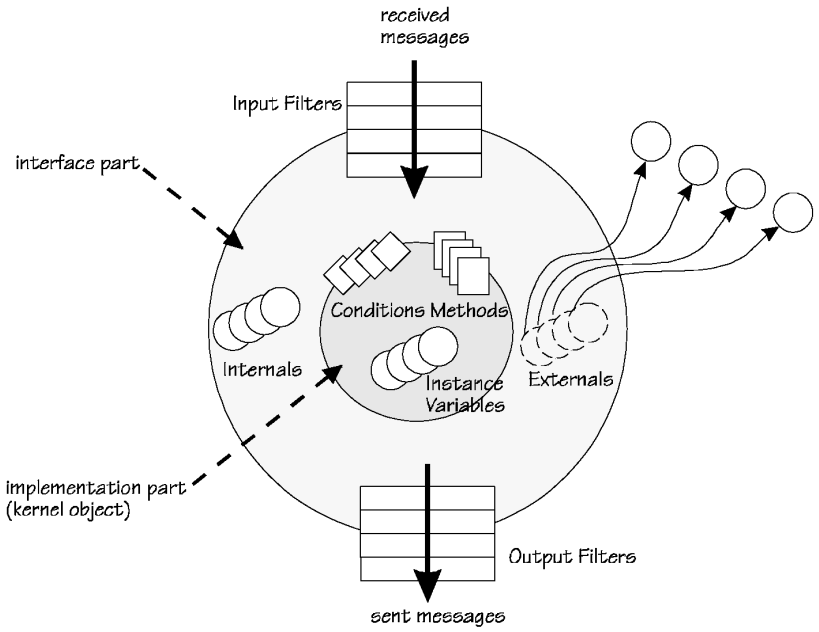

Abbildung 4 Bestandteile eines Objektes beim CF Ansatz
Das eigentliche Objekt (*kernel object)*, das z.B. in C++ oder Java geschrieben ist, wird von einem Wrapper-Objekt, dem *interface part,* umgeben. Dieses kann über eigene Instanzvariablen und Methoden verfügen sowie Referenzen auf externe Objekte enthalten. Alle eingehenden und ausgehenden Nachrichten durchlaufen nach dem Prinzip der Zuständigkeitskette beliebig viele Filter, die Nachrichten modifizieren, umleiten oder unterdrücken können. (Abbildung aus [Ber94])

3.6.3. Filtertypen und Filteranwendung

Filter werden in Form von *Filtertypen* bereitgestellt. Jeder Filtertyp modelliert dabei einen wohldefinierten, abgeschlossenen Aspekt, der orthogonal zu allen weiteren Aspekten liegen sollte. Da jedem Objekt beliebig viele Filter hinzugefügt werden können, ist es sehr einfach verschiedene Aspekte zu kombinieren.

Die konkrete Anwendung eines Filters wird beim CF-Ansatz deklarativ in Form von Prädikaten über Nachricht und Zustand beschrieben. Je nach Filtertyp wird die weitere Bearbeitung der Nachricht verändert, blockiert oder abgebrochen, ansonsten wird sie an den nächsten Filter weitergereicht (Prinzip der Zuständigkeitskette). Ein Filter vom Typ *Error*, würde z.B. die weitere Bearbeitung unterbinden und eine Ausnahme auslösen. Ein Filter vom Typ *Wait* würde die weitere Bearbeitung blockieren, bis eine bestimmte Bedingung (z.B. Mutex) erfüllt ist.

Stellen wir uns eine einfache Klasse `Stack`, mit den folgenden typischen Operationen vor:

```
push( element_t e )
pop
top : element_t
isEmpty : boolean
```

Soll diese Klasse nun um den Aspekt *Pre-Condition Fehlerbehandlung* erweitert werden, so würde man mit CF eine neue Klasse `GuardedStack` folgendermaßen beschreiben:

```
GuardedStack
  stack: Stack;
inputfilters
  guard: Error = { isEmpty ~> {top, pop} };
  execute:  Dispatch  = { true => {stack.*} };
```

`GuardedStack` bildet ein Wrapper um einen `Stack` und enthält zwei Filter. Der erste Filter ist vom Typ *Error* und legt fest, dass im Fall von `isEmpty` alle Methoden *außer* `top` und `pop` aufgerufen werden dürfen. Schlägt der erste Filter nicht an, so wird die Nachricht an den zweiten Filter weitergereicht. Dieser ist vom Typ *Dispatch* und delegiert einfach alle Nachrichten an das Kernobjekt.

Wollte man dieselbe Funktionalität mit OOP-Vererbung implementieren, so müssten die Methoden `push`, `pop` und `top` jeweils überschrieben werden und der Wächter-Code „von Hand“ hineingewebt werden – genau wie bei dem Beispiel zu Vererbungsanomalien in Abschnitt 3.1.1.

3.6.4. Bewertung

CF ist ein eher verhaltensorientierter Ansatz, mit dem bereits Lösungen für viele „typische“ Aspektprobleme und weitere klassische Probleme der OOP vorgestellt wurden [CF]. CF eignet sich insbesondere für flexible Kombination unterschiedlichster Aspekte und Fachkonzepte, das Problem der Kopplung wird auf die Reihenfolge der Filteranwendung reduziert.

Ein Problem bei CF könnte die Ausführungsgeschwindigkeit darstellen, da das Filterkonzept einen vergleichsweise hohen Aufwand zur Laufzeit mit sich bringt. Es gibt allerdings inzwischen Bemühungen, einen Teil der Bindungen schon zur Compilezeit aufzulösen.

3.7. AOP mit AspectJ[12]

3.7.1. Einleitung

AspectJ wird seit Mitte der 90er Jahre von der *Software Design Area (SDA)* Gruppe um GREGOR KICZALES am XEROX Palo Alto Research Center (XEROX PARC) entwickelt. Nach ca. 5 Jahren Entwicklungszeit wurde im November 2001 die Version 1.0 des AspectJ Entwicklungssystems freigegeben.

[12] [SDA], [Ken00], [KHH+01]

Das erklärte Ziel der SDA-Gruppe ist *„to make it possible to cleanly capture complex design structures in software implementations."* [SDA]. In diesem Zusammenhang wird eine umfangreiche Studie zur Bewertung des AOP Ansatzes mit „echten Benutzern" durchgeführt. AspectJ wird darin „nur" als die Basis für eine derartige Studie verstanden.

Was für Aspekte entwickeln die Benutzer? Welche Idiome, Muster und Stile entstehen? Wie effektiv können sie mit dem Prinzip der Querschnittseigenschaften umgehen? Diese und andere Fragen sollen in der Studie beantwortet werden. [KHH+01].

3.7.2. Grundlegende Entwurfsentscheidungen

Ziel der Entwicklung von AspectJ war es eine Sprache zu schaffen, die von der Programmiergemeinde gut aufgenommen und tatsächlich benutzt wird. Ein wesentliches Augenmerk lag daher auf der Kompatibilität zu Java. AspectJ ist eine echte Obermenge von Java, jedes gültige Java-Programm ist auch ein gültiges AspectJ Programm. Ferner ist der erzeugte Bytecode auf jeder Java2 JVM ablauffähig.

Der Kompatibilitätsaspekt erstreckt sich aber auch auf das, was in [KHH+01] *Programmierer-Kompatibilität* genannt wird. Die Benutzer sollen AspectJ als eine *„natürliche Erweiterung"* von Java empfinden. Das betrifft nicht nur Syntax und Semantik neuer Elemente, sondern auch die Gewichtung zwischen imperativen und deklarativen Konstrukten. AspectJ ist deutlich weniger deklarativ als vergleichbare AOP-Ansätze.

3.7.3. Zentrale Begriffe

AspectJ unterstützt zwei grundlegende Konzepte zur Implementierung von Querschnittskonzepten:

- *Static Crosscutting* ermöglicht es Typen (Klassen) um *neue* Operationen und Zustandsvariablen zu erweitern, bietet also eine Art generisches Mixin-Konstrukt, mit dem die Signatur von Typen statisch erweitert werden kann. In anderen Sprachen, wie z.B. C++, kann dieses mit Mehrfachvererbung oder parametrisierbaren Oberklassen erreicht werden. Static Crosscutting ist daher eigentlich nichts Neues und wird an dieser Stelle nicht weiter ausgeführt.
- *Dynamic Crosscutting* ermöglicht es Aspektcode an wohldefinierten Stellen zu injizieren. Die Injektionspunkte werden dabei deklarativ durch *Join-Points* beschrieben, der eigentliche Aspectcode wird als Java-Code in *Advices* hinterlegt. Ein *Aspect* fasst schließlich Join-Points, Advices und optional zusätzliche Variablen und Methoden in Form eines Moduls, vergleichbar einer Klasse, zusammen.

3.7.3.1. Join-Points und Pointcut-Designators

Tabelle 1 zeigt die verschiedenen Typen von Join-Points, die in AspectJ zur Verfügung stehen. Konkrete Joint Points werden festgelegt durch *Pointcut-Designators*. Ein Pointcut-Designator ist ein Prädikat, das eine Menge von Join-Points beschreibt, also eine Menge von „passenden Code-Positionen" an denen der Aspektcode eingefügt werden soll. Dafür

steht eine Reihe primitiver Pointcut-Designators zur Verfügung (Tabelle 2 zeigt eine Auswahl), die dann mit logischen Operatoren zu komplexen Ausdrücken verkettet und benannt werden können. Der Designator

```
receptions( void Point.setX( int ) )
```

passt auf alle Methoden mit dem angegebenen Namen und der angegebenen Signatur, was intuitiv bedeutet, dass der Aspectcode jedes Mal ausgeführt wird, wenn ein Objekt der Klasse `Point` die Nachricht `setX` erhält.

Art des Joint-Points	Unterbricht die Ausführung, wenn
method call constructor call	eine Methode (ein Konstruktor) aufgerufen wird. Die Advice-Ausführung erfolgt im Kontext des *aufrufenden Objektes* unmittelbar vor dem Aufruf (`before`) bzw. nach der Rückkehr (`after`).
method call reception constructor call reception	eine Methode (ein Konstruktor) aufgerufen wird. Die Advice-Ausführung erfolgt im Kontext des *aufgerufenen Objektes* unmittelbar vor dem Betreten (`before`) bzw. nach dem Verlassen (`after`) der Methode.
method execution constructor execution	eine Methode (ein Konstruktor) aufgerufen wird. Die Advice-Ausführung erfolgt im Kontext der *aufgerufenen Methode* unmittelbar vor (`before`) bzw. nach (`after`) der Ausführung ihres Methodenkörpers.
field set field get	ein Datenfeld geschrieben/gelesen wird. Die Advice-Ausführung erfolgt im Kontext des umgebenden Objektes unmittelbar vor (`before`) bzw. nach (`after`) dem Zugriff.
exception handler execution	ein Ausnahmebehandler ausgeführt wird. Die Advice-Ausführung erfolgt im Kontext der umgebenden Methode unmittelbar vor (`before`) dem Betreten bzw. nach (`after`) dem Verlassen des Ausnahmebehandlers.

Tabelle 1 Die wichtigsten Join-Points in AspectJ
An den durch einen Join-Point beschriebenen Stellen erfolgt die Ausführung eines Advices entweder vorher (`before`), nachher (`after`) oder vorher und nachher (`around`).
(Tabelle geändert nach [KHH+01])

Grundlegende Pointcut-Designators
`calls` (*signature*) `receptions` (*signature*) `execution` (*signature*) `gets` (*signature*) [*val*] `sets` (*signature*) [*oldVal*] [*newVal*] `handles` (*ThrowableTypeName*) Pointcut-Designators für die in Tabelle 1 vorgestellten Arten von Join-Points. Die Join-Points passen für alle Methoden/Datenelemente, deren Name und Signatur auf den Ausdruck *signature* passt bzw. für alle Ausnahmebehandler, die Ausnahmen des Typs *ThrowableTypeName* behandeln.
`instanceof` (*ObjectTypeName*) `within` (*ClassName*) Join-Points passen, wenn der dynamische (`instanceof`) bzw. statischen (`within`) Typ der gerade aktiven Objektinstanz dem übergebenen Typbezeichner entspricht.
`cflow` (*pointcut_designator*) Join-Points passen, wenn der Aufruf innerhalb des Ausführungskontextes (Aufrufstapel) des angegebenen Pointcut-Designator erfolgt.
...

Tabelle 2 Einige Pointcut-Desginators
Dargestellt ist eine Auswahl der grundlegenden Pointcut-Designators.
(nach [KHH+01])

Der Designator

```
!within( Line ) &&
( receptions( void *.setX( int ) ) ||
receptions( void *.setY( int ) ) )
```

passt hingegen auf entsprechenden `setX` und `setY` Methoden aller Klassen, mit Ausnahme der Klasse `Line`. Schließlich können Designators noch benannt werden um längere Ausdrücke abzukürzen:

```
pointcut modified():
  receptions( void *.setX( int ) ) ||
  receptions( void *.setY( int ) )
```

3.7.3.2. Advices

Mit *Advices* wird beschrieben, welcher Code zu genau welchem Zeitpunkt in den Join-Points ausgeführt werden soll. AspectJ bietet *before*, *after* und *around* Advices an. Die Advice Deklaration

```
after() : modified() {
  dirty = true;
}
```

legt beispielsweise fest, dass immer *nach* dem Aufruf einer der Methoden `void setX(int)` oder `void setY(int)` das Flag `dirty` auf wahr gesetzt werden soll. Diese Information könnte z.B. zur Optimierung der Bildschirmausgabe verwendet werden. Das Datenelement `dirty` muss dabei im Sichtbarkeitsbereich in Form einer Member- oder Klassenvariablen zur Verfügung stehen, wobei Aspekte ebenfalls derartige Elemente einfügen können.

3.7.3.3. Aspects

Ein *Aspect* ist, ähnlich einer Klasse, ein eigenständiges Modul, das eine quer schneidende Implementierung beschreibt. Aspekte enthalten üblicherweise Pointcuts und Advices, aber auch weitere Methoden und Datenelemente können definiert werden:

```
aspect ModifiedTracking {

  static boolean dirty = false;


  pointcut modified():
    receptions( void *.setX( int ) ) ||
    receptions( void *.setY( int ) )

  after() : modified() {
    dirty = true;
  }
}
```

Um Aspekt-Bibliotheken zu ermöglichen, bietet AspectJ einen einfachen Vererbungsmechanismus für Aspekte an. Es ist möglich Aspekte als *abstract* zu deklarieren um parametrisierbare Pointcuts oder Advices zu erhalten. Das abschließende Beispiel demonstriert dieses anhand eines einfachen Tracers. Dieser verwendet ein spezielles Datenelement `thisJoinPoint`, über das u.a. der genaue Aufruf-Kontext ermittelt werden kann:

```
abstract aspect Tracing {

 abstract pointcut tracePoints();

 before() : tracePoints() {
   traceMethodSig( „Entering“, thisJoinPoint );
 }
 after() : tracePoints() {
   traceMethodSig( „Leaving“, thisJoinPoint );
 }

 void traceMethodSig( String s, JoinPoint jp ) {
    ... // Code zum Ausgeben der Signatur
 }
}
```

Verwendung für die Klasse `point`:

```
aspect TracePointSetXY extends Tracing {

  pointcut tracePoints() :
    receptions( void Point.setX( int ) ) ||
    receptions( void Point.setY( int ) )
}
```

3.7.4. Bewertung

AspectJ verfolgt einen pragmatischen und dennoch leistungsfähigen Ansatz zur AOP. Die Integration in die Sprache darf als gelungen bezeichnet werden und AspectJ ist heute das wohl beliebteste Werkzeug zur AOP.

AspectJ ist ein insgesamt eher verhaltensorientierter Ansatz, enthält mit dem Konzept des Static Crosscutting aber auch strukturorientierte Elemente. Puristen werden bemängeln, dass AspectJ (naturgemäß) auf die Sprache Java beschränkt ist. Ferner ist es bislang noch nicht möglich Aspekte auch zur Laufzeit hinzuzufügen und zu entfernen. Eine weitere prinzipielle Schwierigkeit stellt auch hier die Integration in elementare Werkzeuge, wie den Debugger, dar. Hieran wird jedoch aktiv gearbeitet.

Durch die weitgehende Bindung der Aspekte zur Übersetzungszeit konnte es erreicht werden AOP quasi ohne Laufzeiteinbußen zu nutzen – was gerade für das eher träge Java ein wichtiger Punkt ist.

3.8. Intentional Programming (IP)[13]

3.8.1. Einleitung

Intentional Programming (IP) ist der Name für ein revolutionär wirkendes System zur Programmierung und Metaprogrammierung, das seit Anfang 1991 von der Arbeitsgruppe um CHARLES SIMONYI bei Microsoft Research entwickelt wird. Der Bootstrap erfolgte 1995, 1999 bekam IP den Status eines Produktentwicklungsprojektes, im Frühjahr 2001 wurde die Arbeit an IP offiziell eingestellt.

---

[13] [CE00 S.503ff], [Röd99], [CER00], [Sim95], [Sim96]

Über die Gründe von Microsoft das interessante (und laut Augenzeugen [CE00, Röd99] auch bereits sehr weit fortgeschrittene) Projekt einzustellen, gibt es bislang nur Vermutungen. Der offizielle Grund ist die Beschränkung der maximalen Laufzeit von Projekten bei Microsoft Research auf 10 Jahre. Es darf jedoch angenommen werden, dass einige der Ziele schlichtweg zu ehrgeizig waren und Lösungen für bekannt schwierige Probleme voraussetzten, die auf absehbare Zeit nicht zu erwarten sind. Ein weiterer Grund mag sein, dass IP nicht so recht in die gerade erst groß aufgelegte .NET Strategie hineinpasst.

Nichtsdestotrotz ist IP eine interessante Sache – auch wenn aus dem Projekt nun eher eine Studie geworden ist.[14] Es zeigt, wie weit man den Bausteingedanken und die Idee der Trennung der Belange treiben kann.

3.8.2. Grundlegender Aufbau

Das IP System besteht aus einer generischen Entwicklungsumgebung (mit Editor, Compiler, Debugger usw.), abstrahiert jedoch vollständig von Programmiersprachen und Sprachkonstrukten. Sprachen oder auch nur einzelne Konstrukte werden in Form von *Aktiven Bibliotheken (active libraries)* eingebunden. Eine solche Bibliothek erweitert die Entwicklungsumgebung um Funktionen sowohl zur Darstellung der Konstrukte in Editor und Debugger als auch zur Vereinfachung auf ein generisches, übersetzbares Format.

Aktive Bibliotheken lassen sich beliebig kombinieren und mit geringem Aufwand selber erstellen. Es ist also möglich Sprachkonstrukte verschiedenster Sprachen gemeinsam zu verwenden und das System um domänenspezifische Sprachen zu erweitern. Editor und Debugger sind dabei nicht auf textuelle Darstellung beschränkt, so dass sich auch graphische Sprachen und domänenspezifische Notationsformen verwenden lassen.

3.8.3. Die Konzepte hinter IP

3.8.3.1. Interne Repräsentation des Programms

Im IP System besteht ein Programm nicht mehr aus ASCII Quelltexten, sondern aus so genanntem *Aktiven Quelltext (active source)*. Der „Quelltext" wird bei der Eingabe direkt als abstrakter Syntaxgraph erzeugt und in einem Repository gespeichert, damit kann auf einen separaten Parser verzichtet werden (siehe auch Abbildung 5). Die Repräsentation als Graph bietet eine ganze Reihe von Vorteilen, einige davon sind:

---

[14] Eventuell ist das Projekt doch noch nicht am Ende: Laut einer Meldung der *Seattle Times* vom 17. September 2002 [Dud02] hat CHARLES SIMONYI Microsoft verlassen und zusammen mit GREGOR KICZALES (!) am 8. August 2002 die Firma *Intentional Software* gegründet. Aufgrund seines exzellenten Verhältnisses zu Microsoft-Gründer BILL GATES habe SIMONYI außerdem das Recht erhalten, die Erkenntnisse aus dem IP Projekt mitzunehmen, umgekehrt wurde Microsoft ein exklusives Vorkaufsrecht für zukünftige bei Intentional Software entwickelte Technologien eingeräumt.

- Schon heute haben Sprachen wie C++ das Problem, dass sie kaum noch erweiterbar sind ohne dabei unparsbar zu werden. Erst durch den Verzicht auf einen dedizierten Parser ist es möglich Konstrukte unterschiedlichster Sprachen gemischt zu verwenden und Sprachen einfach zu erweitern.
- Alle Elemente der Entwicklungsumgebung interagieren direkt mit dem als Graph dargestellten Programm. Mit Hilfe von *Editing Enzyms* können Anfragen und Transformationen auf dem Graphen durchgeführt werden. Die Knoten lassen sich mit beliebigen Metadaten attributieren. Dadurch lassen sich z.B. Entwurfsinformationen im Graphen ablegen und eine effiziente Versionskontrolle realisieren.

3.8.3.2. Aktive Bibliotheken und Intentionen

Die dem Programmierer zur Verfügung stehenden Abstraktionen und Sprachkonstrukte heißen *Intentionen*. Intentionen werden zusammengefasst in *Aktiven Bibliotheken*. So „lernt" die Entwicklungsumgebung z.B. die Sprache C++, indem eine Aktive Bibliothek geladen wird, in der die Intentionen (Sprachkonstrukte) von C++ definiert sind.

Die Elemente des Aktiven Quelltextes sind *Instanzen der Intentionen*. Intentionen sind damit Metaobjekte, deren Methoden von der Entwicklungsumgebung aufgerufen werden um ein Sprachkonstrukt darzustellen, zu editieren oder zu kompilieren.

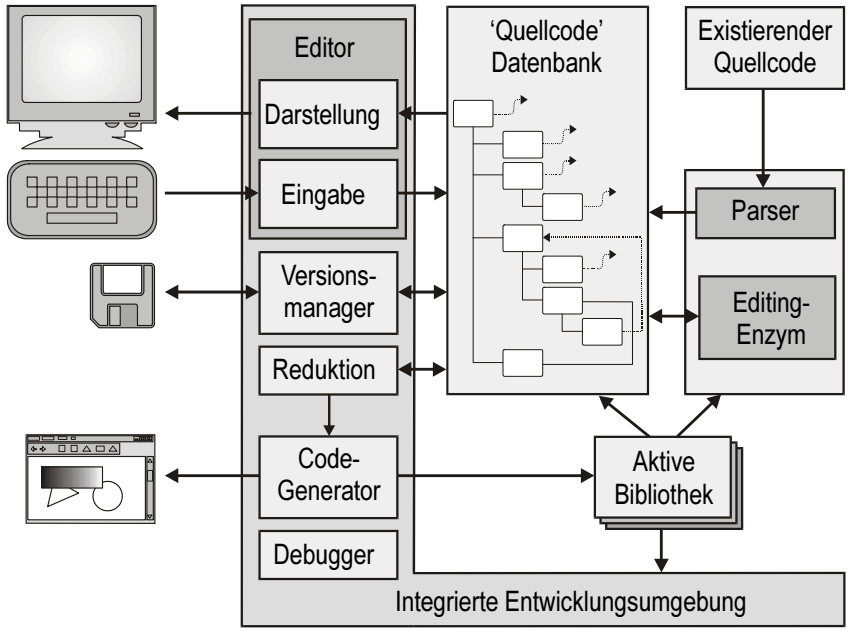

Abbildung 5 Architektur des IP Systems
Zentrales Element sind die *Aktiven Bibliotheken*, in denen die Sprachkonstrukte (*Intentionen*) definiert sind. Das eigentliche Programm besteht aus Instanzen dieser Intentionen (*Aktiver Quelltext*) und wird direkt als abstrakter Syntaxbaum eingegeben, verwaltet und gespeichert. Die Intentionen stellen auch die Methoden zum Darstellen, Editieren und Reduzieren der Sprachkonstrukte bereit. Klassische Parser werden nur noch zum Import existierender Quellen verwendet.
(Abbildung aus [CER00])

3.8.3.3. Reduktion

Wie in den bekannten Programmiersprachen lassen sich Intentionen hierarchisch zusammensetzen. Eine Intention „Prozedur“ besteht z.B. aus einer Sequenz von „Statement“ Intentionen. Das Übersetzen eines Programms erfolgt iterativ in einem *Reduktion* genannten Prozess. Dafür ruft die Entwicklungsumgebung entsprechende Metaprogramme auf (Methoden der Intentionen), deren Aufgabe es ist, das jeweilige Konstrukt in eine primitivere Darstellung zu transformieren. Der Prozess setzt sich fort, bis eine genügend primitive Darstellung erreicht ist, die über ein herkömmliches Compiler-Backend schließlich in ausführbarem Code mündet.

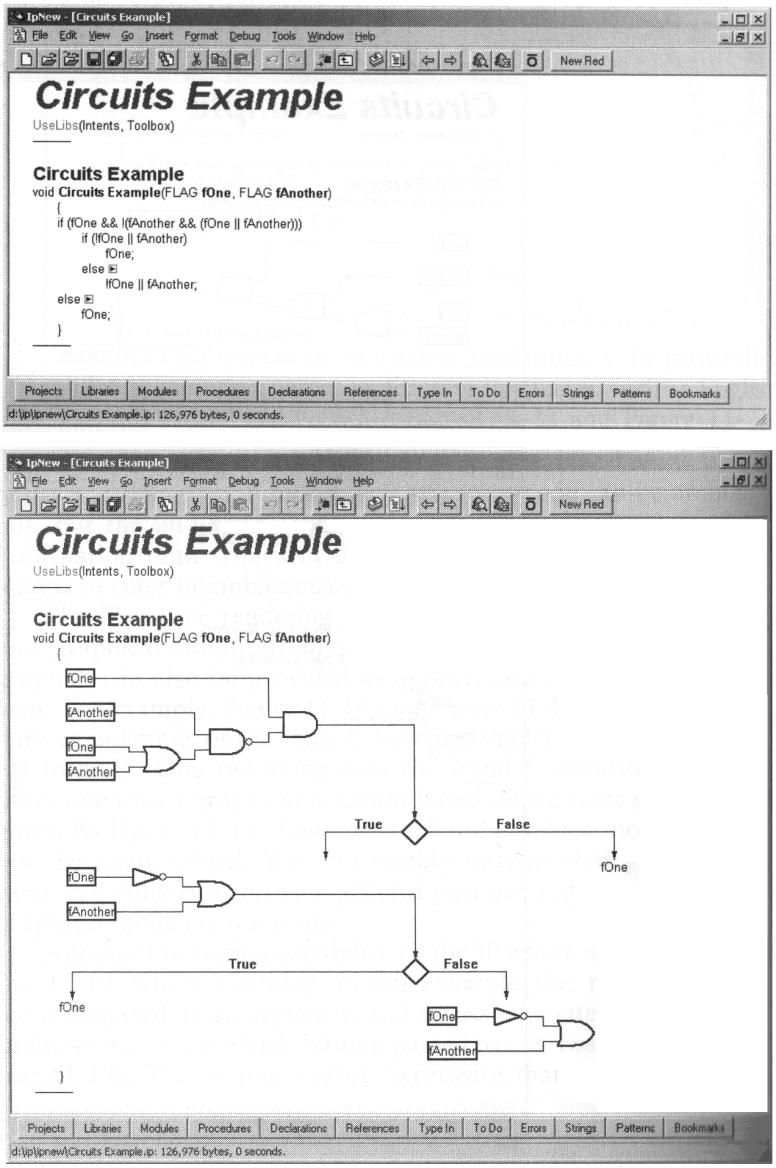

Abbildung 6 Verschiedene Sichten auf ein Programmfragment
Intentionen visualisieren sich selber, daher ist es sehr einfach möglich unterschiedliche Sichten auf ein Programm zu ermöglichen.
(Abbildungen aus [CE00 S.533f])

Insgesamt wirkt das Konzept, Abstraktionen durch den Aufruf entsprechender Methoden sich selber darstellen, editieren und übersetzen zu lassen unter verschiedensten Gesichtspunkten äußerst attraktiv:

- *Visualisierung*: Da Intentionen sich selber visualisieren, ist es einfach möglich verschiedene Sichten, Darstellungsarten und Detaillevel zu ermöglichen (Abbildung 6). Domänenspezifische Abstraktionen können so „wie gewohnt" dargestellt und bearbeitet werden. Eine Bibliothek zur Matrizenalgebra könnte z.B. die gewohnte mathematische Darstellung verwenden.
- *Optimierung:* Dadurch, dass ein Teil des Übersetzungsvorgangs in die Intentionen verlagert wird, ist es möglich domänenspezifisches Fachwissen für Optimierungen zu verwenden. Die oben erwähnte Matrizenbibliothek könnte durch ausgefeilte Strategien (eliminieren temporärer Zwischenwerte, Caching, Kompression schwach besetzter Matrizen usw.) hochgradig optimierten Code erzeugen, der handoptimiertem C oder Assemblercode in nichts nachsteht. Die bislang häufig unüberwindbar wirkende Kluft zwischen hohem Abstraktionsniveau und optimaler Performance wird damit geschlossen.
- *Integration der Entwurfsdokumente*: Durch Aktive Bibliotheken ist es möglich auch die Dokumente der früheren Phasen direkt in das System aufzunehmen. So könnten z.B. UML Diagramme, Petrinetze und andere jeweils geeignete Sprachen zur Modellierung verwendet und automatisch mit der Implementierung abgeglichen werden. Damit bleiben sämtliche Entwurfsinformationen erhalten und sind immer auf dem aktuellen Stand, wodurch das Verständnis, Reengineering und Refactoring drastisch erleichtert werden.

3.8.4. Bewertung

Da das IP-Projekt zunächst eingestellt wurde, halte ich eine Bewertung an dieser Stelle für nur wenig sinnvoll. Über die Gründe von Microsoft das Projekt einzustellen kann, wie gesagt, nur spekuliert werden. Nichtsdestotrotz bietet das IP-Modell viele interessante Ideen.

3.9. Fazit

Nach der Objektorientierung hat die Aspektorientierung gute Chancen den nächsten Paradigmenwechsel in der Softwareentwicklung einzuleiten. Neue Anwendungen müssen heute üblicherweise ein hohes Maß an Querschnittseigenschaften wie Echtzeitfähigkeit, Transaktionsorientierung, Fehlertoleranz usw. erfüllen und die AOP bietet Konzepte, um gerade derartige Anforderungen nach dem fundamentalen Prinzip der Trennung der Belange zu bearbeiten. Die AOP könnte damit auch die vielfach zitierten (und vergleichsweise selten gehaltenen) großen Versprechen der Objektorientierung erfüllen: Wiederverwendbarkeit, Wartbarkeit und Verständlichkeit.

Vergleicht man die bekannten Ansätze zur AOP, so können diese grob in zwei Klassen eingeteilt werden. Zum einen die eher *strukturorientierten Ansätze* wie Adaptive Pro-

gramming und Subject-Oriented Programming, die es ermöglichen Klassen und Klassenbäume statisch und nicht-invasiv zu verknüpfen. Zum anderen die eher *verhaltensorientierten Ansätze* wie Composition Filter und AspectJ, die sich insbesondere zur Modellierung der typischen Querschnittseigenschaften eignen. All diese Ansätze bieten zusätzliche Dimensionen für die Dekomposition, alle haben in ihrer Verwendbarkeit aber auch klare theoretische und praktische Grenzen. So fällt auf, dass sie sich nahezu ausschließlich auf der Implementationsebene bewegen – offensichtlich bereiten hier quer schneidende Konzepte die größten Probleme.

Nichtsdestotrotz ist eine Übertragung des Integrationsgedankens quer schneidender Konzepte auf den gesamten Softwareentwicklungsprozess wünschenswert. Der im nächsten Kapitel beschriebene Hyperspace-Ansatz nimmt dieses für sich in Anspruch.

Kapitel 4

Der Hyperspace-Ansatz[15]

Der *Hyperspace-Ansatz* wird seit Ende der 90er Jahre von WILLIAM HARRISON, HAROLD OSSHER und PERI TARR am T.J. Watson IBM Research Center entwickelt.

Technisch basieren Hyperspaces und insbesondere das in diesem Kapitel ebenfalls vorgestellte Hyper/J auf den Ideen des *Subject-Oriented Programming (SOP)* (Abschnitt 3.5). Während sich SOP jedoch ausschließlich mit der Integration von verschiedenen Sichten auf der *Implementierungsebene* beschäftigte, ist der Hyperspace-Ansatz nun bemüht diesen Integrationsgrad über *alle Phasen des Softwareentwicklungszyklus* und *alle dabei verwendeten Sprachen* zu erreichen. Ziel ist es, echtes *multidimensionales Trennen der Belange* (Kapitel 2.1.4) zu ermöglichen und damit zu überwinden, was OSSHER und TARR als die *„Tyrannei der dominanten Dekomposition“* bezeichnen [TOHS99].

In diesem Kapitel wird nun der Hyperspace-Ansatz im Detail vorgestellt. Ich beginne dabei mit der Erläuterung des theoretischen Modells und der verwendeten Terminologie. In Abschnitt 4.2 wird dann das Hyper/J-System vorgestellt, das eine erste Realisierung der Hyperspace-Idee beschränkt auf die Programmiersprache Java bereitstellt. Sowohl das Hyperspace-Modell als auch die Verwendung von Hyper/J kommen schließlich in Abschnitt 4.3 in einem konkreten Beispiel zur Anwendung.

4.1. Das Modell

4.1.1. Grundbegriffe

Software besteht aus *Artefakten*, wie Anforderungslisten, Entwurfsdokumenten, Quelltext oder Testplänen. Artfakte sind beschreibende Dokumente, abgefasst in einer jeweils geeigneten Sprache, hier Artefaktsprache genannt. Sie bestehen aus *Grundelementen (Units)*. Grundelemente sind *Instanzen* der syntaktischen Konzepte einer Artefaktsprache. Im objektorientierten Entwurf wären z.B. Attribute, Methoden, Klassen und Bezie-

[15] [OT99], [OT00], [TOHS99], [HS]

hungen derartige syntaktische Konzepte. Ein Artefakt, wie z.B. ein UML Klassendiagramm oder eine Java Quelldatei, besteht aus konkreten Instanzen dieser Konzepte.

Ein *Belangraum (concern space)* enthält alle Grundelemente für eine bestimmte Art von Software, wie z.B. eine Komponentenbibliothek, eine Produktlinie, eine Softwarefamilie oder ein Softwaresystem. Die Aufgabe des Belangraumes ist es, die Grundelemente der Software so zu organisieren, dass sich alle wichtigen Belange separieren lassen, die Beziehungen zwischen Belangen deutlich werden und das Softwaresystem anhand der gewünschten Belange aus den einzelnen Grundelementen zusammengesetzt werden kann. Der Belangraum adressiert somit die drei wesentliche Aufgaben, die mit der Anwendung des Prinzips der Trennung der Belange einhergehen: *Identifizieren, Kapseln* und *Integrieren*:

- *Identifizieren* beschreibt den Prozess der Auswahl und Benennung der grundlegenden Belange des Systems, sowie der Zuordnung von Grundelementen zu diesen Belangen.
- *Kapseln* beschreibt den Vorgang, die Grundelemente eines einzelnen Belangs so zusammen zu fassen, dass dieser als first-class Entität unabhängig von anderen Belangen bearbeitet und verwendet werden kann.
- Beim *Integrieren* werden schließlich die gekapselten Belange zusammengefügt um ein Softwaresystem zu erstellen, dass mehrere unterschiedliche Belange behandelt.

Ein *Hyperspace* ist ein solcher Belangraum. Im Folgenden wird nun gezeigt, wie ein Hypespace aufgebaut ist und wie im Hyperspace-Modell das Identifizieren, Kapseln und Integrieren der Belange realisiert wird.

4.1.2. Hyperspaces

Ein Hyperspace besteht aus den folgenden Elementen:

- Eine Menge ***U*** von *Grundelementen*.
- Eine *Belangmatrix* ***M***, in der die Grundelemente den einzelnen Belangen zugeordnet werden. Die Belangmatrix unterstützt das Identifizieren und Kapseln von Belangen.
- Eine Menge ***HM*** von *Hypermodules*. Hypermodules dienen dem Integrieren von Belangen.

4.1.2.1. Identifizieren von Belangen: Die Belangmatrix

In einem Hyperspace werden die Grundelemente aus ***U*** in einer multidimensionalen Belangmatrix ***M*** angeordnet. Jede Achse (Dimension) der Matrix repräsentiert eine *Art* von Belangen und jeder Punkt auf einer Achse einen *konkreten Belang* dieser Art. Vereinfacht kann man sich die Punkte auf einer Achse als das Ergebnis einer Dekompositionstechnik vorstellen. So könnten sich z.B. auf einer Achse Klassen befinden, während auf einer anderen Achse Features aufgetragen werden.

Die Koordinaten eines Grundelements aus $\boldsymbol{U}$ bestimmen eindeutig, welcher Belang in jeder Dimension von dem Grundelement adressiert wird. Jede Dimension $\boldsymbol{d}$ enthält zusätzlich einen *Nullbelang* $\boldsymbol{N_d}$, dem alle Grundelemente zugewiesen werden, die keinen anderen Belang aus $\boldsymbol{d}$ adressieren. $\boldsymbol{N_d}$ ist also die Menge der Grundelemente, die von Änderungen entlang dieser Dimension (z.B. bei der Programmevolution) nicht betroffen sind. Damit adressiert jedes Grundelement *genau einen* Belang in jeder Dimension und jede Dimension *partitioniert* die Menge der Grundelemente.

Definition 1 Belangmatrix M

Seien $\boldsymbol{U}$, $\boldsymbol{C}$ und $\boldsymbol{D}$ disjunkte Mengen. $\boldsymbol{U}$ sei eine Menge von Grundelementen, $\boldsymbol{C}$ eine Menge von Belangen und $\boldsymbol{D} = \{\boldsymbol{d_1}, \ldots, \boldsymbol{d_n}\}$ eine Menge von Belangdimensionen.

Die Matrix $\boldsymbol{M} : \boldsymbol{U} \rightarrow \boldsymbol{d_1} \times \ldots \times \boldsymbol{d_n}$ ist eine ***Belangmatrix***, genau dann wenn gilt:

i) Jeder Belang aus $\boldsymbol{C}$ liegt in genau einer Dimension aus $\boldsymbol{D}$:
$\forall\, \boldsymbol{c} \in \boldsymbol{C} : \exists_1\, \boldsymbol{d} \in \boldsymbol{D} : \boldsymbol{c} \in \boldsymbol{d}$

ii) Jede Dimension aus $\boldsymbol{D}$ verfügt über jeweils genau ein zusätzliches Element $\boldsymbol{N_d}$, das ***Nullbelang*** der Dimension genannt wird:
$\forall\, \boldsymbol{d} \in \boldsymbol{D} : \exists_1\, \boldsymbol{N_d} \in \boldsymbol{d}, \boldsymbol{N_d} \notin \boldsymbol{C}, \boldsymbol{N_d} \notin \bigcup_{\boldsymbol{i}=1}^{\boldsymbol{n}} \boldsymbol{d_i} \setminus \boldsymbol{d}$

iii) Jede Dimension aus $\boldsymbol{D}$ enthält neben dem Nullelement nur Elemente aus $\boldsymbol{C}$:
$\bigcup_{\boldsymbol{i}=1}^{\boldsymbol{n}} \boldsymbol{d_i} \setminus \{\boldsymbol{N_{d_i}}\} = \boldsymbol{C}$

Definition 2 Grundelementmenge eines Belangs

Sei $\boldsymbol{M}$ eine Belangmatrix, $\boldsymbol{d_i}$ eine Belangdimension aus $\boldsymbol{M}$ (mit $1 \le \boldsymbol{i} \le \boldsymbol{n}$) und $\boldsymbol{c} \in \boldsymbol{d_i}$ ein Belang.

Die Menge $\mathcal{U}(\boldsymbol{c})$ aller durch $\boldsymbol{M}$ auf $\boldsymbol{c}$ abgebildeten Grundelemente heißt ***Grundelementmenge des Belangs*** $\boldsymbol{c}$ und ist definiert durch:[16]

$\mathcal{U}(\boldsymbol{c}) := \{\boldsymbol{u} \in \boldsymbol{U} \mid \boldsymbol{c} = \Pi_i(\mathrm{M}(\boldsymbol{u}))\}$

4.1.2.2. Kapseln und Integrieren von Belangen

Die Belangmatrix erlaubt das simultane Erkennen von Belangen anhand verschiedener Dimensionen. Sie bietet jedoch keine *Kapselung* der Belange. Belange lassen sich nur kapseln, wenn die Artefaktsprachen der entsprechenden Grundelemente dafür explizite Konstrukte bereitstellen, üblicherweise in einer Form von *Modulen*. Ein Modul ist eine abgeschlossene Menge von Grundelementen mit definierter Schnittstelle. Bei der überwiegenden Anzahl der heute verwendeten Dekompositionstechniken müssen die Grundelemente dafür jedoch in *ein und derselben* Artefaktsprache beschrieben sein. OSSHER und TARR sehen darin das Hauptproblem der „Tyrannei der dominanten Dekomposition“ [OT99].

4.1.2.3. Hyperslices

Beim Hyperspace-Ansatz wird Kapselung durch *Hyperslices* und *Hypermodules* realisiert. Ein Hyperslice ist eine Menge von Grundelementen. Vergleichbar mit Modulen, dienen

[16] Der *Projektionsoperator* Π_i liefert dabei die die $\boldsymbol{i}$-te Komponente eines $\boldsymbol{n}$-Tupels.

Hyperslices der Kapselung von Belangen. Sie sind jedoch nicht an eine bestimmte Artefaktsprache gebunden und können daher beliebige Grundelemente beinhalten.

Hyperslices sind die Bausteine von Software. Jedes Grundelement gehört zu wenigstens einem Hyperslice. Hyperslices können gruppiert werden zu Hypermodules. Ein Hypermodule besteht aus einer Menge von Hyperslices und einer Beschreibung, wie diese Hyperslices zusammen hängen und wiederum zu einem Hyperslice integriert werden. Ein Softwaresystem ist schließlich ein Hypermodule, das die Vollständigkeitseigenschaft erfüllt. Auf Hypermodules wird später noch genauer eingegangen.

4.1.2.4. Deklarative Vollständigkeit

Grundelemente stehen üblicherweise auf verschiedenste Art und Weise in Beziehung miteinander. Eine Methode ruft z.B. eine andere Methode auf oder greift auf eine Instanzvariable zu, eine Klasse aggregiert eine andere Klasse etc. Auf diese Weise entstehen Abhängigkeiten zwischen den Grundelementen. Finden sich derartige Abhängigkeiten zwischen Grundelementen verschiedener Belange, so ist das Ergebnis eine hohe Kopplung, welche die Vorteile der Kapselung zunichte macht. Nahezu alle Sprachen erlauben oder fordern daher, dass Module nicht direkt auf andere Module verweisen, sondern stattdessen explizit alle Grundelemente *deklarieren*, die extern bereitgestellt werden müssen. Die Deklarationen müssen dann erst beim eigentlichen Integrationsprozess (häufig beim Linken, spätestens jedoch zur Laufzeit) an konkrete *Definitionen gebunden* werden.

Ein Modul, das alle benötigten externen Grundelemente deklariert, heißt *deklarativ vollständig (declarative complete)*. Deklarative Vollständigkeit wird entsprechend auch für Hyperslices gefordert: Falls ein Grundelement $\boldsymbol{u}$ auf ein externes Grundelement $\boldsymbol{v}$ verweist, so wird eine entsprechende Deklaration in Form eines zusätzlichen Grundelements $\boldsymbol{v_{decl}}$ aufgenommen.

Definition 3 Deklaratives Grundelement

Seien $\boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{U}$ Grundelemente.

i) Das Grundelement $\boldsymbol{u_{decl}} \in \boldsymbol{U}$ ist eine Deklaration von $\boldsymbol{u}$ und wird ein ***deklaratives Grundelement*** genannt.

ii) $\boldsymbol{decl}(\boldsymbol{v}) = \boldsymbol{true} \Leftrightarrow$ $\boldsymbol{v}$ ist ein deklaratives Grundelement für ein Grundelement $\boldsymbol{u}$ ($\boldsymbol{v}$ hat die Form $\boldsymbol{u_{decl}}$).

Definition 4 Verwendungsbeziehung zwischen Grundelementen

Seien $\boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{U}$ Grundelemente.

$\boldsymbol{uses}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{true} \Leftrightarrow$ $\boldsymbol{u}$ enthält eine Referenz auf $\boldsymbol{v}$ (verwendet $\boldsymbol{v}$).

Definition 5 Deklarative Vollständigkeit, Hyperslice:

Sei $\boldsymbol{hs} \in \boldsymbol{C}$ ein Belang. Dann gilt:

$\forall \boldsymbol{u} \in \mathcal{U}(\boldsymbol{hs}) \; \forall \boldsymbol{v} \in \boldsymbol{U} \setminus \mathcal{U}(\boldsymbol{hs}) : \boldsymbol{uses}(\boldsymbol{u}, \boldsymbol{v}) \Rightarrow \boldsymbol{v_{decl}} \in \mathcal{U}(\boldsymbol{hs})$

$\Leftrightarrow$ $\boldsymbol{hs}$ ist ***deklarativ vollständig***

$\Leftrightarrow$ $\boldsymbol{hs}$ ist ein ***Hyperslice***

4.1.2.5. Hyperslice-Dimensionen

Hyperslices sind Belange, sie unterscheiden sich von herkömmlichen Belangen nur dadurch, dass ein Hyperslice deklarativ vollständig sein muss. Als spezielle Belange werden sie in eigenen Dimensionen, den *Hyperslice-Dimensionen (hyperslice dimensions)*, in die Belangmatrix aufgenommen. Somit überlappen Hyperslices andere Belange. Dieses ist beabsichtigt, es ist Sinn und Zweck von Hyperslices Belange zu kapseln.

Definition 6 Implementationsmenge eines Belangs

Sei $c \in C$ ein Belang:

$\mathcal{I}(c) = \{ u \in \mathcal{U}(c) \mid \neg decl(u) \}$ heißt ***Implementationsmenge*** des Belangs.

Sei $hs \in C$ ein Hyperslice, $c \in C$ ein Belang.

i) hs ***betrifft*** c gdw. sich die Implementationsmengen schneiden:
$\mathcal{I}(c) \cap \mathcal{I}(hs) \neq \emptyset$

ii) hs ***kapselt*** c gdw. die Implementationsmenge von c in der Implementationsmenge von hs enthalten ist:
$\mathcal{I}(c) \subseteq \mathcal{I}(hs)$

iii) hs ***kapselt*** c ***perfekt*** gdw. die Implementationsmengen übereinstimmen:
$\mathcal{I}(c) = \mathcal{I}(hs)$

4.1.2.6. Integration

Isoliert sind Hyperslices ziemlich nutzlos. Um eine ablauffähige Software zu erhalten, muss es ein Hyperslice geben, das eine *kompatible* Implementierung v zu dem deklarativen Grundelement v_{decl} eines anderen Hyperslice bereitstellt. Diese Art von Beziehung zwischen Grundelementen wird *Korrespondenz (correspondence)* genannt. Korrespondenz ist nicht auf die Quellcodeebene beschränkt. Eine Anforderung (Grundelement der Anforderungsanalyse) kann z.B. an ein oder mehrere Designelemente gebunden sein, welche die Anforderung erfüllen.

Neben dem Binden von deklarativen Grundelementen an Implementierungen gibt es noch weitergehende Integrationstechniken. So könnte z.B. eine Methode in mehreren Hyperslices implementiert sein. Wenn diese Hyperslices integriert werden, so ist zu entscheiden, ob eine der Implementierungen bevorzugt werden soll oder beide zu einer gemeinsamen Implementierung zu integrieren sind. In letzterem Fall muss häufig eine Ausführungsreihenfolge festgelegt werden und es müssen eventuell Transformationsvorschriften für die Übersetzung unterschiedlicher Parameter- und Rückgabetypen bereitgestellt werden. Dieser Prozess der Integration verschiedener Implementierungen wird *Kombination* genannt.

Korrespondenz und Kombination sind im *Kontext* des zu erstellenden Systems zu betrachten, dieselben Grundelemente können für verschiedene Varianten des Systems anhand unterschiedlicher Korrespondenz- und Kombinationssregeln gebunden werden.

Dieser Kontext wird im Hyperspace-Ansatz durch *Hypermodules* beschrieben. Ein Hypermodule besteht aus einer Menge zu integrierender Hyperslices und einer Menge von *Kompositionsbeziehungen*, die den Integrationsprozess festlegen.

Eine Kompositionsbeziehung besteht aus einer *Korrespondenzbeziehung* zur Darstellung der Beziehungen zwischen Grundelementen und (soweit erforderlich) einer *Kombinationsfunktion*[17] zur Transformation dieser Grundelemente in ein neues Grundelement.

Der genaue Aufbau der Korrespondenzrelationen und Kombinationsfunktionen hängt in hohem Maße von den verwendeten Artefaktsprachen ab und wird deshalb hier bewusst offen gelassen. Für den Implementierungsbereich entsprechen sie beispielsweise den vom SOP bekannten *Correspondence-and-Combination-Rules* (siehe Abschnitt 3.5.2).

Definition 7 Kompositionsbeziehung

Sei $HS \subseteq C$ eine Menge von Hyperslices und $\mathcal{U}_{HS} := \bigcup_{hs \in HS} \mathcal{U}(hs)$ die Menge der Grundelemente dieser Hyperslices. Sei ferner:

i) $I = (i_1, \ldots, i_k)$ ein Tupel von Grundelementen aus HS: $I \in \text{seq } \mathcal{U}_{HS}$
ii) $f : (\text{seq } \mathcal{U}_{HS}) \rightarrow U$ eine partielle *Kombinationsfunktion*. Die Aufgabe von f ist es, die Grundelemente aus I zu einem neuen Grundelement zusammen zu fassen (kombinieren).
iii) $o = f(I) \in U$ das *Ausgabeelement* (Ergebnis der Anwendung von f auf I)

Das Tupel $cr = (I, f)$ heißt ***Kompositionsbeziehung*** zwischen Hyperslices aus HS.

4.1.2.7. Hypermodules

Während Hyperslices die Kapselung von Belangen unterstützen, werden mit Hypermodules Belange *integriert*. Ein Hypermodule kombiniert eine Menge von Hyperslices zu einem neuen Hyperslice und beschreibt, wie dieser Integrationsprozess erfolgen kann. Da das Ergebnis der Integration auch wieder ein Hyperslice ist, kann die Integration stufenweise erfolgen, um z.B. ein Artefakt, eine Komponente oder ein System zu erzeugen, das eine Menge ausgewählter Belange realisiert.

Definition 8 Hypermodule

Sei $HS \subseteq C$ eine Menge von Hyperslices und CR eine Menge von Kompositionsbeziehungen über HS.

Dann ist $hm = (HS, CR)$ ein ***Hypermodule***.

Damit sind die grundlegenden Elemente eines Hyperspace beschrieben.

Definition 9 Hyperspace

Sei U eine Menge von Grundelementen, M eine Belangmatrix über U sowie HM eine Menge von Hypermodules über M.

Dann ist $H = (U, M, HM)$ ein ***Hyperspace***.

4.1.2.8. Kombination und Kompatibilität

Ein wichtiger Gesichtspunkt bei der Integration ist zweifelsohne die Frage der *Kompatibilität*. Um ein sinnvolles Hypermodule zu erhalten, muss sichergestellt sein, dass die deklarativen Grundelemente an *kompatible* Implementierungen gebunden werden. Kompa-

[17] Die Kombinationsfunktion wird im Original [OT99] als Kompositionsfunktion (composition function) bezeichnet.

tibilität umfasst hier sowohl syntaktische als auch semantische Gesichtspunkte. Während sich syntaktische Kompatibilität weitgehend automatisch prüfen lässt, ist der Aufwand für eine auch nur halbautomatische Überprüfung der Semantik ungleich höher. Letztendlich ist hier immer der Entwickler gefordert, der ein gesundes Verständnis für das Gesamtsystem mitbringen muss.

4.2. Hyperspaces für Java: Hyper/J[18]

Hyper/J ist eine erste Realisierung des Hyperspace-Ansatzes beschränkt auf Java als Artefaktsprache. Hyper/J fußt zu großen Teilen auf den Konzepten des *Subject-Oriented Programming (SOP)* (Kapitel 3.5) und ist somit auch die „modernste SOP-Sprache". Insbesondere dank der mit dem Hyperspace-Modell einhergehenden Strukturierungsmöglichkeiten durch zusätzliche Belangdimensionen geht Hyper/J aber noch deutlich über das „klassische" SOP hinaus.

4.2.1. Einführung

Hyper/J unterstützt mehrdimensionales Trennen der Belange für Java. Dabei werden die Elemente einer Menge von Java-Paketen in einem Hyperspace angeordnet und somit das simultane Identifizieren, Kapseln und Integrieren von Belangen entlang mehrerer Dimensionen ermöglicht. Im Gegensatz zu den meisten anderen Ansätzen arbeitet Hyper/J mit *binären* Java-Paketen (.class/.jar Dateien), führt also keinerlei Änderungen im Quellcode durch. Um Hyper/J zu verwenden ist es nicht erforderlich, das Projekt von vorneherein mit Hyper/J entwickelt zu haben, vielmehr kann der Einsatz von Hyper/J auch zu einem späteren Zeitpunkt erfolgen. Im Einzelnen:

- Hyper/J arbeitet auf binären Java .class Dateien und erzeugt als Ausgabe wiederum binäre Java .class Dateien. Dadurch kann Hyper/J mit jedem beliebigen Java-Entwicklungswerkzeug verwendet werden, die erzeugten .class Dateien laufen auf jeder standardkonformen Java2 VM. Die technische Einstiegshürde für eine Verwendung von Hyper/J ist somit minimal.
- Der Einsatz von Hyper/J kann sowohl bei der Entwicklung neuer Software als auch bei der Wartung und Weiterentwicklung vorhandener Systeme erfolgen. Das System kann vorher mit oder ohne Hyper/J und mit oder ohne mehrdimensionalem Trennen der Belange entwickelt worden sein. Die organisatorische Einstiegshürde für eine Verwendung von Hyper/J ist damit ebenfalls sehr gering.
- Neben den von der SOP bekannten Möglichkeiten zur Integration verschiedener Objektmodelle (Sichten) in ein Gesamtsystem bietet Hyper/J weitere Möglichkeiten. So ist es möglich, „on demand" zusätzliche Komponenten einzubinden, das System zu remodularisieren, Varianten zu erstellen (product lines) und Einzelteile flexibel wieder zu verwenden. Dimensionen und Belange können nach Bedarf hinzugefügt oder entfernt werden.

[18] [TO99], [OT00], [OT00a], [HS]

- Da Hyper/J allein auf der Basis von .class Dateien arbeitet, sind alle derartigen Vorgänge nicht-invasiv, sie erfordern keinerlei Änderungen im Quellcode – der Quellcode muss nicht einmal vorhanden sein. Hyper/J ermöglicht somit auch die Integration kommerzieller Bibliotheken von Drittherstellern, bei denen oft keine Quellen mitgeliefert werden.
- Die flexiblen Einsatzmöglichkeiten machen das Hyper/J-System auch zu einer hervorragenden Grundlage für Wartungswerkzeuge. Mit HyperProbe [KKO+02] gibt es inzwischen ein darauf aufbauendes Werkzeug, dass die Verwendung von Hyper/J nochmals deutlich vereinfacht und speziell für „Notfall-" Wartungsaufgaben im Umfeld von Application-Servern mit Enterprise Java Beans optimiert ist.[19]

Hyper/J unterstützt, wie gesagt, nur einen einzigen Typ von Artefakten – Java-Code – und realisiert damit eigentlich nur einen Bruchteil der potentiellen Möglichkeiten des Hyperspace-Ansatzes. Insbesondere durch die einfachen Integrations- und Remodularisierungsmöglichkeiten durch Hypermodules ist Hyper/J jedoch schon jetzt sinnvoll einsetzbar.

4.2.2. Hyper/J verwenden

Hyper/J wird von der Kommandozeile gestartet und über eine Reihe von Konfigurationsdateien gesteuert. Dabei müssen übergeben werden:

- Ein *Hyperspace Specification File (HSF)*, in dem Java-Klassen und Pakete (.class Dateien) aufgelistet sind. Das HSF definiert die Menge der *Grundelemente (Eingabe)*, die im Hyperspace angeordnet werden. Grundelemente sind (Java-) Klassen, Schnittstellen, Methoden und Datenelemente. Hyper/J erzeugt hieraus automatisch die Dimension <Classes>, in der jede aufgenommene Klasse oder Schnittstelle als ein Belang dargestellt wird. Jedes Grundelement wird in dieser Dimension dann seiner Klasse/Schnittstelle zugeordnet. Die <Classes>-Dimension ist damit ein exaktes Abbild der importierten Java-Pakete.
- Ein oder mehrere *Concern Mapping Files (CMF, .cm)*, in denen zusätzliche Dimensionen und Belange definiert werden und die Zuordnung der Grundelemente zu diesen Belangen erfolgt.
- Ein oder mehrere *Hypermodule Specification Files (HMF, .hm)*, in denen jeweils ein Hypermodule beschrieben ist. Ein HMF besteht dabei aus einer Definition von Hyperslices und einer Menge von Integrationsregeln für deren Zusammen-

---

[19] HyperProbe wurde für Wartungsaufgaben der so genannten „rauen Wirklichkeit" entwickelt, die jedem ordentlichen Softwaretechniker die Haare zu Berge stehen lassen würden. Bei einer Demonstration von HyperProbe [KKO+02] schilderte DOUG KIMELMAN die Umstände des typischen Einsatzes etwa wie folgt: *„Auf dem Application-Server eines wichtigen Kunden macht irgendein Java-Bean mächtig Zicken und morgen soll über den Nachfolgeauftrag verhandelt werden. Du hast die Stunde zwischen 3 und 4 Uhr nachts zur Verfügung das Problem zu lösen. Auf dem Rechner, der Dir zur Verfügung steht, ist natürlich keine Entwicklungsumgebung installiert – und wo die Quelltexte sind weiß auch niemand."*

führung. Hyper/J erzeugt für jedes Hypermodule ein eigenes Java-Paket. HMFs definieren also die zu erzeugende *Ausgabe* von Hyper/J.

Unter bestimmten Umständen kann Hyper/J bei Kleinstprojekten auch automatisch ein HMF generieren. Ebenfalls für eher kleine Projekte besteht außerdem die Möglichkeit die Angaben aus HSF, CMFs und HMFs in einer einzigen Datei zusammenzufassen.

4.2.3. Aufbau eines Hyperspace Specification File (HSF)

Ein HSF beginnt mit dem Schlüsselwort `hyperspace` und einem Bezeichner für den zu erzeugenden Hyperspace. Daran schließt sich die Liste der einzufügenden Java-Pakete an:

4.2.3.1. Syntax

```
hyperspaceDefinitionFile ::= hyperspace Name
  [classFileSpecification;]*

classFileSpecification ::=
  [composable | uncomposable] unit unitName [, unitName]*
  [except unitName] [, unitName]*

unit ::= class | file
```

Die Pakete (*unitName*) können in der Form von Java-Klassen oder als Dateipfade im Dateisystem angegeben werden. Im ersteren Fall wird das Paket im Java-Klassenpfad gesucht, im zweiten Fall im Dateisystem. Die Namen können auch einfache Wildcards enthalten (*). Mit dem Schlüsselwort `except` lassen sich zusätzlich explizite Ausnahmen formulieren.

4.2.3.2. Beispiel

```
hyperspace DemoHS
  class package1.classA, package1.classB;
  class package2.* except package2.classC, package2.classD;
  composable file c:\demo\myproject\*;
  uncomposable class java.lang.*, java.lang.io.*;
```

Pakete werden entweder als *kombinierbar* (`composable`) oder *nicht kombinierbar* (`uncomposable`) in den Hyperspace eingebunden. Der Default ist dabei kombinierbar. Nur kombinierbare Klassen können später bei der Erstellung von Hypermodules mit anderen Klassen kombiniert werden und in den Integrationsregeln verwendet werden. Standard Java Klassen wie `Object` oder `String` sind typische Kandidaten für nicht kombinierbare Klassen.

Zusätzlich zu den im HSF explizit aufgeführten Klassen bindet Hyper/J jene Klassen als nicht kombinierbar ein, von denen die aufgeführten Klassen abhängen.[20]

[20] Der Begriff der Abhängigkeit bezieht sich hier auf all jene Klassen, die bei der Deklaration oder Definition der aufgeführten Klassen verwendet werden: Oberklassen sowie die Klassen der Membervariablen, Methodenparameter und lokalen Methodenvariablen. Dabei werden natürlich auch wiederum deren Abhängigkeiten überprüft und eingebunden, d.h. es werden alle Klassen der transitiven Hülle bzgl. der Abhängigkeitsrelation eingebunden.

4.2.4. Aufbau eines Concern Mapping File (CMF)

Ein CMF besteht aus einer Folge von *Mapping*s, die jeweils ein oder mehrere Grundelemente einem Belang in einer Dimension zuordnen. Belange und Dimensionen werden dabei implizit mit der ersten Verwendung eingeführt, sie brauchen nicht deklariert zu werden:

4.2.4.1. Syntax

```
concernMappingFile ::=
  [unitKind unitName : dimensionName . concernName ;]*

unitKind ::= package | class | interface | operation | field
```

Bei der Auswertung der Regeln folgt Hyper/J einen einfachen Prinzip: Im Konfliktfall überschreiben spätere Regeln die vorangegangenen. Dieses gilt auch bei der Verwendung mehrerer CMFs, hier ist die Reihenfolge bei der Übergabe auf der Kommandozeile maßgebend.

Dieses Prinzip ermöglicht es auf einfache Weise, zunächst eine Standardzuordnung anzugeben und dann anschließend Ausnahmen explizit zu machen:

4.2.4.2. Beispiel

```
1:  package package1             : Feature.someFeature;
2:  package package2             : Feature.someOtherFeature;
3:  class package1.classA        : Feature.A;
4:  field package1.classA.fieldX : Feature.X;
5:  operation classA.setX        : Feature.X;
6:  field fieldY                 : Feature.Y;
```

Die Zeilen 1 und 2 legen fest, dass alle Elemente aus den Paketen `package1` bzw. `package2` auf die Belange `Feature.someFeature` bzw. `Feature.someOtherFeature` abgebildet werden. Damit werden implizit auch die Dimension `Feature` sowie die entsprechenden Belange eingeführt. In den Zeilen 3 und 4 werden nun Ausnahmen von dieser Zuordnung formuliert. Alle Elemente der Klasse `package1.classA` werden damit `Feature.A` zugeordnet, mit Ausnahme des Feldes `package1.classA.fieldX`, welches `Feature.X` zugeordnet wird. An dieser Stelle wird deutlich, dass die Reihenfolge der Regeln entscheidend ist. Vertauschte man die Zeilen 3 und 4, so würde das Feld `package1.classA.fieldX` ebenfalls `Feature.A` zugeordnet, da die entsprechende Regel nun weiter hinten stände.

Zeile 5 legt fest, dass die Methoden `setX` *aller* Klassen mit dem Namen `classA` ebenfalls `Feature.X` zugeordnet werden. Und schließlich bestimmt die Regel in Zeile 6, dass Felder mit dem Namen `fieldY` aus allen Klassen und Paketen `Feature.Y` zugeordnet werden.

4.2.5. Aufbau eines Hypermodule Specification File (HMF)

Durch ein HMF wird beschrieben, wie ein Hypermodule aus einer Reihe von Belangen integriert wird. Jedes HMF beginnt mit dem Schlüsselwort `hypermodule` gefolgt von einem Bezeichner für das Hypermodule. Hyper/J verwendet diesen Bezeichner als Paketnamen für die zu generierenden Klassen. Im Anschluss wird die Menge der zu integrierenden Belange angegeben (`hyperslices:`), gefolgt von den zu verwendenden Kompositionsbe-

ziehungen (`relationships:`). Mit dem Schlüsselwort `end hypermodule;` wird das HMF schließlich abgeschlossen:

4.2.5.1. Syntax

```
hypermoduleSpecificationFile ::=
  hypermodule hypermoduleName

    hyperslices:
      dimensionName.concernName [, dimensionName.concernName]* ;

    relationships:
      (mergeByName | overrideByName | nonCorrespondingMerge) ;
      [relationship;]*

  end hypermodule;

relationship ::= equateRel | matchRel | noMergeRel | mergeRel | summaryFunctionRel
|
                 overrideRel | bracketRel | orderRel | renameRel
```

Sinn und Zweck eines Hypermodules ist es, verschiedene Belange flexibel kombinieren und zu einem Java Paket integrieren zu können. Dreh- und Angelpunkt dieser Integration sind die im HMF aufgeführten Integrationsbeziehungen (relationships). An dieser Stelle zeigt sich auch die Verwandtschaft von Hyper/J zum SOP: Die möglichen Integrationsbeziehungen in Hyper/J leiten sich direkt aus den Kompositionsregeln des SOP ab. Im Folgenden werden die zur Verfügung stehenden Integrationsregeln anhand ihrer Syntax vorgestellt und erläutert. Für eine formale Beschreibung der Semantik sei auf entsprechende Veröffentlichungen zu SOP verwiesen, wie [OKK+96, OHKK95].

4.2.6. Die generelle Kompositionsstrategie

Hyper/J verfolgt den Ansatz, zunächst eine generelle Strategie für die Komposition festzulegen. Soweit erforderlich, können dann in zusätzlichen Regeln Ausnahmen und Verfeinerungen für bestimmte Grundelemente formuliert werden. Wie beim SOP besteht eine *Kompositionsstrategie (composition)* aus *Korrespondenz* (*correspondence*) und *Kombination (combination)* (siehe auch Abschnitt 3.5.2). Korrespondenzregeln legen fest, *welche* Grundelemente kombiniert werden sollen. Kombinationsregeln legen fest, *wie* die Definitionen korrespondierender Grundelemente zu integrieren sind. Die Unterscheidung dieser – unglücklicherweise recht ähnlich klingenden – Begriffe ist essentiell für das Verständnis, weshalb ich sie hier nochmals explizit aufführe:

- Mit *Korrespondenzregeln* wird festgelegt, *welche* Grundelemente miteinander in Beziehung stehen und zusammengefügt werden sollen. Beispielsweise könnten alle Klassen mit identischen Bezeichnern zusammengefügt werden.
- *Kombinationsregeln* steuern hingegen das eigentliche Zusammenfügen. Eine Kombinationsregel legt fest, *wie* die in Beziehung stehenden (korrespondierenden) Grundelemente zusammengefügt werden. Beispielsweise könnte die zusammengefügte Klasse aus der Vereinigung aller Methoden und Datenelemente der korrespondierenden Klassen bestehen.
- Die *generelle Kompositionsstrategie* ist schließlich eine *Korrespondenz- und Kombinationsregel*. Sie besteht sowohl aus einer Korrespondenzregel als auch ei-

ner Kombinationsregel und ist damit eine Schreibabkürzung, durch die das eigentlich zweistufige Verfahren mit einer Regel zusammengefasst wird.

In jedem HMF muss zunächst genau eine der drei generellen Kompositionsstrategien `mergeByName`, `overrideByName` oder `nonCorrespondingMerge` angegeben werden:

- `mergeByName` legt fest, dass Grundelemente aus verschiedenen Hyperslices genau dann korrespondieren, wenn sie identische Bezeichner haben. Die generelle Kombinationssregel ist `merge`, wodurch jeweils korrespondierende Grundelemente zu einem neuen Grundelement integriert werden.
- Mit `overrideByName` korrespondieren Grundelemente ebenfalls genau dann, wenn sie identische Bezeichner haben. Die generelle Kombinationsregel ist hier jedoch `override`, es wird jeweils das Grundelement des zuletzt angegebenen Hyperslice übernommen. Dieses wirkt sich jedoch nur auf Methoden aus. Klassen, Datenelemente usw. werden weiterhin nach der `merge` Regel kombiniert.
- Durch `nonCorrespondingMerge` werden zunächst keinerlei Grundelemente als korrespondierend angesehen. Die generelle Kompositionsstrategie ist wiederum `merge`. Welche Elemente zu kombinieren sind muss hier durch zusätzliche Korrespondenzregeln (`equate`, `match`) angegeben werden.

Grundsätzlich korrespondieren nur Grundelemente gleichen Typs. Ein Datenfeld und eine Methode mit identischen Bezeichnern würden also in keinem Fall kombiniert werden. Datenfelder werden nur kombiniert, wenn ihr Typ übereinstimmt.

4.2.7. Korrespondenzregeln

Die Regeln `equate`, `match` und `noMerge` definieren explizite Korrespondenzbeziehungen zwischen Grundelementen. Durch `equate` wird eine n→1 Korrespondenz definiert, die festlegt, dass eine Menge von Grundelementen zu einem gemeinsamen Grundelement kombiniert werden soll. Mit `match` wird eine (1×n)→n Korrespondenz definiert, die bestimmt, dass ein Grundelement mit einer Menge von weiteren Grundelementen elementweise kombiniert werden soll. Wichtig ist, dass durch `equate` und `match` nur die Korrespondenz definiert wird. Wie die korrespondierenden Elemente kombiniert werden, ist durch die generelle Kompositionsstrategie vorgegeben.

Mit `noMerge` lassen sich schließlich Korrespondenzen, die durch die generelle Kompositionsstrategie vorgegeben wurden, explizit auflösen. Dadurch wird eine sonst erfolgende Kombination der entsprechenden Grundelemente unterbunden.

Für Methoden wird bei einigen Regeln zwischen den Grundelementtypen `action` und `operation` unterschieden:

- Eine `operation` bezieht sich jeweils auf die Menge *aller* Methoden mit dem angegebenen Namen und der angegebenen Signatur aus beliebigen Klassen. Eine `operation` repräsentiert damit quasi eine generische Methode.
- Eine `action` bezieht sich hingegen auf eine konkrete Methode in einer bestimmten Klasse.

4.2.7.1. Equate

```
equateRel ::=
  equate unitKind unitName [, unitName]* [into newUnitName];

unitKind ::= class | interface | operation | action | field
```

Mit `equate` wird eine Korrespondenzbeziehung zwischen einer Menge von Grundelementen *unitName* definiert, die alle miteinander zu einem Ausgabeelement kombiniert werden (n→1). Der Name des Ausgabeelementes kann optional mit `into` *newUnitName* angegeben werden, ansonsten synthetisiert Hyper/J selber einen Namen aus den Bezeichnern der Grundelemente.

4.2.7.2. Match

```
matchRel ::=
  match unitKind unitName with unitNamePattern;

unitKind ::= class | interface | operation | action | field
```

Mit `match` wird eine elementweise Korrespondenz zwischen dem Grundelement *unitName* und der Grundelementmenge *unitNamePattern* definiert ((1×n)→n). Die Menge *unitNamePattern* wird dabei durch einen regulären Ausdruck über den Namen der Grundelemente bestimmt.

4.2.7.3. NoMerge

```
noMergeRel ::=
  noMerge unitKind unitName [, unitName]* ;

unitKind ::= class | interface | operation | action | field
```

Mit `noMerge` wird die Korrespondenz zwischen den angegebenen Grundelementen aufgelöst. Damit werden diese Grundelemente nicht mehr kombiniert, auch wenn sie laut der generellen Kompositionsstrategie korrespondieren.

4.2.8. Kombinationsregeln

Durch die Regeln `merge`, `override` und `bracket` werden Kombinationsbeziehungen definiert. Streng genommen handelt es sich dabei sogar um Korrespondenz- und Kombinationsbeziehungen, da die zu kombinierenden Elemente jeweils mit angegeben werden.

Durch `merge` wird, unabhängig von der generellen Kompositionsstrategie, eine n→1 Kombinationsbeziehung definiert, durch die eine Menge von Grundelementen zu einem gemeinsamen Grundelement kombiniert wird. Die `override` Regel bewirkt hingegen, dass die Definitionen der angegebenen Methoden nicht kombiniert werden, sondern einer bestimmten Definition der Vorzug gegeben werden soll. Mit `bracket` schließlich lässt sich festlegen, dass vor und/oder nach der Ausführung einer Methode bestimmter Code ausgeführt wird.

4.2.8.1. Merge

```
MergeRel ::=
  merge unitKind unitName [, unitName]* [into newUnitName];

unitKind ::= class | interface | operation | action | field
```

Mit `merge` wird zunächst eine `equate` Korrespondenz zwischen den angegebenen Grundelementen definiert. Darüber hinaus werden die korrespondierenden Grundelemente *unitName* jedoch in jedem Fall zu einem gemeinsamen Grundelement *newUnitName* zusammengefasst, unabhängig von der generellen Kompositionsstrategie. Bei der Kombination von Methoden wird deren Implementierung in der angegebenen Reihenfolge zusammengefügt. Dadurch wird standardmäßig auch der Rückgabewert des zuletzt angegebenen Grundelementes als Rückgabewert des Komposits verwendet. Für den Fall, dass dieses Verhalten nicht ausreichend ist, kann der Rückgabewert durch eine *Summary Function* auch aus den Rückgabewerten der einzelnen Methoden berechnet werden:

```
summaryFunctionRel ::=
  set summary function for unitType outputUnitName to summaryFunction ;

summaryFunction ::= [external] unitName

unitType ::= action | operation
```

Eine Summary Function muss eine statische Java-Methode mit beliebiger Sichtbarkeit sein. Sie bekommt einen Vektor der Rückgabewerte der kombinierten Methoden übergeben und muss daraus einen einzelnen Rückgabewert berechnen. Die Methode muss nicht unbedingt in einem der zu integrierenden Hyperspace enthalten sein, durch Angabe des Schlüsselwortes `external` kann auch eine externe Methode verwendet werden. Hyper/J liefert eine Reihe häufig benötigter Summary Functions bereits mit.

4.2.8.2. Override

```
overrideRel ::=
  override unitKind unitName [, unitName]* with
  unitKind otherUnitName ;

unitKind ::= class | interface | operation | action
```

Auch `override` definiert zunächst eine Korrespondenz zwischen den angegebenen Grundelementen. Die korrespondierenden Grundelemente *unitName* werden jedoch, unabhängig von der generellen Kompositionsstrategie, mit dem Grundelement *otherUnitName* überschrieben. `override` wirkt sich nur auf Methoden aus. Zur Schreibabkürzung lassen sich neben einzelnen Methoden (`operation`, `action`) auch Methoden-Container (`class`, `interface`) angeben, in diesem Fall wird die Regel auf jede Methode des Containers angewendet.

4.2.8.3. Bracket

```
bracketRel ::=
  bracket [classMatchPattern .] operationMatchPattern
  [from unitKind unitName [, unitName]* ] [with]
    (before fullyQualifiedMethodName [methodParams] ;)
  | (after fullyQualifiedMethodName [methodParams] ;)
  | (before fullyQualifiedMethodName [methodParams],
     after fullyQualifiedMethodName [methodParams] ; )

unitKind ::= hyperslice | class | operation | action
methodParams ::= ( Param [, Param ] )
Param ::= $ClassName, $OperationName
```

Durch eine `bracket` Regel wird eine Menge von Methoden „geklammert", d.h. vor (`before`) und/oder nach (`after`) ihrer Ausführung wird zunächst eine andere Methode aufgerufen. Bei der Methodenausführung kann wahlweise nur vorher, nur nachher oder vorher und nachher in eine angegebene „Klammermethode" verzweigt werden.

Die Menge der zu klammernden Operationen wird durch reguläre Ausdrücke über Klassen- und Methodennamen festgelegt. Der reguläre Ausdruck *operationMatchPattern* beschreibt die Methoden, die geklammert werden sollen. Optional lässt sich die Menge der zu klammernden Methoden durch den Ausdruck *classMatchPattern* auf Methoden bestimmter Klassen beschränken.

Mit Hilfe der `from` Klausel lässt sich der Kontext, in dem die `before`/`after` Methode aufgerufen wird, zusätzlich durch Bedingungen über den Programmfluss einschränken. Der Aufruf erfolgt nur im Kontext der hinter dem Schlüsselwort `from` angegebenen Methoden, d.h. nur wenn wenigstens eine dieser angegebenen Methoden zum Zeitpunkt des Aufrufs im Aufrufstapel enthalten ist. Das Schlüsselwort `with` ist optional und dient nur der besseren Lesbarkeit.

Optional können der `before`/`after` Methode Informationen über die geklammerte Methode als Parameter übergeben werden. Dazu stehen die Schlüsselworte `$ClassName` und `$OperationName` zur Verfügung, mit denen der Klassenname respektive Methodenname als String übergeben wird.[21]

4.2.9. Weitere Anweisungen

Die Regeln `order` und `rename` entsprechen ihrem Charakter nach eher Anweisungen als Korrespondenz- oder Kombinationsregeln. Durch `order` lässt sich die durch Anordnung der Hyperslices vorgegebene Ordnung der Grundelemente ändern und somit Einfluss auf die Kombination der Elemente nehmen. Mit `rename` kann man Elementen des Hypermodules explizit andere Namen zuweisen.

[21] Laut Hyper/J Manual [TO99, S.16] soll es prinzipiell auch möglich sein, weitere Parameter sowie die Aufrufparameter der geklammerten Methode an die `before`/`after` Methoden zu übergeben. Leider sind Details dazu weder im Hyper/J Manual noch in den mitgelieferten Bespielen zu finden.

4.2.9.1. Order

```
orderRel ::=
  order unitKind unitName [, unitName]* (before | after)
  unitKind unitName [, unitName]*;

unitKind ::= hyperslice | class | interface | operation | action
```

Bei der Kombination von Methoden entscheidet normalerweise die Reihenfolge der Hyperslices darüber, in welcher Reihenfolge (merge) die zu kombinierenden Methodendefinitionen in das Komposit aufgenommen werden. Mit Hilfe von `order` Anweisungen lässt sich diese Reihenfolge beeinflussen, indem eine partielle Ordnung (`before` oder `after`) zwischen zwei Mengen von Grundelementen definiert wird. Neben Methoden (`operation`, `action`) lassen sich als Grundelemente dabei auch Methoden-Container wie `class`, `interface` und `hyperslice` angeben. Dieses entspricht dann einer `order` Anweisung für jede einzelne Methode des Containers.

4.2.9.2. Rename

```
renameRel ::=
  rename unitKind unitName to newUnitName;

unitKind ::= class | interface | operation | action | field
```

Mit der `rename` Anweisung lässt sich die Benennung von Grundelementen in dem zu erzeugenden Hypermodule (in der Ausgabe) beeinflussen. Damit kann ein Element, das normalerweise in der Ausgabe *unitName* hieße zu *newUnitName* umbenannt werden.

4.3. Beispiel

Das Beispielprogramm stellt eine einfache Softwareentwicklungsumgebung (SEU) (*software engineering environment, SEE*) dar und entspricht, mit Anpassungen und Ergänzungen, dem Beispiel aus [TOHS99, OT99, OT00]. Die SEU erlaubt, einfachste Programme auf der Basis von Ausdrücken zu erstellen und bietet dazu eine Reihe von Werkzeugen, die auf einer gemeinsamen Datenstruktur in Form eines abstrakten Syntaxbaums (ASB) arbeiten. Abbildung 7 zeigt die grundlegende Architektur.

4.3.1. Anforderungen und Entwurf

Die Anforderungen an die erste Version der SEU sind folgende:

Die SEU ermöglicht es, Expression-Programme zu erzeugen und bearbeiten. Sie beinhaltet eine Reihe von Werkzeugen (Tools). Als Werkzeuge sollen angeboten werden:

Evaluation Tool: Ermittelt das Ergebnis eines Ausdrucks und zeigt dieses auf dem Bildschirm an.

Display Tool: Gibt einen Ausdruck symbolisch auf dem Bildschirm aus.

Check Tool: Überprüft einen Ausdruck auf syntaktische und semantische Korrektheit.

Die Werkzeuge sollen auf einer gemeinsamen Datenstruktur für Ausdrücke arbeiten. Zur Laufzeit lassen sich anhand dieser Datenstruktur beliebige Ausdrücke erstellen, bearbeiten und auslesen.

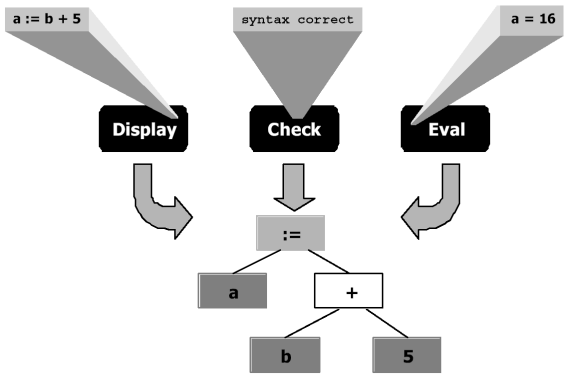

Abbildung 7 Architektur der SEU
Werkzeuge wie Display, Check und Eval arbeiten auf einer gemeinsamen Datenstruktur in Form eines abstrakten Syntaxbaums.
(Abbildung aus [OT00])

In Abbildung 8 ist dargestellt, wie diese Anforderungen in einem typischen objektorientierten UML-Design realisiert werden. Die möglichen Programmelemente werden dabei jeweils als eine Klasse, die Werkzeuge als eine oder mehrere Methoden modelliert. Der eigentliche Programmcode wäre ähnlich strukturiert.

4.3.2. Dimensionen

Schon an diesem einfachen Beispiel lassen sich eine Reihe verschiedener Arten (Dimensionen) von Belangen identifizieren, darunter:

Klassen: Jede Klasse aus dem Design (und dem Code) repräsentiert einen einzelnen Belang des Objektmodells.

Features: Aus den Anforderungen lässt sich das System in vier zusammenhängende Features zerlegen: Drei davon bilden die angesprochenen Tools mit dem *Check-Tool*, dem *Eval-Tool* und dem *Display-Tool*. Außerdem wird noch eine gemeinsame Datenstruktur gefordert, mit der sich beliebige Expression-Programme darstellen, verändern und auslesen lassen. Letzteres sind die eigentlichen Kernfunktionen der SEU, sie werden hier im Feature *Kernel* zusammengefasst.

Artefakte: In den verschiedenen Phasen des Softwarelebenslaufes entstehen jeweils un-

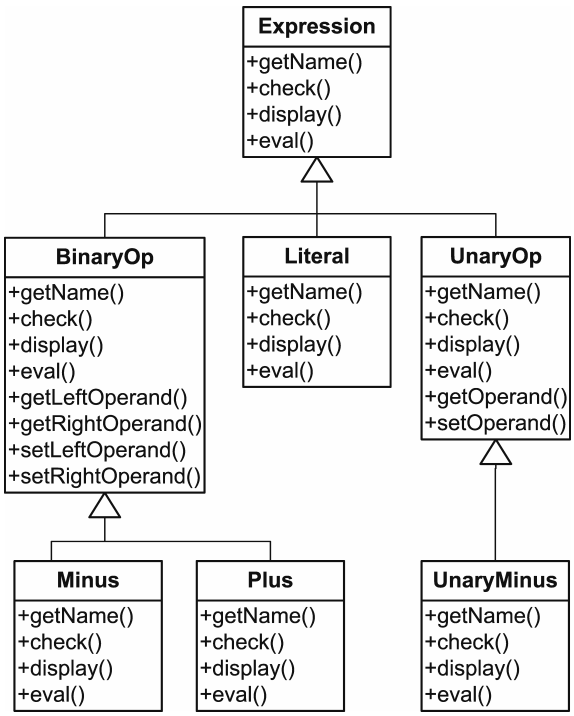

Abbildung 8 UML Entwurf der SEU (Ausschnitt)
Ausgehend von einer gemeinsamen Root-Klasse `Expression` sind die möglichen Programmelemente (ASB-Klassen) als Unterklassen modelliert. Die Werkzeuge sind als Methoden der ASB-Klassen implementiert.
(Abbildung verändert nach [OT00])

terschiedliche Artefakte zur Beschreibung des Programms, wie z.B. Anforderungslisten, Architekturbeschreibungen, Entwurfsdokumente, Quelltext oder Testpläne.

4.3.3. Erstellen der Belangmatrix

Um die Übersichtlichkeit zu erhalten, beschränke ich mich bei der Erstellung der Belangmatrix auf zwei der oben angesprochenen Dimensionen: Features und Klassen. Da sich außerdem die ASB-Klassen alle sehr ähneln, nehme ich auch hier nur eine Auswahl in die Belangmenge auf. Die Belangmatrix $\boldsymbol{M} = (\boldsymbol{C}, \boldsymbol{D})$ sowie die Menge $\boldsymbol{U}$ der Grundelemente werden wie folgt festgelegt:

$$
\begin{aligned}
\boldsymbol{C} &= \{\text{Expression, Literal, UnaryOp, Kernel, Eval, Display, Check}\} \\
\boldsymbol{D} &= \{\boldsymbol{Classes}, \boldsymbol{Features}\} \\
\boldsymbol{Classes} &= \{\text{Expression, Literal, UnaryOp, } \boldsymbol{N_{Classes}}\} \\
\boldsymbol{Features} &= \{\text{Kernel, Eval, Display, Check, } \boldsymbol{N_{Features}}\} \\
\boldsymbol{U} &= \{\text{Expression, Literal, UnaryOp, Expression.getName,} \\
&\quad \text{Expression.check, Expression.display, Expression.eval,} \\
&\quad \text{Literal.getName, Literal.check, Literal.display, Literal.eval,} \\
&\quad \text{UnaryOp.getName, UnaryOp.check, UnaryOp.display,} \\
&\quad \text{UnaryOp.eval, UnaryOp.getOperand, UnaryOp.setOperand}\}
\end{aligned}
$$

In Hyper/J könnte das Hyperspace Specification File (HSF) für die dargestellte Grundelementmenge $\boldsymbol{U}$ folgendermaßen aussehen:

```
hyperspace SEUDemo
  composable class SEUDemo.Expression
  composable class SEUDemo.Literal
  composable class SEUDemo.UnaryOp
```

Die identifizierten Belange müssen nun spezifiziert werden, indem man ihnen Grundelemente zuordnet. Jedes Grundelement muss dabei in jeder Dimension genau einem Belang zugewiesen werden. Die Zuordnungsrelation kann sehr umfangreich werden, weshalb sich intentionale Spezifikationen anbieten. In Hyper/J sieht der entsprechende Regelsatz in Form eines Concern Mapping Files (CMF) (siehe 4.2.4) folgendermaßen aus:

```
package ExpressionSEU   : Features.Kernel
operation display       : Features.Display
operation eval          : Features.Eval
operation check         : Features.Check
```

Hyper/J verfolgt den Ansatz zunächst allgemeine Zuordnungen vorzunehmen, um diese dann mit zusätzlichen Regeln für einzelne Elemente zu überschreiben. Mit der ersten Zeile werden alle Grundelemente dem Feature *Kernel* zugewiesen. Damit wird implizit auch die Dimension <Features> eingeführt. Die folgenden Zeilen beschreiben Ausnahmen von der allgemeinen Regel, indem einzelne Operationen anderen Features zugeordnet werden. Die Dimension <Classes> wird in Hyper/J automatisch aus den entsprechenden `.class` Dateien generiert und muss deshalb nicht explizit spezifiziert wer-

den. Abbildung 9 zeigt die Belangmatrix mit den zugeordneten Grundelementen in einer graphischen Darstellung.

4.3.4. Definieren der Hyperslices und des Hypermodules

Nachdem alle Grundelemente in die Belangmatrix eingefügt wurden, müssen nun Hyperslices definiert werden, welche die einzelnen Belange als abgeschlossene, wieder verwendbare Module bereitstellen. Im SEU-Hyperspace soll jeder der identifizierten Belange einzeln wieder verwendbar sein, deshalb wird für jeden Belang ein Hyperslice definiert, das den entsprechenden Belang perfekt kapselt.

Der Unterschied zwischen Belangen und Hyperslices ist, dass Hyperslices *deklarativ vollständig* sein müssen. Dazu müssen Abhängigkeiten zwischen den Grundelementen identifiziert werden und, immer wenn ein benutztes Grundelement nicht Element derselben Implementationsmenge ist, ein entsprechendes deklaratives Grundelement eingefügt werden.

Die konkreten Abhängigkeiten zwischen Grundelementen sollten eigentlich aus der Spezifikation hervorgehen. In der Praxis werden sie meistens durch die Implementierungen der Grundelemente definiert. Das Ermitteln der Abhängigkeiten und Erzeugen der de-

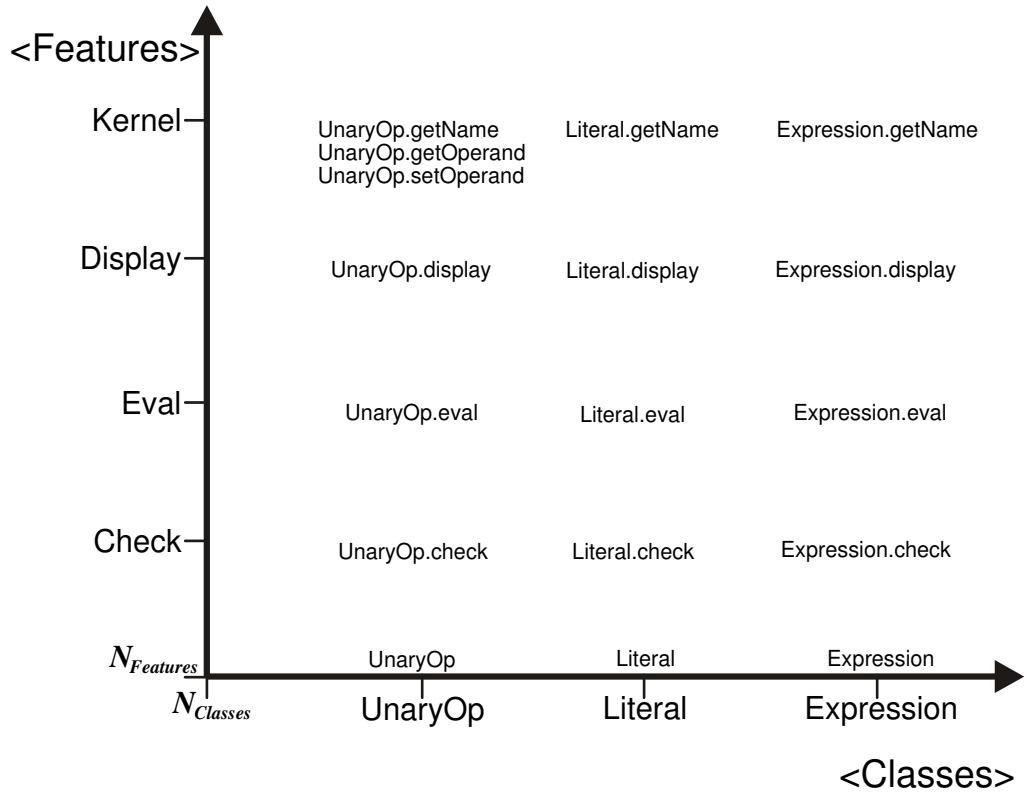

Abbildung 9 Belangmatrix der SEU (Ausschnitt)
Auf der Hochachse ist die Belangdimension <Features> dargestellt, auf der Längsachse die Belangdimension <Classes>, jeweils mit den entsprechenden Belangen. Jedes Grundelement ist genau einem Belang in jeder Dimension zugeordnet. Jede Dimension verfügt über einen speziellen *Nullbelang* N, dem diejenigen Grundelemente zugeordnet werden, die in der entsprechenden Dimension keine Bedeutung haben.

klarativen Grundelemente kann dabei, je nach Artefaktsprache, weitestgehend automatisch erfolgen. Für die SEU werden folgende Abhängigkeiten angenommen:

- Die Methode `eval` soll, laut Anforderungen, das Ergebnis auf dem Bildschirm ausgeben. Sie benötigt daher Zugriff auf die Anzeige durch `display`.
- `UnaryOp.eval` benötigt ferner Zugriff auf den Operanden und verwendet dazu `UnaryOp.getOperand`.
- `display` verwendet für die Ausgabe die Operation `getName`.

Zu jedem Belang, der als Modul verwendet werden soll, ist nun ein entsprechendes Hyperslice zu erstellen, das den Belang deklarativ vervollständigt. Um die deklarativen Grundelemente unterbringen zu können, müssen die Hyperslices in eigenen Dimensionen liegen. Die Belangmatrix $\boldsymbol{M} = (\boldsymbol{C}, \boldsymbol{D})$ und die Menge $\boldsymbol{U}$ der Grundelemente sehen dann wie folgt aus:

$\boldsymbol{C}$	=	{Expression, Literal, UnaryOp, Kernel, Eval, Display, Check, ExpressionHS, LiteralHS, UnaryOpHS, KernelHS, EvalHS, DisplayHS, CheckHS}
$\boldsymbol{D}$	=	{***Classes***, ***Features***, ***ClassesHD***, ***FeaturesHD***}
Classes	=	{Expression, Literal, UnaryOp, $\boldsymbol{N_{Classes}}$ }
Features	=	{Kernel, Eval, Display, Check, $\boldsymbol{N_{Features}}$}
ClassesHD	=	{ExpressionHS, LiteralHS, UnaryOpHS, $\boldsymbol{N_{ClassesHD}}$ }
FeaturesHD	=	{KernelHS, EvalHS, DisplayHS, CheckHS, $\boldsymbol{N_{FeaturesHD}}$}
$\boldsymbol{U}$	=	{Expression, Literal, UnaryOp, Expression.getName, Expression.check, Expression.display, Expression.eval, Literal.getName, Literal.check, Literal.display, Literal.eval, UnaryOp.getName, UnaryOp.check, UnaryOp.display, UnaryOp.eval, UnaryOp.getOperand, UnaryOp.setOperand, UnaryOp.getName_d, Literal.getName_d, Expression.getName_d, UnaryOp.display_d, UnaryOp.getOperand_d, Literal.display_d, Expression.display_d }

Die Bezeichner der deklarativen Grundelemente bestehen hier aus dem Bezeichner des Elementes auf das sie verweisen, angereichert durch das Suffix „_d“. Diese einfache Bildung der Bezeichner ist nur möglich, weil es im Beispiel kein Grundelement gibt, auf das mehr als einmal verwiesen wird. Prinzipiell kann es jedoch beliebig viele Deklarationen zu einem Grundelement geben. In einem solchen Fall müsste für jede Deklaration jeweils ein eigener Bezeichner vergeben werden.

Die Zuordnung der neuen deklarativen Grundelemente zu den einzelnen Dimensionen kann, wie gesagt, weitestgehend automatisch erfolgen. Dabei werden die deklarativen Elemente in den Hyperslice-Dimensionen den entsprechenden Belangen (Hyperslices)

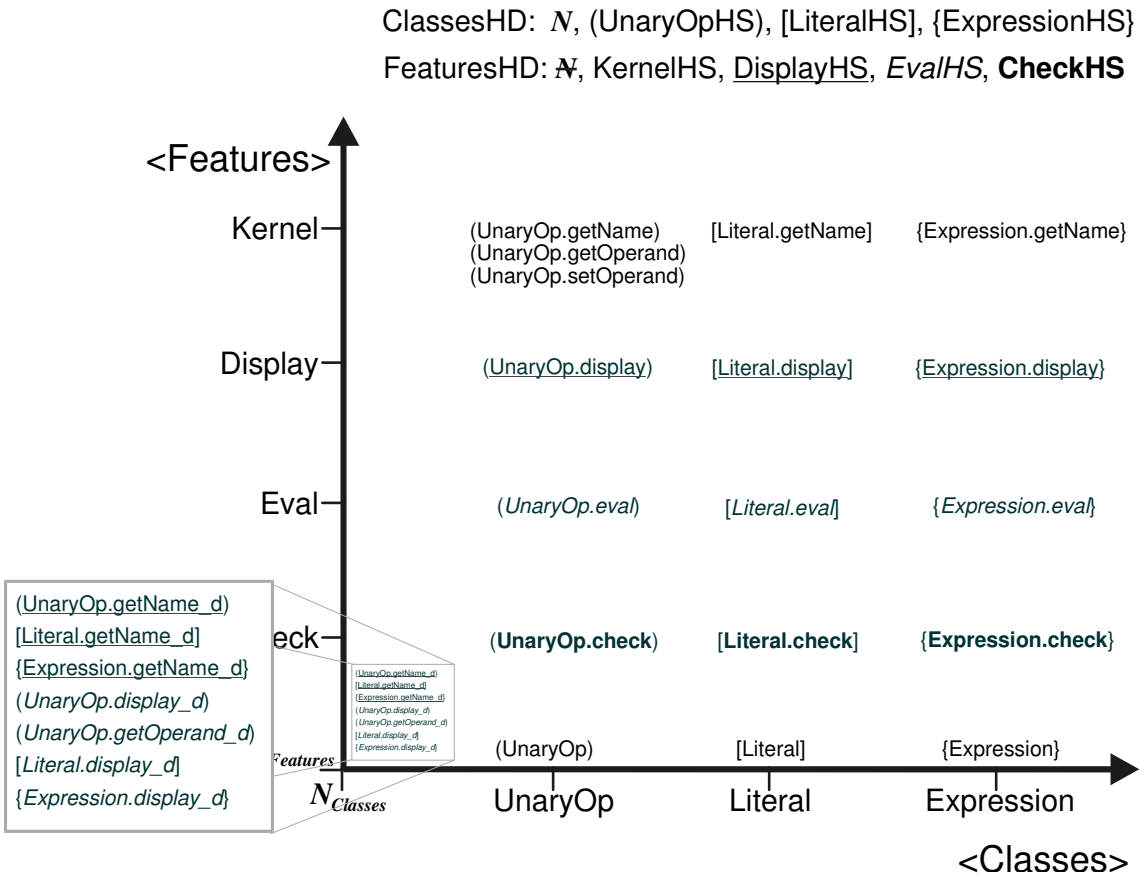

Abbildung 10 Belangmatrix der SEU mit Hyperslice-Dimensionen (Ausschnitt)
Die Darstellung entspricht Abbildung 9, zusätzlich zu den Dimensionen <Features> und <Classes> sind hier auch die Hyperslice-Dimensionen <FeaturesHD> und <ClassesHD> dargestellt. Die Zugehörigkeit zu den Belangen der Hyperslice-Dimension <FeaturesHD> wird durch Textauszeichnungen dargestellt: N, KernelHS, DisplayHS, *EvalHS*, **CheckHS**. Die Zugehörigkeit zu den Belangen der Hyperspace-Dimension <ClassesHD> wird durch Klammerung dargestellt: N, (UnaryOpHS), [LiteralHS], {ExpressionHS}. Deklarative Grundelemente sind durch das Suffix „_d“ gekennzeichnet.

zugeordnet und in allen anderen Dimensionen dem Nullbelang. Abbildung 10 zeigt die erweiterte Belangmatrix in der graphischen Darstellung.

4.3.4.1. Integrieren der Hyperslices zu Hypermodules

Die im vorangegangenen Abschnitt definierten Hyperslices können nun schrittweise zu Hypermodules zusammengefasst werden bis man schließlich ein Hypermodule hat, welches das gesamte System integriert. Dieses muss dann die Vollständigkeitseigenschaft erfüllen, d.h. es darf keine offenen Referenzen mehr beinhalten. (Alle deklarativen Grundelemente müssen gebunden sein.)

In der Beispielanwendung reicht ein einzelnes Hypermodule für das Gesamtsystem aus. Dieses Hypermodule integriert die für das System gewünschten Belange. Dazu muss die Menge der entsprechenden Hyperslices angegeben werden sowie ein Regelsatz, der beschreibt, wie die Hyperslices zu integrieren sind. In Hyper/J sähe das entsprechende Hypermodule Specification File (HMF) (siehe 4.2.5) z.B. folgendermaßen aus:

```
hypermodule SEU
  hyperslices:
    Feature.Kernel, Feature.Display, Feature.Check, Feature.Eval;

  relationships:
    mergeByName

end hypermodule;
```

Die generelle `mergeByName` Kompositionsstrategie legt fest, dass zwei oder mehrere Grundelemente korrespondieren, wenn sie über identische Bezeichner und Signaturen verfügen. Sie werden dann durch eine `merge` Kombination zu einem gemeinsamen Grundelement zusammengefasst. Durch weitere Korrespondenz- und Kombinationsregeln ließen sich Ausnahmen von der generellen Kompositionsstrategie definieren, was hier jedoch nicht benötigt wird.

4.3.5. Evolution der SEU: Hinzufügen eines Aspektes

Angenommen, die SEU sei bereits eine Weile im Einsatz. Einige Anwender, die das System in der universitären Lehre einsetzen, wünschen sich nun eine Möglichkeit der Ablaufverfolgung, um die Vorgänge in der SEU beim Auswerten von Ausdrücken besser visualisieren zu können. Zu den bisherigen Anforderungen (siehe 4.3.1) kommt nun also ein weiteres Werkzeug hinzu:

Logging Tool: Protokolliert das Betreten und Verlassen der einzelnen Methoden (Programmfluss) auf der Standardausgabe.

Logging ist ein typischer querschneidender Belang, also ein Aspekt (im technischen Sinne). Im Folgenden möchte ich nun kurz zeigen, wie sich ein solcher Aspekt mit Hyper/J zu der bestehenden SEU hinzufügen lässt.

4.3.5.1. Wiederverwendung eines bestehenden Logging Moduls

Nun ist Ablaufverfolgung eine vergleichsweise häufig benötigte Funktionalität, so dass es aus einem anderen Projekt bereits ein bestehendes Modul zum Loggen von Methodenaufrufen gibt. Abbildung 11 zeigt die Struktur des vorhandenen Logging-Moduls, das nun wieder verwendet werden soll und dazu in die SEU zu integrieren ist.

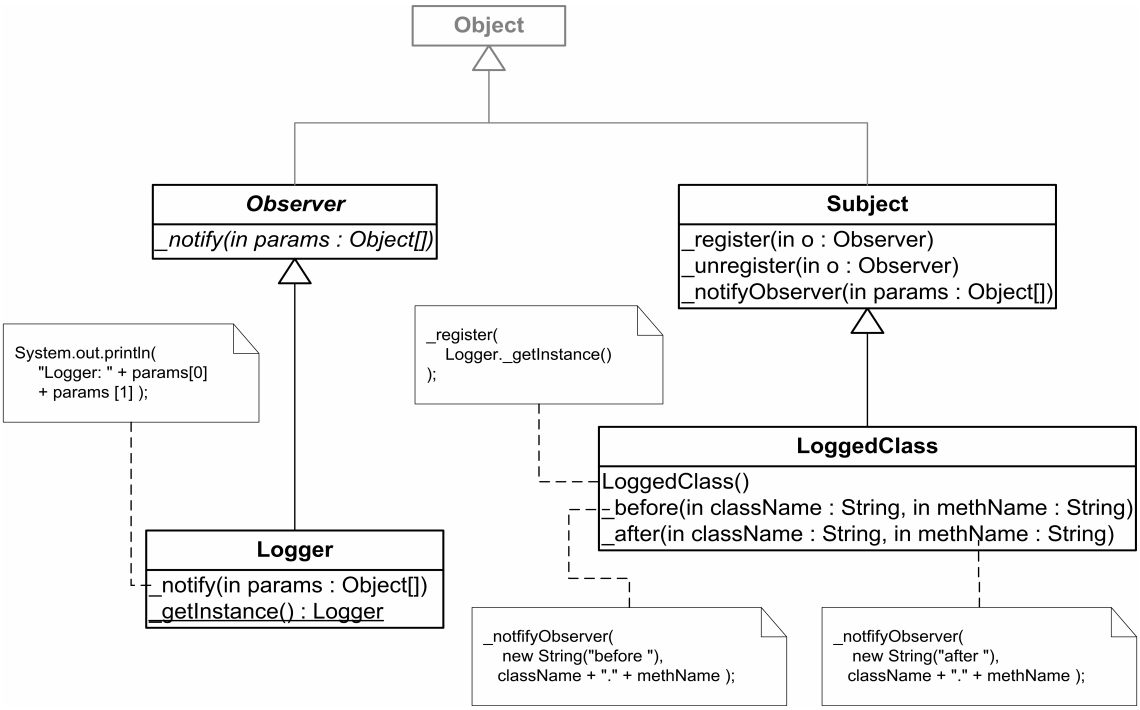

Abbildung 11 Klassendiagramm des Logging Moduls
Die Klasse `Logger` stellt in Form eines Singleton die eigentliche Ausgabe-Funktionalität bereit. `Logger` beobachtet mit Hilfe des Observer-Musters die zu protokollierenden Instanzen von `LoggedClass`. Bei der Erzeugung einer Instanz von LoggedClass, wird durch den Konstruktor die Verbindung zum Logger-Singleton hergestellt. Die wesentlichen Bestandteile der Implementierung sind in Form von Java Pseudocode annotiert. Die gemeinsame Oberklasse `Object` ist nur der Vollständigkeit halber skizziert.
(Abbildung angelehnt an [TO99])

4.3.5.2. Integration des Logging-Moduls in den bestehenden Hyperspace

Um Logging als einen eigenständigen Belang in die SEU aufnehmen zu können, sind zunächst erst mal die verwendeten Grundelemente in den Hyperspace aufzunehmen. Das HSF aus Abschnitt 4.3.3 wird dazu entsprechend erweitert:[22]

```
hyperspace SEUDemo
  composable class SEUDemo.Expression
  composable class SEUDemo.Literal
  composable class SEUDemo.UnaryOp
  composable class util.Logging.*
```

Die Logging-Funktionalität soll als ein eigener Belang dargestellt werden können. Dazu wird ein zusätzliches CMF erstellt:

```
package util.Logging    : Features.Logging
```

[22] Der Einfachheit halber erspare ich mir hier die bisher verwendete Mengendarstellung der Grundelemente und Belange sowie die graphische Darstellung der Belangmatrix. Stattdessen verwende ich nur noch die kompakteren Hyper/J Notationen.

Die Logging-Funktionalität wird hier also als neuer Belang *Logging* in die Dimension *Features* eingetragen.[23]

Schließlich ist noch ein Hypermodule für unsere SEU mit Logging-Funktionalität zu erstellen. Da nur sich nur bestimmte Anwender die Logging-Funktionalität gewünscht haben, erstellen wir eine Variante der SEU. Dazu wird ein eigenes Hypermodule definiert, das im Vergleich zu dem in Abschnitt 4.3.4 verwendeten HMF zusätzlich das Logging einbindet:

```
hypermodule SEU_with_logging
  hyperslices:
    Feature.Kernel, Feature.Display, Feature.Check, Feature.Eval,
    Feature.Logging;

  relationships:
    mergeByName
    bracket "{~_}*" with
      before Feature.Logging.LoggedClass._before($ClassName, $OperationName),
      after Feature.Logging.LoggedClass._after(($ClassName, $OperationName);

end hypermodule;
```

Wesentliches Element ist hier die Verwendung der `bracket` Kombinationsregel. Jede Methode, die auf den regulären Ausdruck {~_}* passt, wird mit den Methoden `LoggedClass._before` und `LoggedClass._after` „geklammert“. Dabei wird außerdem jeweils der Klassen- und Methodenname der geklammerten Methode mit übergeben.

Der reguläre Ausdruck {~_}* passt dabei auf alle Methoden in allen Klassen, deren Namen nicht mit einem Unterstrich beginnt. Auf diese Weise wird hier verhindert, dass die Methoden des Java Packets `util.logging` ebenfalls geklammert werden.

[23] Es lässt sich sicherlich trefflich darüber streiten, ob Logging in der Features-Dimension wirklich gut aufgehoben ist, da es sich seinem Wesen nach durchaus von den eher funktionalen Features unterscheidet. Letztlich scheint es mir aber übertrieben hier nur für das Logging eine eigene Dimension zu erstellen.

Kapitel 5

Erweiterung des Hyperspace-Modells

Das Hyperspace-Modell und seine Realisierung in Hyper/J zeigen sehr schön das Potential, das in der Idee des mehrdimensionalen Trennens der Belange steckt. Hyper/J erlaubt durch die Definition von Belangen und Hypermodules eine feingranulare Separierung und Integration von Java-Code. Anstelle der eher technischen Gliederung des Codes in *Pakete*, *Klassen* und *Methoden* tritt eine Gliederung des Codes in *Belange* und *Hypermodule*. Diese Form der Gliederung ist deutlich näher am Anwendungskontext, sie ist „intuitiver". Damit erleichtert sie sowohl das Verständnis des Systems als auch seine Wiederverwendung und Erweiterbarkeit.

Die grundlegenden Ideen des multidimensionalen Trennens der Belange bieten jedoch auch weit über Hyper/J hinaus einen viel versprechenden Ansatz – sowohl für die Praxis als auch in der Theorie. Der Gewinn für die Praxis deutet sich mit Hyper/J bereits an, er wird ungleich höher ausfallen, wenn es gelingt ähnliche Konzepte auch in den frühen Phasen des Entwicklungsprozesses zu verwenden. In der Theorie bildet multidimensionales Trennen der Belange einen aussichtsreichen Rahmen für das Verstehen und Entwickeln von Artefaktsprachen sowie für eine tiefere Durchdringung der komplexen Zusammenhänge in Softwaresystemen.

In diesem Kapitel soll nun untersucht werden, wie das Hyperspace-Modell für derartige Zwecke verwendet werden kann. Der Schwerpunkt liegt dabei auf einer Verwendung in den frühen Phasen des Softwareentwicklungsprozesses. Dazu werden zunächst die Grenzen und impliziten Annahmen des bestehenden Modells aufgezeigt. Anschließend wird, ausgehend vom Hyper/J-Prozess, das Modell erweitert und verallgemeinert. Der verallgemeinerte Hyperspace-Prozess wird dann anhand eines Beispielszenarios durchgespielt. Im abschließenden Diskussionsteil erfolgt noch ein Vergleich der beiden Ansätze sowie eine Einordnung zur aspektorientierten Programmierung.

5.1. Die Grenzen von Hyper/J

Hyper/J beschränkt sich auf nur einen Typ von Artefakten: Java-Code. Es unterstützt somit auch nur die späten Phasen des Softwareentwicklungsprozesses wie Implementie-

rung und Wartung. Trotz aller Vorzüge ist Hyper/J also noch weit von dem entfernt, was in Kapitel 2.1.4 als *multidimensionales Trennen der Belange* eingeführt wurde:

> OSSHER und TARR definieren den Begriff *multidimensionales Trennen der Belange* als Trennung der Belange [OT00]
>
> - mit beliebig vielen Dimensionen von Belangen,
> - *simultanem* Trennen der Belange entlang dieser Dimensionen,
> - der Möglichkeit, neue Belange und Dimensionen *dynamisch* zu handhaben, so wie sie im Softwareentwicklungsprozess erscheinen, und
> - überlappenden und interagierenden Belangen. (Es ist verlockend sich Belange als unabhängig oder „orthogonal" vorzustellen, aber derartige Belange kommen in der Praxis nur selten vor.)

Die Kernidee von multidimensionalem Trennen der Belange ist demnach die Unterstützung des *gesamten* Softwareentwicklungsprozesses. Die in den jeweiligen Phasen betrachteten Belange und erstellten Artefakte sollen entlang ihrer Betrachtungsdimensionen, die oftmals verschiedenen Artefaktsprachen entsprechen, in ein konsistentes Gesamtmodell integriert werden können.

Damit dürfte klar sein, dass Hyper/J mit seiner Beschränkung auf einen einzigen Typ von Artefakten nur einen ersten Schritt darstellt. Und wie bei einer näheren Betrachtung ersichtlich wird, unterliegt auch die formale Basis von Hyper/J, das in Kapitel 4.1 vorgestellte Hyperspace-Modell, einer Reihe von Einschränkungen, die eine Verwendung mit beliebigen Artefaktsprachen schwierig machen. Überlappende und interagierende Belange und Dimensionen sowie hierarchische Beziehungen zwischen Belangen lassen sich damit nicht darstellen. Gerade bei den Artefaktsprachen, die in den frühen Phasen des Softwareentwicklungsprozesses zum Einsatz kommen, sind derartige Beziehungen zwischen Artefakten jedoch nicht ungewöhnlich.

Im Folgenden soll nun deshalb untersucht werden, ob und wie sich diese Einschränkungen überwinden lassen. Der Schwerpunkt der Betrachtungen liegt dabei auf den Schwierigkeiten, notwendigen Änderungen und Erweiterungen für eine simultane Verwendung von mehreren Artefaktsprachen.

5.2. Problempunkte

Bei den Überlegungen zur Integration von Artefakten anderer Artefaktsprachen in einen Hyperspace stößt man auf einige Fragen und Probleme, denen offensichtlich eine Schlüsselrolle bei der Erweiterung und Verallgemeinerung zukommt. Diese Punkte sollen nun hier kurz angesprochen und dann im weiteren Verlauf dieses Kapitels ausführlicher diskutiert werden:

1. *Der Hyperspace als geometrischer Raum*. Im Hyperspace-Modell handelt es sich bei der Belangmatrix um einen geometrischen Raum, dessen Achsen die Belangdimensionen und dessen Inhalt die Grundelemente sind. Jedes Grundelement ist

damit genau einem Belang in jeder Dimension zugeordnet. Ist diese Forderung für die Verwendung höherer Artefaktsprachen praktikabel?

2. *Übertragung der Begriffe.* Im Hyperspace-Ansatz werden eine Reihe von speziellen Begriffen definiert: Belang, Hyperslice, deklaratives Element, Hypermodule, Korrespondenz, Kombination. Wie sind diese Begriffe bei einer Übertragung auf andere Artefaktsprachen zu verstehen?
3. *Trennung zwischen Grundelementen und Belangen*. Das Hyperspace-Modell unternimmt eine strenge Trennung zwischen Belangen und Grundelementen: Die Menge der Grundelemente und die Menge der Belange sind disjunkt, Belange enthalten ausschließlich Grundelemente. Soll diese scharfe Trennung aufrechterhalten werden?
4. *Grundelemente aus mehreren Belangdimensionen*. Hyper/J verwendet nur einen Typ von Artefakten und damit nur Grundelemente einer Belangdimension. Was ist bei einer Erweiterung zur Integration unterschiedlicher Artefakttypen zu beachten?

5.2.1. Der Hyperspace als geometrischer Raum

HAROLD OSSHER und PERI TARR verstehen unter dem Begriff des Belangraums einen *geometrischen* Raum. Dieses legt schon die von ihnen verwendete Terminologie wie *Hyperspace, Concern Matrix, Hyperslice* etc. nahe. Formal festgelegt ist die Raumeigenschaft durch die Definition der Belangmatrix (Definition 1 auf S.43). Jedes Grundelement ist an genau einer Position in der Matrix angeordnet. Es gehört damit zu genau einem Belang in jeder Belangdimension.

Die Raumeigenschaft bietet zweifellos eine Reihe von Vorteilen. Sie erleichtert das formale und algorithmische Arbeiten mit Hyperspaces. Vor allem aber fördert sie eine saubere Trennung der Belange, zwingt sie doch dazu, dass jedes Grundelement (als atomarer Baustein eines Belangraumes) genau *einem* Belang (in jeder Dimension) zugeordnet wird. Fällt diese Zuordnung schwer, so ist das ein Hinweis darauf, dass die in Frage kommenden Belange noch nicht hinreichend voneinander abgegrenzt sind.

Eng verbunden mit der Raumeigenschaft sind im Hyperspace-Ansatz die *deklarativen Grundelemente*. Ein Grundelement, wie z.B. eine Methode, wird üblicherweise für die Implementierung verschiedenster Belange benutzt. Es kann jedoch nur an einem Punkt im Belangraum abgebildet sein. Daher gibt es die deklarativen Grundelemente, die eine Verwendungsbeziehung zu einem solchen Grundelement darstellen. Anstelle des Grundelementes selber wird also eine Deklaration, ein Platzhalter, eingebunden, die später an ein konkretes Grundelement gebunden wird.

Diese Trennung in Deklaration und Definition entspricht dem Charakter formaler Sprachen wie den üblichen Programmiersprachen, also insbesondere auch dem Hyper/J zugrunde liegenden Java.[24] Sie entspricht jedoch *nicht* der Art und Weise, wie mit den

[24] Wobei Kritiker zu Recht behaupten, dass diese Trennung gerade durch die in Java eingeführte in-place Definition der Methoden wieder aufgeweicht wurde. Technisch handelt es sich dabei jedoch nur um eine

weniger formalen Sprachen der früheren Phasen, wie z.B. den UML-Sprachen, gearbeitet wird. Hier werden Entitäten nicht unbedingt durch Deklarationen eingeführt und explizit definiert, vielmehr erfolgen Deklaration *und* Definition oft implizit durch „Verwendung“ oder „Nennung“ der Entität.

Eine schönes Beispiel dafür sind Akteure in UML Anwendungsfalldiagrammen. Anwendungsfalldiagramme bestehen aus Anwendungsfällen und Akteuren. Jeder einzelne Anwendungsfall stellt einen Belang dar; ein einzelner Akteur ist üblicherweise an mehreren Anwendungsfällen beteiligt.[25] Doch welchem der verschiedenen Belange (Anwendungsfälle) soll man den Akteur nun zuschlagen? Im Gegensatz zu z.B. einer Methode, die im Kontext genau einer Klasse definiert wird, steht ein Akteur „alleine“ da, er wird durch simple Nennung und Verwendung in das Modell eingeführt und ist allen Belangen (Anwendungsfällen), an denen er beteiligt ist, in gleichem Maße zuzuordnen.

An dieser Stelle wird deutlich, dass es die Raumeigenschaft schwierig macht Artefakte beliebiger Artefaktsprachen in einen Hyperspace zu integrieren. Für eine Verwendung der Hyperspace-Idee über den gesamten Softwareentwicklungszyklus ist jedoch genau dieses erforderlich![26]

Ich habe mich daher entschlossen für die weiteren Überlegungen die Raumeigenschaft aufzugeben. Die Zuordnungsrelation von Grundelementen zu Belangen ist damit nicht mehr rechtseindeutig, vielmehr kann ein Grundelement auch mehreren Belangen einer Belangdimension zugeordnet werden.

5.2.2. Übertragung der Begriffe

Bei der Einführung des Hyperspace-Modells in Kapitel 4 wurden einige spezielle Begriffe definiert, wie *deklaratives Element*, *Hyperslice* oder *Hypermodule*, die in dem Modell eine große Rolle spielen. Ich habe mich bemüht, diese Begriffe sinnvoll in das erweiterte Modell zu übertragen, im Diskussionsteil am Ende dieses Kapitels (Abschnitt 5.4.2) wird dazu eine detaillierte Gegenüberstellung vorgenommen. Für das Textverständnis kann es jedoch hilfreich sein, einiges davon schon hier einmal kurz anzureißen. Schließlich steht

---

Schreibabkürzung für den Programmierer. Intern unterscheidet Java sehr wohl zwischen Deklaration und Definition einer Methode.

[25] Die hier vorgenommene Aufteilung in Belange (Anwendungsfälle) und Grundelemente (Anwendungsfälle und Akteure) ist bewusst knapp gehalten. Warum sie sinnvoll ist, wird in Abschnitt 5.3 ab Seite 74 noch genauer diskutiert.

[26] Auf der AOSD 2002 hatte ich Gelegenheit, das Problem der Raumeigenschaft kurz mit PERI TARR zu diskutieren. Der Wunsch nach einer Aufweichung würde oft an sie herangetragen, sie stehe dem jedoch eher kritisch gegenüber, da in eigentlich allen Fällen, bei denen dieser Wunsch laut wurde, die Leute schlichtweg ihre Belange nicht sauber getrennt hätten. Das von mir angeführte Beispiel der Use-Case Diagramme fand sie jedoch auf den ersten Blick recht überzeugend, sie bezeichnete es als „das erste überzeugende Beispiel überhaupt“. Weiterhin teilte sie mir mit, dass für die nächste Version des Hyperspace-Modells vorgesehen sei, die Modellierung von Beziehungen zwischen Grundelementen zu ermöglichen und in diesem Rahmen durchaus darüber nachgedacht wird, Identitätsbeziehungen zwischen Grundelementen zu erlauben. Damit würde die Raumeigenschaft faktisch aufgegeben, das formale Modell wäre jedoch weiterhin das eines geometrischen Raums.

ein Teil der Begriffe in direktem Zusammenhang mit der Raumeigenschaft des Hyperspace-Modells, die hier ja bewusst aufgegeben wird:

Die *deklarativen Elemente* werden nicht mehr benötigt. Auch *Nullbelange* sind nicht mehr erforderlich. Mit den deklarativen Elementen fallen außerdem die Begriffe der *deklarativen Vollständigkeit* und des darüber definierten *Hyperslice*. Ohne Hyperslices werden auch keine speziellen *Hyperslice-Dimensionen* mehr benötigt, Belange werden im erweiterten Modell ohne den Umweg über die Hyperslices direkt in *Hypermodules* kombiniert. Schon hier dürfte deutlich werden, dass die Aufgabe der Raumeigenschaft das Modell insgesamt eher vereinfacht als komplizierter macht.

5.2.3. Trennung zwischen Grundelementen und Belangen

Das Hyperspace-Modell unternimmt eine strenge Trennung zwischen Belangen und Grundelementen: Die Menge der Grundelemente und die Menge der Belange wurden in Kapitel 4.1.2, Definition 1 (Seite 43) als disjunkt festgelegt.

Prinzipiell ist diese strenge Trennung durchaus sinnvoll. Belange und Grundelemente sind von ihrem Wesen her grundverschieden: Grundelemente sind *Entitäten* in Artefakten, sie sind konkrete Instanzen der Elementtypen einer Modellierungssprache. Belange hingegen sind *intentional*, sie sind die kognitiven Konstrukte eines beteiligten Menschen und damit keine Entitäten im Sinne einer Artefaktsprache oder eines formalen Modells.[27]

Nun ist es Sinn und Zweck einer Artefaktsprache *Belange* zu beschreiben. In konkreten Artefakten gibt es daher sinnvollerweise Entitäten, die für einen bestimmten Belang stehen. Eine *Klasse* in einem Klassendiagramm steht z.B. für den entsprechenden Belang in der Belangdimension <Klassen>, ein *Anwendungsfall* für den beschriebenen Belang in der Dimension <Anwendungsfälle>. Nichtsdestotrotz handelt es dabei weiterhin um syntaktische Konstrukte der verwendeten Modellierungssprache – und damit um *Grundelemente*. Derartige Grundelemente haben jedoch eine so enge semantische Kopplung zu den Belangen, die sie beschreiben, dass sie kaum noch von diesen zu unterscheiden sind. Sie *koinzidieren* mit dem entsprechenden Belang. Ein solches Grundelement kann daher als Repräsentant des gesamten Belangs aufgefasst werden und wird *repräsentatives Grundelement* genannt.

[27] Vergleiche auch Diskussion des Belangbegriffs in Kapitel 2.

5.2.4. Grundelemente aus mehreren Belangdimensionen

Es gibt einen interessanten Aspekt (i.w.S.) von Hyper/J, der in der Literatur nirgendwo angesprochen wird und vielleicht den Entwicklern des Hyperspace-Ansatzes und von Hyper/J gar nicht mal bewusst ist: Hyper/J unterscheidet *zwei verschiedene Arten von Belangdimensionen*. Ich möchte diese im Folgenden *artefaktbasierende* und *selektierende* Belangdimensionen nennen.[28]

- Eine *artefaktbasierende Belangdimension* ist eine Belangdimension, über die Grundelemente in den Hyperspace eingebracht werden. Grundelemente sind syntaktische Konstrukte einer Artefaktsprache; wie der Name schon sagt, basieren die Belange artefaktbasierender Belangdimensionen demnach auf konkreten Artefakten. Sie definieren die Menge der im Hyperspace enthaltenen Grundelemente. Die Belange einer artefaktbasierenden Belangdimension haben sinnvollerweise jeweils ein zugeordnetes repräsentatives Grundelement.
- Eine *selektierende Belangdimension* fügt dem Hyperspace hingegen keine weiteren Grundelemente hinzu. Sie stellt quasi nur eine *weitere Sicht*, Datenbanker würden sagen einen *View*, auf die Menge der Grundelemente bereit. Sie dient einer alternativen Trennung der Belange.

Natürlich ist jede artefaktbasierende Belangdimension auch eine selektierende, da sie ja ebenfalls die Grundelemente einzelnen Belangen zuordnet.

5.2.4.1. Die artefaktbasierende Belangdimension in Hyper/J

Es wurde bereits mehrfach angemerkt, dass Hyper/J mit Java .class Dateien nur einen einzigen Typ von Artefakten unterstützt. Der eigentliche Punkt ist jedoch, dass Hyper/J damit auch nur *eine artefaktbasierende Belangdimension* erlaubt: Die Dimension <Classes>, die zudem implizit vorhanden ist und von Hyper/J automatisch erstellt wird.

Dieses ist für den Zweck von Hyper/J auch ausreichend, da in einem Java-Programm jedes Element zu genau einer Klasse gehört. Anders ausgedrückt: Jedes Java-Programm lässt sich damit in den Hyperspace abbilden. Die <Classes> Dimension ist dabei das Rückgrat des Hyperspace, sie ist unverzichtbar. Daher bezeichne ich im Folgenden, HAROLD OSSHER und PERI TARR mögen es mir verzeihen, das Hyper/J Modell als eindimensional (nur eine artefaktbasierende Belangdimension).

5.2.4.2. Der Hyperspace Prozess von Hyper/J

Um die zentrale Bedeutung der artefaktbasierenden Dimension besser zu verstehen, ist es hilfreich sich noch einmal zu verdeutlichen, wie die Abläufe beim multidimensionalen Trennen der Belange mit Hyper/J eigentlich sind: Hyper/J bekommt als Eingabe ein gültiges Java-Modell (eine Menge von .class Dateien), aus dessen Grundelementen durch

[28] An und für sich verwendet Hyper/J mit den Hyperslice-Dimensionen intern sogar noch eine dritte Art von Dimensionen. Diese sind für die weitere Betrachtung jedoch irrelevant.

Selektion (Bildung von Belangen) eine Teilmenge ausgewählt wird. Anhand eines gegebenen Regelsatzes (Hypermodule) transformiert Hyper/J diese Auswahl dann wieder in ein gültiges Java-Modell. Auf das Wesentliche reduziert besteht dieser Prozess also aus drei Schritten:[29]

1. *Integration*: Einbinden eines gültigen Java-Modells in den Hyperspace. Das Eingabemodell wird dabei zerlegt, es wird in einzelne Klassen auf der artefaktbasierenden Belangdimension <Classes> mit jeweils zugehörigen Grundelementen aufgeteilt.
2. *Selektion*: Je nach Bedarf werden selektierende Belangdimensionen erzeugt und die Grundelemente den Belangen der selektierenden Belangdimensionen zugeordnet.
3. *Kombination*: Auswahl einer Menge zu kombinierender Belange (Hypermodule). Die dadurch ausgewählte Teilmenge der Grundelemente stellt eventuell kein gültiges Java-Modell dar. In einem Transformationsprozess werden die Grundelemente nun wieder zu einem gültigen Java-Modell, dem Ausgabemodell, kombiniert.

Den im Hyper/J-Prozess verarbeiteten Daten liegt implizit ein *Metamodell* zugrunde: Das Metamodell von Java-Programmen, kurz Java-Metamodell. Sowohl die Eingabe als auch die Ausgabe von Hyper/J sind Instanzen dieses Metamodells. Die im Hyperspace angeordneten Grundelemente (wie Klassen oder Operationen) sind Instanzen der im Java-Metamodell definierten Elementtypen.

Wie oben ausgeführt, werden Grundelemente in einen Hyperspace ausschließlich über die artefaktbasierenden Belangdimensionen eingebracht. Das dem Hyperspace zugrunde liegende Metamodell entspricht daher dem Metamodell der artefaktbasierenden Belangdimensionen, hier dem Metamodell der <Classes> Dimension.[30]

5.2.4.3. Mehrere artefaktbasierende Belangdimensionen

Für eine Erweiterung des Modells zur Integration von Artefakten verschiedener Artefaktsprachen muss man sich nun zwangsläufig der Frage stellen, wie mit der Existenz mehrerer artefaktbasierender Belangdimensionen umzugehen ist.

Zunächst bedeutet das Vorhandensein mehrerer artefaktbasierender Belangdimensionen, dass wir auch mehrere unterschiedliche Metamodelle berücksichtigen müssen. Es ist jedoch ausgesprochen unwahrscheinlich, dass die verschiedenen Belangdimensionen (und damit die entsprechenden Metamodelle, Modelle und Grundelemente) vollständig isoliert zueinander stehen und in keiner Weise aufeinander Bezug nehmen. Vielmehr gibt es hier eine Kopplung zwischen den Grundelementtypen verschiedener Metamodelle. Der

---

[29] Wobei die hier verwendete sequentielle Darstellung ausschließlich der Erläuterung dient und nicht im Sinne eines Vorgehensmodells aufgefasst werden darf. Es zeichnet Hyper/J ja gerade aus, dass man diese Schritte in nahezu beliebiger Reihenfolge immer wieder iterieren kann.

[30] Genau genommen sind es natürlich nicht die Metamodelle der artefaktbasierenden Belangdimensionen, sondern die Metamodelle der den artefaktbasierenden Belangdimensionen zugrunde liegenden Artefaktsprachen.

Elementtyp „Operation“ des Klassendiagramms steht in enger Verbindung zum Elementtyp „Aktivität“ des Aktivitätendiagramms usw.

Diese Beziehungen gelten dementsprechend natürlich auch für die konkreten *Instanzen* der Elementtypen – für die Grundelemente. Neben der expliziten Zuordnung von Grundelementen zu Belangen haben wir also nun zu berücksichtigen, dass es auch Beziehungen *zwischen Grundelementen* aus *verschiedenen artefaktbasierenden* Belangdimensionen gibt.

Natürlich stehen auch im eindimensionalen Fall von Hyper/J die Grundelemente miteinander in Beziehung. Eine Operation gehört beispielsweise immer zu genau einer Klasse. Die möglichen Beziehungen gehen hier jedoch eindeutig aus dem Metamodell, die konkreten aus den Modellen hervor – das ist schließlich Sinn und Zweck der Modelle. Bei der Integration von Grundelementen aus verschiedenen Belangdimensionen verlassen wir jedoch die Domäne der einzelnen Modelle und Metamodelle. Im Unterschied zum eindimensionalen Fall sind sowohl die möglichen Beziehungen zwischen verschiedenen Elementtypen (auf der Metamodellebene) als auch die konkreten Beziehungen zwischen Grundelementen (auf der Modellebene) hier nicht mehr eindeutig zu ermitteln.

Es kann daher kaum sinnvoll sein mehrere unterschiedliche Metamodelle parallel in einem Hyperspace zu verwenden. Vielmehr ist es vor einer Integration der Modelle in den Hyperspace zunächst erforderlich, die verwendeten Metamodelle zu einem gemeinsamen Metamodell zu integrieren. Dieses *integrierte Metamodell* ist dann das zugrunde liegende Metamodell des Hyperspace. Es definiert die möglichen Beziehungen zwischen den Grundelementen. Wie im eindimensionalen Fall entspricht es dem Metamodell der artefaktbasierenden Belangdimensionen.

Neben dem integrierten Metamodell werden noch weitere Hilfsmittel gebraucht, um die konkreten Beziehungen zwischen Grundelementen verschiedener Modelle zu beschreiben, da sich diese ja hier nicht eindeutig aus den beteiligten Modellen ermitteln lassen.

5.2.4.4. Der Hyperspace Prozess mit mehreren artefaktbasierenden Belangdimensionen

Anhand des oben Gesagten lässt sich nun skizzieren, wie der erweiterte Hyperspace-Prozess mit mehreren artefaktbasierenden Belangdimensionen aussieht. Auch hier handelt es sich, ausgehend von einem leeren Hyperspace, um einen iterativen Prozess, dessen einzelne Schritte bei der Arbeit mit dem Modell immer wieder und nach Bedarf durchgeführt werden:

1. *Integration*: Einbinden eines Modells in den Hyperspace. Das *Eingabemodell* muss ein gültiges Modell eines zugehörigen Metamodells sein.
    a) Integration des zugehörigen Metamodells in das Metamodell des Hyperspace.
    b) Hinzufügen oder Auswählen einer geeigneten artefaktbasierenden Belangdimension.

   c) Einfügen der Belange und Grundelemente. Die identifizierten Belange werden in die artefaktbasierende Belangdimension eingefügt. Den einzelnen Belangen werden jeweils die sie beschreibenden Grundelemente zugeordnet.
   d) Definieren der Beziehungen zwischen den neuen Grundelementen und bereits vorhandenen Grundelementen anderer artefaktbasierender Belangdimensionen.
2. *Selektion*: Je nach Bedarf werden selektierende Belangdimensionen erzeugt und die Grundelemente den Belangen der selektierenden Belangdimensionen zugeordnet.
3. *Kombination*: Auswahl einer Menge zu kombinierender Belange (Hypermodule). Die damit ausgewählte Teilmenge der Grundelemente stellt eventuell kein gültiges Modell des integrierten Metamodells dar. In diesem Fall werden die Grundelemente in einem Transformationsprozess zu einem gültigen Modell, dem *Ausgabemodell*, kombiniert.
4. *Projektion*: Das Ausgabemodell ist ein gültiges Modell im Sinne des *integrierten* Metamodells. Es ist damit nicht in einer der üblichen Artefaktsprachen darstellbar. Zur Weiterverarbeitung kann es daher erforderlich sein, dass integrierte Modell wieder abzubilden auf Einzelmodelle der verwendeten Artefaktsprachen.

Gegenüber dem Hyper/J-Prozess sind also einige Schritte hinzugekommen, welche die Behandlung zusätzlicher artefaktbasierender Belangdimensionen und Metamodelle betreffen. Dieses sind zunächst die Unterschritte von Schritt 1, die so im Hyper/J-Prozess naturgemäß nicht vorkommen. Außerdem neu ist der Schritt 4, in dem bei Bedarf die in Schritt 1 getätigte Integration der Modelle zu einem gemeinsamen Modell wieder rückgängig gemacht wird.

Für das Modell gehe ich in dieser Arbeit von der Annahme aus, dass ein konkretes Artefakt immer nur Belange genau einer Belangdimension modelliert. In der Realität gibt es sicherlich auch Sprachen, die erlauben gleichzeitig Belange mehrerer Dimensionen zu beschreiben. In einem solchen Fall haben wir es wahrscheinlich mit einer nicht ideal entworfenen Sprache zu tun. Dennoch wäre für eine Anwendung des Modells in der Praxis noch zu untersuchen, inwiefern derartige Sprachen berücksichtigt werden können.

Es dürfte offensichtlich sein, dass der hier dargestellte Prozess eine echte Generalisierung des in Abschnitt 5.2.4.2 beschriebenen Hyper/J-Prozesses darstellt.

5.2.5. Zusammenfassung

Hyper/J verwendet nur einen Typ von Artefakten und damit nur Grundelemente einer Artefaktsprache. Ziel des erweiterten Hyperspace-Modells ist die Unterstützung von beliebigen und mehreren Sprachen, insbesondere denen, die in den frühen Phasen wie Spezifikation und Entwurf zum Einsatz kommen. Dazu scheint es erforderlich, die Raumeigenschaft des Hyperspace-Modells aufzugeben. Ferner zeigt sich, dass es zwei verschiedene Arten von Belangdimensionen zu berücksichtigen gilt. Artefaktbasierende Belangdimensionen basieren auf konkreten Artefakten und fügen dem Hyperspace neue

Grundelemente hinzu. Ihre Belange haben üblicherweise ein zugeordnetes repräsentatives Grundelement. Sie sind zu unterscheiden von den selektierenden Belangdimensionen, die nicht auf Artefakten basieren und zusätzliche Sichten auf das Softwaresystem bereitstellen.

Eine zentrale Rolle bei der Unterstützung mehrerer artefaktbasierender Belangdimensionen spielen Metamodelle. Vor einer Integration von Artefakten ist es erforderlich, die zugehörigen Metamodelle in das Metamodell des Hyperspace zu integrieren. Dieses ist im Falle von Hyper/J nicht notwendig, da dort nur mit einer artefaktbasierenden Dimension gearbeitet wird. Die Integration der Metamodelle und Modelle führt daher zu zusätzlichen Schritten im erweiterten Prozessmodell.

Im Folgenden soll nun das erweiterte Hyperspace-Prozessmodell einmal anhand eines Szenarios mit einem konkreten Beispiel durchgespielt werden. Als Artefakttypen kommen dabei zwei Diagrammtypen der UML zum Einsatz: Anwendungsfalldiagramme (use case diagrams) und Klassendiagramme (class diagrams). Diese gilt es entsprechend in Belange und Belangdimensionen zu zergliedern und in einen Hyperspace zu integrieren. Anschließend werden dem Hyperspace zwei selektierende Belangdimensionen hinzugefügt und gezeigt, wie mit dem Modell gearbeitet werden könnte.

5.3. Beispiel: Ein Inventarisierungssystem

Das hier verwendete Beispiel entstammt, mit leichten Modifikationen, der Diplomarbeit *„Ein Inventarisierungssystem für das Landesamt für Denkmalpflege – Bedarfsanalyse, Entwurf, prototypische Implementierung“* von BERNHARD GABLER [Gab02]. Die Arbeit beschreibt den Entwurf eines Softwaresystems zur Inventarisierung von Daten archäologischer Fundstücke (wie Tonscherben oder Werkzeuge), die häufig bei Grabungsarbeiten entdeckt werden. Die Sicherstellung, Lagerung und Inventarisierung derartiger Funde fällt in Deutschland in den Zuständigkeitsbereich einer Landesbehörde, die in ihrer Aufgabe durch das Softwaresystem unterstützt werden sollte.

Ich habe dieses Beispiel aus zwei Gründen gewählt: Zum einen benutzt GABLER in seiner Arbeit ein übliches Vorgehen beim Softwareentwurf und verwendet typische Tätigkeiten und Artefaktsprachen. Zum anderen handelt es sich einfach um einen real existierenden Entwurf. Das Beispiel, gleichwohl noch gut überschaubar, entbehrt damit der Trivialität und Glattheit, die mit rein akademischen Beispielen so oft einhergeht.

5.3.1. Ein paar Worte zum Thema Metamodelle

Wie im vorangegangenen Abschnitt dargelegt, spielen Metamodelle eine zentrale Rolle bei der Verallgemeinerung des Hyperspace-Ansatzes – und damit natürlich auch in diesem Beispiel. Aus diesem Grund scheinen mir ein paar Hinweise zu dem hier verwendeten Metamodell wichtig zu sein.

Metamodelle zu den Sprachen der UML und ihren Vorläufern gibt es bereits wie „Sand am Meer“. Dennoch habe ich für das Beispiel kein bestehendes Metamodell gewählt, sondern ebenfalls ein eigenes erstellt. Warum?

Bei der Diskussion um Metamodelle wird gerne übersehen, dass ein Metamodell in erster Linie ein *Modell* ist. Modelle sind *grundsätzlich intentional*, sie werden immer für ein bestimmtes Ziel erstellt, betonen die in diesem Rahmen relevanten Aspekte (i.w.S.) und vernachlässigen die anderen. Mit anderen Worten: Ein Metamodell ist immer zweckgebunden, es kann nicht *das* geeignete Metamodell für alle Zwecke geben.

Mein Ziel war es, ein minimales und einfaches Metamodell zu verwenden, um das Beispiel überschaubar zu halten. Dementsprechend ist das hier verwendete Metamodell gezielt auf das Beispiel zugeschnitten. Es enthält jeweils nur genau die Elemente, die in den konkreten Artefakten verwendet werden. So sind z.B. in Klassendiagrammen keine Methoden vorgesehen – sie spielen im Kontext des Beispiels keine Rolle. Das verwendete Metamodell ist also als Hilfsmittel zur Darstellung der Vorgehensweise zu verstehen. Es sollte nicht als umfassendes Metamodell der entsprechenden Artefaktsprachen aufgefasst werden.

5.3.2. GABLERS Vorgehensweise beim Entwurf

Beim Entwurf des Inventarisierungssystems ging GABLER wie folgt vor:

1. Durch Interviews mit den Mitarbeitern der Behörde wurden die internen Prozesse analysiert und als *Analysemodell* in textueller Form sowie als Use-Case Modell beschrieben.
2. Aus dem Analysemodell wurden diejenigen Prozesse (Anwendungsfälle) ausgewählt, die von dem Softwaresystem unterstützt werden sollen. Auf dieser Basis wurde ein EER *Datenmodell* in Form eines UML Klassendiagramms erstellt.
3. Ausgehend von dem Analysemodell wurden die gewählten Anwendungsfälle verfeinert und in einem weiteren Use-Case Modell, dem *Entwurfsmodell* beschrieben. Dieses Modell ist das zentrale Entwurfsdokument (Abbildung 12).
4. Die einzelnen Prozesse des Entwurfsmodells wurden dann textuell und mit Hilfe von *Aktivitätendiagrammen* spezifiziert.
5. Anhand der Spezifikation wurden die benötigten *Masken* sowie, ausgehend von dem Datenmodell, die benötigten *Datenbanktabellen* ermittelt.
6. Das Modell wurde prototypisch implementiert.

Dieses Vorgehen ist vom Grundsatz her sicherlich typisch für derartige Entwicklungsprojekte und bietet daher einen geeigneten Rahmen für unser Szenario. Da es uns um die simultane Verwendung verschiedener Artefaktsprachen geht, steigen wir hier ein bei Punkt 3 mit dem Entwurfsmodell. Dieses wird zunächst in den Hyperspace eingebunden. Anschließend wird dann das Datenmodell eingebunden.

5.3.3. Integration des Entwurfsmodells

Zentrales Artefakt des Entwurfs ist das in Abbildung 12 dargestellte und erläuterte Anwendungsfalldiagramm. Dieses stellt den Rahmen für die Spezifikation und Implementierung dar und soll nun in den Hyperspace eingefügt werden. Die Integration erfolgt dabei schrittweise anhand von Schritt 1 des verallgemeinerten Hyperspace-Prozesses aus Abschnitt 5.2.4.4:

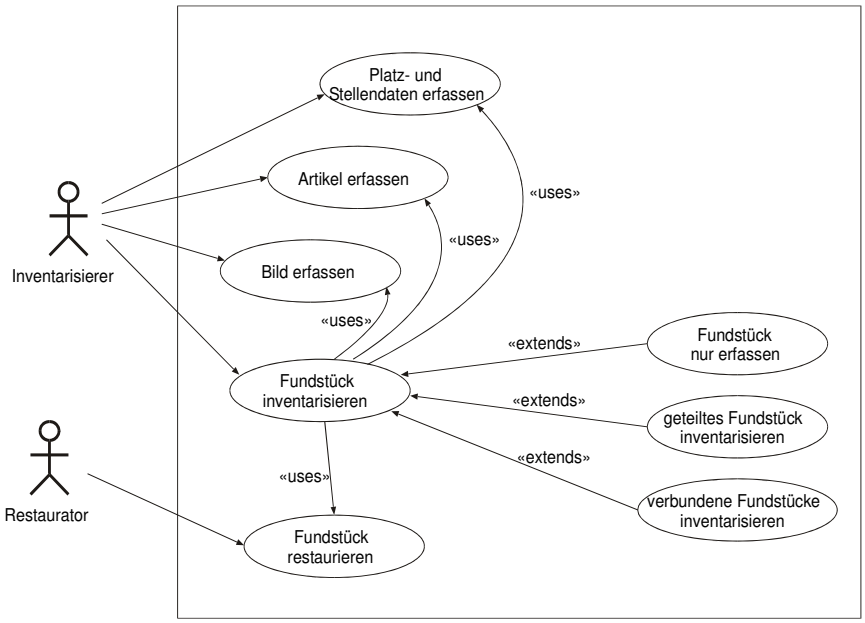

Abbildung 12 Anwendungsfalldiagramm des Entwurfs für ein Inventarisierungssystem (Ausschnitt)
Dargestellt sind die Anwendungsfälle, die von dem System unterstützt werden sollen. Das Modell sieht 5 verschiedene Anwendungsfälle vor:

1. **Fundstück inventarisieren**: Dieser Anwendungsfall bildet den Kern des Modells. Der *Inventarisierer* erfasst Informationen zu einem Fundstück und inventarisiert es. Dabei wird bei Bedarf auf die anderen hier beschriebenen Anwendungsfälle zurückgegriffen. Ferner gibt es von dem Anwendungsfall *Fundstück inventarisieren* drei zusätzliche Varianten:
   a) **verbundene Fundstücke inventarisieren**: Mehrere einzelne Fundstücke sollen unter einer gemeinsamen Inventarposition abgelegt werden, wie z.B. zusammen gehörende Scherben eines Tongefäßes.
   b) **geteiltes Fundstück inventarisieren**: Ein einzelnes Fundstück soll aufgeteilt und unter verschiedenen Inventarpositionen abgelegt werden. Ein Beispiel hierfür wäre ein Gefäß mit Inhalt wie eine Schmuckschatulle, wo der Inhalt getrennt inventarisiert werden soll.
   c) **Fundstück nur erfassen**: Es werden nur die Fundstückdaten erfasst, das Fundstück wird jedoch nicht inventarisiert, weil es z.B. jemand anderem gehört.
2. **Platz- und Stellendaten erfassen**: In diesem Anwendungsfall erfasst der *Inventarisierer* Informationen über einen geographischen Ort. Für die archäologische Forschung ist es oft wichtig zu wissen, wo genau bestimmte Fundstücke gefunden wurden. Da an einer Stelle üblicherweise mehrere Fundstücke entdeckt werden (z.B. weil sich dort eine antike Siedlung befand), werden die Informationen zu den Stellen getrennt von den Fundstücken erfasst und verwaltet.
3. **Artikel erfassen**: In diesem Anwendungsfall erfasst der *Inventarisierer* Artikel aus Fachzeitschriften über bestimmte Fundstücken
4. **Bild erfassen**: In diesem Anwendungsfall erfasst der *Inventarisierer* Bilder zu den Fundstücken.
5. **Fundstück restaurieren**: Manche Fundstücke müssen speziell aufbereitet oder konserviert werden, da sie sonst vergehen würden. In diesem Anwendungsfall führt der *Restaurator* die dazu erforderlichen Tätigkeiten aus.

In Abbildung 13 ist das zugehörige Metamodell dargestellt.
(Abbildung geändert nach [Gab02, S.36])

Integration: Einbinden eines Modells in den Hyperspace. Das *Eingabemodell* muss ein gültiges Modell eines zugehörigen Metamodells sein.

a) Integration des zugehörigen Metamodells in das Metamodell des Hyperspace.
b) Hinzufügen oder Auswählen einer geeigneten artefaktbasierenden Belangdimension.
c) Einfügen der Belange und Grundelemente. Die identifizierten Belange werden in die artefaktbasierende Belangdimension eingefügt. Den einzelnen Belangen werden jeweils die sie beschreibenden Grundelemente zugeordnet.
d) Definieren der Beziehungen zwischen den neuen Grundelementen und bereits vorhandenen Grundelementen anderer artefaktbasierender Belangdimensionen.

Bevor jedoch mit der eigentlichen Integration begonnen werden kann, ist grundsätzlich zu entscheiden wie Use-Case Modelle in einem Belangraum verwendet werden.

5.3.3.1. Vorarbeiten

Vor der erstmaligen Einbindung eines Anwendungsfalldiagramms in einen Belangraum sollte zunächst geklärt werden, was ein solches Modell eigentlich darstellt. Folgende Fragen sind dabei zu beantworten:

- Welche Belange sind dargestellt?
- Was ist die Belangdimension dieser Belange?
- Was sind die Grundelemente?

Die zentrale Frage ist die nach den dargestellten Belangen. Nun, diese ist hier relativ einfach zu beantworten. Ein Anwendungsfalldiagramm stellt Anwendungsfälle dar, gleichsam eine Mischung aus Interaktionen mit dem System und abzubildenden Prozessen. Der *Anwendungsfall* ist damit eine sowohl nahe liegende als auch geeignete (entsprechend der Diskussion zum Belangbegriff in Kapitel 2) Ausprägung eines Belangs. Die Belangdimension ist entsprechend *Anwendungsfälle*.

Anwendungsfalldiagramme bestehen aus *Akteuren* und *Prozessen*, die damit auch die wesentlichen Grundelemente sind. Die Prozessknoten werden in UML ebenfalls *Anwendungsfälle* genannt. Damit wird ihre zentrale Bedeutung im Anwendungsfalldiagramm betont. Ein Anwendungsfall-Grundelement ist demnach das *repräsentative Grundelement* eines Anwendungsfall-Belanges.

5.3.3.2. Schritt a) Integration des Metamodells

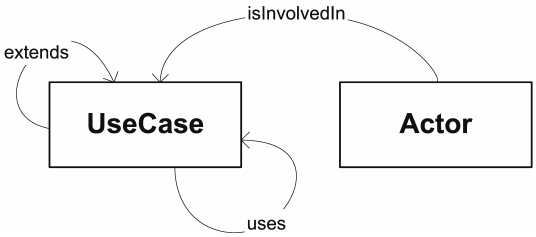

Abbildung 13 Metamodell des Use-Case Modells

Gemäß dem verallgemeinerten Hyperspace-Prozess ist zunächst das Metamodell des einzufügenden Modells in das Metamodell des Hyperspace zu integrieren.

Da unser Anwendungsfalldiagramm das erste Artefakt ist, das in den Hyperspace eingefügt wird, ist dieses hier sehr einfach: Das integrierte Metamodell entspricht dem Metamodell des Anwendungsfalldiagramms aus Abbildung 12. Dieses ist dargestellt in Abbildung 13.

5.3.3.3. Schritt b) Hinzufügen einer artefaktbasierenden Belangdimension

Als nächstes fügen wir dem Hyperspace eine neue artefaktbasierende Belangdimension hinzu, die Dimension <Anwendungsfälle>.

5.3.3.4. Schritt c) Einfügen der Belange und Grundelemente

Die identifizierten Belange des Anwendungsfalldiagramms werden nun in den Hyperspace eingefügt. Jeder der fünf in Abbildung 12 erläuterten Anwendungsfälle, sowie deren Varianten, gelten hier als ein einzelner Belang: Der Anwendungsfall *Fundstück inventarisieren* mit den Varianten *verbundene Fundstücke inventarisieren*, *geteiltes Fundstück inventarisieren* und *Fundstück nur erfassen* sowie die Anwendungsfälle *Platz- und Stellendaten erfassen*, *Artikel erfassen*, *Bild erfassen* und *Fundstück restaurieren* ergeben jeweils einen Belang in der Belangdimension <Anwendungsfälle>.

Den einzelnen Belangen werden nun ihre Grundelemente zugeordnet. Einzelne Grundelemente, hier ist es der Akteur *Inventarisierer*, werden dabei *mehreren Belangen* zugeordnet. Dieses Vorgehen ist hier sicherlich nahe liegend; es ist jedoch nur möglich, weil der verallgemeinerte Hyperspace die geometrische Raumeigenschaft nicht mehr erfüllen muss (siehe Abschnitt 5.2.1).

Der Verzicht auf die Raumeigenschaft eines Hyperspace hat außerdem Einfluss auf die grafische Darstellung der Belangdimension in Abbildung 14. Die Belangdimensionen lassen sich nicht mehr in Form von geometrischen Achsen darstellen und die Zuordnung der Grundelemente zu den Belangen kann nicht länger durch ihre Positionierung im Raum erfolgen. Stattdessen wird eine Belangdimension mit ihren Belangen nun in einem *Cluster* dargestellt, der neben den Belangen auch die repräsentativen Grundelemente umfasst. Alle weiteren (nicht-repräsentativen) Grundelemente befinden sich außerhalb des Clusters und werden über explizite Pfeile den Belangen zugeordnet.

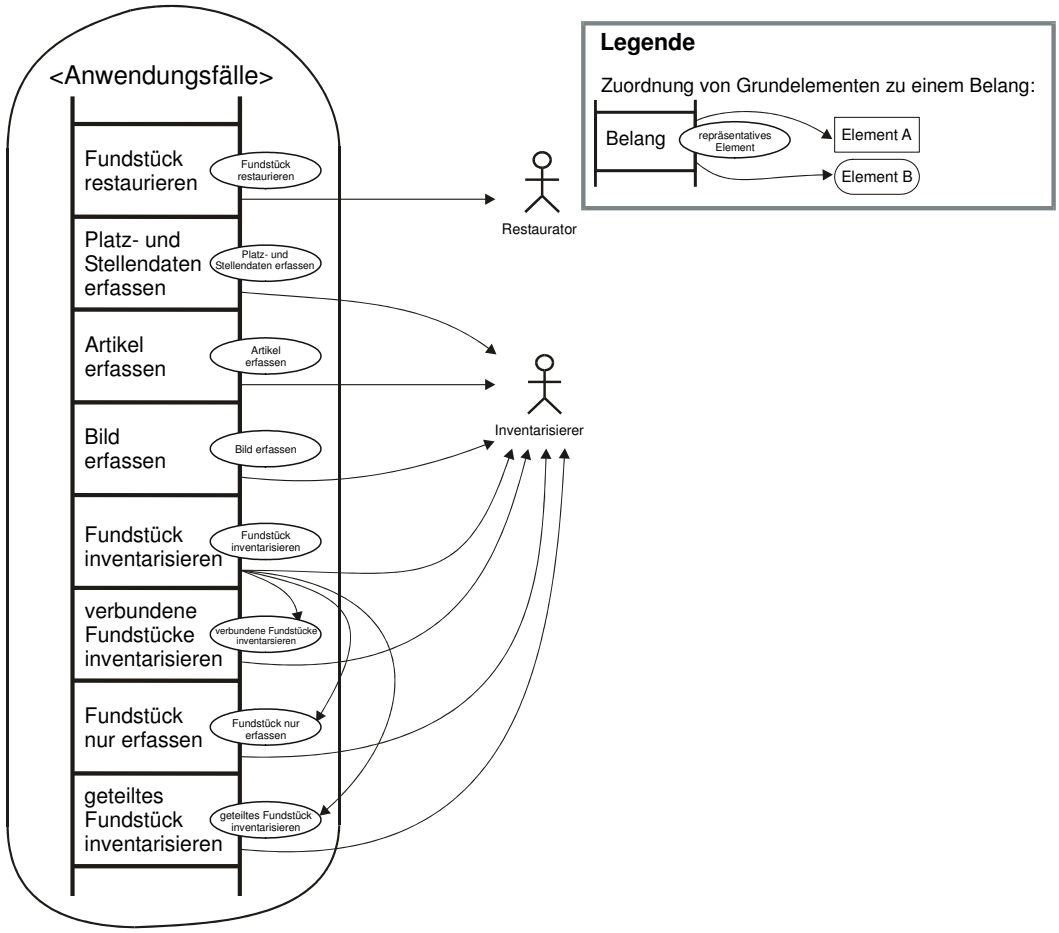

Abbildung 14 Darstellung der Anwendungsfälle als Belange einer Belangdimension
Dargestellt ist die Belangdimension <Anwendungsfälle> in einem Cluster mit den zu ihr gehörenden Belangen und repräsentativen Grundelementen. Repräsentative Grundelemente liegen direkt bei ihrem jeweiligen Belang, alle weiteren sind über Pfeile den Belangen zugeordnet. Jeder Anwendungsfall des Anwendungsfalldiagramms aus Abbildung 12 stellt einen spezifischen Belang dar.
Die drei Varianten des Belangs *Fundstück inventarisieren* sind diesem über den Umweg ihrer repräsentativen Grundelemente zugeordnet. Eine derartige Hilfskonstruktion ist erforderlich, da Beziehungen zwischen Belangen in dieser Arbeit bewusst noch nicht berücksichtigt wurden.
Der Übersichtlichkeit wegen nicht dargestellt sind die Beziehungen zwischen den einzelnen Grundelementen, zumal diese ja bereits eindeutig aus den zugrunde liegenden Artefakten (dem Anwendungsfalldiagramm) hervorgehen.

In dieser Arbeit noch nicht berücksichtigt ist eine explizite Darstellung von Beziehungen zwischen Belangen. Dieses betrifft im Beispiel die Spezialisierungsbeziehungen (extends) zwischen dem Anwendungsfall *Fundstück inventarisieren* und seinen Varianten *verbundene Fundstücke inventarisieren*, *Fundstück nur erfassen* und *geteiltes Fundstück inventarisieren*. Man kann sich in diesem Fall jedoch durch eine Hilfskonstruktion helfen, indem man die Beziehungen zwischen dem Belang und seinen Varianten abbildet auf Beziehungen zwischen dem Belang und den repräsentativen Grundelementen seiner Varianten (Abbildung 14). Nichtsdestotrotz ist es wünschenswert, dass sich in einer zukünftigen Version des erweiterten Hyperspace-Modells derartige Beziehungen direkt darstellen lassen.

5.3.3.5. Schritt d) Definieren der Beziehungen zwischen Grundelementen

Entsprechend dem verallgemeinerten Prozessmodell folgt nun die Integration der Grundelemente mit denen aus den anderen artefaktbasierenden Belangdimensionen. Dieser Schritt entfällt hier, da es sich bei der Dimension <Anwendungsfälle> um die bislang einzige artefaktbasierende Belangdimension des Hyperspace handelt.

5.3.4. Integration des Datenmodells

Als nächstes soll nun mit dem in Abbildung 15 dargestellten Datenmodell ein weiteres Artefakt, ein UML Klassendiagramm, in den Hyperspace eingebunden werden. Wir erhalten also eine weitere artefaktbasierende Belangdimension. Die Integration erfolgt wiederum schrittweise anhand von Schritt 1 des verallgemeinerten Hyperspace-Prozesses aus Abschnitt 5.2.4.4.

5.3.4.1. Vorarbeiten

Auch bei der Integration des Datenmodells stellen wir uns zunächst wieder die grundlegenden Fragen:

- Welche Belange sind dargestellt?
- Was ist die Belangdimension dieser Belange?
- Was sind die Grundelemente?

Die Fragen lassen sich hier unmittelbar beantworten, schließlich wurden in den vorangegangenen Beispielen zu Hyper/J schon mehrfach Klassenmodelle in einen Hyperspace eingefügt: Die Belange eines Klassendiagramms sind *Klassen*, die Belangdimension dementsprechend <Klassen>. Die Klassenknoten eines UML Klassendiagramms sind die repräsentativen Grundelemente. Als weitere Grundelemente kommen Operationen und Attribute in Betracht. Das Klassenmodell von GABLER ist ein reines Datenmodell, daher gibt es hier keine Operationen. Der Übersicht halber wurde außerdem auf die Darstellung der Attribute verzichtet, die jeweils ausschließlich ihrem Klassen-Belang zugeordnet werden und daher für die Anordnung im Hyperspace nur von untergeordnetem Interesse sind. Das Datenmodell ist dargestellt in Abbildung 15.

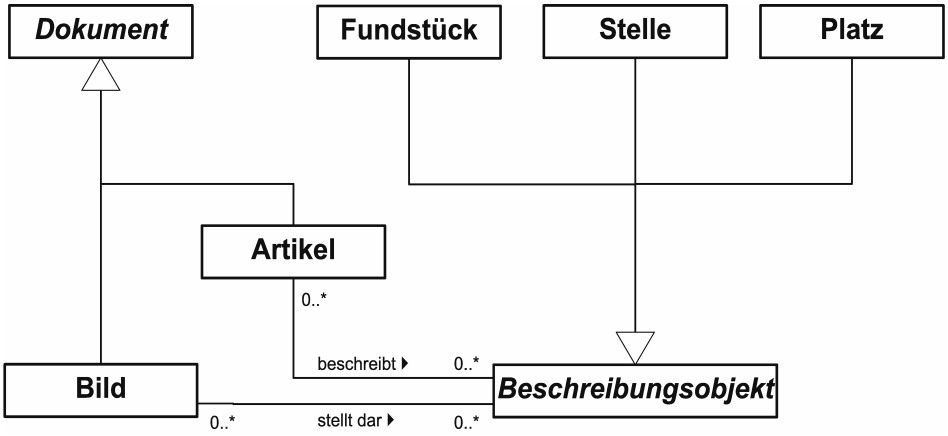

Abbildung 15 Datenmodell des Inventarisierungssystems (Ausschnitt).
Fundstück, *Platz* und *Stelle* sind die zentralen Klassen der Objekte, die mit dem Inventarisierungssystem verwaltet werden sollen. Neben den (hier nicht dargestellten) Standarddaten dieser Objekte lassen sich ihnen zusätzlich noch *Artikel*, wie z.B. Veröffentlichungen aus Fachzeitschriften sowie *Bilder* zuordnen. Die rein abstrakten Klassen *Beschreibungsobjekt* und *Dokument* haben im Anwendungskontext keine weitergehende Funktion und sind im Wesentlichen als technische Hilfsmittel für Substituierbarkeit zu verstehen.
Bei dem Modell von GABLER handelt es sich um ein reines Datenmodell, die Klassen haben keine Methoden. Sie verfügen jedoch über einen ausgesprochen umfangreichen Satz an Attributen, welche der Übersichtlichkeit halber hier nicht mit aufgenommen wurden. Das zugehörige Metamodell ist in Abbildung 16 dargestellt.
(Abbildung angelehnt an [Gab02, S.30])

5.3.4.2. Schritt a) Integration des Metamodells

Zunächst ist das Metamodell des einzufügenden Modells in das Metamodell des Hyperspace zu integrieren. Das Metamodell für das Klassendiagramm aus Abbildung 15 ist dargestellt in Abbildung 16.

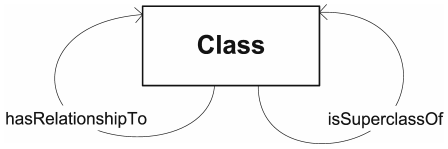

Abbildung 16 Metamodell des Datenmodells
Da das in Abbildung 15 dargestellte Datenmodell weder Methoden noch Attribute enthält, fällt das Metamodell entsprechend einfach aus.

Dieses muss nun mit dem bisherigen Metamodell des Hyperspace, dem Metamodell des Anwendungsfalldiagramms, zusammengefügt werden. Dabei ist zu überprüfen, ob und wo weitere Beziehungen zwischen den Elementen aus dem bisherigen Metamodell des Hyperspace und den neu hinzugekommenen Elementen einzufügen sind.

In unserem Fall besteht ein Zusammenhang zwischen den Klassen des Datenmodells und den Anwendungsfällen des Use-Case Modells. Die Anwendungsfälle beschreiben Prozesse, in denen Instanzen der einzelnen Klassen erzeugt, verändert oder genutzt werden. Wir definieren deshalb eine neue Beziehung *isUsedIn* zwischen Klassen und Anwendungsfällen. Das resultierende Metamodell ist dargestellt in Abbildung 17.

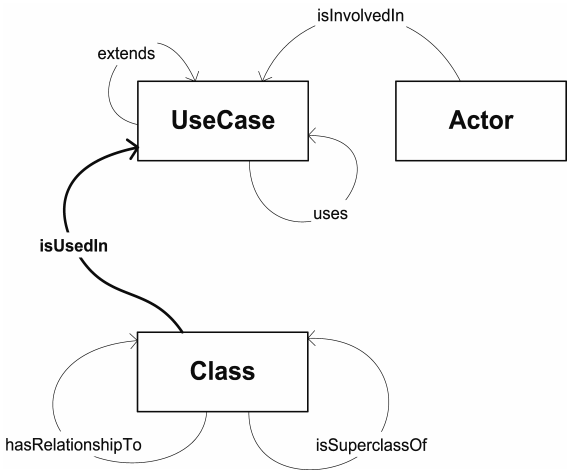

Abbildung 17 Integriertes Metamodell für Datenmodell und Use-Case Modell
Das integrierte Metamodell setzt sich aus den Einzelmodellen aus Abbildung 13 und Abbildung 16 zusammen, zusätzliche neue Elemente sind fett dargestellt.
Die Anwendungsfälle aus dem Use-Case Modell beschreiben Verrichtungen auf Objekten der Klassen des Datenmodells. Deshalb wurde hier eine neue Beziehung *isUsedIn* zwischen den Entitätentypen *Class* und *UseCase* eingeführt.

5.3.4.3. Schritt b) Hinzufügen einer artefaktbasierenden Belangdimension

Da es in unserem Hyperspace bislang keine geeignete artefaktbasierende Belangdimension für die Grundelemente des Datenmodells gibt, fügen wir mit der Dimension <Klassen> eine neue artefaktbasierende Belangdimension hinzu.

5.3.4.4. Schritt c) Einfügen der Belange und Grundelemente

Die Belange des Datenmodells werden nun dem Hyperspace hinzugefügt. Für jede nichtabstrakte Klasse des in Abbildung 15 dargestellten Klassenmodells wird ein Belang erzeugt und in der Belangdimension <Klassen> angeordnet. Der resultierende Hyperspace mit den Belangdimensionen <Anwendungsfälle> und <Klassen> ist dargestellt in Abbildung 18.

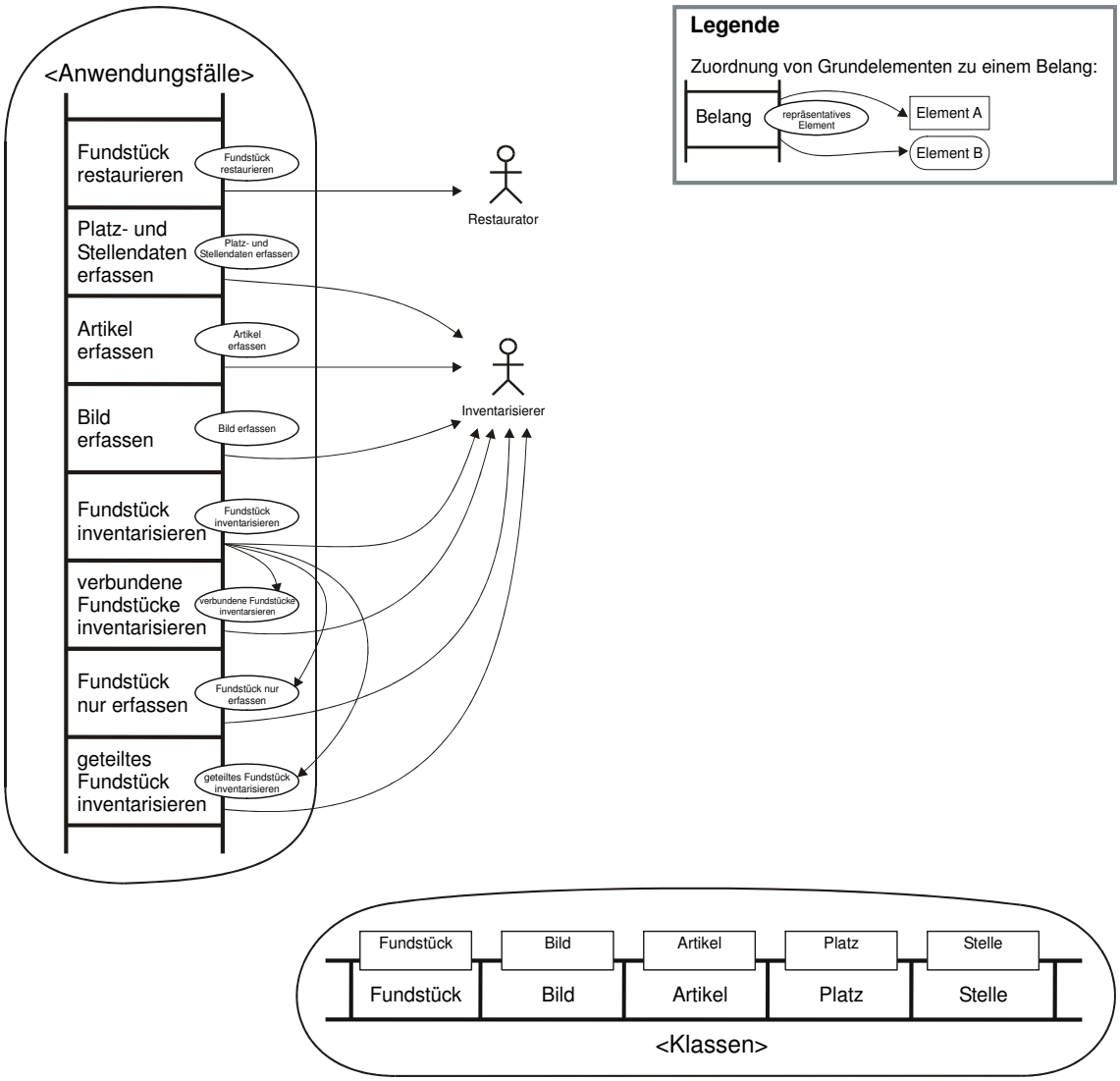

Abbildung 18 Einbindung des Datenmodells in den Hyperspace
Neben der in Abbildung 14 erläuterten Belangdimension <Anwendungsfälle> ist nun mit <Klassen> eine weitere artefaktbasierende Belangdimension in den Hyperspace aufgenommen worden, mit den konkreten Klassen *Fundstück, Bild, Artikel, Platz* und *Stelle* des Klassendiagramms aus Abbildung 15 als Belange. Die Oberklassen *Dokument* und *Beschreibungsobjekt* stellen als rein abstrakte Klassen keine echten Belange dar und wurden deshalb nicht mit aufgenommen. Sie sind vielmehr als ein in der Objektorientierung übliches Hilfsmittel zur Definition von Substituierbarkeit zu verstehen.

5.3.4.5. Schritt d) Definieren der Beziehungen zwischen Grundelementen

Der Hyperspace, so wie in Abbildung 18 dargestellt, enthält nun zwei artefaktbasierende Belangdimensionen mit je einer Reihe von Belangen und Grundelementen. Die Grundelemente sind jeweils ihren Belangen zugeordnet. Was jedoch noch fehlt, sind die Beziehungen zwischen den Grundelementen verschiedener Belangdimensionen, hier also zwischen Klassen auf der einen Seite und Anwendungsfällen/Akteuren auf der anderen Seite. Die dabei möglichen Beziehungen wurden mit der Integration der Metamodelle in Schritt a) (Abbildung 17) bereits festgelegt. Die konkreten Beziehungen gehen jedoch nicht aus den zugrunde liegenden Artefakten, dem Datenmodell aus Abbildung 15 und

dem Anwendungsfalldiagramm aus Abbildung 12, hervor und müssen deshalb nun definiert werden. Das Ergebnis ist dargestellt in Abbildung 19.

5.3.5. Zwischenbilanz I

Wir haben nun mit dem Anwendungsfalldiagramm (Abbildung 12) und dem Klassendiagramm (Abbildung 15) des Inventarisierungssystems zwei verschiedene Artefakte in den

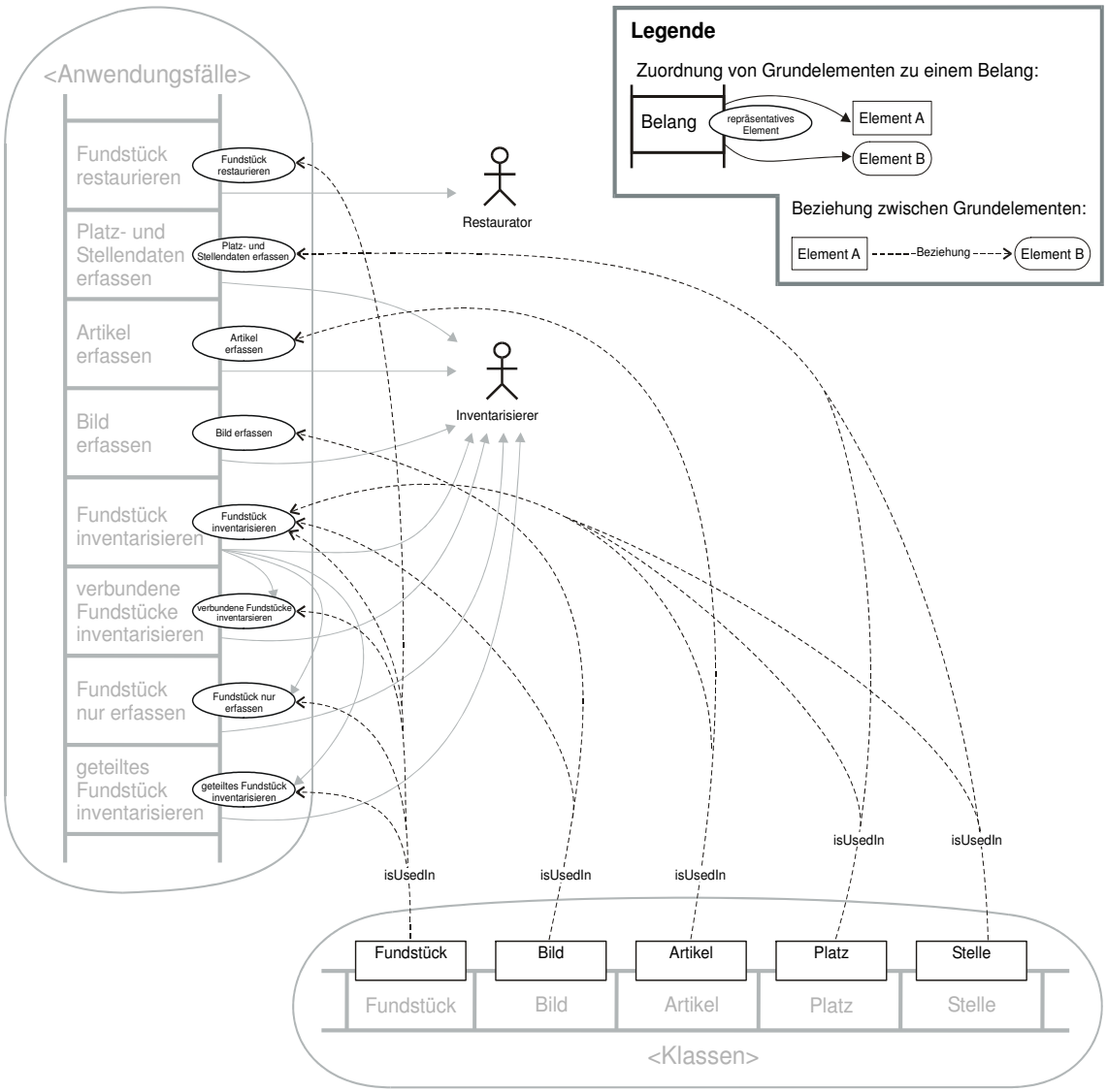

Abbildung 19 Definition der Beziehungen zwischen den Grundelementen

Dargestellt sind die Grundelemente der artefaktbasierenden Belangdimensionen mit den Beziehungen zwischen Grundelementen *verschiedener* artefaktbasierender Belangdimensionen. (Die Beziehungen zwischen Grundelementen innerhalb derselben Belangdimension werden nicht dargestellt, diese gehen eindeutig aus den zugrunde liegenden Artefakten hervor.) Alle sonstigen Elemente des Belangraums sind hier nicht relevant und nur zum Zwecke der Orientierung in hellem Grau angedeutet.

Bei der Integration der Metamodelle des Anwendungsfalldiagramms und des Klassendiagramms (Abbildung 17) wurde der neue Beziehungstyp *isUsedIn* zwischen Klassen und Anwendungsfällen eingeführt. Hier wird nun gezeigt, zwischen welchen konkreten Grundelementen eine solche Beziehung besteht.

Hyperspace integriert. Dabei haben wir mit den Belangdimensionen <Anwendungsfälle> und <Klassen> zwei artefaktbasierende Belangdimensionen erstellt und die in den jeweiligen Artefakten dargestellten Belange identifiziert. Den einzelnen Belangen wurden dann die Grundelemente zugeordnet (Abbildung 18).

Zusätzlich erfolgte im Hyperspace eine Integration der aus den verschiedenen Artefakten stammenden Grundelemente miteinander (Abbildung 19). Diese wurde anhand eines integrierten Metamodells (Abbildung 17) vorgenommen. Das integrierte Metamodell wurde erstellt durch die Verknüpfung des Anwendungsfalldiagramm-Metamodells (Abbildung 13) und des Klassendiagramm-Metamodells (Abbildung 16).

Damit bietet uns der Hyperspace bereits eine Reihe von Strukturierungsmöglichkeiten sowie eine detailliertere Sicht auf das Inventarisierungssystem als die Einzelartefakte. So lässt sich z.B. bei Änderungen an einer Klasse einfach feststellen, welche Anwendungsfälle eventuell angepasst werden müssen. Ebenso lässt sich ermitteln, welche Klassen überflüssig werden, wenn man einzelne Anwendungsfälle entfernt.

Im Folgenden sollen nun weitere Selektions- und Strukturierungsmöglichkeiten für unseren Hyperspace bereitgestellt werden. Dazu werden zu den artefaktbasierenden Belangdimensionen nun noch zwei selektierende Belangdimensionen hinzugefügt.

5.3.6. Hinzufügen selektierender Belangdimensionen

Selektierende Belangdimensionen bieten die Möglichkeit alternative Sichten und Strukturierungen auf der Grundelementmenge zu definieren. Sie dienen einer alternativen Trennung der Belange, bei der die so getrennten einzelnen Belange abgeschlossene Charakteristika des Softwaresystems aus einer anderen Sicht darstellen.

Das Hinzufügen von selektierenden Belangdimensionen ist im erweiterten Hyperspace-Prozess (5.2.4.4) unter Punkt 2 beschrieben:

> *Selektion*: Je nach Bedarf werden selektierende Belangdimensionen erzeugt und die Grundelemente den Belangen der selektierenden Belangdimensionen zugeordnet.

Wir fügen dem Hyperspace zwei selektierende Belangdimensionen hinzu:

- Die aus den vorangegangenen Kapiteln bereits bekannte und bewährte Belangdimension <Features>, die eine Gliederung der Grundelemente nach funktionalen Gesichtspunkten ermöglicht.
- Die Belangdimension <Verrichtungen>, durch die eine Gliederung nach Tätigkeiten der Anwender vorgenommen wird.

5.3.6.1. Die selektierende Belangdimension <Features>

Features sind einzelne Belange eines Softwaresystems aus der funktionalen Sicht des Anwenders. Lastenhefte bestehen oft aus Listen der zu unterstützenden Features. Die Implementierung bestimmter Features ist typischerweise Vertragsgrundlage für Softwareprojekte, wobei die einzelnen Features oft in ihrer Relevanz für den Anwender ge-

wichtet werden, z.B. in *unabdingbar*, *wichtig*, *optional*. Insgesamt sind Features damit guter Kandidat für eine eigene Belangdimension.

Für das Inventarisierungssystem seien die folgenden vier Features identifiziert:

1. *Funddatenverwaltung* umfasst das Erfassen, Bearbeiten und Ausgeben der wichtigsten Daten, die in der Anwendungsdomäne vorkommen. Im Falle des Inventarisierungssystems wäre hierunter z.B. die Verwaltung von *Fundstücken*, *Plätzen* und *Stellen* zu verstehen. Die Funddatenverwaltung ist ein unabdingbares Feature, das System ist ohne nicht sinnvoll einzusetzen.
2. *Bilderverwaltung* umfasst das Erfassen, Bearbeiten und Ausgeben von Bilddokumenten zu den Datenelementen der Funddatenverwaltung. Da Skizzen und Fotos eine große Rolle für die Arbeit des Anwenders spielen, ist sie ein wichtiges Feature.
3. *Artikelverwaltung* umfasst das Erfassen, Bearbeiten und Ausgeben von Artikeln (Veröffentlichungen) in Fachzeitschriften zu einzelnen Fundstücken, Plätzen oder Stellen. Die Artikelverwaltung ist ein optionales Feature.
4. *Recherchefunktion* umfasst die Möglichkeit im erfassten Datenbestand nach unterschiedlichsten (auch kombinierten) Kriterien zu suchen. Für die hier relevante erste Phase des Projektes, bei dem der bestehende bislang in Papierform verwaltete Datenbestand erfasst werden soll, ist die Recherchefunktion ein optionales Feature. Für den späteren Vollbetrieb, die wissenschaftlichen Nutzung des Archivs, ist sie hingegen ein unabdingbares Feature.

Nachdem die einzelnen Features als Belange der Belangdimension <Features> in den Belangraum eingetragen wurden, müssen ihnen Grundelemente zugeordnet werden. Sinnvollerweise sollten das Elemente sein, die mit dem Feature eng verbunden sind, es näher spezifizieren oder implementieren. Im Beispiel zu Hyper/J und dem Hyperspace-Ansatz, der Softwareentwicklungsumgebung (SEU) aus Abschnitt 4.3, wurden den Features jeweils die *Methoden* zugeordnet, in denen die jeweilige Funktion *implementiert* ist. Hier auf der Entwurfsebene erscheint es nun sinnvoll den Features die *Anwendungsfälle* zuzuordnen, die das jeweilige Feature *spezifizieren*. Alternativ oder zusätzlich könnte man einem Feature auch alle beteiligten *Klassen* zuordnen, dies erscheint jedoch unnötig. Zum einen sind, durch das integrierte Metamodell der Artefaktsprachen, die Klassen bereits mit den Anwendungsfällen verbunden, die zur Implementierung eines Features verwendeten Klassen lassen sich also eindeutig ermitteln. Zum anderen würde dieses eine unnötig hohe Kopplung bedeuten. Aus der Sicht der Dimension <Features> ist es letztendlich egal, ob ein Feature nun objektorientert, prozedural oder funktional implementiert ist.

Die Integration der selektierenden Belangdimension <Features> mit den nach diesen Kriterien zugeordneten Grundelementen ist dargestellt in Abbildung 20.

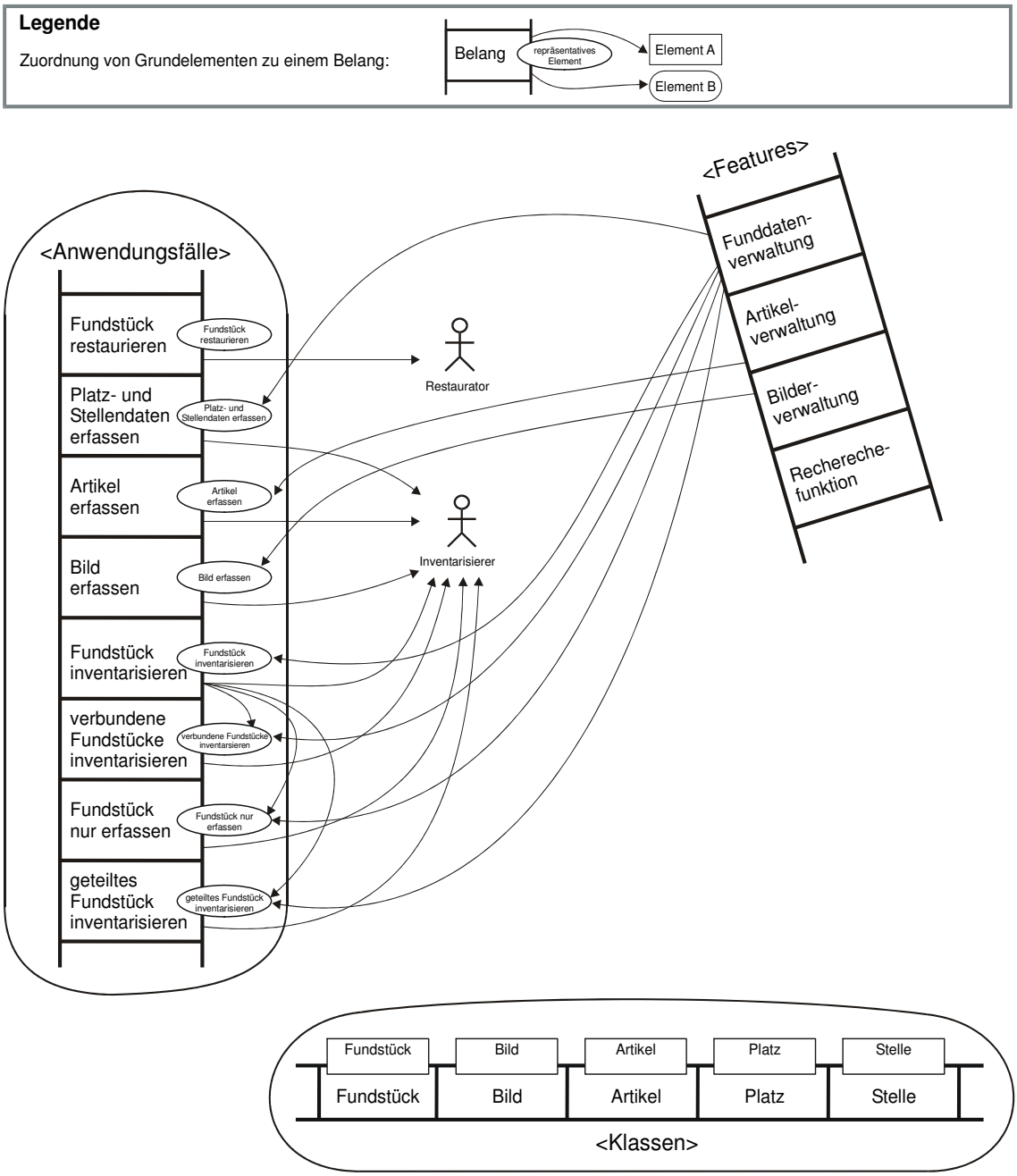

Abbildung 20 Der Hyperspace nach Hinzufügen der selektierenden Belangdimension <Features>
Als selektierende Belangdimension fügt <Features> dem Belangraum keine eigenen Grundelemente hinzu. Den Belangen werden Grundelemente aus den artefaktbasierenden Belangen zugeordnet, hier sind es die Anwendungsfälle, die das entsprechende Feature näher beschreiben.

5.3.6.2. Die selektierende Belangdimension <Verrichtungen>

Eine *Verrichtung* ist eine Tätigkeit, die an einem Objekt vollzogen wird. Beispiele für Verrichtungen sind *etwas erfassen*, *etwas restaurieren*, *etwas drucken*. Der Fokus liegt auf den Tätigkeiten; eine Verrichtung ist damit als ein *generischer Prozess* zu verstehen, der mit einem beliebigen Objekt durchgeführt werden kann.

Ein wichtiges Kriterium für die Anwenderakzeptanz einer Software ist die Erlernbarkeit. Softwareergonomen führen dazu gerne das *Ähnlichkeitsprinzip* an: Es erleichtert die Erlernbarkeit ungemein, wenn ähnliches auch ähnlich behandelt wird. Der Mensch erfasst und nutzt Ähnlichkeit dabei sowohl bei Objekten wie auch bei Prozessen. So erwartet man von einem Textsatzsystem, dass Absätze, Tabellen und Grafiken frei positioniert und wahlweise mit einem Rahmen umgeben werden können – also ähnliche Objekte ähnlich behandelt werden. Man erwartet aber auch, dass die Druckfunktion des Textsatzsystems ähnlich der Druckfunktion der Adressdatenbank aufgebaut ist. In beiden Fällen wird der zu druckende Bereich des Dokuments/der Daten festgelegt, ein Drucker

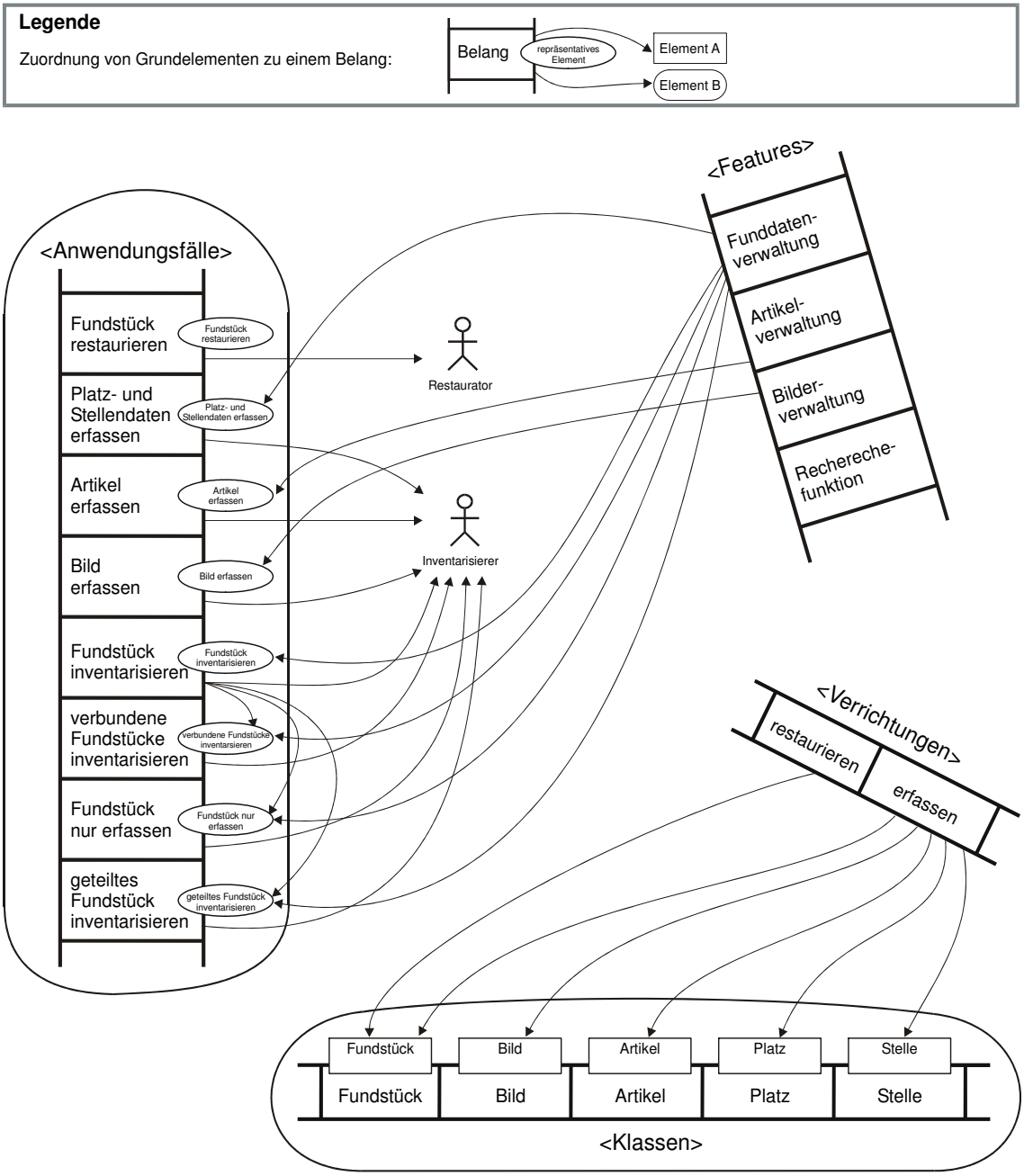

Abbildung 21 Der Hyperspace nach Hinzufügen der selektierenden Belangdimension <Verrichtungen>

ausgewählt und eine Reihe von Druckoptionen angegeben. Obwohl die zu druckenden Objekte hier grundverschieden sind, haben die Prozesse eine große Ähnlichkeit.

Das objektorientierte Paradigma unterstützt das Ähnlichkeitsprinzip bei Objekten in geradezu idealer Weise. Es unterstützt das Prinzip jedoch so gut wie gar nicht bezogen auf Prozesse. Verrichtungen sind also hochgradig quer schneidende Belange.

Verrichtungen beschreiben Prozessähnlichkeit aus Anwendersicht. Sie sind damit eine geeignete Belangdimension. Um die grafische Darstellung des Belangraums nicht vollends zu überfrachten, erhält die Belangdimension <Verrichtungen> hier nur zwei beispielhafte Belange: *erfassen* und *restaurieren*. Als Grundelemente ordnen wir den Belangen diejenigen Klassen zu, auf deren Instanzen eine derartige Verrichtung vorgesehen ist. Anhand dieser Zuordnung kann dann z.B. die Qualitätskontrolle untersuchen, ob das Ähnlichkeitsprinzip erfüllt wird. Ein Maß dafür könnte sein, wie ähnlich die Masken zum *erfassen* der einzelnen Objekte aufgebaut sind. Abbildung 21 zeigt den resultierenden Belangraum.

5.3.7. Zwischenbilanz II

Unser Hyperspace, wie in Abbildung 21 dargestellt, enthält nun insgesamt vier Belangdimensionen. Die artefaktbasierenden Belangdimensionen <Anwendungsfälle> und <Klassen> sowie die selektierenden Belangdimensionen <Features> und <Verrichtungen>. Dieses *integrierte* vierdimensionale Modell des Softwaresystems ermöglicht uns ein tiefer gehendes Verständnis für die inneren Zusammenhänge. Wir können damit:

- *Aufgabenbezogene Sichten erstellen*. Schon bei mittelgroßen Softwaresystemen ist es für den einzelnen Entwickler schwierig, den Überblick zu behalten. Dank der Integration aller Artefakte und Belange in den Hyperspace ist es jedoch nun möglich, das System auf die im Rahmen der Erfüllung einer Aufgabe relevanten Teile zu reduzieren. So könnte z.B. der für das Feature *Artikelverwaltung* zuständige Entwickler eine Sicht erhalten, in der nur noch die von der Artikelverwaltung betroffenen Klassen und Anwendungsfälle dargestellt sind sowie die Grundelemente, von denen sie direkt abhängen.
- *Entwurfselemente wieder verwenden*. Es ist einfach möglich einzelne Belange zu extrahieren, um sie in anderen Systemen wieder zu verwenden. Nehmen wir an, für die Entwicklung eines Warenwirtschaftssystems wünscht der Kunde, Bilder zu den gehandelten Waren hinterlegen zu können, um diese z.B. im Online-Shop zu präsentieren. Dafür könnte dann der Entwurf für die Bilderverwaltung aus dem Inventarisierungssystem extrahiert und für das Warenwirtschaftssystem wieder verwendet werden.

Der nächste Abschnitt demonstriert die Anwendung am Beispiel der aufgabenbezogenen Sichten für unser Inventarisierungssystem.

5.3.8. Kombination und Projektion einer Auswahl von Belangen

Der Entwurf für das Inventarisierungssystem steht nun soweit fest und wurde komplett in den Hyperspace aufgenommen. Aus Kostengründen wünscht der Kunde jedoch plötzlich eine möglichst einfache Lösung, in der nur noch die unverzichtbaren Features implementiert werden. Als Implementierungsrahmen bleibt damit aus den in Abschnitt 5.3.6.1 identifizierten Features nur noch das Feature *Funddaten verwalten* übrig.

Aus den mit diesem Feature verbundenen Grundelementen soll deshalb nun ein gültiges Modell erzeugt werden und, für das bessere Verständnis, wieder auf die ursprünglichen Artefaktsprachen (Klassendiagramm und Anwendungsfalldiagramm) abgebildet werden. Das Vorgehen hierfür wird in den Schritten 3 und 4 des erweiterten Hyperspace-Prozesses aus Abschnitt 5.2.4.4 beschrieben:

> *Kombination*: Auswahl einer Menge zu kombinierender Belange (Hypermodule). Die damit ausgewählte Teilmenge der Grundelemente stellt eventuell kein gültiges Modell des integrierten Metamodells dar. In diesem Fall werden die Grundelemente in einem Transformationsprozess zu einem gültigen Modell, dem *Ausgabemodell*, kombiniert.
>
> *Projektion*: Das Ausgabemodell ist ein gültiges Modell im Sinne des *integrierten* Metamodells. Es ist damit nicht in einer der üblichen Artefaktsprachen darstellbar. Zur Weiterverarbeitung kann es daher erforderlich sein, dass integrierte Modell wieder abzubilden auf Einzelmodelle der verwendeten Artefaktsprachen.

5.3.8.1. Kombination

Ziel der Kombination ist, dass die durch die zu kombinierenden Belange ausgewählten Grundelemente (das Ausgabemodell) ein in sich gültiges Modell im Sinne des integrierten Metamodells sind. Wie dieses im Einzelnen sicher zu stellen ist, hängt in hohem Maße von den verwendeten Artefaktsprachen ab. In vielen Sprachen gibt es Konstrukte (Grundelemente), die „nicht alleine stehen dürfen“. Eine Methode kann z.B. nicht ohne eine Klasse existieren, zu der sie gehört. Ist die Methode, nicht jedoch die Klasse von der sie abhängt, Element der zu kombinierenden Grundelementmenge, so muss der Kombinationsprozess sicherstellen, dass sie im Ausgabemodell wieder zu einer Klasse gehört. Dafür sind eventuell Modelltransformationen notwendig, indem z.B. eine umgebende Klasse generiert wird oder die Methode einer anderen im Ausgabemodell vorhandenen Klassen zugeschlagen wird. Eine dritte Alternative wäre die Hüllenbildung, bei der die Grundelementmenge solange erweitert wird, bis das Ausgabemodell gültig ist. Für die angesprochene Methode würde das bedeuten, dass die sie umgebende Klasse des Ursprungsmodells mit aufgenommen wird.

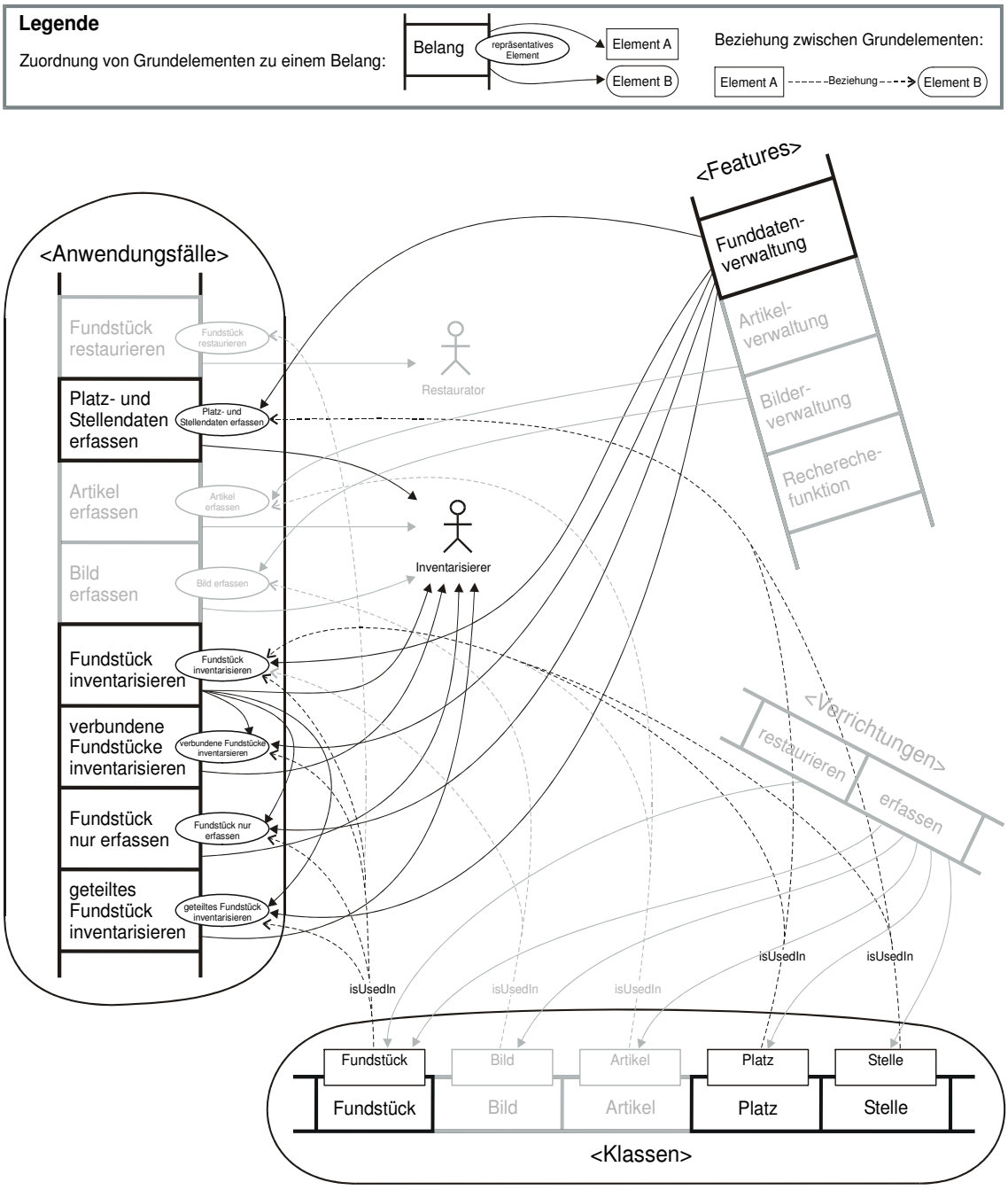

Abbildung 22 Zu kombinierende Belange und zugeordnete Grundelemente

Dargestellt ist die Menge der Belange und Grundelemente, die für das Ausgabemodell herangezogen werden. Diese Menge wird durch Bildung einer Hülle entlang der dargestellten Beziehungen nach folgendem Algorithmus ermittelt:

1. Definiere einen Startzustand, bestehend aus einer Menge von Belangen.
2. Füge alle Grundelemente hinzu, die einem der enthaltenen Belange zugeordnet sind.
3. Für alle enthaltenen repräsentativen Grundelemente: Füge den gleichnamigen Belang hinzu.
4. Füge alle Grundelemente hinzu, die zu einem enthaltenen Grundelement eine *isUsedIn* Beziehung haben.
5. Wiederhole Schritte 2-4, bis sich die Hülle nicht mehr vergrößert.

Der Startzustand besteht aus dem Belang *Funddatenverwaltung*. Mit Schritt 2 werden die Anwendungsfall-Grundelemente *Platz- und Stellendaten erfassen*, *Fundstück inventarisieren* und die drei Varianten des letzteren aufgenommen. Mit Schritt 3 werden die gleichnamigen Anwendungsfall-Belange aufgenommen, mit Schritt 4 die Klassen-Grundelemente *Fundstück*, *Platz* und *Stelle*. Durch die nochmalige Wiederholung der Schritte 2 und 3 kommen schließlich noch der Akteur *Inventarisierer* sowie die Klassen-Belange *Fundstück*, *Platz* und *Stelle* in die Hülle.

In unserem Fall ist die Menge der zu kombinierenden Grundelemente über den Belang *Funddatenverwaltung* zu ermitteln. Dazu wird, ausgehend von *Funddatenverwaltung* entlang der Beziehungen, eine Hülle gebildet. Wie weit man die Hüllenbildung treiben sollte hängt wiederum von den verwendeten Artefaktsprachen ab, eventuell auch noch vom konkreten Modell. Das hier verwendete Vorgehen der Hüllenbildung und die daraus resultierende Menge der Grundelemente sind beschrieben und dargestellt in Abbildung 22. Da die resultierende Grundelementmenge hier außerdem bereits ein gültiges Modell ist, sind keine weiteren Modelltransformationen mehr erforderlich.

5.3.8.2. Projektion

Ziel der Projektion ist es, aus dem Ausgabemodell der Kombination wieder Artefakte in den bekannten Artefaktsprachen zu erzeugen. Dies kann z.B. zur Weiterverarbeitung in anderen Werkzeugen, zur Kommunikation zwischen Entwicklern, zur besseren Lesbarkeit oder aus anderen Gründen erforderlich sein. Bei der Projektion wird bewusst ein gewisser Informationsverlust in Kauf genommen, da die zusätzlichen Beziehungen, die bei der Integration der Metamodelle entstanden sind, verloren gehen.

In unserem Fall sind die resultierenden Artefakte wieder ein Anwendungsfalldiagramm und ein Klassendiagramm, die Beziehung *isUsedIn* zwischen Klassen und Anwendungsfällen, die wir bei der Integration definiert hatten, entfällt dabei.

Abbildung 23 zeigt das Anwendungsfalldiagramm des zu implementierenden Teilsystems, Abbildung 24 das Klassendiagramm.

Damit haben wir alle Schritte des erweiterten Hyperspace-Prozesses aus Abschnitt 5.2.4.4 durchlaufen und sind am Ende unseres Beispiels angekommen.

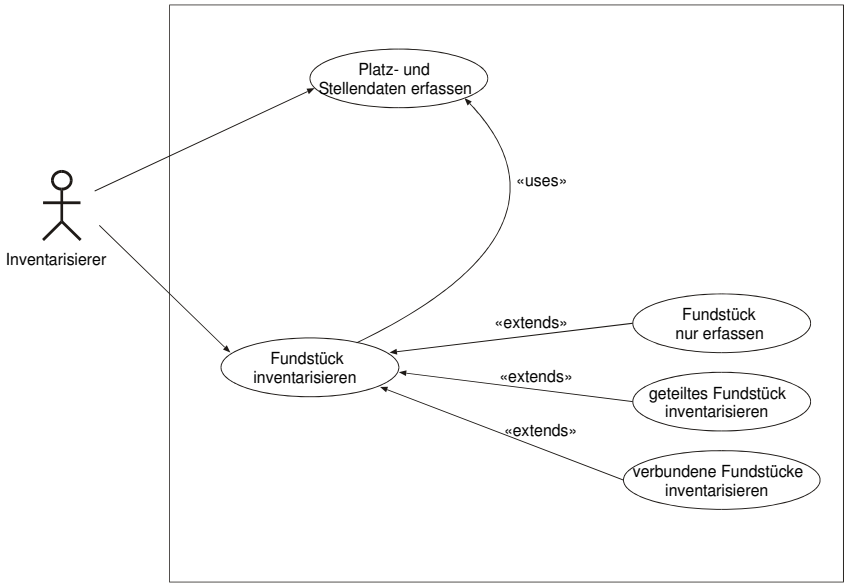

Abbildung 23 Anwendungsfälle des Implementierungsrahmens
Dargestellt ist das Anwendungsfallmodell aus Abbildung 12, reduziert auf die für das Feature *Funddatenverwaltung* relevanten Anwendungsfälle.

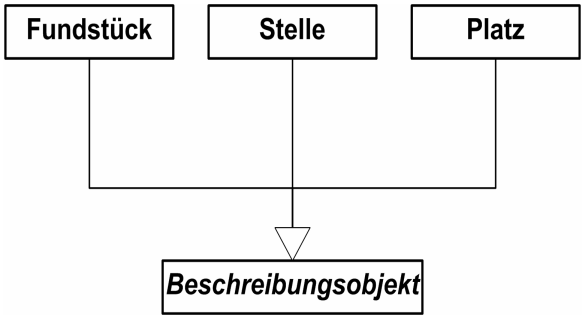

Abbildung 24 Klassen des Implementierungsrahmens
Dargestellt ist das Klassendiagramm aus Abbildung 15, reduziert auf die für das Feature *Funddatenverwaltung* relevanten Klassen.

5.4. Diskussion

Wie das Beispiel gezeigt hat, ist das erweiterte Hyperspace-Modell ein tragfähiges und anwendbares Modell zum multidimensionalen Trennen der Belange mit mehreren beteiligten Artefaktsprachen. Im Folgenden soll nun das Modell diskutiert und dabei insbesondere mit den in Kapitel 3 (Aspektorientierte Programmierung) und Kapitel 4 (Der Hyperspace-Ansatz) vorgestellten Ansätzen verglichen werden.

5.4.1. Vergleich mit aspektorientierter Programmierung

Die aspektorientierte Programmierung adressiert gezielt das Problem der quer schneidenden Belange. Sie stellt Techniken bereit zur Dekomposition eines Problems sowohl in seine funktionalen Anteile (Fachkonzepte) als auch in eine Reihe von Aspekten – Konzepte, die als Querschnittseigenschaften die Fachkonzepte überdecken. Die bekannten Ansätze zur aspektorientierten Programmierung sind dabei meistens aus einer ganz spezifischen Not heraus entstanden. *Adaptive Programming* (Abschnitt 3.4) nimmt sich des Problems der Strukturanomalitäten in objektorientierter Software an und erlaubt das Trennen der Algorithmen von der Klassenstruktur. *Subject-Oriented Programming* (Abschnitt 3.5) stellt Techniken bereit, um verschiedene Sichten (oder Varianten) einer Klassenstruktur zusammen zu führen. *Composition Filters* (Abschnitt 3.6) beschäftigt sich schwerpunktmäßig mit der Koordination des Nachrichtenaustausches zwischen Objekten. Das deutlich neuere *AspectJ* (Abschnitt 3.7) schließlich versteht sich als ein Framework zur Exploration von Aspekttechniken und den dafür benötigten Hilfsmitteln.

Die genannten Ansätze sind in den letzten Jahren teilweise deutlich weiterentwickelt worden. Ihnen ist jedoch gemein, dass sie das Problem der quer schneidenden Belange nur auf der *Implementierungsebene* betrachten. Die Modelle, die sie bearbeiten, sind immer *Code-Modelle*, quer schneidende Eigenschaften werden gleichgesetzt mit *tangled code*. Es gibt bislang kaum Anstöße, die Frage der quer schneidenden Belange in den Sprachen der höheren Ebenen wie Analyse und Entwurf zu behandeln. Die mir bekannten Veröffentlichungen zu Themen wie „Aspektorientierter Entwurf“ oder „Aspekte in UML“ betrachten überwiegend Implementierungsmodelle, sie gehen der Frage nach, wie

sich die Aspekte der *Implementierungsebene* im Entwurf zeigen oder wie sie sich in UML ausdrücken lassen.

Eine Ausnahme könnte hier *Intentional Programming* (3.8) darstellen, das sich bemüht vollständig von konkreten Sprachen und den damit verbundenen Limitationen zu abstrahieren. Prinzipiell ist die Idee der Intentionen auch auf die höheren Sprachen von Analyse und Entwurf übertragbar. Der geringe Umfang der Veröffentlichungen über Intentional Programming bieten jedoch keine geeignete Grundlage für eine abschließende Beurteilung in diesem Punkt.

Die grundsätzlich starke Fokussierung auf Implementierungssprachen ist verständlich. Intuitiv steigt die Wahrscheinlichkeit, dass ein Belang quer schneidend ist (und damit zum „Problem“ wird) mit dem Formalisierungsgrad der Sprache, in der wir ihn ausdrücken müssen. Von den bei der Softwareentwicklung verwendeten Sprachen haben die Implementierungssprachen das stärkste formale Korsett, ganz offensichtlich ist die Not hier am größten. Zeigt sich das Problem der Querschnittseigenschaften auf dieser Ebene doch schon bei so alltäglichen Belangen wie Synchronisation oder Ablaufverfolgung.

Das hier vorgestellte erweiterte Hyperspace-Modell zielt auf die höheren Ebenen des Softwareentwurfs ab. Es unterstützt im Besonderen die dort übliche simultane Verwendung mehrerer Artefaktsprachen. Insofern ist ein direkter Vergleich mit der aspektorientierten Programmierung wenig zielführend, das erweiterte Modell kann vielmehr als *Ergänzung* zu AOP verstanden werden. Ich bin mir sicher, dass sich das Problem der quer schneidenden Belange (Aspekte) in nächster Zeit auch oder gerade in Analyse- und Entwurfssprachen, wie den Sprachen der UML, zeigen wird. Hier tut sich ein weites Feld für zukünftige Forschung und Entwicklung auf. Das erweiterte Hyperspace-Modell bietet einen viel versprechenden Ansatz für derartige Untersuchungen.

5.4.2. Vergleich mit Hyper/J

Die Idee des *Hyperspace-Modells*, so wie in Abschnitt 4.1 vorgestellt, ist, die Integration von Artefakten über *alle Phasen* des Softwareentwicklungszyklus und *sämtliche dabei verwendeten Sprachen* zu realisieren [TOHS99, OT99]. HAROLD OSSHER und PERI TARR haben den Begriff des *multidimensionalen Trennens der Belange* geprägt. Von ihnen stammt die Idee, Belange auf Belangdimensionen abzubilden und Artefakte in Grundelemente zu zerlegen, die dann den Belangen zugeordnet werden.

Hyper/J, die erste und bislang einzige Instantiierung des Hyperspace-Modells, beschränkt sich mit Java jedoch wiederum auf *eine* Sprache aus der *Implementierungsebene*, die bearbeiteten Modelle sind wiederum *Implementierungsmodelle*. Hyper/J abstrahiert durch die Zuordnung von Java-Elementen zu abstrakten Belangen vergleichsweise stärker als die aspektorientierten Programmierung – und das ist zweifellos ein Gewinn. Dennoch dürfte unstrittig sein, dass damit noch kein echtes multidimensionales Trennen der Belange erreicht ist. Dafür ist eine Integration der in den früheren Phasen des Softwareentwicklungszyklus verwendeten Sprachen unumgänglich.

Wie die in dieser Arbeit durchgeführten Überlegungen zeigen, darf bezweifelt werden, dass eine derartige Integration mit dem Hyperspace-Modell von Hyper/J, das ich im Fol-

genden als *klassisches (Hyperspace-)Modell* bezeichne, gelingt. Die formalen Voraussetzungen des klassischen Modells, insbesondere die Raumeigenschaft, erwiesen sich als ungeeignet für die Integration weiterer Sprachen – Sprachen, wie sie vor allem bei Analyse und Entwurf genutzt werden. Aus diesem Grund wurde das erweiterte Modell entwickelt.

Das erweiterte Modell baut somit direkt auf der Idee des multidimensionalen Trennens der Belange mit Hyperspaces auf. Es verwendet bewusst dieselbe Terminologie. Einige Begriffe haben dabei eine generalisierte Bedeutung erhalten. Andere sind entfallen oder sind neu hinzugekommen. Im Folgenden sollen die Ähnlichkeiten und Unterschiede der Modelle im Detail diskutiert werden.

5.4.2.1. Gemeinsame Grundlagen

Software besteht aus *Artefakten*, beschreibenden Dokumenten abgefasst in einer *Artefaktsprache*. Artefakte bestehen wiederum aus Instanzen der syntaktischen Konzepte einer Artefaktsprache, den *Grundelementen*. Ein *Belangraum* enthält alle Grundelemente eines Softwaresystems. Er ist strukturiert anhand von *Belangdimensionen*. Eine Belangdimension repräsentiert dabei eine Art von Belangen, jeder konkrete Belang liegt in genau einer Belangdimension. Grundelemente werden Belangen zugeordnet, ein Belang bekommt all diejenigen Grundelemente zugeordnet, die diesen Belang adressieren.

5.4.2.2. Der wesentliche Unterschied: Die Raumeigenschaft

An dieser Stelle zeigt sich der wohl wesentlichste Unterschied zwischen den Modellen. Das klassische Modell betrachtet die Dimensionen als *orthogonale Achsen eines diskreten, geometrischen Raumes*, Belange werden als Punkte auf den Achsen und Grundelemente als Punkte im Raum dargestellt. Die Zuordnung von Grundelementen zu Belangen ist damit eine *Abbildung*, jedes Grundelement ist *genau einem* Belang in jeder Dimension zugeordnet.

Das erweiterte Modell sieht Belangdimensionen hingegen als ein *Behälter* für Belange einer Art. Die Zuordnung von Grundelementen zu Belangen ist eine *Relation*, ein Grundelement kann damit auch *mehreren* oder *gar keinem* Belang einer Belangdimension zugeordnet werden.

Das klassische Modell betreibt beträchtlichen Aufwand zur Aufrechterhaltung der Raumeigenschaft. Diese führt zur Definition einer Reihe zusätzlicher Begriffe: Jede Belangdimension hat genau einen *Nullbelang*, dem diejenigen Grundelemente zugewiesen werden, die in der Belangdimension keine Bedeutung haben. Des Weiteren gibt es das Konzept der *deklarativen Elemente*. Steht ein Grundelement A mit einem weiteren Grundelement B in Beziehung, wobei B jedoch nicht demselben Belang wie A zugeordnet ist, so wird in den Belang von A eine Deklaration von B aufgenommen. Genau genommen wird die Deklaration jedoch nicht in den Belang von A aufgenommen, sondern in einen *Hyperslice*. Hyperslices dienen der Verwaltung der deklarativen Elemente, sie sind *deklarativ vollständige* Belange. Zur Verwaltung der Hyperslices ist es wiederum erforderlich, eigene *Hyperslice-Dimensionen* einzuführen. Faktisch führt dies zu einer Verdopplung des Raumes.

All diese zusätzlichen Begriffe sind im erweiterten Modell obsolet, sie werden nicht mehr benötigt. Die Unterscheidung zwischen Deklarationen und Definitionen erwies sich als untauglich für höhere Artefaktsprachen, durch die Aufgabe der Raumeigenschaft ist sie außerdem formal nicht mehr erforderlich. Damit ist das erweiterte Modell an dieser Stelle deutlich einfacher und verständlicher. Die im klassischen Modell vorgenommene Unterscheidung zwischen Belangen und Hyperslices und die damit verbundene Verdopplung der Dimensionen wirkt vergleichsweise kompliziert und unelegant.

5.4.2.3. Neue Begriffe und Generalisierungen

Das erweiterte Modell unterscheidet zwischen *artefaktbasierenden* und *selektierenden* Belangdimensionen. Artefaktbasierende Belangdimensionen fußen auf konkreten Artefakten, sie bringen die Grundelemente in den Hyperspace. Die Belange einer artefaktbasierenden Belangdimension haben üblicherweise ein zugeordnetes *repräsentatives Grundelement*. Letztere wurden aus der Beobachtung heraus eingeführt, dass konkrete Artefaktsprachen meistens zur Modellierung bestimmter Belange dienen und sich genau diese Belange sinnvollerweise auch als Entitäten (Grundelemente) in den Artefakten wieder finden. Selektierende Belangdimensionen fügen dem Hyperspace hingegen keine weiteren Grundelemente hinzu, sie haben demnach auch keine zugeordneten repräsentativen Grundelemente. Selektierende Belangdimensionen stellen zusätzliche Sichten auf die Menge der Grundelemente bereit.

Die Unterscheidung zwischen artefaktbasierenden und selektierenden Belangdimensionen findet sich implizit auch im klassischen Modell. Sie fällt dort nicht auf, da Hyper/J nur eine artefaktbasierende Belangdimension unterstützt: Die Dimension <Classes>, die zudem immer automatisch generiert wird. An dieser Stelle wird noch mal deutlich, wo Hyper/J und das klassische Hyperspace-Modell die Idee des multidimensionalen Trennens der Belange nicht konsequent umsetzen: Multidimensionales Trennen der Belange funktioniert nur, wenn die simultane Verwendung mehrerer *artefaktbasierender* Belangdimensionen möglich ist.

Im erweiterten Modell erfordert die Unterstützung mehrerer artefaktbasierender Belangdimensionen darüber hinaus eine Integration der *Metamodelle* von Artefaktsprachen. Soll ein Artefakt einer bislang nicht verwendeten Sprache in den Hyperspace integriert werden, so muss zunächst eine Integration des zugehörigen Metamodells erfolgen. Dabei sind im Regelfall zusätzliche Beziehungstypen zwischen den Grundelementtypen verschiedener Artefaktsprachen zu definieren und deren konkrete Instanzen zu identifizieren. Erst durch diese Integration auf Schema- und Modellebene führt das Hinzufügen neuer Artefakte in den Hyperspace zu einem echten Informationsgewinn.

Auch der klassische Ansatz verwendet *implizit* ein Metamodell, dort ist es das Metamodell von Java. Aufgrund der Beschränkung auf nur eine Artefaktsprache ist das Metamodell in Hyper/J jedoch abgeschlossen und kann hart codiert sein.

Die zusätzlichen Schritte der Definition weiterer artefaktbasierender Belangdimensionen sowie der Integration von Metamodellen und Modellen müssen außerdem in den Arbeitsabläufen beim Umgang mit dem erweiterten Modell an gegebener Stelle berück-

sichtigt werden. Das erweiterte Hyperspace-Prozessmodell trägt diesen Erfordernissen bei der Integration und durch den neu hinzugekommenen Schritt der Projektion Rechnung.

5.4.2.4. Kombination und Modellmanagement

Das integrierte Metamodell spielt außerdem bei der *Kombination* eine wichtige Rolle. Der Kombinationsprozess transformiert die durch eine Menge von Belangen gewählten Teilmodelle (*Hypermodule*) in ein gültiges Ausgabemodell – gültig entsprechend des integrierten Metamodells. Dieses gilt sowohl für das klassische wie auch für das erweiterte Modell. Durch das abgeschlossene Metamodell ist der Kombinationsprozess in Hyper/J jedoch ebenso abgeschlossen. Im Unterschied dazu muss der Kombinationsprozess beim erweiterten Modell flexibel erweiterbar sein. Bei der Integration eines weiteren Metamodells wird es in der Regel erforderlich sein, den Kombinationsprozess entsprechend anzupassen.

Das erweiterte Modell ist daher sowohl beim Hinzufügen neuer Artefakte und ihrer Metamodelle als auch für den Kombinationsprozess auf Verfahren zur Integration, Manipulation und Transformation von generischen Modellen angewiesen. Untersuchungen über derartige Verfahren zum Modellmanagement finden bislang überwiegend im Datenbanken-Bereich statt [BHP00]. Die Übertragung und Erweiterung auf die eher graphenorientierten Modelle der Softwaretechnik bietet ein wichtiges Feld für zukünftige Forschung.

5.4.2.5. Beziehungen zwischen Belangen

Ein Kritikpunkt am klassischen Modell ist die fehlende Möglichkeit Beziehungen, wie Aggregation oder Spezialisierung, zwischen Belangen darstellen zu können. Durch derartige *und/oder*-Zergliederungen ließen sich einzelne Belange weiter unterteilen und so z.B. Abhängigkeiten zwischen Belangen ausdrücken. Dieser Punkt ist auch im erweiterten Modell bislang nicht adäquat umgesetzt. Hier kann man sich allerdings meistens damit helfen, die repräsentativen Grundelemente der Subbelange dem übergeordneten Belang zuzuordnen.[31] Dennoch halte ich eine Ergänzung des Modells um typisierte Beziehungen zwischen Belangen für eine sinnvolle zukünftige Erweiterung.

5.4.2.6. Fazit

Wie die Diskussion gezeigt hat, ist das erweiterte Hyperspace-Modell in allen Punkten eine echte Generalisierung des klassischen Modells. Alle zusätzlich eingeführten Begriffe sind implizit auch im klassischen Modell vorhanden, selbiges gilt für die zusätzlichen Schritte des erweiterten Hyperspace-Prozessmodells. Es hat sich weiterhin gezeigt, dass die Generalisierung das Modell nicht unbedingt auch komplizierter macht. Durch den Verzicht auf die Raumeigenschaft ist es stellenweise sogar deutlich einfacher.

Eines sollte an dieser Stelle jedoch auch deutlich gemacht werden: Der Vergleich zwischen dem klassischen und dem erweiterten Ansatz erfolgte hier rein auf der Modell-

[31] So wurde im Beispiel bei dem Anwendungsfall *Fundstücke inventarisieren* und seinen drei Varianten vorgegangen.

ebene. Hyper/J ist jedoch nicht nur ein Modell, es ist auch ein konkretes Werkzeug und erfreut sich in dieser Funktion großer Anerkennung.

5.4.3. Das erweiterte Modell als Testumgebung für Artefakte und Sprachen

Bei der Integration eines Softwaresystems in einen Hyperspace wird man gezwungen, Belange zu benennen und sauber zu trennen. Das führt dazu, dass man ein tieferes Verständnis über das Softwaresystem erhält.

Multidimensionales Trennen der Belange mit dem erweiterten Hyperspace-Ansatz könnte jedoch außerdem aus einer anderen Hinsicht interessant sein: Bei der Integration von Artefakten und deren Metamodellen in einen Hyperspace lernt man eine Menge über die verwendeten Artefakte und ihre *Sprachen*. Damit könnten Hyperspaces auch als eine Art „Lackmustest“ für Sprachen verwendet werden. Gelingt es nicht, die in einer Sprache modellierten Belange zu benennen und auf *einer* Belangdimension abzubilden, oder fällt das Aufstellen und Integrieren des Metamodells schwer, so könnte dies als Indikator für Mängel in der Konzeption einer Sprache gewertet werden.

Im Hinblick auf die Entwicklung von Sprachen und Artefakten sind auch die selektierenden Belangdimensionen interessant. Eine selektierende Belangdimension hat keine zugrunde liegenden Artefakte, ihre Belange keine repräsentativen Grundelemente. Stellt sich jedoch heraus, dass eine solche Belangdimension für das Softwaresystem eine große Bedeutung hat, so kann dieses als ein Hinweis auf *fehlende Artefakte* gewertet werden. Eine weitere Untersuchung mag dann zu der Erkenntnis führen, dass für die benötigten Artefakte eine *geeignete Sprache fehlt*.

Damit könnte das erweiterte Hyperspace-Modell auch als eine Umgebung für die praxisorientierte Entwicklung neuer Sprachen dienen. Stellt man fest, dass es für ein eigentlich benötigtes Artefakt keine geeignete Sprache gibt, so kann man versuchen, eine eigene zu entwickeln. Das geht vergleichsweise einfach, da das interne Metamodell des erweiterten Hyperspace-Ansatzes minimal sein kann. Die Metamodellierung der Sprache ist nur für die Elemente erforderlich, die in den Artefakten tatsächlich verwendet werden, die Sprache kann also inkrementell „nach Bedarf“ weiterentwickelt werden. Akzeptierte man außerdem die Eignung von Hyperspaces als „Lackmustest“ für Sprachen, so wäre für die neue Sprache automatisch eine gewisse Mindestqualität sichergestellt. Bewährt sich eine derartige Sprache dann in mehreren Projekten, so könnte man darangehen, sie weiter zu generalisieren, um sie als „Haussprache“ oder sogar noch weiter zu propagieren.

Kapitel 6

Zusammenfassung und Ausblick

6.1. Zusammenfassung

Die Unterstützung des *Prinzips der Trennung der Belange* ist in der Softwaretechnik eine treibende Kraft bei der Entwicklung neuer Paradigmen, Methoden und Werkzeuge. Dies führte über strukturierte Programmierung und Modulkonzept bis zur Objektorientierung, die sich als das beherrschende Paradigma bei der Entwicklung von Softwaresystemen durchgesetzt hat.

In den letzten Jahren wurde zunehmend deutlich, dass auch die Objektorientierung das Prinzip der Trennung der Belange noch nicht ausreichend unterstützt. Es gibt eine Reihe von Belangen, die sich mit den Strukturierungsmitteln der objektorientierten Dekomposition nicht vernünftig separieren lassen. Diese *quer schneidenden Belange* führen daher zu *tangled code*, sie finden sich im Programmcode verteilt über mehrere Methoden und Klassen wieder. Typische quer schneidende Belange sind *Synchronisation*, *Ablaufverfolgung*, *entfernte Prozeduraufrufe* oder *Fehlerbehandlung*.

Quer schneidende Belange werden auch als *Aspekte* bezeichnet. Ansätze, die versuchen das Problem der quer schneidenden Belange zu lösen, entsprechend Ansätze zur *aspektorientierten Programmierung (AOP)*. AOP ist ein hochgradig aktives Feld in der Forschungslandschaft und versteht sich nicht als Alternative, sondern als Ergänzung zur Objektorientierung.

In dieser Arbeit wurden mit *Adaptive Programming*, *Subject-oriented Programing*, *Composition Filters* und *AspectJ* die bekanntesten Ansätze zur aspektorientierten Softwareentwicklung vorgestellt. Den genannten Ansätzen ist gemein, dass sie das Problem der quer schneidenden Belange ausschließlich auf der *Implementierungsebene* angehen, sie arbeiten allesamt auf Code-Modellen.

Weitgehend ununtersucht ist bislang die Frage der quer schneidenden Belange auf den höheren Ebenen der Softwareentwicklung. Zweifellos gibt es derartiges jedoch auch auf der *Entwurfsebene*. Entwurfsmuster sind ein Beispiel für quer schneidende Entwurfselemente, ihre Anwendung findet sich oftmals verteilt über eine ganze Klassenhierarchie wieder.

Ebenfalls zu den AOP-Ansätzen zählend, geht die Idee des *multidimensionalen Trennens der Belange* mit dem *Hyperspace-Modell* von HAROLD OSSHER und PERI TARR einen deutlichen Schritt weiter. Nach dem Hyperspace-Modell, das den Schwerpunkt dieser Arbeit bildet und im Detail vorgestellt wurde, treten Belange in unterschiedlichen Arten oder Gestalten auf, die *Dimensionen* genannt werden. Die Ursache für das Problem der quer schneidenden Belange sehen OSSHER und TARR in der Tatsache, dass die bekannten Dekompositionstechniken immer eine Art von Belangen (und damit eine Dimension) bevorzugen, während sie andere unterdrücken. Das Hyperspace-Modell tritt dem entgegen indem es eine simultane Trennung der Belange entlang beliebig vieler Dimensionen und über alle Phasen des Softwarelebenszyklus erlaubt.

Die bislang einzige Realisierung des Hyperspace-Modells findet sich in *Hyper/J*, das ebenfalls in dieser Arbeit betrachtet wurde. Hyper/J ermöglicht multidimensionales Trennen der Belange für die Sprache Java und damit die Gliederung von Java-Codeelementen entlang zusätzlicher Belangdimensionen. Wiederverwendung und Wartung von Java-Code werden so erheblich erleichtert.

Durch die Beschränkung auf Java als einzige Artefaktsprache unterstützt Hyper/J ebenfalls ausschließlich die Implementierungsebene. Die Unterstützung beliebiger Artefaktsprachen, insbesondere auch derer, die in den frühen Phasen wie Analyse und Entwurf verwendet werden, ist jedoch unabdingbare Vorraussetzung für ein echtes multidimensionales Trennen der Belange.

Wie die in dieser Arbeit durchgeführten Untersuchungen zeigen, ist die Integration von weiteren Sprachen mit dem bestehenden Hyperspace-Modell kaum möglich. Die formalen Voraussetzungen des bestehenden Modells, insbesondere die Raumeigenschaft, erwiesen sich als ungeeignet für die weniger formalen Sprachen, die bei Analyse und Entwurf zum Einsatz kommen.

In dieser Arbeit wurde daher das *erweiterte Hyperspace-Modell* entwickelt und anhand eines Beispiels vorgestellt. Das erweiterte Modell zielt vorrangig auf simultane Verwendung mehrerer Artefaktsprachen ab und baut direkt auf den Konzepten und Begriffen des bestehenden Modells auf.

Eine zentrale Rolle kommt im erweiterten Modell den Metamodellen zu. Eine sinnvolle Integration von Entwurfsdokumenten verschiedener Artefaktsprachen ist nur möglich, wenn zunächst die zugrunde liegenden Metamodelle integriert werden.

Es konnte gezeigt werden, dass das erweiterte Modell eine echte Generalisierung des bestehenden Modells darstellt. Trotz der Erweiterungen fällt es, aufgrund des Verzichtes auf die Raumeigenschaft, in seiner Gesamtheit sogar eher einfacher aus.

6.2. Ausblick

Das erweiterte Hyperspace-Modell bietet die Möglichkeit für ein echtes multidimensionales Trennen der Belange über alle Phasen des Softwarelebenszyklus und sämtliche dabei verwendeten Artefaktsprachen. Damit bildet das in dieser Arbeit entwickelte Modell

eine viel versprechende Ausgangsbasis für weitergehende Aktivitäten in Forschung und Entwicklung:

Vervollständigen des Modells: Das erweiterte Modell konnte in dieser Arbeit seine grundsätzliche Anwendbarkeit für unterschiedliche Problemstellungen an einem Beispiel zeigen. Es befindet sich jedoch zweifellos noch in einem sehr frühen Stadium. Neben weiteren Tests anhand von zusätzlichen Beispielen steht auch noch eine Formalisierung des Modells aus. Weitere zu beantwortende Fragen sind in diesem Zusammenhang:

- Wie lassen sich die in der Praxis relevanten Artefaktsprachen in einen Hyperspace abbilden? Wo tauchen hier besondere Schwierigkeiten auf?
- Das erweiterte Modell geht davon aus, dass ein einzelnes Artefakt immer nur Belange genau einer Belangdimension modelliert. Ist es eventuell notwendig, diese Einschränkung aufzuheben?
- Wie lässt sich das Modell sinnvoll erweitern, damit Beziehungen zwischen Belangen darstellbar sind? Denkbar sind hier z.B. Überlappung, Spezialisierung oder Komposition. Welche Beziehungstypen werden in der Praxis gebraucht?

Techniken zum Umgang mit generischen Modellen: Ein wichtiges Ziel des erweiterten Hyperspace-Modells ist die Unterstützung von beliebigen Artefaktsprachen. Dafür sind generische Verfahren zum Umgang mit Modellen erforderlich, sowohl für das Einbinden der Metamodelle und der Modelle während der Integration als auch bei der Transformation von Teilmodellen zu einem gültigen Ausgangsmodell während der Kombination. Allein hier tut sich ein weites Feld für zukünftige Forschung auf:

- In wie fern lässt sich bereits bei der Integration der Metamodelle beschreiben, wie die konkreten Modelle zu integrieren sind? Wie sähe eine geeignete Sprache für derartige Beschreibungen aus? Welche Arten von Modelltransformationen werden hierbei benötigt?
- Bei der Kombination muss, anhand einer Menge von Belangen, eine minimale aber sinnvolle Hülle der zu kombinierenden Grundelemente ermittelt werden. Wie weit lässt sich dieser Vorgang automatisieren? Ist es möglich und sinnvoll, diesen Vorgang bereits im integrierten Metamodell zu beschreiben?

Einsatz des erweiterten Modells bei der Entwicklung von Sprachen: Ein ebenfalls interessantes Projekt wäre zu untersuchen, in wie fern sich das erweiterte Hyperspace-Modell als Test- und Entwicklungsumgebung für Artefaktsprachen eignet:

- Welche Erkenntnisse lassen sich mit dem erweiterten Modell über Artefaktsprachen gewinnen? Wie lassen sich diese Erkenntnisse verwenden?
- Ist das erweiterte Hyperspace-Modell ein geeignetes Werkzeug für die Entwicklung von neuen oder die Verbesserung von bestehenden Sprachen?

Multidimensionales Trennen der Belange: Das Problem der quer schneidenden Belange wird bislang überwiegend auf der Implementierungsebene betrachtet. Ich bin überzeugt davon, dass sich quer schneidende Belange auch in den frühen Phasen des Softwareentwicklungsprozesses finden und dass ihre Identifizierung und Separierung gerade auch hier zu besser verständlicher, wartbarer und wieder verwendbarer Software führt. Entwurfsmuster wurden als Beispiel für die Entwurfsebene bereits angeführt:

- Welche weiteren quer schneidenden Belange lassen sich in den frühen Phasen der Softwareentwicklung finden? Wo bereiten sie Probleme?
- Welche Belangdimensionen sind in der Praxis relevant?
- Wie verändern sich Belange bei ihrer zunehmenden Konkretisierung durch den Softwareentwicklungsprozess? Werden z.B. quer schneidende Belange der Analyse auch zu quer schneidenden Belangen in Entwurf und Implementierung?

Gerade bei der letzten Fragestellung ist es erforderlich ein Modell zu haben, das den gesamten Softwareentwicklungszyklus und alle dabei verwendeten Sprachen unterstützt.

6.3. Schlusswort

Softwaresysteme gelten als inhärent komplex. FRED BROOKS schilderte dieses in seinem berühmten Buch *The Mythical Man-Month* [Bro75] mit den Worten:

„Die Komplexität ist eine grundlegende Eigenschaft von Software und keine zufällige."

Es ist Aufgabe der Softwaretechnik diese Komplexität beherrschbar zu machen. Multidimensionales Trennen der Belange mit dem erweiterten Hyperspace-Modell könnte ein wichtiger Schritt zu diesem Ziel sein.

Literatur

[Ale01] **Andrei Alexandrescu:** *Modern C++ Design – Generic Programming and Design Patterns Applied.* The C++ In-Depth Series, Addison-Wesley, 2001

[Aks96] **Mehmet Aksit:** *Separation and Composition of Concerns in the Object-Oriented Model.* In ACM Computing Surveys (CSUR) 28(4es), 1996

[AspektC++] Homepage des **AspektC++ Projekts** http://www.aspectc.org/

[AspektS] Homepage des **AspektS Projekts** von Robert Hirschfeld http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/

[AT98] **Mehmet Aksit, Bedir Tekinerdogan:** *Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters.* In Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP ´98), 1998

[BA01] **Lodewijk Bergmans, Mehmet Aksit:** *Composing Crosscutting Concerns Using Compsition Filters.* In Communications of the ACM (CACM) 44(10), pp. 51-57, 2001

[Ber94] **Lodewijk Bergmans:** *Composing Concurrent Objects.* Ph.D. thesis, University of Twente, The Netherlands, 1994

[BHP00] **Phillip A. Bernstein, Alon Y. Levy, Rachel A. Pottinger:** *A Vision for Management of Complex Models.* In SIGMOD Record (ACM Special Interest Group on Management of Data) 29(4), pp. 55-63, 2000

[BKF94] **Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein:** *A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification.* In Software Engineering Journal 20(10), pp. 760-773, 1994

[Boo94] **Grady Booch:** *Objektorientierte Analyse und Design.* Addison-Wesley, 1994

[Boost] Homepage der **Boost.org C++ Library Community** http://www.boost.org/

[Bro75] **Fred Brooks:** *The Mythical Man-Month.* Addison-Wesley, 1975

[CE00] **Krzysztof Czarnecki, Ulrich W. Eisenecker:** *Generative Programming – Methods, Tools, and Applications.* Addison-Wesley, 2000

[CE00a] **Constantinos A. Constantinides, Tzilla Elrad:** *Towards a Two-dimensional Separation of Concerns.* In Proceedings of the 15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´00), pp. 63-64, October 2000

[CE99] **Krzysztof Czarnecki, Ulrich W. Eisenecker:** *Components and Generative Programming (invited paper).* In Proceedings of the 7th European Engineering Conference, ACM SIGSOFT Software Engineering Notes 24(6), October 1999

[CER00] **Krzysztof Czarnecki, Ulrich W. Eisenecker, Lutz Röder:** *Freiheit für Entwickler – Intentional Programming: Programmierung ohne Sprache.* In IX – Magazin für Professionelle Computertechnik, November 2000, pp. 142-147

[CF] Homepage der **Aspect-Oriented Research on Composition Filters Group** der Universität Twente, Niederlande
http://trese.cs.utwente.nl/composition_filters/

[CKFS01] **Yvonne Coady, Gregor Kiczales, Mike Freeley, Greg Smolyn:** U*sing aspectC to improve the modularity of path-specific customization in operating system code*, ACM SIGSOFT Software Enginering Notes 26 (5), pp. 88-98, September 2001

[Dij76] **Edsger W. Dijkstra:** *A Discipline of Programming*. Prentince-Hall, 1996

[Dud02] **Brier Dudley:** *Set to take software to a higher plane: Key Microsoft engineer, professor launch firm.* In The Seattle Times, September 17th, 2002
http://seattletimes.nwsource.com/html/businesstechnology/134536972_simonyi17.html

[EFB01] **Tzilla Elrad, Robert E. Filman, Atef Bader:** *Aspect-Oriented Programming.* In Communications of the ACM 44(10), pp. 29-32, 2001

[FS96] **Anthony Finkelstein, Ian Sommerville:** *The Viewpoints FAQ.* In Software Engineering Journal: Special Issue on Viewpoints for Software Engineering 11(1), pp. 2-4, 1996

[Gab02] **Bernhard Gabler:** *Ein Inventarisierungssystem für das Landesamt für Denkmalpflege – Bedarfsanalyse, Entwurf, prototypische Implementierung.* Diplomarbeit, Fachbereich Informatik, Universität Koblenz, Februar 2002

[GoF94] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:** *Design Patterns – Elements of Reusable Object Oriented Software.* Addison-Wesley Professional Computing Series, Addison-Wesley, 1994

[GV00] **Holger Giese, Alexander Vilbig:** *Towards Aspect-oriented Design and Architecture.* In Proceedings of the 15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´00). Workshop: Advanced Separation of Concerns, 2000

[Hil99] **Rich Hilliard:** *Aspects, Concerns, Subjects, Views, … *.* In Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´99). Workshop: Advanced Separation of Concerns, 1999

[HL95] **Walter L. Hürsch, Cristina V. Lopes:** *Separation of Concerns.* In Technical Report NU-CSS-9503, Northeaster University Boston, 1995
http://www.parc.xerox.com/csl/members/lopes/publications/techrep95/

[HO93] **William Harrison, Harold Ossher:** *Subject-Oriented Programming (A Critique of Pure Objects).* In Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´93). ACM SIGPLAN Notices 28(10), pp. 411-428, 1993

[HS] Homepage der **Hyperspace Group** bei IBM Research
http://www.research.ibm.com/hyperspace/

[Ken00] **Elizabeth A. Kendall:** *Aspect-oriented Programming in AspectJ™.* Proceedings of Evolve 2000 Conference, Sydney, 2000
http://www.pscit.monash.edu.au/~kendall/evolve2000.pdf

[Kic02] **Gregor Kiczales (Editor):** Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), Enschede, The Netherlands, April 2002

[KHH+01] **Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold:** *An Overview of AspectJ.* In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP ´01), pp. 327-353, 2001

[KKO+02] **Doug Kimelman, Vincent Kruskal, Harold Ossher, Tova Roth, Peri Tarr:** *HyperProbe – An Aspect-Oriented Instrumentation Tool for Troubleshooting Large-Scale Production Systems.* Demonstration at the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), Enschede, Netherlands, April 2002
http://trese.cs.utwente.nl/aosd2002/index.php?content=hyperprobe

[KS00] **Mohamed Mancona Kandé, Alfred Strohmeier:** *On the Role of Multi-Dimensional Separation of Concerns in Software Architecture.* Position Paper for the 15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000). Workshop: Advanced Separation of Concerns, 2000
http://lglwww.epfl.ch/~kande/Publications/role-of-mdsoc-in-swa.pdf

[Lie96] **Karl J. Lieberherr:** *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996

[Lie97] **Karl J. Lieberherr:** *Demeter and Aspect Oriented Programming*. Keynote. Smalltalk und Java in Industrie und Ausbildung Konferenz (STJA ´97), Erfurt, Deutschland, 1997
http://www-ia.tu-ilmenau.de/~czarn/generate/stja97/

[LL95] **Christina V. Lopes, Karl J. Lieberherr:** *AP/S++: Case-study of a MOP for Purposes of Software Evolution*. In Proceedings of Reflection ´96, San Francisco, California, April 1996
http://www.parc.xerox.com/csl/groups/sda/projects/reflection96/abstracts/lopes.html

[LOO01] **Karl J. Lieberherr, Doug Orleans, Johan Ovlinger:** *Aspect-Oriented Programming with Adaptive Methods*. Technical Report NU-CCS-2001-01, Northeastern University, Boston, February 2001
http://www.ccs.neu.edu/research/demeter/biblio/aspectual-methods.html

[Lop97] **Christina V. Lopes:** *D: A Language Framework for Distributed Programming*. Ph.D. Thesis, Notheastern University, Boston, Februar 1997

[MWY90] **Satoshi Matsuoka, Ken Wakita, Akinori Yonezawa:** *Synchronization constaints with inheritance: What is not possible – so what is?* Technical Report 10, Department of Information Science, The University of Tokyo, 1990
http://www.is.s.u-tokyo.ac.jp/tech-reports/FILES.html

[OHKK95] **Harold Ossher, William Harrison, Alexander Katz, Vincent Kruskal:** *Subject-Oriented Composition Rules.* In Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´95). ACM SIGPLAN Notices 30(10), pp. 235-250, 1995

[OKK+96] **Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, Vincent Kruskal:** *Specifying Subject-Oriented Composition.* In In Theory and Practice of Object Systems 2(3), pp. 179-202, 1996

[OT00] **Harold Ossher, Peri Tarr:** *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2000

[OT00a] **Harold Ossher, Peri Tarr:** *Hyper/J: multi-dimensional separation of concerns for Java.* In Proceedings of the 22nd International Conference on Software Engineering (ICSE ´00), pp. 734-737, 2000

[OT99] **Harold Ossher, Peri Tarr:** *Multi-Dimensional Separation of Concerns in Hyperspace.* Research Report RC21452(96717)16APR99, IBM Research Division, April 1999
http://www.research.ibm.com/hyperspace/Papers/tr21452.ps

[Par72] **David L. Parnas:** *On the Criteria to be used in Decomposing Systems into Modules.* In Communications of the ACM 15(12), pp. 1053-1058, 1972
Reprinted as ACM Classic of the Month, May 1996
http://www.acm.org/classics/may96/

[PC01] **J. Andrés Díaz Pace, Marcelo R. Campo:** *Analyzing the Role of Aspects in Software Design.* In Communications of the ACM 44(10), pp. 67-73, 2001

[Röd99] **Lutz Röder:** *Transformation and Visualization of Abstractions using the Intentional Programming System.* First International Symposium on Generative and Component-Based Software Engineering (GCSE ´99), Erfurt, Deutschland, 1999
http://www.aisto.com/roeder/paper/gcse99.pdf

[SDA] Homepage der **Software Design Area Group** am XEROX PARC
http://www.parc.xerox.com/csl/groups/sda

[SGS02] **Olaf Spinczyk, Andreas Gal, Wolfgang Schröder-Preikschat:** *AspectC++: An Aspect-Oriented Extension to C++.* In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, February, 2002

[Sim95] **Charles Simonyi:** *The Death of Computer Languges – The Birth of Intentional Programming.* Technical Report, Microsoft Research (MSR-TR-95-52), 1995 ftp://ftp.research.microsoft.com/pub/tech-reports/Summer95/TR-95-52.doc

[Sim96] **Charles Simonyi:** *Intentional Programming - Innovation in the Legacy Age.* Presented at the 1996 meeting of the International Federation for Information Processing, Working Group 2.1 (IFIP WG 2.1), June 4, 1996 http://www.aisto.com/roeder/active/ifip96.pdf

[SLL99] **Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine:** *The generic graph component library.* In Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´99). ACM SIGPLAN Notices 34(10), pp. 399-414, 1999

[SOP] Homepage der **Subject Oriented Programming Group** bei IBM Research http://www.research.ibm.com/sop/

[Sul01] **Gregory T. Sullivan:** *Aspect-Oriented Programming using Reflection and Metaobject Protocols.* In Communications of the ACM 44(10), pp. 95-97, 2001

[SY99] **Junichi Suzuki, Yoshikazu Yamamoto:** *Extending UML with Aspects: Aspect Support in the Design Phase.* In Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP ´99). Workshop: Aspect-Oriented Programming (AOP), 1999

[TO99] **Peri Tarr, Harold Ossher:** *Hyper/J™ User and Installation Manual.* 1999 http://www.research.ibm.com/hyperspace/

[TOHS99] **Peri Tarr, Harold Ossher, William Harrison, Stanley M. Sutton Jr.:** *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In Proceedings of the 21nd International Conference on Software Engineering (ICSE ´99), pp. 107-119, May 1999

[Vel96] **Todd Veldhuizen:** *Rapid Linear Algebra in C++.* In Dr. Dobb's Journal, August 1996

[Vlis98] **John Vlissides:** *Pattern Hatching – Subject-Oriented Design.* In C++ Report, Februrary 1998 http://www.research.ibm.com/designpatterns/pubs/ph-feb98.pdf

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Hilfsmittel benutzt und alle Stellen, die anderen Werken dem Sinn oder Wortlaut nach entnommen sind, unter Angabe der Quelle kenntlich gemacht habe.

Koblenz, 30. September 2002

Daniel Lohmann