Bachelorarbeit

Rick Eisermann

Aspektorientierte Programmierung für Smalltalk

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Rick Eisermann

Aspektorientierte Programmierung für Smalltalk

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Böhm
Zweitgutachter: Prof. Dr. Guido Pfeiffer

Abgegeben am 24. Oktober 2007

Rick Eisermann

Thema der Bachelorarbeit
Aspektorientierte Programmierung für Smalltalk

Stichworte
aspektorientierte Programmierung, symmetrische aspektorientierte Programmierung, dynamische aspektorientierte Programmierung, aspektbewusste Refaktorisierung, Smalltalk

Kurzzusammenfassung
Aspektorientierte Programmierung ermöglicht eine bessere Modularisierung als klassische Ansätze der Softwarezerlegung. Diese Arbeit behandelt den Entwurf und die Implementation von AspectTalk, einer Mehrzweckerweiterung Smalltalks um dynamische, symmetrische aspektorientierte Programmierung im Bereich der Verhaltensanpassung. Es werden keine neuen Sprachkonstrukte verwendet, sondern vollständig auf den Eigenschaften und Möglichkeiten Smalltalks aufgebaut. Darüber hinaus bettet sich das Rahmenwerk nahtlos in die Entwicklungsumgebung ein. Außerdem wird aufgezeigt wie Refaktorisierungsauswirkungen auf Aspektorientierung mittels Smalltalks Reflexion und aspektorientierter Programmierung selbst begegnet werden kann.

Rick Eisermann

Title of the paper
Aspect-oriented Programming for Smalltalk

Keywords
aspect-oriented programming, symmetrical aspect-oriented programming, dynamic aspect-oriented programming, aspect-aware refactoring, Smalltalk

Abstract
Aspect-oriented programming facilitates better modularisation than classical approaches to software decomposition. This thesis deals with the design and implementation of AspectTalk, a general purpose augmentation of Smalltalk to dynamic, symmetrical aspect-oriented programming in the field of behaviour modification. It uses no new language constructs but bases upon the features and possibilities of Smalltalk. Furthermore the framework is seamlessly embedded into the development environment. Besides it is shown how the effects of refactoring on aspect-orientation can be faced by means of Smalltalk's reflection and aspect-oriented programming itself.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Modularisierung ist ein grundlegender Ansatz in der Softwareentwicklung. Dabei wird das Gesamtsystem in Untersysteme zerlegt, bis die Tiefe der Gliederung und der Umfang der Einzelsysteme angemessen erscheint. Angemessen ist die Auflösung dann, wenn jedes Modul der Umsetzung genau eines Anliegens dient, wobei ein Anliegen eine Anforderung, ein Merkmal oder eine einfache Aufgabe ist.
Modularisierung bietet zwei wesentliche Vorteile. Zum einen ermöglicht sie die Strukturierung des Gesamtsystems und damit die Wiederverwendung und Austauschbarkeit der einzelnen Untersysteme. Zum anderen dient sie der Komplexitätsreduktion, da vom Gesamtproblem abstrahiert werden kann und sich die Lösung auf die Bearbeitung einfacherer und kleinerer Teilprobleme konzentriert.
Allerdings ist Art und Feinheit der Modularisierung abhängig von den Mitteln und Möglichkeiten der verwendeten Programmiersprache. Eine natürliche (KIL+97), problemspezifische Modularisierung umzusetzen ist oftmals unmöglich oder aufgrund der gegebenen Möglichkeiten äußerst umständlich. In diesem Zusammenhang spricht man auch von der Tyrannei der vorherrschenden Zerlegung (OT99).

Aspektorientierte Programmierung verspricht hier Abhilfe und zeigt ihre Erfolge vor allem in Form von AspectJ, einer aspektorientierten Erweiterung Javas. Darüber hinaus findet die Aspektorientierung aber kaum Verbreitung, obwohl sie bereits älter als zehn Jahre ist. Zwar gibt es auch Realisierungen für andere Sprachen. Doch werden diese nicht in dem Umfang (kommerziell) eingesetzt wie AspectJ. Einerseits hängt dies sicherlich mit der derzeitigen Verbreitung und Beliebtheit Javas zusammen. Bedeutsam ist auch, dass AspectJ von den Urhebern aspektorientierter Programmierung stammt und somit die älteste Lösung ist. Wirklich ausschlaggebend erscheint aber die gebotene Werkzeugunterstützung. Neben der reinen Möglichkeit aspektorientiert zu programmieren verfügt AspectJ über eine Reihe von Zusatzprogrammen, die eng in gängige Entwicklungsumgebungen wie z.B. Eclipse eingebunden sind. Dies erhöht den Nutzen erheblich, da die aspektorientierte Programmierung so zu einer einfach benutzbaren und gängigen Technik wird.

1.2 Zielsetzung

Ziel dieser Arbeit ist der Entwurf und die Implementation einer aspektorientierten Erweiterung der Sprache Smalltalk. Dies beinhaltet eine Erörterung der grundlegenden Konzepte aspektorientierter Programmierung, ihrer Umsetzung in Smalltalk und die Entwicklung (einfacher) Werkzeuge zur Anwenderunterstützung. Neben den Anforderungen aspektorientierter Programmierung ergeben sich durch die Verwendung Smalltalks eine Reihe weiterer Anfor-

derungen, die an die für die Sprache und ihre Entwicklungsumgebung typischen Merkmale angelehnt sind. Somit ergeben sich die folgenden Kriterien anhand derer AspectTalk entwickelt worden ist:

- **Offenheit und Erweiterbarkeit:** Der Quelltext von Smalltalkprogrammen ist jederzeit einsehbar oder lässt sich durch Dekompilierung zurückgewinnen. Daher steht es Programmieren frei nach ihren Wünschen Anpassungen und Erweiterungen an fremden Programmen vorzunehmen. Deshalb sollte ein Entwurf solche Änderungen durch klare Strukturen und Konzepte vereinfachen und sie problemlos integrieren.

- **Dynamik:** Die Entwicklung in Smalltalk gestaltet sich äußerst dynamisch. Zur Laufzeit ist das System beliebig änderbar. Klassen, Methoden und Variablen können zu jedem Zeitpunkt erzeugt, angepasst und entfernt werden. Diese Änderungen werden automatisch ausgeführt und wirken unmittelbar. Solch ein inkrementelles Vorgehen sollte auch mit aspektorientierter Programmierung möglich sein.

- **Reflexion:** In Smalltalk ist alles ein Objekt, selbst Klassen und Methoden. Die vollständige Vergegenständlichung des Systems und seiner Eigenschaften wird intensiv von Werkzeugen wie Objektinspektoren und Systembrowsern genutzt. Ebenso sollte eine aspektorientierte Erweiterung ihre Elemente zu Bürgern erster Klasse machen um es den Benutzern und besonders den Werkzeugen zu ermöglichen sie als gewöhnliche Smalltalkelemente zu betrachten und entsprechend mit ihnen zu arbeiten.

- **Werkzeugunterstützung:** Typische Entwicklungvorgänge wie das Erzeugen von Klassen oder Kompilieren von Methoden sind durch entsprechende Methoden und Objekte gänzlich in Smalltalk selbst möglich. Jedoch bietet eine Werkzeugunterstützung besseren Komfort und Arbeitserleichterungen als die rein manuelle Entwicklung. Dies gilt besonders für Visualisierung und wiederkehrende Abläufe. Daher sollte aspektorientiertes Programmieren ebenfalls durch (grafische) Werkzeuge unterstützt werden.

Zusammen mit der Bereitstellung aspektorientierter Möglichkeiten ist die Erfüllung dieser Forderungen grundlegend für die Akzeptanz und den Nutzen einer Erweiterung Smalltalks um aspektorientierte Programmierung.

1.3 Kapitelübersicht

Kapitel 2 ist eine Darstellung der wichtigsten Grundlagen auf denen diese Arbeit aufbaut. Dazu zählt eine Darstellung aspektorientierter Programmierung, eine kurze Einführung in die Sprache Smalltalk sowie die Vorstellung bisheriger aspektorientierter Lösungen für Smalltalk, die im Rahmen der Recherchen zu dieser Arbeit gefunden worden sind.

In Kapitel 3 werden die Konzepte von AspectTalk und die gebotene Funktionalität beschrieben. Dies beinhaltet die Umsetzung grundlegender Prinzipien der aspektorientierten Programmierung sowie die vorhandene Werkzeugunterstützung.

Kapitel 4 ist eine Beschreibung der wichtigsten Details des Entwurfs und der Implementation von AspectTalk.

In Kapitel 5 wird die notwendige Anpassung der Aspektorientierung bei Änderungen des Basisprogramms detaillierter betrachtet. Die Anpassung geschieht unter Ausnutzung der in Smalltalk gegebenen Reflexion und den Prinzipien aspektorientierter Programmierung selbst. Somit sind die gewonnenen Erkenntnisse auch auf andere Sprachen übertragbar.

Kapitel 6 umfasst eine Zusammenfassung der Ergebnisse dieser Arbeit sowie ein Ausblick auf mögliche Weiterentwicklungen von AspectTalk.

2 Grundlagen

In diesem Kapitel werden die wichtigsten Grundlagen auf denen diese Arbeit aufbaut vorgestellt. Zunächst werden die Grundzüge aspektorientierter Programmierung erläutert. Danach werden die für die aspektorientierte Programmierung wichtigsten Sprachelemente Smalltalks dargestellt. Aufbauend auf dieser Basis werden anschließend bisherige aspektorientierte Lösungen für Smalltalk beschrieben, die im Rahmen der Recherchen zu dieser Arbeit gefunden worden sind.

2.1 Aspektorientierte Programmierung

2.1.1 Charakterisierung

Das Konzept der aspektorientierten Programmierung entstammt der Arbeit einer Forschungsgruppe des Palo Alto Research Center um Gregor Kiczales (KIL+97). Sie hat die Ausdrucksfähigkeit von Programmiersprachen untersucht und herausgefunden, dass Programmiersprachen, die lediglich ein Abstraktionskonzept, z.B. funktionale oder objektorientierte Programmierung, unterstützen, für komplexe Softwaresysteme letztlich unangemessen sind.
Solche Programmiersprachen erlauben es nicht alle unabhängigen Entwurfsparameter durch unabhängige Programmkonstrukte abzubilden, da sie die Sprachgrenzen überschreiten (SGC02). Die lokale Isolierung bzw. Modularisierung bestimmter Entwurfsparameter oder Anliegen bzw. Aufgaben (engl. *seperation of concerns*) ist mit den gegebenen Sprachmitteln unmöglich, sodass ihre Implementierung stets reproduziert werden muss, sobald sie benötigt wird. Dies führt zu einer modulübergreifenden Verteilung dieser Anliegen, weshalb man auch von Querschnittsanliegen (engl. *cross-cutting concerns*) spricht. Diese Zerstreuung erzeugt zu pflegende Redundanzen und wechselseitige Abhängigkeiten, wodurch sich negative Auswirkungen auf Eigenschaften wie z.B. Wiederverwendung, Wartbarkeit oder Weiterentwicklung ergeben und somit die Softwarequalität gemindert wird (Lau03; CBE+00). Um diese negativen Konsequenzen zu vermeiden hat die Aspektorientierung daher den Anspruch den Programmierer zu befähigen Querschnittsanliegen modular auszudrücken. Jedes Anliegen soll so angemessen und separat wie möglich entwickelt und behandelt werden können. Durch diese strikte Trennung werden das System und seine Bestandteilte übersichtlicher und verständlicher, da jedes Modul nur tatsächlich ein Anliegen erfüllt. Weiterhin führt die Lokalisierung zu einer besseren Änderbarkeit und Wartbarkeit, da Anpassungen nur an einer zentralen Stelle vorzunehmen sind. Außerdem lässt sich neue Funktionalität nachträglich einfacher hinzufügen, da sich Entwurfsentscheidungen hinauszögern lassen. Insgesamt ergibt sich damit eine höhere Produktivität.

Um diesen Anforderungen gerecht zu werden ermöglicht die aspektorientierte Program-

mierung, allgemein formuliert, dem Entwickler Aussagen folgender Art machen zu können (FF00):

> Sobald in einem Programm P eine Bedingung C auftritt, führe Aktion A aus.

Der Programmierer spezifiziert ähnlich wie in kontextsensitiven Grammatiken eine Regel, die im Kontext C zur Produktion von A führt. Die Produktionsausführung wird dabei vom System gewährleistet und automatisch durchgeführt. Welche Bedingungen und Aktionen dabei zur Verfügung stehen, hängt von der Sprache und ihrer aspektorientierten Erweiterung ab.
In diesem Zusammenhang ist Transparenz (engl. *obliviousness*) eine wünschenswerte aber nicht notwendige Eigenschaft (Fil01). Sie drückt sich in unterschiedlichen Formen aus (SGS+05). Zu den wichtigsten Ausprägungen zählt, dass der Programmierer des Basisprogramms P keine speziellen Vorkehrungen für den Kontext C, z.B. Namenskonventionen oder Hookmethoden, zu treffen hat, damit die Aktionen A angewendet werden können und dass durch die Aspektorientierung bewirkte Veränderungen, z.B. Quelltextanpassungen, für ihn unsichtbar sind.

2.1.2 Kritik

Allerdings liegt in der Transparenz gleichzeitig der Hauptkritikpunkt aspektorientierter Programmierung (CSS). Sie führt dazu, dass das Programmverhalten nicht durch bloßes Betrachten des Quelltexts zu erschließen ist. Dies ist umso problematischer, wenn das eigentlich beabsichtigte Verhalten komplett verändert wird.
Die zugrunde liegende Nichtlokalität hat aber auch weitere Auswirkungen. Dies betrifft die Verletzung klassischer Eigenschaften der Modularisierung wie z.B. das Geheimnisprinzip oder die Datenkapselung (Lau03; Ste06), da von einem anderen Ort als dem Modul selbst Modulinterna einsehbar und veränderbar sind.
Jedoch ist anzumerken, dass die Möglichkeiten aspektorientierter Programmierung erst durch die entsprechende Verwendung und nicht per se negative Folgen haben. Es obliegt allein dem Programmierer diese Möglichkeiten verantwortungsvoll einzusetzen. Außerdem spielt Werkzeugunterstützung dabei eine wichtige Rolle, worauf in Abschnitt 3.5 eingegangen wird. Von der Hand zu weisen sind die vorgebrachten Kritkpunkte allerdings nicht.

2.1.3 Elemente

Nachfolgend werden die wichtigsten Elemente aus dem Bereich der Aspektorientierung erläutert. Die Mehrheit der Nomenklatur hat sich dabei unter dem Einfluss von AspectJ, der bekanntesten aspektorientierten Programmiersprache, etabliert.

Aspekt

Mit Aspekt ist ursprünglich ein Anliegen bezeichnet worden, das nicht sauber kapselbar, d.h. lokal modularisierbar, ist (KLM+97). Heutzutage wird dies als Querschnittsanliegen bezeichnet, während man unter Aspekt eher das Sprachelement zur Implementation des Anliegens versteht.

Anreicherung

Anreicherungen entsprechen den in Unterabschnitt 2.1.1 erwähnten Aktionen. Sie verändern die Struktur oder das Verhalten des Basisprogramms. So kann es sich z.B. um die Manipulation der Klassenstruktur (engl. *introduction*) oder die Ausführung zusätzlicher Methoden (engl. *advice*), z.B. bei Auftreten eines Variablenzugriffs, handeln.

Verbindungspunkt

Verbindungspunkte (engl. *join points*) sind durch Anreicherungen anpassbare Elemente des Basisprogramms. Damit entsprechen sie den in Unterabschnitt 2.1.1 erwähnten Bedingungen. Die unterschiedlichen Verbindungspunkte auf die verwiesen werden können werden auch mit dem Begriff Quantifikation (engl. *quantification*) bezeichnet. Verbindungspunkte lassen sich in zwei Dimensionen beschreiben. Einerseits gibt es strukturelle und verhaltensbasierte Verbindungspunkte. Zu der Struktur zählen z.B. Klassenschnittstellen, Vererbungsbeziehungen oder Klassenattribute. Zum Verhalten werden Laufzeitereignisse wie z.B. Ausnahmen, Methodenaufrufe oder Variablenzugriffe gerechnet. Andererseits lassen sie sich in statische und dynamische Verbindungspunkte unterteilen. Statische Verbindungspunkte sind durch Quelltextanalyse direkt bestimmbar, während dynamische Verbindungspunkte erst zur Laufzeit ermittelt werden können.

Verbindungspunktauswahl

Verbindungspunktauswahlen (engl. *pointcuts*) sind ein Mechanismus zur Spezifizierung von Verbindungspunkten. Auswahlen basieren auf strukturellen und Laufzeitmerkmalen. Zu strukturellen Eigenschaften zählen z.B. Methoden- oder Variablennamen, während z.B. dynamische Typen oder Kontrollflussabhängigkeiten zu Laufzeiteigenschaften zählen.

Aspektweber

Der Aspektweber ist dafür verantwortlich das Basisprogramm und die Aspekte bzw. Anreicherungen zu einem lauffähigen Gesamtprogramm zu vereinen. Der Name ist dadurch enstanden, dass dies ursprünglich durch Einweben zusätzlichen Quelltexts in das Basisprogramm erreicht worden ist (KIL+97; KLM+97). Jedoch ist dies nicht strikt erforderlich.

Aspektorientierte Programmierung befasst sich allgemein mit Programmiersprachen und -umgebungen und ist nicht an bestimmte Programmierstile wie z.B. objektorientierte Programmierung gebunden (KIL$^+$97; FF00; Fil01). Daher ist die Technik der Umsetzung sehr stark mit der zu betrachtenden Umgebung gekoppelt. Allerdings gibt es allgemeine Konzepte, anhand derer man eine konkrete Realisierung aspektorientierter Programmierung analysieren kann.

Ebene des Webens

Die Ebene des Webprozesses bestimmt wo und wann Anreicherungen eingebettet werden. Dies kann auf Quelltextebene vor oder während des Kompilierens geschehen, im Ergebnis der Kompilierung, z.B. Bytecode, stattfinden oder erst im Interpreter vollzogen werden. Damit bestimmt die Webebene den Transparenzgrad: Direkte Quelltextanpassungen sind unmittelbar einsehbar, während ein Mechanismus im Interpreter nicht ersichtlich ist.

Statisches und dynamisches Weben

Statisches Weben bedeutet, dass das Einbinden von Aspekten bzw. Anreicherungen einmal vor Programmstart geschieht und zur Laufzeit nicht mehr änderbar ist. Dieses Vorgehen wird häufig bei Quelltextmodifikationen verwendet. Spezielle Anweisungen werden direkt in den Basisquelltext eingefügt. Dadurch ergeben sich Optimierungsmöglichkeiten und nur geringe Leistungseinbußen in Vergleich zur manuellen Programmierung. Allerdings muss man dafür auf nachträgliches Hinzufügen und Entfernen von Anreicherungen verzichten. Das Programm müsste angehalten und neu übersetzt werden um Änderungen vorzunehmen.
Beim dynamischen Weben hingegen lassen sich Anreicherungen an jedem Punkt der Laufzeit einfügen und löschen. Dies ist häufig in aspektorientierten Erweiterungen zu finden, die auf Interpreterebene arbeiten und somit keine oder nur geringe Quelltextmodifikationen durchführen müssen. Diese Flexibilität geht zumeist auf Kosten der Ausführungsgeschwindigkeit.

2.1.4 Anwendungsgebiete

Die meisten Bereiche in denen aspektorientierte Programmierung bisher angewendet wird decken technische Funktionalität ab. Typische Anwendungsfelder sind z.B.:

- Fehlerbehandlung
- Sicherheitsüberprüfung
- Verfolgung
- Transaktionen

- entfernte Aufrufe
- Synchronisation

In der Geschäftslogik finden sie weniger Verwendung. Das mag damit zusammenhängen, dass die verbreiteten aspektorientierten Spracherweiterungen, allen voran AspectJ, einen asymmetrischen Ansatz verfolgen und in ihrer Mächtigkeit teilweise beschränkt sind (s. Abschnitt 3.1).
Nichtsdestotrotz sind auch durch die Nutzung eher technischer Gesichtspunkte weitergehende Anwendungen möglich. Zum Beispiel lässt sich aspektorientierte Programmierung in der „Software-Archäologie“ bei Altanwendungen und/oder mangelhafter oder gar fehlender Dokumentation zur Ermittlung von Programmabläufen einsetzen (Böh) oder zur Analyse und Verbesserung der Gebrauchstauglichkeit von Programmen verwenden (Bru05). Eine ausführliche Betrachtung inklusive Problemanalyse, Beispielen, Lösungen und Vergleichen für Interaktionen und Rollen im Bereich der Multi-Agenten-Systeme findet sich z.B. in (GKCL) und (GCKL).

2.2 Smalltalk

Dieser Abschnitt ist keine vollständige Einführung in Smalltalk. Solch eine findet sich z.B. in (GR83). Statt dessen werden im folgenden die Eigenschaften Smalltalks kurz beschrieben, die für die aspektorientierte Programmierung in Smalltalk ausschlaggebend sind.

2.2.1 Grundlagen

Smalltalk ist eine dynamisch typisierte, objektorientierte Programmiersprache. Smalltalkprogramme werden von einem Kompiler in Bytecode übersetzt, der dann von einer virtuellen Maschine ausgeführt wird. Wie aspektorientierte Programmierung und AspectJ ist auch Smalltalk ein Produkt des Xerox PARC.
Die Objektorientierung in Smalltalk geht dabei so weit, dass alles ein Objekt und somit Bürger erster Klasse ist, von den elementaren Datentypen wie Zahlen und Zeichenketten bis zu Klassen, Methoden und dem Kompilierer. Daraus folgt, dass die Ausführung eines Smalltalkprogramms letztlich aus drei elementaren Aktionen besteht, nämlich dem Variablenzugriff, dem Methodenaufruf und der Methodenausführung. Weiter führt die konsequente Objektorientierung zu einem hohen Maß an Reflexion. Diese beiden Eigenschaften erleichtern eine Aspektorientierung für Smalltalk und bieten großes Potential (s. Kapitel 3 und 5).
Allerdings gibt es auch eine Kehrseite. Aufgrund der dynamischen Typisierung ist der Typ einer Variable und des an ihr gebundenen Objekts zur Kompilierzeit nicht ermittelbar. Dies führt zu einem Mehraufwand zur Laufzeit, hauptsächlich Überprüfungen, und verringert Optimierungsmöglichkeiten erheblich, wodurch die Effizienz leidet. Jedoch bietet Smalltalk dafür ein großes Potential, dass das Kosten-Nutzen-Verhältnis angemessen macht.

2.2.2 Strukturierungselemente

Das Hauptstrukturelement Smalltalks ist, wie in den meisten objektorientierten Programmiersprachen, die Klasse. Jedes Objekt hat eine Klasse, die seine Struktur und sein Verhalten durch lokale Definition oder Erben von der einzigen Oberklasse bestimmt.
Das Verhalten wiederum wird durch Methoden, dem zweitwichtigsten Strukturierungselement, definiert. Sie können über Argumente und lokale Variablen verfügen.
Sowohl Methoden als auch Klassen lassen sich in Kategorien gruppieren. Diese Kategorien haben für die Programmausführung keine weitere Bedeutung, sondern sind lediglich dokumentierender Natur.
Das letzte Strukturierungselement Smalltalks ist der Block. Er ist einer Methode sehr ähnlich, da er es ermöglicht Quelltext verzögert auszuführen, über Argumente und lokale Variablen verfügen kann und mit seinem Definitionskontext verbunden ist.

2.2.3 Variablen

Neben den Argumenten und lokalen Variablen von Methoden und Blöcken, gibt es in Smalltalk drei andere Variablenarten: Instanz-, Klassen- und Pool-Variablen.
Instanzvariablen sind exklusive Attribute ihrer Objekte und nur in deren Methoden sichtbar. Da auch Klassen Objekte sind, verfügen sie ebenso über Instanzvariablen, die manchmal als Klasseninstanzvariablen bezeichnet werden, aber keine eigenständige Gruppe bilden. Demnach sind sie auch nur in Klassenmethoden sichtbar.
Klassenvariablen werden in einer Klasse definiert und stehen allen Instanz- und Klassenmethoden der definierenden Klasse und all ihren Unterklassen zur Verfügung, wobei es sich überall um dieselbe Variable handelt.
Auch bei Pool-Variablen handelt es sich um eine einzige Variable. Um sie zu verwenden, müssen sie explizit von einer Klasse importiert werden. Dann sind sie in all ihren Methoden sichtbar.
Da Klassen- und Pool-Variablen von bestimmten Objekten gemeinsam verwendet werden, sind sie in VisualWorks zum Konzept der geteilten Variablen verallgemeinert worden. Wenn daher im Verlauf dieser Arbeit von geteilten Variablen gesprochen wird, umfasst dies Klassen- und Pool-Variablen, es sei denn, die genaue Variablenart wird explizit angegeben.

2.2.4 Dialekte

Es gibt eine Vielzahl an Implementationen von Smalltalk. Sie alle basieren auf dem ursprünglichen Smalltalk-80 (GR83) und dem ANSI-Standard, unterscheiden sich aber durch geringe Änderungen und in der mitgelieferten Entwicklungsumgebung. Zu den bekanntesten Smalltalksystemen zählen Squeak und VisualWorks. VisualWorks ist im Rahmen dieser Arbeit verwendet worden. Allerdings ist für die Funktion von AspectTalk nicht eine bestimmte Eigenschaft eines Dialekts zwingend notwendig. AspectTalk baut auf dem Smalltalkstandard

auf und ist somit in jedem Dialekt verwendbar. Bietet ein Dialekt allerdings ein besonderes Merkmal, wie Namensräume in VisualWorks, so wird dieses dennoch verwendet, sofern es nützlich erscheint.

2.3 Aspektorientierte Programmierung für Smalltalk

Während der Recherchen im Rahmen dieser Arbeit sind drei bereits bestehende und sehr unterschiedliche aspektorientierte Erweiterungen für Smalltalk näher betrachtet worden, die nachfolgend kurz vorgestellt werden um eine Vergleichsmöglichkeit mit AspectTalk zu schaffen.

2.3.1 AOP/ST

AOP/ST (Böl98) ist das Ergebnis einer Fallstudie zur Anwendung aspektorientierter Programmierung in einem Systemverwaltungswerkzeug. Auf Grundlage eines Aspektwebers sind dafür zwei domänenspezifische Aspekte und Sprachen für die Verfolgung und Prozesssynchronisation entwickelt worden. Demnach gibt es keine allgemeine, vorgefertigte Lösung für Aspekte, Anreicherungen, Verbindungspunkte und Verbindungspunktauswahlen.
Der Aspektweber erlaubt lediglich die Anreicherung von Methodenausführungen. Dazu wird die ursprüngliche Klasse durch eine andere ersetzt um den Methodenaufruf abzufangen (s. Unterabschnitt 4.3.3).

2.3.2 Apostle

Apostle (Alw02) ist eine Erweiterung von VisualAge for Smalltalk um aspektorientierte Programmierung.
Apostle ist sehr nah an eine frühe Version von AspectJ angelehnt. Aspekte werden als spezielle Klasse umgesetzt, wohingegen Anreicherungen und Verbindungspunktauswahlen nach dem Vorbild von AspectJ als eigene Konstrukte repräsentiert werden. Die beiden einzigen Verbindungspunkte sind die Methodenausführung und der Nachrichtenempfang, wobei dieser eine Methodenausführung ist, die nicht Ergebnis eines *super*-Aufrufs ist. Anreicherungen können vor, nach und um diese Methodenpunkte herum gebunden werden. Weder die Aspektinstanziierung noch die Anreicherungsreihenfolge kann gesteuert werden.
Anreicherungen werden mit dem Basisprogramm verwoben, indem der Quelltext der betroffenen Methoden verändert wird. Sie sind somit unmittelbar einsehbar.
Nach Angaben der zugehörigen Internetseite[1] ist das System verloren gegangen und nicht weiterentwickelt worden.

[1] http://www.cs.ubc.ca/labs/spl/projects/apostle

2.3.3 AspectS

AspectS (Hir03) erweitert Squeak um aspektorientierte Programmierung.
Wie in AspectJ werden Aspekte durch eine spezielle Klasse vertreten und Anreicherungen als Blöcke und somit anonyme Konstrukte umgesetzt. Anreicherungen können vor, nach und um eine Methodenausführung, dem einzigen Verbindungspunkt, herum ausgeführt werden. Es gibt keine Verbindungspunktauswahlen und die anzureichernden Methoden müssen manuell durch eigenen Quelltext ausgewählt werden. Weder die Aspektinstanziierung noch die Anreicherungsreihenfolge kann gesteuert werden. Allerdings ist es möglich, Anreicherungen nicht nur klassen- sondern auch instanzspezifisch anzuwenden.
Anreicherungen werden mit dem Basisprogramm verwoben, indem die betroffenen Methoden durch Methodenwrapper (s. Unterabschnitt 4.3.3) (BFJR98) ersetzt werden.

3 Aspektorientierte Programmierung mit AspectTalk

In diesem Kapitel werden die Prinzipien und Funktionalität von AspectTalk beschrieben. Dies geschieht formal und anhand von Beispielen. Der Fokus liegt dabei auf den Umgang mit AspectTalk. Technische Ausführungen werden im nächsten Kapitel behandelt.

3.1 Aspekte

Im klassichen Sinne ist ein Aspekt ein Modul, das ein Querschnittsanliegen implementiert. Viele aspektorientierte Sprachen bzw. Spracherweiterungen unterschieden dabei strikt zwischen Aspekten und Klassen. In Bezug auf AspectJ und ähnliche Sprachen wird in (RS05) darauf hingewiesen, dass diese Trennung und der damit verbundene Verzicht auf Allgemeinheit und Orthogonalität die konzeptuelle Integrität aspektorientierter Programmierung gefährdet und die entstehende Asymmetrie zwischen Klassen und Aspekten die Systemzusammensetzung kompliziert. Diese Asymmetrie äußert sich in den unterschiedlichen Möglichkeiten von Aspekten und Klassen:

- Ein Aspekt kann von einer Klasse erben, aber nicht umgekehrt.
- Ein Aspekt kann Methoden beliebig anreichern, aber nicht Anreicherungen von Aspekten.
- Aspektinstanziierung ist nicht wie Objekterzeugung beliebig steuerbar.

AspectTalk verfolgt einen symmetrischen Ansatz, da Aspekte keine gesonderten Konstrukte sind. Jede beliebige Klasse kann ein Aspekt sein. Eine besondere Schnittstelle oder das Erben von einer Aspektoberklasse ist unnötig. Somit kann eine Klasse gleichzeitig sowohl als Aspekt als auch im „gewöhnlichen“ Sinne verwendet werden. Dies ermöglicht es Querschnittsanliegen höherer Ordnung, also solchen die auf Aspektebene wirken, zu realisieren. Außerdem hat es den Vorteil, dass Aspekte keiner besonderen Behandlung durch die Entwicklungsumgebung bedürfen. Werkzeuge für Systemanalysen, Quelltextverwaltung oder Versionsverwaltung müssen nicht angepasst werden, sondern können mit Aspekten wie mit normalen Klassen arbeiten.

Klassen die als Aspekte fungieren werden mit einer Instanziierungsstrategie verbunden. Sie bestimmt welches Objekt in einem gegebenen Verbindungspunktkontext verwendet wird. Instanziierungsstrategien werden nicht vererbt. Vorgegebene Strategien sind:

- Singleton (eine einzige Instanz)
- Klassensingleton (die Klasse selbst)

- pro Verbindungspunkt (eine Instanz für jede Kombination aus Kontextobjektklasse und Kontextmethode)
- pro Thread (eine Instanz für den aktiven Thread)
- pro Kontextobjekt (eine Instanz für das den Verbindungspunkt enthaltene Objekt)
- pro Kontextobjektklasse (eine Instanz für die Klasse des den Verbindungspunkt enthaltenen Objekts)

Unter dem Gebot der Erweiterbarkeit lassen sich darüber hinaus eigene, beliebig komplexe Strategien definieren. Dem Benutzer obliegt somit bei Bedarf die volle Steuerung über die Aspekterzeugung. Die Minimalanforderung ist dabei die Instanzverwaltung (Abb. 15).

3.2 Anreicherungen

AspectTalk beschränkt sich auf Verhaltensanreicherungen, da strukturelle Modifikationen in den meisten Smalltalksystemen durch sog. Klassenerweiterungen (engl. *class extensions*) bereits möglich sind. Daher sind im folgenden mit Anreicherungen der Einfachheit halber Verhaltensanreicherungen gemeint.
Darüber hinaus sind Verhaltensanreicherungen interessanter und häufiger, da Verhaltensbeziehungen in gängigen (objektorientierten) Programmiersprachen äußerst schlecht oder gar nicht modular ausdrückbar sind (SGS$^+$05).

Wie Aspekte sind auch Anreicherungen in AspectTalk keine selbstständigen Konstrukte, sondern gewöhnliche Methoden. Damit sind sie keine anononymen Codeblöcke, wodurch, um der Forderung nach Dynamik zu genügen, zur Laufzeit wechselbare Bindungen zwischen Anreicherungen und Verbindungspunkten, abstrakte/geerbte/überladene Anreicherungen sowie Anreicherungen von Anreicherungen unterstützt werden. Anreicherungen können wie normale Methoden Ausnahmen werfen, die aber nicht automatisch abgefangen werden. Eine nicht gefangene Ausnahme verhindert die Ausführung jeder nachfolgender Anreicherung und u.U. der ursprünglichen Verbindungspunktaktion selbst.
Die einzige Anforderung an Anreicherungen ist, dass sie genau ein Argument erwarten müssen. Wird eine Anreicherung ausgeführt, wird ihr ein Objekt übergehen, das den Verbindungspunkt an dem sie momentan gebunden ist beschreibt. Trotz der in Abschnitt 3.4 beschriebenen verschiedenen Verbindungspunktarten verfügen alle über die gleiche Informationsgrundlage, sodass es unter dem Gebot der Erweiterbarkeit möglich ist eigene Verbindungspunktarten zu erstellen (Abb. 18).
Je nach Verbindungspunktart haben diese Angaben unterschiedliche Ausprägungen:

- **Kontextobjekt:** Objekt in dem der Verbindungspunkt liegt

- **Kontextmethode:** Name der Methode in der der Verbindungspunkt liegt
- **Ergebnis:** Ausgang des Verbindungspunkts

Damit eine Anreicherung effektiv wird, muss sie über eine Verbindungspunktauswahl an einen Verbindungspunkt gebunden werden. Dafür gibt es drei Möglichkeiten:

vor: Die Anreicherung wird ausgeführt unmittelbar bevor der Verbindungspunkt eintritt.

nach: Die Anreicherung wird ausgeführt unmittelbar nachdem der Verbindungspunkt passiert worden ist.

herum: Die Anreicherung wird um den Verbindungspunkt herum ausgeführt. In dieser Situation kommt der Anreicherung eine Sonderstellung zu, da sie das Eintreten des Verbindungspunkts steuern kann. Dass heißt sie kann ihn beliebig oft oder auch gar nicht eintreten lassen. Dies geschieht durch den Aufruf der Methode *JoinPoint»#proceed* auf dem Verbindungspunktobjekt, die die nächste Bindung bzw. schließlich die eigentliche Verbindungspunktaktion ausführt.

Standardmäßig hat eine Bindung keinen Vorrang vor einer anderen. Dies lässt sich durch Angabe einer Ordnungszahl ändern. Je höher diese Zahl ist, umso später, bei vor- und herum-Bindungen also zeitlich näher am eigentlichen Verbindungspunkt, wird sie ausgeführt. Vorränge auf Bindungs- und damit Verbindungspunktebene und nicht wie in AspectJ auf Aspektebene zu bestimmen hat den Vorteil einer feineren und genaueren Steuerung.

3.3 Elementfilter

Elementfilter haben in AspectTalk die Aufgabe die an einem Verbindungspunkt beteiligten Programmelemente zu beschreiben. Dafür gibt es zwei Extreme. Zum einen die komplett händische Variante wie z.B. in AspectS:

```
Morph withAllSubclasses
    select: [:each | each includesSelector: #mouseEnter:]
    thenCollect: [:each | AsJoinPointDescriptor
                              targetClass: each
                              targetSelector: #mouseEnter:]
```

In diesem Beispiel soll eine Methode *#mouseEnter:* in der Klasse *Morph* und ihren Unterklassen angereichert werden. Nun ist bei einfachen Beispielen wie diesem der Aufwand Elemente direkt in Smalltalk auszudrücken vergleichsweise gering. Sobald aber komplexe Bedingungen und Abhängigkeiten zu beschreiben sind, wird der Aufwand größer.
Das andere Extrem ist die Verwendung einer eigenen „Sprache“ unter Beschränkung auf einfache Zeichenketten wie in AspectJ und Apostle:

around<kindOf(Morph) & executions(#mouseEnter:)>

Der Preis für diese Kürze und Einfachheit ist eine eingeschränkte Ausdrucksfähigkeit.
In AspectTalk werden die Vorteile beider Wege kombiniert. Um einen Filter zu erhalten wird an eine Zeichenkette, die wie in AspectJ ein Namensmuster angibt, eine Nachricht geschickt, die den Filtertyp bestimmt und die wiederkehrende Arbeit der händischen Variante kapselt. Das obige Beispiel sieht in AspectTalk wie folgt aus:

'Morph' sub » 'mouseEnter:' method

Die gewählte Form ist komfortabler und setzt keine speziellen Systemkenntnisse voraus. Sie beschränkt sich auf das notwendigste, betont den beschreibenden Aspekt und ermöglicht es (komplexe) Zusammenhänge klar, knapp und verständlich auszudrücken. Nichtsdestotrotz lassen sich auch eigene Formen und spezielle Typen der Elementauswahl ausdrücken, da auch sie letztlich Bürger erster Klasse sind. Darauf wird in Abschnitt 4.7 näher eingegangen.

Nachfolgend werden der Gebrauch und die Wirkung der vorgegebenen Elementfilter vorgestellt. Die Beispiele der folgenden Abschnitte basieren auf dem in Abb. 1 dargestellten System.

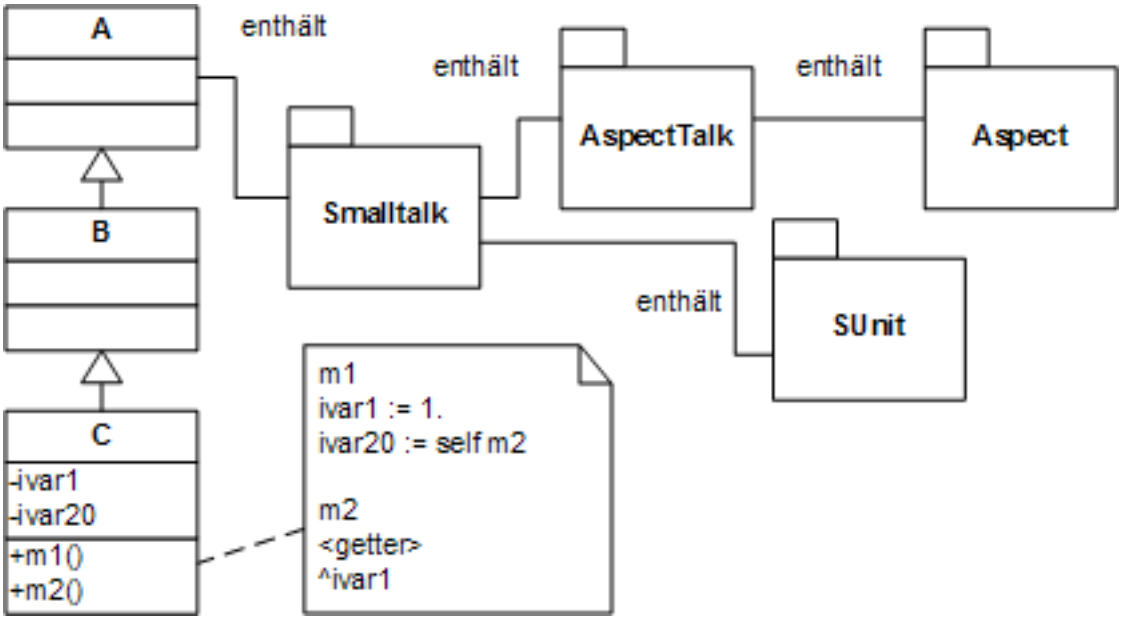

Abbildung 1: Beispielsystem

Zuvor ist aber noch eine Regel wichtig, da sie in allen Elementfiltern eingesetzt wird: *Aus-*

druck. Sie gestattet es auf in Smalltalk bekannte Weise Namensmuster auszudrücken: Die Angabe einer beliebigen Zeichenkette, wobei ein # für genau ein beliebiges Zeichen, während ein * für eine beliebige, also auch leere, Zeichenkette steht.

3.3.1 Filteroperatoren

Da Elementfilter eine Menge von Elementen spezifizieren, lassen sich auf ihnen die drei Standardmengenoperationen anwenden: Vereinigung, Durchschnitt und Differenz. Mit ihnen lassen sich komplexe Zusammenhänge beschreiben, die nicht durch Ausdrücke zu formulieren sind, und unterschiedliche Elemente verknüpfen. Sie stehen bei allen Elementfiltern über eine Operatorangabe zur Verfügung:

Operator → ' & ' | ' | ' | ' \ '

Eine Vereinigung zweier Elementfilter A und B erzeugt die Gesamtmenge aller Elemente die von A oder B abgedeckt werden.
Ein Durchschnitt zweier Elementfilter A und B erzeugt die Menge nur der Elemente die sowohl von A als auch von B angesprochen werden.
Eine Differenz zweier Elementfilter A und B erzeugt die Menge der Elemente die A spezifiziert, mit Ausnahme derer die auch von B abgedeckt werden.

3.3.2 Namensraum

Ein Namensraum ist ein integriertes Sprachmerkmal, dass von einigen Smalltalkdialekten angeboten wird. Klassen und geteilte Variablen die in einem Namensraum erzeugt werden, sind nur innerhalb des Namensraums sichtbar. Soll von außerhalb des Namensraumes auf eine seiner Klassen oder geteilten Variablen zugegriffen werden, muss er importiert oder die Referenz qualifiziert werden. Namensräume können beliebig geschachtelt werden. Damit ähneln sie den Paketen in Java. Dieses Sprachmerkmal ist aus dem Bedürfnis entstanden Namenskonflikte in großen Systemen zu vermeiden, da Klassen und geteilte Variablen in Smalltalk standardmäßig global sichtbar sind (Cin). Außerdem stellt ein Namensraum ein probates Mittel der Suchraumeinschränkung dar, da sonst die Klasse *Object* und all ihre Unterklassen untersucht werden müssten! In Smalltalkdialekten die über keine Namensräume verfügen, dient dieser Filter als Beschreibung von Pool-Variablen-Verzeichnissen.
Ein Namensraum lässt sich wie folgt beschreiben:

Namensraum → *Ausdruck* (' nspace' | ' subspace' | ' superspace') {' » ' *Namensraum*}
Namensraum → '(' *Namensraum Operator Namensraum* ')'

Mittels ' nspace' werden alle Namensräume ausgewählt auf die das Namensmuster zutrifft, während ' subspace' und ' superspace' auch all deren Kinder- bzw. Elternnamensräume

auswählen, wobei diese nicht das angegebene Namensmuster aufweisen müssen.
Damit lassen sich z.B. die folgenden Filter anlegen:

Auswahl	**Elemente**
'A*' nspace	AspectTalk
'*' nspace	AspectTalk, SUnit
'*' subspace	AspectTalk, AspectTalk.Aspect, SUnit
'*' subspace & '*p*' nspace	AspectTalk, AspectTalk.Aspect

Tabelle 1: Beispiele für Namensraumfilter

3.3.3 Klasse

Es gibt drei unterschiedliche Klassenfilter. Alle können optional auf einem Namensraum- und Klassenkategoriefilter arbeiten, aus der die Klassen auszuwählen sind. Wird auf die Namensraumangabe verzichtet bzw. verfügt ein Smalltalkdialekt über keine Namensräume, findet die Filterung im globalen Namensraum statt. Mittels eines Klassenkategoriefilters lässt sich der Suchraum innerhalb eines Namensraumes weiter einschränken:

Klassenkategorie → [*Namensraum* ' » '] *Ausdruck* ' ccat'
Klassenkategorie → '(' *Klassenkategorie Operator Klassenkategorie* ')'
Klasse → [*Klassenkategorie* ' » '] *Ausdruck* (' class2' | ' sub' | ' super')
Klasse → '(' *Klasse Operator Klasse* ')'

Durch ' class' werden alle Klassen ausgewählt auf die das Namensmuster zutrifft, während ' sub' und ' super' auch all deren Unter- bzw. Oberklassen auswählen, wobei diese nicht das angegebene Namensmuster haben müssen.
Damit lassen sich z.B. die folgenden Filter anlegen:

Auswahl	**Elemente**
'A' sub \ 'A' class2	B, C
'C' super	A, B, C
'A' sub & 'C' super	B

Tabelle 2: Beispiele für Klassenfilter

3.3.4 Methode

Es gibt direkte und indirekte Methodenfilter. Ein direkter Filter wählt Methoden aufgrund ihrer Signatur aus, während indirekte Methodenfilter auf bestimmte Eigenschaften ausgerichtet sind. Dementsprechend arbeiten Direktfilter auf einem Klassenfilter, während indirekte Filter einen anderen Filter erwarten:

Methodenkategorie → *Ausdruck* ' mcat'
Methodenkategorie → '(' *Methodenkategorie Operator Methodenkategorie* ')'
Methode → *Klasse* ' » ' [*Methodenkategorie* ' » '] *Ausdruck* ' method'
Methode → *Klasse* ' » ' *Ausdruck* ' pragma'
Methode → *Variable* (' readers' | ' writers')
Methode → *Methode* ' derived'
Methode → '(' *Klasse Operator Klasse* ')'

Mittels ' method' lassen sich Methoden, die der angegebenen Klasse gehören, nach ihrer Kategorie oder ihren Namen auswählen. Auch wenn die Implementation einer Methode vererbt werden kann, wird nur die Methode in der gegebenen Klasse(n) ausgewählt. Dies gilt für alle direkten Methodenfilter.
Über ' pragma' werden Methoden angesprochen, die der angegebenen Klasse gehören, und über eine Annotation, einem sog. Pragma, verfügen. Dies gestattet es Methoden zu erfassen, die kein gemeinsames Namensmuster aufweisen oder eine bestimmte Semantik besitzen. So werden in VisualWorks Methoden die eine Oberflächenspezifikation erzeugen mit dem Pragma *<resource:>* annotiert. Damit sind diese unter Verwendung von '<ressource:>' pragma einfacher zu ermitteln als durch separates Aufzählen jeder einzelnen Methode. Dafür müsste man allerdings den Namen jeder einzelnen Methode kennen. Durch das Pragma ist man davon unabhängig. Eine anschauliche Diskussion über die Benutzung von Annotationen und expliziten Signaturen sowie den jeweiligen Vorteilen findet sich in (KM). Dies geschieht anhand eines konkreten Beispiels in AspectJ, wobei die gefundenen Ergebnisse von allgemeiner Bedeutung sind.
Mit Hilfe von ' readers' und ' writers' werden die Methoden ausgewählt, die auf eine Variable der angegebenen Variablenauswahl lesend oder schreibend zugreifen. Daraus ergibt sich zur Vermeidung ausdrücklicher Aufzählung oder in Unkenntnis genauer Methoden ein ähnlicher Nutzen wie bei 'pragma'.
Durch ' derived' werden die Methoden ausgewählt, die letztlich aus der Ausführung einer Methode des angegebenen Methodenfilters hervorgehen. Dieser Filter entspricht damit der Angabe cflowbelow() aus AspectJ.
Damit lassen sich z.B. die folgenden Filter anlegen:

Auswahl	Elemente
'C' class2 » 'm*' method	C»#m1, C»#m2
'C' class2 » 'm1' method derived	C»#m2
'C' class2 » '<getter>' pragma	C»#m2
'C' class2 » 'ivar1' ivar readers	C»#m2

Tabelle 3: Beispiele für Methodenfilter

3.3.5 Variablen

Auch bei den Variablen gibt es direkte und indirekte Filter. Direktfilter wählen auf Basis von Namen und des Definitionsbereichs aus. Der Kontext ist Grundlage für die indirekten Filter:

Variable → '(' *Variable Operator Variable* ')'
Variable → (*Instanzvariable* | *GeteilteVariable* | (*Methode* (' read' | ' written')))
Instanzvariable → *Klasse* ' » ' *Ausdruck* ' ivar'
GeteilteVariable → (*Klasse* | *Namensraum*) ' » ' *Ausdruck* ' svar'

Durch ' ivar' werden die Instanzvariablen der angegebenen Klassen ausgewählt deren Namen dem gewählten Muster entsprechen.
Bei ' svar' wird entsprechend verfahren. Allerdings kann neben einer Klasse auch ein Namensraum spezifiziert werden. Im ersten Fall werden, sofern es sich um eine Metaklasse handelt, Klassenvariablen angesprochen, während sich im letzten Fall allgemeine geteilte Variablen ergeben.
Wie auch schon bei den Methodenfiltern können Variablen auch indirekt angegeben werden. Mittels ' read' und ' written' werden die Variablen ausgewählt, die von bestimmten Methoden gelesen bzw. geschrieben werden.
Damit lassen sich z.B. die folgenden Filter anlegen:

Auswahl	Elemente
'C' class2 » 'ivar#' ivar	C.ivar1
'C' class2 » 'm1' method written	C.ivar1, C.ivar20

Tabelle 4: Beispiele für Variablenfilter

3.4 Verbindungspunktauswahl

AspectTalk verfügt über drei unterschiedliche Verbindungspunktarten: Methodenaufruf, Methodenausführung und Zugriff auf eine Instanz- bzw. geteilte Variable. Dies sind gleichzeitig die einzigen Aktionen in einem Smalltalkprogramm. Fälle wie in AspectJ bzw. Java die

Konstruktoren, die einzelnen Initialisierungspunkte und die Ausnahmebehandlung müssen nicht gesondert erfasst werden, da sie in Smalltalk nicht durch eigenständige Konstrukte vertreten sind. Statt dessen werden sie durch gewöhnliche Methoden, die meistens bestimmten Konventionen unterliegen, umgesetzt. Dadurch sind sie in AspectTalk dennoch erfassbar.

Alle Auswahlen verfügen über einen Namen und einen Aspekt zu dem sie gehören, wobei diese Eigenschaften jederzeit geändert werden können. Dies ermöglicht es sie zu strukturieren und wiederzuverwenden.
Verbindungspunktauswahlen arbeiten mit Elementfiltern. Anhand der von ihnen gelieferten Programmelemente werden die tatsächlichen Verbindungspunkte ermittelt. Im Sinne der geforderten Dynamik ist es möglich, ihre Filter zur Laufzeit beliebig zu verändern.
Eine Verbindungspunktauswahl hat zwei unterschiedliche Arbeitsweisen. Eine erhaltende Auswahl wird stets und nur die Verbindungspunkte auswählen, die zum Zeitpunkt ihrer Erzeugung durch die Elementfilter spezifiziert worden sind. Solange die Verbindungspunkte existieren, werden Änderungen an ihnen, z.B. Umbenennungen, automatisch berücksichtigt.
Die andere Arbeitsweise ist die Erweiterung. Solche Auswahlen schließen zusätzlich Verbindungspunkte ein, die im Laufe der Zeit neu entstehen und den Elementfiltern entsprechen.
Die verschiedenen Verbindungspunktauswahlen werden durch Unterklassen der Klasse *Pointcut* umgesetzt und sind damit Bürger erster Klasse (Abb. 19). Dadurch wird der Benutzer befähigt, gemäß der geforderten Erweiterbarkeit, eigene Auswahlarten zu erstellen. Hierzu ist es lediglich notwendig ein Verzeichnis zu erstellen, das die ausgewählten Verbindungspunkte identifiziert und dem Aspektweber mitzuteilen in welchen Methoden diese Verbindungspunkte liegen. Weiterhin müssen die verwendeten Elementfilter geliefert werden, damit die Verbindungspunktauswahl bei Refaktorisierungen automatisch angepasst werden kann.
Nachfolgend werden die einzelnen Verbindungspunktauswahlen vorgestellt.

3.4.1 Methodenaufruf

Der Aufruf einer Methode ist die Anweisung eines Senders an einen gegebenen Empfänger eine bestimmte Nachricht zu senden. Dies wird durch die Klasse *Call* (Abb. 19) beschrieben, die zwei Methodenfilter, welche Sender und Empfänger spezifizieren, erwartet.
Ist eine Anreicherung an einen Methodenaufruf gebunden, verfügt das ihr übergebene Verbindungspunktobjekt (Abb. 18) über folgende Informationen:

- Sender (Kontextobjekt)
- aufrufende Methode (Kontextmethode)
- Empfänger
- aufzurufende Methode

- Argumentenliste der aufzurufenden Methode
- Rückgabewert der aufzurufenden Methode (Ergebnis)
- Pragmas der aufzurufenden Methode

Die einzelnen Argumente, der Rückgabewert und der Empfänger sind durch Anreicherungen veränderbar. Die Modifikation der Argumente und des Empfängers wird sich nur in vor- und herum-Bindungen auswirken, wohingegen ein veränderter Rückgabewert nur in nach- und herum-Bindungen Bestand hat.

3.4.2 Methodenausführung

Die Ausführung einer Methode findet statt, nachdem der Interpreter den Nachschlagemechanismus erfolgreich durchgeführt hat und den zugehörigen Bytecode verarbeitet. Dieses Merkmal wird durch die Klasse *Execution* (Abb. 19) vertreten, die einen Methodenfilter erwartet.
Ist eine Anreicherung an eine Methodenausführung gebunden, verfügt das ihr übergebene Verbindungspunktobjekt (Abb. 18) über folgende Informationen:

- Sender
- aufrufende Methode
- Empfänger (Kontextobjekt)
- auszuführende Methode (Kontextmethode)
- Argumente der auszuführende Methode
- Rückgabewert der auszuführende Methode (Ergebnis)
- Pragmas der auszuführenden Methode

Die einzelnen Argumente und der Rückgabewert sind durch Anreicherungen veränderbar. Die Modifikation der Argumente wird sich nur in vor- und herum-Bindungen auswirken, wohingegen ein veränderter Rückgabewert nur in nach- und herum-Bindungen Bestand hat.

3.4.3 Variablenzugriff

Ein Variablenzugriff ist das Auslesen oder Schreiben einer Variable. Dieses Merkmal wird durch die Klassen *Read* und *Write* (Abb. 19) repräsentiert, die einen Variablen- und Methodenfilter erwarten.
Ist eine Anreicherung an einen Variablenzugriff gebunden, verfügt das ihr übergebene Verbindungspunktobjekt (Abb. 18) über folgende Informationen:

- zugreifendes Objekt (Kontextobjekt)
- zugreifende Methode (Kontextmethode)
- Variablenname
- Variablenwert (nur bei Lesezugriff Ergebnis)
- neuer Variablenwert (nur bei Schreibzugriff, Ergebnis)

Der Variablenwert beim Lesezugriff und der neue Variablenwert beim Schreibzugriff sind durch Anreicherungen veränderbar. Die Modifikation des alten Variablenwertes wirkt sich nur in vor- und herum-Bindungen aus, während ein veränderter neuer Wert in herum- und nach-Bindungen effektiv ist. Eine Veränderung des alten Variablenwertes ist während des Zugriffspunktes sichtbar und wird nicht auf die eigentliche Variable übertragen, während der neue Variablenwert auch nach Durchlaufen des Zugriffspunktes sichtbar ist und der eigentlichen Variable zugewiesen wird.

3.5 Konfliktanalyse

Ein Kritikpunkt aspektorientierte Programmierung ist, dass sie zu einem unerwarteten oder gar unvorhersehbaren Programmverhalten führen kann. Dies ist möglich, da sie den Programmablauf, z.B. durch Ausnahmen oder Wertemanipulation, stark beeinflussen kann. In diesem Fall lässt sich die geforderte Transparenz (s. Unterabschnitt 2.1.1) nicht aufrechterhalten bzw. rechtfertigen, da sie bedeutet, dass sich der Programmierer der Anwesenheit von Anreicherungen nicht bewusst sein muss. Dadurch ist es ihm dann aber unmöglich modulare Schlussfolgerungen zu treffen. Nach (CL) sind modulare Schlussfolgerungen dann durchführbar, wenn die Aktionen eines Moduls M allein aufgrund des Quelltexts dieses Moduls und dem Verhalten anderer Module auf die sich M bezieht verstanden werden können. Es geht also um ein korrektes Programmverständnis.
In diesen Zusammenhang fallen auch Anreicherungskonflikte. Sie entstehen, sobald die an einen Verbindungspunkt gebundenen Anreicherungen eine bestimmte Reihenfolge erfordern um ein ordnungsgemäßes Verhalten zu erzeugen. Beispielsweise muss ein Datum erst verschlüsselt, werden bevor es persistent gemacht werden kann.

Zur Lösung dieser Probleme gibt es unterschiedliche Ansätze. In (CL) wird vorgeschlagen Aspekte in Zuschauer und Assistenten zu unterteilen und diesen unterschiedliche Mächtigkeiten zu erlauben. Ein Zuschauer darf den Programmfluss an einem Verbindungspunkt lediglich beobachten, aber nicht verändern. Assistenten hingegen unterliegen keinen Beschränkungen. Allerdings müssen sie explizit akzeptiert werden.
Ein anderes Konzept sind sog. offene Module (Ald). Hierbei werden Aspekte bzw. Anreicherungen nicht in ihren Fähigkeiten beschränkt. Jedoch dürfen nur solche Verbindungspunkte

angereichert werden, die ein offenes Modul durch eine Verbindungspunktauswahl explizit verfügbar macht.
Solche Beschränkungen durch ausdrückliche Berechtigungen aufzuerlegen erscheint sinnvoll in „geschlossenen“ Umgebungen in denen Programme nicht in vollem Umfang zur Verfügung stehen bzw. manipulierbar sind. Zum Beispiel sind in Jave fremde Programme in Form von Bibliotheken nicht beliebig einsehbar bzw. änderbar und AspectJ ermöglicht lediglich die Anreicherung von mit AspectJ übersetzten Programmen. In einem offenen System wie Smalltalk in dem Programmquellen komplett einzusehen und modifizierbar sind leidet die Effektivität von expliziten Restriktionen, da die entsprechenden Mechanismen ausschaltbar sind.
AspectTalk beschränkt sich daher auf eine Anreicherungsanalyse anhand derer der Programmierer sehen kann, ob er sich der Anwesenheit der Anreicherungen bewusst sein muss. Somit kann er entscheiden, ob es einer speziellen Anreicherungsreihenfolge bedarf und ob das Programmverhalten u.U. von seiner Erwartung abweichen wird. Notfalls kann er sogar Anreicherungen entfernen.
Dazu wird ähnlich wie in (CL) untersucht, wie eine Anreicherung mit dem Verbindungspunkt interagiert. Dies geschieht unter Betrachtung folgender Fragen:

- Zugriffspunkt:
  - Liest die Anreicherung den neuen/alten Variablenwert aus?
  - Ändert die Anreicherung den neuen/alten Variablenwert?
- Aufrufs-/Ausführungspunkt:
  - Liest die Anreicherung die Argumente aus?
  - Ändert die Anreicherung die Argumente?
  - Liest die Anreicherung den Empfänger aus?
  - Ändert die Anreicherung den Empfänger?
- herum-Bindung:
  - Wird die ursprüngliche Aktion überhaupt ausgeführt?
  - Wird die ursprüngliche Aktion immer genau einmal, mehrmals oder nur bedingt ausgefüht?

Abgesehen davon, dass durch Reduktion auf das Halteproblem unentscheidbar ist, ob der Programmfluss unter allen Umständen nicht gestört wird, sollte eine umfassende Betrachtung auch die Möglichkeit von Ausnahmen und die Änderungen von anderen, äußeren Zuständen von der die angereicherte Methode abhängig ist umfassen. Aufgrund von Umfangs-

und Zeitbeschränkungen wird dies hier nicht weiter betrachtet. Außerdem stellt sich die Frage, inwieweit dies wegen der dynamischen Typisierung in Smalltalk durch Mittel der Typinferenz feststellbar ist.
Nichtsdestoweniger lassen sich mit den ermittelten Informationen Anreicherungen solide einschätzen. Daraus ergibt sich ähnlich wie in (CL) eine zweiteilige Klassifikation. Demnach ist eine Anreicherung bewahrend, sofern sie nur lesend auf den Verbindungspunkt zugreift und in einer herum-Bindung zusätzlich immer genau einmal *JoinPoint»#proceed* aufruft. Trifft dies zu, bleibt das ursprüngliche Programmverhalten bewahrt. Jede andere Konstellation führt zu einer eingreifenden Anreicherung. Diese, wenn auch einfache, Unterscheidung ermöglicht es dem Programmierer weitere Überlegungen durchzuführen um Anreicherungskonflikte aufzulösen, die letztlich auf klassische Lese-Schreib-Konflikte zurückführbar sind.

3.6 Umgebungseinbettung

Bereits seit Anbeginn von Smalltalk erfolgt die Programmierung und Systeminteraktion durch graphische Benutzerschnittstellen (Gol84). Werkzeuge wie der Workspace, der Systembrowser oder der UI-Painter kapseln die notwendigen programmatischen Vorgänge durch komfortablere graphische Oberflächen. Dieser Philosophie folgt auch AspectTalk. Daher werden im folgenden die Oberflächen vorgestellt, die die Arbeit mit AspectTalk erleichtern. Da AspectTalk in VisualWorks implementiert worden ist, fällt das Erscheinungsbild entsprechend aus. Demnach hätte die Einbettung in die Entwicklungsumgebung andere Dialekte ein jeweils typisches Aussehen. Die gebotene Funktionalität bleibt jedoch stets erhalten.

Verbindungspunktauswahl

Verbindungspunktauswahlen werden Aspekten, also Klassen, zugewiesen. Es scheint daher sinnvoll sie wie Methoden oder Instanzvariablen als weitere Klasseneigenschaft zu betrachten. In VisualWorks geschieht dies durch Hinzufügen eines weiteren Registerreiters, der eine gewohnte Übersicht aus vorhandenen Auswahltypen und bestehenden Auswahlen bietet (Abb. 13). Damit verbunden ist eine Definitionsoberfläche, die es gestattet Verbindungspunktauswahlen anzulegen, zu ändern oder zu löschen. Außerdem enthält sie eine Übersicht der erfassten Elemente sowie der vorhandenen Bindungen.
Diese Informationen dienen dem besseren Verständnis und der einfachen Handhabung. Weiterhin sollen sie zusammen mit einem gewohnten Erscheinungsbild zu einer höheren Akzeptanz beitragen. Dies erscheint wichtig, bedenkt man die geringe Verbreitung aspektorientierter Programmierung. Gemäß der Forderung nach Erweiterbarkeit ist die Kenntnis über und Darstellung der verschiedenen Auswahlarten nicht im Werkzeug hardkodiert. Statt dessen wird jeder Auswahlart ein Visualisierer zugeordnet, der für ihre Darstellung und Erzeugung verantwortlich ist (Abb. 16).

Verbindungspunkte

Das Verbindungspunktwerkzeug wird aktiv, sobald eine Methode ausgewählt wird und diese über angereicherte Verbindungspunkte verfügt. Es beinhaltet einen Überblick über die angereicherten Verbindungspunkte, die vorhandenen Bindungen sowie spezifische Informationen über die gewählte Anreicherung (Abb. 14). Dies gestattet eine intensivere Betrachtung bei der Konfliktermittlung. Für eine schnelle Übersicht wird die Auswertung der Anreicherungsanalyse optisch aufbereitet. Gemäß der Forderung nach Erweiterbarkeit ist die Kenntnis über und Darstellung der verschiedenen Verbindungspunktarten nicht im Werkzeug hardkodiert. Statt dessen wird jeder Verbindungspunktart ein Visualisierer zugeordnet, der für ihre Darstellung verantwortlich ist (Abb. 17).

Instanziierungsstrategie

Über das Kontextmenü einer Klasse kann ein Dialog aktiviert werden, über den die Instanziierungsstrategie des Aspekts angesehen und geändert werden kann (Abb. 12). Zudem werden alle derzeit bestehenden Aspektinstanzen gezeigt, die mittels des Objektinspektors einsehbar sind.

3.7 Anwendungsbeispiele

3.7.1 Beobachtermuster

Von den klassischen Entwurfsmustern befassen sich vor allem die Verhaltensmuster wie der Beobachter, der Besucher oder der Mediator mit Verhaltensbeziehungen. Wie in Abschnitt 3.2 erwähnt, sind besonders solche schwer modular ausdrückbar. Dementsprechend weisen die Implementationen dieser Entwurfsmuster verstreute Elemente auf. Beim Beobachtermuster handelt es sich dabei um die explizite Benachrichtigungen des Beobachteten (change) und die Reaktionen des Beobachters (update). Mittels aspektorientierter Programmierung lässt sich dies beseitigen, wie nachfolgend am Beispiel des Beobachtermuster gezeigt wird. Der zugehörige Quelltext ist in Anhang C zu finden.

Ein Aspekt *Interest* ist die abstrakte Oberklasse für alle Beziehungen zwischen Beobachter und Beobachteten. Jede Unterklasse definiert zunächst eine Verbindungspunktauswahl (*Interest»#pointcut*), die die Ereignisse beschreibt über die der Beobachter informiert werden soll. Weiterhin muss eine Methode implementiert werden, die das notwendige Aktualisierungsverhalten des Beobachters festlegt (*Interest»#notify:about:in:*).
Somit ist eine verbesserte Modularität erreicht, da weder im Beobachter noch im Beobachteten der Quelltext angepasst werden muss. Außerdem ergibt sich eine verbesserte Wiederverwendbarkeit, da der Benachrichtigungsmechanismus (*Interest»#notifyObservers:*)

und die Abbildung von Beobachteten auf Beobachter (*Interest class»#add:to:*, *Interest»#remove:from:*) universell gültig sind.

3.7.2 Synchronisierte Methoden

Es gibt Methoden, die von mehreren Threads nicht nebenläufig aufgerufen werden dürfen, und deren Ausführung somit serialisiert werden muss um unerwünschte Nebeneffekte zu vermeiden. In Java lässt sich dies einfach durch den Modifizierer *synchronized* erreichen, der eine Methode als kritischen Abschnitt erklärt. Dadurch wird automatisch verhindert, dass mehrere Threads andere, ebenfalls mit *synchronized* bezeichnete Methoden desselben Objekts ausführen. Der Thread der sich derzeit im kritischen Bereich befindet, darf jedoch beliebig synchronisierte Methoden des Objekts aufrufen. Smalltalk bietet diese Möglichkeit nicht. Sie muss jedes Mal manuell reproduziert werden, z.B. durch Benutzung von Semaphoren. Hierbei handelt es sich also eindeutig um ein Querschnittsanliegen, dessen Umsetzung mittels aspektorientierter Programmierung im folgenden beschrieben wird. Der zugehörige Quelltext ist in Anhang C zu finden.

Soll eine Methode synchronisiert werden, wird ihrer Definition das Pragma *<synchronized>* hinzugefügt. Dies bietet einen ähnlichen Komfort wie der Modifizierer in Java, da keine weiteren Vorkehrungen zu treffen sind. Weiterhin gibt es einen Aspekt *Synchronizer*, der eine Verbindungspunktauswahl definiert, die die Ausführung aller mit *<synchronized>* annotierten Methoden abdeckt (*Synchronizer class»#install*). Um die Ausführung herum wird die Anreicherung *Synchronizer»#synchronize:* gebunden, mittels der das o.g. Verhalten umgesetzt wird.

4 Realisierung von AspectTalk

Dieses Kapitel beschreibt die wichtigsten Details der Umsetzung von AspectTalk. Dies betrifft insbesondere die Ermittlung von Verbindungspunkten und die Einbettung von Anreicherungen.

4.1 Kompilierte Methoden

Methoden sind in Smalltalk als Objekte der Klasse *CompiledMethod* Bürger erster Klasse. Wichtige Vorgänge in der Implementation von AspectTalk, die in den folgenden Abschnitten erläutert werden, arbeiten mit solchen Objekten. Daher werden zum besseren Verständnis dieser Vorgänge zunächst die wichtigsten Eigenschaften kompilierter Methoden aufgezeigt. Die Beschreibung stützt sich auf (GR83).

Kompilierte Methoden verfügen über ihren Quelltext, den daraus kompilierten Bytecode, der Klasse in der sie kompiliert worden sind, einen Literal- und Temporärrahmen sowie einen Kopf.

Der Quelltext wird durch den Kompilierer in den entsprechenden Bytecode übersetzt. Für die korrekte Funktion einer kompilierten Methode ist der Quelltext nicht erforderlich und kann somit nach erfolgreicher Kompilation auch entfernt werden. Allerdings ist das Vorhandensein des Quelltext unabdingbar, da er die Grundlage für die Transformationen ist.

Das Bytecode-Array beinhaltet die Instruktionen für den Interpreter. Die Bytecodes unterteilen sich in fünf Wirkungsgruppen, wobei nur die ersten drei für die Arbeitsweise von AspectTalk relevant sind:

1. Ein Objekt wird auf den Stapel gelegt (push).
2. Das oberste Objekt des Stapels wird einer Variablen zugewiesen (store).
3. An ein Objekt wird eine Nachricht geschickt (send).
4. Der Rückgabewert einer Methode wird zurückgeliefert (return).
5. Es wird an eine bestimmte Stelle des Bytecode-Arrays gesprungen (jump).

Der Literalrahmen einer Methode enthält die Objekte, die nicht direkt durch Bytecodes repräsentiert werden können. Dazu gehören Klassen- und andere geteilte Variablen sowie die meisten Konstanten (Nummern, Zeichen, Zeichenketten, Symbole und Arrays). Diese Objekte werden angesprochen, indem spezielle Bytecodes auf ihre Position im Literalrahmen verweisen.

Der Temporärrahmen enthält die beim Aufruf der Methode übergebenen Argumente und die in der Methode definierten temporären Variablen. Der Methodenkopf speichert Informationen u.a. darüber, ob die Methode eine Primitive aufruft, wie viele Argumente und temporäre Variablen sie benötigt und wie groß ihr Literalrahmen ist. Diese beiden Strukturen werden in AspectTalk nicht gebraucht, seien aber der Vollständigkeit halber erwähnt.

4.2 Verbindungspunktermittlung

Nachfolgend werden die notwendigen Schritte konzeptionell dargelegt, die zur Auffindung der unterschiedlichen Verbindungspunkte führen. Sie alle arbeiten direkt mit kompilierten Methoden. Eine weitere Möglichkeit wäre die Analyse des Syntaxbaums der Methoden. Dieser müsste aber erst generiert werden, während das Bytecode-Array und der Literalrahmen bereits vorliegen. Dies gestattet eine schnelle Analyse, die besonders für die geforderte Dynamik wichtig ist.

4.2.1 Methodenaufruf

Wie in Unterabschnitt 3.4.1 beschrieben worden ist, umfasst ein Methodenaufruf die Spezifikation des Senders und Empfängers, die über entsprechende Methodenfilter gegeben ist. Auf dieser Grundlage wird wie folgt verfahren:

1. Erstelle eine Liste aller Methoden die der Senderfilter auswählt.
2. Iteriere diese Liste und überprüfe jede zugehörige kompilierte Methode auf Nachrichten.
   (a) Das Bytecode-Array enthält eine Versandinstruktion. Ist die Empfängerklasse eindeutig (*self*, *super*, ein Literal oder eine Klasse) betrachte nur diese, sonst jede mögliche Klasse.
   (b) Das Methodensymbol ist entweder ein Spezialsymbol das direkt durch die Instruktion repräsentiert wird oder ein gewöhnliches Symbol auf dessen Literalrahmenposition verwiesen wird.
   (c) Wird die Kombination von Klasse und Methode vom Empfängerfilter abgedeckt, trage den ermittelten Methodenaufruf in das Aufrufverzeichnis ein.

4.2.2 Methodenausführung

Wie in Unterabschnitt 3.4.2 dargelegt worden ist, wird für die Beschreibung einer Methodenausführung ein Methodenfilter benötigt. Die Ermittlung der auszuführenden Methoden

ist trivial, da keine weiteren Randbedingungen zu betrachten sind. Für jede Methodenspezifikation des Filters wird ein Eintrag der gegebenen Methode und zugehörigen Klasse im Ausführungsverzeichnis erzeugt.

4.2.3 Variablenzugriff

Wie in Unterabschnitt 3.4.3 gezeigt worden ist, werden für die Beschreibung eines Variablenzugriffs ein Methoden- und ein Variablenfilter benötigt. Auf dieser Grundlage wird wie folgt verfahren:

1. Erstelle eine Liste aller Methoden die der Methodenfilter auswählt.
2. Iteriere diese Liste und überprüfe jede zugehörige kompilierte Methode auf Variablenzugriffe.
   (a) Die Methode greift auf eine Instanzvariable zu. Dann enthält das Bytecode-Array der Methode eine Lese- bzw. Schreiboperation, die auf die Position verweist die die Variable in der Variablenliste der zugehörigen Klasse einnimmt. Findet die Operation hingegen in einem Block statt, handelt es sich um einen Sonderfall. Der Block greift nicht direkt auf die Instanzvariable zu, sondern indirekt über das Objekt dem sie gehört. Darauf ist zu achten.
   (b) Die Methode greift auf eine geteilte Variable zu. Dann befindet sie sich im Literalrahmen der Methode und das Bytecode-Array der Methode verfügt über eine Lese- bzw. Schreiboperation, die auf die zugehörige Literalrahmenposition verweist.
   (c) Wird die ermittelte Variable vom Variablenfilter erfasst, trage den ermittelten Variablenzugriff in das entsprechende Zugriffsverzeichnis ein.

4.3 Anreicherungseinbettung

4.3.1 Transformation

Ziel ist es dynamische aspektorientierte Programmierung zu erlauben. Anreicherungen sollen also zur Laufzeit beliebig anwendbar und entfernbar sein. Daher ist es primäres Ziel die Kosten für diese Operationen gering zu halten, während die Ausführungszeit der Anreicherungen zunächst zweitrangig ist.
Dies wird durch einen Indirektionsschritt erreicht. Sobald ein Verbindungspunkt das erste Mal anzureichern ist, wird der entsprechende Quelltextabschnitt transformiert. Dabei wird der ursprünglich effekttragende Vorgang umgeleitet, indem die zuständige Methode der für die Verbindungspunktart verantwortlichen Verbindungspunktklasse aufgerufen wird. Diese Methode führt zunächst eine Überprüfung durch, ob für den angegebenen Verbindungspunkt

tatsächlich Anreicherungen vorliegen. Dies ist notwendig, da zur Kompilierzeit wegen der dynamischen Typisierung nicht alle benötigten Informationen verfügbar sind. Ist der Laufzeittest positiv, wird aus dem Verbindungspunktregister der Klasse der zugehörige Eintrag (Abb. 21) ausgelesen. Dieser wird aktiviert und führt daraufhin alle Anreicherungen aus, die an dem Verbindungspunkt gebunden sind. Die Aktivierungsmethode liefert schließlich einen Wert zurück, der die Wirkung der ursprünglichen Aktion ersetzt. Bei einer negativen Überprüfung wird die ursprüngliche Aktion direkt ausgeführt. Der Verlauf des Vorgangs ist in Abb. 2 illustriert.

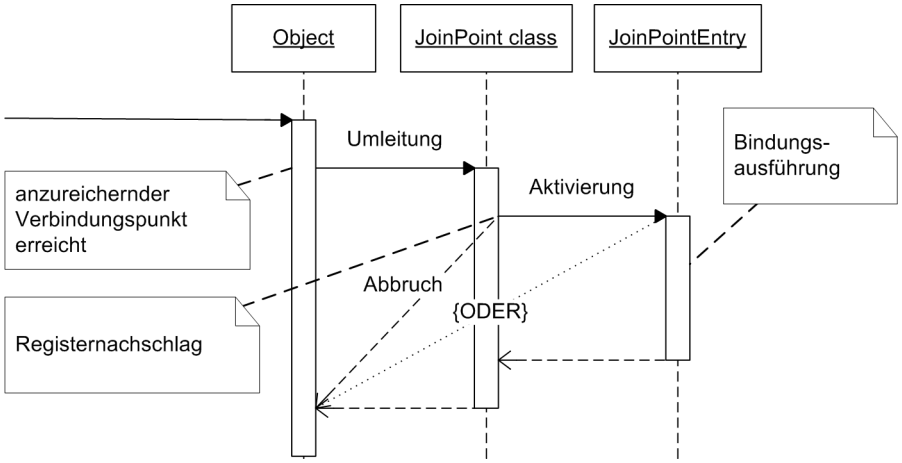

Abbildung 2: Ablauf des Durchschreitens eines Verbindungspunkts

Die Ausführung der Anreicherungen zeigt Abb. 3. Für jede Bindung einer Anreicherung

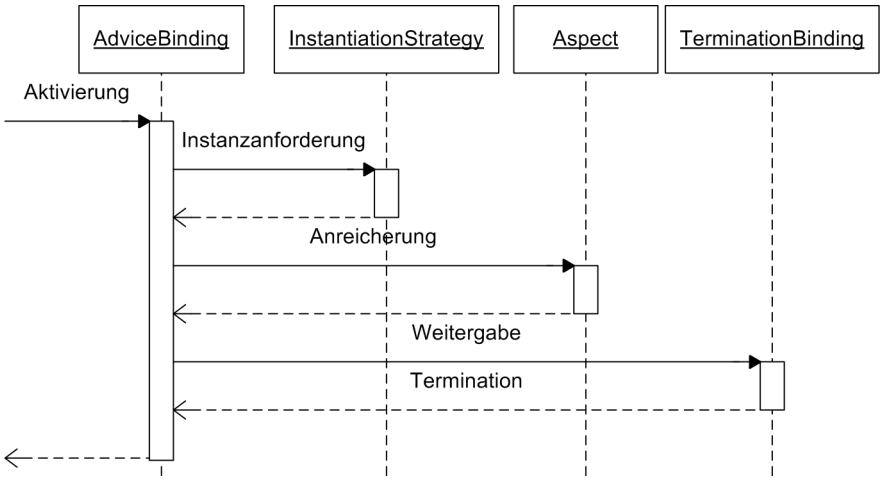

Abbildung 3: Aktivierung einer Bindung

existiert ein Bindungsobjekt (Abb. 20), das unter Verwendung der mit dem Aspekt ver-

bundenen Instanziierungsstrategie den eigentlichen Aufruf der Anreicherungsmethode des Aspekts durchführt. Danach aktiviert das Bindungsobjekt seinen Nachfolger, bis schließlich die Aufrufkette durch eine besondere Terminationsbindung beendet wird. Für vor-, herum- und nach-Bindungen gibt es je eine solche Kette.

Weder im ursprünglichen Quelltextabschnitt noch im Verbindungspunkteintrag werden die Anreicherungen direkt in den Quelltext eingebettet. Dies gestattet es neue Anreicherungen hinzuzufügen oder bestehende zu entfernen, ohne dass eine erneute Transformation erforderlich ist. Erst wenn an einem Verbindungspunkt keinerlei Anreicherungen mehr gebunden sind, wird der ursprüngliche Quelltextabschnitt wiederhergestellt.
Darüber hinaus ist der Aufwand bei einer fehlgeschlagenen Verbindungspunktüberprüfung gering, da es sich lediglich um eine fehlgeschlagene Schlüsselsuche in einer Hashtabelle handelt, die einen Zugriff in konstanter Zeit erlaubt.

4.3.2 Vorgehensweise

Die notwendigen Transformationen werden vor der Kompilierung einer Methode vom Methodentransformator durchgeführt. Der Transformator fungiert damit als Präkompilierer. Das hat den Vorteil, dass die Transformationen nach Veränderungen durch den Programmierer stets automatisch angewendet werden.
Damit dies geschehen kann, wird einer Klasse der Aspektweber (Abb. 22) als Kompilierer zugewiesen, sobald das erste Mal eine ihrer Methoden angereichert werden soll. Er arbeitet wie der Standardkompilierer mit der Ausnahme, dass er den Syntaxbaum nach seinem Aufbau an den Methodentransformator übergibt. Dieser traversiert den Baum der Tiefe nach und ersetzt solche Knoten die (möglicherweise) einen anzureichernden Verbindungspunkt repräsentieren. Anschließend wird der modifizierte Syntaxbaum dem normalen Smalltalkkompilierer übergeben. Die jeweiligen Vorgänge werden im folgenden dargelegt.

Methodenknoten

Ein Methodenknoten repräsentiert einen Ausführungspunkt.
Methodenknoten werden nur dann transformiert, wenn die Ausführung der Methode für die Klasse oder eine erbende Klasse angereichert werden soll. Nachdem der gesamte Syntaxbaum traversiert worden ist, wird zunächst eine Kopie des transformierten Methodenrumpfs erstellt und daraus ein Block erzeugt, der ausgeführt wird, nachdem die letzte Anreicherung *JoinPoint»#proceed* aufgerufen hat. Dieser Block ist aber keine bloße Kopie der Methode, sondern wird einigen Änderungen unterzogen um das gewünschte Verhalten zu erzielen und zu optimieren. Zunächst werden die Namen der Blockargumente umbenannt, damit sie nicht mit den Namen der Methodenargumente kollidieren. Weiterhin wird dem Block als letzte Anweisung *self* hinzugefügt, sofern die Methode keinen besonderen Rückgabewert liefert.

Dies ist notwendig, da eine Methode standardmäßig *self* zurückgibt, das Ergebnis eines Blocks aber das Ergebnis der letzten Anweisung ist. Aus diesem Grund wird die Rückgabeanweisung aus dem Block entfernt und durch den bloßen Rückgabewert ersetzt, sollte die Methode einen speziellen liefern. Handelt es sich dabei um die einzige Rückgabeanweisung, verhindert dies die Erzeugung eines sog. vollständigen Blocks, der mit seinem umgebenden Kontext verbunden und ineffizienter ist.
Diese Zusammenhänge werden in Abb. 4 als Transformation von *Object»#changed:* und in Abb. 5 als Transformation von *Object»#isKindOf:* verdeutlicht.

```
changed: anAspectSymbol

    self
        changed: anAspectSymbol
        with: nil
```

=>

```
changed: anAspectSymbol

    ^ExecutionPoint
        receiver: self
        receiverSelector: #changed:
        arguments: ((Array new: 1)
                        at: 1 put: anAspectSymbol;
                        yourself)
        proceed: [:_arg1 |
                    self changed: _arg1 with: nil.
                    self]
```

Abbildung 4: Transformation einer *self* zurückliefernden Methode

```
isKindOf: aClass

    ^self class includesBehavior: aClass
```

=>

```
isKindOf: aClass

    ^ExecutionPoint
        receiver: self
        receiverSelector: #isKindOf:
        arguments: ((Array new: 1)
                        at: 1 put: aClass;
                        yourself)
        proceed: [:_arg1 |
                    self class includesBehavior: _arg1]
```

Abbildung 5: Transformation einer nicht *self* zurückliefernden Methode

Nachrichtenknoten

Ein Nachrichtenknoten repräsentiert einen Aufrufpunkt.
Ob der Knoten transformiert werden muss, hängt davon ab, wie Sender und Empfänger aussehen. Über den Sender ist die Methode sowie die Klasse, nämlich die implementierende Klasse selbst oder eine ihrer erbenden Unterklassen, bekannt. Der Empfänger ist hingegen nicht immer eindeutig zu ermitteln. Eine Einschränkung ergibt sich bei *self*, *super*, einer Klasse oder einem Literal. Allerdings kann der Empfänger auch eine Variable oder wiederum Ergebnis eines weiteren Aufrufs sein. In diesem Fall ist nur der Name der aufzurufenden Methode sicher bekannt. Daher muss der Aufruf transformiert werden, sobald im Aufrufspunktverzeichnis für einen der möglichen Empfänger ein Eintrag existiert, der sich auf den Namen der aufzurufenden Methode bezieht. Hier zeigt sich ein Nachteil der dynamischen Typisierung. Einzig in den zuvor genannten Fällen der sicheren Empfänger kann eine Transformationen u.U. ausgeschlossen werden.
Abbildung 6 zeigt ein Beispiel für einen sicheren Empfänger. Soll der Aufruf von *Object»#allOwnersWeakly:* in *Object»#allOwners* nicht angreichert werden, kann eine Transformation entfallen.

```
allOwners                          allOwners

    ^self allOwnersWeakly: false       ^CallPoint
                                           sender: self
                                           senderSelector: #allOwners
                                   =>      receiver: self
                                           receiverSelector: #allOwnersWeakly:
                                           arguments: ((Array new: 1)
                                                       at: 1 put: false;
                                                       yourself)
```

Abbildung 6: Transformation des Aufrufs einer Methode von *self*

Kaskadenknoten

Eine Nachrichtenkaskade besteht aus mindestens zwei Nachrichten an denselben Empfänger, wobei der Empfänger nur einmal genannt wird. Der Rückgabewert einer Kaskade ist der Rückgabewert ihrer letzten Nachricht.
Stellt eine dieser Nachrichten einen möglichen Verbindungspunkt dar, lässt sich die Transformation des Nachrichtenknotens zunächst nicht durchführen. Zuvor muss die Kaskade aufgebrochen werden. Dazu wird sie in die Form gebracht, für die sie als Verkürzung bzw. Vereinfachung steht. Enthält der Kaskadenknoten also n Nachrichten, wird er durch eine Sequenz

aus n Nachrichtenknoten ersetzt (Abb. 7). Ist der Empfänger der Kaskade das Ergebnis einer Nachricht, muss dieses Ergebnis zuvor in einer Variable abgespeichert werden, da die Nachricht u.U. nicht idempotent ist und somit mehrfache Aufrufe unterschiedliche Ergebnisse liefern (Abb. 8). Diese Maßnahme entfällt, wenn der Empfänger der Kaskade *self*, *super*, eine Klasse, ein Literal oder bereits eine Variable ist. Die erzeugten Nachrichtenknoten können dann vom Transformator bei Bedarf bearbeitet werden.

```
transform                          transform
    self                               self createGetterAccessor.
        createGetterAccessor;   =>     self createSetterAccessor
        createSetterAccessor
```

Abbildung 7: Transformierte, sichere Nachrichtenkaskade

```
asSortedStrings: aSortBlock                asSortedStrings: aSortBlock
                                               | _cascadeReceiver1 |
    ^(SortedCollection forStrings: self size)
        sortBlock: aSortBlock;                 _cascadeReceiver1 := SortedCollection forStrings: self size.
        addAll: self;                   =>     _cascadeReceiver1 sortBlock: aSortBlock.
        yourself                               _cascadeReceiver1 addAll: self.
                                               ^_cascadeReceiver1 yourself
```

Abbildung 8: Transformierte, unsichere Nachrichtenkaskade

Variablenknoten

Ist ein Variablenknoten nicht Variablenknoten eines Zuweisungsknotens, steht er für das Auslesen einer Variable. Soll z.B. das Auslesen der Instanzvariable *superclass* in der Methode *Behavior»#superclass* angereichert werden, ergibt sich die in Abb. 9 gezeigte Transformation.

Zuweisungsknoten

Der Variablenknoten, der die zu belegende Variable repräsentiert, wird nicht transformiert, um das fälschliche Erkennen eines Auslesevorgangs zu vermeiden.
Soll z.B. das Setzen der Instanzvariable *environment* in der Methode *Class»#environment:* angereichert werden, ergibt sich die in Abb. 10 gezeigte Transformation.

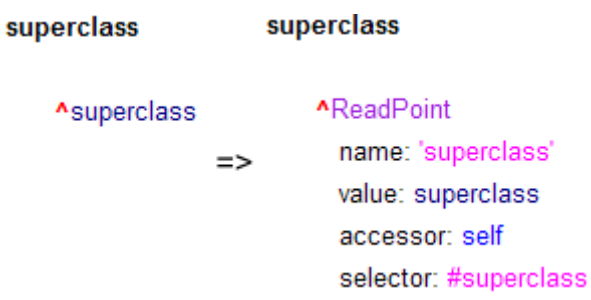

Abbildung 9: Transformation des Auslesens einer Instanzvariable

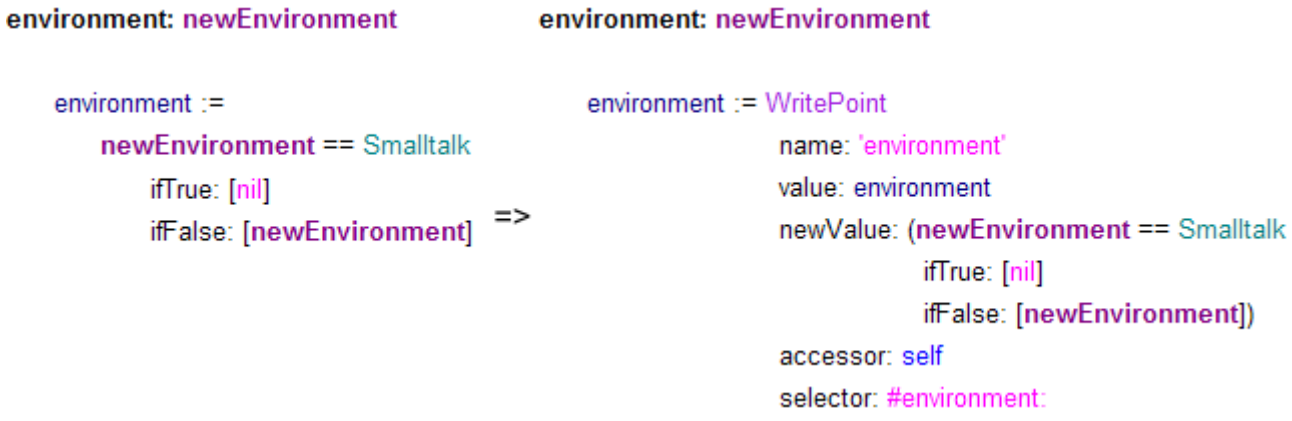

Abbildung 10: Transformation des Setzens einer Instanzvariable

4.3.3 Alternativen

Smalltalk gewährt eine Reihe von Möglichkeiten auf den Kontrollfluss einzuwirken, wobei die Mehrzahl den Nachrichtenversand betrifft (BFJR98; Duc99). Allerdings unterscheiden sich diese in ihrer Mächtigkeit und Aufwendigkeit erheblich. Die möglichen Alternativen werden im folgenden hinsichtlich ihres Ablaufs, ihrer Granularität (Verbindungspunktabdeckung) und Auswirkungen auf das Kriterium der Transparenz untersucht um aufzuzeigen warum sie keine Anwendung finden.

Modifikation auf Quelltextebene

Bei dieser Veränderung werden die nötigen Umwandlungen direkt an den im Quelltext identifizierten Verbindungspunkten vorgenommen und das Ergebnis wird kompiliert.
Da die vorgenommenen Änderungen direkt sichtbar sind und diese vom Programmierer bei eigenen Quelltextveränderungen berücksichtigt werden müssen, wird die Forderung nach Transparenz verletzt. Um die Modifikationen zu verbergen müsste der Zeiger der auf den Quelltext verweist so verändert werden, dass der ursprüngliche Quelltext erscheint. Ansons-

ten ist das Vorgehen der Transformation identisch zum in AspectTalk gewählten Weg.
Die Quelltextmodifikation ermöglicht es alle drei Verbindungspunktarten zu erfassen.

Modifikation auf Methodenebene

Auf Methodenebene gibt es zwei Möglichkeiten Modifikationen vorzunehmen. Beiden ist gemein, dass die ursprüngliche Methode erhalten bleibt, aber durch eine neue ersetzt wird, die die Transformation enthält und die alte Methode bei Bedarf aufruft. Damit ist nur die Methodenausführung direkt anreicherbar. Die Anreicherung eines Methodenaufrufs muss simuliert werden, indem bei der Ausführung der aufzurufenden Methode Anreicherungen in Abhängigkeit vom Methodenkontext (*thisContext*) eingebracht werden. Nach diesem Prinzip lässt sich auch ein Variablenzugriff abdecken, wobei der Zugriff zuvor durch einen Methodenaufruf, der die betroffene Variable liefert bzw. belegt, zu ersetzen ist.

Bei der ersten Variante wird die ursprüngliche Methode umbenannt. Unter dem alten Namen wird eine Methode gespeichert, die für die Ausführung der anzuwendenden Anreicherungen sorgt. Dieses Verfahren verletzt die Transparenz, da die Klassenschnittstelle um eine neue Methode erweitert wird. Außerdem kann es durch bereits vorhandene Namen bzw. mit zukünftigen Namen zu Kollisionen kommen. Alte wie auch neue Methode sind direkt einsehbar. Um sie zu verbergen müsste der Quelltext der neuen Methode auf den der alten verweisen. Weiterhin müssten die Systemwerkzeuge derart modifiziert werden, dass die alte Methode ausgeblendet wird.
Bei der zweiten Variante handelt es sich um sog. *MethodWrapper* wie sie auch in AspectS eingesetzt werden. Hierbei wird die ursprüngliche kompilierte Methode ebenfalls ersetzt. Allerdings wird die alte Methode nicht unter einem neuen Namen gespeichert und aufgerufen, sondern direkt über *CompiledMethod»#valueWithReceiver:arguments:* vom Wrapper ausgeführt. Wie auch zuvor müsste der Quelltext der neuen Methode auf den der alten Methode verweisen, da sonst die Modifikation unmittelbar ersichtlich ist.

Vorteilhaft bei diesen Verfahren ist, dass es nicht des Kompilierers bedarf. Die Kompilation lässt sich durch Prototypen von Methoden unterschiedlicher Argumentanzahl vermeiden. Diese Prototypen müssen lediglich kopiert werden und die notwendigen Informationen werden direkt in den Literalrahmen der Methode eingebracht.

Modifikation auf Klassenebene

Auch auf Klassenebene gibt es zwei Vorgehensweisen. Beiden ist gemein, dass die ursprüngliche Methode vollständig erhalten bleibt, aber eine neue eingesetzt wird, die die Transformation enthält und die alte Methode bei Bedarf aufruft. Dadurch ist wie schon auf der Methodenebene nur die Ausführung einer Methode direkt anreicherbar. Der Aufruf einer

Methode muss wie zuvor simuliert werden. Allerdings ist der Variablenzugriff nicht erfassbar, da die Originalmethode unverändert bleibt und er somit nicht durch einen entsprechenden Methodenaufruf ersetzbar ist.

Bei der ersten Variante, die in AOP/ST zum Einsatz kommt, wird eine neue Unterklasse der Klasse angelegt, die die anzureichernde Methode enthält. In der Unterklasse wird dann eine Methode gleichen Namens definiert, die für die Ausführung der Anreicherungen verantwortlich ist. Die Ausführung der ursprünglichen Methode geschieht durch Aufruf der Oberklassenmethode. Damit dieses Verfahren wirksam werden kann, müssen schließlich noch alle Instanzen der Oberklasse ermittelt und ihre Klasse durch die neue Unterklasse ersetzt werden. Außerdem muss dafür gesorgt werden, dass neue Instanzen ebenfalls die neue Klasse zugewiesen bekommen. Weiterhin ergibt sich noch ein Identitätsproblem. Fragt man ein angepasstes Objekt nach seiner Klasse wird mit der neuen Unterklasse anstatt der erwarteten Oberklasse geantwortet.
Bei der zweiten Variante handelt es sich um Proxies, die ein gegebenes Objekt kapseln. Hierbei werden sog. minimale Klassen verwendet, deren Oberklasse *nil* ist. Dadurch wird der Aufruf jeder Methode zu einem Fehler und somit zur Aktivierung von *#doesNotUnderstand:* führen. Innerhalb dieser Methode werden die Anreicherungen ausgeführt. Schließlich wird die ursprüngliche Methode auf dem gekapselten Objekt aufgerufen. Soll eine Methode nicht angereichert werden, wird sie einfach an das gekapselte Objekt weitergeleitet. Wie schon zuvor müssen alle Instanzen der Klasse zu der die anzureichernde Methode gehört ermittelt werden. Anstatt ihre Klasse zu ändern, müssen ihre Objektzeiger mittels *#become:* auf einen Proxy umgewandelt werden. Dabei ergibt sich ein weiteres Problem. Da lediglich Objektzeiger verändert werden, können nur von außen eingehende Nachrichten abgefangen werden. Methoden die über *self* aufgerufen werden bleiben unerreichbar.

Modifikation auf Bytecodeebene

Hierbei werden die Veränderungen direkt am Bytecode vorgenommen, wie es z.B. in AspectJ geschieht. Aus der ursprünglichen Methode wird direkt eine neue kompilierte Methode erzeugt, die um notwendige Befehle und Literale angepasst wird. Alle anderen Attribute bleiben erhalten. Dieses Vorgehen ähnelt damit der Modifikation auf Quelltextebene mit der Ausnahme, dass der Kompilierer nicht verwendet werden muss. Allerdings ist eine direkte Bytecodebehandlung äußerst komplex, da ein Großteil des Wissens des Kompilierungsrahmenwerks reproduziert werden muss und die Bytecodes nicht dialektübergreifend standardisiert sind. Dies ist aber durch vorgefertigte Schablonen der erforderlichen Befehlsfolgen ausgleichbar. Mittels Bytecodemodifikation sind alle Verbindungspunktarten abdeckbar.

Modifikation des Interpreters

Bei dieser Variante wird der Vorgang der Bytecode-Interpretation angepasst. Dies hat den Vorteil, dass weder an vorhandenen Quelltexten noch an den Objekten selbst bestimmte Änderungen vorzunehmen sind. Damit ist maximale Transparenz gewährleistet. Allerdings entsteht auch ein ständiger Mehraufwand, da bei jeder betroffenen Aktion nach Anreicherungen gesucht werden muss, selbst wenn gar keine Anreicherungen vorgenommen worden sind.
Der Interpretationsvorgang wird in (GR83) ausführlich und in Smalltalk selbst beschrieben, wobei Bytecodes mit Methoden assoziiert werden. Durch Veränderung dieser Methoden lässt sich aspektorientierte Programmierung unterstützen. Im folgenden werden dabei nur das Prinzip aber keine konkreten Maßnahmen dargelegt, da solche stark von der tatsächlichen Implementation des Interpreters abhängig sind.
Von Bedeutung sind die Methoden *#activateNewMethod*, *#pushReceiverVariable:* und *#storeAndPopReceiverVariableBytecode*. Die erste Methode ist für die Ausführung einer Methode verantwortlich. Durch ihre Modifikation lassen sich Anreicherungen von Methodenausführungen und auch Methodenaufrufen (durch Zugriff auf den intern vorhandenen Stapel) umsetzen. Die anderen beiden Methoden verarbeiten das Auslesen und Belegen einer Instanzvariable. Mittels ihrer Anpassung sind somit auch Variablenzugriffe anreicherbar. Geteilte Variablen werden als Assoziationen (Schlüssel-Wert-Paare) im Literalrahmen einer zugreifenden Methode gespeichert. Um ihren Zugriff anzureichern müssen die beiden Methoden *#pushLiteralVariable:* und *#extendedStoreBytecode* verändert werden.
Mittels Interpretermodifikation sind alle Verbindungspunktarten abdeckbar.

4.4 Zeitaufwand

Neben den vielfältigen Möglichkeiten der aspektorientierten Programmierung ist es ebenso wichtig diese auch mit möglichst geringen Zeitkosten einsetzen zu können. Kritisch sind dabei der Aufwand für die Anwendung von Anreicherungen (Tabelle 5) und den notwendigen Webvorgang (Tabelle 6). Zuerst hängt dies natürlich vom verwendeten System ab, auf dem die Messungen durchgeführt werden[2]. Jedoch geben die folgenden Ergebnisse einen Eindruck der ungefähren Größenordnungen.

Die Laufzeit einer Methode und auch Anreicherung hängt von ihrem Umfang und ihren tatsächlichen Aktionen ab. Von Bedeutung ist außerdem die verwendete Instanziierungsstrategie und die Anzahl der Anreicherungen. Um daher möglichst allgemeingültige Aussagen treffen zu können, wird der Zeitaufwand für das Durchlaufen des für die Anreicherungsaktivierung zuständigen Mechanismus gemessen. Der Verbindungspunkt ist also transformiert,

[2]Hier: AMD-Athlon-64-3000-Prozessor, 1,5 GB RAM, Windows XP, VisualWorks 7.4.1, Durchschnitt von 5.000.000 Wiederholungen

verfügt aber über keine Anreicherungen, sondern nur die in Unterabschnitt 4.3.1 erwähnten Terminationsbindungen.
Dem gegenüber steht der Zeitaufwand für das Durchlaufen des untransformierten Verbindungspunkts. Diese sind so einfach wie möglich gehalten. Die Methode deren Ausführung und Aufruf anzureichern ist liefert lediglich *self* zurück. Die Variablen deren Zugriff anzureichern ist werden nur zurückgegeben bzw. mit *nil* belegt.
Somit ist die Differenz der benötigten Zeit für den transformierten und untransformierten Verbindungspunkt ein Maß für die Kosten, die für eine Anreicherung des jeweiligen Verbindungspunkts einmalig anfallen, unabhängig von den Anreicherungen selbst:

	Methode		**Instanzvariable**		**Geteilte Variable**	
	Ausführung	**Aufruf**	**Lesen**	**Schreiben**	**Lesen**	**Schreiben**
untransformiert	1,19 μs	1,21 μs	1,21 μs	1,21 μs	1,21 μs	1,23 μs
transformiert	2,50 μs	2,54 μs	2,55 μs	2,45 μs	2,50 μs	2,44 μs
Kosten	1,31 μs	1,33 μs	1,34 μs	1,24 μs	1,29 μs	1,21 μs

Tabelle 5: Kosten von Verbindungspunktaktionen

Es zeigt sich, dass in allen Fällen je Verbindungspunkt durchschnittlich ein einmaliger Mehraufwand von etwa 1,3 μs entsteht. Dies ist sehr schnell und etwas länger als ein einfacher Methodenaufruf. Weiterhin sind die Kosten relativ betrachtet ähnlich zu denen von AspectJ und deutlich geringer als die anderer aspektorientierter Erweiterungen Javas[3].
Neben den Kosten der Anreicherungsaktivierung ist es ebenso wichtig zu wissen, wie lange es dauert eine Anreicherung überhaupt zu binden und zu entfernen. Hierbei sind aufgrund des in Abschnitt 4.3 beschriebenen Transformationsvorgangs vier Fälle von Bedeutung: der Aufwand des Hinzufügens bzw. Löschens der ersten bzw. letzten Anreicherung und des Hinzufügens bzw. Löschens jeder anderen Anreicherung. Neben dem eigentlichen Webvorgang gibt es wie zuvor weitere Einflussfaktoren wie die verwendeten Elementfilter und die Anzahl der anzureichernden Verbindungspunkte bzw. den Umfang der Methode. Daher sind die in Tabelle 6 gelisteten Werte die Ergebnisse der schon zuvor verwendeten vereinfachten Methoden und schließen die Filterung nicht mit ein:
Wie zu erwarten ist, sind das Hinzufügen und Entfernen von ersten bzw. letzten Anreicherung um durchschnittlich zwei Größenordnungen teurer als die anderen Fälle, wobei das Entfernen günstiger als das Hinzufügen ist. Dies hängt damit zusammen, dass in beiden Fällen die entsprechende Methode kompiliert, beim Hinzufügen aber auch transformiert werden muss.

[3] http://docs.codehaus.org/display/AW/AOP+Benchmark

	Methode		Instanzvariable		Geteilte Variable	
	Ausführung	**Aufruf**	**Lesen**	**Schreiben**	**Lesen**	**Schreiben**
Erste hinzufügen	1130,03 μs	1198,19 μs	1169,60 μs	1091,61 μs	1117,15 μs	1134,43 μs
Weitere hinzufügen	8,23 μs	10,72 μs	8,93 μs	9,46 μs	15,44 μs	15,25 μs
Letzte löschen	1039,80 μs	1061,18 μs	992,33 μs	994,96 μs	1006,66 μs	1032,47 μs
Andere löschen	31,40 μs	36,80 μs	26,51 μs	29,50 μs	29,45 μs	30,88 μs

Tabelle 6: Kosten von Verbindungspunktaktionen

4.5 Anreicherungsklassifikation

Um eine Anreicherung zu analysieren muss, wie in Abschnitt 3.5 dargelegt worden ist, ermittelt werden, wie sie mit dem Verbindungspunkt interagiert. Dafür ist die Klasse *AdviceAnalyzer* zuständig. Sie veranlasst zunächst den Aufbau des Syntaxbaums der Anreicherung und traversiert diesen anschließend. Dabei hängen die vorzunehmenden Untersuchungen vom jeweiligen Knotentyp ab:

- **Zuweisungsknoten:** Wird einer Variablen das ursprüngliche Verbindungspunktobjekt, d.h. das einzige Argument der Anreicherung, oder einer seiner Aliase zugewiesen, ist die Variable ein neuer Alias des Verbindungspunktobjekts.
- **Nachrichtenknoten:** Ist der Empfänger das Verbindungspunktobjekt, d.h. das einzige Argument der Anreicherung oder einer seiner Aliase, hängt das weitere Vorgehen vom Methodennamen ab.
  - **Lesezugriff:** Handelt es sich bei der zu sendenden Methode um *#argumentAt:*, *#receiver*, *#value* oder *#newValue*, greift die Anreicherung lesend auf den Verbindungspunkt zu. Der Methodenname wird dann in die Verlaufsliste eingetragen.
  - **Schreibzugriff:** Handelt es sich bei der zu sendenden Methode um *#argumentAt:Put:*, *#receiver:*, *#value:* oder *#newValue:*, greift die Anreicherung schreibend auf den Verbindungspunkt zu. Der Methodenname wird dann in die Verlaufsliste eingetragen.
  - **Fortfahren:** Handelt es sich bei der zu sendenden Methode um *#proceed*, wird die nächste Anreicherung bzw. die ursprüngliche Aktion des Verbindungspunktes aufgerufen. *#proceed* wird dann in die Verlaufsliste eingetragen. Damit *#proceed* stets genau einmal aufgerufen wird, müssen zusätzlich die folgenden Bedingungen gelten:
    - *#proceed* darf nicht nach einer Rückgabeanweisung auftreten.
    - *#proceed* darf nicht Teil eines Blocks sein. Damit entfallen bedingte oder mehrfache Ausführungen z.B. in Schleifen. Davon ausgenommen sind Fall-

unterscheidungen wie *#ifTrue:ifFalse:* oder *#ifNil:ifNotNil:*, vorausgesetzt *#proceed* wird in jedem Block genau einmal aufgerufen.

- *#proceed* muss, mit zuvor genannter Ausnahme, genau einmal in der Verlaufsliste erscheinen.

Die während der Prozedur erstellte Verlaufsliste soll Grundlage für eine spätere automatische Konfliktanalyse (s. Unterabschnitt 6.2) sein. Durch die Kombination der Verlaufslisten der einzelnen Anreicherungen eines Verbindungspunktes lassen sich die klassischen Lese-Schreib-Konflikte und die daraus resultierenden Anomalien erkennen.

4.6 Datenhaltung

AspectTalk verfügt über eine umfassende Informationssammlung die dezentral in den jeweils verantwortlichen Teilsystemen abgelegt ist. Sie setzt sich wie folgt zusammen:

- **Verbindungspunktregister**
  Jede Verbindungspunktklasse führt ein Register, das jedem anzureichernden Verbindungspunkt die erforderlichen Bindungen zuordnet (Abb. 21).

- **Verbindungspunktauswahlregister**
  Die Klasse *Pointcut class* führt ein Verzeichnis, das jedem Aspekt seine Verbindungspunktauswahlen zuordnet.

- **Bindungsregister**
  Die Klasse *AdviceBinding class* führt ein Verzeichnis, das jeder Anreicherung eines Aspekts ihre Bindungen zuordnet.

- **Instanziierungsstrategieregister**
  Die Klasse *InstantiationStrategy class* führt ein Verzeichnis, das jedem Aspekt seine Instanziierungsstrategie zuordnet.

Für einen schnellen Datenzugriff kommen soweit es möglich und erforderlich ist Hashverfahren zum Einsatz. Angesichts der Vielzahl an Daten handelt es sich um einen klassischen Zeit-Speicher-Zielkonflikt. Aus den nachfolgend angegebenen Gründen ist zugunsten der Zeit entschieden worden.
Zum einen soll unter dem Gebot der Transparenz AspectTalk so wenig wie möglich in Basisprogramme eingreifen. Somit verbietet es sich Daten über zusätzliche Instanzvariablen direkt in Klassen und Methoden einzubetten. Zwar erlaubt dies einen schnelleren Zugriff. Doch beschränkt dies den Basisprogrammierer in seiner Freiheit der Namens- und Strukturwahl. Damit ist die Transparenz eingeschränkt, da er die Aspektorientierung berücksichtigen muss. Außerdem besitzen spezielle Klassen einen festen Aufbau, der von der virtuellen Maschine erwartet wird und nicht verändert werden darf bzw. kann.

Zum anderen dient die Datenhaltung der schnellen und einfachen Verarbeitung von Änderungen, speziell von Refaktorisierungen (s. Kapitel 5), um so dem Gebot der Dynamik gerecht zu werden. Es ist effizient entscheidbar, ob und in welchem Umfang Anpassungen nötig sind um einen konsistenten Systemzustand zu gewährleisten, da die benötigten Informationen bereits vorhanden sind und nicht erst ermittelt werden müssen. Darüber hinaus basieren Werkzeuge wie die Anreicherungsanalyse oder die Oberflächenerweiterungen wesentlich auf der Verfügbarkeit dieser Informationen.

4.7 Effiziente Filterung

Wie in Abschnitt 3.3 erwähnt worden ist, lassen sich eigene Formen und Typen von Elementfiltern erzeugen. Die Verbindungspunktauswahlen erwarten lediglich, dass die ihn als Elementfilter übergebenen Objekte *#asFilter* verstehen und diese Methode ein entsprechendes Filterobjekt liefert. Das Standardprotokoll der Filterobjekte ist relativ klein. Allerdings gibt es zwei Methoden, die weitreichende Auswirkungen haben und daher hier hervorgehoben werden sollen. Es handelt sich dabei um *#»* und *#extent*.
Der Aufbau komplexer Filter erfolgt in einer festen hierarchischen Ordnung bzw. Schachtelung. Die einzelnen Teile eines zusammengesetzten Elementfilters lassen sich nicht beliebig anordnen. Will man z.B. die Methode *Set»#yourself* auswählen, muss dies in folgender Form geschehen:

```
'Set' class2 » 'yourself' method
```

Eine Ordnung wie

```
'yourself' method » 'Set' class2
```

ist ungültig. Daraus ergibt sich zwar eine starre Hierarchie bzw. Schachtelung von Filtern. Allerdings wäre der Zeitaufwand bei freier Ordnung u.U. immens. Im letzten Beispiel müsste man zunächst alle Methoden namens *#yourself* filtern, wobei dies jede Klasse des Systems betrifft, nur um schließlich herauszufinden, dass lediglich die Klasse *Set* gemeint ist. Erst die Klasse zu ermitteln, in diesem Fall *Set*, und danach die Methode, in diesem Fall *#yourself*, ist hingegen schneller. Dies gilt nicht nur für Klassen und Methoden, sondern für jedes Element. Daher muss ein Erweiterer darauf achten, dass sein Elementfilter nicht in beliebiger Reihenfolge erscheinen kann und über die Methode *#»* am hierarchischen Aufbau des Gesamtfilters teilnehmen muss.
Ähnliches gilt für die Methode *#extent*. Diese Methode liefert einen numerischen Wert größer 0 zurück, der Auskunft über die Kosten der Filterung aller Elemente des Filters gibt. Je höher dieser Wert ist, desto mehr Elemente liefert der Filter und/oder umso länger bzw. aufwändiger ist die Filterung. Der Wert 0 bedeutet dabei, dass kein oder genau ein Element

geliefert wird. Die Information über die Kosten einer Filterung dienen den Und-Filtern als Sortierkriterium. Da bei einer Und-Verknüpfung all die Elemente ermittelt werden, die von jedem einzelnen Filter abgedeckt werden, ist es offensichtlich am effizientesten, nur die Elemente zu überprüfen, die der Filter mit den wenigsten bzw. am einfachsten zu ermittelnden Elementen zurückliefert. Dazu ein Beispiel: Um alle geteilten Variablen anzusprechen die eine bestimmte Methode liest, gibt es zwei Möglichkeiten:

('C' class2 » 'm' method read) & ('*' shared)
('*' shared) & ('C' class2 » 'm' method read)

Es ist eindeutig günstiger, zunächst die Variablen zu ermitteln, die von *C»#m* gelesen werden und daraus die geteilten Variablen zu filtern, als alle geteilten Variablen des Systems zu finden und daraus die tatsächlich gelesenen auszuwählen. Die Kosten sind dabei aber nur eine Heuristik. Eine genaue Elementanzahl zu finden erfordert letztlich die Überprüfung jedes Elements, was aber gerade vermieden werden soll. Zumindest handelt es sich aber um ein NP-hartes Problem, dass in der Logik als Konjunktordnungsproblem bekannt ist:

> Finde eine Ordnung der Konjunkte der Voraussetzung einer Regel, sodass die Gesamtkosten zur Wahrheitsbestimmung der Regel minimal sind.

Nichtsdestotrotz kann eine gute Heuristik die benötigten Kosten erheblich senken. Die Heuristik der Standardfilter ist recht einfach. Alle Standardfilter arbeiten mit einem regulären Ausdruck und/oder weiteren Filtern. Die Kosten sind daher der Quotient aus Anzahl des Vorkommens von '*' und der Gesamtlänge des Ausdrucks. Bei zusammengesetzten Filtern werden zusätzlich die Kosten der weiteren Filter aufaddiert. Schon vier einfache Beispiele für Namensraumfilter (Tabelle 7) zeigen die Wirksamkeit der Heuristik:

Filter	**Kosten**
'*' nspace	$\frac{1}{1} = 1$
'*a*' nspace	$\frac{2}{3} = 0{,}\overline{6}$
'A*' nspace	$\frac{1}{2} = 0{,}5$
'Kernel' nspace	$\frac{0}{6} = 0$

Tabelle 7: Filterungskostenbeispiel

Im ersten Fall wird jeder Namensraum ausgewählt, was am teuersten ist. Die zweite Auswahl erfasst alle Namensräume, die ein a im Namen enthalten. Dies sind weniger Möglichkeiten als vorher, aber die Kosten sind noch relativ hoch. Im dritten Fall sind die Kosten wieder niedriger, da nun nur Namensräume deren Namen mit A beginnen ausgewählt werden. Am günstigsten ist jedoch die letzte Auswahl, da nur ein Namensraum erfasst wird.

Herkunftsfilter

Ein spezieller Filter, dessen Arbeitsweise hervorgehoben werden soll, ist der Herkunftsfilter. Er wählt diejenigen Methoden aus, die letztlich auf den Aufruf einer bestimmten Methode m zurückgehen. Ein geeignetes Mittel dies zu überprüfen ist das Durchsuchen des Stapels, der in Smalltalk mittels der Pseudovariable thisContext verfügbar ist. Dieses Vorgehen ist jedoch teuer, da der Kontext erst erzeugt und im schlimmsten Fall der gesamte Stapel nach m durchsucht werden muss. Statt dessen wird an die Ausführung von m zwei Anreicherungen gebunden. Eine wird vor der Ausführung aktiviert und erhöht einen Zähler für m und den aktiven Thread, während die andere nach der Ausführung läuft und den Zähler erniedrigt. Soll nun im Kontext von m eine Anreicherung eingesetzt werden, wird vor ihrer Ausführung überprüft, ob ein Zähler für m und den aktiven Thread vorliegt.
Einen Zähler anstelle eines nahe liegenden bool'schen Werts zu verwenden ist wichtig, da eine Methode sich z.B. rekursiv aufrufen kann und das Beenden des letzten Aufrufs den Wert auf false setzt und damit fälschlicherweise der gesamte Aufruf als beendet angesehen wird. Den Zähler weiterhin in Abhängigkeit von Threads zu führen verhindert bei nebenläufigen Ausführungen der gleichen Methode Wechselwirkungen, die ebenfalls zum vorzeitigen Erkennen des Ausführungsendes führen.

Standardmäßig wählt der Herkunftsfilter nur die Methoden aus und analysiert diese rekursiv, die sicher bestimmbar sind, also auf *self*, *super*, einer Klasse oder einem Literal aufgerufen werden. Der Grund dafür ist, dass im Falle des Aufrufs auf einer Variable wegen der dynamischen Typisierung jede Klasse berücksichtigt werden muss, die die Methode implementiert. Im Falle von z.B. *#yourself* betrifft dies alle Klassen des Systems. Dafür wird unverhältnismäßig viel Zeit benötigt und u.U. werden sehr viele überflüssige Auswahlen getroffen. Bei Bedarf bzw. Interesse kann dieser Fall aber aktiviert werden, da er im Quelltext lediglich auskommentiert ist. Eine mögliche Lösung dieses Problems ist in Unterabschnitt 6.2 beschrieben.

5 Anpassung an Änderungen durch Reflexion

In diesem Kapitel wird eine Eigenschaft AspectTalks, die Anpassung an Änderungen des Basisprogramms, detaillierter betrachtet. Die Anpassung geschieht unter Ausnutzung der in Smalltalk gegebenen Reflexion und den Prinzipien aspektorientierter Programmierung selbst. Somit sind die gewonnenen Erkenntnisse auch auf andere Sprachen übertragbar, die über vergleichbare oder weitergehende Reflexion verfügen.

5.1 Problem der fragilen Verbindungspunktauswahl

Programme unterliegen meistens einer kontinuierlichen Weiterentwicklung. Die Anwesenheit von Aspektorientierung kann ihre Entwicklungsfähigkeit allerdings erschweren. (KGBM) definieren in diesem Zusammenhang das Problem der fragilen Verbindungspunktauswahl:

> Das Problem der fragilen Verbindungspunktauswahl tritt in aspektorientierten System auf, wenn Verbindungspunkte als Folge ihrer Fragilität in Bezug auf scheinbar sichere Änderungen des Basisprogramms unbeabsichtigt ausgewählt oder ausgelassen werden.

Dies ist eine triviale Erkenntnis, die in der Praxis allerdings von entscheidener Bedeutung ist. Viele Verbindungspunkte stützen sich auf Quelltexteigenschaften wie z.B. Variablen- oder Methodennamen, wodurch sie eng an diese gekoppelt sind. Bei Veränderungen dieser Eigenschaften müssen auch die Auswahlen entsprechend angepasst werden. Dafür hat meistens der verändernde Programmierer manuell zu sorgen, was jedoch der von aspektorientierter Programmierung geforderten Transparenz widerspricht. Eine Automatisierung solcher Vorgänge durch Werkzeuge ist nicht gängig, obwohl die klassischen Refaktorisierungen in jeder gehobenen Entwicklungsumgebung zu finden sind.
(KGBM) schlagen zur Lösung des Problems sog. modellbasierte Auswahlen vor. Sie wählen Verbindungspunkte auf Grundlage eines konzeptionellen Modells des Basisprogramms und nicht seiner Implementierung aus. Diese Abstraktion ähnelt dem Konzept der semantischen Verbindungspunktauswahl nach (ACK). Hierbei werden Verbindungspunkte auf Basis von Annotationen ausgewählt, die sich auf ein semantisches Modell des Programms beziehen. Die Nutzung solcher Metainformationen findet sich seit Version 5.0 auch in AspectJ durch Einbeziehung von Java-Annotationen. Auch in AspectTalk bzw. VisualWorks bietet sich durch Pragmas und Pragmafilter diese Möglichkeit.
Solche Vorgehensweisen ermöglichen eine neue Qualität der Ausdrucksfähigkeit und sind sicherlich wünschenswert. Allerdings verlagert sich das Problem lediglich auf eine abstraktere Ebene, da nun das konzeptuelle Modell mit dem Basisprogramm bei Abänderungen synchronisiert werden muss, worauf sogar in (KGBM) hingewiesen wird. Auch wenn Änderungen auf dieser Ebene seltener sind als auf Quelltextebene, ist eine automatische Anpassung wie sie

sich in typischen Refaktorisierungswerkzeugen findet letztlich unausweichlich. Außerdem ist dies grundlegend um der Forderung nach Dynamik gerecht zu werden.

5.2 Smalltalks Metaobjektprotokoll

Smalltalk ist nicht nur eine Programmiersprache, sondern zugleich eine graphische, interaktive Programmierumgebung (GR83). Dieser Umstand ergibt sich aus der umfassenden reflexiven Architektur auf der Smalltalk aufbaut. Smalltalksysteme besitzen eine vollständige Selbstbeschreibung. Durch Manipulation bzw. Erzeugung dieser Metainformationen definiert sich das System in seinem Zustand und Verhalten selbst. Um die notwendigen Informationen verfügbar zu machen werden sie in sog. Metaobjekten vergegenständlicht (reifiziert). Die Funktionalität dieser Metaobjekte wird als Metaobjektprotokoll bezeichnet.
Smalltalk verfügt über eine Reihe unterschiedlicher Metaobjekte. Eine umfassende Beschreibung ist in (Riv) zu finden. Für die Betrachtungen des nächsten Abschnitts sind jedoch nur die relevant, die die Struktur von Klassen beschreiben.

Alles in Smalltalk ist ein Objekt. Da Objekte Instanzen von Klassen sind, sind auch Klassen Objekte, die wiederum eigene Klassen besitzen. Diese Beziehung setzt sich allerdings nicht endlos fort. Das genaue Verhältnis zwischen Objekten und Klassen lässt sich in fünf Sätzen beschreiben (GR83):

1. Jede Klasse ist letztlich eine Unterklasse der Klasse *Object*. *Object* selbst besitzt keine Oberklasse.

2. Jedes Objekt ist Instanz einer Klasse.

3. Jede Klasse ist Instanz einer Metaklasse.

4. Jede Metaklasse ist letztlich einer Unterklasse der Klasse *Class*.

5. Jede Metaklasse ist Instanz der Klasse *Metaclass*.

Daraus ergibt sich das in Abb. 11 gezeigte Bild. Jeder der unterschiedlichen Klassentypen hat spezielle Aufgaben, die für den nächsten Abschnitt bedeutsam sind. Daher werden sie nachfolgend näher erläutert. Dabei wird die jeweilige Aufgabe von einer oder mehreren Methoden bewerkstelligt, die je nach Dialekt u.U. unterschiedlich benannt sind. Daher wird zur besseren Übersicht jede Aufgabe mit einem Kürzel aus MOP und einer fortlaufenden Nummer versehen. Im weiteren Verlauf wird dann auf dieses Kürzel anstatt der Methoden und Aufgaben verwiesen.

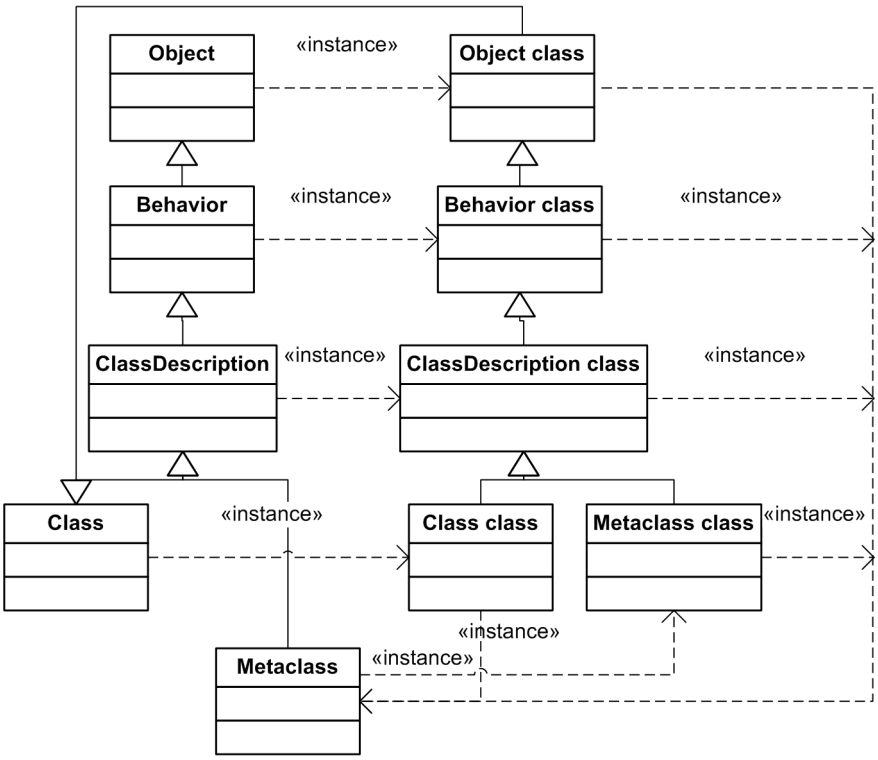

Abbildung 11: Beziehungen zwischen Objekten und Klasssen

Klasse *Behavior*

Behavior definiert den kleinstmöglichen Zustand, der für Objekte notwendig ist. Dazu gehören die Oberklasse, die Unterklassen sowie das Methodenverzeichnis. *Behavior* übernimmt folgende Aufgaben:

- Hinzufügen einer Methode (MOP1)
- Entfernen einer Methode (MOP2)
- Setzen der Oberklasse (MOP3)

Klasse *ClassDescription*

ClassDescription implementiert das gemeinsame Verhalten von Klassen und Metaklassen. Dazu gehören vor allem beschreibende Eigenschaften wie die Instanzvariablen, die Kategorisierung der Methoden, die Klassenkategorie und der Klassenname. *ClassDescription* übernimmt folgende Aufgaben:

- Hinzufügen einer Instanzvariable (MOP4)
- Entfernen einer Instanzvariable (MOP5)
- Setzen der Instanzvariablenliste (MOP6)
- Setzen der Klassenkategorie (MOP7)
- Setzen der Kategorie einer Methode (MOP8)

Klasse *Class*

Class beschreibt das Aussehen und Verhalten von Objekten. Außerdem wird zusätzliche Programmierunterstützung angeboten. *Class* übernimmt folgende Aufgaben:

- Hinzufügen einer Klassenvariable (MOP9)
- Entfernen einer Klassenvariable (MOP10)
- Erzeugen einer neuen (Unter-)Klasse (MOP11)
- Entfernen der Klasse (MOP12)
- Umbenennen der Klasse (MOP13)

Organisatoren

Die Organisation von Klassen und Methoden in Kategorien wird in Smalltalk nicht vollständig auf der Klassenhierarchieebene sondern von gesonderten Objekten der Klassen *SystemOrganizer* und *ClassOrganizer* verwirklicht. Sie übernehmen folgende Aufgaben:

- Hinzufügen einer Klassen- bzw. Methodenkategorie (MOP14 bzw. MOP15)
- Umbenennen Klassen- bzw. Methodenkategorie (MOP16 bzw. MOP17)

5.3 Verhaltenserhaltung

Vor der Betrachtung der elementaren Refaktorisierungen nach (Opd92) im nächsten Abschnitt werden zunächst zusätzliche Eigenschaften bestimmt, die der Verhaltenserhaltung in Anwesenheit von Aspektorientierung dienen. Die Erhaltung des Verhaltens vor und nach der Programmtransformation ist die grundlegende Voraussetzung jeder Refaktorisierung. Dafür sind in (Opd92) sieben Bedingungen aufgestellt worden. Sie sollen hier nicht weiter betrachtet werden.
Die Verwendung von Aspektorientierung erfordert eine neue zu erfüllende Bedingung, nämlich die Gleichwertigkeit von Verbindungspunkten zu erhalten. Das bedeutet, dass sich die Verbindungspunkte jeder Verbindungspunktauswahl sowohl vor als auch nach einer Refaktorisierung weder in ihrer Anzahl noch in ihrer Art verändern. Unter Arterhaltung ist zu verstehen, dass ein Aufrufpunkt ein Aufrufpunkt bleibt und seine Position im Programmfluss behält aber nicht notwendigerweise die gleichen Kontexteigenschaften wie z.B. Methodennamen aufweist.
Um zu gewährleisten, dass vor und nach einer Refaktorisierung die gleichen Verbindungspunkte ausgewählt werden, sind i.A. Filter die einen Verbindungspunkt erst durch die Refaktorisierung auswählen durch einen Differenzfilter einzuschränken und solche die einen Verbindungspunkt durch die Refaktorisierung nicht mehr auswählen durch einen Vereinigungsfilter zu erweitern.

Aspekte und Anreicherungen sind in AspectTalk als gewöhnliche Klassen und Methoden verwirklicht. Sie bedürfen daher keiner zusätzlichen Eigenschaften der Verhaltenserhaltung, sondern fallen unter die bereits bekannten.

5.4 Elementare Refaktorisierungen

In den folgenden Abschnitte werden die Besonderheiten dargelegt, die sich durch die Aspektorientierung beim Refaktorisieren ergeben. Dazu werden die in (Opd92) definierten elementaren Refaktorisierungen untersucht, da sich komplexere Refaktorisierungen aus ihnen

zusammensetzen und somit ebenfalls erfasst werden. Der dort begründete Katalog von Refaktorisierungen findet sich in den meisten Refaktorisierungswerkzeugen, so auch in VisualWorks, wieder und ist damit von entscheidender Relevanz. Es werden nur solche Refaktorisierungen betrachtet, die in Smalltalk anwendbar sind. Demnach entfallen die Refaktorisierungen *Ändere Typ*, *Ändere Zugriffsmodus*, *Konvertiere Instanzvariable nach Zeiger* und *Abstrahieren* bzw. *Konkretisieren einer Methode*. Allerdings werden zusätzliche smalltalkspezifische Refaktorisierungen untersucht: Erzeugung, Entfernung, Umbenennung und Änderung einer Klassen-/Methodenkategorie. Zu jeder Refaktorisierung werden folgende Angaben gemacht:

- Kurzbeschreibung
- Einfluss auf Verbindungspunktauswahlen
- Einfluss auf Aspekte
- Überwachungsmaßnahmen

Auf eine umfassende Wiedergabe aller Bedingungen, die die einzelnen Refaktorisierungen genügen müssen, damit sie verhaltenserhaltend sind, wird verzichtet. Diese können bei Bedarf in (Opd92) nachgelesen werden.

5.4.1 Erzeugung einer Programmentität

Erzeugung einer leeren Klasse

Beschreibung

Es wird eine neue Klasse *K* als direkte Unterklasse einer Klasse *O* definiert. Die Klasse enthält weder lokal definierte Methoden noch Instanz- bzw. geteilte Variablen. Sie besitzt keine Unterklassen.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die nach der Ausführung von MOP11 gebunden ist, erfasst.

Einfluss auf Verbindungspunktauswahlen

Da die Klasse weder Methoden noch Variablen definiert, gibt es keine lokalen Verbindungspunkte. Allerdings erbt sie u.U. Methoden und Instanzvariablen ihrer Oberklasse. Werden in diesen Methoden Verbindungspunkte angereichert, sind bestimmte Maßnahmen

zu ergreifen.
Erhaltende Verbindungspunktauswahlen schränken alle ihre Klassenfilter, die die neue Klasse erfassen, so ein, dass die neue Klasse ausgeschlossen wird. Dadurch entfallen Maßnahmen in Zusammenhang mit Methoden und Variablen, die der neuen Klasse eventuell nachträglich hinzugefügt werden.
Erweiternde Verbindungspunktauswahlen deren Klassenfilter die neue Klasse erfassen, führen eine erneute Filterung durch. Werden dadurch neue Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Aspekte sind zwar gewöhnliche Klassen und Anreicherungen gewöhnliche Methoden. Doch gibt es keinerlei Auswirkungen, da Anreicherungsbindungen und Instanziierungsstrategien nicht vererbt werden. Somit sind keine Maßnahmen zu ergreifen.

Erzeugung einer Instanz- bzw. Klassenvariable

Beschreibung

Einer Klasse wird eine Instanz- oder Klassenvariable hinzugefügt. Die neue Variable wird nicht referenziert.

Überwachung

Die Erzeugung einer Instanzvariablen wird durch eine Anreicherung, die nach der Ausführung von MOP4 gebunden ist, erfasst. Da eine Instanzvariable an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Erzeugung für jede der erbenden Klassen gemeldet werden.
Die Erzeugung einer Klassenvariable wird durch eine Anreicherung, die nach der Ausführung von MOP9 gebunden ist, erfasst.
Die Überwachung der Erzeugung einer allgemeinen geteilten oder Pool-Variablen hängt davon ab, wie ein Smalltalkdialekt dies umsetzt. Daher kann hier keine allgemeiner Weg angegeben werden. Die zu ergreifenden Maßnahmen bleiben aber dennoch gleich.

Einfluss auf Verbindungspunktauswahlen

Erhaltende Verbindungspunktauswahlen schränken alle ihre Zugriffsfilter, die die neue Variable erfassen, so ein, dass sie ausgeschlossen wird. Auch wenn die Variable noch nicht referenziert wird, verhindert dies eine Erfassung jeder späteren Referenzierung.

Erweiternde Verbindungspunktauswahlen müssen keine Maßnahmen treffen, selbst wenn ihre Zugriffsfilter die neue Variable erfassen, da sie nicht referenziert wird.

Einfluss auf Aspekte

Weder Instanz- noch geteilte Variablen beeinflussen Aspekte oder die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Erzeugung einer Methode

Beschreibung

Einer Klasse wird eine lokal definierte Methode hinzugefügt. Die Methode ist entweder neu oder ersetzt eine geerbte Methode.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die um die Ausführung von MOP1 herum gebunden ist, erfasst. Da eine Methode an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Erzeugung für jede der erbenden Klassen gemeldet werden.

Einfluss auf Verbindungspunktauswahlen

Erhaltende Verbindungspunktauswahlen schränken all ihre Methodenfilter, die die neue Methode auswählen, so ein, dass sie ausgeschlossen wird. Ist die Methode neu, und es somit weder sie betreffende Ausführungs- noch Aufrufpunkte aber durchaus Zugriffspunkte innerhalb der Methode gibt, verhindert dies eine Erfassung späterer Ausführungen und Aufrufe.
Erweiternde Verbindungspunktauswahlen deren Methodenfilter die Methode erfassen führen eine erneute Filterung durch. Werden dadurch neue Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Anreicherungen sind zwar gewöhnliche Methoden. Doch gibt es keinerlei Auswirkungen, da Anreicherungsbindungen nicht vererbt werden und bei der Überschreibung einer geerbten und gebundenen Anreicherung einfach die neue Methode aufgerufen wird. Somit sind keine Maßnahmen zu ergreifen.

Erzeugung einer leeren Klassen- bzw. Methodenkategorie

Beschreibung

Es wird eine neue Klassen- bzw. Methodenkategorie hinzugefügt. Die Kategorie enthält keine Elemente.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die nach der Ausführung von MOP14 (Klassenkategorie) bzw. MOP15 (Methodenkategorie) gebunden ist, erfasst.

Einfluss auf Verbindungspunktauswahlen

Erhaltende Verbindungspunktauswahlen schränken alle ihre Klassen- bzw. Methodenkategoriefilter, die die neue Kategorie erfassen, so ein, dass sie ausgeschlossen wird. Auch wenn die Kategorie noch keine Elemente enthält, verhindert dies eine Erfassung jedes späteren Elements.
Erweiternde Verbindungspunktauswahlen müssen keine Maßnahmen treffen, selbst wenn ihre Klassen- bzw. Methodenkategoriefilter die neue Kategorie erfassen, da sie keine Elemente enthält.

Einfluss auf Aspekte

Klassen- bzw. Methodenkategorien beeinflussen weder Aspekte noch die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

5.4.2 Entfernung einer Programmentität

Entfernung einer unreferenzierten Klasse

Beschreibung

Eine Klasse wird gelöscht. Sie wird nicht mehr referenziert und besitzt keine Unterklassen.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die um die Ausführung von MOP12 herum gebunden ist, erfasst.

Einfluss auf Verbindungspunktauswahlen

Da die Klasse nicht mehr referenziert wird, sind alle Verbindungspunkte die sich auf Methoden und Instanz- bzw. geteilte Variablen der Klasse beziehen nicht mehr existent oder liegen innerhalb von Methoden der Klasse selbst. Somit ist keiner dieser Verbindungspunkte mehr erreichbar. Alle Verbindungspunktauswahlen deren Klassenfilter die Klasse erfassen führen eine erneute Filterung durch um alle eventuellen Bindungen die an die verbliebenen Verbindungspunkte gebunden sind zu entfernen, damit die Verbindungspunktregister aktualisiert werden.

Einfluss auf Aspekte

Aspekte sind gewöhnliche Klassen. Handelt es sich bei der zu löschenden Klasse um einen Aspekt, der noch über gebundene Anreicherungen oder Verbindungspunktauswahlen verfügt, wird der Vorgang angehalten und dies dem Benutzer angezeigt. Dieser kann entscheiden, den Vorgang endgültig abzubrechen oder fortzufahren und mit dem Aspekt auch alle Bindungen und Verbindungspunktauswahlen zu entfernen.

Entfernung einer unreferenzierten Instanz- bzw. geteilten Variable

Beschreibung

Eine nicht mehr referenzierte Instanz- bzw. geteilte Variable wird entfernt.

Überwachung

Diese Refaktorisierung bedarf keiner Überwachung.

Einfluss auf Verbindungspunktauswahlen

Da die Variable unreferenziert ist, existieren keine Zugriffspunkte die sich auf sie beziehen. Daher sind bereits alle Bindungen an die ehemaligen Zugriffspunkte entfernt worden. Somit sind keine Maßnahmen zu ergreifen.

Einfluss auf Aspekte

Weder Instanz- noch geteilte Variablen beeinflussen Aspekte oder die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Entfernung einer Menge von unreferenzierten Methoden

Beschreibung

Ein Teil der Methoden einer Klasse wird gelöscht, vorausgesetzt dass jede zu löschende Methode nur innerhalb der zu löschenden Methoden referenziert oder durch eine geerbte Methode ersetzt wird.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die um die Ausführung von MOP2 herum gebunden ist, erfasst. Da eine Methode an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Löschung jeder Methode für jede der erbenden Klassen gemeldet werden.

Einfluss auf Verbindungspunktauswahlen

Alle Verbindungspunkte die sich auf eine der Methoden beziehen sind nicht mehr existent oder liegen innerhalb der zu löschenden Methoden selbst. Somit ist keiner dieser Verbindungspunkte mehr erreichbar. Alle Verbindungspunktauswahlen deren Methodenfilter die Methode erfassen führen eine erneute Filterung durch um alle eventuellen Bindungen die an die verbliebenen Verbindungspunkte gebunden sind zu entfernen, damit die Verbindungspunktregister aktualisiert werden.

Einfluss auf Aspekte

Anreicherungen sind gewöhnliche Methoden. Da sie in AspectTalk indirekt aufgerufen werden, kann ihre Referenzierung nicht gewöhnlich ermittelt werden. Für jede Methode die als Anreicherung noch gebunden ist, wird der Vorgang angehalten und die vorhandenen Bindungen dem Benutzer angezeigt. Dieser kann entscheiden, den Vorgang endgültig abzubrechen oder fortzufahren und mit der Anreicherung auch alle Bindungen zu entfernen.

Entfernung einer leeren Klassen- bzw. Methodenkategorie

Beschreibung

Es wird eine Klassen- bzw. Methodenkategorie entfernt, die keine Elemente mehr enthält.

Überwachung

Diese Refaktorisierung bedarf keiner Überwachung.

Einfluss auf Verbindungspunktauswahlen

Da die Kategorie keine Elemente mehr enthält, existieren keine Verbindungspunkte, die sich auf sie beziehen. Daher sind bereits alle Bindungen an die ehemaligen Verbindungspunkte entfernt worden. Somit sind keine Maßnahmen zu ergreifen.

Einfluss auf Aspekte

Klassen- bzw. Methodenkategorien beeinflussen weder Aspekte noch die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

5.4.3 Veränderung einer Programmentität

Umbenennung einer Klasse

Beschreibung

Der Name einer Klasse wird verändert. Alle Referenzen auf die Klasse werden entsprechend angepasst.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die um die Ausführung von MOP13 herum gebunden ist, erfasst.

Einfluss auf Verbindungspunktauswahlen

Durch den Namenswechsel verschwinden alle Verbindungspunkte, die sich auf den alten Klassennamen beziehen. An ihre Stelle treten analoge Verbindungspunkte, die sich auf den neuen Klassennamen beziehen.
Alle Verbindungspunktauswahlen deren Klassenfilter die Klasse nach der Namensänderung nicht mehr erfassen, müssen so erweitert werden, dass sie wieder eingeschlossen wird. Außerdem werden alle Verzeichnisse auf den neuen Namen aktualisiert.
Erhaltende Verbindungspunktauswahlen schränken all ihre Klassenfilter, die die Klasse erst durch die Namensänderung auswählen, so ein, dass sie ausgeschlossen wird.
Erweiternde Verbindungspunktauswahlen deren Klassenfilter die Klasse erst durch die Namensänderung erfassen führen eine erneute Filterung durch. Werden dadurch neue

Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Aspekte sind gewöhnliche Klassen. Allerdings hat die Umbenennung eines Aspekts keinerlei weiteren Einfluss, da er nicht über seinen Namen sondern als Klasse referenziert wird. Somit werden diese Referenzen durch das System automatisch angepasst und es sind keine Maßnahmen zu ergreifen.

Umbenennung einer Instanz- bzw. Klassenvariable

Beschreibung

Einer Instanz- bzw. Klassenvariable wird ein neuer Name zugewiesen. Alle Referenzen auf die Variable werden so angepasst, dass sie den neuen Namen verwenden.

Überwachung

Für die Umbenennung einer Instanzvariable gibt es keine spezielle Methode. Allerdings gibt es zwei Wege diese Refaktorisierung umzusetzen.
Der erste Weg ist das Hinzufügen einer Instanzvariablen mit dem neuen Namen, der Umbenennung aller Referenzen und der Entfernung der alten Instanzvariablen. Die Umbenennung der Referenzen wird erkannt, indem die Anreicherung, die an MOP1 gebunden ist, so erweitert wird, dass auf Gleichheit der Syntaxbäume unter Berücksichtigung von Variablennamensänderungen geachtet wird. Eine Anreicherung, die nach der Ausführung von MOP5 gebunden ist, meldet schließlich die Umbenennung.
Der zweite Weg besteht darin der Klasse eine neue Instanzvariablenliste zu geben. Dabei wird an der Position des alten Namens der neue Name eingesetzt und anschließend die Referenzen aktualisiert. Dieser Vorgang wird durch eine Anreicherung, die um die Ausführung von MOP6 herum gebunden ist, erfasst.
Da eine Instanzvariable an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Umbenennung für jede der erbenden Klassen gemeldet werden.

Für die Umbenennung einer Klassenvariablen gibt es lediglich einen Weg, der dem ersten Weg bei der Umbenennung einer Instanzvariablen ähnelt nur mit der Methode MOP10. Dementsprechend wird auch hier verfahren.
Die Überwachung der Umbenennung einer allgemeinen geteilten oder Pool-Variablen hängt davon ab, wie ein Smalltalkdialekt dies umsetzt. Daher kann hier keine allgemeiner Weg angegeben werden. Die zu ergreifenden Maßnahmen bleiben aber dennoch gleich.

Einfluss auf Verbindungspunktauswahlen

Durch den Namenswechsel verschwinden alle Zugriffspunkte, die sich auf den alten Variablennamen beziehen. An ihre Stelle treten analoge Zugriffspunkte, die sich auf den neuen Variablennamen beziehen.
Alle Verbindungspunktauswahlen deren Variablenfilter die Variable nach der Namensänderungen nicht mehr erfassen müssen so erweitert werden, dass sie wieder eingeschlossen wird. Außerdem werden alle Verzeichnisse auf den neuen Namen aktualisiert.
Erhaltende Verbindungspunktauswahlen schränken all ihre Variablenfilter, die die Variable erst durch die Namensänderung auswählen, so ein, dass sie ausgeschlossen wird. Da eine Instanzvariable an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Überprüfung für jede der erbenden Klassen durchgeführt werden.
Erweiternde Verbindungspunktauswahlen deren Variablenfilter die Variable erst durch die Namensänderung erfassen führen eine erneute Filterung durch. Werden dadurch neue Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Weder Instanz- noch geteilte Variablen beeinflussen Aspekte oder die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Umbenennung einer Methode

Beschreibung

Einer Methode wird ein neuer Name zugewiesen. Alle Referenzen auf die Methode werden so angepasst, dass sie den neuen Namen verwenden.

Überwachung

Es gibt keine einzelne Methode zur Methodenumbennung. Der Ablauf der Refaktorisierung besteht darin, dass eine Methode mit dem neuen Namen hinzugefügt wird, alle Referenzen aktualisiert werden und schließlich die Methode mit dem alten Namen gelöscht wird. Diese Refaktorisierung wird daher durch eine Anreicherung, die um die Ausführung von MOP2 herum gebunden ist, erfasst. Wird nämlich eine Methode für die eine gleichwertige andere Methode vorhanden ist entfernt, handelt es sich um eine Umbenennung. Zwei Methoden sind dann gleichwertig, wenn ihre abstrakten Syntaxbäume bis auf den Methodennamen gleich sind. Idealerweise sollte eine Methode zur Umbenennung in Behavior aufgenommen

werden.
Da eine Methode an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Umbennung für jede der erbenden Klassen gemeldet werden.

Einfluss auf Verbindungspunktauswahlen

Durch den Namenswechsel verschwinden alle Verbindungspunkte, die sich auf den alten Methodennamen beziehen. An ihre Stelle treten analoge Verbindungspunkte, die sich auf den neuen Methodennamen beziehen.
Alle Verbindungspunktauswahlen deren Methodenfilter die Methode nach der Namensänderung nicht mehr erfassen müssen so erweitert werden, dass sie wieder eingeschlossen wird. Außerdem werden alle Verzeichnisse auf den neuen Namen aktualisiert.
Erhaltende Verbindungspunktauswahlen schränken all ihre Methodenfilter, die die Methode erst durch die Namensänderung auswählen, so ein, dass sie ausgeschlossen wird.
Erweiternde Verbindungspunktauswahlen deren Methodenfilter die Methode erst durch die Namensänderung erfassen führen eine erneute Filterung durch. Werden dadurch neue Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Anreicherungen sind gewöhnliche Methoden. Handelt es sich bei der umzubennenden Methode um eine Anreicherung, kann der Name nicht automatisch vom System angepasst werden, da Anreicherungen in AspectTalk indirekt aufgerufen werden. Daher müssen alle Bindungen der Anreicherung aktualisiert werden.

Umbenennung einer Klassen- bzw. Methodenkategorie

Beschreibung

Einer Klassen- bzw. Methodenkategorie wird ein neuer Name zugewiesen.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die nach der Ausführung von MOP16 (Klassenkategorie) bzw. MOP17 (Methodenkategorie) gebunden ist, erfasst. Da eine Methode und somit indirekt ihre Kategorie an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Umbenennung für jede der erbenden Klassen gemeldet werden.

Einfluss auf Verbindungspunktauswahlen

Durch den Namenswechsel verschwinden keinerlei Verbindungspunkte. Allerdings können sie bei erneuten Filterungen nicht mehr erfasst werden.
Alle Verbindungspunktauswahlen deren Klassen- bzw. Methodenkategoriefilter die Kategorie nach der Namensänderung nicht mehr erfassen müssen so erweitert werden, dass sie wieder eingeschlossen wird.
Erhaltende Verbindungspunktauswahlen schränken all ihre Klassen- bzw. Methodenkategoriefilter, die die Kategorie erst durch die Namensänderung auswählen, so ein, dass sie ausgeschlossen wird.
Erweiternde Verbindungspunktauswahlen deren Klassen- bzw. Methodenkategoriefilter die Kategorie erst durch die Namensänderung erfassen führen eine erneute Filterung durch. Werden dadurch neue Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Klassen- bzw. Methodenkategorien beeinflussen weder Aspekte noch die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Hinzufügung eines Parameters zu einer Methode

Beschreibung

In Smalltalk ist dieser Vorgang äquivalent zur Umbenennung einer Methode. Die Gleichwertigkeit der alten und neuen Methode lässt sich ermitteln, da der neue Parameter in umbenannten Methode noch nicht referenziert wird und somit bis auf den Methodennamen keine Änderung vorliegt.

Entfernung eines Parameters einer Methode

Beschreibung

In Smalltalk ist dieser Vorgang äquivalent zur Umbenennung einer Methode. Die Gleichwertigkeit der alten und neuen Methode lässt sich ermitteln, da der zu entfernende Parameter in der alten Methode nicht mehr referenziert wird und somit bis auf den Methodennamen keine Änderung vorliegt.

Umordnung der Parameter einer Methode

Beschreibung

In Smalltalk ist dieser Vorgang äquivalent zur Umbenennung einer Methode. Die Gleichwertigkeit der alten und neuen Methode lässt sich ermitteln, da weder neue Parameter hinzugefügt noch entfernt werden und somit bis auf den Methodennamen keine Änderung vorliegt. Diese Refaktorisierung ist auch der Grund, warum die Gleichwertigkeit zweier Methoden anhand ihrer abstrakten Syntaxbäume getestet werden muss. Ein Vergleich des Bytecodearrays würde in diesem Fall negativ ausfallen, da die Parameter anhand ihrer Indizes angesprochen werden, die sich jedoch durch die Umordnung verändert haben.

Ersetzung von Instanzvariablenzugriffen durch Zugriffsmethoden

Beschreibung

Dieser Vorgang ist identisch mit der Ersetzung einer Anweisungsliste durch einen Methodenaufruf.

Ersetzung einer Anweisungsliste durch einen Methodenaufruf

Beschreibung

Eine Folge von Anweisungen wird durch einen Methodenaufruf ersetzt, wobei die Methode semantisch äquivalent zu der Anweisungsliste ist.

Überwachung

Für diese Refaktorisierung gibt es keine gesonderte Methode. Da hierbei eine bestehende Methode verändert und somit dem Methodenverzeichnis erneut hinzugefügt wird, wird die Refaktorisierung durch eine Anpassung der Anreicherung, die um die Ausführung von MOP1 herum gebunden ist, erfasst. Um diesen Fall zu erkennen ist eine erweiterte Analyse der alten und neuen Methode erforderlich. Hierbei werden alle Differenzen zwischen den Methoden ermittelt. Treten dabei Unterschiede auf, muss es sich dabei in der neuen Methode genau um ausschließlich einen Methodenaufruf und in der alten Methode um genau eine zusammenhängende Anweisungsfolge handeln. Schließlich müssen die abstrakten Syntaxbäume der entfernten Anweisungsfolge und des Rumpfes der aufzurufenden Methode einschließlich aller Variablennamen gleich sein.
Da eine Methode an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Ersetzung für jede der erbenden Klassen gemeldet werden.

Einfluss auf Verbindungspunktauswahlen

Diese Refaktorisierung betrifft Aufrufs- und Zugriffspunkte, die in den zu ersetzenden Anweisungen auftreten. Da sie mit den Anweisungen aus der betroffenen Methode verschwinden, müssen. Alle Verbindungspunktauswahlen die Aufrufs- und Zugriffspunkte in der zu ersetzenden Anweisungsliste auswählen müssen ihre Methodenfilter, die den Kontext bestimmen, so anpassen, dass auch die aufzurufende Methode ausgewählt wird. Da die ersetzende Methode nicht neu erstellt, sondern bereits vorhanden sein kann und später eventuell auch aus anderen Methoden aufgerufen wird, muss zusätzlich ein Abstammungsfilter eingesetzt werden. Dieser sorgt dafür, dass die Bindungen nur dann ausgeführt werden, wenn die ersetzende Methode tatsächlich aus der Methode deren Anweisungen sie ersetzt aufgerufen wird. Dies garantiert, dass die Verbindungspunkte vor und nach der Refaktorisierung sich weder in ihrer Anzahl noch in ihrem Wesen verändert haben.

Einfluss auf Aspekte

Diese Refaktorisierung hat keinen Einfluss auf die Ausführung von Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Ersetzung eines Methodenaufrufs durch die Anweisungsliste des Methodenrumpfs

Beschreibung

Ein Methodenaufruf wird durch die Anweisungsfolge der aufzurufenden Methode ersetzt.

Überwachung

Für diese Refaktorisierung gibt es keine gesonderte Methode. Da hierbei eine bestehende Methode verändert und somit dem Methodenverzeichnis erneut hinzugefügt wird, wird die Refaktorisierung durch eine nochmalige Anpassung der Anreicherung, die um die Ausführung von MOP1 herum gebunden ist, erfasst. Um diesen Fall zu erkennen ist eine erweiterte Analyse der alten und neuen Methode erforderlich. Hierbei werden alle Differenzen zwischen den Methoden ermittelt. Die Art der Differenzen ist dabei umgekehrt zur vorherigen Refaktorisierung.
Da eine Methode an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Ersetzung für jede der erbenden Klassen gemeldet werden.

Einfluss auf Verbindungspunktauswahlen

Ist der Aufruf der zu ersetzenden Methode angereichert, wird der Vorgang angehalten, da kein gleichwertiger Aufrufpunkt entstehen wird. Dies wird dem Benutzer mitgeteilt und er kann entscheiden, ob mit dem Aufruf auch all seine Bindungen entfernt werden sollen.
Ferner sind angereicherte Aufrufs- und Zugriffspunkte in der aufzurufenden Methode betroffen, da sie mit dem Aufruf verschwinden. Alle Verbindungspunktauswahlen, die diese Verbindungspunkte auswählen, müssen ihre Methodenfilter, die den Kontext bestimmen, so anpassen, dass auch die den ursprünglichen Aufruf enthaltende Methode ausgewählt wird. Dies garantiert, dass die Verbindungspunkte vor und nach der Refaktorisierung sich weder in ihrer Anzahl noch in ihrem Wesen verändert haben.

Einfluss auf Aspekte

Diese Refaktorisierung hat keinen Einfluss auf die Ausführung von Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Änderung der Oberklasse

Beschreibung

Die Oberklasse einer Klasse wird verändert. Alle bisher geerbten Methoden und Instanzvariablen werden auch von der neuen Oberklasse vererbt.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die um die Ausführung von MOP3 herum gebunden ist, erfasst.

Einfluss auf Verbindungspunktauswahlen

Durch die Veränderung der Oberklasse gehen keine Verbindungspunkte verloren, da alle lokal definierten und geerbten Methoden sowie Instanz- und Klassenvariablen erhalten bleiben.
Allerdings ist es möglich, dass die Klasse von ihrer Oberklasse Methoden und Instanzvariablen erbt, die ihr vorher nicht vererbt worden sind. Für diese Elemente wird für die Klasse und all ihren erbenden Unterklassen die Erzeugung einer Methode bzw. Instanzvariable gemeldet, damit die Verbindungspunktauswahlen entsprechend verfahren können.

Einfluss auf Aspekte

Aspekte sind zwar gewöhnliche Klassen und Anreicherungen gewöhnliche Methoden.

Doch gibt es keinerlei Auswirkungen, da Anreicherungsbindungen und Instanziierungsstrategien nicht vererbt werden und geerbte Anreicherungen auch weiterhin geerbt werden. Somit sind keine Maßnahmen zu ergreifen.

5.4.4 Verschiebung einer Programmentität

Verschiebung einer Instanzvariable in die Oberklasse

Beschreibung

Eine Instanzvariable wird aus allen definierenden Unterklassen in die gemeinsame Oberklasse verschoben. Die Oberklasse hat die Instanzvariable bisher nicht enthalten.

Überwachung

Für diese Refaktorisierung gibt es keine eigene Methode. Ihr Ablauf besteht darin, dass zunächst die Instanzvariable aus allen definierenden Unterklassen entfernt wird und sie anschließend der neuen Oberklasse hinzugefügt wird.
Das Hinzufügen einer Instanzvariable wird bereits durch eine Anreicherung überwacht. Diese muss lediglich so angepasst werden, dass überprüft wird, ob die hinzugefügte Instanzvariable bereits referenziert wird. In diesem Fall wird das Hinzufügen der Instanzvariable nur für die Oberklasse aber für keine der Unterklassen gemeldet. Dadurch wird verhindert, dass erhaltende Verbindungspunktauswahlen die bisher erfassten Instanzvariablen fälschlicherweise ausschließen.

Einfluss auf Verbindungspunktauswahlen

Durch das Verschieben der Instanzvariable geht kein Zugriffspunkt verloren, da lediglich ihre Definition verschoben wird. Die betroffenen Unterklassen verfügen weiterhin über die Instanzvariable. Da die Oberklasse die Instanzvariable bisher nicht enthalten hat, gibt es keine neuen Zugriffspunkte. Allerdings müssen erhaltende Verbindungspunktauswahlen ihre Variablenfilter, die die neue Instanzvariable in der Oberklasse auswählen, so anpasssen, dass sie ausgeschlossen wird. Dies verhindert eine Erfassung jeder späteren Referenzierung.

Einfluss auf Aspekte

Weder Instanz- noch geteilte Variablen beeinflussen Aspekte oder die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Verschiebung einer Instanzvariable in die Unterklassen

Beschreibung

Eine Instanzvariable wird aus einer Klasse in all ihre direkten Unterklassen verschoben. Die ursprünglich definierende Klasse enthält keine Referenz mehr auf die Instanzvariable.

Überwachung

Für diese Refaktorisierung gibt es keine eigene Methode. Ihr Ablauf besteht darin, dass zunächst die Instanzvariable aus der Oberklasse entfernt und sie anschließend allen direkten Unterklassen hinzugefügt wird.
Das Hinzufügen einer Instanzvariable wird bereits durch eine Anreicherung überwacht. Da sie so arbeitet, dass das Hinzufügen einer Instanzvariable nicht gemeldet wird, wenn diese bereits referenziert wird, ist auch in diesem Fall gewährleistet, dass erhaltende Verbindungspunktauswahlen die bisher erfassten Instanzvariablen nicht fälschlicherweise ausschließen.

Einfluss auf Verbindungspunktauswahlen

Durch das Verschieben der Instanzvariablen geht kein Zugriffspunkt verloren, da lediglich ihre Definition verschoben wird. Die betroffenen Unterklassen verfügen weiterhin über die Instanzvariable. Da in der Oberklasse die Instanzvariable nicht referenziert wird, gibt es keine Zugriffspunkte.

Einfluss auf Aspekte

Weder Instanz- noch geteilte Variablen beeinflussen Aspekte oder die Ausführung ihrer Anreicherungen. Somit sind keine Maßnahmen zu ergreifen.

Verschiebung einer Klasse in eine andere Kategorie

Beschreibung

Einer Klasse wird eine andere Kategorie zugewiesen.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die um die Ausführung von MOP7 herum gebunden ist, erfasst.

Einfluss auf Verbindungspunktauswahlen

Durch das Verschieben der Klasse in einer andere Kategorie geht kein Verbindungspunkt verloren. Allerdings können sie bei erneuten Filterungen nicht mehr erfasst werden.
Alle Verbindungspunktauswahlen deren Klassenfilter die Klasse nach der Verschiebung nicht mehr erfassen müssen so erweitert werden, dass sie wieder eingeschlossen wird. Es ist wichtig den Klassen- und nicht den Klassenkategoriefilter zu ändern, da sonst eventuell neue Klassen ausgewählt werden.
Erweiternde Verbindungspunktauswahlen deren Klassenfilter die Klasse erst durch die Verschiebung erfassen führen eine erneute Filterung durch. Werden dadurch neue Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Aspekte sind zwar gewöhnliche Klassen. Doch gibt es keinerlei Auswirkungen, da keine Eigenschaft eines Aspekts von seiner Kategorie abhängt. Somit sind keine Maßnahmen zu ergreifen.

Verschiebung einer Methode in eine andere Kategorie

Beschreibung

Einer Methode wird eine andere Kategorie zugewiesen.

Überwachung

Diese Refaktorisierung wird durch eine Anreicherung, die um die Ausführung von MOP8 herum gebunden ist, erfasst. Da eine Methode an alle Unterklassen der sie definierenden Klasse vererbt wird, muss die Verschiebung für jede der erbenden Klassen gemeldet werden.

Einfluss auf Verbindungspunktauswahlen

Durch das Verschieben der Methode in einer andere Kategorie geht kein Verbindungspunkt verloren. Allerdings können sie bei erneuten Filterungen nicht mehr erfasst werden.
Alle Verbindungspunktauswahlen deren Methodenfilter die Methode nach der Verschiebung nicht mehr erfassen müssen so erweitert werden, dass sie wieder eingeschlossen wird. Es ist wichtig den Methoden- und nicht den Methodenkategoriefilter zu ändern, da sonst eventuell neue Methoden ausgewählt werden.
Erweiternde Verbindungspunktauswahlen deren Methodenfilter die Methode erst durch die

Verschiebung erfassen führen eine erneute Filterung durch. Werden dadurch neue Verbindungspunkte ausgewählt, werden alle Anreicherungen die bisher an die Auswahl gebunden worden sind auch an die neuen Verbindungspunkte gebunden.

Einfluss auf Aspekte

Anreicherungen sind zwar gewöhnliche Methoden. Doch gibt es keinerlei Auswirkungen, da Anreicherungsbindungen nicht von der Methodenkategorie abhängen. Somit sind keine Maßnahmen zu ergreifen.

6 Fazit

6.1 Zusammenfassung

In dieser Arbeit ist der Entwurf und die Implementation von AspectTalk behandelt worden. AspectTalk ist eine Mehrzweckerweiterung Smalltalks um aspektorientierte Programmierung im Bereich der Verhaltensanpassung. Aspektorientierte Programmierung ermöglicht eine natürliche, problemspezifische Modularisierung (KIL$^+$97) von Programmen die mit den gegebenen Mitteln und Möglichkeiten einer „gewöhnlichen“ Programmiersprache nicht erreichbar ist. Die Besonderheit von AspectTalk liegt in dem dynamischen, symmetrischen Ansatz und der automatischen Anpassung an Refaktorisierungen.
Zunächst sind die dafür notwendigen Grundlagen, nämlich Smalltalk und die aspektorientierte Programmierung, untersucht worden. Von Smalltalk sind nur die wichtigsten Eigenschaften betrachtet worden, während die Konzepte der aspektorientierten Programmierung ausführlicher erläutert worden sind. Aus diesen beiden Grundlagenbereichen sind die Anforderungen an AspectTalk ermittelt worden.
Die Erfüllung dieser Anforderungen ist anhand der Umsetzung der aspektorientierten Programmierung in AspectTalk gezeigt worden. Dies umfasst die Ausprägung und Verwendung der Grundelemente des Aspekts, der Anreicherung, der Verbindungspunkte und deren Auswahlen. Grundlegend ist dabei der Verzicht auf gesonderte Aspektklassen, jede Klasse kann also als Aspekt auftreten, und die Benutzung normaler Methoden als Anreicherungen. Ebenso sind Verbindungspunkte und ihre Auswahlen gewöhnliche Objekte, wodurch eigene Arten hinzugefügt werden können. Programmelemente werden durch frei definierbare Filter ausgewählt, wobei der vorgegebene Weg die Prägnanz der in AspectJ verwendeten Namensmuster durch zusätzliche Angaben bereichert. Die Darstellung ist mittels zweier konkreter Anwendungsbeispiele abgeschlossen und veranschaulicht worden.
Wie die einzelnen Eigenschaften konkret verwirklicht worden sind, ist zum Abschluss mit der Betrachtung der wichtigsten technischen Gesichtspunkte AspectTalks gezeigt worden. Hauptaugenmerk ist dabei auf den Webvorgang und die Anreicherungsanwendung gelegt worden. Ausführlich sind die für die Anpassung an Refaktorisierungen nötigen Vorgänge betrachtet worden. Dabei ist jeder der auf Smalltalk anwendbaren elementaren Refaktorisierungen nach (Opd92) auf ihr Erkennen, ihren Einfluss und notwendige Maßnahmen untersucht worden, mit dem Ergebnis, dass auf jede Refaktorisierung angemessen reagiert werden kann.

6.2 Ausblick

Während der Realisierung von AspectTalk sind einige Gesichtspunkte aufgekommen, deren Untersuchung und Umsetzung sich vorteilhaft für das Gesamtsystem erweisen können. Diese werden im folgenden angerissen.

Refaktorisierung

Neben der Anpassung der klassischen Refaktorisierungen um Aspektorientierung gibt es bereits neue, rein aspektorientierte Refaktorisierungen (MF05; HMK05; RL; Mon05). Sie befassen sich hauptsächlich mit der Behandlung bestehender Konstrukte der Aspektorientierung und ihrer automatischen Erzeugung. Während eine Vielzahl dieser Maßnahmen in AspectTalk entfallen, da es ohne neue Konstrukte arbeitet, gibt es dennoch einige Refaktorisierungen um die AspectTalk erweitert werden kann.

Aspektfindung

In engem Zusammenhang mit aspektorientierten Refaktorisierungen steht das „Schürfen“ von Aspekten (engl. *aspect mining*). Hierbei handelt es sich um Verfahren zur Ermittlung von Aspekten, Anreicherungen, Bindungen und Verbindungspunkten (AX07). Dazu werden hauptsächlich einfache Mustererkennung auf Quelltextbasis (HK) oder Überwachung von Objektverwendungen zur Laufzeit (BDET05) eingesetzt. Gerade Smalltalk mit seinen vielfältigen Möglichkeiten der Reflexion bietet hier ein großes Potential.

Anreicherungsanalyse

Neben den direkten Lese- und Schreiboperationen des Verbindungspunktobjekts wäre es nützlich den Anwender eigene Operationen angeben zu lassen, die dann bei der Anreicherungsanalyse berücksichtigt werden. Dies ermöglicht eine weitergehende Untersuchung.
Außerdem wäre eine automatische Konfliktaufzeigung wünschenswert, sodass der Anwender Konflikte nicht selbst ermitteln muss.

Herkunftsfilter

Um zeitsparend auch die unsicheren Methodenaufrufe, d.h. die deren Empfänger erst zur Laufzeit bekannt ist, effizient zu erfassen, ist es denkbar, diese Aufrufe mit einer Anreicherung zu versehen. Diese Anreicherung kann dann zur Laufzeit den Typ des Empfängers ermitteln und den Filter veranlassen, die sich ergebende Methode zu analysieren, sofern dies nicht bereits geschehen ist. Zwar verursacht dies zur Laufzeit einen zusätzlichen Aufwand. Doch ist dieser geringer im Vergleich zur Berücksichtigung aller Klassen die die betreffende Methode implementieren.

A Screenshots

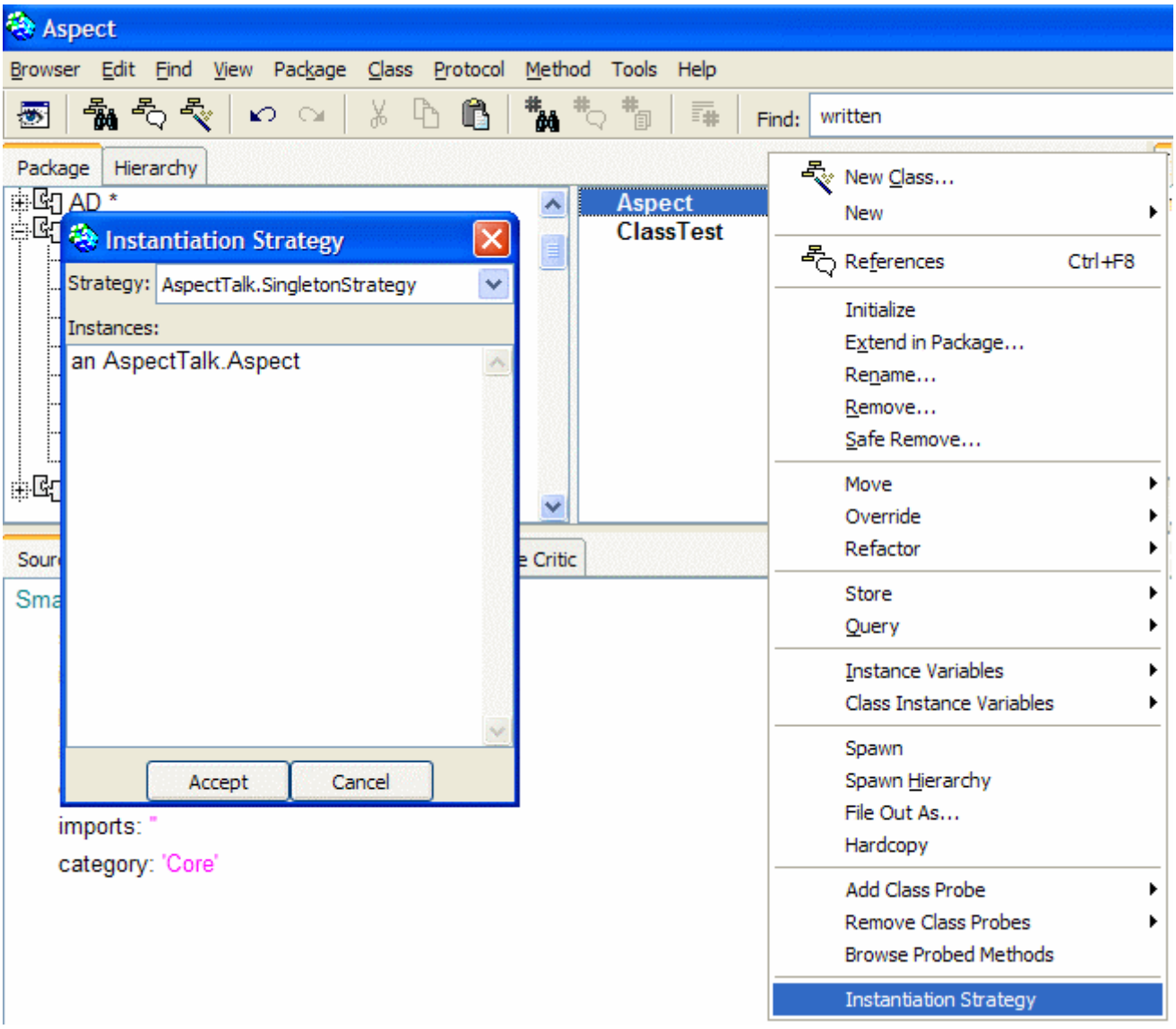

Abbildung 12: Menüpunkt und Dialog der Instanziierungsstrategie

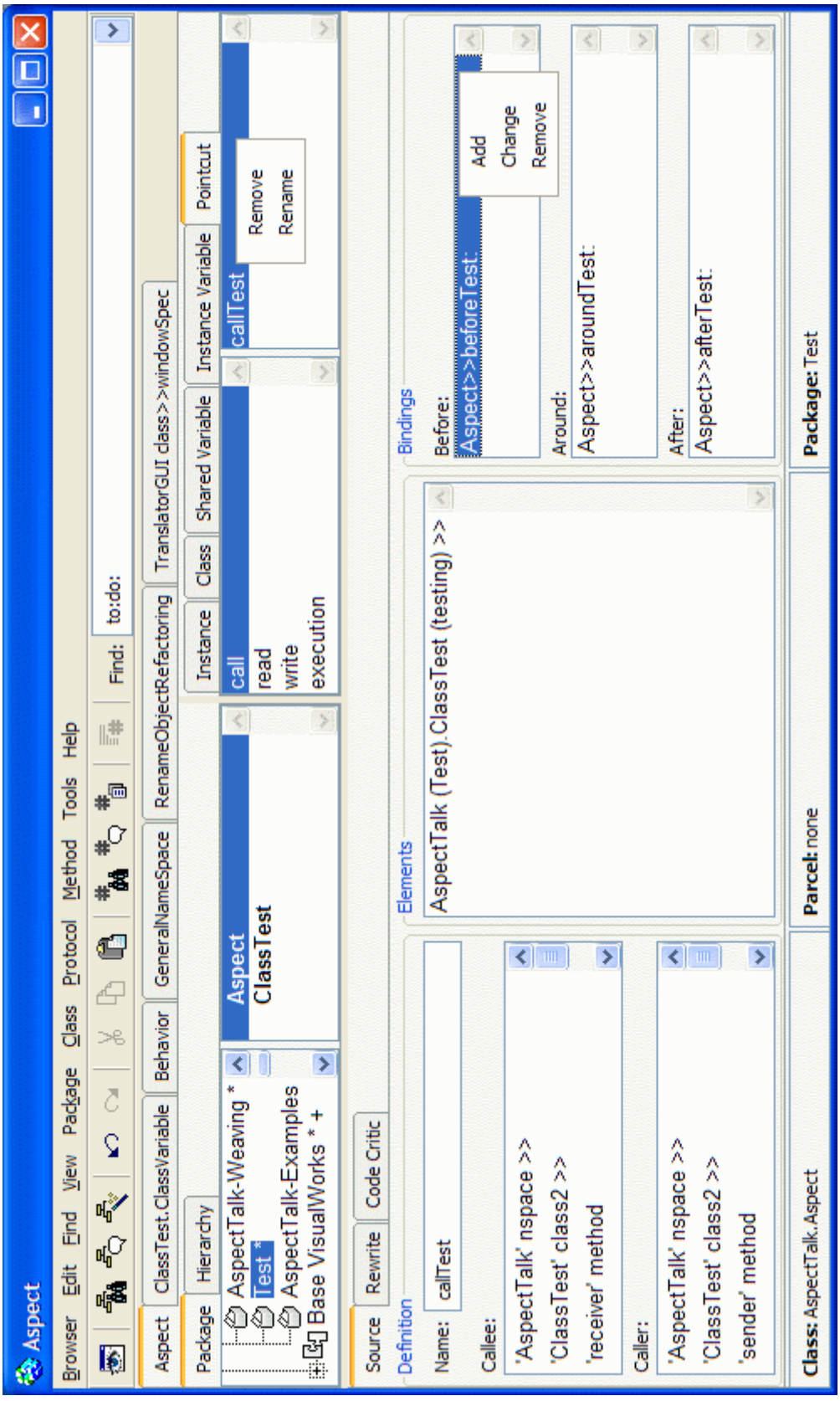

Abbildung 13: Werkzeug für Verbindungspunktauswahlen

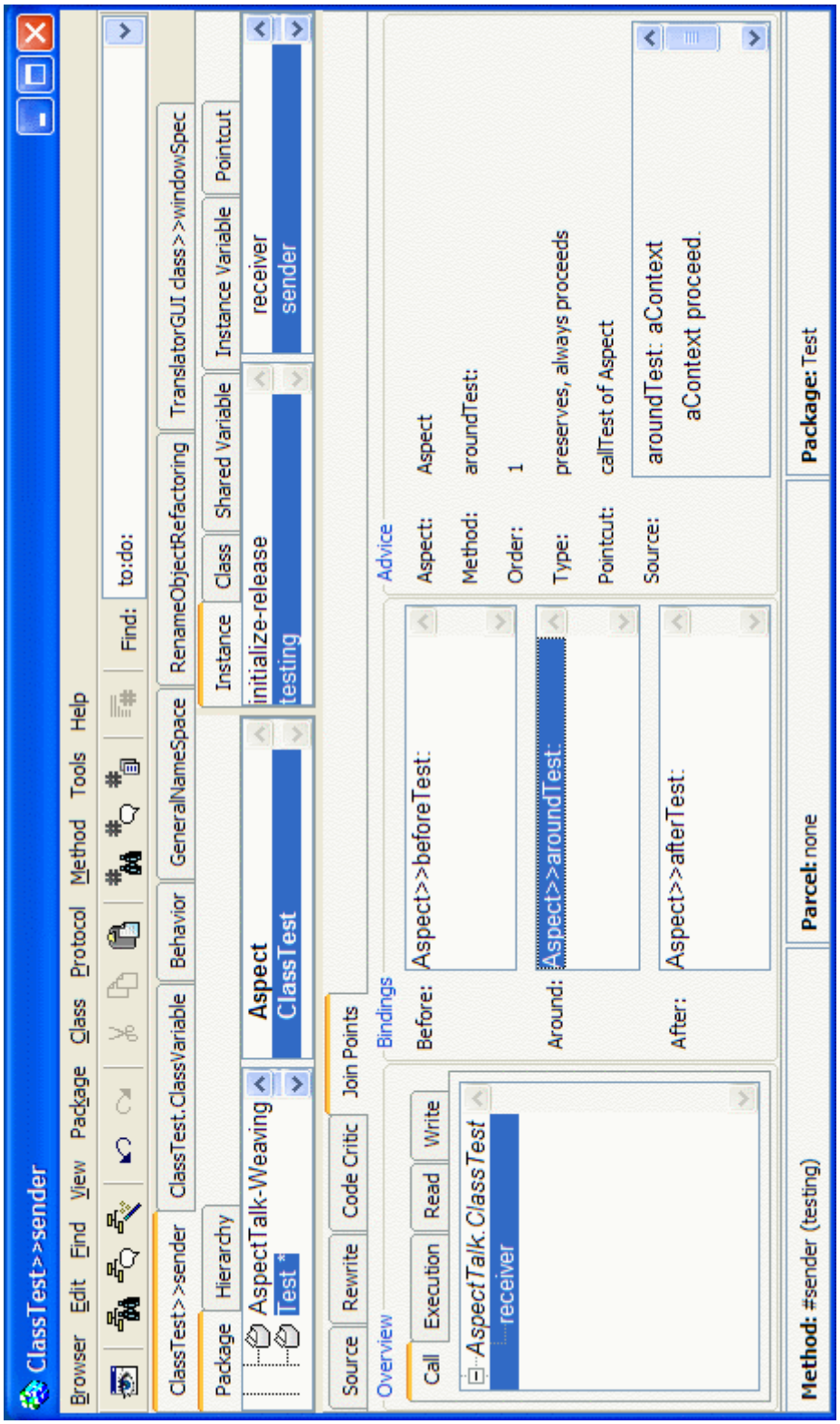

Abbildung 14: Werkzeug für Verbindungspunkte

B Ausgewählte Klassendiagramme

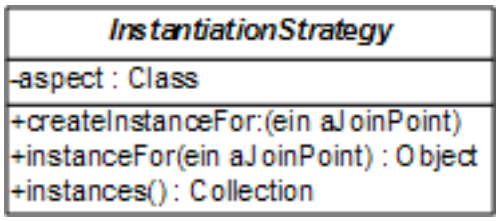

Abbildung 15: Schnittstelle von Instanziierungsstrategien

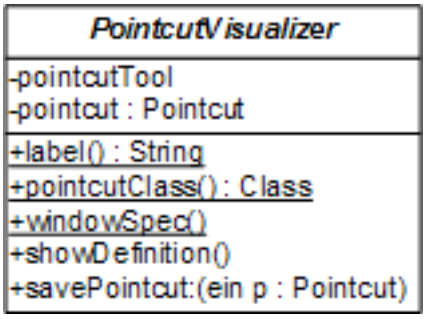

Abbildung 16: Visualisierer einer Verbindungspunktauswahl

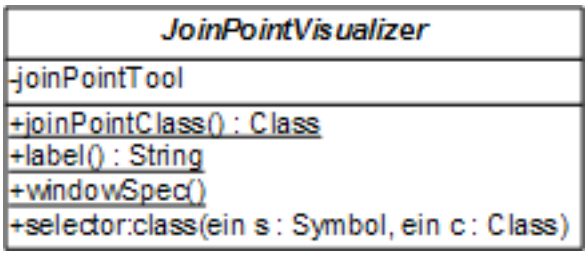

Abbildung 17: Visualisierer eines Verbindungspunkts

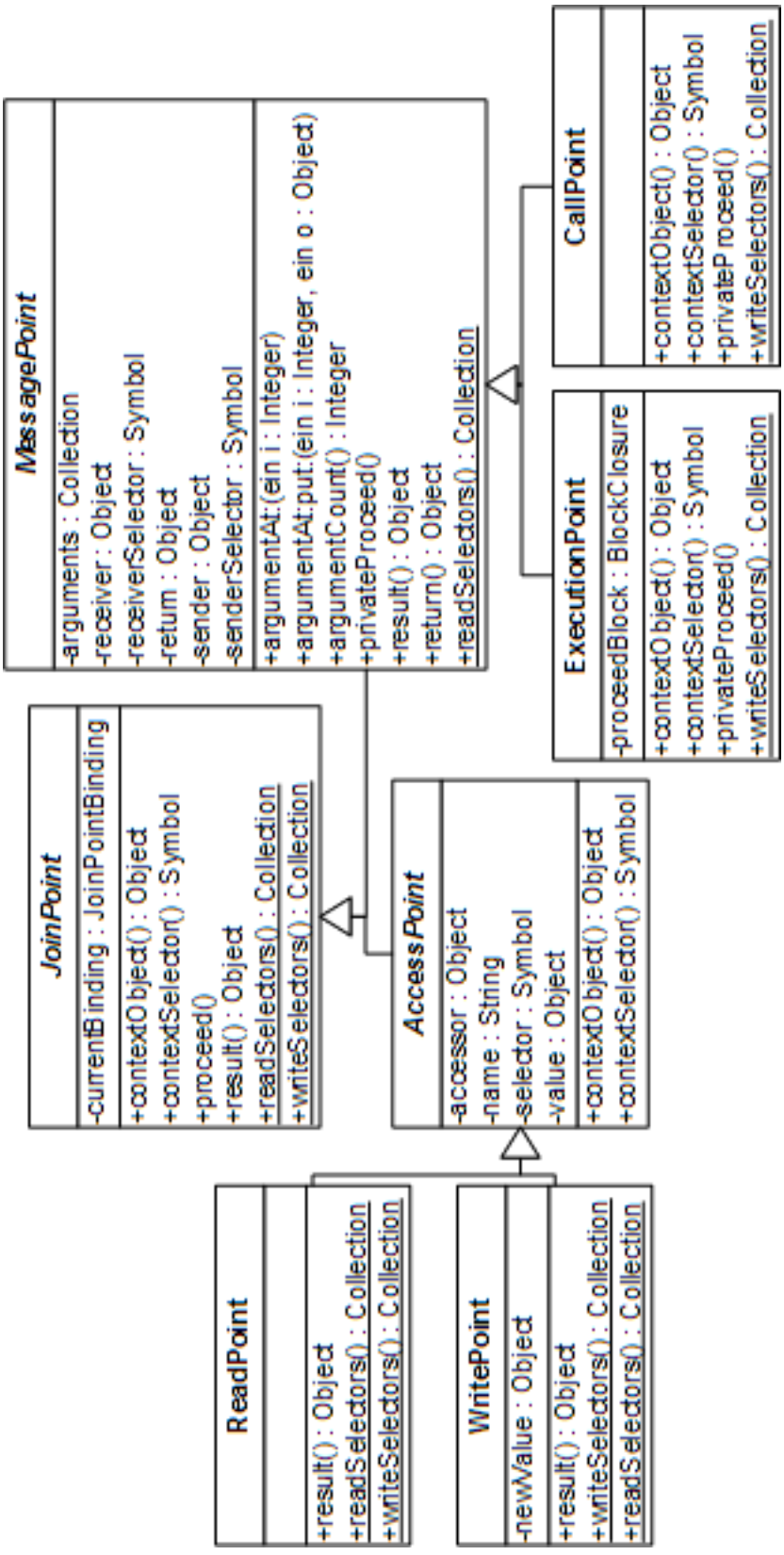

Abbildung 18: Hierarchie aller Verbindungspunkte

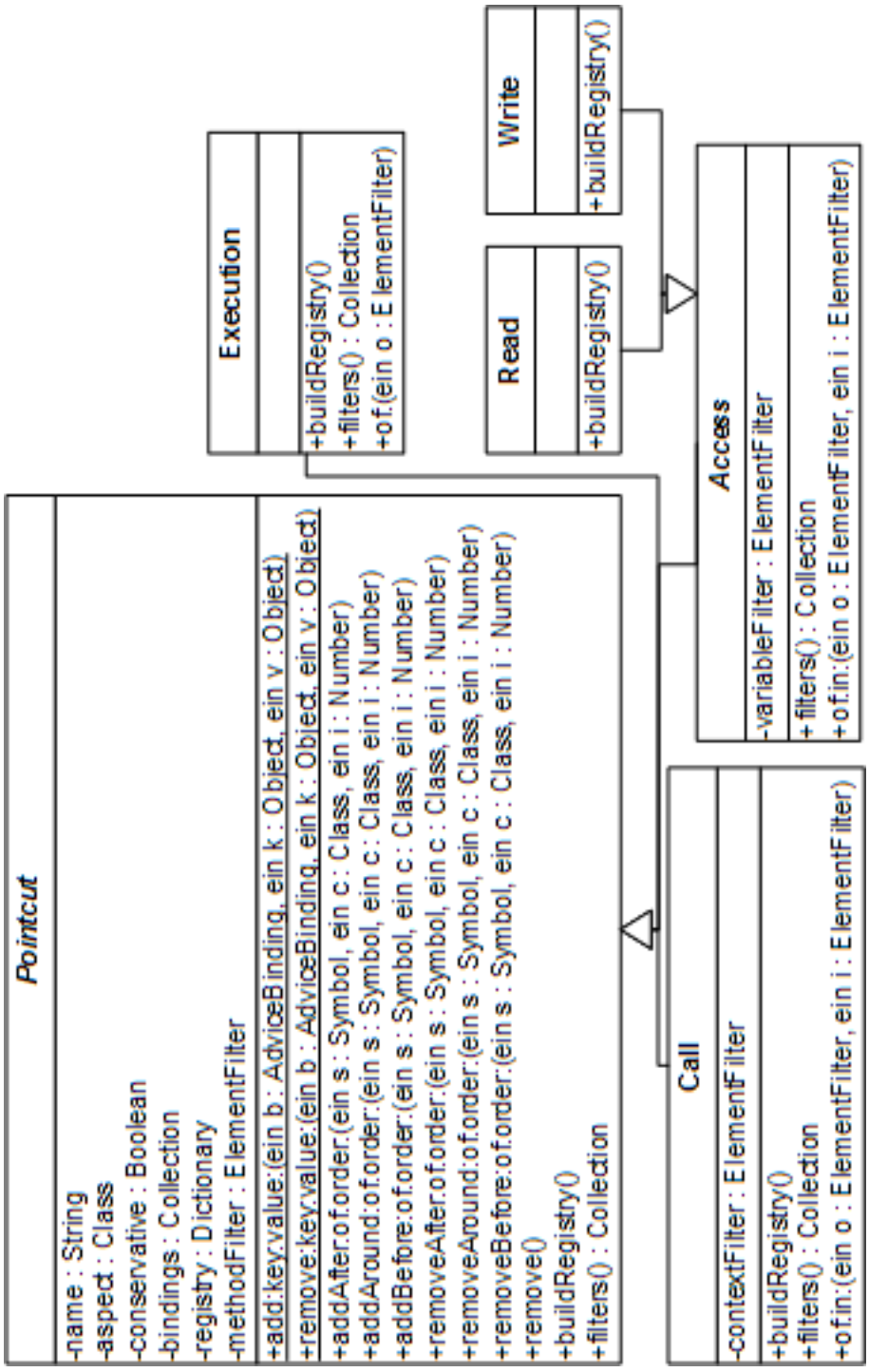

Abbildung 19: Hierarchie aller Verbindungspunktauswahlen

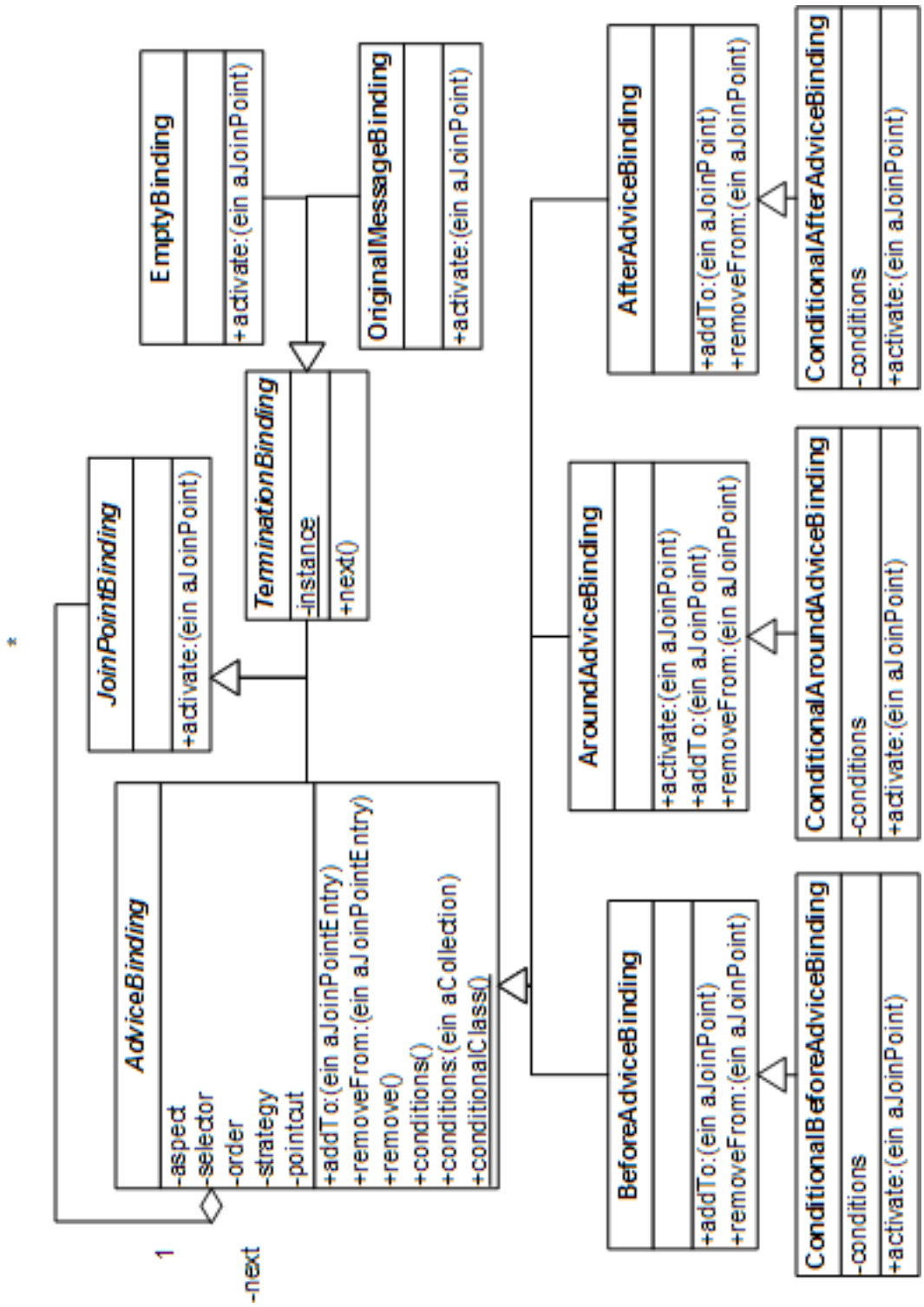

Abbildung 20: Hierarchie aller Verbindungspunktbindungen

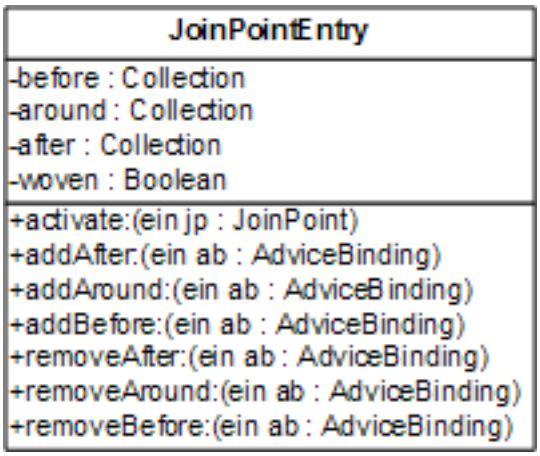

Abbildung 21: Verbindungspunkteintrag

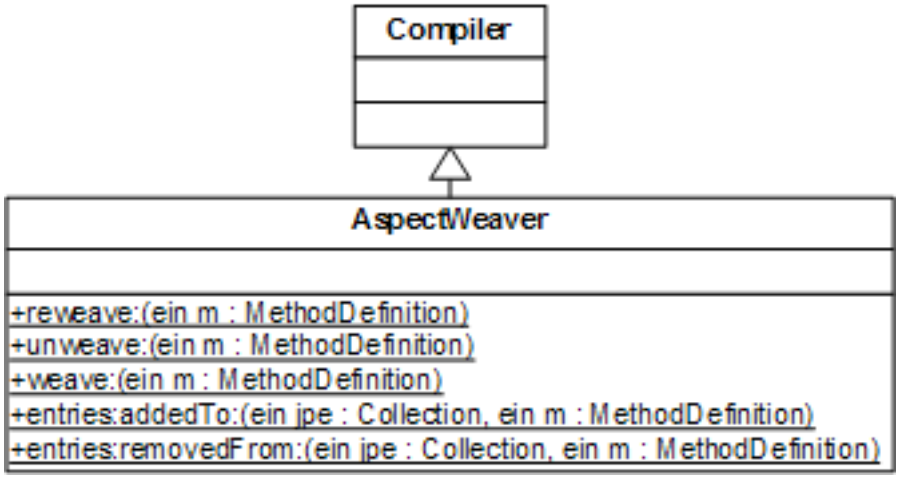

Abbildung 22: Aspektweber

C Quelltextbeispiele

Interest

```
Smalltalk.AspectTalk defineClass: #Interest
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'observers '
   classInstanceVariableNames: 'pointcut '
   imports: ''
   category: 'AspectTalk-Aspect'

"AspectTalk.Interest class methods"

pointcut
   ^self subclassResponsibility

add: observer to: subject
   ((InstantiationStrategy of: self)
      instanceAt: subject) addObserver: observer

getPointcut
   ^pointcut isNil
      ifTrue: [pointcut := self pointcut]
      ifFalse: [pointcut]

remove: observer from: subject
   ((InstantiationStrategy of: self)
      instanceAt: subject) removeObserver: observer

install
   PerContextObjectStrategy aspect: self.
   self getPointcut addAfter: #notifyObservers: of: self

uninstall
   self getPointcut removeAfter: #notifyObservers: of: self

"AspectTalk.Interest methods"
```

```
notifyObservers: aJoinPoint
   observers do: [
      :observer |
      self notify: observer about: aJoinPoint in: aJoinPoint contextObject
   ]

notify: observer about: aJoinPoint in: subject
   self subclassResponsibility

addObserver: anObject
   observers add: anObject

removeObserver: anObject
   observers remove: anObject
```

Synchronizer

```
Smalltalk.AspectTalk defineClass: #Synchronizer
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'current lock '
   classInstanceVariableNames: ''
   imports: ''
   category: 'AspectTalk-Aspect'

"AspectTalk.Synchronizer class methods"

new
   ^super new initialize

install
   PerContextObjectStrategy aspect: self.

   ((Execution name: 'methodsToSynchronize' aspect: self)
      of: '*' class2 » '<synchronized>' pragma)
         addAround: #synchronize: of: self
```

```
uninstall
   (Execution name: 'methodsToSynchronize' aspect: self)
      removeAround: #synchronize: of: self

"AspectTalk.Synchronizer methods"

initialize
   lock := Semaphore forMutualExclusion

synchronize: anExecutionPoint
   current == Processor activeProcess
      ifTrue: [
         anExecutionPoint proceed
      ]
      ifFalse: [
         lock critical: [
         anExecutionPoint proceed.
         current := nil
      ].
   ]
```

D CD-ROM

Die dieser Arbeit beliegende CD-ROM hat folgenden Inhalt:

Verzeichnis	Inhalt
Bachelorarbeit	Diese Arbeit als PDF-Dokument
AspectTalk	Der Quelltext des Rahmenwerks

Tabelle 8: Inhalt der CD-ROM

Glossar

Anreicherung: Aktion, die die Struktur oder das Verhalten eines Programms ändert.

Aspekt: Sprachelement zur Implementation eines Querschnittsanliegens.

Aspektorientierte Programmierung: Paradigma der modularen Umsetzung von Querschnittsanliegen. Man unterscheidet zwischen statischer, d.h. nur zur Kompilierzeit anwendbarer, und dynamischer, d.h. auch während der Laufzeit anwendbarer, Aspektorientierung.

Aspektweber: Programm, dass für die Anwendung von Anreicherungen sorgt.

Querschnittsanliegen: Mit den gegebenen Mitteln einer Sprache nicht modular umsetzbares Anliegen.

Verbindungspunkt: Durch Anreicherungen anpassbare Struktur- und Verhaltenselemente eines Programms.

Verbindungspunktauswahl: Mechanismus zur Angabe einer Menge bestimmter Verbindungspunkte.

Literatur

ACK ALTMAN, Rubén ; CYMENT, Alan ; KICILLOF, Nicolás: *On the need for setpoints*

AId ALDRICH, Jonathan: *Open Modules: Modular Reasoning about Advice*

Alw02 ALWIS, Brian S.: *Aspects of Incremental Programming*. 2002

AX07 ANBALAGAN, Prasanth ; XIE, Tao: Automated Inference of Pointcuts in Aspect-Oriented Refactoring. In: *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007

BDET05 BRUNTINK, Magiel ; DEURSEN, Arie van ; ENGELEN, Remco van ; TOURWE, Tom: On the Use of Clone Detection for Identifying Crosscutting Concern Code. In: *IEEE Trans. Softw. Eng.* (2005)

BFJR98 BRANT, John ; FOOTE, Brian ; JOHNSON, Ralph E. ; ROBERTS, Donald: Wrappers to the Rescue. In: *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, 1998

Böh BÖHM, Oliver L.: *SW-Archäologie mit AOP (Praxisbericht)*

Böl98 BÖLLERT, Kai: *Aspect-Oriented Programming Case Study: System Management Application*. 1998

Bru05 BRUNS, Andreas: *Integration von Usability-Methoden in den agilen Entwicklungsprozess am Beispiel der Erfassung und Auswertung von Tracingdaten*, Hochschule für angewandte Wissenschaften Hamburg, Diplomarbeit, 2005

CBE+00 CONSTANTINIDES, Constantinos A. ; BADER, Atef ; ELRAD, Tzilla H. ; FAYAD, Mohamed E. ; NETINANT, P.: Designing an Aspect-Oriented Framework in an Object-Oriented Environment. In: *ACM Comput. Surv.* (2000)

Cin CINCOM (Hrsg.): *Application Developer's Guide*. Cincom

CL CLIFTON, Curtis ; LEAVENS, Gary T.: *Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning*

CSS CONSTANTINIDES, Constantinos ; SKOTINIOTIS, Therapon ; STOERZER, Maximilian: *AOP considered harmful*

Duc99 DUCASSE, Stephane: Evaluating Message Passing Control Techniques in Smalltalk. In: *Journal of Object-Oriented Programming (JOOP)* (1999)

FF00 FILMAN, Robert E. ; FRIEDMAN, Daniel P.: Aspect-Oriented Programming is Quantification and Obliviousness. 2000. – Forschungsbericht

Fil01 FILMAN, Robert E.: What Is Aspect-Oriented Programming, Revisited. 2001. – Forschungsbericht

GCKL GARCIA, Alessandro ; CHAVEZ, Christina ; KULESZA, Uirá ; LUCENA, Carlos: *The Role Aspect Pattern*

GKCL GARCIA, Alessandro ; KULESZA, Uirá ; CHAVEZ, Christina ; LUCENA, Carlos: *The Interaction Aspect Pattern*

Gol84 GOLDBERG, Adele: *Smalltalk - The Interactive Programming Environment*. Addison-Wesley, 1984

GR83 GOLDBERG, Adele ; ROBSON, David: *Smalltalk-80 The Language and its Implementation*. Addison-Wesley, 1983

Hir03 HIRSCHFELD, Robert: AspectS - Aspect-Oriented Programming with Squeak. In: *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, 2003

HK HANNEMANN, Jan ; KICZALES, Gregor: Overcoming the Prevalent Decomposition in Legacy Code.

HMK05 HANNEMANN, Jan ; MURPHY, Gail C. ; KICZALES, Gregor: Role-Based Refactoring of Crosscutting Concerns. In: *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, 2005

KGBM KELLENS, Andy ; GYBELS, Kris ; BRICHAU, Johan ; MENS, Kim: *A Model-driven Pointcut Language for More Robust Pointcuts*

KIL+97 KICZALES, Gregor ; IRWIN, John ; LAMPING, John ; LOINGTIER, Jean-Marc ; LOPES, Christina V. ; MAEDA, Chris ; MENDHEKAR, Anurag: Aspect-oriented Programming / Xerox PARC. 1997 (SPL97-008 P9710042). – Forschungsbericht

KLM+97 KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Cristina V. ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect-oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997

KM KICZALES, Gregor ; MEZINI, Mira: *Separation of Concerns with Procedures, Annotations, Advice and Pointcuts*

Lau03 LAU, Martin: *Aspektorientierte Programmierung und interne Softwarequalität: Eine Untersuchung am Beispiel von AspectJ*, Hochschule für angewandte Wissenschaften Hamburg, Diplomarbeit, 2003

MF05 MONTEIRO, Miguel P. ; FERNANDES, João M.: Towards a Catalog of Aspect-Oriented Refactorings. In: *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, 2005

Mon05 MONTEIRO, Miguel Jorge Tavares P.: *Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts*, Universidade do Minho, Diss., 2005

Opd92 OPDYKE, William F.: *Refactoring object-oriented frameworks*, University of Illinois at Urbana-Champaign, Diss., 1992

OT99 OSSHER, Harold ; TARR, Peri: Multi-dimensional Separation of Concerns in Hyperspace / IBM T.J. Watson Research Center. 1999 (RC 21452(96717)16APR99). – Forschungsbericht

Riv RIVARD, Fred: *Smalltalk: a Reflective Language*

RL RURA, Shimon ; LERNER, Barbara: *A Basis for AspectJ Refactoring*

RS05 RAJAN, Hridesh ; SULLIVAN, Kevin J.: Classpects: Unifying Aspect- and Object-Oriented Language Design. In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005

SGC02 SULLIVAN, Kevin ; GU, Lin ; CAI, Yuanfang: Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ. In: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, 2002

SGS+05 SULLIVAN, Kevin ; GRISWOLD, William G. ; SONG, Yuanyuan ; CAI, Yuanfang ; SHONLE, Macneil ; TEWARI, Nishit ; RAJAN, Hridesh: Information Hiding Interfaces for Aspect-Oriented Design. In: *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005

Ste06 STEIMANN, Friedrich: The Paradoxical Success of Aspect-Oriented Programming. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 24. Oktober 2007 Rick Eisermann