Universität Hamburg
Fachbereich Informatik

Diplomarbeit

Dynamische Komponenten in einer objektorientierten Programmiersprache
—
Modell, sprachliche Umsetzung und Visualisierung

Betreuer:
Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
TU Hamburg-Harburg

Prof. Dr. Rüdiger Valk
Theoretische Grundlagen der Informatik
Fachbereich Informatik
Universität Hamburg

vorgelegt von:
Axel Wienberg
Hinzeweg 9
21075 Hamburg
Tel.: 040 - 76751840

Hamburg, den 30. März 1999

Typographie

In dieser Arbeit werden unterschiedliche Schriftarten verwendet, um die Lesbarkeit zu verbessern. Die Bedeutung der einzelnen Schriftarten ist in der folgenden Tabelle wiedergegeben:

Schriftart	**Bedeutung**	**Beispiel**
Emphasized	englischer Begriff oder Definition oder Zitat aus Programmtext	*class precedence list*
Sans Serif	Name einer Programmiersprache oder anderen formalen Sprache	Tycoon-2 HTML, UML
`Typewriter`	TL-2-Klassennamen	`SendValue`

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Problemstellung 2

1.2. Struktur der Arbeit 2

2. Das Modell der Dynamischen Komponenten **5**

2.1. Granularitätsebenen in der Systemanalyse 5

2.2. Statische und Dynamische Ebene 7

2.3. Softwarekomponenten 8

2.3.1. Statische und Dynamische Komponenten 8

2.3.2. Lokalität 9

2.3.3. Interaktion 11

2.3.4. Migration 13

2.3.5. Komposition 13

2.4. Aggregation 14

2.4.1. Aggregation in der objektorienterten Analyse 14

2.4.2. Aggregation im objektorientierten Entwurf 15

2.4.3. Komponentenbindung als Aggregation 16

2.4.4. Rekursive Aggregation 18

3. Verwandte programmiersprachliche Konzepte **21**

3.1. Bindungen in existierenden Programmiersprachen 21

3.1.1. Strukturierte Variablen in Pascal 22

3.1.2. C 26

3.1.3. Sprachen ohne Aggregation 26

3.1.4. BETA 27
3.1.5. Eiffel 29
3.1.6. Standard ML 31
3.1.7. Aggregation in einem OODBMS 33
3.1.8. Syntaktische Aspekte 34
3.2. Gruppierung von Objekten 35
3.2.1. Emerald und Verteilung 35
3.2.2. DOWL und lokale Variablen 36
3.2.3. Java und Persistenz 37
3.2.4. Eiffel und Prozessoren 39
3.3. Kapselung und Schutz vor Mehrfachbenennung 40
3.3.1. Inseln und Brücken 41
3.3.2. Einzige Zeiger 42
3.3.3. Ballons 44
3.3.4. Flexibler Schutz vor Mehrfachbenennung 45
3.4. Folgerungen für die Umsetzung dynamischer Komponenten 46

4. Dynamische Komponenten in der Programmiersprache Tycoon-2 **49**
4.1. Überblick über Tycoon-2 49
4.2. Überblick der Spracherweiterung 51
4.3. Erweiterungen der Grammatik 57
4.4. Der erweiterte abstrakte Syntaxbaum 58
4.5. Zusätzliche Regeln für die Typüberprüfung 61
4.5.1. Subsumption 62
4.5.2. Wert- und Typbezeichner 62
4.5.3. Signaturen 63
4.5.4. Subtypisierung auf Schnittstellen 63
4.5.5. Block-Ausdruck 64
4.5.6. Bindungs-Ausdruck 64
4.5.7. Methodenaufruf 64

4.5.8. Funktionsabstraktionen 65
4.5.9. Zuweisung 65
4.5.10. Methodendefinition 65
4.6. Objektmodell 65
4.7. Anpassung der Standardbibliothek 68
4.7.1. Methoden in der Klasse `Object` 68
4.7.1.1. drop 68
4.7.1.2. fetch 69
4.7.1.3. superComponent 69
4.7.1.4. copy 70
4.7.1.5. doesNotUnderstand, perform 70
4.7.1.6. if@, try@ 72
4.7.2. Behälterklassen für Komponenten 73

5. Dynamische Komponenten im Tycoon-2-System **75**
5.1. Lexikalische und Syntaktische Analyse 75
5.2. Abstrakter Syntaxbaum 75
5.3. Typüberprüfung 77
5.4. Objektspeicher 77
5.4.1. Verweis auf die Superkomponente 77
5.4.2. Unabhängigkeit der Lebenszeiten 78
5.4.3. Repräsentation von Komponentenbindungen 78
5.4.4. Ausrichtung 79
5.4.5. Lokalität 79
5.5. Virtuelle Maschine und Codeerzeugung 80
5.5.1. Zugriffsarten 81
5.5.2. Endrekursion 82
5.5.3. Ausnahmen 82
5.5.4. Entkommende Variablen 83

6. Anwendung in der Programmvisualisierung **84**

6.1. Programmvisualisierung 84

6.2. Reduktion der präsentierten Datenmenge 85

6.2.1. Quantitative Zusammenfassung 86

6.2.2. Gruppierung dynamischer Phänomene anhand von statischer Information 86

6.2.3. Vergröberung der Aufrufhierarchie 87

6.2.4. Vergröberung Dynamischer Komponenten 87

6.3. Visualisierung unter Verwendung Dynamischer Komponenten 89

6.3.1. Technische Randbedingungen 89

6.3.2. Visualisierung des Objektspeichers 90

6.3.3. Interaktion zwischen dynamischen Komponenten 92

6.3.4. Zugriff auf Interaktionsinformation 93

7. Schluß **95**

7.1. Ergebnisse 95

7.2. Bewertung 96

7.3. Ausblick 98

A. Erweiterte Grammatik von Tycoon-2 **103**

A.1. Werte 104

A.2. Typen 105

A.3. Klassen und Methoden 105

A.4. Signaturen 106

Literaturverzeichnis **107**

1. Einleitung

Zur Beschreibung großer objektorientierter Systeme werden vier wesentliche Abstraktionsmechanismen eingesetzt: Klassifikation, Generalisierung, Assoziation und Aggregation [Brodie 84]. Aggregation faßt Objekte als Bestandteile eines übergeordneten Objekts zu einem Ganzen zusammen, während Klassifikation Objekte anhand gemeinsamer Eigenschaften einordnet. Durch Generalisierung werden gemeinsame Eigenschaften verschiedener spezieller Unterklassen in einer Oberklasse ausgedrückt. Assoziation stellt Beziehungen zwischen Objekten verschiedener Klassen her.

Im Laufe der Entwicklung zu immer größeren, verteilten, persistenten und nebenläufigen Systemen wird Aggregation zunehmend wichtig, da die durch Aggregation gegebene hierarchische Zerlegung in konzeptuelle Einheiten die Modellierung erleichtert. Durch die Aspekte der Verteilung und der daraus folgenden Nebenläufigkeit fällt ein verstärktes Augenmerk auf Interaktion in Form von Nachrichtenaustausch.

Die Stärke objektorientierter Programmiersprachen besteht darin, daß die Konzepte Klassifikation, Generalisierung und Interaktion durch Klassen, Vererbung und Nachrichten direkt auf Sprachebene ausgedrückt werden können, so daß sie ohne Bruch von der Entwurfsebene übertragbar sind. Auch Assoziation findet sich in Form von Referenzen und allgemeiner Behälter-Klassen wieder. So erstaunt es, daß ein analoger Schritt für Aggregation in heutigen Programmiersprachen nicht oder nur stark eingeschränkt vorliegt.

Vererbung hat sich auch durch den praktischen Vorteil der Wiederverwendung von Quelltext bewährt. Aggregation hat in der allgemeinsten Form keine vergleichbaren Implementationsvorteile zu bieten: Vorteile werden traditionell nur gewonnen durch die Verknüpfung mit Fragen der Lebenszeit von Subkomponenten, durch die Unterscheidung in änderbare und nichtänderbare Referenzen und die Unterscheidung in Wert-Kopie vs. Referenz (C++, BETA, Eiffel). Die mit dieser Verknüpfung einhergehende konzeptuelle Komplexität erschwert aber die Programmierung, weshalb in Sprachen wie Smalltalk und Java auf Aggregation zwischen Objekten vollständig verzichtet wird.

Es fällt auf, daß moderne objektorientierte Programmiersprachen (Java, Sather) Generalisierung nicht mehr ausschließlich durch Vererbung der Implementation, sondern durch eine eigene Schnittstellen-Heterarchie ausdrücken. Der Implementationsvorteil der Wiederverwendung von Quelltext wird hier vom Vorteil einer klaren Umsetzung des Analyse- bzw. Entwurfsmodells getrennt.

Entsprechend wird in dieser Arbeit Aggregation in einer Programmiersprache unabhängig von den traditionellen Implementationsvorteilen betrachtet. Dadurch wird der Bruch zwischen Entwurfs- und Implementationsebene vermindert.

Der verwendete Aggregationsbegriff gibt Anlaß zur Definition von *dynamischen Komponenten*. Eine dynamische Komponente faßt alle von einem Objekt aggregierten Objekte zu einer Einheit zusammen. Dynamische Komponenten drücken Lokalität aus; damit bieten sie ein Modellierungsmittel für physische oder logische Objekträume, als Abstraktion von verteilten, kommunizierenden Systemen. Von außerhalb kann nur mit Hilfe von Nachrichten Einfluß auf ein solches System genommen werden, und das Verhalten des Systems äußert sich nur in den Nachrichten, die das System verlassen. Von der Kommunikation innerhalb einer dynamischen Komponente kann daher abstrahiert werden.

Anders als ein Modul oder eine Softwarekomponente, die eine Zusammenfassung von statischem Programmcode darstellen, faßt eine dynamische Komponente zustandsbehaftete Objekte zusammen. Dynamische Komponenten bilden so ein dynamisches Gegenstück zu (statischen) Softwarekomponenten.

In der Modellierung objektorientierter Systeme werden Assoziationen durch Aggregation nicht eingeschränkt. Das bedeutet, daß Assoziationen auch über die Grenzen von dynamischen Komponenten hinweg existieren können. Die Möglichkeit, flexible Assoziationsstrukturen zwischen beliebigen Objekten aufzubauen, ist eine Stärke des objektorientierten Ansatzes, weswegen dynamische Komponenten nicht zur Modellierung von Kapselung verwendet werden sollen.

1.1. Problemstellung

Das Modell der dynamischen Komponenten soll im Kontext der objektorientierten Modellierung entwickelt und als Erweiterung der Programmiersprache Tycoon-2 umgesetzt werden. Die Spracherweiterung ist im Tycoon-2-System zu implementieren, um die Machbarkeit zu zeigen und die Grundlage für weitere Untersuchungen zu schaffen. Der Nutzen der Spracherweiterung soll beispielhaft im Rahmen der Programmvisualisierung demonstriert werden.

Um die Spracherweiterung einordnen zu können, sollen existierende programmiersprachliche Konzepte auf ihre Tauglichkeit für die Umsetzung von dynamischen Komponenten untersucht werden. Die Ergebnisse der Untersuchung fließen in den Entwurf der Spracherweiterung ein.

1.2. Struktur der Arbeit

Im anschließenden Kapitel 2 wird der Begriff der dynamischen Komponente als Konzept zur hierarchischen Strukturierung des Zustands eines objektorientierten Systems

hergeleitet. Im Zusammenhang mit Aggregation ergibt sich eine spezielle Beziehung zwischen Objekten, aus der sich die dynamische Komponentenstruktur konstruieren läßt. Unter dem Namen *Komponentenbindung* wird diese Beziehung im folgenden Kapitel 3 in existierenden Programmiersprachen gesucht, um zu einer programmiersprachlichen Umsetzung von dynamischen Komponenten zu gelangen.

Die gefundenen Ansätze werden in Kapitel 4 eingesetzt, um die Programmiersprache Tycoon-2 so zu erweitern, daß dynamische Komponenten im Ausführungszustand eines Tycoon-2-Programms erkennbar sind. Kapitel 5 beschreibt die Implementierung der Spracherweiterung im Tycoon-2-System.

Zur Programmlaufzeit werden dynamische Komponenten durch das in Kapitel 6 vorgestellte Programmvisualisierungswerkzeug sichtbar gemacht. Dynamische Komponenten werden sowohl für die Visualisierung des Systemzustandes als auch für die überblicksartige Visualisierung der Systemausführung eingesetzt. Die Kommunikation zwischen dynamischen Komponenten wird durch automatisch generierte UML-Sequenzdiagramme [Fowler, Scott 97] dargestellt.

Diese Arbeit findet ihren Abschluß im Kapitel 7 mit einer Zusammenfassung und einer Bewertung des entstandenen Modells und Systems.

2. Das Modell der Dynamischen Komponenten

In diesem Kapitel wird der Begriff der dynamischen Komponente als Mittel zur hierarchischen Strukturierung des Zustandes eines objektorientierten Systems entwickelt.

Zunächst wird in Abschnitt 2.1 gezeigt, wie eine auf allen Detailebenen einheitliche Notation eine hierarchische Systembeschreibung erlaubt, die mit wählbarer Granularität betrachtet werden kann.

Um den Begriff des Systemzustandes von dem der statischen Systemstruktur zu trennen, muß, wie in Abschnitt 2.2 erläutert, zwischen dynamischer und statischer Ebene unterschieden werden. Trotz der Wiederverwendung statischer Komponenten auf verschiedenen Granularitätsebenen bilden ihre Exemplare auf der dynamischen Ebene eine hierarchische Struktur.

Dies führt in Abschnitt 2.3 zum Begriff der dynamischen Komponente als dynamischem Gegenstück zur statischen Softwarekomponente. Die Definition dynamischer Komponenten erlaubt im Kontext von Softwarekomponenten eine Präzisierung der Begriffe Lokalität, Interaktion, Migration und Komposition.

Als Abschluß des Kapitels wird die Beziehung eines Objekts zu seinen Subkomponenten in Abschnitt 2.4 als spezielle Art der Aggregation klassifiziert, was eine Grundlage für die programmiersprachliche Umsetzung des Konzepts schafft.

2.1. Granularitätsebenen in der Systemanalyse

In der Systemtheorie nach [Forrester 72] ist ein System definiert als eine Menge miteinander in Beziehung stehender Komponenten, die zu einem gemeinsamen Zweck interagieren. Die Komponenten eines offenen Systems interagieren sowohl untereinander als auch mit der Umgebung. Auf einer detaillierteren Betrachtungsebene kann auch jede Komponente eines Systems wieder als System aufgefaßt werden, was zu einer Hierarchie der Systeme führt.

Überträgt man diesen Ansatz auf die objektorientierte Modellierung, so steht zur Beschreibung der internen Struktur eines Objekts die volle Mächtigkeit der objektorientierten Modellierung zur Verfügung. Dazu gehört auch, daß Anzahl und Beziehungen der

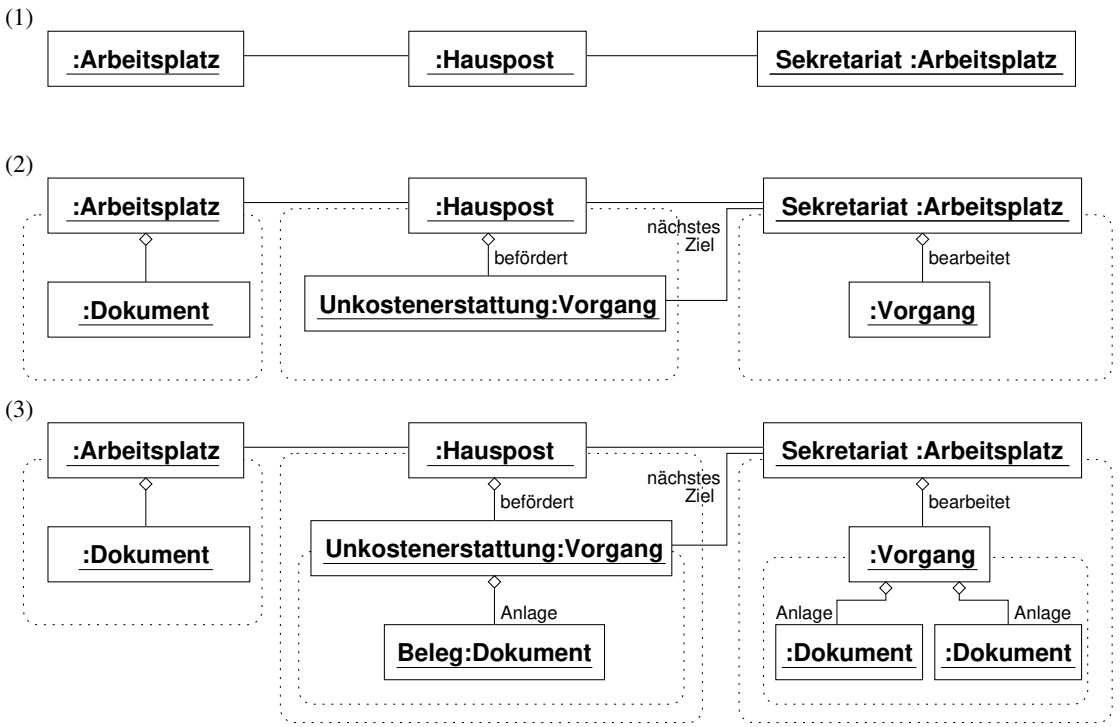

Abbildung 2.1.: Schrittweise Verfeinerung eines Büromodells

internen Komponenten eines Objektes sich im Verlauf der Zeit im Rahmen der statischen Invarianten ändern können.

Abbildung 2.1 zeigt, wie das Modell eines Bürosystems schrittweise verfeinert wird. Als Notation wird eine leicht erweiterte Variante des UML-Objektdiagramms verwendet: Die durch die Verfeinerung eines Objekts eingeführten Teilobjekte werden durch eine gestrichelte Grenze zusammengefaßt. Die Einheitlichkeit der Notation auf allen Granularitätsebenen ermöglicht es, Objekte verschiedener Granularitätsebenen in einem gemeinsamen Diagramm darzustellen.

Auf der gröbsten Ebene (1) wird dargestellt, daß die *Hauspost* die Verbindung zwischen zwei *Arbeitsplätzen* herstellt. Bei weiterer Verfeinerung (2) werden *Vorgänge* sichtbar, wovon einer an einem Arbeitsplatz in Bearbeitung ist, während der andere gerade von der Post befördert wird. An einem Arbeitsplatz wird gerade ein *Dokument* erstellt. Bei weiterer Verfeinerung (3) wird sichtbar, daß ein Dokument auch als Anlage in einem Vorgang enthalten sein kann. Das Klassendiagramm zu diesem Beispiel wird in Abbildung 2.2 gezeigt.

Alle Beziehungen in dem Büro-Modell sind über die Zeit änderbar; so können z.B. einem Vorgang Dokumente hinzugefügt werden, ein Vorgang kann von der Hauspost von einem Arbeitsplatz zu einem anderen befördert werden, und im Verlaufe der Bearbeitung kann ein Vorgang verschiedene nächste Ziele ansteuern.

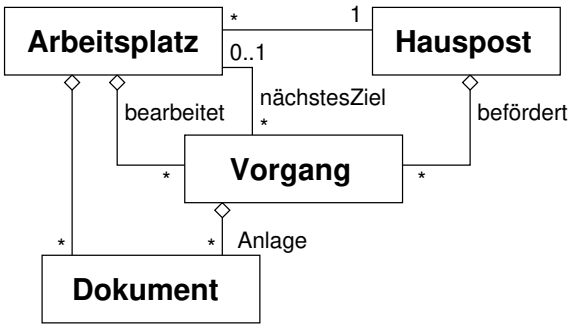

Abbildung 2.2.: Klassendiagramm des Büromodells

2.2. Statische und Dynamische Ebene

In der objektorientierten Modellierung wird klar zwischen der Struktur auf der dynamischen und auf der statischen Ebene getrennt. Ein konkreter, vom Beobachtungszeitpunkt abhängiger Systemzustand wird der dynamischen Ebene zugeordnet, während die Invarianten in der dynamischen Struktur des Systems durch die statische Struktur beschrieben werden. Diese Unterteilung läßt sich auch auf Interaktion anwenden: Interaktion findet zwischen Objekten der dynamischen Ebene statt, entspricht aber Regeln, die auf der statischen Ebene festgelegt sind.

Objektorientierte Modellierungssprachen wie die in dieser Arbeit verwendete Unified Modelling Language (UML) [Fowler, Scott 97] benutzen verschiedene Diagramme für die jeweiligen Ebenen: Die auf der statischen Ebene eingesetzten Diagrammarten heißen in UML *Klassendiagramm*, *Zustands-* und *Aktivitätsdiagramm*, und beschreiben die Invarianten in der Objektstruktur sowie die möglichen Interaktionen. Auf der dynamischen Ebene werden *Objektdiagramme* zur Darstellung eines konkreten Systemzustandes und *Sequenz-* und *Kollaborationsdiagramme* zur Darstellung konkreter Interaktionen verwendet.

Die in dieser Arbeit verwendeten Begriffe auf dynamischer und statischer Ebene sind einander in Tabelle 2.1 gegenübergestellt.

Beim Vergleich des Objektdiagramms des Büro-Beispiels in Abbildung 2.1 mit dem Klassendiagramm in Abbildung 2.2 fällt auf, daß auf der statischen Ebene die Klasse `Vorgang` sowohl von der Klasse `Hauspost` als auch von der Klasse `Arbeitsplatz` aggregiert wird. Jedes Exemplar der Klasse `Vorgang` läßt sich auf der dynamischen Ebene jedoch eindeutig entweder einem übergeordneten Exemplar der Klasse `Hauspost` oder der Klasse `Vorgang` zuordnen. Auf der dynamischen Ebene entsteht durch die schrittweise Verfeinerung eine Hierarchie der Objekte, der auf der statischen Ebene i.A. jedoch keine Hierarchie von Klassen gegenübersteht.

statisch	dynamisch
Klasse	Objekt
Assoziation Aggregation	Bindung
Algorithmus	Ausführung Interaktion
Methode	Aktivierung
Methodensignatur	Nachricht

Tabelle 2.1.: Phänomene der statischen Ebene und ihre Ausprägung auf der dynamischen Ebene

2.3. Softwarekomponenten

Die Softwaretechnik beschäftigt sich hauptsächlich mit der statischen Ebene, entsprechend ihrem Ansatz, die Produktion von Software, d.h. statischer Programmtexte zu behandeln. Zur Zeit in der Diskussion ist der Begriff der Komponenten-Software: Die statische Struktur eines Softwaresystems wird aus vorgefertigten, Softwarekomponenten genannten Versatzstücken zusammengesetzt. Das Hauptproblem besteht darin, unabhängig entwickelten Softwarekomponenten die Zusammenarbeit zu ermöglichen, etwa durch standardisierte Sprachen und Konzepte zur Spezifikation von Schnittstellen, aber auch durch Standards, die das binäre Format der Interaktion festlegen.

Im folgenden wird die Unterscheidung zwischen statischer und dynamischer Ebene und die Unterteilung in Granularitätsebenen im Kontext der Softwarekomponenten betrachtet.

2.3.1. Statische und Dynamische Komponenten

Eine Softwarekomponente kann zur Erfüllung ihres Zwecks wiederum auf Softwarekomponenten basieren. Diese Beziehung führt im allgemeinen nicht zu einer Hierarchie, denn eine generische Softwarekomponente kann von verschiedenen anderen eingesetzt werden. So können zum Beispiel zwei Softwarekomponenten, wovon die eine eine Terminverwaltung und die andere eine Finanzbuchhaltung implementiert, auf dasselbe Datenbankverwaltungssystem (DBMS) zurückgreifen (Abbildung 2.3).

Man kann aber davon ausgehen, daß eine konkrete Terminverwaltung mit einer anderen Datenbank als eine konkrete Finanzbuchhaltung arbeiten wird, und daß die Finanzbuchhaltungen verschiedener Organisationen ebenfalls auf verschiedenen Ausprägungen des Datenschemas arbeiten. Wird ein Exemplar einer Softwarekomponente als System aufgefaßt, so findet sich auf der dynamischen Ebene die eingangs erwähnte Hierarchie der Systeme wieder. Das Exemplar einer Softwarekomponente soll als *dynamische Komponente* bezeichnet werden, während die Softwarekomponente *statische Komponente* heißt.

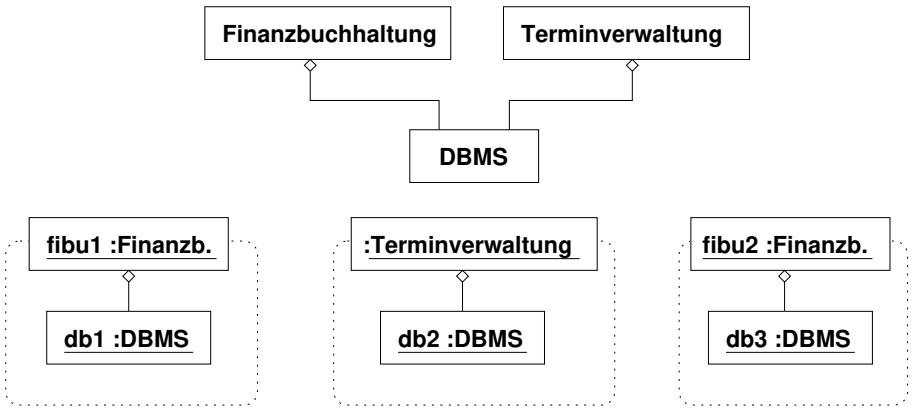

Abbildung 2.3.: Hierarchie auf dynamischer, aber nicht auf statischer Ebene

Eine Komponente einer dynamischen Komponente wird im folgenden *Subkomponente* genannt.

Wie von [Szyperski 98] angemerkt, wird in der Diskussion über Softwarekomponenten die statische und die dynamische Ebene oft vermischt. Da in vielen Systemen nur eine einzige Ausprägung einer Softwarekomponente vorkommt, scheint die begriffliche Trennung zwischen statischer und dynamischer Komponente dem Anwender oft nicht nötig. Spätestens dann aber, wenn der begriffliche Rahmen auf mehrere Systeme erweitert wird (etwa aus der Perspektive des Produzenten einer Softwarekomponente, oder wenn verschiedene Organisationen fusionieren) ist die Unterscheidung unvermeidlich.

2.3.2. Lokalität

Szyperski [Szyperski 98] bezeichnet Softwarekomponenten als Einheit der Lokalität. Lokalität ist dabei definiert als die Zuordnung zu einem bestimmten Rechnerknoten in einem verteilten System.

Was bedeutet es aber, Einheit der Lokalität zu sein? Da eine Softwarekomponente ein Phänomen der statischen Ebene ist, kann man von Lokalität nur in Bezug auf eine dynamische Repräsentation (eine *Kopie*) des Programmcodes sprechen. Kopien einer Softwarekomponente sollen nach der in dieser Arbeit und bei Szyperski verwendeten Definition nicht unterscheidbar sein, insofern kann man nicht von *dem* Knoten sprechen, an dem sich eine Softwarekomponente befindet. Ich deute die Aussage so, daß Kopien immer vollständig angelegt werden, daß eine Softwarekomponente in einem Rechnerknoten also ganz oder gar nicht repräsentiert wird. Dann ist aber immer noch nicht klar, ob die von einer Softwarekomponente verwendeten Softwarekomponenten als Bestandteile der Kopie betrachtet werden, also ebenfalls lokal vorhanden sein müssen, und insbesondere bleibt offen, auf welchem Knoten ein Exemplar der Softwarekomponente angelegt wird.

Die Einheit der Lokalität könnte auch so gedeutet werden, daß eine Softwarekomponente

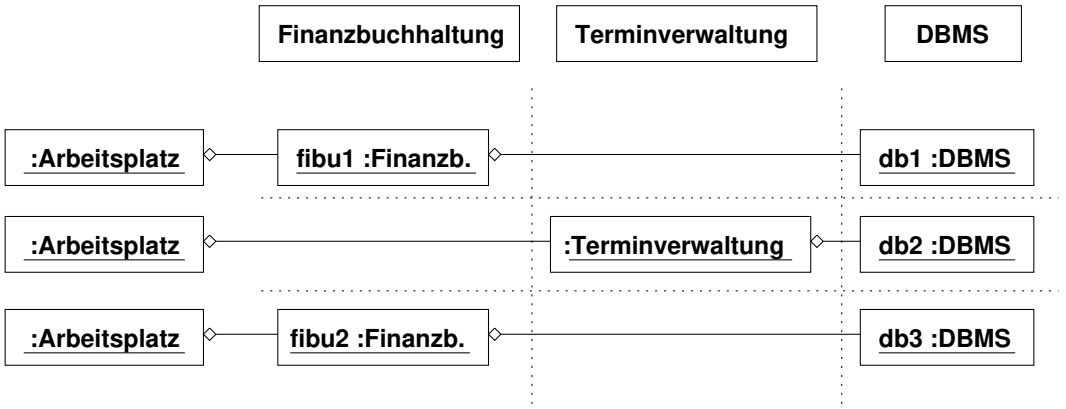

Abbildung 2.4.: Gruppierung von Exemplaren nach statischen Gesichtspunkten (vertikal) und dynamischen Gesichtspunkten (horizontal).

nur auf einem einzigen Rechnerknoten installiert und nicht an weitere Knoten übermittelt oder kopiert wird. Dadurch wäre es nur auf diesem Knoten möglich, Exemplare der Komponente zu erzeugen und ablaufen zu lassen. Jede Interaktion mit so einem Exemplar muß sich daher über das Netzwerk an den Rechnerknoten wenden. Da die entfernte Kommunikation in verteilten Systemen aber um Größenordnungen langsamer als lokale Kommunikation ist, empfiehlt es sich aus Effizienzgründen, dynamische Komponenten nicht nach ihrer statischen Struktur, sondern nach ihrem dynamischen Kommunikationsverhalten auf Rechnerknoten zu verteilen.

In Abbildung 2.4 erkennt man, daß bei einer Unterteilung nach statischen Gesichtspunkten (senkrechte Linien) die Bindungen, entlang derer die Kommunikation verläuft, über die Grenzen der Rechnerknotens hinweggehen. Bei Unterteilung nach dynamischen Gesichtspunkten (waagerechte Linien) laufen die Bindungen innerhalb eines Knotens. Diese Unterteilung entspricht der in Abbildung 2.3 vorgenommenen Unterteilung in dynamische Komponenten.

Der Begriff der Lokalität soll sich daher auf dynamische Komponenten (*component instances*) beziehen. Dadurch kann die für dynamische Komponenten vorgegebene hierarchische Struktur zur Bestimmung der Lokalität verwendet werden, indem alle Subkomponenten einer dynamischen Komponente zu ihr lokal gesetzt werden. Diese Zuordnung ist eindeutig, da jede dynamische Komponente Teil höchstens einer übergeordneten dynamischen Komponente ist. Man kann davon ausgehen, daß die Kommunikation innerhalb einer dynamischen Komponente stärker als die mit der Umgebung ist; in diesem Fall ist effiziente, lokale Kommunikation gewährleistet.

Die Verwendung eines hierarchischen Lokalitätsbegriffs statt einer flachen Partitionierung des Objektraums findet sich unter anderem bei [Cardelli, Gordon 98] und [Stoutamire 97].

2.3.3. Interaktion

Interaktion ist ein Phänomen der dynamischen Ebene. Die Regeln, nach der sie abläuft, finden sich wieder auf der statischen Ebene. Eine konkrete Interaktion zwischen Objekten läßt sich also als Exemplar einer statischen Verhaltensstruktur ansehen.

In der Modellierungssprache UML wird ein konkreter Ablauf auf der dynamischen Ebene mit Hilfe eines Sequenz- oder Kollaborationsdiagramms beschrieben, während die möglichen Abläufe statisch durch Zustands- und Aktivitätsdiagramme erfaßt werden. Die Grenze ist jedoch verschwommen: Auch in Sequenz- und Kollaborationsdiagrammen können Notationen gebraucht werden, die wiederholbare oder optionale Teile andeuten. Damit kann ein Sequenzdiagramm auch auf die statische Ebene gehören. In dieser Arbeit soll unter einem Sequenzdiagramm jedoch immer die Exemplar-Form verstanden werden.

Abbildung 2.5 zeigt ein Beispiel für ein Sequenzdiagramm. Dargestellt ist die Übergabe eines Vorgangs von einem Arbeitsplatz an einen anderen mit Hilfe der Hauspost.

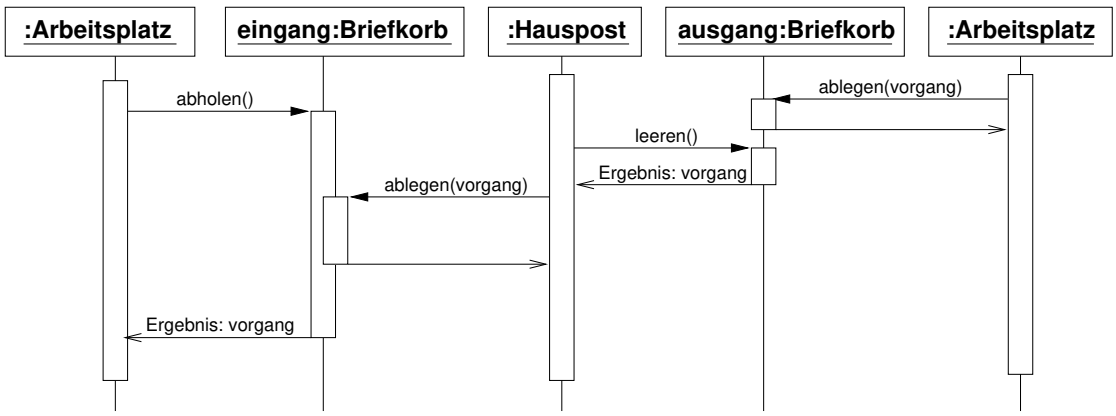

Abbildung 2.5.: Ein UML-Sequenzdiagramm

Die Zeit verläuft von oben nach unten, Nachricht und Ergebnis werden durch horizontale Pfeile repräsentiert. Zu jedem Zeitpunkt (dargestellt durch einen horizontalen Schnitt) gibt es eine Anzahl Lebenslinien, die lebendige Objekte darstellen. Jedes lebendige Objekt kann eine Anzahl Aktivierungen haben, die jeweils eine laufende Berechnung auf dem Objekt darstellen.

Interaktion läßt sich auf wählbarer Granularitätsebene betrachten. In der Darstellung als Kollaborationsdiagramm (Abbildung 2.6) wird besonders deutlich, was geschieht, wenn Objekte zu einer dynamischen Komponente zusammengefaßt werden. Hier werden die Briefkörbe als Subkomponenten der dynamischen Komponente Hauspost aufgefaßt.

Im Kollaborationsdiagramm werden die Nachrichten durchnumeriert über ein Objektdiagramm gezeichnet. Man erkennt, welche Nachrichten über die Grenze der dynamischen

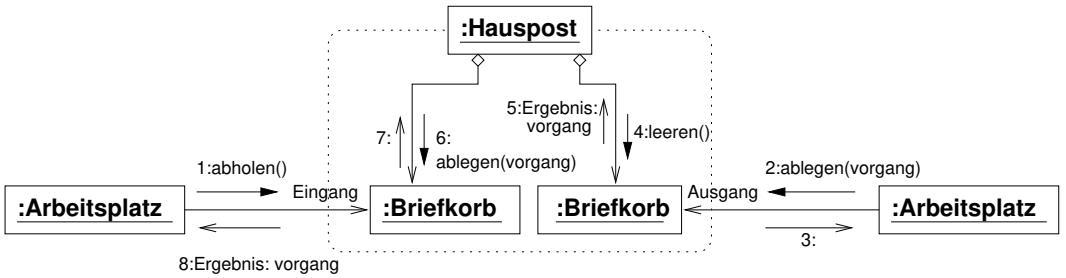

Abbildung 2.6.: Die Interaktion aus Abbildung 2.5 als UML-Kollaborationsdiagramm

Komponente hinweglaufen, und welche innerhalb der Komponente bleiben. Die Vergröberung auf der Objektebene erlaubt damit auch eine Vergröberung der Interaktion: Von Kommunikation innerhalb einer dynamischen Komponente wird abstrahiert, und Kommunikation mit einem beliebigen Objekt innerhalb der dynamischen Komponente wird als Kommunikation mit der dynamischen Komponente gedeutet. Damit ergibt sich das vergröberte Sequenzdiagramm in Abbildung 2.7. Es fällt auf, daß gegenüber Abbildung 2.5 nicht nur Nachrichten weggefallen sind, sondern auch Aktivierungen.

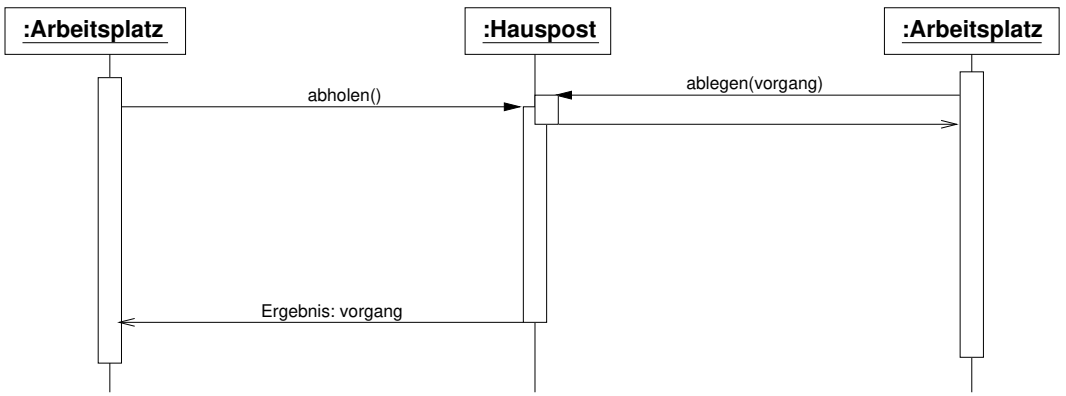

Abbildung 2.7.: Vergröbertes UML-Sequenzdiagramm

Der Unterschied zwischen dynamischer und statischer Ebene wird hier noch einmal deutlich. In einem System können mehrere Exemplare der Klasse `Arbeitsplatz` auftreten, die bei der Darstellung der Interaktion unterschieden werden müssen. Zum Beispiel können verschiedene Exemplare auch auf unterschiedlicher Detailebene betrachtet werden.

In dem Beispiel von Abschnitt 2.3.1 wird es so möglich, die Kommunikation zwischen einer Finanzbuchhaltung und ihrer Datenbank zu zeigen, von Kommunikation mit der Datenbank der anderen Finanzbuchhaltung und der Datenbank der Terminverwaltung aber zu abstrahieren. Interaktion und damit auch ihre Vergröberung gehört auf die dynamische, nicht auf die statische Ebene.

2.3.4. Migration

Die Lokalität einer Komponente muß nicht feststehen. Im Beispiel des Bürosystems befindet sich ein Vorgang im Verlauf seiner Lebenszeit in verschiedenen dynamischen Komponenten: Er wird als Subkomponente eines Arbeitsplatzes konstruiert (1), durch die Hauspost weitergereicht (2–4) und wird schließlich zu einer Subkomponente eines anderen Arbeitsplatzes (5). Abbildung 2.8 stellt diesen Ablauf dar.

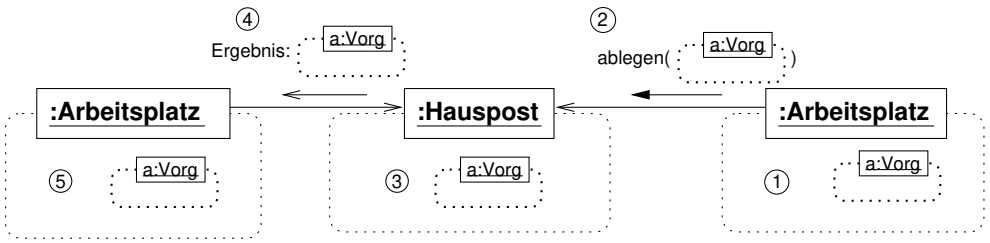

Abbildung 2.8.: Migration von und zwischen dynamischen Komponenten

In einer räumlichen Metapher sind die von der Post übermittelten Nachrichten nicht so sehr Briefe, die nur Text transportieren, sondern Pakete, die beliebige weitere Objekte enthalten können. Der Vorgang wird also bei der Zustellung nicht kopiert, sondern bewegt. Insbesondere bleibt die Identität enthaltener Objekte erhalten.

Ähnlich wie Interaktion bezieht sich der Begriff der Migration nur auf die dynamische Ebene. Migration bedeutet, daß dasselbe Phänomen vor der Migration an Ort A ist, sich nach der Migration aber nicht mehr an Ort A befindet, sondern an Ort B. Zum einen kann einem statischen Phänomen nur schwer eine Identität zugeordnet werden, weshalb nicht zu sagen ist, ob das Phänomen an Ort B dasselbe wie bei A ist, und zum anderen kann ein statisches Phänomen wie in Abschnitt 2.3.2 beschrieben auch an mehreren verschiedenen Orten gleichzeitig wirken.[1]

2.3.5. Komposition

Wie Lokalität tritt auch Komposition sowohl auf der statischen als auch auf der dynamischen Ebene auf. Zum einen wird eine Anwendung oder eine übergeordnete Softwarekomponente aus Softwarekomponenten zusammengestellt; hier entsteht also ein Programm durch Komposition passender Programmbestandteile. Diese Komposition spiegelt sich direkt auf der dynamischen Ebene wider, da z.B. eine zur Laufzeit angelegte Datenbank Exemplar des statisch ausgewählten Datenbanksystems ist, oder im Falle eines Dialog-Exemplars die Anordnung und Art der Dialogelemente genau der statisch festgelegten entspricht.

[1] Aus diesem Grund wird Code in verteilten Systemen gewöhnlich kopiert. Ein Beispiel ist der Aufruf eines *applet* in Java, wodurch die benötigten Klassen über das Internet auf den lokalen Rechner übertragen, d.h. kopiert werden.

Auf der dynamischen Ebene sind aber noch weitere, flexiblere Kompositionsmöglichkeiten gegeben. So kann etwa die Entscheidung, welcher Dienstleister zu verwenden ist, bis zur Laufzeit aufgeschoben werden (späte Bindung). Zum Beispiel könnte eine Textverarbeitung mit Hilfe eines Verzeichnisdienstes nach einer beliebigen gerade verfügbaren Rechtschreibprüfung suchen, die lediglich eine von der Textverarbeitung verstandene Schnittstelle anbieten muß.

In objektorientierten Programmiersprachen wird dies durch Entwurfsmuster wie das der abstrakten Fabrik (*abstract factory*, [Gamma et al. 95]) oder allgemeiner durch abstrakte Schnittstellen und *callbacks* unterstützt. Stärker als andere Paradigmen bietet Objektorientierung die Möglichkeit, aus generischen Bestandteilen Strukturen aufzubauen, deren Zweck von keinem Bestandteil vorausgesehen wurde. Es ist daher entscheidend, dynamische Komponenten in ihrem dynamischen Kontext und nicht nur in Bezug auf ihre statische Struktur zu sehen.

2.4. Aggregation

Die Beziehung zwischen einer dynamischen Komponente und ihren Subkomponenten (die *Komponentenbindung*) ist eine Aggregation, d.h. eine Teil-Ganzes-Beziehung. Neben Klassifikation, Generalisierung und Assoziation handelt es sich bei Aggregation um ein grundlegenes Modellierungsmittel der objektorientierten Analyse, das in verschiedenen Methoden aber sehr unterschiedlich gedeutet und eingesetzt wird. Fast jede Beziehung läßt sich in irgendeiner Hinsicht als Beziehung zwischen einem Ganzen und einem Teil sehen, was [Civello 93] zu folgender Feststellung führt:

> "*...the difference between whole-part associations and other class associations is often only cosmetic and diagrammatic. While it is generally acknowledged that whole-part associations bind classes more strongly than other associations, there are no further rules or constraints to guide design and implementation decisions.*"

Um diesem Mißstand abzuhelfen, gibt Civello für Analyse und Entwurf jeweils einen Liste von Merkmalen an, die auf eine Aggregation zutreffen können, und durch deren Angabe die Art der Aggregation genauer spezifiziert wird.

2.4.1. Aggregation in der objektorienterten Analyse

Auf der Analyse-Ebene unterscheidet Civello zwischen funktionaler und nicht-funktionaler Aggregation. Eine Aggregation ist funktional, wenn das Teil zur Funktion des Ganzen beiträgt. Objekte, die nur aufgrund einer gemeinsamen Eigenschaft zu einer Menge zusammengefaßt werden, stehen in einer nicht-funktionalen Aggregationsbeziehung zu dem Objekt, das die Menge repräsentiert. Beispiele für nicht-funktionale

Aggregation sind die Beziehung Komitee-Mitglied und die Menge der Studenten, die ihre Vordiplomsprüfung abgelegt haben.

Im einzelnen hängt die Entscheidung zwischen funktionaler und nicht-funktionaler Aggregation von der Funktion ab, die man dem Ganzen zuschreibt. So führt z.B. auch räumliche oder zeitliche Inklusion nur dann zu einer Aggregation im Analysemodell, wenn die Anwendung räumliche oder zeitliche Sachverhalte modelliert.

Aus der in Abschnitt 2.1 gegebenen Definition eines Systems folgt, daß es sich bei der Komponentenbindung um eine funktionale Aggregation handelt.

2.4.2. Aggregation im objektorientierten Entwurf

Auf der Entwurfsebene liefert Civello recht differenzierte Unterscheidungsmerkmale. Dadurch ist das volle Spektrum der in den gängigen Modellierungssprachen vorkommende Aggregationskonzepte ausdrückbar.

Als notwendiges Kriterium für eine Aggregation fordert Civello nur, daß das Teil für das Ganze sichtbar sein muß, das Ganze dem Teil also Nachrichten senden kann. Frei spezifiziert werden können folgende Eigenschaften:

Sichtbarkeit Schränkt die Aggregation die Sichtbarkeit ein? D.h. von welchen anderen Objekten (innerhalb oder außerhalb) kann ein Teil abhängen, und welche anderen Objekte können das Teil aktiv beeinflussen?

Trennbarkeit Können Teil oder Ganzes auch ohne das jeweils andere existieren? Ein Segelschiff ohne Segel kann nicht segeln, und das Segel funkioniert nur im Zusammenhang mit einem Schiff. Diese Beziehung ist also in beide Richtungen nicht trennbar. Soll jedoch die Montage eines Segelschiffs modelliert werden, so macht ein einzelnes Segel und ein Schiff ohne Segel Sinn.

Exklusivität Kann ein Teil zu mehreren Ganzen gehören?

Änderbarkeit Ist die Beziehung änderbar? Das Segel eines Segelschiffs kann jederzeit gegen ein anderes ausgetauscht werden, ohne die Funktion zu beeinträchtigen. Umgekehrt kann dasselbe Segel nacheinander auf verschiedenen Schiffen eingesetzt werden.

Kaskadiertes Löschen Werden die Teile automatisch freigegeben, wenn das Ganze freigegeben wird? Diese Eigenschaft hängt eng mit Trennbarkeit zusammen.

Civello wendet die Eigenschaften jeweils auf eine konkrete Aggregationsbeziehung zwischen zwei Klassen an. Die Eigenschaften können jedoch auch benutzt werden, um die in einer bestimmten Modellierungssprache verwendete Art von Aggregation genauer zu beschreiben. Die definierten Einschränkungen gelten dann für alle in der jeweiligen Modellierungssprache als Aggregation bezeichneten Beziehungen.

In der UML wird z.B. eine *Komposition* genannte Variante der Aggregation definiert, die weder trennbar noch änderbar ist, Exklusivität ausschließt und kaskadiertes Löschen impliziert. Der Begriff *Aggregation* ist in UML laut [Fowler, Scott 97] nur anschaulich definiert und impliziert gegenüber Assoziation keine Einschränkungen.

2.4.3. Komponentenbindung als Aggregation

In diesem Abschnitt soll die Komponentenbindung als eine Art der Aggregation unter Verwendung der von Civello aufgestellten Eigenschaften charakterisiert werden. In den Begriffen von UML wird Komponentenbindung also als *Stereotyp* aufgefaßt, dessen Eigenschaften für alle als Komponentenbindung ausgezeichneten Aggregationen gelten sollen.

Eine Aggregation muß gewissen Einschränkungen genügen, um Komponentenbindung sein zu können. Wesentlich ist, daß die durch Komponentenbindungen erzeugte Struktur zu jedem Zeitpunkt eine hierarchische Verfeinerung zuläßt, also mathematisch gesehen ein Wald ist. Dies wird durch folgende Anforderungen sichergestellt:

A1 (Exklusivität) Die Bindung einer Subkomponente in verschiedenen dynamischen Komponenten wird ausgeschlossen. Jedes Objekt hat höchstens eine Superkomponente.

A2 (Zyklenfreiheit) Kein Objekt ist transitiv Subkomponente seiner selbst.

Zyklenfreiheit wird von Civello nicht betrachtet; aufgrund ihrer Relevanz für die Komponentenbindung soll sie hier aber dennoch aufgeführt werden. Man beachte, daß **A2** (Zyklenfreiheit) nicht aus **A1** (Exklusivität) folgt.

Andere Eigenschaften der Aggregation werden durch die Komponentenbindung nicht vorgeschrieben: Sichtbarkeit, Trennbarkeit, Änderbarkeit und Kaskadiertes Löschen hängen von der jeweiligen Aggregation ab.

Die Komponenten eines offenen Systems können mit der Umgebung interagieren, neben der Sichtbarkeit innerhalb des Systems ist in diesem Fall also auch eine Sichtbarkeit von und nach außen gegeben. Andererseits ist nicht jedes System offen, weshalb die Sichtbarkeit nicht erzwungen wird. Die Anforderung lautet wie folgt:

A3 (Sichtbarkeit) Eine Komponentenbindung schränkt die Sichtbarkeit i.A. nicht ein. Komponentenbindung impliziert also nicht Kapselung.

Die Anzahl und Struktur der Subkomponenten soll über die Zeit flexibel sein können. Kardinalitäten größer als 1 werden auf der Entwurfsebene durch rekursive Datenstrukturen oder Behälterobjekte variabler Größe erreicht. Dafür müssen folgende Möglichkeiten gegeben sein:

A4 (Polymorphie) Die von verschiedenen Ausprägungen einer Komponentenbindung gebundenen Objekte können von verschiedener Form, d.h. von verschiedener Größe oder Exemplare verschiedener Klassen sein. Insbesondere sollen auch rekursive Datenstrukturen möglich sein.

A5 (Änderbarkeit der Subkomponente) Eine Komponentenbindung kann änderbar sein. Im Verlauf der Zeit können sowohl Zustand als auch Identität und Form (Klasse bzw. Größe) des gebundenen Objekts variieren.

A6 (Trennbarkeit der Superkomponente) Eine Komponentenbindung kann leer sein, d.h. das Ganze kann auch ohne das Teil existieren.

Die Eigenschaft der Polymorphie stammt nicht von Civello, ist aber zur Unterscheidung verschiedener Arten von Aggregation nützlich. Eine nicht-polymorphe Art der Aggregation (wie etwa die Komposition in UML) eignet sich nicht dazu, Komponentenbindung auszudrücken.

Zu klären ist noch die Änderbarkeit der Superkomponente: Ist eine Komponente auch in einem anderen als ihrem ursprünglichen System vorstellbar? Verschiedene dynamische Komponenten können als Subkomponente ein Exemplar der gleichen statischen Komponente verwenden. Es ist damit denkbar, daß eine solche Subkomponente ihre dynamische Komponente verläßt und von einer anderen aufgenommen wird. Als Beispiel hierfür dient das beschriebene Bürosystem: Ein Dokument kann vom Arbeitsplatz einem Vorgang hinzugefügt werden. Dabei wechselt das Dokument sogar die Granularitätsebene; dies ist möglich, weil Systeme und Komponenten auf jeder Ebene einheitlich durch Objekte und Klassen beschrieben sind. Die Anforderung lautet also:

A7 (Änderbarkeit der Superkomponente) Im Verlauf seiner Lebenszeit kann ein Objekt verschiedene Superkomponenten haben.

Die Klasse `Dokument` steht in zwei Komponentenbindungen, einmal zu `Vorgang` und einmal zu `Arbeitsplatz`. Da jedes Exemplar von `Dokument` nur eine Superkomponente haben kann, bleibt die andere Bindung leer; insofern ist die Superkomponente in einer Komponentenbindung optional. Dies gilt insbesondere für die Wurzeln des von der Komponentenbindung gebildeten Komponenten-Waldes: diese Objekte haben keine Superkomponente. Es ergibt sich folgende Anforderung:

A8 (Trennbarkeit der Subkomponente) Ein Objekt kann auch ohne Superkomponente existieren.

Da ein Objekt also keine Superkomponente haben muß, und sich dieser Status aufgrund von **A7** (Änderbarkeit der Superkomponente) während der Lebenszeit eines Objekts ändern kann, besteht i.A. kein Grund, die Freigabe der Superkomponente auf die Subkomponenten zu übertragen. Wird ein Objekt freigegeben, so verlieren seine Subkomponenten ihre Superkomponente, können aber weiterexistieren. Daraus ergibt sich

A9 (Kein kaskadiertes Löschen) Wird ein Objekt freigegeben, so werden seine Subkomponenten nicht automatisch mit freigegeben. Dies gilt auch für die automatische Speicherbereinigung: Ein unerreichbares Objekt kann freigegeben werden, auch wenn seine Subkomponenten noch erreichbar sind und daher weiterexistieren.

Aus der Änderbarkeit der Superkomponente folgen aber noch weitreichendere Konsequenzen. Wenn alle transitiven Subkomponenten einer dynamischen Komponente sich potentiell auf jede Granularitätsebene innerhalb der dynamischen Komponente bewegen können, dann sind alle diese Objekte prinzipiell gleichberechtigt. Was von innen wie eine dynamische Komponente, also wie ein System wirkt, wird von außen zu einem Objekt, also zu einer Subkomponente der nächsthöheren Ebene. Aus diesem Blickwinkel ist jede dynamische Komponente ein Objekt, und umgekehrt jedes Objekt eine dynamische Komponente. Eine zunächst undifferenzierte Menge von Objekten wird erst durch die Komponentenbindung zu dynamischen Komponenten verschiedener Granularitätsebenen zusammengefaßt.

Eine vollständige Liste der in diesem Kapitels aufgestellten Anforderungen an Komponentenbindung gibt Tabelle 2.2.

A1	Exklusivität
A2	Zyklenfreiheit
A3	Sichtbarkeit
A4	Polymorphie
A5	Änderbarkeit der Subkomponente
A6	Trennbarkeit der Superkomponente
A7	Änderbarkeit der Superkomponente
A8	Trennbarkeit der Subkomponente
A9	Kein kaskadiertes Löschen

Tabelle 2.2.: Überblick der Anforderungen an Komponentenbindung

2.4.4. Rekursive Aggregation

In Abschnitt 2.1 wurde gezeigt, daß Exemplare einer Klasse auf verschiedenen dynamischen Granularitätsebenen auftreten können. Insbesondere im Falle einer rekursiven Datenstruktur ist es aufgrund der Zyklen in der statischen Struktur nicht möglich, eine Klasse einer bestimmten Ebene zuzuordnen. Dies soll am Beispiel der syntaktischen Analyse eines Textes erläutert werden.

Wird eine Folge von Terminalsymbolen anhand einer kontextfreien Grammatik strukturiert, so läßt sich jedem Knoten des Ableitungsbaums ein Bereich der Eingabe zuordnen. Der Bereich eines Knoten umfaßt dabei den Bereich all seiner Unterknoten. Ein Zwischenschritt in der Ableitungssequenz wird dargestellt, indem von den Unterknoten eines Knoten abstrahiert wird.

Werden wie im *Interpreter-Muster* von [Gamma et al. 95] die Produktionen und Nonterminale als Elemente der statischen Ebene als Klassen aufgefaßt, so wird jeder Knoten des Ableitungsbaums zu einem Objekt. Die durch den Anwendungsbereich vorgegebene hierarchische Struktur erlaubt es, die Unterknoten eines Knoten als Subkomponenten des Objektes darzustellen. Die Entsprechung von Grammatik und Klassendiagramm ist in Abbildung 2.9 dargestellt; die Beispielgrammatik stammt aus [Hopcroft, Ullman 79].

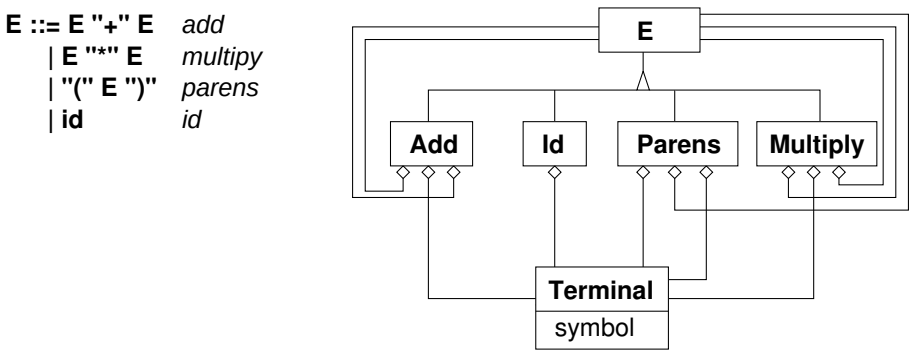

Abbildung 2.9.: Kontextfreie Grammatik und UML-Klassendiagramm

Im Objektdiagramm in Abbildung 2.10 kann man sehen, daß Exemplare von Subklassen von `E` auf allen Granularitätsebenen auftreten. Das Klassendiagramm enthält sogar eine zyklische Aggregation der Klasse `E`. Auf der dynamischen Ebene sind Zyklen aufgrund der Konstruktion des Ableitungsbaumes ausgeschlossen.

Erst Anforderung **A4** (Polymorphie) aus Abschnitt 2.4.3 macht es möglich, in diesem Beispiel Komponentenbindung zu verwenden, da die Anforderung es gestattet, Exemplare verschiedener Subklassen von `E` als Subkomponenten einzusetzen.

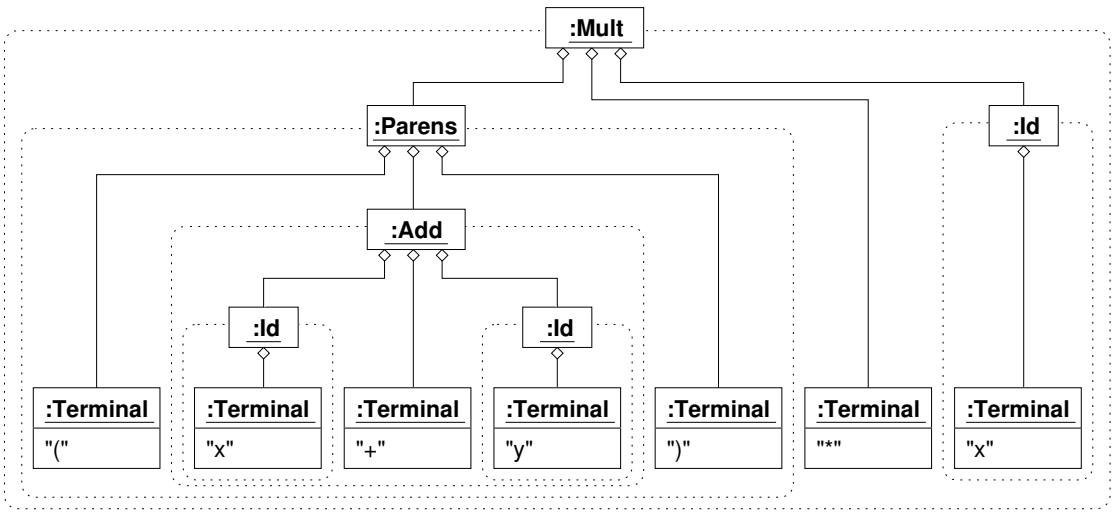

Abbildung 2.10.: Ein Ableitungsbaum aus dynamischen Komponenten

Rekursive, hierarchische Datenstrukturen wie die in diesem Abschnitt gezeigte sind ein allgegenwärtiges Konzept in der Programmierung, besonders in der objektorientierten. Das Muster *composite* [Gamma et al. 95] tritt in fast jedem objektorientierten Entwurf auf. Die Verarbeitung von Texten (Programmiersprachen, SGML, XML, TEX) wird durch das gegebene Grammatik-Beispiel als Anwendungsgebiet nahegelegt. Auch bei der grafischen Darstellung zweidimensionaler (z.B. Dialogelemente) und dreidimensionaler Objekte (PHIGS, VRML) und im computerunterstützten Entwurf (*computer aided design, CAD*) spielen derartige Hierarchien eine wichtige Rolle. Es lohnt sich daher, rekursive, hierarchische Datenstrukturen durch Konzepte in der Modellierung zu unterstützen und durch programmiersprachliche Konzepte wie etwa *environmental acquisition* [Gil, Lorenz 96] oder die in dieser Arbeit beschriebenen dynamischen Komponenten auszunutzen.

3. Verwandte programmiersprachliche Konzepte

Nachdem im vorangegangenen Kapitel das Konzept der dynamischen Komponente entwickelt wurde, wird es im Rest der Arbeit um seine Umsetzung und Anwendung in einer Programmiersprache gehen. In diesem Kapitel sollen verwandte programmiersprachliche Konzepte untersucht werden, um Vorarbeiten zu identifizieren und, soweit geeignet, im Analogieschluß auf die für Tycoon-2 zu entwickelnde Lösung zu übertragen.

Dazu werden zunächst in Abschnitt 3.1 die in existierenden Programmiersprachen vorhandenen Bindungsarten betrachtet, sodann in Abschnitt 3.2 allgemeinere sprachliche Mechanismen, die die Gruppierung von Objekten ausdrücken, und schließlich wird in Abschnitt 3.3 ein Einblick in die Literatur zum Thema Kapselung und Mehrfachbenennung in objektorientierten Sprachen gegeben.

Abschnitt 3.4 stellt die Ergebnisse für die Umsetzung dynamischer Komponenten zusammen.

3.1. Bindungen in existierenden Programmiersprachen

In diesem Abschnitt soll beschrieben werden, inwieweit in existierenden Programmiersprachen Komponentenbindung durch die vorhandenen Bindungsarten ausgedrückt werden kann. Dazu wird für die untersuchten Sprachen zunächst beschrieben, welche Arten von Bindungen es zwischen Objekten in der jeweiligen Sprache geben kann, um dann zu zeigen, inwieweit diese Beziehungen den aufgestellten Anforderungen an die Komponentenbindung genügen. Die Ergebnisse dieses Abschnitts sind in Tabelle 3.1 zusammengestellt.

Um die gewählte Lösung besser einordnen zu können, und um auch von Erfahrungen in anderen Bereichen profitieren zu können, werden auch nicht-objektorientierte imperative oder imperativ-funktionale Programmiersprachen betrachtet. Gegebenenfalls ist dann im einzelnen zu identifizieren, was als Objekt gelten soll. Objekte müssen zumindest benennbar sein. Als weiterer Anhaltspunkt soll die Objektidentität dienen, die sich darin äußert, daß unter einem Namen vorgenommene Änderungen auch unter allen anderen an dasselbe Objekt gebundenen Namen sichtbar werden (*aliasing*). Ob diese Objekte auch noch Verhalten kapseln, interessiert hier nicht, da der Schwerpunkt auf dem Datenmodell liegt.

Sprache	Name des Konzepts	A1	A2	A3	A4	A5	A6	A7	A8	A9
Pascal	**record**,**array**	√	√	–	–	–	–	–	√	–
C	**struct**, **union**, *array*	√	√	√	–	–	–	–	√	–
BETA	statische Referenz	√	√	√	–	–	–	–	√	?
Eiffel	**expanded**	√	√	–	–	–	–	–	√	–
SML	Bindung	–	√	√	√	–	–	√	√	√
ORION	*composite link*	√	√	√	–	√	√	√	–	–
–	*balloon*	√	√	–	√	√	√	–	–	√
Tyc@@n-2	Komponentenbindung	√	√	√	√	√	√	√	√	√
Pascal	*pointer*	–	–	√	–	√	√	√	√	√
C	*pointer*	–	–	√	√	√	√	√	√	√
BETA	dynamische Referenz	–	–	√	√	√	√	√	√	√
Eiffel	Bindung	–	–	√	√	√	√	√	√	√
SML	Speicherzelle	–	–	√	√	√	–	√	√	√
ORION	*non-composite link*	–	–	√	?	√	√	√	√	√
Tyc@@n-2	Referenzbindung	–	–	√	√	√	√	√	√	√

Tabelle 3.1.: Eigenschaften der untersuchten Bindungskonzepte

3.1.1. Strukturierte Variablen in Pascal

Als erstes vorgestellt werden soll die älteste betrachtete Sprache, Pascal, aus dem Jahre 1971 [Wirth 71]. Die Sprache Pascal wurde ausgewählt, weil sie aufgrund der damaligen Rechnerkapazitäten sehr implementationsorientiert ist, auf der anderen Seite aber über eine klare Semantik verfügt [Hoare, Wirth 73]. Das in Pascal ausgenutzte Implementationsprinzip der geschachtelten Speicherbereiche fester Größe findet sich unter anderem auch in jüngeren Sprachen wie C [Kernighan, Ritchie 78] [Kernighan, Ritchie 88], C++ [Stroustrup 86] [Stroustrup 92] und BETA [Madsen et al. 93]. Die so erreichte Bindungsart wird oft mit Aggregation gleichgesetzt. In diesem Abschnitt wird daher detailliert untersucht, was für eine Art von Aggregation tatsächlich umgesetzt wird.

Pascal ist eine imperative Programmiersprache in der Algol-Tradition. Ihr Typsystem beinhaltet neben einigen Basistypen Typkonstruktoren zur Definition von Datensätzen (*record*), Feldern (*array*) und Zeigern (↑). Ein Datensatztyp kann verschiedene Varianten enthalten. Die angegebenen Typkonstruktoren können beliebig geschachtelt werden, allerdings mit der Einschränkung, daß der Compiler statisch den maximalen Speicherbedarf einer Variable des konstruierten Typs bestimmen kann. Das bedeutet zum einen, daß die Anzahl der Elemente eines Feldtyps statisch feststehen muß, und zum anderen, daß rekursive Typdefinitionen nur durch Zeiger erlaubt sind.

Die Terminologie des Pascal-Sprachstandards bezeichnet auch einen Ausdruck wieder als *Variable*, der auf ein Datenfeld einer Datensatz-Variable oder auf eine Komponente einer Feld-Variable zugreift. Auch ein dereferenzierter Zeigerausdruck ist eine Variable.

Wesentlich für eine Variable ist also, daß sie verschiedene Werte annehmen kann, und daß ihr neue Werte zugewiesen werden können.

Der Sprachstandard trennt dabei nicht zwischen einer Variable als Teil des Programmtextes und einer Variable als einem Ort, an dem der aktuelle Speicherzustand einen Wert ablegt. Variablen im zweiten Sinne sollen vorerst *Adresse* heißen. Eine Variable denotiert in einer gegebenen Umgebung eine Adresse. In Pascal kann man auch vom Typ der Adresse sprechen, da alle an die Adresse gebundenen Variablen denselben Typ haben müssen.

Wie entstehen überhaupt mehrfache Bindungen an dieselbe Adresse? Zum einen ist es möglich, über Zeigervariablen dynamisch neue Adressen anzufordern, und den Wert der Zeigervariablen auch anderen Zeigervariablen zuzuweisen. Egal, welche dieser Zeigervariablen dann dereferenziert wird, immer wird die Adresse des angeforderten Speichers denotiert.

Zum anderen können Variablen aller angegebenen Typen an Variablenparameter übergeben werden. In der aufgerufenen Prozedur oder Funktion ist der Variablenparameter dann an die gleiche Adresse wie die als Argument übergebene Variable gebunden (*call-by-reference*). Daher können nicht nur dynamisch angeforderte Adressen mehrfach benannt werden, sondern tatsächlich alle Adressen.

Aus den oben angegebenen Gründen soll daher der Begriff der Objektidentität in Pascal auf den Begriff der Adresse abgebildet werden, auch wenn ein naiver Ansatz dies nur für dynamisch angeforderte Adressen tun würde. Zu einem *Objekt* gehören Identität und aktueller Zustand. Diesen entspricht in Pascal die Adresse und der durch den Speicher gegebene Inhalt der Adresse.

Am folgenden Codebeispiel lassen sich einige Konsequenzen dieses Objektbegriffs zeigen. Das Beispiel ist an [Blake, Cook 87] angelehnt.

```
type
  Strichmann = record
      rechterArm, linkerArm: Arm;
      rechtesBein, linkesBein: Bein
    end;
  Arm = record oberarm, unterarm: Linie end;
  Bein = record oberschenkel, unterschenkel, fuss: Linie end;
  Linie = record x1,y1,x2,y2: Integer end;
procedure dreh(var eineLinie: Linie, alpha: Real);
  ...;
procedure anwinkeln(var einArm: Arm);
  begin dreh(einArm.unteram, pi/4) end;
procedure test();
  var axel: Strichmann;
  begin ...; anwinkeln(axel.rechterArm); ... end;
```

Wenn zur Laufzeit die Variable *axel* vereinbart wird, also der Variable ein Objekt zugeordnet wird, dann wird sofort auch zu jeder durch Feldzugriff oder Indizierung abgeleiteten Variablen ein Objekt angelegt, z.B. *axel.rechterArm.oberarm*. Dies gilt auch für abgeleitete Variablen in Varianten von Datensätzen. Sobald die Lebenszeit der Variable endet, werden auch alle abgeleiteten Objekte freigegeben. Die erzeugte Objektstruktur entspricht der Struktur des Typausdrucks, ist also ein Baum.

Die Adressen von abgeleiteten Variablen sind nicht änderbar. Bei einer Zuweisung wie zum Beispiel

procedure *transplantiereRechtenArm(*
var *spender, empfaenger: Strichmann);*
begin *empfaenger.rechterArm := spender.rechterArm;* **end***;*

wird nicht etwa die Adresse des Spenderarms statt des Empfängerarms eingesetzt, sondern sein Zustand wird kopiert. Zur genauen Beschreibung dieser Zuweisung wird der Begriff des Komponentengraphen benötigt.

Der *Komponentengraph* ist ein gerichteter Graph, dessen Knoten die Objekte sind. Gerichtete Kanten laufen von einem Feld-Objekt zu seinen indizierten Komponenten-Objekten und von einem Datensatz-Objekt zu seinen benannten Datenfeld-Objekten. Die Adressen in Zeiger-Objekten sollen nicht verfolgt werden. Dadurch werden alle Objekte mit Zeiger- oder Basistyp zu Blättern dieses Graphen; Objekte gleichen Typs haben isomorphe Objektgraphen.

Da Quelle und Ziel bei der Zuweisung vom gleichen Typ sein müssen, kann jedem Blatt des Zielgraphen ein Blatt des Quellgraphen zugeordnet werden, und der Inhalt der jeweiligen Adresse (ein Basiswert oder Zeiger) wird kopiert.

Daraus folgt, daß die Datensatz- und Feld-Objekte selbst unveränderlich sind, nur die referenzierten Objekte vom Zeiger- oder Basistyp können ihren Zustand über die Zeit ändern. Die Struktur des Komponentengraphen bleibt dadurch immer die eines Waldes: Bei der Vereinbarung einer neuen Variable kommt lediglich ein neuer Baum zu dem Komponentengraphen hinzu, der mit dem Ende der Lebenszeit der Variable wieder aus dem Komponentengraphen verschwindet.

Damit erfüllt die durch Datensatz- und Feldtypen in Pascal gegebene Beziehung zwischen Objekten schon zwei wesentliche Anforderungen an Komponentenbindung: Es gibt immer höchstens eine Superkomponente (**A1**), und der Komponentengraph ist zyklenfrei (**A2**). Im Widerspruch zu den Anforderungen ist die Beziehung aber nicht änderbar (**A5, A7**) und nicht trennbar (**A6, A8**) und erzwingt kaskadiertes Löschen (**A9**). Polymorphie (**A4**) ist nur eingeschränkt durch variante Datensätze möglich; da bei der Vereinbarung einer Variable Objekte für alle abgeleiteten Variablen miterzeugt werden, ist deren Anzahl für die Lebenszeit der Variable fest. Dies ist auch der Grund, weshalb die Sprachdefinition in Datensätzen und Feldern keine Typrekursion erlaubt, auch nicht in den Varianten eines varianten Datensatztyps. Folgende Definition ist zum Beispiel illegal:

```
type Strichleute = record case leer :Boolean of
    true:  ();
    false: (erster: Strichmann;
            rest: Strichleute);    (* illegal *)
  end;
```

Objektstrukturen wie diese müssen daher mit Hilfe von Zeigern implementiert werden.

```
type Strichleute = ↑ record
    erster: Strichmann;
    rest: Strichleute;
  end;
```

Durch die Verwendung von Zeigern ergibt sich eine sehr flexible Art der Bindung. Zeiger erlauben nicht nur Polymorphie beliebiger Größe, sie sind auch zur Laufzeit änderbar. Dadurch können wie im folgenden Beispiel Mehrfachbenennung und zyklische Strukturen erzeugt werden.

```
var ringelrein: Strichleute;
begin
  new(ringelrein);
  new(ringelrein↑.rest);
  ringelrein↑.rest↑.rest := ringelrein
end
```

Ein Zeigerobjekt kann nur die Adresse eines dynamisch angelegten Objekts enthalten, die möglichen Zeiger werden also durch den oben definierten Komponentengraph eingeschränkt (insofern verletzt die bisher betrachtete Bindungsart Anforderung **A3**, Sichtbarkeit). Eine aussagekräftige Struktur für den *Referenzgraphen* ergibt sich daher nur, wenn die Zeiger zum oben definierten Komponentengraphen hinzugenommen werden. Eine Kante in diesem Referenzgraphen gibt dann Navigierbarkeit an.

Blätter im Referenzgraphen sind nur noch die Objekte mit Basistyp. Wie eben gesehen ist der Graph kein Wald mehr, da auf ein dynamisch angelegtes Objekt mehrere Zeiger existieren können (**A1**), und auch Zyklen sind möglich (**A2**). Diese beiden wesentlichen Anforderungen an Komponentenbindungen werden also nicht erfüllt. Dafür sind aber sämtliche anderen Punkte erfüllt: Die Referenzbindung ist änderbar (**A5, A7**[1]), trennbar (**A6, A8**), schränkt die Sichtbarkeit nicht ein (**A3**) und führt nicht zu kaskadiertem Löschen (**A9**). Die Polymorphie (**A4**) wird in Pascal nicht geboten, die Größe eines durch den **new**-Ausdruck angelegten Objekts wird durch den statischen Typ festgelegt. Polymorphie muß daher wie im obigen Beispiel mit Hilfe der Trennbarkeit der Superkomponente (**A6**, ausgedrückt durch **nil**-wertige Zeiger) simuliert werden.

Aus dieser Charakterisierung der Referenzbindung folgt, daß sich Komponentenbindungen (wie im übrigen beliebige Assoziationen) mit Zeigern implementieren lassen. Da die wesentlichen Einschränkungen nicht ausgedrückt werden, geht der Unterschied zwischen Komponentenbindung und Referenzbindung dabei aber verloren.

[1] Da keine Exklusivität gegeben ist (**A1**), soll Anforderung **A7** so gedeutet werden, daß die *Menge* der Superkomponenten variabel ist.

3.1.2. C

Es ist möglich, in der Programmiersprache C Programme zu verfassen, die den Speicher ebenso strukturiert behandeln wie im gerade vorgestellten Modell. Der erste Unterschied besteht darin, daß es keine Variablenparameter gibt; stattdessen kann die Adresse jeder Variable (dort *lvalue*) als Zeigerwert erfragt werden kann. Dies lockert gegenüber Pascal die Beschränkung von Referenzen auf Subkomponenten und erlaubt so uneingeschränkte Sichtbarkeit (**A3**).

Ein deutlicher Unterschied zu Pascal besteht jedoch darin, daß mit Adressen, die ein Element eines Feldes bezeichnen, gerechnet werden kann (*pointer arithmetics*): Das i-te Feld plus n ergibt das $i+n$-te Feld. Zudem ist auf den abgeleiteten Objekten eines Objekts eine Ordnung definiert: Zuerst deklarierte Datenfelder eines Datensatzes (*struct*) haben kleinere Adressen; bei Komponenten eines Feldes entscheidet der Index. Von den Varianten einer *union* abgesehen ist diese Ordnung total. Interessanterweise ist die Adresse eines Datensatzes gleich der Adresse seines ersten Datenfeldes, und die Adresse eines Feldes gleich der Adresse seiner ersten Komponente. Dazu passend ist ein Objekt definiert als Bereich von Adressen, und der *sizeof*-Operator erlaubt es, die (statisch feststehende) Größe des Bereichs zu erfragen.

C erlaubt es, Zeiger auf einen Typ in Zeiger auf einen anderen Typ umzuwandeln, und erlaubt auch den Zugriff, wenn das *alignment* stimmt. Im Falle des Basistyps *char* ist das immer der Fall; C setzt also ein Speichermodell aus ungetypten Folgen von *bytes* voraus.

Das macht explizit, was in Pascal implizit bleibt: Komposition wird durch geschachtelte Speicherbereiche implementiert. Datensätze und Felder treten im Speicher nicht auf, sie existieren nur als Abstraktion im Programmtext. Damit ist auch klar, weshalb ein abgeleitetes Objekt nicht ohne sein Ganzes existieren kann; weshalb jedes Teilobjekt eine feste Maximalgröße haben muß; und weshalb die Identität eines Teilobjekts nicht änderbar ist, wird sie doch aus der Identität (und nicht etwa dem Zustand) des umfassenden Objekts durch Zeiger-Arithmetik errechnet.

Im Unterschied zu Pascal sind Zeiger in C polymorph (**A4**), da durch Aufruf einer Prozedur des Laufzeitsystems (`malloc`) eine beliebige Zahl von Bytes angefordert werden kann. Die Größe des zu alloziierenden Objekts kann also zur Laufzeit festgelegt werden, was u.a. Datenfelder variabler Größe ermöglicht.

3.1.3. Sprachen ohne Aggregation

Aufgrund der Implementationsnähe und daraus folgenden Komplexität von Komposition in Pascal und C verzichten viele Hochsprachen ganz auf ein vergleichbares Sprachkonstrukt. Dazu gehören u.a. Lisp, Smalltalk [Goldberg, Robson 83], Self [Ungar, Smith 87] und das am Arbeitsbereich STS der TU Hamburg-Harburg entwickelte TL und Tycoon-2. In diesen Sprachen gibt es kein Konstrukt, das implizite Erzeugung oder kaskadiertes

Löschen von Teilobjekten ausdrückt. Alle Beziehungen zwischen Objekten werden durch i.A. änderbare Referenzen dargestellt, wodurch größte Flexibilität gegeben ist, am Programm aber keine Unterscheidung zwischen Komponentenbindung und Referenzbindung mehr möglich ist.

3.1.4. BETA

Die Beschreibung der Programmiersprache BETA in [Madsen et al. 93] orientiert sich an einem konzeptuellen Rahmenwerk, nicht an der Hauptspeicherorganisation. Aus diesem konzeptuellen Rahmen entsteht der Wunsch, auch Aggregation auszudrücken; wie im folgenden gezeigt wird, benutzt BETA dennoch denselben Begriff von Komposition wie C.

BETA ist eine objektorientierte, statisch typisierte Programmiersprache, die auch funktionale und prozedurale Elemente ausdrückt. Klassen, Prozeduren und Funktionen werden durch einen einzigen Abstraktionsmechanismus ausgedrückt, das Muster (*pattern*). Muster können geschachtelt werden, und definieren statische Sichtbarkeitsbereiche (*static block scoping*). Ein Exemplar eines Musters entspricht sowohl einem Objekt als auch einer Prozeduraktivierung in anderen Programmiersprachen.

Ein Muster enthält eine Reihe von benannten Attribut-Deklarationen. Neben weiteren, verschachtelten Mustern (etwa für Prozedur-Muster) können auch statische oder dynamische Referenzen deklariert werden. Eine statische Referenz wird durch einen Klammeraffen (@) ausgezeichnet:

X: @T

Dabei ist X der Name der Referenz und T ein Muster. Bei der Erzeugung eines Exemplars des umgebenden Musters wird automatisch ein Exemplar von T miterzeugt. Das Attribut X denotiert daraufhin konstant dieses T-Objekt. Ein so erzeugtes Objekt wird auch Teil-Objekt genannt, da es permanenter Bestandteil des umgebenden Objekts ist.

Dynamische Referenzen haben folgende Syntax:

X: $\uparrow T$

Ein so deklariertes Attribut wird leer initialisiert, und kann später zu verschiedenen Zeitpunkten verschiedene Referenzen auf Objekte aufnehmen.

In Kapitel 10 der Sprachbeschreibung wird ausdrücklich gesagt, daß Teil-Hierarchien in Beta durch statische Referenzen umgesetzt werden. Die Definition des Strichmännchens aus Abschnitt 3.1.1 sieht in BETA wie folgt aus:

```
(#
  Linie:
    (# x1,y1,x2,y2: @integer;
       dreh: (# alpha: @integer
              in alpha
              do ... #);
    #);
  Arm:
    (# oberarm, unterarm: @Linie;
       anwinkeln: (# do pi/4 → unterarm.dreh #) #);
  Bein: (# oberschenkel, unterschenkel, fuss: @Linie #);
  Strichmann:
    (# linkerArm, rechterArm: @Arm;
       linkesBein, rechtesBein: @Bein;
    #);
  axel: @Strichmann;
  do ...; axel.rechterArm.anwinkeln; ...
#)
```

Da wie in Pascal alle Teilobjekte mit angelegt werden, sind rekursive Strukturen (**A4**) auch in BETA verboten. Außerdem sind statische Referenzen nicht änderbar (**A5, A7**) und nie leer (**A6**).

Um diese Eigenschaften auszudrücken, ist man wieder auf dynamische Referenzen, d.h. beliebig änderbare Zeiger angewiesen, was wie in Abschnitt 3.1.1 zyklische Strukturen (**A2**) erlaubt:

```
(#
  Strichleute:
   (# erster: @Strichmann;
      rest: ↑Strichleute;
   #)
  ringelrein: ↑Strichleute;
 do
   &Strichleute→ringelrein[];
   &Strichleute→ringelrein.rest[];
   ringelrein[]→ringelrein.rest.rest[]
#)
```

Das Waren-Und (*&*) bezeichnet die Erzeugung eines neuen Objekts. Durch die eckigen Klammern hinter einem qualifizierten Namen (z.B. *ringelrein.rest[]*) bezeichnet der Ausdruck die Identität des benannten Objekts. Durch diesen Referenzoperator läßt sich wie in C prinzipiell jedes Objekt referenzieren, Referenzen werden also durch Komposition nicht eingeschränkt (**A3**). Die Identität von dynamisch referenzierten Komponenten ist im Gegensatz zu statischen Referenzen änderbar (**A5**), wie die Zuweisung im obigen Beispiel zeigt. Da dynamische Referenzen leer (**NIL**) sein können, ist diese Bindungsart trennbar (**A6**). Dynamische Referenzen sind polymorph (**A4**), da ein zugewiesenes

Objekt Exemplar einer beliebigen Erweiterung des deklarierten Musters sein darf, und da im Fall von Datenfeldern die Größe des Feldes erst zur Laufzeit festgelegt wird. Im Gegensatz zu C geschieht dies typsicher.

BETA erlaubt eingeschränkte Polymorphie auch bei statischen Referenzen durch die Verwendung von virtuellen Mustern: In Erweiterungen des ursprünglichen Musters kann auch der Typ einer statischen Referenz erweitert werden. In jedem Fall wird das für die Erzeugung des Teilobjekts zu verwendende Muster statisch durch das umgebende Muster festgelegt; rekursive Strukturen bleiben unmöglich.

In der Sprachbeschreibung wird angedeutet, daß die Speicherverwaltung Objekte automatisch freigibt – zumindest stellt die Sprache keine Konstrukte zur expliziten Freigabe zur Verfügung. Welche Objekte jedoch genau als noch lebendig gelten, geht aus der Sprachbeschreibung nicht hervor; insofern kann hier nicht beantwortet werden, ob eine Subkomponente die Superkomponente am Leben erhält (**A9**). Ein Konstrukt, das die Lokalität eines gegebenen Objekts liefert, also auch die Umkehrung der Komponentenbindung navigierbar macht, wird zwar als mögliche Spracherweiterung genannt, ist aber nicht Bestandteil von BETA. Daher nimmt die Superkomponente i.A. keinen Einfluß auf die weitere Berechnung und könnte somit trotz existierender Subkomponenten freigegeben werden.

Im obigen Beispiel wurden Ganzzahlen (*integer*) wie in BETA üblich als Teilobjekte dargestellt. Der Grund für diese Umsetzung von Basiswerten als aggregierte Objekte wird in der Programmiersprache Eiffel noch deutlicher, und soll dort stellvertretend diskutiert werden.

3.1.5. Eiffel

Ähnlich wie BETA erhebt auch die Programmiersprache Eiffel den Anspruch, ein Konstrukt anzubieten, mit dem der Programmierer Aggregation ausdrücken kann. Wie im folgenden gezeigt wird, ist die Umsetzung in Eiffel aber noch restriktiver als in den anderen betrachteten Sprachen.

Eiffel ist eine statisch typisierte, klassenbasierte objektorientierte Programmiersprache, die als Vehikel für die klare Vermittlung und Umsetzung von objektorientiertem Entwurf geschaffen wurde. Das Typsystem von Eiffel bietet beschränkte Typparameter für Klassen (*bounded polymorphism*), etwa für Behälterklassen. Subtypisierung wird in Eiffel durch Vererbung definiert, nicht strukturell; dadurch beinhaltet eine Typaussage auch immer eine Aussage über die Repräsentation, u.a. über die vorhandenen Exemplarvariablen.

Eiffel wird von Meyer als fast zwangsläufiger Ausfluß der objektorientierten Weltsicht vorgestellt [Meyer 97]. Ähnlich wie in der Beschreibung von BETA wird daher auch Aggregation erwähnt, und ein Sprachkonstrukt für die Umsetzung angeboten.

Wird eine Exemplarvariable mit dem Schlüsselwort **expanded** deklariert, so enthält sie laut Sprachdefinition nicht eine Referenz auf ein Objekt, sondern das Objekt selbst. Diese Art der Bindung kann wie in den bisher beschriebenen Sprachen dargestellt werden als

nicht-änderbare Referenz, deren Ziel bei der Erzeugung des umfassenden Objekts miterzeugt wird. Wie bisher folgt daraus, daß dieser Kandidat für Komponentenbindung nicht polymorph und nicht änderbar oder trennbar (**A4–A7**), dafür aber zyklenfrei ist (**A2**). Jede Subkomponente hat höchstens eine Superkomponente (**A1**), und ihre Lebenszeiten stimmen überein (**A9**).

Ähnlich wie in Pascal sind Subkomponenten nicht referenzierbar (**A3**). Sie können nur als Empfänger einer Nachricht oder als Quelle oder Ziel einer Zuweisung auftreten. Man beachte, daß Eiffel hier noch restriktiver als Pascal ist, da es in Eiffel keine Variablen-Parameter gibt. Argumentübergabe und Zuweisung von Subkomponenten haben in Eiffel die aus Abschnitt 3.1.1 bekannte Kopiersemantik, d.h. die Inhalte der Blätter werden übertragen, die Objektstruktur bleibt jedoch gleich. Polymorphismus wird nur insofern unterstützt, als die Quelle der Zuweisung an eine expandierte Variable Exemplar einer Subklasse sein kann. Die zusätzlich verfügbaren Felder werden einfach weggelassen.

Wird eine Klasse als **expanded** definiert, so überträgt sich das Schlüsselwort auf sämtliche Verwendungen der Klasse in Variablendeklarationen. Diese Möglichkeit wird insbesondere für die Basistypen (*INTEGER*, *REAL*, ...) ausgenutzt, wodurch Exemplare dieser Klassen i.A. als Werte übergeben werden.

Warum expanded?

Es stellt sich die Frage, weshalb eine moderne Programmiersprache mit hohen konzeptionellen Ansprüchen im Jahre 1992 [Meyer 92] den gleichen Begriff von Aggregation einführt wie Pascal zwanzig Jahre zuvor. Dazu soll die in [Meyer 97] gelieferte Herleitung genauer betrachtet werden.

Im Punkte der Referenzen auf Subkomponenten gibt sich Meyer offen für Kritik, und räumt ein, daß ihr Verbot die Modellierungsmächtigkeit einschränkt. Er berichtet, daß Referenzen auf Subkomponenten ursprünglich vorgesehen waren, aber "mehr Probleme verursachen, als sie Wert sind" (S.261): Zum einen beeinträchtigen sie die Performanz der Speicherbereinigung, und zum anderen erschweren sie die Modellierung, weil Referenzen sowohl auf Objekte als auch auf Subkomponenten verweisen können.

Das erste Argument rührt daher, daß Eiffel expandierte Objekte durch verschachtelte Speicherbereiche darstellt, und die Speicherbereinigung durch die mögliche Anwesenheit von Referenzen ins Innere eines angeforderten Speicherbereichs gebremst wird. Der Performanzverlust wird als signifikant bezeichnet.

Die durch das zweite Argument kritisierte Komplexität ist manchmal unvermeidlich. Die eingeschränkte Definition von Expansion ist zwar leichter handhabbar, weil sie weniger komplexe Objektstrukturen liefert, sie ist aus demselben Grunde aber auch seltener einsetzbar. Die Vereinfachung wird auch nicht als Ziel beschrieben, sondern nur als angenehme Konsequenz der Einschränkung.

Letztendlich sind Referenzen auf Subkomponenten also aus Implementationserwägungen heraus verboten. Ähnlich sieht es bei der ursprünglichen Einführung des **expanded**-Schlüsselwortes aus. Meyer nennt drei Gründe, weshalb das Sprachmerkmal "nötig" ist:

1. Gesteigerte Effizienz
2. Verbesserte Modellierung durch Umsetzung von Aggregation
3. Unterstützung von Basistypen in einem einheitlichen objektorientierten Typsystem

Zwar behauptet Meyer, daß der wichtigste Grund der zweite, d.h. die Unterstützung der Konzeptebene ist. Wie gezeigt paßt Expansion aber nur sehr eingeschränkt auf Aggregation; zu diesem Ergebnis kommt auch [Kent, Maung 95].

Am besten nachvollziehen läßt sich der erste Grund, nämlich die Implementationsvorteile durch expandierte Variablen. Als Vorteile werden genannt: daß im Gegensatz zu Referenzvariablen beim Zugriff nicht getestet werden muß, ob eine leere Referenz vorliegt; daß für den Zugriff keine Referenz verfolgt (dereferenziert) werden muß; und daß kein zusätzlicher Speicherbedarf für den Zeiger auf die Subkomponente anfällt. Als weiteren Vorteil könnte man nennen, daß der Speicher für alle Subkomponenten in einem einzigen Schritt angefordert werden kann.

Bei dem dritten Punkt geht es darum, die aus Sprachen wie C++ oder Java bekannte Trennung zwischen Werten und Objekten zu vermeiden, und auch Basistypen als Klassen darzustellen.

Anders als in Sather, Smalltalk und Tycoon-2 sind die in Eiffel und BETA für die Darstellung von Basiswerten verwendeten Objekte änderbar. Einer Variable, die mit expandiertem Typ deklariert ist, werden also nicht verschiedene Werte zugewiesen, sondern sie wird fest an ein Exemplar ihres Typs gebunden, dessen Zustand sich ändern kann. Ein Basiswert wird in einem expandierten Objekt gespeichert und nicht durch das Objekt repräsentiert. Die in der Standardbibliothek bereitgestellten Basistyp-Klassen könnten ihre Aufgabe also auch ohne das **expanded**-Attribut erfüllen, wenn auch weniger effizient.

Zusammenfassend kann man sagen, daß die Verwendung von **expanded** den Programmierer mit einer Entscheidung belastet, die auf Performanz und nicht auf Modellierung zielt. Es wäre zu überlegen, ob solche Entscheidungen nicht besser automatisch vom Übersetzer getroffen werden, oder zumindest getrennt von Modellierungsbelangen behandelt werden sollten.

3.1.6. Standard ML

In allen bisher betrachteten Sprachen wird die Zyklenfreiheit der jeweiligen Komponentenbindung dadurch erreicht, daß bei der Erzeugung eines Objekts alle Subkomponenten

sofort miterzeugt werden, und die Struktur im nachhinein nicht mehr änderbar ist. Die so erzeugten Strukturen können jedoch nicht polymorph, insbesondere nicht rekursiv sein.

Um zu demonstrieren, daß auch polymorphe Datenstrukturen beliebiger Größe durch eine zyklenfreie Beziehung zwischen Objekten ausgedrückt werden können, soll die funktional-imperativen Sprache Standard ML (kurz SML) betrachtet werden. Zum Einstieg wird das obige Pascal- bzw. BETA-Beispielprogramm direkt umgesetzt, auch wenn das Ergebnis etwas unidiomatisches SML ist.

```
type Linie = { x1:int ref,y1:int ref,x2:int ref,y2:int ref };
type Arm = { oberarm:Linie, unterarm:Linie };
type Bein = { oberschenkel:Linie, unterschenkel:Linie, fuss: Linie };
type Strichmann = { rechterArm:Arm, linkerArm:Arm,
                    rechtesBein:Bein, linkesBein:Bein };
fun dreh(eineLinie: Linie, alpha) = ...;
fun anwinkeln(einArm: Arm) = dreh(#unterarm einArm, pi/4);
fun neuLinie() = { x1=ref 0, y1=ref 0, x2=ref 0, y2=ref 0 };
fun neuArm() = { oberarm=neuLinie(), unterarm=neuLinie() };
...
fun test() =
  let val axel = neuStrichmann() in
    ...; anwinkeln(axel.rechterArm); ...
  end;
```

Die Blätter der Datenstruktur (die ganzzahligen Koordinaten in *Linie*) wurden als änderbare Speicherzellen (**ref**) umgesetzt, alle anderen Elemente als nicht-änderbare zusammengesetzte Typen (genauer *records*). Dies entspricht der Umsetzung in den bisher beschriebenen Sprachen. Speicherzellen in SML erfüllen die Rolle der Zeiger oder dynamischen Referenzen: Der durch den Inhalts-Operator (*!*) erfragte Wert kann durch Zuweisung (:=) im Verlauf des Programms geändert werden. Durch die Einführung von Speicherzellen erlaubt SML auch den hier verwendeten imperativen Stil, und erlaubt auch erst die Erklärung von Objektidentität.[2]

Für das rekursive Beispiel, die Liste von Strichmännchen, sind im Gegensatz zu den bisher beschriebenen Sprachen keine Speicherzellen nötig. Die Liste läßt sich als rekursiver, nicht-änderbarer Datentyp formulieren:[3]

[2] Objektidentität in SML in diesem Sinne ist fälschbar, da verschiedene zusammengesetzte Objekte aus denselben Bestandteilen bestehen können. Der folgende Code erzeugt zum Beispiel zwei nicht unterscheidbare Linien, die dennoch verschiedenen Konstruktor-Ausdrücken entspringen.

```
local val rx1=ref 0 and ry1=ref 0 and rx2=ref 0 and ry2=ref 0 in
  val l1={x1=rx1,y1=ry1,x2=rx2,y2=ry2}
  and l2={x1=rx1,y1=ry1,x2=rx2,y2=ry2}
end
```

[3] Während Pascal Polymorphie (**A4**) über Trennbarkeit (**A6**) simulieren mußte, wird in SML Trennbarkeit mit Hilfe der Polymorphie simuliert, wie hier durch die Variante *keineLeute*.

```
datatype Strichleute =
    keineLeute
  | leute of Strichmann * Strichleute ;
```

Die beliebige Größe wird dadurch erreicht, daß die Werte der nicht-änderbaren Bindungen zur Erzeugungszeit angegeben werden, etwa wie folgt:

```
val doppelaxel = leute(axel,leute(axel,keineLeute));
```

Wie man sieht, können dabei existierende Objekte mehrfach eingetragen werden. Zyklische Strukturen sind so allerdings nicht möglich, da Verweise nur von jüngeren zu älteren Objekten verlaufen; der aufgebaute Graph ist also in jedem Fall ein gerichteter azyklischer Graph (GAG).

SML bietet ein Konstrukt für rekursive Bindung von Funktionen (**val rec**), dies ist jedoch nicht auf Datenstrukturen anwendbar. Um zyklische Strukturen zu erreichen, sind wie bisher Speicherzellen erforderlich.

```
datatype Strichleute =
    keineLeute
  | leute of Strichmann * Strichleute ref;
let val ringelrein = ref keineLeute in
  ringelrein :=
    leute(neuStrichmann(),
      ref(leute(neuStrichmann(), ringelrein)));
  !ringelrein
end
```

Zusammenfassend kann man sagen, daß durch die Kombination von Konstruktoren und nicht-änderbaren zusammengesetzten Typen eine zyklenfreie (**A2**), rekursiv polymorphe (**A4**) Komponentenbindung erreicht wird. Da beliebige Objekte als Argument eines Konstruktors verwendet werden können, kann ein Objekt jedoch mehrere Superkomponenten haben (**A1**).

3.1.7. Aggregation in einem OODBMS

Aggregation im Kontext einer objektorientierten Datenbank wird in [Kim et al. 87] anhand des Prototyps ORION beschrieben. Die Autoren beschreiben, wie Aggregation zur Steigerung von Referenzlokalität (*clustering*), für die Versionierung (*versioning*) und für die Synchronisation (*locking*) nutzbringend eingesetzt werden kann, und betrachten Aggregation auch in Hinsicht auf Schemaevolution.

Das verwendete Datenmodell erlaubt die Auszeichnung von Exemplarvariablen, die Komponentenbindungen (dort *composite link*) enthalten. Durch diese Komponentenbindungen werden die Objekte der Datenbank zu Komponentenhierarchien (*composite object hierarchies*) zusammengefaßt. Der Komponentengraph bildet damit wie gefordert einen Wald, d.h. jedes Objekt hat höchstens eine Superkomponente, und der Graph enthält keine Zyklen.

Subkomponenten werden nicht automatisch miterzeugt. Komponentenbindungen sind ursprünglich leer, können aber später mit einer Bindung an ein neu erzeugtes Objekt gefüllt werden. Die restlichen Operationen, die eine Komponentenbindung ändern, sind das Löschen des enthaltenen Objekts und der Austausch zwischen zwei Exemplarvariablen. Dadurch ist sichergestellt, daß jedes Objekt höchstens eine Superkomponente hat. Das Kriterium der Sichtbarkeit ist ebenfalls erfüllt, da von außerhalb der Komponentenhierarchie beliebige Referenzen auf Subkomponenten möglich sind.

Dem Text ist nicht klar zu entnehmen, ob bei der Erzeugung einer Subkomponente auch eine andere als die deklarierte Klasse der Variablen verwendet werden kann; werden Subkomponenten immer als Exemplar der deklarierten Klasse erzeugt, so ist Polymorphie nur über Trennbarkeit darstellbar.

Selbst wenn man davon ausgeht, daß Subkomponenten polymorph erzeugt werden, erhält man nicht die gewünschte rekursive Polymorphie. Diese wird in dem Vorschlag von Kim et al. dadurch verhindert, daß die Autoren auch eine hierarchische Struktur der beteiligten Klassen fordern.

Da Pfade im Komponentengraph also durch die Länge der Pfade im Klassen-Komponentengraph begrenzt sind, kann der Komponentengraph keine Zyklen enthalten, und somit ergibt sich Anforderung **A2**.

Im Punkte des kaskadierten Löschens (**A9**) weicht der Vorschlag von Kim et al. wieder von den Anforderungen ab. Die Subkomponenten werden generell als abhängige Objekte (*dependent objects*) bezeichnet, und werden gelöscht, wenn ihre Superkomponente gelöscht wird. Eventuell noch existierende Referenzen auf die gelöschten Objekte werden dabei nicht berücksichtigt.

Von den untersuchten Systemen kommt die Aggregation bei Kim et al. der gesuchten Komponentenbindung noch am nächsten: Die Bindungsart ist exklusiv (**A1**), zyklenfrei (**A2**), schränkt die Sichtbarkeit nicht ein (**A3**) und ist dabei immernoch änderbar und trennbar (**A5, A6, A7**).

3.1.8. Syntaktische Aspekte

Vergleicht man die Syntax der Dualität von Referenz und Komposition in den diskutierten Sprachen, so fällt auf, daß Pascal ($\uparrow$), C ($*$) und SML (**ref**) die flexiblere, aber komplizierter zu handhabende Referenzbindung im Gegensatz zu Komponentenbindung syntaktisch auszeichnen. Der Normalfall ist dort also die Komponentenbindung. Dies hängt auch damit zusammen, daß Referenzen in diesen Sprachen in eigenen Objekten gespeichert werden, und jeder Zugriff als Operation gewertet wird. Speicher- und Laufzeitkosten werden dem Programmierer durch syntaktische Kosten sichtbar gemacht.

Eiffel hingegen nimmt mit dem Schlüsselwort **expanded** den umgekehrten Weg: Die Komponentenbindung wird als komplizierte Einschränkung der Referenzbindung behandelt und bekommt der Komplexität entsprechend einen höheren syntaktischen Preis. In

Eiffel kommt hinzu, daß das Konzept der expandierten Objekte in der ersten Version der Sprache [Meyer 88] noch nicht existierte, und aus Kompatibilitätsgründen Programme in der bisherigen Syntax ihre Semantik behalten sollten.

In BETA wird keine der beiden Bindungsarten syntaktisch bevorzugt, der Programmierer muß sich immer zwischen dynamischer (↑) und statischer (@) Referenz entscheiden. Diese Gleichberechtigung erleichtert eine Orientierung an der konzeptuellen Ebene.

3.2. Gruppierung von Objekten

Im vorangegangenen Abschnitt wurden Bindungen zwischen Objekten betrachtet, die sich auch durch die in der Sprache dafür verfügbaren Operationen unterschieden. Die Bindungsarten wurde daraufhin untersucht, inwieweit sie den Anforderungen an Komponentenbindung entsprechen, um so dynamische Komponenten umsetzen zu können. In diesem Abschnitt soll von der anderen Seite betrachtet werden, wie existierende Systeme die Gruppierung von Objekten zu logischen Einheiten handhaben.

Dabei finden sich auch sprachlich ausgezeichnete Variablendeklarationen, welche die in der Programmiersprache möglichen Operationen auf der Variable nicht beeinflussen. Dazu gehören z.B. das Schlüsselwort **attached** der Programmiersprache Emerald, und das Schlüsselwort **transient** in Java, die als Attribut einer Variablendeklaration angegeben werden können. Der Sinn solcher Attribute liegt darin, dem Laufzeitsystem zusätzliche Information zur Verfügung zu stellen.

Die Bindungsarten treten nicht gesondert in der Tabelle 3.1 auf, da ihre Zeilen sich nicht von der Referenzbindung in der jeweiligen Sprache unterscheiden würden.

Auch das Schlüsselwort **separate** der Programmiersprache Eiffel soll in diesem Abschnitt betrachtet werden, da sich die ausgedrückte Semantik schlecht auf die definierten Anforderungen abbilden läßt.

3.2.1. Emerald und Verteilung

Emerald [Jul 89] ist eine rein objektorientierte, statisch typisierte Programmiersprache, die für verteilte Anwendungen entworfen wurde. Emerald kennt keine Vererbung von Implementationen, nur Subtypisierung zwischen Objektschnittstellen. Klassen im üblichen Sinne gibt es nicht: Objekte werden durch einen **object**-Ausdruck erzeugt, der Repräsentation und Verhalten des zu erzeugenden Objekts vollständig beschreibt. **object**-Ausdrücke können geschachtelt werden und gehorchen dann statischen Sichtbarkeitsregeln.

Alle Objekte in Emerald sind potentiell mobil, d.h. sie können sich mit Hilfe spezieller Sprachprimitive zwischen verschiedenen Rechnerknoten bewegen. Jede Objektreferenz kann daher transparent auf ein entferntes Objekt verweisen, und jeder Aufruf kann so zu einem entfernten Prozeduraufruf (*remote procedure call*) werden.

Da die Kommunikation mit entfernten Objekten aber um Größenordnungen teurer ist als die mit lokalen, ist es wünschenswert, eng kooperierende Objekte auf dem gleichen Rechnerknoten zu positionieren. Aus technischen Gründen ist es außerdem effizienter, mehrere Objekte auf einmal an einen anderen Knoten zu übermitteln. Gruppen eng kooperierender Objekte sollen daher nach Möglichkeit gemeinsam migrieren. Emerald verwendet dafür den Begriff der Migrations-Gruppe (*move group*).

Der Übersetzer des Emerald-Systems stellt unter gewissen Umständen automatisch fest, wenn ein Objekt nur innerhalb eines anderen Objekts verwendet wird. Solche lokalen Objekte (*local objects*) migrieren immer gemeinsam mit ihrem umgebenden Objekt. Darüber hinaus wertet Emerald aber auch Hinweise des Programmierers aus: Variablen können mit dem Attribut **attached** deklariert werden. Enthält eine solche Variable eines Objekts a eine Referenz auf ein Objekt b, und soll a migrieren, so migriert b ebenfalls. b ist also Mitglied der Migrations-Gruppe von a. Diese Beziehung ist transitiv: Die Migrations-Gruppe von b ist daher eine Teilmenge der Migrations-Gruppe von a. Da einer **attached**-Variablen wie jeder anderen ein neuer Wert zugewiesen werden kann, kann sich die Migrations-Gruppe über die Zeit dynamisch ändern.

Interessanterweise ist *attachment* nicht symmetrisch, was der intuitiven Bedeutung von *Befestigung* widerspricht. Ein befestigtes Objekt kann auch ohne das befestigende Objekt migrieren. Dadurch kann es vorkommen, daß sich die Mitglieder einer Migrations-Gruppe auf verschiedenen Rechnerknoten befinden; es steht zu vermuten, daß entfernte Referenzen bei der Bestimmung der Migrations-Gruppe nicht weiterverfolgt werden.

Ein weiterer interessanter Punkt ist, daß durch Zuweisung von Referenzen ein Objekt an zwei verschiedenen anderen Objekten befestigt sein kann. Das befestigende Objekt, das zuerst migriert, nimmt das befestigte dann mit sich. Dabei handelt es sich nicht um einen Programmierfehler, denn ein Objekt kann ja tatsächlich eng mit zwei verschiedenen anderen zusammenarbeiten, und die Entscheidung, bei welchem es sicht aufhalten soll, kann i.A. gar nicht optimal gefällt werden.

Daß der durch die Befestigungs-Relation gegebene Graph Zyklen enthalten kann, stellt hingegen kein Problem dar. Die Rekurrenz, die die Migrationsgruppe definiert, ist auch in der Gegenwart von Zyklen leicht auflösbar.

Aus Befestigung folgt also nicht Lokalität. Insofern ist Befestigung ein Hinweis an das Laufzeitsystem, der heuristisch Effizienzsteigerung erlaubt, aber keine Invarianten definiert.

3.2.2. DOWL und lokale Variablen

Einen interessanten Aspekt bringt die verteilte Programmiersprache DOWL ein (Distributed OWL, [Achauer 93]). Da in DOWL wie in Emerald alle Aktivierungen eines Objektes mit diesem gemeinsam migrieren, ist es konsequent, Befestigung auch für die in einer Aktivierung gebundenen Parameter und lokalen Variablen zu spezifizieren. DOWL

erlaubt daher, das Schlüsselwort **attached** auf sämtliche Arten von Variablen anzuwenden. Zusätzlich werden die Schlüsselworte **move** und **visit** eingeführt. Bei der Zuweisung (oder Argumentübergabe!) an eine derart gekennzeichnete Variable migriert das zugewiesene Objekt an den aktuellen Ort der Variable. Man könnte sagen, die Zuweisung verwandelt sich von einer einfachen Kopie in eine Bewegung.

Genau wie **attached** machen auch die neu eingeführten Schlüsselworte keine feste Aussage über die Lokalität ihres gebundenen Objekts. Zwar wird das Objekt bei der Zuweisung zur Variablen bewegt, kann danach aber unabhängig von der Variablen den Ort wechseln.

Was passiert, wenn die Lebenszeit einer Variable vorüber ist? Im Falle einer **move**- oder **attached**-Variablen bleibt das Objekt, wo es ist. Im Falle einer **visit**-Variablen bewegt es sich jedoch zum vorherigen Rechnerknoten zurück. Dadurch wird eine Verknüpfung von Lokalität und *block scoping* erreicht.

Festzuhalten ist, daß die Unterscheidung verschiedener Bindungsarten für alle Variablen, nicht nur für Exemplarvariablen gelten kann; und daß Zuweisung unter gewissen Umständen auch die Änderung von Lokalität bedeuten kann.

3.2.3. Java und Persistenz

Wie das Attribut **attached** in Emerald der Verteilung diente, so wird das Attribut **transient** in Java für Persistenz genutzt. Persistenz wird in Java durch Serialisierung erreicht: Der aktuelle Zustand eines Objektgraphen wird bei der Serialisierung in eine lineare Folge von Bytes umgewandelt. In dieser serialisierten Form kann der Objektgraph beliebig lange aufbewahrt (und auch dupliziert und übermittelt) werden. Aus der serialisierten Form kann dann ein Objektgraph konstruiert (deserialisiert) werden, der i.A. isomorph zum ursprünglich serialisierten ist. Das heißt, die relative Identität der Objekte innerhalb der serialisierten Objektmenge bleibt erhalten.

Dabei ist wieder zu entscheiden, welche Objekte gemeinsam serialisiert werden sollen. Probleme entstehen sowohl, wenn diese Menge zu klein ist, als auch wenn sie zu groß ist.

Wird ein Objekt ausgeschlossen, so kann nämlich nicht einfach wie bei der Migration eine entfernte Referenz auf das außerhalb liegende Objekt gespeichert werden. Im Fall der Migration führt eine zu kleine Migrationsgruppe lediglich zu Ineffizienz, da nicht migrierte Objekte entfernt angesprochen oder durch zusätzliche Schritte nachgeholt werden können. Solange alle Rechnerknoten erreichbar sind, ist die Funktionalität nicht beeinträchtigt.

Bei der Serialisierung aber muß eine Referenz auf ein außerhalb liegendes Objekt durch eine Beschreibung des geforderten Objektes dargestellt werden, denn textuell läßt sich ein Objekt immer nur relativ zu einem Bezugspunkt identifizieren (z.B. in einem Objektspeicher, oder über einen Namensdienst). Es muß daher genug Information serialisiert werden oder durch den Kontext bekannt sein, um bei der Deserialisierung ein geeignetes,

evtl. anderes Objekt auszuwählen oder neu zu erzeugen (Assoziation statt Referenz). In Java ist es immer Aufgabe des Programmierers, den zugehörigen Code zu verfassen, und dies ist teuer.

Ist die Menge der gemeinsam serialisierten Objekt hingegen zu groß, dann wird ein Objekt, das an zwei Serialisierungen beteiligt ist, bei der Deserialisierung unerwünscht dupliziert. Dies ist erstens dann unangenehm, wenn die Funktionalität der beiden serialisierten Gruppen davon abhängt, daß sie das gleiche Objekt referenzieren, etwa weil sie darüber kommunizieren; zweitens dann, wenn das Objekt und der daran hängende Objektgraph sehr groß sind, da dadurch unnötig Speicherplatz verbraucht wird; und drittens dann, wenn sich der Zustand des referenzierten Objekts zwischen den Serialisierungen geändert hat, denn dadurch arbeitet ein Teil der deserialisierten Objekte auf veralteten Daten. Als Beispiel betrachte man ein Modell eines Betriebs, in dem jeder Angestellte einen Verweis auf seinen Vorgesetzten, und jeder Vorgesetzte Verweise auf seine Untergebenen hat. Würde diesen Verweisen bei der Serialisierung gefolgt, so wäre es unmöglich, einen Angestellten persistent zu speichern, ohne zugleich alle zu speichern, mit allen erwähnten Nachteilen.

Java bietet mehrere Mechanismen an, um dem Programmierer Kontrolle über die Menge der zu serialisierenden Objekte zu geben. Durch Implementation der Schnittstelle *Externalizable* können die zu verfolgenden Referenzen durch vom Programmierer geschriebenen Code zur Laufzeit berechnet werden. Dies erlaubt größtmögliche Flexibilität, erfordert aber auch den größten Aufwand. Durch Implementation der Schnittstelle *Serializable* werden automatisch alle nicht-flüchtigen Referenzen verfolgt. Hier kommt das Schlüsselwort **transient** ins Spiel: Eine Referenz gilt als flüchtig, wird also nicht automatisch verfolgt, wenn sie in einer mit dem Schlüsselwort **transient** versehenen Variable gespeichert ist. Der Programmierer hat die Möglichkeit, durch Implementation der Methoden *writeObject* und *readObject* zu den automatisch geschriebenen Informationen eigene hinzuzufügen, etwa um bei der Deserialisierung die flüchtigen Verweise rekonstruieren zu können.

Man beachte, daß die Rolle von **transient** genau umgekehrt zu der von **attached** ist: **transient** schließt Objekte aus, während **attached** Objekte aufnimmt. Im Normalfall, wenn also kein Attribut angegeben wird, gehen beide Ansätze damit in die sichere Richtung: Emerald migriert nicht zu viele Objekte, und Java serialisiert genug Objekte, um den Objektgraph auch ohne Hilfe des Programmierers rekonstruieren zu können.

Im Vergleich mit Emerald und DOWL ist zu bemerken, daß nur Objekte, aber nie Aktivierungen serialisiert werden. Damit ist das Schlüsselwort **transient** nur für Exemplarvariablen definiert und sinnvoll. Dies deutet auch auf eine Schwäche des Persistenzmechanismus hin: Ist ein Objekt gerade in Bearbeitung, während es serialisiert wird, so ist der serialisierte Zustand evtl. inkonsistent. Auf jeden Fall wird der Bearbeitungszustand nicht serialisiert und daher auch nicht automatisch wiederhergestellt, wenn das Objekt deserialisiert wird.

3.2.4. Eiffel und Prozessoren

Die Programmiersprache Eiffel ist schon einmal in Abschnitt 3.1.5 untersucht worden, dort in Hinblick auf das Schlüsselwort **expanded**. In diesem Abschnitt soll ein anderer Aspekt betrachtet werden, ausgedrückt durch ein anderes Schlüsselwort: **separate**.

Dieses Schlüsselwort ist der syntaktisch sichtbare Teil eines ehrgeizigen Versuches, die Probleme von Verteilung und Nebenläufigkeit auf radikale Weise in den Griff zu bekommen. Jedes Objekt ist einem *Prozessor* zugeordnet, der eine Abstraktion eines physikalischen Prozessors darstellt. Prozessoren sind zum einen die Einheit der Nebenläufigkeit, insofern als es pro Prozessor nur einen *thread* zur Zeit geben kann. Zum anderen bilden Prozessoren auch die Einheit der Verteilung, da sich alle Objekte eines Prozessors auf demselben Rechnerknoten befinden.

Der Prozessor eines Objekts wird unveränderlich festgelegt, wenn das Objekt erzeugt wird. Entweder, das Objekt wird lokal in dem Prozessor angelegt, zu dem auch das erzeugende Objekt gehört, oder das neue Objekt wird entfernt in einem für diesen Zweck angelegten Prozessor erzeugt.

Durch die An- oder Abwesenheit von **separate** wird zwischen lokalen und separaten Referenzen unterschieden, wobei sichergestellt ist, daß lokale Referenzen nur zwischen Objekten desselben Prozessors verlaufen können. Dadurch ergibt sich eine Ähnlichkeit zu dynamischen Komponenten: Die lokale Referenz entspricht der Komponentenbindung, während die entfernte Referenz der Referenzbindung entspricht.

Man beachte jedoch, daß die Zugehörigkeit zu einem Prozessor nicht durch die aktuellen Komponentenbindungen bestimmt wird, sondern ein für alle mal feststeht. Es ist daher sehr wohl möglich, daß ein Prozessor zwei disjunkte Komponentengraphen enthält, ohne daß er sich dadurch in zwei Prozessoren aufspalten würde. Sollen Datenstrukturen zwischen Prozessoren kommuniziert werden, so geschieht dies per Kopie, Migration ist nur auf der Basis ganzer Prozessoren denkbar.

Die so definierte Komponentenbindung ist zwar auf der Ebene der einzelnen Variablen änderbar, die Aufteilung in dynamische Komponenten ist aber unveränderlich. Auch sind diese Komponentenbindungen innerhalb eines Prozessors rekursiv polymorph, es ergibt sich jedoch nicht die gewünschte hierarchische Strukturierung ineinander verschachtelter dynamischer Komponenten. Die Prozessoren befinden sich gleichberechtigt und ortstransparent auf beliebigen Rechnerknoten und bilden so im Gegensatz zu dynamischen Komponenten eine flache Struktur. Der Grund dafür liegt darin, daß ein Prozessor kein Objekt ist, in den Begriffen von Abschnitt 2.1 also nur als System, aber nie als Komponente auftritt.

Im Vergleich zu Emerald und DOWL fällt auf, daß die Komponentenbindung nun tatsächlich eine harte Aussage über Lokalität beinhaltet. Im Unterschied zu diesen verteilten Programmiersprachen ist in Eiffel aber Lokalität der Normalfall. Der Grund ist wie schon beim Schlüsselwort **expanded** historisch, denn die ursprüngliche Fassung von Eiffel [Meyer 88] kennt weder Nebenläufigkeit noch Verteilung. Die in dieser einfachen

Situation möglichen Zusicherungen lassen sich durch die abwärtskompatible Erweiterung auch auf verteilte und nebenläufige Umstände übertragen, wodurch die Produktion korrekter Programme unterstützt werden soll. Meyer nimmt dabei in Kauf, daß die resultierenden Programme weniger nebenläufig sind als vielleicht möglich.

3.3. Kapselung und Schutz vor Mehrfachbenennung

In diesem Abschnitt sollen einige in der Literatur vorgeschlagene Konzepte betrachtet werden, die bis jetzt noch nicht den Weg in eine verfügbare Programmiersprache gefunden haben.

In der Literatur der letzten Jahre finden sich verschiedene Ansätze, die die Mehrfachbenennung (*aliasing*) unter Kontrolle bringen sollen, ohne gängige objektorientierte Idiome und Entwurfsmuster übermäßig einzuschränken [Hogg 91; Minsky 96; Almeida 97; Noble et al. 98].

Mehrfachbenennung wird als Problem angesehen, da sie der modularen Struktur des Quelltextes zuwiderläuft. Einem Teil des Programmtextes ist i.A. nicht anzusehen, von welchen anderen Teilen aus zur Laufzeit Verweise darauf bestehen können. Dadurch ist es möglich, daß sich Objekte "hinter dem Rücken" einer Klasse ändern, was besonders im Zusammenhang mit Nebenläufigkeit zu Inkonsistenzen führen kann.

Ziel der genannten Ansätze ist es daher, auch auf der dynamischen Ebene eine modulare Struktur zu erreichen, die gewisse Zugriffsbeschränkungen garantiert. Objekte werden zu Gruppen zusammengefaßt, die jeweils *island* [Hogg 91], *balloon* [Almeida 97] oder *shadow* [Noble et al. 98] genannt werden. Hier sieht man die Parallele zu dynamischen Komponenten.

Ein weiterer Bezug zwischen Komponentenbindungen und Mehrfachbenennung liegt in der Forderung, daß jedes Objekt höchstens eine Superkomponente hat (**A1**). Das bedeutet gerade, daß es in Bezug auf Komponentenbindungen keine Mehrfachbenennung gibt. Die beschriebenen sprachlichen Mechanismen könnten also auch auf dynamische Komponenten übertragbar sein.

Im Unterschied zu dynamischen Komponenten werden in den zitierten Artikeln jedoch keine operationalen Möglichkeiten erschlossen. Die Objektgruppe wird rein deklarativ beschrieben, und dient ausschließlich der statischen Programmanalyse. Andererseits drücken dynamische Komponenten keinerlei Beschränkung der Sichtbarkeit aus, und sind damit nicht direkt zur Umsetzung von Kapselung geeignet. Eine Kombination wäre als Gegenstand zukünftiger Untersuchungen also denkbar.

Im folgenden werden die einzelnen Ansätze betrachtet.

3.3.1. Inseln und Brücken

Der erste betrachtete Artikel [Hogg 91] führt den Begriff des *island* ein. Ein *island* besteht aus allen von seinem primären Objekt, der *bridge*, transitiv über Exemplarvariablen erreichbaren Objekten. Hogg definiert eine Spracherweiterung, die sicherstellt, daß kein Objekt außerhalb des *island* eine Referenz auf ein Objekt innerhalb des *island* in einer Exemplarvariable speichern kann, so daß ein *island* die enthaltenen Objekte kapselt.

Eine in einer Exemplarvariable gespeicherte Referenz nennt Hogg *static*. Referenzen, die in einer lokalen Variablen gespeichert werden, Empfänger oder Argument einer Nachricht sind oder als Ergebnis auftreten, werden *dynamic* genannt. Da diese Begriffe den in dieser Arbeit verwendeten zuwiderlaufen, soll stattdessen von flüchtigen und nicht-flüchtigen Referenzen gesprochen werden. Die Kapselung der *islands* kann dann wie folgt beschrieben werden: Kein Objekt außerhalb des *island* kann eine nicht-flüchtige Referenz auf ein Objekt innerhalb des *island* halten. Flüchtige Referenzen von außen sind hingegen erlaubt, erlauben aber nur lesenden Zugriff.

Um dies zu erreichen, führt Hogg insgesamt sechs verschiedene Zugriffsmodi für Variablen ein. Ein Zugriffsmodus wird definiert als eine statisch prüfbare Einschränkung der auf einer Variable möglichen Operationen. Die Zugriffsmodi werden so gewählt, daß sie eine Aussage über die Anzahl nicht-flüchtiger Referenzen auf das referenzierte Objekt machen: Keine nicht-flüchtige Referenz (**free**)[4], höchstens eine (**unique**) oder beliebig viele (uneingeschränkt). Zusätzlich sind diese drei Modi jeweils in Kombination mit einem Schreibschutz-Attribut (**read**) möglich.

Um die Anzahl nicht-flüchtiger Referenzen sicherzustellen, wird eine spezielle Operation eingeführt: Variablen können destruktiv gelesen werden. Die enthaltene Referenz wird dabei aus der Variablen herausgenommen, d.h. die Variable ist danach leer (**nil**). Die Anzahl Referenzen auf das enthaltene Objekt bleibt dadurch gleich.

Wird der Inhalt einer **free**-Variable an eine Exemplarvariable zugewiesen, steigt die Anzahl statischer Referenzen auf eins. Die Exemplarvariable muß also als **unique** deklariert sein. Keine **free**-Variable darf mehr eine Referenz auf das zugewiesene Objekt enthalten, weshalb sichergestellt sein muß, daß der Zugriff auf **free**-Variablen immer durch destruktives Lesen erfolgt.

Umgekehrt kann eine **unique**-Exemplarvariable destruktiv gelesen werden, was zu einem Ergebnis ohne nicht-flüchtige Referenzen führt (**free**). Wird eine beliebige **unique**-Variable nicht-destruktiv gelesen, bleibt der Modus bei **unique**. Eine **unique**-Referenz darf nicht an eine uneingeschränkte Variable gebunden oder zugewiesen werden.

Die Modi werden nun verwendet, um zu definieren, Exemplare welcher Klassen als *bridge* gelten sollen. Zunächst wird gefordert, daß in der Schnittstelle des Klasse nirgends der

[4] Das Kriterium bei Hogg lautet, daß eine **free**-Referenz die einzige Referenz auf das Objekt im System ist. In Verbindung mit dem destruktiven Lesen von **unique**-Variablen wird dieses Kriterium jedoch nicht eingehalten; die Betrachtung in dieser Arbeit kommt aber auch mit dem schwächeren Kriterium aus.

uneingeschränkte, schreibbare Modus verwendet wird, weder als Empfänger, noch als Parameter oder Ergebnis. Um die Kapselung zu garantieren, wird zusätzlich allgemein gefordert, daß über **unique**-Referenzen keine uneingeschränkten Referenzen übermittelt werden.

Hogg weist darauf hin, daß Zugriffsmodi orthogonal zu Typen sind. Um dies zu demonstrieren, definiert er seine Erweiterung als eine Variante der nicht statisch typisierten Programmiersprache Smalltalk. In Smalltalk ist statisch unbekannt, welche Nachrichten ein referenziertes Objekt versteht. Bei jedem Methodenaufruf muß daher geprüft werden, ob der Empfänger tatsächlich eine Methode mit dem angegebenen Selektor zur Verfügung stellt. Insbesondere ist daher auch nicht bekannt, welche Parameter-Modi die gefundene Methode definiert, so daß ein Test nötig wird, um inkompatible Modi beim Aufruf zu verhindern. Hogg schlägt vor, die Parameter-Modi als Teil des Methodenselektors aufzufassen, und fehlerhafte Modi mit den gleichen Laufzeitmitteln wie fehlende Methoden zu behandeln (`doesNotUnderstand`). Die Kodierung der Methodensignatur im Selektor findet sich auch im Bytecode von Java [Gosling et al. 96] und wird dort für die Typprüfung beim Aufruf spät gebundener Methoden verwendet.

Der Artikel [Hogg 91] liefert also drei Beiträge zur Umsetzung von dynamischen Komponenten: Zum ersten die Erkenntnis, daß durch destruktives Lesen Aussagen über die Anzahl möglicher Referenzen auf ein Objekt getroffen werden können; zum zweiten, daß dazu auch Zugriffsmodi für flüchtige Variablen nötig sind; und zum dritten, daß der in objektorientierten Sprachen vorhandene Mechanismus des späten Bindens auch für Laufzeitprüfungen eingesetzt werden kann. Die scharfe Trennung zwischen flüchtigen und nicht-flüchtigen Variablen wird von Hogg nur eingeführt, um zumindest temporäre Referenzen in das Innere der Insel zu ermöglichen. Da dynamische Komponenten die Sichtbarkeit nicht einschränken, ist diese Entwurfsentscheidung nicht übertragbar.

3.3.2. Einzige Zeiger

Das Prinzip des destruktiven Lesens findet sich auch in [Minsky 96], dort unter dem Namen *unique pointer*. Wird eine Variable als *unshareable* deklariert, so ist garantiert, daß es im System keine weitere Referenz auf das referenzierte Objekt gibt. Jeder Zugriff auf die Variable führt zu einem destruktiven Lesezugriff. Minsky spricht davon, daß die Referenzen bewegt und und nicht kopiert werden.

Flüchtige Variablen nehmen auch bei Minsky eine Sonderstellung ein. Parameter können als *non-consumable* deklariert werden, ähnlich dem **unique** von Hogg. Auf ein *unshared object* dürfen zusätzlich zu dem *unique pointer* beliebig viele Referenzen aus solchen Parametern existieren. Wird ein *unique pointer* an einen solchen Parameter gebunden, entfällt daher das destruktive Lesen. Durch zusätzliche Regeln ist sichergestellt, daß der Parameter nur an weitere *non-consumable* Parameter weitergegeben wird.

Minsky stellt auch einen Bezug zur Speicherverwaltung her. Wenn ein *unique pointer* verloren geht, etwa weil er überschrieben wird oder weil die Lebenszeit seiner enthaltenden Variable zu Ende ist, so kann laut Minsky das referenzierte Objekt sofort freigegeben

werden. Die Freigabe überträgt sich rekursiv auf alle als *unshareable* deklarierten Exemplarvariablen des freigegebenen Objekts. Unklar ist, was mit den verbleibenden Referenzen aus *non-consumable*-Parametern geschieht: Wird die Freigabe wie angegeben durchgeführt, so werden diese zu baumelnden Zeigern (*dangling pointers*).

Weiter behauptet Minsky, daß sich durch *unique pointers* Mehrfachbenennung vermeiden ließe. Wie von [Noble et al. 98] erkannt beschränkt sich diese Kapselung jedoch auf die erste Ebene: Zwar kann kein weiterer Verweis auf das *unshareable object* selbst existieren, evtl. vorhandene interne Datenstrukturen sind jedoch nicht geschützt (Abbildung 3.1).

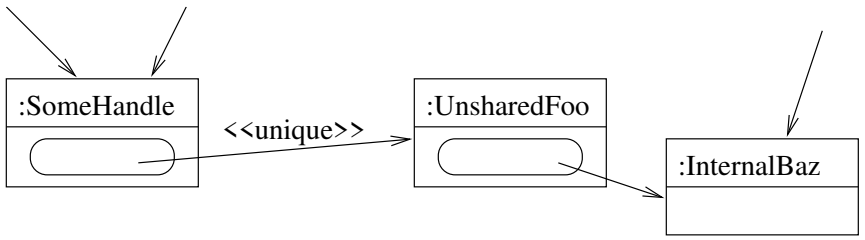

Abbildung 3.1.: Mehrfachbenennung trotz *unique pointer*

Auf eine andere Art demonstriert Minsky selbst, wie sich trotz *unique pointer* Mehrfachbenennung erreichen läßt. Indem der *unique pointer* in einem frei referenzierbaren Objekt (*handle*) gespeichert wird und diese *handle* herumgereicht wird, können beliebige Programmteile auf das *unshared object* zugreifen. Ein gewisser Schutz ist dadurch gegeben, daß alle Zugriffe durch das *handle*-Objekt stattfinden.

Kurz erwähnt Minsky noch das Zusammenspiel mit der Kopieroperation. Wird ein Objekt dupliziert, das einen *unique pointer* enthält, so entständen bei naiver Umsetzung zwei Referenzen auf das *unshared object*, dies darf aber nicht sein. Minsky nennt drei Auswege: 1. Der *unique pointer* wird im Quellobjekt belassen, und im Zielobjekt wird **nil** eingetragen, oder 2. Der *unique pointer* wird ins Zielobjekt verschoben, oder 3. die Operation wird komplett verboten. Eine weitere, bei Minsky nicht genannte Lösung liegt jedoch nahe. Wird eine tiefe Kopie angelegt, d.h. werden alle transitiv erreichbaren Objekt gemeinsam kopiert, so bleibt die Invariante der *unique pointer* erhalten. Hier ist also gar keine besondere Behandlung nötig. Im Falle der flachen Kopie könnte man dieses Verhalten annähern, indem alle über *unique pointer* erreichbaren Objekte mitkopiert werden (Abbildung 3.2).

In Abschnitt 3.1.7 wurde eine Austausch-Operation beschrieben, die sicherstellt, daß höchstens eine Komponentenbindung an ein Objekt existiert, und sich somit als Alternative zum destruktiven Lesen anbietet. Auch Minsky betrachtet mit Verweis auf [Harms, Weide 91] eine solche Operation, stellt jedoch fest, daß der Austausch als symmetrische Operation nicht zu der in objektorientierten Sprachen bei Zuweisung üblichen Subtypisierung paßt. Da der deklarierte Typ einer Variable nur Mindestanforderungen an den zur Laufzeit enthaltenen Wert stellt, dürfen üblicherweise auch Objekte zugewiesen werden, die einem Subtyp entsprechen. Der Ausdruck auf der rechten Seite der Zuweisung

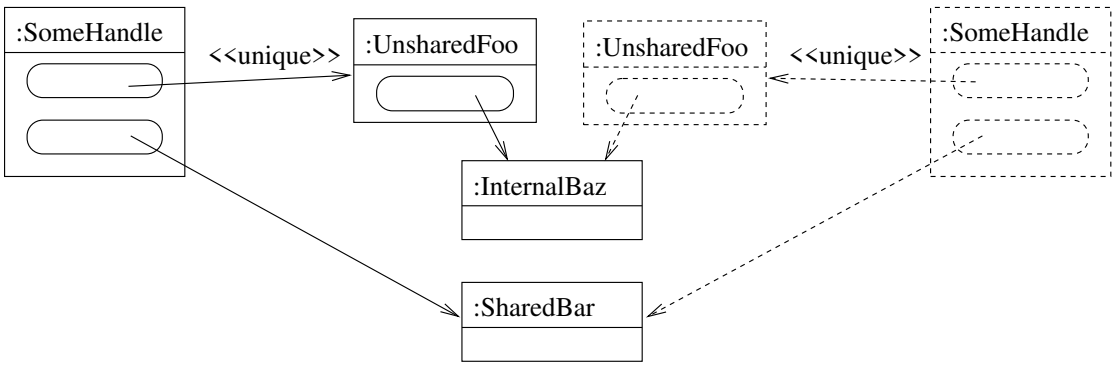

Abbildung 3.2.: Flache Kopie: *unshared objects* müssen dupliziert werden

darf daher auch von einem Subtyp deklariert sein. Beim Austausch werden die Werte jedoch in beide Richtungen übertragen, also darf keiner der deklarierten Typen ein Subtyp des jeweils anderen sein. Die Austausch-Operation ist also für eine objektorientierte Programmiersprache ungeeignet, da sie die Polymorphie untergräbt.

3.3.3. Ballons

Eine Weiterentwicklung der *islands* von [Hogg 91] findet sich in [Almeida 97]. Almeida vermeidet die Vielzahl an Zugriffsmodi, indem er einen komplexen Typüberprüfungsalgorithmus einsetzt. Der Programmierer deklariert lediglich, daß eine Klasse einen *balloon* implementieren soll (die Entsprechung zu Hogg's *islands*), und der Übersetzer prüft automatisch, ob die nötigen Bedingungen eingehalten werden. Ein Nachteil dieses Ansatzes ist, daß die vom Übersetzer generierten Fehlermeldungen nicht in dem Programmierer geläufigen Begriffen formuliert werden können.

Balloon-Objekte dürfen uneingeschränkt an flüchtige *balloon*-Variablen zugewiesen werden. Zuweisung an Exemplarvariablen ist jedoch nur per tiefer Kopie gestattet. Dadurch wird sichergestellt, daß auf ein Exemplar einer *balloon*-Klasse höchstens eine nichtflüchtige Referenz zur Zeit existieren kann. Tiefe Kopie stellt damit neben Austausch und destruktivem Lesen einen dritten Mechanismus dar, der Mehrfachbenennung vermeidet.

Almeida fügt hinzu, daß die Kopie nicht physikalisch stattfinden muß, wenn der Compiler feststellen kann, daß die Quelle im folgenden nicht mehr verwendet wird. Dies entspricht der Situation beim destruktiven Lesen. Auch wenn sich der Zustand des *balloon*-Objektes nicht mehr ändern kann, seine Klasse also einen zustandslosen Wert implementiert, ist eine Kopie überflüssig (unter der Voraussetzung, daß die Objektidentität nicht feststellbar ist). Für den letzteren Fall führt er eine Variante des *balloon* ein: Ein *value balloon* ist ein nicht-änderbarer *balloon*, der außerdem vor flüchtigen Referenzen ins Innere geschützt ist. Dies entspricht genau der Semantik von atomaren Datentypen in gängigen Programmiersprachen (vgl. Abschnitt 3.1.5). Man beachte, daß ein *value balloon* im Gegensatz zu

einem nicht-änderbaren Objekt à la Emerald auch transitiv keine Referenzen auf änderbare Objekte enthalten darf, und daher immer das gleiche Verhalten zeigt.

Als mögliche Anwendungsgebiete von *balloons* nennt Almeida auch Nebenläufigkeit und Verteilung. Da ein *balloon* das für verteilte Systeme geforderte Kriterium der Bindungsarmut erfüllt, bieten sich *balloons* als Einheit der Verteilung und Migration an. Sie sind auch als Einheit für *locking* geeignet, da jeder Zugriff auf die internen Objekte eines *balloon* durch das *balloon*-Objekt selbst geht.

Die Struktur der *balloons* kommt der der gesuchten dynamischen Komponenten schon recht nahe. Vergröbert man den Referenzgraph so, daß die nicht-*balloon*-Objekte ihrem jeweiligen *balloon* zugeordnet werden, ergibt sich ein Komponentengraph, dessen Knoten *balloons* und dessen Kanten Referenzen auf *balloons* sind. Jeder Knoten hat höchstens eine Superkomponente (**A1**), und der Graph ist zyklenfrei (**A2**). Die Struktur ist rekursiv polymorph (**A4**), und drückt keine Abhängigkeit der Lebenszeiten aus (**A9**). Die Sichtbarkeit auf untergeordnete *balloons* ist auf flüchtige Referenzen eingeschränkt, nicht-*balloon*-Objekte sich außerhalb gar nicht sichtbar (**A3**). Änderbarkeit ist Trennbarkeit ist nur im Sinne von Anforderungen **A5** und **A6** gegeben: Zwar kann eine *balloon*-Variable an ein neues Objekt gebunden werden, wodurch das bisherige Objekt ersetzt wird, das neu erzeugte Objekt kann aber Zeit seines Lebens an keine andere *balloon*-Exemplarvariable gebunden werden (**A7, A8**). *Balloons* können sich also nicht bewegen.

3.3.4. Flexibler Schutz vor Mehrfachbenennung

Der jüngste Artikel in der Diskussion ist [Noble et al. 98]. Der dort eingeführte Begriff des Schattens (*shadow*) entspricht dem *balloon* oder *island*. Ein wesentlicher Beitrag von Noble et al. besteht darin, daß in ihrem Ansatz die speziellen Anforderungen von Behälterklassen berücksichtigt werden.

Ein Behälter muß seine Repräsentation (*representation*), d.h. die internen Verwaltungsstrukturen kapseln, sollte dem Klienten aber uneingeschränkten Zugriff auf die im Behälter gespeicherten Objekte gewähren. Die in einem Behälter gespeicherten Objekte werden Argumente (*arguments*) genannt, und sind von seiner Repräsentation disjunkt. Aufgrund der möglichen Mehrfachbenennung darf der Behälter nicht von änderbaren Eigenschaften seiner Argumente abhängen. Besonders interessant wird der Fall, wenn ein Behälter in der Repräsentation eines anderen Behälters eingesetzt wird: Die Argumente des inneren Behälters gehören dadurch zur Repräsentation oder zu den Argumenten des äußeren Behälters, ohne daß dies den inneren Behälter beeinflußt. Der Status der Argumente des äußeren Behälters hängt wiederum von den Bedürfnissen seines Klienten ab.

Ein Beispiel zeigt Abbildung 3.3. Die Repräsentation eines Kurses verwendet eine Hashtabelle, während Studenten und Dozent zu den Argumenten des Kurses gehören. Die Hashtabelle speichert eine Bewertung für jeden Studenten. Die Bewertungen sind Argumente der Hashtabelle, aber Teil der Repräsentation des Kurses.

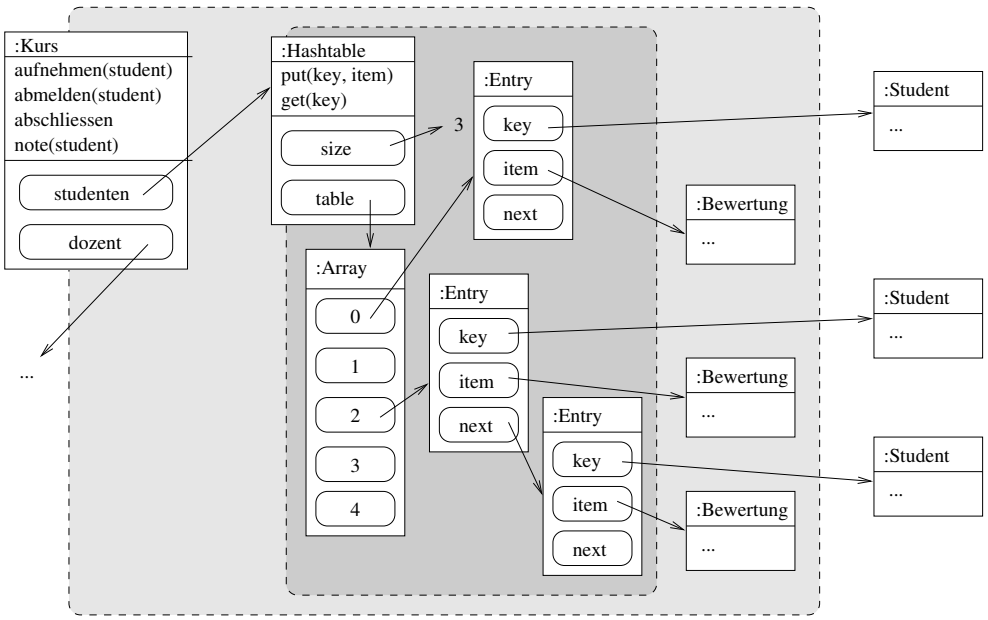

Abbildung 3.3.: Verschachtelte Behälter

Die programmiersprachliche Umsetzung des *shadow*-Konzeptes ähnelt eher Hoggs *islands* als Almeidas *balloons*, insofern als vom Programmierer Zugriffsmodi deklariert werden müssen. Der Fortschritt besteht zum einen darin, daß zwischen Referenzen auf Repräsentations-Objekte (**rep**) und Argument-Objekte (**arg**) unterschieden wird. Zum anderen können Zugriffsmodi ähnlich wie Typen parametrisiert werden. Ein Klient gibt also nicht nur an, welchen Typ die Elemente eines Behälters haben sollen, sondern auch, welche Art von Zugriff auf die Elemente er benötigt.

3.4. Folgerungen für die Umsetzung dynamischer Komponenten

In den untersuchten Ansätzen werden jeweils zwei Bindungsarten angeboten, wovon eine hierarchische Unterteilung bietet und die andere größte Flexibilität erlaubt. Diese Unterscheidung geschieht syntaktisch über die Deklaration der Variablen bzw. das bindende Konstrukt. Zur Umsetzung von Dynamischen Komponenten müssen analog Konstrukte zur Deklaration von Komponenten- und Referenzvariablen existieren.

Bei [Noble et al. 98] ist zusätzlich eine (statische) Parametrisierung der Bindungsart möglich. Eventuell ist eine solche Parametrisierung auch für die Unterscheidung zwischen Komponenten- und Referenzbindung möglich; die Frage geht jedoch über den Umfang dieser Arbeit hinaus.

In den Abschnitten dieses Kapitels wurden drei Ansätze für die Erzeugung hierarchischer

Strukturen beschrieben:

Gemeinsame Erzeugung Die Subkomponenten werden bei der Erzeugung des umgebenden Objekts sämtlich miterzeugt.

Komposition Das Ganze wird aus bereits existierenden Teilen zusammengesetzt.

Verzögerte Erzeugung Die Komponentenbindung ist zunächst leer. Sie kann später durch Erzeugung einer Subkomponente gefüllt werden.

Gemeinsame Erzeugung entspricht dabei nicht den Anforderungen an Komponentenbindung, da sie keine polymorphen Strukturen aufbauen kann (**A4**). Bei Komposition muß durch zusätzliche Mechanismen sichergestellt sein, daß die Teile nicht schon in anderen Komponentenbindungen stehen (**A1**). Verzögerte Erzeugung kommt den Anforderungen am nächsten, insbesondere wird hier explizit die Trennbarkeit der Superkomponente unterstützt (**A6**).

Aufgrund der geforderten Änderbarkeit (**A5**) verschwimmt der Unterschied zwischen Komposition und verzögerter Erzeugung. Komposition läßt sich umsetzen als leere Erzeugung gefolgt von Zuweisung der Teile, und verzögerte Erzeugung läßt sich umsetzen als Komposition leerer Komponentenbindung gefolgt von Zuweisung neu erzeugter Subkomponenten. Eine programmiersprachliche Umsetzung der Objekterzeugung im Kontext von dynamischen Komponenten sollte eine Kombination der Möglichkeiten bieten, um u.a. auch nicht-trennbare Aggregation darstellen zu können:

1. Komponentenbindungen werden leer angelegt.
2. Als Argumente der Objekterzeugung übergebene Teile werden an Komponentenbindungen des Objekts zugewiesen. Dabei ist sicherzustellen, daß die Teile nicht bereits in einer Komponentenbindung stehen. (Komposition)
3. Während der Initialisierung des Objekts können Subkomponenten erzeugt und zugewiesen werden. (Gemeinsame Erzeugung)
4. Bei Bedarf können die Komponentenbindungen später verändert werden. Insbesondere können neue Subkomponenten erzeugt und zugewiesen werden. (Verzögerte Erzeugung)

Die Änderbarkeit wird durch Operationen umgesetzt, die die Invarianten des Komponentengraphs aufrechterhalten (**A1, A2**). Mögliche Kandidaten sind:

Austausch Zwei Komponentenbindungen tauschen die gebundenen Objekte.

Destruktives Lesen Ein Objekt wird aus einer Komponentenbindung herausgenommen. Die ursprüngliche Komponentenbindung ist danach leer. Das Objekt ist ungebunden und kann sicher in eine andere Komponentenbindung hineinbewegt werden.

Tiefe Kopie Bei der Zuweisung von einer Komponentenbindung an eine andere Bindung werden das zugewiesene Objekte und alle seine transitiven Subkomponenten kopiert.

Alle diese Operationen erhalten Anforderung **A1** aufrecht (Exklusivität). Die Zyklenfreiheit (**A2**) muß jedoch sowohl beim Austausch als auch beim destruktiven Lesen zusätzlich geprüft werden. Gegen die Austausch-Operation spricht, daß sie die Subtypisierung und damit letztlich die Polymorphie (**A4**) unterbindet. Die tiefe Kopie ist nicht in der Lage, Migration (**A7**) auszudrücken, und kommt daher nur ergänzend zum destruktiven Lesen in Frage.

Die Umsetzung des destruktiven Lesens erfordert es, auch die Bindungsarten flüchtiger Variablen zu kennzeichnen. Es muß daher in allen Variablendeklarationen (von Exemplarvariable, lokaler Variable, Parameter und Methodenergebnis) zwischen Komponentenbindung und Referenzbindung unterschieden werden.

Diese allgemeinen Ergebnisse werden im folgenden Kapitel 4 konkret auf die Programmiersprache Tycoon-2 angewandt.

4. Dynamische Komponenten in der Programmiersprache Tycoon-2

Nachdem im vorangegangenen Kapitel die verwandten existierenden und in der Literatur vorgeschlagenen Konzepte und Lösungen ausgiebig betrachtet wurden, sollen diese nun auf die Programmiersprache Tycoon-2 übertragen und angewendet werden, um das in Kapitel 2 entwickelte Modell der dynamischen Komponenten an einem konkreten Beispiel programmiersprachlich umzusetzen.

In Abschnitt 4.1 werden zunächst die wesentlichen Punkte der Programmiersprache Tycoon-2 beschrieben; für eine ausführlichere Beschreibung verweise ich auf [Wahlen 98]. Die Umsetzung dynamischer Komponenten in der Sprache und in den zugehörigen Standardbibliotheken wird daraufhin in Abschnitt 4.2 überblicksartig dargestellt und gerechtfertigt, und in den folgenden Abschnitten detailliert definiert. Die technische Umsetzung wird in Kapitel 5 beschrieben.

4.1. Überblick über Tycoon-2

Die Programmiersprache Tycoon-2 läßt sich konzeptuell als statisch typisierte Variante der Programmiersprache Smalltalk [Goldberg, Robson 83] beschreiben. Wie Smalltalk ist Tycoon-2 eine rein objektorientierte, klassenbasierte Programmiersprache. Daten- und Kontrollfluß wird einheitlich durch Nachrichten dargestellt: Schleifen durch Endrekursion und bedingte Verzweigung mit Hilfe der späten Bindung von Methodenaufrufen. Die aufgerufene Methode hängt nur von der Klasse des Empfängers und dem Selektor der Nachricht ab.

In der Standardbibliothek stehen wie in Smalltalk Kontrollabstraktionen wie *if* und *while* zur Verfügung, die mit Hilfe von Funktionsausdrücken definiert sind. Funktionsausdrücke entsprechen den Codeblöcken in Smalltalk, und sind Sprachobjekte erster Klasse. Sie gehorchen statischen Sichtbarkeitsregeln, d.h. freie Variablen eines Funktionsausdrucks werden bei der Erzeugung eines Funktionsabschlusses gebunden, nicht erst beim Aufruf. Die Funktionsapplikation findet über Nachrichten statt.

Auch atomare Datentypen sind Objekte, und werden aussschließlich über Nachrichten zugegriffen. Die Oberflächensyntax definiert einige vereinfachende Schreibweisen (syntaktischen Zucker) für Nachrichten in Infix-Form und für Funktionsapplikation bzw. indizierten Zugriff.

Durch das einheitliche Objektmodell gibt es wie in Smalltalk nur eine Sorte von Variablen. Jede Variable enthält eine Referenzbindung an ein Objekt oder *nil*. Jedes Objekt hat unbeschränkte Lebenszeit.

Klassen sind ebenfalls Objekte, und als solche Exemplar einer Metaklasse. Das Klassenobjekt ist für die Erzeugung von Exemplaren der Klasse zuständig. Im Gegensatz zu Smalltalk werden Metaklassen getrennt von der Klasse selbst definiert und auch getrennt vererbt. Dies ist zur Typisierung der Objekterzeugung notwendig.

Im Unterschied zu Smalltalk erlaubt Tycoon-2 Mehrfachvererbung, und verwendet dazu das aus verschiedenen Lisp-Dialekten bekannte Konzept der *class precedence list* [Barret et al. 96]. Dies gestattet die Programmierung mit *Mixin*-Klassen.

Ein weiterer Unterschied besteht in der Handhabung von Exemplarvariablen: Durch die Definition einer Exemplarvariablen werden wie z.B. in Trellis/OWL automatisch eine lesende und eine schreibende Zugriffsmethode erzeugt. Auch das Objekt selbst kann auf seine Exemplarvariablen nur mit Hilfe dieser Methoden vermittels Nachrichten zugreifen. Dies erlaubt die Implementierung einer Nachrichtenschnittstelle mit Hilfe einer Exemplarvariable, und erlaubt eine transparente Redefinition des Zugriffs auf eine Variable durch benutzerdefinierte Methoden. Um den Zugriff syntaktisch zu erleichtern, gibt es auch hier vereinfachende Schreibweisen für die entsprechenden Nachrichtenausdrücke.

Laufzeitfehler wie z.B. der Aufruf einer nicht implementierten Methode werden durch Ausnahmen gemeldet. Ausnahmen unterbrechen den normalen Kontrollfluß, und können mit Hilfe der eingebauten Methode *try* abgefangen werden.

Wesentlich für die Programmiersprache Tycoon-2 ist das statische Typsystem, das strukturelle Subtypisierung mit beschränkt parametrischem Polymorphismus bietet. Ein Typ macht keine Aussage über die Repräsentation eines Objekts, nur über die von dem Objekt verstandenen Nachrichten. Klassen und Methoden können beschränkte Typparameter empfangen, was u.a. die Definition typsicherer Behälterklassen und binärer Methoden erlaubt. Ein typkorrektes Programm wird zur Laufzeit keine unverstandenen Nachrichten versenden, lediglich Nachrichten an den polymorphen Nullwert *nil* sind nicht auszuschließen.

Typen befinden sich rein auf der statischen Ebene, das Laufzeitverhalten wird nur durch Klassen bestimmt. Typparameter sind zur Laufzeit nicht beobachtbar, weshalb die Manipulation von Typen nie zu einer Laufzeitoperation führt.

Um *rapid prototyping* zu unterstützen ist die Typprüfung optional. Auch ein nicht statisch typkorrektes Programm kann ausgeführt werden, löst dann aber eventuell Ausnahmen aus. Die Systemintegrität ist nicht gefährdet.

In der Oberflächensyntax lehnt sich Tycoon-2 an die Programmiersprachen Java bzw. C++ an. Anstatt voluminöser Schlüsselworte werden verschiedene nichtalphanumerische Zeichenkombinationen eingesetzt. Die Syntax für Signaturen entspricht jedoch eher der Pascal- als der C-Tradition, insofern als Typen nachgestellt sind ($x{:}T$ statt $T\ x$).

Die Programmiersprache Tycoon-2 muß im Kontext des Systems Tycoon-2 gesehen werden. Das Tycoon-2-System unterstützt Nebenläufigkeit und automatische Speicherbereinigung. Der Objektspeicher inklusive der laufenden Tycoon-2-Prozesse kann persistent gesichert und zu einem späteren Zeitpunkt reaktiviert werden. Das System ist plattformunabhängig, die aktuelle Implementierung ist unter verschiedenen Windows- und Unix-Varianten verfügbar. Ein auf einer Plattform gespeicherter Objektspeicher kann auf einer beliebigen anderen reaktiviert werden.

Das Tycoon-2-System ist reflektiv und größtenteils in Tycoon-2 implementiert. Der Übersetzer ist als Objekt zugreifbar und erlaubt zur Laufzeit die Definition neuer Klassen und Methoden. Darüber hinaus ist es möglich, das Klassenobjekt zur Klasse eines beliebigen Objekts zu erfragen und so unter anderem die vorhandenen Exemplarvariablen und verfügbaren Methoden zu inspizieren.

Zur Reflektion gehört auch, daß eine nicht verstandene Nachricht in ein Objekt umgewandelt und dem Programm zur weiteren Verarbeitung zur Verfügung gestellt wird (*doesNotUnderstand*), und daß umgekehrt eine beliebige zur Laufzeit berechnete Nachricht versendet werden kann (*perform*). Durch die Verarbeitung und Übermittlung von Nachrichtenobjekten kann so z.B. entfernter Methodenaufruf mit vorhandenen Mitteln (*add-on*) implementiert werden.

4.2. Überblick der Spracherweiterung

In diesem Abschnitt wird beschrieben, wie dynamische Komponenten in die Programmiersprache Tycoon-2 eingefügt werden.

Die Unterscheidung zwischen Komponentenbindung und Referenzbindung soll explizit durch den Programmierer erfolgen, indem Variablen entweder als Referenzvariable oder als Komponentenvariable deklariert werden. Als Variable bezeichnen wie hier alle variablen Teile des Programmes, also alle Teile, die zur Laufzeit Bindungen erhalten[1]: Exemplarvariablen, Parameter, lokale Variablen und Ergebnisse. Bei der Auswertung von Ausdrücken werden ebenfalls temporäre Bindungen angelegt. Auch hier soll statisch feststellbar sein, ob es sich um eine Referenz- oder Komponentenbindung handelt.

Da in Tycoon-2 jede Variable explizit deklariert wird, muß hierzu nur die Deklarationssyntax erweitert werden. Aus Kompatiblitätsgründen soll die bisherige Syntax, deren Semantik einer Referenzbindung entspricht, für Referenzvariablen beibehalten werden:

```
x : T          ;; Referenzvariable
```

Für Komponentenvariablen wird der aus BETA bekannte Klammeraffe verwendet (@), der in der deutschen[2] wie in der englischen Lesart (*at*) Lokalität andeutet.

```
x @: T         ;; Komponentenvariable
```

[1] Dies entspricht dem Begriff der Entität in Eiffel.
[2] Die Subkomponente klammert sich an ihre Superkomponente.

Mit der *Sorte* einer Variable, Bindung oder eines Ausdrucks ist die Unterscheidung zwischen Referenz und Komponente gemeint. Der Klammeraffe wird auf die Seite der Variable und nicht auf die Seite des Typs geschrieben, um deutlich zu machen, daß Sorte und Typ orthogonale Konzepte sind (vgl. Abschnitt 3.3.1). Zudem ergibt sich eine optische Ähnlichkeit zum *diamond* in UML, der ebenfalls auf der Seite der Superkomponente eingezeichnet wird.

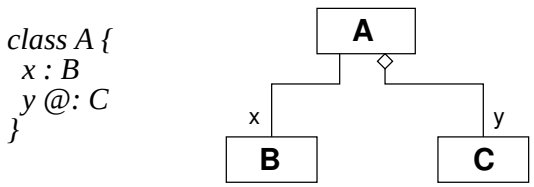

Abbildung 4.1.: Komponentendeklaration und Aggregationsnotation

Abbildung 4.2 zeigt ein Beispiel, in dem Komponenten als Parameter, Ergebnis und Exemplarvariable auftreten. Dargestellt ist ein Briefkorb, in dem Vorgänge abgelegt und (vermutlich von einem anderen Programmteil) abgeholt werden können. Der Vorgang wird dem Briefkorb als Komponente übergeben, wird als Komponente aufbewahrt und von dem anderen Programmteil als Komponente abgeholt. Eine Implementierung dieser Schnittstelle wird weiter unten gezeigt.

```
class Briefkorb
{
  ablegen(vorgang @: Vorgang) :Void
  { ... }

  abholen() @: Vorgang
  { ... }

  leeren() @: Vorgang
  { ... }

private
  abgelegterVorgang @: Vorgang
}
```

Abbildung 4.2.: Schnittstelle mit Komponenten

Die Operationen auf Komponentenvariablen sollen sicherstellen, daß die Invarianten der Komponentenbindung aufrechterhalten werden. Für jede neu angelegte Bindung einer Komponentenvariablen muß garantiert sein, daß keine weiteren Komponentenbindungen an das gebundene Objekt existieren. Dasselbe gilt für spätere Änderungen der Bindung. In Kapitel 3 wurden drei Mechanismen genannt, die dies leisten: Austausch, Kopie und destruktives Lesen. Die Austauschoperation ist für polymorphe Sprachen nicht geeignet,

da sie die Subtypisierung untergräbt. Eine Kopiersemantik paßt schlecht zum Konzept der Objektidentität. Als Basis wird daher die destruktive Leseoperation gewählt, die kurz *Herausnehmen* genannt werden soll.

In dem in Abschnitt 3.3.2 behandelten Artikel [Minsky 96] wird keine spezielle Syntax für destruktives Lesen benutzt, dort genügt es, den Namen der Variablen hinzuschreiben. Dies halte ich für gefährlich, da so implizit, d.h. möglicherweise aus Versehen der Inhalt einer Variablen verändert wird. Für eine derartige Operation soll also auch ein sichtbares Symbol verwendet werden.

Als Zeichen wird wie in der Deklaration von Komponentenvariablen der Klammeraffe gewählt (@), um die Analogie zwischen Deklarations- und Operationsebene zu betonen. Ein solcher Wiedererkennungseffekt wird z.B. auch in Pascal bei der doppelten Verwendung des Pfeiles (↑) ausgenutzt. Die Syntax ist also wie folgt:

x@ *;; herausnehmen*

Ist x als Komponentenvariable deklariert, so wird durch den Ausdruck $x@$ das aktuell an x gebundene Objekt herausgenommen und als Ergebnis des Ausdrucks geliefert. Die Komponentenbindung der Variable x ist danach leer.

Einer Komponentenvariable darf jederzeit ein Komponentenausdruck zugewiesen werden. Die Komponentenbindung der Variable wird auf das zugewiesene Objekt umgestellt, ihr bisher gebundenes Objekt wird fallengelassen. Mit dieser Operation ist die Komponentenbindung offensichtlich wie gefordert änderbar, und um die geforderte Polymorphie zu erreichen, soll einfach dieselbe Subtypisierung wie bei der Zuweisung an Referenzvariablen gelten. Da auf der rechten Seite ein Komponentenausdruck stehen muß, bleibt die Anzahl der Komponentenbindungen an das zugewiesene Objekt eins.

x := $\mathcal{A}$ *;; hineinbewegen. $\mathcal{A}$ ist Komponenten-Ausdruck*

Die Komponenten-Zuweisung verwendet dieselbe Syntax wie die Zuweisung an Referenzvariablen, da durch die Syntax der rechten Seite für den Programmierer klar erkenntlich ist, um welche Art von Zuweisung es sich handelt.

Zuletzt wird noch eine Operation benötigt, um aus einer Komponenten- eine Referenzbindung zu erzeugen. Diese Operation soll *referenzieren* heißen. Es bietet sich an, dieselbe Syntax wie zum Auslesen einer Referenzvariable zu benutzen: Der Name der Variablen wird ohne weitere Symbole hingeschrieben. Da mit Hilfe dieser Operation beliebig viele Referenzen auf eine Subkomponente möglich sind, gilt auch die geforderte beliebige Sichtbarkeit von Subkomponenten.

x *;; referenzieren*

Ist also x eine Komponentenvariable, dann ergibt der Ausdruck x eine Referenzbindung an das aktuell von der Variable x gebundene Objekt. Wenn die Komponentenbindung leer (*nil*) ist, ist es die Referenzbindung ebenfalls.

Die Vorstellung ist hilfreich, daß eine Referenzbindung die Identität eines Objektes enthält, während sich in einer Komponentenbindung das Objekt selbst befindet. Beim

Referenzieren würde dann die Identität des enthaltenen Objektes abgelesen, während beim *Herausnehmen* und *Hineinbewegen* das Objekt selbst bewegt wird. Unter diesem Blickwinkel wird klar, weshalb ein Komponentenausdruck nicht einer Referenzvariable zugewiesen werden kann: Das Objekt würde fallengelassen, ohne daß der Programmierer es bemerkt.

Wird eine Nachricht gesendet, so müssen wie bei der Zuweisung die Sorten der Argumente und der Parameter übereinstimmen. Dies kann wie in Abschnitt 3.3.1 beschrieben sowohl statisch als auch zur Laufzeit geprüft werden. Die erwartete Ergebnissorte soll in jedem Fall direkt im Programmtext sichtbar sein, um Überraschungen für den Programmierer zu minimieren. Hinter einem Aufruf, der eine Komponentenbindung liefern soll, muß daher ein Klammeraffe stehen.

```
r.m(...)      ;; Aufruf einer Methode mit Referenzergebnis
r.m(...)@     ;; Aufruf einer anderen Methode mit Komponentenergebnis
```

Um den Unterschied zwischen Referenzieren und Herausnehmen auch in benutzerdefinierten Methoden nachbilden zu können, soll die Ergebnissorte statisch überladen sein. Es werden also tatsächlich verschiedene Methoden aufgerufen, je nachdem, welche erwartete Ergebnissorte beim Aufruf angegeben wurde. Dieser Mechanismus ist besonders für Behälterklassen nützlich, und wird auch für die automatisch generierten Zugriffsmethoden auf Exemplarvariablen ausgenutzt.

Die üblichen Argumente gegen überladene Methodensignaturen wie in Java und C++ treffen hier nicht zu, da die Entscheidung zwischen den auszuwählenden Methoden auf der An- oder Abwesenheit eines vom Programmierer gesetzten Zeichens basiert, nicht auf vom Übersetzer inferierten, auch von anderen Programmteilen abhängigen Typen.

Wird eine Komponentenbindung verändert, oder läuft die Lebenszeit einer Komponentenvariable ab, so geht die einzige Komponentenbindung an das bisher enthaltene Objekt verloren, man sagt, das Objekt wird ungebunden. Es ist dann nur noch über Referenzbindungen zu erreichen. Da Referenzbindungen aber uneingeschränkt existieren können, darf das Objekt keineswegs freigegeben werden; seine Lebenszeit kann die der Superkomponente also durchaus übersteigen. Damit ist die geforderte Unabhängigkeit der Lebenszeiten schon fast erfüllt; zusätzlich wird nur noch gefordert, daß eine Subkomponente die Superkomponente bei der automatischen Speicherbereinigung nicht am Leben hält. Darauf ist bei der Implementierung des Laufzeitsystems zu achten.

Unter Umständen kann auch die zum Referenzieren inverse Operation benötigt werden. Im Gegensatz zum Referenzieren ist es aber nicht immer möglich, aus einer Referenzbindung eine Komponentenbindung zu erzeugen, ohne die Invarianten der Komponentenbindung zu verletzen. Vorraussetzung ist, daß es keine Komponentenbindung an das referenzierte Objekt gibt, so daß die neu erzeugte Komponentenbindung tatsächlich die einzige ist. Dies kann nicht lokal am Programmtext abgelesen werden, weshalb die Operation einen Laufzeittest erfordert. Sie soll daher mit höheren syntaktischen Kosten belastet werden, indem sie als Methode des Laufzeitsystems definiert wird.

```
r.fetch@      ;; Komponentenbindung aus Referenzbindung erzeugen
```

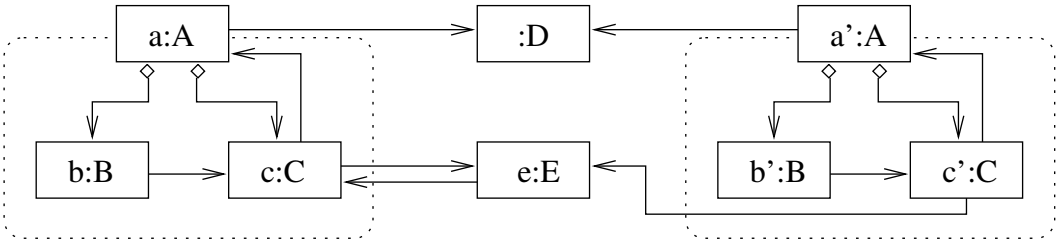

Abbildung 4.3.: Tiefe Kopie a' der Komponente a

Wie eingangs erwähnt bleiben beim Kopieren eines Objekts die Invarianten der Komponentenbindung erhalten, wenn alle Komponentenbindungen verfolgt werden, d.h. wenn in Bezug auf die Komponentenbindungen eine tiefe Kopie angefertigt wird. Referenzbindungen innerhalb der kopierten Objektgruppe werden wie in Abbildung 4.3 dargestellt angepaßt. Die Kopieroperation soll aufgrund des beliebig hohen Laufzeit- und Speicheraufwandes ebenfalls als Methode definiert werden.

r.copy@ *;; Tiefe Kopie einer Komponente*

Dem aufmerksamen Leser wird nicht entgangen sein, daß die Zyklenfreiheit durch die bisher definierten Operationen nicht garantiert ist. Zwar kann jedes Objekt nur eine Superkomponente haben, ein "Ringelrein" wie in den Beispielen in Abschnitt 3.1 ist aber nach wie vor möglich. Um Zyklen auszuschließen, müßte bei jeder Komponentenzuweisung an eine Exemplarvariable geprüft werden, ob die Exemplarvariable nicht transitiv Teil des zugewiesenen Objekts ist.

Man kann hier eine Parallele zum *occur check* in Prolog-Programmen ziehen: In Prolog ist es von der logischen Grundlage her nicht sinnvoll, zyklische, d.h. nicht-endliche Datenstrukturen zu generieren. Eine Unifikation, die solche Strukturen erzeugen müßte, sollte daher fehlschlagen. Da der dazu erforderliche *occur check*[3] aber sehr aufwendig ist, und zyklische Datenstrukturen den gängigen Prolog-Implementierungen keinerlei Probleme bereiten, bieten diese Systeme die Möglichkeit, die Prüfung zu deaktivieren.

Um diesen Ansatz auf Tycoon-2 übertragen zu können, muß nur sichergestellt sein, daß die im Laufzeitsystem verwendeten Algorithmen auch mit zyklischen Datenstrukturen zurechtkommen. Der Programmierer kann die Zyklenprüfung dann wie eine Zusicherung (*assertion*) während der Programmentwicklung aktivieren, sie unter geschwindigkeitskritischen Umständen aber wieder ausschalten.

Als kleine Demonstration der Sprachmittel soll zum Abschluß dieses Abschnittes die oben definierte Schnittstelle der Klasse `Briefkorb` implementiert werden (Abbildung 4.4). Auf die dazu benötigten Synchronisationsprimitive wird hier nicht weiter eingegangen, wichtig ist lediglich, daß eine explizite Synchronisation notwendig ist. Genau wie bei quasiparallelen Schreib- und Lesezugriffen ist das Systemverhalten bei nicht synchronisiertem destruktiven Lesen undefiniert.

[3] Check *if the variable* occurs *within the unification term.* Dies ist z.B. bei der Unifikation von X mit $f(X)$ der Fall, so daß diese Terme nicht unifizierbar sein dürften.

```
class Briefkorb
super Monitor
{
  ablegen(vorgang @: Vorgang) :Void  {
    synchronize({
      ;; warte, bis freier Platz vorhanden ist
      while({ abgelegterVorgang !== nil } do: {
        wait(nichtsAbgelegt)
      })
      ;; bewege den Vorgang aus dem Parameter in die Exemplarvariable
      abgelegterVorgang := vorgang@
      etwasAbgelegt.signal
    })
  }

  abholen() @: Vorgang  {
    ergebnis @:Message    ;; eine lokale Komponentenvariable
    synchronize({
      ;; warte, bis ein Vorgang abgelegt ist
      while({ abgelegterVorgang == nil } do: {
        wait(etwasAbgelegt)
      })
      ;; nimm den Vorgang aus der Exemplarvariable in die lokale Variable
      ergebnis := abgelegterVorgang@
      nichtsAbgelegt.signal
    }) ;; verlasse den Monitor
    ;; nimm die Komponente aus der lokalen Variable und liefere sie als Ergebnis
    ergebnis@
  }

  leeren() @: Vorgang  {
    synchronize({
      nichtsAbgelegt.signal
      abgelegterVorgang@
    })@
  }

private
  abgelegterVorgang @: Vorgang
  nichtsAbgelegt :Condition
  etwasAbgelegt :Condition
}
```

Abbildung 4.4.: Implementierung mit Komponenten

Eine Alternative hätte darin bestanden, bei nebenläufigem destruktiv lesenden Zugriff auf eine Komponentenvariable genau einem beteiligten Ausführungskontext den ehemaligen Variableninhalt zukommen zu lassen, oder sogar einen destruktiv lesenden Prozeß aufzuhalten, bis zu entnehmender Inhalt zur Verfügung steht. Da die Operation *Herausnehmen* aber relativ häufig ist, hätte die im Laufzeitsystem jedesmal ausgeführte Synchronisation die Effizienz sehr beeinträchtigt.

4.3. Erweiterungen der Grammatik

Die Spracherweiterung soll nun detaillierter beschrieben werden. Als erstes wird mit diesem Abschnitt die syntaktische Ebene behandelt.

Eine vollständige Beschreibung der erweiterten Grammatik findet sich in Anhang A. Hier beschreibe ich nur kurz, in welchen Punkten sich die Grammatik von der in [Gawecki, Wienberg 98] beschriebenen unterscheidet.

Auf der Ebene der Deklarationen wird wie im vorangegangenen Abschnitt skizziert für jede Variablen- und Ergebnisdeklaration zusätzlich eine Komponenten-Variante mit einem Klammeraffen (*@:* statt *:*) eingeführt. Dies betrifft folgende Nonterminale:

ValueSignature Dieses Nonterminal steht in Parameterlisten für Parameter-Variablen. Eine zusätzliche Regel erlaubt Komponenten-Parameter.

Binding Ein *binding* führt eine neue lokale Referenz- oder Komponenten-Variable ein.

SlotDefinition Dieses Nonterminal steht für die Definition einer Exemplarvariablen.

MethodDefinition Das Ergebnis einer Methode kann als Referenz oder als Komponente deklariert werden.

Value Das gleiche gilt für Funktionsausdrücke (`fun`).

Type Unterschiedliche Ergebnissorten gibt es daher auch für den Typ eines Funktionsausdrucks (`Fun`).

Auf der Wertebene gibt es neue Regeln für das Nonterminal **Value**. Die Sprache Tycoon-2 bietet einige syntaktische Varianten, um Nachrichtenausdrücke zu formulieren. Bei einigen dieser Varianten kann durch einen nachgestellten Klammeraffen angegeben werden, daß ein Komponenten-Ergebnis erwartet wird. Da in der Oberflächensyntax kein Unterschied zwischen einer argumentlosen Nachricht an **self** und dem Zugriff auf eine statisch gebundene Variable besteht, ist die Syntax für das *Herausnehmen* bereits in der Regel für Nachrichten an **self** enthalten.

Die vollständige Grammatik erscheint in Anhang A.

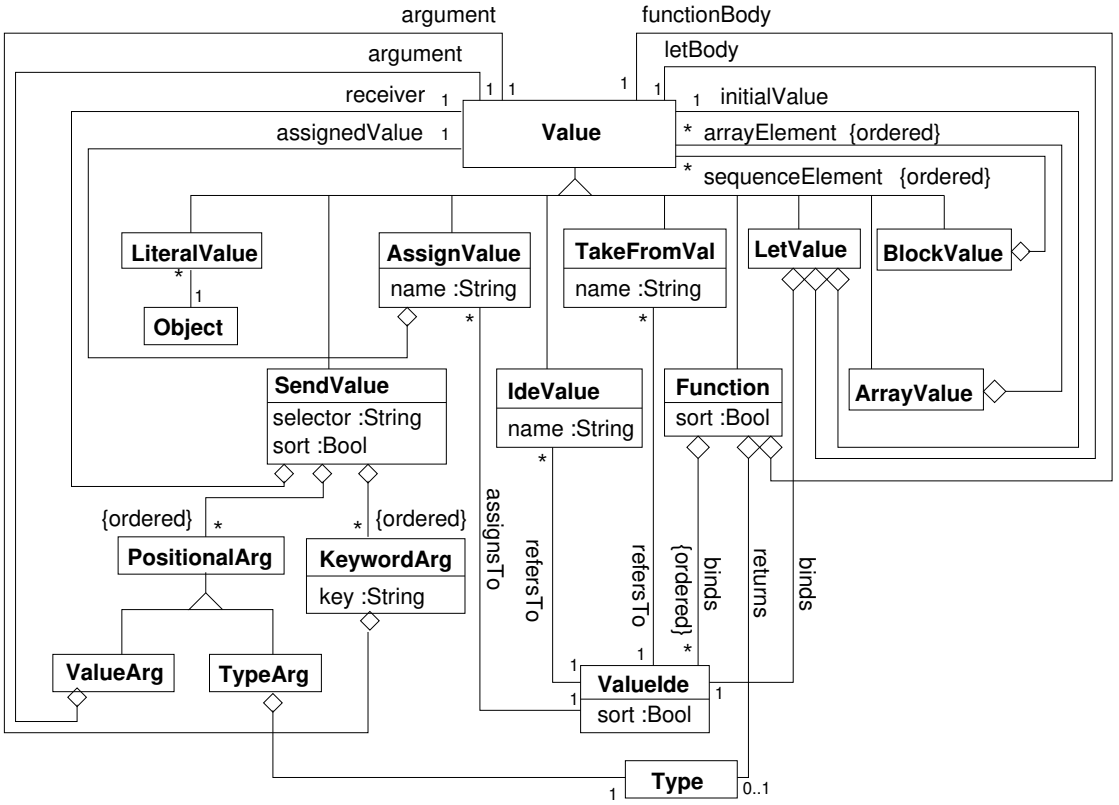

Abbildung 4.5.: Abstrakter Syntaxbaum (Werte)

4.4. Der erweiterte abstrakte Syntaxbaum

Da die Oberflächensyntax Redundanz und syntaktischen Zucker enthält, wird für die inhaltliche Beschreibung der Sprache eine abstrakte Syntax verwendet. Diese wird auch intern vom Übersetzer eingesetzt (siehe Abschnitt 5.2).

Die abstrakte Syntax ist in den Abbildungen 4.5, 4.6 und 4.7 als UML-Klassendiagramm wiedergegeben. Diese Darstellung bietet im Vergleich zu einer formalen Grammatik mehrere Vorteile: Erstens erlaubt das Modellierungskonzept der Generalisierung, die Klassen sinnvoll zu gruppieren. Zweitens sind auch die statischen Bindungen der Bezeichner darstellbar – ein `IdeValue` bezieht sich immer auf einen `ValueIde`, ein `IdeType` auf einen `TypeIde` etc. Weiter erlaubt die Darstellung einen direkten Bezug zu der in Kapitel 5 beschriebenen Implementierung des Übersetzers. Viertens erlaubt sie auch den Vergleich mit dem in [Wienberg 97] gegebenen Klassendiagramm.

In mehreren Klassen des abstrakten Syntaxbaumes muß ein boolesches Attribut eingeführt werden, das zwischen Referenz und Komponente unterscheidet. Im Diagramm wird es durchgehend mit `sort` benenannt. Der Wert des Attributs ist durch die Oberflächensyntax vorgegeben. Im einzelnen ist die Bedeutung des Attributs wie folgt:

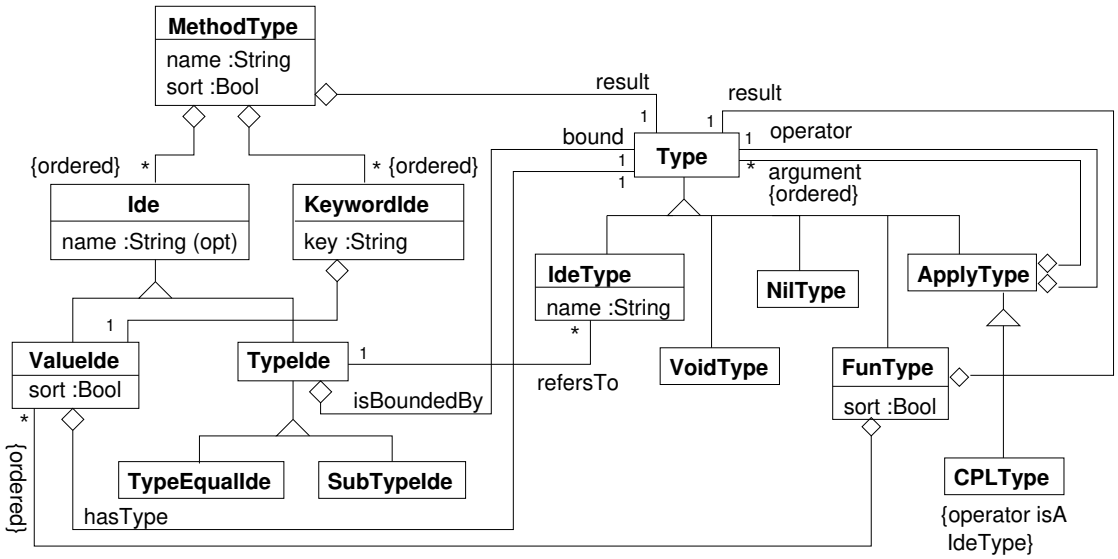

Abbildung 4.6.: Abstrakter Syntaxbaum (Typen, Signaturen)

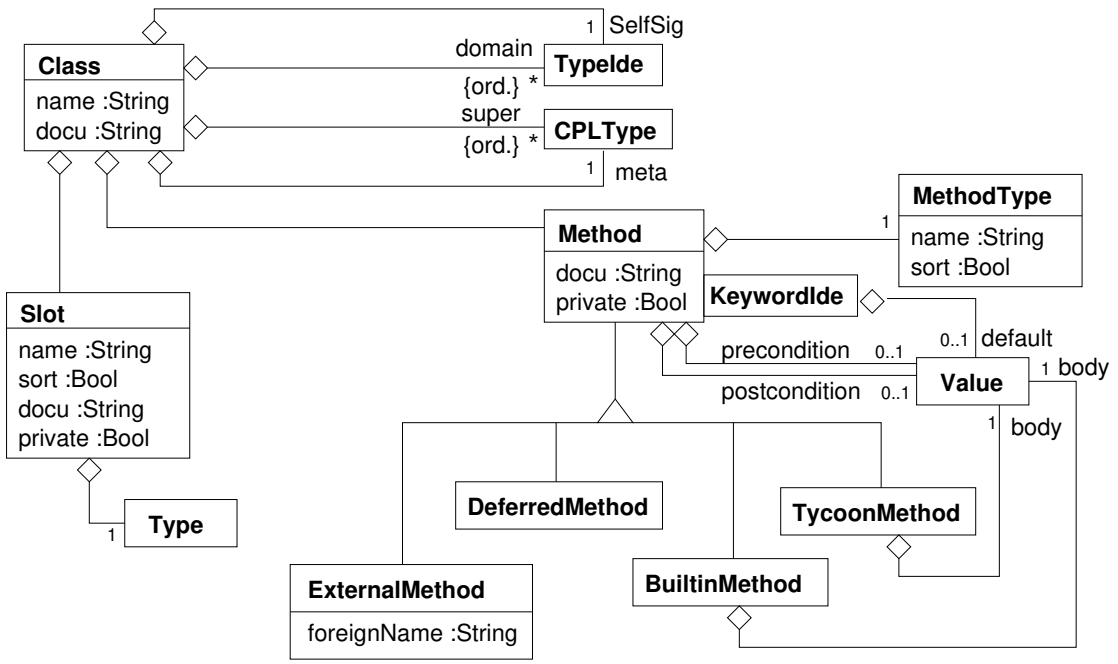

Abbildung 4.7.: Abstrakter Syntaxbaum (Klassen, Methoden)

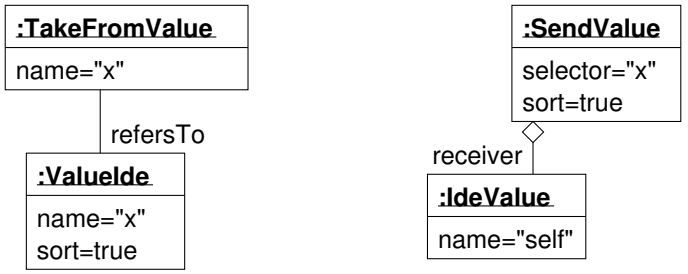

Abbildung 4.8.: Verschiedene Darstellungen der Oberflächensyntax *x@*

SendValue Das Attribut gibt an, welche Ergebnissorte der Nachrichtenausdruck von der aufgerufenen Methode fordert.

Function Das Attribut gibt die deklarierte Ergebnissorte des Funktionsausdrucks an.

ValueIde Diese Klasse steht für die Definition eines Wertbezeichners, also für eine Ausprägung der Nonterminale *ValueSignature* oder *Binding*. Das Attribut drückt die Sorte der definierten Parametervariable bzw. lokalen Variable aus.

MethodType Das Attribut gibt die deklarierte Ergebnissorte der Methode an.

Slot Diese Klasse entspricht der im vorherigen Abschnitt erwähnten Regel *SlotDefinition* der Oberflächensyntax. Das Attribut drückt die Sorte der definierten Exemplarvariable aus.

Die einzige neu eingeführte Klasse, `TakeFromValue` (Herausnehmen) in Abbildung 4.5, repräsentiert den destruktiven Lesezugriff auf eine statisch gebundene Komponentenvariable. Als statisch gebunden bezeichne ich Parameter und lokale Variablen, wobei es sich im Zusammenspiel mit Funktionsausdrücken auch um statisch weiter außen definierte Variablen handeln kann. Auf Exemplarvariablen wird in Tycoon-2 immer über Nachrichten zugegriffen, diese werden also dynamisch gebunden. Die identische Oberflächensyntax wird abhängig davon, ob der angegebene Name statisch gebunden ist, entweder durch einen Knoten der Klasse `TakeFromValue` oder durch eine `SendValue` mit Ergebnissorte *Komponente* dargestellt (Abbildung 4.8).

Für die Unterscheidung zwischen Referenzieren einer Komponentenvariable und Auslesen einer Referenzvariable existiert keine eigene Klasse, für beide Operationen wird die Klasse `IdeValue` benutzt. Der Unterschied besteht lediglich in der Sorte der Variablen. Beim Zugriff auf eine dynamisch gebundene Variable ist die Sorte statisch unbekannt (die Zugriffsmethode könnte ja in einer Subklasse umdefiniert werden) und wird im abstrakten Syntaxbaum nicht repräsentiert (Abbildung 4.9).

Auch für die Zuweisung eines Komponentenausdrucks an eine Komponentenvariable (Hineinbewegen) existiert keine eigene Klasse, der Unterschied besteht lediglich in der

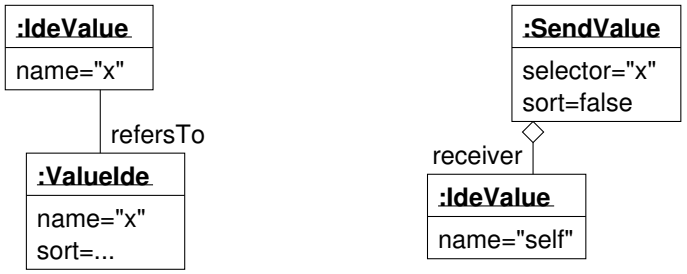

Abbildung 4.9.: Verschiedene Darstellungen der Oberflächensyntax x

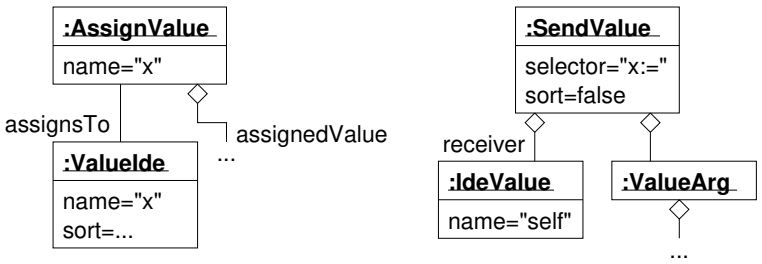

Abbildung 4.10.: Verschiedene Darstellungen der Oberflächensyntax $x{:=}\mathcal{A}$

Sorte des zugewiesenen Wertausdrucks. Im Falle eines nicht statisch gebundenen Bezeichners entspricht dies der Sorte des Argumentausdrucks in dem erzeugten Nachrichtenausdruck (Abbildung 4.10).

4.5. Zusätzliche Regeln für die Typüberprüfung

Die Programmiersprache Tycoon-2 ist wie erwähnt statisch getypt. Das eingesetzte Typsystem wird detailliert in [Ernst 98] beschrieben. Ich setze die dort verwendete Notation voraus; hier erscheinen nur die Ergänzungen, die durch die Einführung von dynamischen Komponenten nötig werden. Die in [Ernst 98] und im folgenden verwendeten Metavariablen sind noch einmal in Tabelle 4.1 wiedergegeben.

Die in [Ernst 98] verwendete Typaussage $\Gamma \vdash a : A$ wird gedeutet als "In der Umgebung Γ hat der Ausdruck a die Sorte *Referenz* und den Typ A". Eine Aussage über die Sorte *Komponente* soll passend zur Oberflächensyntax geschrieben werden als $\Gamma \vdash a \mathbin{@:} A$. Die in [Ernst 98] angegebene abstrakte Syntax muß ebenfalls an die im vorangegangenen Abschnitt beschriebene angepaßt werden. Dies geschieht entsprechend der Oberflächensyntax, die im folgenden verwendete Syntax sollte also keiner Erläuterung mehr bedürfen.

Schlüsselwortparameter werden in [Ernst 98] nicht behandelt. Dies soll hier auch nicht nachgeholt werden, da Schlüsselwortparameter orthogonal zu dynamischen Komponenten sind.

Variable	Bedeutung
a, b, c	Wertausdruck
x, y	Wertbezeichner
A, B, C	Typausdruck
T, U	Typbezeichner
S	Signatur
M	Methodensignatur
m	Methodenselektor
s	Sorte
$\vec{\mathcal{A}}$	Liste von $\mathcal{A}$
$[]$	leere Liste
$\mathcal{A} : \vec{\mathcal{B}}$	Element $\mathcal{A}$ vor Liste $\vec{\mathcal{B}}$
$[\mathcal{A}, \mathcal{B}, \mathcal{C}]$	Liste mit drei Elementen $= \mathcal{A} : \mathcal{B} : \mathcal{C} : []$

Tabelle 4.1.: Metavariablen

Alle in [Ernst 98] definierten Typregeln bleiben weiterhin gültig, die hier definierten kommen ergänzend hinzu. Das bedeutet insbesondere, daß ein nach den bisherigen Regeln typkorrektes Programm auch in der erweiterten Sprache typkorrekt bleibt.

4.5.1. Subsumption

Für Komponenten soll wie für Referenzen das Prinzip der Subsumption gelten, das besagt, daß ein Objekt eines Typs A eingesetzt werden kann, wenn ein Objekt eines allgemeineren Typs B erwartet wird. Das Prinzip wird durch die folgende Regel ausgedrückt. Diese Regel ist nicht direkt Teil des Typprüfungsalgorithmus, sondern tritt implizit in den anderen umgesetzten Regeln auf.

$$\frac{\Gamma \vdash x \ @\!: A \qquad \Gamma \vdash A <\!: B}{\Gamma \vdash x \ @\!: B} \ \textit{component subsumption}$$

4.5.2. Wert- und Typbezeichner

Es gibt zwei Möglichkeiten, auf eine statisch gebundene Komponentenvariable zuzugreifen: Durch Referenzieren und durch Herausnehmen. Das Ergebnis des Referenzierens ist eine Referenzbindung, durch Herausnehmen erhält man eine Komponentenbindung. Die zugehörigen Typregeln folgen.

$$\frac{}{\ldots, x \ @\!: T, \ldots \vdash x : T} \ \textit{reference}$$

$$\frac{}{\ldots, x\ @\!:\ T, \ldots \vdash x@\ @\!:\ T}\ \textit{take out}$$

4.5.3. Signaturen

Wie bei Referenzsignaturen kann man eine Subsignaturbeziehung zwischen Komponentensignaturen durch die Subtypisierung der deklarierten Typen herstellen. Wertsignaturen mit verschiedenen Sorten sind nicht vergleichbar. Die Beziehung wird rekursiv für ganze Signaturlisten definiert.

$$\frac{\Gamma \vdash A <\!: A' \qquad \Gamma \vdash \vec{S} <\!:: \vec{S'}}{\Gamma \vdash (x\ @\!:\ A) : \vec{S} <\!:: (x'\ @\!:\ A') : \vec{S'}}\ \textit{Subsig Component}$$

In [Ernst 98] wird eine Funktion Sig_p eingeführt, die aus einer Liste von Argumentausdrücken eine Liste der geforderten Parametersignaturen konstruiert. Diese Funktion wird benutzt, um die Gültigkeit eines Methodenaufrufs mit Hilfe der Subsignatur-Beziehung auszudrücken. Es wird eine zusätzliche Regel für Komponenten-Argumente gebraucht: Tritt in der Argumentliste ein Komponenten-Argument-Ausdruck a auf, so wird in der zugehörigen Parameterliste eine Komponentensignatur mit passendem Typ erwartet.

$$\frac{\Gamma \vdash a\ @\!:\ A}{\Gamma \vdash Sig_p(a : \vec{p}) = (?\ @\!:\ A) : Sig_p(\vec{p})}\ \textit{Sig Param 4}$$

In Tycoon-2 existiert eine kontravariante Regel für die Subtypisierung von Methodenparametern, die im Zusammenhang mit dynamischen Komponenten natürlich weitergelten soll. Außerdem wird gefordert, daß zwei Methodensignaturen dieselbe Ergebnissorte haben müssen, um überhaupt vergleichbar zu sein.

$$\frac{\Gamma \vdash \vec{S'} <\!:: \vec{S} \qquad \Gamma \vdash T <\!: T'}{\Gamma \vdash m(\vec{S})@ : T <\!: m(\vec{S'})@ : T'}\ \textit{Subsig Component}$$

4.5.4. Subtypisierung auf Schnittstellen

Ein Schnittstellentyp hat die Form $\sigma[\overrightarrow{Methodensignatur}]$. Es soll möglich sein, in einer Schnittstelle zwei Methoden zur Verfügung zu stellen, die denselben Namen, aber unterschiedliche Ergebnissorten haben; das bedeutet eben, daß Ergebnissorten statisch überladen sind. Das Paar aus Methodenname und Ergebnissorte soll daher *erweiterter Selektor* heißen. Zu einem erweiterten Selektor (m, s) gibt es in einem Schnittstellentyp T also höchstens eine Methodensignatur, die mit $meth(T, (m, s))$ bezeichnet werden soll. Wird $pubSel(T)$ definiert als die Menge der öffentlichen erweiterten Selektoren von T, dann behält die Typregel aus [Ernst 98] fast ihre dortige Form.

$$\frac{\Gamma \vdash pubSel(T') \subseteq pubSel(T) \quad \Gamma \vdash \forall (m,s) \in pubSel(T') : meth(T,(m,s)) <: meth(T',(m,s))}{\Gamma \vdash T <: T'} \textit{ Subtype Interface}$$

4.5.5. Block-Ausdruck

Nun werden die Wertausdrücke behandelt. Die Sorte eines Block-Ausdrucks ist die Sorte seines letzten Elements. Alle Blockelemente außer dem letzten müssen Referenzen sein, damit Komponentenbindungen nicht unabsichtlich verloren gehen.

$$\frac{\Gamma \vdash a \ @: A}{\Gamma \vdash [a] \ @: A} \textit{ Val Component Block 2}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash \vec{b} \ @: B}{\Gamma \vdash (a : \vec{b}) \ @: B} \textit{ Val Component Block 3}$$

4.5.6. Bindungs-Ausdruck

Die neu eingeführten Regeln besagen lediglich, daß die Sorte des gebundenen Ausdrucks die gleiche sein muß wie die der deklarierten Variable, und daß der Bindungs-Ausdruck die Sorte seines Körper-Ausdrucks annimmt. Wird kein Körper-Ausdruck angegeben, übernimmt er die Sorte des gebundenen Ausdrucks.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B \quad \Gamma, x : B \vdash c \ @: C}{\Gamma \vdash (\text{let } x : B = a \text{ in } c) \ @: C} \textit{ Val let 3}$$

$$\frac{\Gamma \vdash a \ @: A \quad \Gamma \vdash A <: B}{\Gamma \vdash (\text{let } x \ @: B = a \text{ in } []) \ @: B} \textit{ Component let}$$

$$\frac{\Gamma \vdash a \ @: A \quad \Gamma \vdash A <: B \quad \Gamma, x \ @: B \vdash c : C}{\Gamma \vdash (\text{let } x \ @: B = a \text{ in } c) : C} \textit{ Component let 2}$$

$$\frac{\Gamma \vdash a \ @: A \quad \Gamma \vdash A <: B \quad \Gamma, x \ @: B \vdash c \ @: C}{\Gamma \vdash (\text{let } x \ @: B = a \text{ in } c) \ @: C} \textit{ Component let 3}$$

4.5.7. Methodenaufruf

Die Sorte eines Methodenaufrufs muß mit der deklarierten Ergebnissorte der aufgerufenen Methode übereinstimmen. Ansonsten gleicht die hier neu definierte Regel genau der bereits aus [Ernst 98] bekannten. Insbesondere darf als Empfänger der Nachricht nur ein Referenzausdruck erscheinen. Man beachte, daß die überladene Ergebnissorte bereits durch die Subtypisierung der Schnittstellen abgehandelt wird.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: \sigma[m(\vec{S})@ : T] \quad \Gamma \vdash Sig_p(\vec{p}) = \vec{S'} \quad \Gamma \vdash \vec{S'} <:: \vec{S}}{\Gamma \vdash a.m(\vec{p})@\ @{:}\ T[\vec{p}/\vec{S}][A/Self]}\ \textit{Component send}$$

4.5.8. Funktionsabstraktionen

Die abstrakte Syntax ist um Funktionsausdrücke und Funktionstypen mit Komponentenergebnis erweitert worden. Ein Funktionstyp mit Komponentenergebnis entspricht folgendem Schnittstellentyp:

$$Fun(\vec{S})@ : T = \sigma[/](\vec{S})@ : T]$$

Die Sorte des Körpers einer Funktionsabstraktion muß mit der deklarierten Ergebnissorte übereinstimmen. Der Funktionsausdruck selbst führt immer zu einer Referenz auf den Funktionstyp.

$$\frac{\Gamma, \vec{S} \vdash \vec{a}\ @{:}\ A \quad \Gamma, \vec{S} \vdash A <: B}{\Gamma \vdash (fun(\vec{S})@ : B\{\vec{a}\}) : Fun(\vec{S})@ : B}\ \textit{Component fun}$$

Die Funktionsapplikation muß in diesem Fall ein Komponentenergebnis erwarten.

4.5.9. Zuweisung

Der letzte zu betrachtende Wertausdruck ist die Zuweisung, bzw. im Falle von Komponenten das Hineinbewegen. Bei der Zuweisung mußten linke und rechte Seite von der Sorte *Referenz* sein, beim Hineinbewegen müssen beide die Sorte *Komponente* haben.

$$\frac{\Gamma \vdash a\ @{:}\ A \quad \Gamma \vdash A <: B \quad \Gamma = \ldots, x\ @{:}\ B, \ldots}{\Gamma \vdash (x := a) : \mathbf{Void}}\ \textit{Move Into}$$

4.5.10. Methodendefinition

Die Sorte eines Methodenkörpers muß mit der deklarierten Ergebnissorte der Methode übereinstimmen. Auch bei den *default*-Werten der Schlüsselwortparameter muß die Sorte der angegebenen Wertsignatur verwendet werden. Vor- und Nachbedingung einer Methode müssen zu einer Referenz auf `Bool` evaluieren.

4.6. Objektmodell

Abbildung 4.11 zeigt ein Klassendiagramm, dem man entnehmen kann, wie und wo dynamische Komponenten im Laufzeitzustand eines Tycoon-2-Systems auftreten.

Das zugrundeliegende Objektmodell von Tycoon-2 ist sehr einfach. Jedes Objekt ist Exemplar einer Klasse. Für jede Exemplarvariable seiner Klasse hat das Objekt eine Bindung, welche wiederum auf ein Objekt verweist oder *nil* ist. Zu den Exemplarvariablen und Methoden einer Klasse zählen die ererbten hinzu.

Zu dem Zustand eines Tycoon-2-Systems gehören neben den Objekten auch Aktivierungen und Nachrichten. Eine Aktivierung wird durch eine Nachricht ausgelöst, und kann wiederum eine Nachricht versenden, auf deren Ergebnis sie dann wartet. Im Systemzustand sieht man daher zu jedem Zeitpunkt einen Stapel, der abwechselnd Aktivierungen und Nachrichten enthält. An der Spitze des Stapels befindet sich die einzige handlungsfähige Aktivierung, die ein Ergebnis berechnet haben kann. Das Ergebnis ist sehr flüchtig und nur zu dem Zeitpunkt zwischen Beendigung einer Aktivierung und Fortführung der auf das Ergebnis wartenden Aktivierung beobachtbar.

Eine Nachricht ist ein Exemplar des aufgerufenen Methodentyps, insofern als die in der Nachricht transportierten Argumentbindungen den Parametersignaturen entsprechen. Genauso ist die Aktivierung einer Methode ein Exemplar dieser Methode, und das Ergebnis einer Aktivierung ist ein Exemplar der deklarierten Ergebnissignatur.

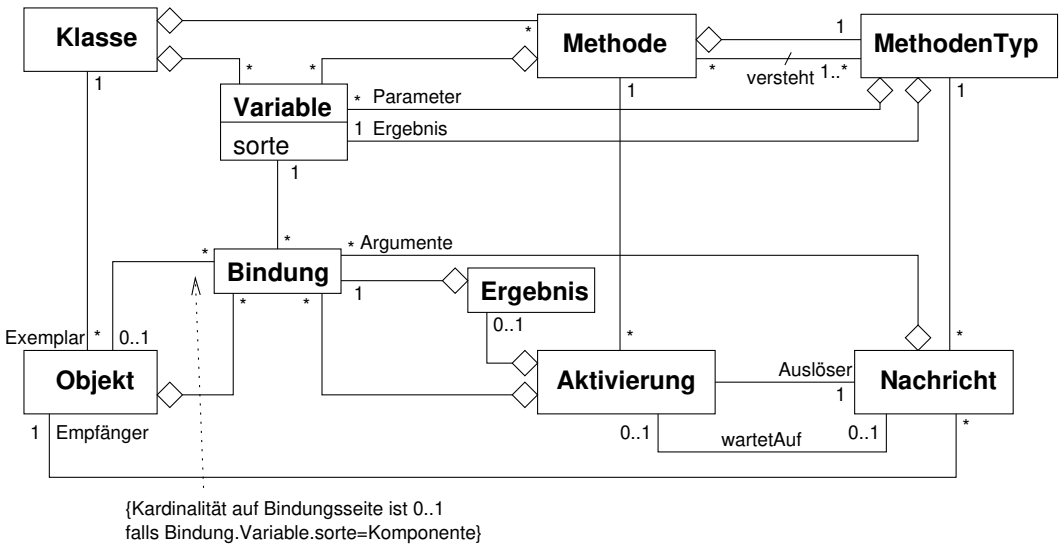

Abbildung 4.11.: Modell des Systemzustandes

Das dargestellte Modell ist etwas vereinfacht, insofern als weder indizierte Objekte (*Array*, *String*) noch Funktionsabschlüsse auftreten. Zu indizierten Objekten sei nur gesagt, daß die Sorte aller enthaltenen Bindungen gleich ist, ansonsten kann man sich ein *Array* vorstellen als Objekt mit den berechenbaren Exemplarvariablennamen $1 \ldots n$. Ein Funktionsabschluß fängt die Bindungen aller freien Variablen seines Funktionsausdrucks ein. Ein Funktionsausdruck tritt sowohl in der Rolle der Klasse auf, weil die Funktionsabschlüsse seine Exemplare sind, als auch in der Rolle der Methode, weil bei jedem Aufruf eine Aktivierung erzeugt wird. Wie durch Javas *inner classes* demonstriert, kann

ein Funktionsausdruck im Rahmen des bestehenden Objektmodells durch eine Klasse dargestellt werden, die eine Exemplarvariable für jede freie Variable des Funktionsausdrucks und als einzige Methode den Funktionsausdruck enthält. Funktionen werden in Abschnitt 5.5 noch einmal behandelt.

Kommen nun dynamische Komponenten ins Spiel, kann jede Variable entweder eine Komponenten- oder eine Referenzvariable sein. Dies wird im Diagramm wie gehabt durch das Attribut `sort` ausgedrückt. Die Sorte einer Bindung ergibt sich aus der Sorte der zugehörigen Variable.

Da jedes Objekt Ziel höchstens einer Komponentenbindung sein kann, und aufgrund der Aggregation jede Bindung nur entweder zu einem Objekt, einer Aktivierung, einem Ergebnis oder einer Nachricht gehört, kommt zu jedem Zeitpunkt nur eine dieser vier Arten von Superkomponente in Frage. Die möglichen Zustände eines Objekts und ihre Übergänge sind in Abbildung 4.12 in einem UML-Zustandsdiagramm zu sehen.

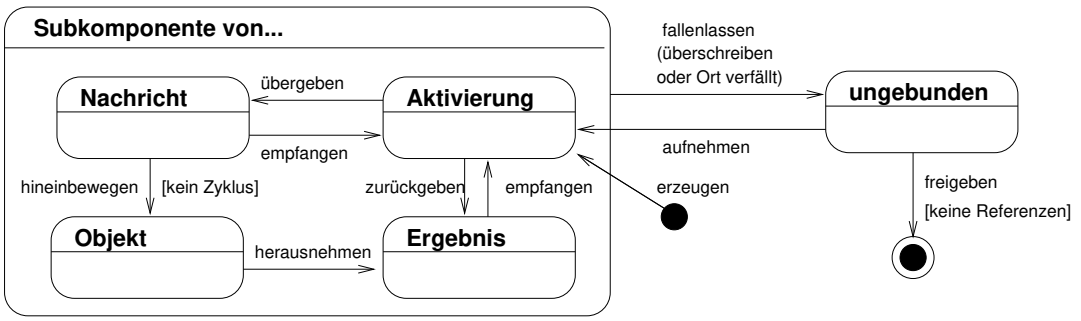

Abbildung 4.12.: Objektzustände in Bezug auf Komponentenbindung

Bei der Erzeugung erhält die erzeugende Aktivierung eine Komponentenbindung an das Objekt, die sie anschließend in einer Nachricht oder einem Ergebnis an eine andere Aktivierung übergeben kann. Durch Aufruf einer durch Definition einer Komponenten-Exemplarvariable generierten *Hineinbewegen*-Methode kann das Objekt an eine Komponenten-Exemplarvariable gebunden werden. Durch Aufruf der zugehörigen *Herausnehmen*-Methode verläßt es die Bindung wieder.

Läuft die Lebenszeit einer nicht-leeren Komponentenbindung ab, oder wird ein neues Objekt hineinbewegt, so existiert keine Komponentenbindung an das bisher gebundene Objekt mehr, man sagt das Objekt wird fallengelassen. Das Objekt befindet sich dann im Zustand *ungebunden*, und kann freigegeben werden, wenn keine Referenzbindungen mehr existieren. Mit Hilfe der oben beschriebenen Laufzeitmethode *fetch* kann ein ungebundenes Objekt ausgehend von einer Referenzbindung wieder aufgenommen werden. Befindet sich das Objekt nicht im Zustand *ungebunden*, wird zur Laufzeit die Ausnahme *FetchBoundComponent* ausgelöst.

Wird versucht, durch Aufruf einer *Hineinbewegen*-Methode eine zyklische Komponentenstruktur herzustellen, so wird zur Laufzeit die Ausnahme *CyclicComponent* ausgelöst, und die Operation findet nicht statt.

Es ist nicht ganz klar, welcher Superkomponente Funktionsabschlüsse zugeordnet werden sollen. Auf der einen Seite sollten sie wie andere erzeugte Objekte als Subkomponente der Aktivierung entstehen; auf der anderen Seite ist es umständlich und hinderlich, bei jedem Umgang mit Funktionsabschlüssen mit dem Klammeraffen zu hantieren. Oft macht es auch Sinn, einen Funktionsabschluß als Teil des Objekts zu betrachten, in dessen Kontext er erzeugt wurde (dem *static origin* in BETA-Terminologie). Möglicherweise sind Funktionsabschlüsse auch eher Werte als Objekte; in diesem Fall kann gar nicht sinnvoll von einer Superkomponente gesprochen werden. Ich habe die Beantwortung der aufgeworfenen Fragen vermieden und festgelegt, daß ein Funktionsausdruck immer eine Referenzbindung auf den erzeugten Funktionsabschluß liefert, so daß Funktionsabschlüsse immer ungebunden sind.

4.7. Anpassung der Standardbibliothek

In diesem Abschnitt werden die zum Umgang mit dynamischen Komponenten benötigten bzw. zur Unterstützung bereitgestellten Klassen und Methoden der Standardbibliothek beschrieben.

4.7.1. Methoden in der Klasse `Object`

Jede in Tycoon-2 definierte Klasse erbt implizit von `Object`. Der öffentliche Teil der Schnittstelle wird daher von jedem Objekt verstanden, die Implementierung kann im allgemeinen aber redefiniert werden. Private Methoden aus `Object` werden ähnlich einer Prozedur-Bibliothek ererbt, und sind nur für die Implementierung der Funktionalität einer Klasse interessant. Durch private Methoden werden auch Sprachmittel zur Verfügung gestellt, die nicht Teil des Sprachkerns sind, z.B. Kontrollstrukturen, Zusicherungen und Ausnahmebehandlung. Es ergibt sich eine gewohntere Syntax, da kein Empfänger angegeben werden muß. Wie eingangs erwähnt finden sich hier auch einige Methoden im Zusammenhang mit dynamischen Komponenten.

4.7.1.1. drop

Da Komponentenbindungen relativ kostbar sind - wenn ein Objekt erst einmal fallengelassen ist, kann es nicht sicher wieder aufgenommen werden - ist die Spracherweiterung so gewählt, daß Komponentenbindungen nicht leicht verloren gehen. Ein Komponentenausdruck kann weder an einer ignorierten Position in einer Sequenz auftreten noch an einer Argumentposition, an der ein Referenzargument erwartet wird, noch in einer Zuweisung an eine Referenzvariable. Wenn der Programmierer tatsächlich wünscht, das Objekt fallenzulassen, so kann er dies mit Hilfe der Methode *drop* angeben. *drop* ist eine private Methode von `Object` und hat folgende Signatur:

drop(T <: Void, x @: T) :T

Die Methode *drop* erwartet also eine Komponentenbindung von beliebigem Typ T, und liefert eine Referenzbindung an dasselbe Objekt zurück. Eine Verwendung könnte wie folgt aussehen.

```
aktualisiereAnlage(alt :Dokument, neu @: Dokument) :Void {
  if(anlagen.includes(alt) then: {
    drop(anlagen.takeOut(alt)@)
  })
  anlagen.add(neu@)
}
```

Derselbe Effekt läßt sich weniger deutlich auch durch Einführung einer lokalen Variable erzielen, deren Inhalt am Ende der Methode implizit fallengelassen wird. Die folgende Variante wird jedoch nicht empfohlen, und deutet im allgemeinen eine mögliche Fehlerquelle an.

```
if(anlagen.includes(alt) then: {
  ignoriert @: Dokument := anlagen.takeOut(alt)@
})
```

4.7.1.2. fetch

Soll aus einer Referenzbindung eine Komponentenbindung rekonstruiert werden, geschieht dies mit der öffentlichen Methode *fetch*. Falls das referenzierte Objekt nicht ungebunden ist, wird eine Ausnahme `FetchBoundComponent` ausgelöst. Der Typ des Methodenergebnisses ist der Typ des Empfängers, *Self*.

```
fetch @: Self
```

Ich gebe kein Beispiel für die Verwendung von *fetch* an, da der Aufruf von *fetch* ähnlich wie eine explizite Typumwandlung (*type cast*) im Normalfall nicht nötig ist.

4.7.1.3. superComponent

Das System bietet die Möglichkeit, mit Hilfe der öffentlichen Methode *superComponent* zur Laufzeit die Superkomponente eines beliebigen Objektes zu erfragen. Ist es Subkomponente eines anderen Objekts, so wird eine Referenz auf dieses Objekt zurückgegeben. Andernfalls wird *nil* geliefert.

```
superComponent :Object
```

Diese Methode gehört zu den Reflektionsmöglichkeiten des Systems. Sie wird unter anderem von dem in Kapitel 6 beschriebenen Visualisierungswerkzeug benutzt. Um Zyklenprüfung implementieren zu können muß die Information ohnehin vorhanden sein, so daß ihre Bereitstellung per Reflektion keinen zusätzlichen Aufwand erfordert.

Im folgenden Beispiel werden die Klassen aller transitiven Superkomponenten des Empfängers ausgegeben.

```
printSupercomponents(out :Output) :Void {
  o ::= self
  while({ o.isNotNil } do: {
    out << o.class.name << "\n"
    o := o.superComponent
  })
}
```

4.7.1.4. copy

Die Signatur der öffentlichen Methode *copy* ist

```
copy @:Self
```

Das Ergebnis hat also denselben Typ wie der Empfänger. Um die Invariante aufrechtzuerhalten, daß ein Objekt höchstens eine Superkomponente hat, werden beim Kopieren wie schon in Abbildung 4.3 dargestellt alle Subkomponenten mitkopiert. Außerdem werden alle Verweise innerhalb der dynamischen Komponente in der Kopie angepaßt. Die Kopie eines Objekts o läßt sich genauer wie folgt beschreiben:

1. Jedem von o aus transitiv über Komponentenbindungen erreichbaren Objekt s wird durch eine Abbildung σ ein neu angelegtes Exemplar $s' =: \sigma(s)$ der Klasse von s zugeordnet. Es ist $Dom(\sigma)$ die dynamische Komponente von o.

2. Die Bindung einer Exemplarvariable v in einem Objekt $s' = \sigma(s)$ ergibt sich aus der Bindung von v in s wie folgt:

   - Ist v eine Komponentenvariable, so treten zwei Fälle auf.
     (a) Enthält v in s eine Komponentenbindung an ein Objekt s_v, so erhält v in s' die Komponentenbindung an $\sigma(s_v)$.
     (b) Ist v in s *nil*, so ist v auch in s' *nil*.
   - Ist v eine Referenzvariable, so sind drei Fälle zu unterscheiden.
     (a) Enthält v in s eine Referenzbindung an ein Objekt $s_v \in Dom(\sigma)$, so erhält v in s' eine Referenzbindung an $\sigma(s_v)$.
     (b) Enthält v in s eine Referenzbindung an ein Objekt $s_v \notin Dom(\sigma)$, so erhält v in s' eine Referenzbindung an s_v.
     (c) Ist v in s *nil*, so ist es auch in s' *nil*.

4.7.1.5. doesNotUnderstand, perform

Die Methoden *doesNotUnderstand* und *perform* sind nicht eigentlich Teil der Komponentenerweiterung, ihre Signatur mußte aber verändert werden.

Die Methode *doesNotUnderstand* eines Objekts wird aufgerufen, wenn das Objekt eine Nachricht empfängt, für die keine Implementierung zur Verfügung steht. Als Argument wird eine Beschreibung der empfangenen Nachricht übergeben. Normalerweise löst die Methode einfach eine Ausnahme aus. Sie kann jedoch in einer Subklasse überschrieben werden. Wenn sie ein Ergebnis liefert, wird dieses zum Ergebnis der nicht verstandenen Nachricht.

Die Methode *perform* erwartet als Argument die Beschreibung einer Nachricht, und sendet die beschriebene Nachricht an ihren Empfänger. Dessen Ergebnis wird zum Ergebnis des *perform*-Aufrufes.

Durch die Einführung von dynamischen Komponenten haben sich Nachrichten in zweierlei Hinsicht verändert: Zum einen wird zur Auswahl der aufzurufenden Methode auch die erwartete Ergebnissorte benutzt, und zum anderen können als Argumente und Ergebnis sowohl Referenz- als auch Komponentenbindungen auftreten. Diese Information ist in dem übergebenen Exemplar der Klasse `Selector` kodiert. Die Schnittstellen sind wie folgt:

;; öffentlich:
perform(selector :Selector, args :Array(Object)) :Object
;; privat:
_doesNotUnderstand(selector :Selector, :Array(Object)) :Object

Da ein *Array* nur Referenzbindungen enthält, findet vor dem Aufruf von *_doesNotUnderstand* ein implizites *drop* aller Komponentenargumente statt, und beim Aufruf von *perform* wird ein implizites *fetch* auf alle Argumente für Komponentenparameter angewendet. Dadurch kann auch eine Ausnahme ausgelöst werden. Analog läßt *perform* vor der Rückgabe eines Komponentenergebnisses das Objekt fallen, und das Ergebnis von *_doesNotUnderstand* wird gegebenenfalls aufgenommen, bevor es an den ein Komponentenergebnis erwartenden Aufrufer weitergegeben wird. Auch hier kann es zu einer Ausnahme kommen.

Im obigen Modell des Systemzustandes werden Nachrichten als Exemplar eines Methoden-Typen angesehen. Der Selektor nimmt dabei die Rolle dieses Methoden-Typen ein, und bestimmt damit auch die Sorten der in der Nachricht enthaltenen Bindungen. Die Klasse *Selector* sieht idealisiert wie folgt aus:

class *Selector* {
  name :String
  positionalValueParameters :Array(Bool)
  keywordParameters :Array(Pair(String,Bool))
  result :Bool
}

Im Falle der Ergebnisse wäre es möglich gewesen, jeweils zwei Varianten von *_doesNotUnderstand* und *perform* anzubieten, die ein Referenz- bzw. Komponentenergebnis liefern. Aus Symmetriegründen habe ich darauf jedoch verzichtet, da davon auszugehen ist, daß Nachrichten mit verschiedenen Ergebnissorten im wesentlichen gleich

behandelt werden. Zum Beispiel ist es so möglich, eine Nachricht unabhängig von der Ergebnissorte direkt weiterzuleiten:

```
class Forwarder {
 private
  destination :Object
  _doesNotUnderstand(selector :Selector, args :Array(Object)) :Object {
    destination.perform(selector, args)
  }
}
```

Der Preis dafür ist ein leicht erhöhter Laufzeitaufwand, der im Vergleich zum Rest des Reflektionsmechanismus aber nicht ins Gewicht fallen dürfte.

4.7.1.6. if@, try@

Kontrollstrukturen werden im Tycoon-2-System mit Hilfe von privaten Methode der Klasse `Object` zur Verfügung gestellt. Die Schnittstelle der Methode *if* sieht so aus:

```
if(T <: Void, cond :Bool
  then: trueCase :Fun():T
  else: falseCase :Fun():T) :T
```

Das liest man wie folgt: In einem Aufruf der Methode *if* können neben der Bedingung *cond* optional mit Hilfe von Schlüsselworten ein *then*- und ein *else*-Zweig in Form eines Funktionsabschlusses angegeben werden. Der Ergebnistyp des *if*-Aufrufs ist der gemeinsame Ergebnistyp der übergebenen Funktionsabschlüsse.

Das beabsichtigte Verhalten ist, daß die Methode abhängig vom Wert von *cond* entweder *trueCase* oder *falseCase* aufruft, und das Ergebnis des Funktionsaufrufs als Ergebnis des *if*-Aufrufs zurückgibt.

Dieses Verhalten ist nicht von der Ergebnissorte abhängig, es gibt also keinen Grund, weshalb ein Konditional nicht auch eine Komponentenbindung liefern sollte. Da es aber nicht möglich ist, die Methode wie mit dem Ergebnistyp T mit der erwarteten Ergebnissorte zu parametrisieren, muß die Schnittstelle (und die Implementierung) verdoppelt werden. Die Komponenten-Variante der Schnittstelle lautet dann wie folgt:

```
if(T <: Void, cond :Bool
  then: trueCase :Fun()@:T
  else: falseCase :Fun()@:T) @:T
```

Eine ähnliche Verdopplung muß für die Ausnahmebehandlung stattfinden. Die eingebaute Methode *try* ruft einen übergebenen Funktionsabschluß auf. Wenn dieser ein Ergebnis liefert, wird es direkt als Ergebnis der *try*-Methode zurückgegeben. Liefert der Funktionsaufruf aber eine Ausnahme, so wird der zweite übergebene Funktionsabschluß mit dem Ausnahmeobjekt als Argument aufgerufen, und dessen Ergebnis oder Ausnahme zurückgegeben. Die Ergebnissorte ist für den internen Ablauf wieder unerheblich.

try(T <: Void, body: Fun():T
else: Fun(e :Exception):T) :T
try(T <: Void, body: Fun()@:T
else: Fun(e :Exception)@:T) @:T

4.7.2. Behälterklassen für Komponenten

Die Standardbibliothek von Tycoon-2 enthält einen reichen Vorrat an Behälterklassen: Listen, Felder (von fester und variabler Größe, sortiert und unsortiert), Hashtabellen, Stapel, Warteschlangen und Mengen. Die Bibliothek definiert außerdem einige gemeinsame Schnittstellen, z.B. `KeyedContainer` für einen Behälter, auf dessen Elemente mit Schlüssel zugegriffen werden kann, und `Sequence` für einen `KeyedContainer` mit den Schlüsseln $1 \ldots n$.

Behälterklassen setzen in der Entwurfsebene geforderte Assoziationen mit höherer Kardinalität auf der Implementierungsebene um. Abbildung 4.13 zeigt einige Entsprechungen zwischen Klassendiagramm und Programmtext.

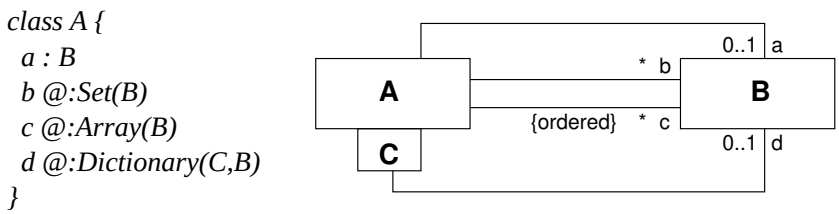

Abbildung 4.13.: Kardinalität und Behälterklassen

Die Behälterobjekte werden als Subkomponenten des Exemplars von *A* deklariert. Einem Objekt auf der Modellebene stehen mehrere Objekte in der Implementierung gegenüber, daher ist es sinnvoll, diese Objekte als eine dynamische Komponente zu gruppieren.

Durch die angegebene Komponentenbindung werden nur die Behälterobjekte, nicht aber ihre Elemente zu Subkomponenten. Die Sorte der Bindung an das Behälterobjekt und der Bindungen an die Elemente müssen also getrennt spezifiziert werden. Da es in der gewählten Spracherweiterung nicht möglich ist, eine Klasse mit Sorten zu parametrisieren, müssen sämtliche Behälterklassen in zwei Varianten definiert werden. So enthält ein *Array* eine Sequenz von Referenzbindungen, während ein *AtArray* eine Sequenz von Komponentenbindungen enthält. Die Vorsilbe *At...* steht für die englische Aussprache des Klammeraffen. Abbildung 4.14 zeigt die Entsprechungen zwischen Aggregation im Klassendiagramm und Programmtext.

Die Schwächen dieser Umsetzung sollen nicht unerwähnt bleiben. Zum einen führt die Verdopplung der Behälterklassen zu Redundanz und damit zu erhöhtem Wartungsaufwand und Speicherbedarf. Das andere Problem liegt eher auf der konzeptuellen Ebene: Werden Komponenten-Behälterklassen eingesetzt, so ist die dynamische Komponente

```
class A {
  a @: B
  b @:AtSet(B)
  c @:AtArray(B)
  d @:AtDictionary(C,B)
}
```

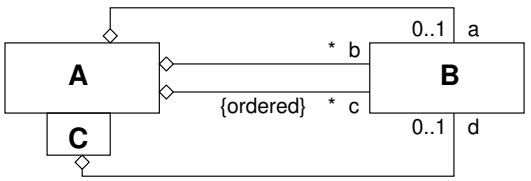

Abbildung 4.14.: Kardinalität von Aggregation und Komponenten-Behälterklassen

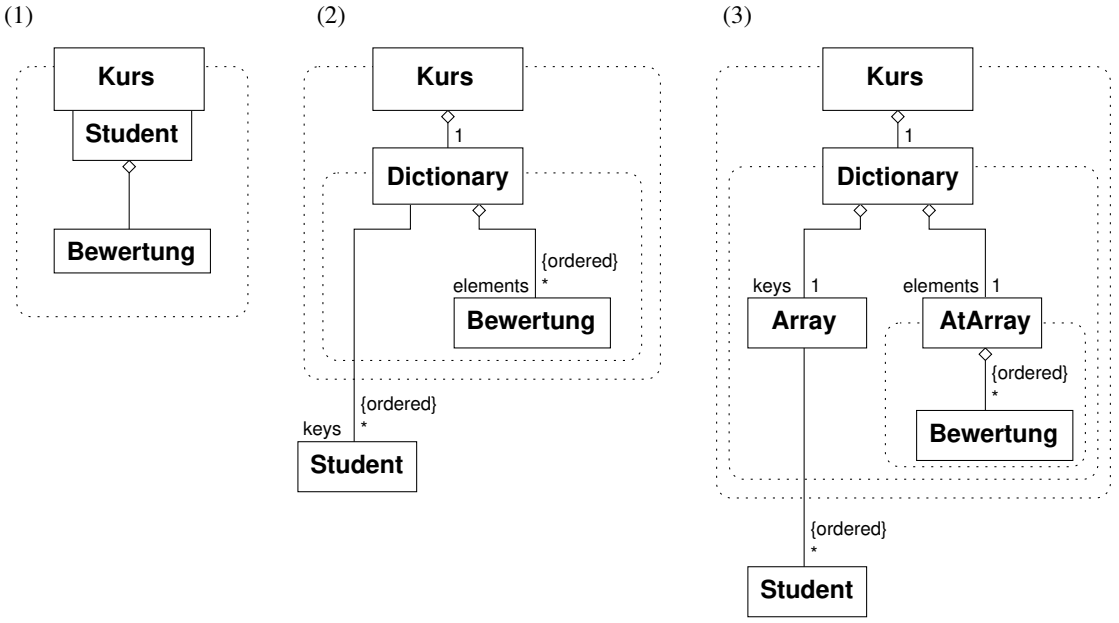

Abbildung 4.15.: Die Gruppierung durch Komponentenbindungen entspricht mit Komponenten-Behälterklassen nicht den beabsichtigen Granularitätsebenen.

eines Objekts wider Erwarten nicht mehr die Menge der Objekte, die gemeinsam seinen Zustand repräsentieren.

Das Problem liegt darin, daß nicht zwischen Bindungen zwischen dem Objekt und seiner Repräsentation und Bindungen zwischen dem Objekt und seinen Subkomponenten unterschieden wird. Zur Illustration dient das Beispiel aus Abschnitt 3.3.4. In Abbildung 4.15 wird die Repräsentation der aus Abbildung 3.3 bekannten Klasse *Kurs* schrittweise entwickelt. Ein Exemplar der Klasse *Bewertung* liegt bei zunehmender Verfeinerung nicht mehr direkt in der dynamischen Komponente von *Kurs*, sondern wird zur Subkomponente der internen Repräsentation der Behälterklassen. Durch Vergröberung von dynamischen Komponenten kann also nicht von der Repräsentation abstrahiert werden, ohne auch die Klasse *Bewertung* aus dem Blick zu verlieren.

5. Dynamische Komponenten im Tycoon-2-System

In diesem Kapitel wird die technische Umsetzung der im Kapitel 4 beschriebenen Spracherweiterung behandelt. Die Erweiterung zieht sich vertikal durch alle Ebenen: Sie betrifft lexikalische und syntaktische Analyse (Abschnitt 5.1), die Implementierung des abstrakten Syntaxbaumes (Abschnitt 5.2), den Typüberprüfungs-Algorithmus (Abschnitt 5.3), den Objektspeicher (Abschnitt 5.4) und die virtuelle Maschine und die darauf angepaßte Codeerzeugung (Abschnitt 5.5). Die Änderungen auf allen diesen Ebenen sollen im Folgenden in der angegebenen Reihenfolge beschrieben werden.

5.1. Lexikalische und Syntaktische Analyse

Im Tycoon-2-System wird wie in [Wienberg 97] beschrieben ein LALR(1)-Parser-Generator eingesetzt, um aus einer mit semantischen Aktionen versehenen Grammatik der Eingabesprache einen ablauffähigen Parser zu erzeugen. Um die Spracherweiterung nutzbar zu machen, müssen lediglich einige Regeln zu der Eingabe des Parser-Generators hinzugefügt werden. Die durch die semantischen Aktionen konstruierten abstrakten Syntaxbäume werden im anschließenden Abschnitt beschrieben.

Auch der Scanner wird von einem Generator erzeugt. Hier mußte nur der Klammeraffe (@) als terminales Symbol eingeführt werden. Die Zeichenfolge *@:* wird durch zwei terminale Symbole dargestellt (@ und :).

5.2. Abstrakter Syntaxbaum

Das Klassendiagramm des Abstrakten Syntaxbaums entspricht im wesentlichen den in den Abbildungen 4.5, 4.6 und 4.7 dargestellten Analyse-Diagrammen. Auf der Entwurfs- und Implementierungs-Ebene ergeben sich einige kleinere Abweichungen, die jedoch mit einer Ausnahme keinen Einfluß auf die Spracherweiterung haben.

Bei dieser Ausnahme handelt es sich darum, daß es in der Implementierung des abstrakten Syntaxbaums keine Klasse für Funktionstypen gibt. Diese werden intern als `ApplyType` dargestellt. In der bisherigen Fassung des Systems wurde für jede Stelligkeit

des Typs bis zu einer festgelegten Grenze eine Klasse definiert, die eine entsprechende Anzahl Typparameter erwartet. So wird z.B. der Funktionstyp

Fun(x1:T1, x2:T2, x3:T3):R

zu

Fun3(T1,T2,T3,R)

wobei die Klasse *Fun3* definiert ist als

```
class Fun3(P1<:Void, P2<:Void, P3<:Void, R<:Void)
meta AbstractClass
{ "[]"(:P1, :P2, :P3):R
    deferred
}
```

Diese Kodierung ist möglich, weil die aktuelle Fassung der Sprache keine Typparameter oder Schlüsselwortparameter in Funktionssignaturen erlaubt.[1]

Um nun aber Komponenten in Funktionssignaturen verwenden zu können, müssen auch die Sorten der Parameter und des Ergebnisses im Namen des Typoperators kodiert werden, so wie dies bisher mit der Stelligkeit geschehen ist. Zum Beispiel wird der Funktionstyp

Fun(x :S, y @:T)@:R

zu

FunRCC(S,T,R)

Diese Klasse ist wie folgt definiert:

```
class FunRCC(P1<:Void, P2<:Void, R<:Void)
meta AbstractClass
{ "[]"(:P1, @:P2)@:R
    deferred
}
```

Da alle diese Typoperatoren als Klassen definiert sein müssen, und die Anzahl möglicher Sorten-Kombinationen exponentiell mit der Parameterzahl wächst (2^{n+1} Kombinationen für n Parameter), legt das System eine niedrige Obergrenze für die Stelligkeit fest. Funktionstypen, die mehr als zwei Parameter haben, dürfen weder Komponentenparameter noch ein Komponentenergebnis deklarieren. Für diese Typen wird wie bisher nur die Stelligkeit im Namen kodiert.[2]

Ansonsten erwies sich die Erweiterung als sehr geradlinig. Die verschiedenen Phasen des Übersetzers sind mit Hilfe des *Interpreter*-Musters [Gamma et al. 95] als Methoden der Klassen des abstrakten Syntaxbaums definiert. Die Einführung einer neuen Klasse (`TakeFrom`) ist aufgrund des verwendeten Entwurfsmusters relativ einfach und erfordert nur lokale Änderungen, da nur lokal in der neuen Klassen die Methoden aller Übersetzer-Phasen implementiert werden müssen. Die Einführung einer neuen Übersetzer-Phase hingegen führt zu Änderungen an allen beteiligten Klassen.

[1] Tatsächlich hat hier (leider) die Implementierung die Sprachdefinition geleitet.
[2] Funktionen mit mehr Komponentenparametern können durch *currying* simuliert werden.

5.3. Typüberprüfung

Wie die anderen Phasen des Übersetzers ist die Typüberprüfung mit Hilfe des *Interpreter*-Musters auf die Klassen des abstrakten Syntaxbaumes verteilt.

Die Spracherweiterung ist so gewählt, daß die Sorte eines Ausdrucks auch ohne Kenntnis der beteiligten Typen bestimmt werden kann, sobald die Sichtbarkeitsbereiche berechnet worden sind. Bis auf die Regeln zum Methodenaufruf *Component Send* und *Val Send* können so die Sortenprüfungen aller in Abschnitt 4.5 angegebenen Regeln auch ohne Bestimmung der Typen stattfinden.

Beim Methodenaufruf besteht das Problem darin, daß der Typ des Empfängers bekannt sein muß, um die Parametersignaturen und damit auch die Parametersorten der ausgewählten Methode bestimmen zu können. Dies kann also erst während der eigentlichen Typprüfung geschehen. Um zur Laufzeit korrekte Sorten sicherzustellen, werden die schon während der Sortenprüfung bestimmbaren Argumentsorten im Aufruf kodiert und zur Laufzeit mit den Parametersorten der aufgerufenen Methode verglichen. Dabei kommt die schon in Abschnitt 4.7.1.5 erwähnte Klasse *Selector* zum Einsatz. Passen die Selektoren nicht zusammen, wird zur Laufzeit eine Ausnahme ausgelöst.

Die Berechnung und weitestmögliche Überprüfung der Sorten findet in einer konzeptuell eigenständigen Phase des Übersetzers zwischen der Berechnung der Sichtbarkeitsbereiche (*scoping*) und der Codeerzeugung statt. Ihr Ergebnis steht damit für die Codeerzeugung zur Verfügung.

5.4. Objektspeicher

Der Objektspeicher des Tycoon-2-Systems, wie er in [Weikard 98] beschrieben ist, ermöglicht über seine Schnittstelle die Allokation und den Zugriff auf Objekte. Er verfügt über eine automatische Speicherbereinigung und ist zudem in der Lage, ein persistentes Abbild des Systemzustandes auf einem nichtflüchtigen Speichermedium zu sichern und wiederherzustellen. Zu dem gespeicherten Systemzustand gehören neben Objekten und Code auch die Ausführungszustände laufender Tycoon-2-Threads.

5.4.1. Verweis auf die Superkomponente

Um eine effiziente Zyklenprüfung zu ermöglichen, ist der Objektspeicher so erweitert, daß jedes Objekt einen Verweis auf seine Superkomponente enthält. Zur Unterstützung der Methode *fetch* des Laufzeitsystems ist außerdem festgehalten, ob ein Objekt gerade ungebunden ist.

Der Objektspeicher verwaltet jedes alloziierte Objekt mit Hilfe eines Objektkopfes (*object header*), in dem unter anderem die Klasse und Größe des Objektes festgehalten sind.

Dieser Objektkopf erhält nun zusätzlich einen Verweis auf die Superkomponente. Ist das Objekt ungebunden, so wird als Superkomponente *nil* eingetragen.

Die im Objektmodell beschriebenen Nachrichten, Aktivierungen und Ergebnisse (siehe Abbildung 4.11) sind im Objektspeicher nicht als eigenständige Objekte vorhanden. Sie werden zu einem Stapel zusammengefaßt, dargestellt durch ein Objekt der Klasse *Stack*. Daher kann für ein Objekt in einem der Zustände *Subkomponente von Nachricht, Aktivierung* oder *Ergebnis* die Quelle der Komponentenbindung nicht genau angegeben werden. Stattdessen wird ein Verweis auf den Thread eingetragen, zu dem der Stapel (*Stack*) gehört. Da ein Objekt der Klasse *Thread* nie in der Rolle der Superkomponente auftritt, kann keine Verwechslung mit den Zuständen *ungebunden* oder *Subkomponente von Objekt* vorkommen.

Der durch das zusätzliche Wort pro Objekt verursachte Zuwachs im Speicherbedarf des Systems beträgt ca. zehn Prozent. Dies entspricht einer durchschnittlichen bisherigen Objektgröße von zehn Worten inklusive Verwaltungsmehraufwand.

5.4.2. Unabhängigkeit der Lebenszeiten

In Kapitel 2 ergab sich die Anforderung, daß eine Subkomponente ihre Superkomponente nicht am Leben erhält. Läuft die Lebenszeit der Superkomponente ab, so wird die Subkomponente ungebunden. Der Verweis auf die Superkomponente ist also eine schwache Referenz (*weak reference*). Da der Objektspeicher für die Speicherbereinigung zuständig ist, findet sich hier auch die Umsetzung dieser Forderung.

Während der Speicherbereinigung werden zur Bestimmung der noch erreichbaren Objekte ausgehend von einem Wurzelobjekt die im Objektspeicher vorhandenen Bindungen verfolgt. Die im Objektkopf enthaltenen Superkomponenten-Verweise werden dabei zunächst nicht berücksichtigt. Erst wenn feststeht, welche Objekte die Speicherbereinigung überlebt haben, wird in einer eigenen Phase für jedes Objekt der Verweis auf die überlebende Superkomponente angepaßt oder auf *nil* gesetzt, falls die Superkomponente freigegeben wurde. Der Aufwand ist proportional zur Anzahl der überlebenden Objekte.

Um die Lokalität dieses Algorithmus zu verbessern und eventuell auch Speicherplatz zu sparen, hätten die Superkomponenten-Verweise auch außerhalb der Objekte, z.B. in einer Hashtabelle gespeichert werden können. Dieses Vorgehen führt jedoch bei jeder Veränderung des Komponenten-Zustandes eines Objekts zu erheblichem Laufzeitaufwand, was der häufigen Verwendung von Komponentenbindungen im Wege steht.

5.4.3. Repräsentation von Komponentenbindungen

Komponentenbindungen werden im Speicher genauso repräsentiert wie Referenzbindungen, um Laufzeitaufwand durch Hinzufügen und Entfernen von Markierungen zu vermeiden (*tagging/untagging*). In Objekten kann bei Bedarf anhand der Exemplarvariablen

der Klasse rekonstruiert werden, welche Bindungen Komponentenbindungen und welche Referenzbindungen sind. Dies entspricht der Darstellung im Objektmodell (Abbildung 4.11), wo die Sorte einer Bindung durch die Sorte der bindenden Variable bestimmt wird.

Auf dem Stapel ist es nicht so einfach möglich, die bindende Variable einer Speicherstelle zu bestimmen, da diese je nach Ausführungszustand wechseln kann. Der ausgeführte Code kann z.B. eine Bindung für eine lokale Komponenten- oder Referenzvariable auf dem Stapel anlegen. Wenn die Lebenszeit der Variablen abläuft, kann die Speicherstelle für die Bindung einer anderen Variable, eventuell auch von einer anderen Sorte, wiederverwendet werden. Letztendlich hängt die Sorte der Speicherstelle also von dem in der Aktivierung bereits ausgeführten Code ab. Die Sorte könnte durch Analyse des Codes rekonstruiert werden; eine Unterstützung durch statisch vom Übersetzer erzeugte Tabellen ist denkbar.

Im implementierten System ist diese Möglichkeit nicht umgesetzt. Die Sorte einer auf dem Stapel liegenden Bindung wird zur Laufzeit nie festgestellt. Insbesondere prüft der Objektspeicher nie, ob eine Komponentenvariable auch als solche gehandhabt wird. Die Verantwortung hierfür liegt bei der virtuellen Maschine und bei der Codeerzeugung. Die virtuelle Maschine trägt auch dafür Sorge, daß der im Objektkopf gespeicherte Objektzustand in Bezug auf Komponentenbindung immer aktuell ist. Änderungen werden dem Objektspeicher durch Aufruf der Funktion `tsp_setSuperComponent` mitgeteilt.

5.4.4. Ausrichtung

Die Schnittstelle des Objektspeichers garantiert, daß alloziierte Objekte auf einer durch acht teilbaren Adresse beginnen. Der Grund dafür liegt darin, daß in Objekten 64 Bit breite Datenworte gespeichert werden können, und einige Maschinenarchitekturen für diese eine Ausrichtung (*alignment*) von acht fordern. Der bisherige Objektkopf hatte genau eine Größe von acht Bytes, so daß für den Objektkopf dieselben Anfangsadressen wie für die Nutzdaten in Frage kamen. Durch die Erweiterung um ein zusätzliches Wort ist dies nicht mehr der Fall. In der Implementierung des Objektspeichers wird nun stets auf die Ausrichtung der Nutzdaten und nicht auf die der Objektköpfe geachtet.

Der Objektspeicher ist in Seiten (*pages*) von z.Zt. 512 Bytes unterteilt. Am Anfang jeder Seite bleibt aufgrund des Zusammenwirkens von Ausrichtung und Objektkopfgröße ein Wort Speicher ungenutzt. Abbildung 5.1 illustriert diesen Zusammenhang. Ansonsten entstehen geringe Verluste durch verstärkte Fragmentierung, da die durchschnittliche Allokationsgröße ein Wort mehr als ohne den Superkomponentenverweis beträgt.

5.4.5. Lokalität

Da davon auszugehen ist, daß die Objekte einer dynamischen Komponente eng zusammenarbeiten, kann die Speicherlokalität des Systems gesteigert werden, indem Objekte

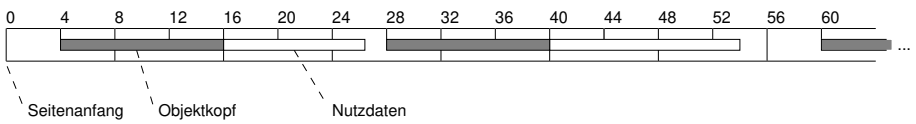

Abbildung 5.1.: Ausrichtung von Objekten

der selben dynamischen Komponente in aufeinanderfolgenden Speicherbereichen abgelegt werden [Stoutamire 97]. Dabei handelt es sich nur um eine Heuristik; das System bleibt funktionsfähig, egal wo die Objekte gespeichert sind.

Die im Tycoon-2-System implementierte Speicherbereinigung gehört in die Gruppe der kopierenden Speicherbereinigungsalgorithmen. Bei jeder Speicherbereinigung werden alle als lebendig erkannten Objekte umkopiert. Hierbei ergibt sich die Gelegenheit, auch ihre Anordnung zu beeinflussen.

Der vorhandene Algorithmus führt eine Breitensuche auf dem Objektgraph aus. Diese führt im allgemeinen zu schlechter Zugriffslokalität, da der Benutzerprozeß (der *mutator* in der Terminologie der *garbage collection*) bei aufeinanderfolgenden Objektzugriffen gewöhnlich Verweisen folgt, sein Zugriffsmuster also eher der Tiefensuche entspricht. Kopierende Speicherbereinigung mit Tiefensuche wird selten implementiert, da sie während des Kopierens die Verwaltung eines Stapels erfordert, der im schlimmsten Fall einen Eintrag für jedes Objekt des Objektspeichers enthält.

Ich schlage vor, einen Hybrid-Algorithmus zu verwenden, der für Komponentenbindungen Tiefensuche benutzt, Referenzbindungen aber wie gehabt durch Breitensuche verfolgt. Um den Speicherbedarf des hierzu erforderlichen Stapels zu begrenzen, kann eine Maximaltiefe festgelegt werden, bis zu der Komponentenbindungen verfolgt werden. Auch wenn so nicht garantiert die gesamte dynamische Komponente in aufeinanderfolgenden Speicherbereichen liegt, wird die Lokalität im allgemeinen doch gesteigert.

Es wäre zu prüfen, ob der Zusatzaufwand bei der Speicherbereinigung durch die gesteigerte Zugriffslokalität ausgeglichen wird. Vorraussetzung ist, daß im Objektspeicher Komponentenbindungen tatsächlich breiten Einsatz finden. Da dies zur Zeit noch nicht der Fall ist, wurde der skizzierte Algorithmus nicht implementiert.

5.5. Virtuelle Maschine und Codeerzeugung

Da Typdeklarationen im bisherigen System keinen Einfluß auf das Laufzeitverhalten haben, werden sie bei der Codeerzeugung ignoriert. Bei der Deklaration von Sorten liegt der Fall jedoch anders: Um den Komponenten-Zustand eines Objekts im Objektspeicher immer auf dem aktuellen Stand halten zu können, braucht die virtuelle Maschine spezielle Befehle für die Manipulation von Komponentenvariablen.

Zugriffsart	Methodenklasse
Anlegen	–[3]
Referenzieren	`SlotReferenceMethod`
Herausnehmen	`SlotTakeoutMethod`
Hineinbewegen	`SlotMoveToMethod`
Freigeben	–[4]

Tabelle 5.1.: Methodenklassen für den Zugriff auf Komponenten-Exemplarvariablen

Zugriffsart	Variablenart		
	lokale Variable	Parameter	entkommende Variable
Anlegen	–[5]	–[6]	`componentCellNew`
Referenzieren	`referenceLocal`	`referenceArgument`	`referenceCell`
Herausnehmen	`takeoutLocal`	`takeoutArgument`	`takeoutCell`
Hineinbewegen	`moveToLocal`	–[7]	`moveToCell`
Freigeben	`componentPop`, `componentAdjust`	–[8]	–[9]

Tabelle 5.2.: Bytecodes für Komponentenvariablen

5.5.1. Zugriffsarten

Für den Zugriff auf Variablen verwendet das Tycoon-2-System zwei verschiedene Mechanismen: Auf statisch gebundene Variablen (lokale Variablen, Parameter und freie Variablen) wird durch Ausführung von *bytecodes* zugegriffen, während der Zugriff auf dynamisch gebundene Variablen (Exemplarvariablen und globale Variablen) durch den Aufruf spezieller Methoden stattfindet. Für jede Kombination aus Zugriffs- und Variablenart existiert ein eigener *bytecode* bzw. eine eigene Methodenklasse.

Durch die Einführung von Komponentenvariablen und die damit einhergehenden neuen Zugriffsarten werden sowohl neue *bytecodes* als auch neue Methodenklassen nötig. Diese sind in Tabelle 5.1 und Tabelle 5.2 dargestellt. Die von der virtuellen Maschine jeweils auszuführenden Operationen werden im folgenden beschrieben.

Wird der Inhalt einer Komponentenvariable destruktiv gelesen (herausgenommen), so wird das gegebenenfalls enthaltene Objekt Subkomponente der aktuellen Aktivierung. Da im Objektspeicher kein Unterschied zwischen der Subkomponente einer Aktivierung,

[3] Die Komponentenbindung wird gemeinsam mit dem umgebenden Objekt angelegt.
[4] Die Komponentenbindung wird gemeinsam mit dem umgebenden Objekt von der Speicherbereinigung freigegeben.
[5] Die Komponentenbindung wird als Ergebnis des gebundenen Ausdrucks erzeugt.
[6] Die Komponentenbindung wird als Ergebnis des Argumentausdrucks erzeugt.
[7] Zuweisung an Parametervariablen ist in Tycoon-2 nicht erlaubt.
[8] Komponenten-Parameter werden sofort nach Aufruf der Methode destruktiv gelesen und sind am Methodenende daher immer leer.
[9] Entkommende Bindungen werden von der Speicherbereinigung freigegeben.

einer Nachricht oder eines Ergebnisses gemacht wird, wird der Objektspeicher beim Herausnehmen aus solchen Variablen nicht benachrichtigt. Ein Aufruf an den Objektspeicher findet nur beim destruktiven Lesen einer Exemplarvariable statt.

Beim Hineinbewegen eines Objekts aus der aktuellen Aktivierung in eine existierende Komponentenbindung wird deren vorheriger Inhalt ungebunden. Dies gilt für alle Variablenarten. Außerdem wird das hineinbewegte Objekt zu einer Subkomponente des jeweiligen Zieles. Wie beim Herausnehmen wird der Objektspeicher nur beim Hineinbewegen in eine Exemplarvariable benachrichtigt.

Beim Referenzieren einer Komponentenvariable wird aus der Komponentenbindung eine Referenzbindung erzeugt. Da die Repräsentation in beiden Fällen gleich ist, hat Referenzieren dieselbe Implementierung wie das Auslesen einer Referenzvariable. Um die virtuelle Maschine und Codeerzeugung von der Repräsentation von Komponentenbindungen unabhängig zu halten, wird das Referenzieren dennoch als eigenständiger Befehl behandelt.

Wenn die Lebenszeit einer Komponentenbindung endet, wird ein noch gebundenes Objekt fallengelassen. Für Exemplarvariablen ist dieser Zeitpunkt nur der Speicherbereinigung bekannt, für lokale Variablen und Nachrichten gehorchen die Lebenszeiten jedoch der Stapelstruktur. Es werden daher Befehle benötigt, um stapelbasierte Komponentenbindungen freizugeben.

5.5.2. Endrekursion

Ein Nachteil dieser Behandlung stackbasierter Komponentenvariablen besteht darin, daß solche Variablen Endrekursion verhindern. Da am Ende des Sichtbarkeitsbereiches noch Aufräumaktionen nötig sind, kann die Kontrolle nicht vollständig an eine an Endposition aufgerufene Methode übergeben werden. Ein optimierender Übersetzer könnte in vielen Fällen erkennen, daß eine Komponentenbindung garantiert leer ist und ihr Inhalt daher nicht mehr fallengelassen werden muß. In der vorhandenen einfachen Übersetzerstruktur war eine derartige Optimierung nicht möglich.

5.5.3. Ausnahmen

Eine Komplikation entsteht im Zusammenspiel mit Ausnahmen. Die Lebenszeit einer stapelbasierten Variable endet nicht nur, wenn das Bindungskonstrukt regulär verlassen wird, sondern auch, wenn es mit einer Ausnahme abgebrochen wird. Es muß also eine Ausnahmebehandlung für den Bereich des Bindungskonstuktes hinzugefügt werden, die die Bindung freigibt und die Ausnahme dann erneut auslöst.

Das gilt nicht nur für lokale Variablen. Auch in einer Nachricht empfangene Komponentenbindungen müssen fallengelassen werden. Dies wird erreicht, indem als erste Aktion einer Methode mit Komponentenparametern die Komponentenargumente in lokale Variablen bewegt werden, die dann wie normale lokale Variablen behandelt werden.

Während der Konstruktion einer Nachricht werden die Argumente nacheinander berechnet und auf dem Stapel abgelegt. Liefert die Berechnung eines späteren Arguments eine Ausnahme, so müssen bereits abgelegte Komponentenbindungen fallengelassen werden. Die Verantwortung für die Komponentenbindungen geht erst mit dem eigentlichen Aufruf an die aufgerufene Methode über.

5.5.4. Entkommende Variablen

Die Lebenszeit einer Variablen, die aus einem Funktionsabschluß zugegriffen wird, ist genau wie die des Funktionsabschlusses im allgemeinen unbegrenzt. Man sagt dann, die Variable entkommt (*escapes*). Die Variable kann zudem freie Variable mehrerer Funktionsausdrücke sein, und Änderungen sind dann für alle aus diesen Funktionsausdrücken gebildeten Funktionsabschlüsse möglich und sichtbar. Dies wird durch eine Indirektion erreicht: Die Funktionsabschlüsse erhalten eine Bindung an eine gemeinsame Speicherzelle, Exemplar der Klasse `Cell`, in deren einziger Exemplarvariable die Bindung gespeichert wird.

Handelt es sich bei der freien Variablen um eine Komponentenvariable, dann muß es sich bei dieser Exemplarvariable ebenfalls um eine Komponentenvariable handeln. Es wird also ein Exemplar der Klasse `AtCell` angelegt. Beim Zugriff auf eine Zelle gelten dieselben Regeln wie beim Zugriff auf andere Exemplarvariablen, auch wenn die Maschine aus Effizienzgründen andere Befehle benutzt.

Leider läßt sich eine in einer Zelle abgelegte Komponentenbindung weder dem Funktionsabschluß noch der Aktivierung zuordnen, in der die Zelle ursprünglich angelegt wurde, da von keiner Seite eine Komponentenbindung an die Zelle existiert. Es wäre zwar möglich, die Zelle zur Subkomponente der Aktivierung zu machen, und den Funktionsabschlüssen eine Referenzbindung an die Zelle zu geben. Überlebt jedoch ein Funktionsabschluß die Aktivierung, so ist die Zelle wieder ungebunden und damit nicht mehr zuordenbar, selbst wenn sie nur von einem einzigen Funktionsabschluß referenziert wird.

6. Anwendung in der Programmvisualisierung

Durch den vermehrten Einsatz von Softwarekomponenten und Frameworks und durch den hohen Grad der Wiederverwendung in objekt-orientierten Sprachen entsteht immer häufiger eine Situation, in der ein bereits existierendes Programm oder Programmfragment von anderen als dem ursprünglichen Autor eingesetzt oder verändert werden soll. Das erfordert ein Verständnis des Programms, sowohl in seiner Struktur als auch in seinem Verhalten. Programmvisualisierung unterstützt den Benutzer bei der Untersuchung des Programms, indem verschiedene aufbereitete Sichten auf Programmtext und ablaufendes Programm präsentiert werden.

Gerade die dynamische Ebene ist zum Verständnis objektorientierter Programme entscheidend, da in der objektorientierten Programmierung zum einen komplexe Verhaltensmuster auftreten und zum anderen die Gesamtfunktionalität erst durch die Kooperation generischer Bestandteile erbracht wird.

Programmvisualisierung steht vor dem Problem, die voluminösen Datenmengen, die bei der Beobachtung eines laufenden Programms anfallen, zu filtern, aufzubereiten und in eine für den Benutzer brauchbare Form zu bringen. In diesem Kapitel soll gezeigt werden, daß das Konzept der dynamischen Komponenten geeignet ist, durch Vergröberung von für den Benutzer irrelevanten Details sowohl im Systemzustand als auch in der beobachteten Interaktion zu einer angemessenen Darstellung zu gelangen. Das Kapitel schließt mit der Beschreibung des Prototyps eines Visualisierungswerkzeugs, das die dynamische Komponentenstruktur ausnutzt.

6.1. Programmvisualisierung

Eine eingehende Beschreibung des Themas Programmvisualisierung gibt [Hamer 97]. Hier soll nur ein Überblick gegeben werden.

Das Ziel der Programmvisualisierung besteht darin, dem Programmierer Werkzeuge zur Verfügung zu stellen, die ihn durch grafische Darstellung verschiedener Aspekte beim Verstehen eines Programms unterstützen. Bei dem untersuchten Programm kann es sich um eigenen oder fremden Code handeln, in dem Fehler oder Performanzschwächen erkannt und behoben werden sollen, oder auch um Bibliotheken und Frameworks, die verstanden werden müssen, bevor sie benutzt werden können. Das Werkzeug kann auch als

externes Gedächtnis dienen, das dem Programmierer erlaubt, Aspekte seines Programms festzuhalten und zwischenzeitlich zugunsten aktueller Probleme zu "vergessen".

Wichtig ist dabei, daß die Form der Darstellung dem aktuellen Problem entspricht, daß die gerade relevanten Aspekte also klar erkennbar sind. Die Maschine soll dem Programmierer so weit wie möglich entgegenkommen und so die erforderliche Verständnisarbeit minimieren. Die Sichtbarkeit eines Aspekts geht jedoch meist auf Kosten anderer Aspekte [Petre et al. 98]. Da verschiedene Probleme unterschiedliche Anforderungen stellen, wird daher ein Vorrat unterschiedlicher Darstellungen benötigt.

Die Visualisierung kann aus dem in Modellierungssprachen vorhandenen Diagrammvorrat schöpfen. Der erfolgreiche Einsatz der Diagramme bei der vorwärtsgerichteten Softwareentwicklung belegt ihre Eignung für die Beschreibung der verschiedenen Programmaspekte.

In der vorwärtsgerichteten Softwareentwicklung entsteht das Programm aus den beschreibenden Dokumenten. Die Programmvisualisierung wird auf dem umgekehrten Weg eingesetzt. Aus einem bereits erstellten Programm wird Dokumentation generiert, etwa weil die vorhandene Dokumentation ungenügend ist oder spezielle Fragen offen läßt.

Durch die automatische Extraktion ist im Gegensatz zu getrennt verfaßter Dokumentation sichergestellt, daß tatsächlich das implementierte System beschrieben wird. Wird dieselbe Notation wie für den Entwurf verwendet, so kann direkt zwischen geplantem und implementiertem System verglichen und die Implementierung so validiert werden. Das setzt jedoch voraus, daß die vom Visualisierungswerkzeug benötigte Information im Programm verfügbar oder daraus ableitbar ist.

Programmvisualisierung wird auch zur Bestimmung quantitativer Eigenschaften des Systemverhaltens eingesetzt; diese haben häufig keine Entsprechung auf der Entwurfsebene. Quantitative Eigenschaften werden zur Überwachung der Systemperformanz (*monitoring*) eingesetzt, und sie liefern Information darüber, wie eng verschiedene Systemkomponenten zusammenarbeiten.

6.2. Reduktion der präsentierten Datenmenge

Die Untersuchung eines Programms geschieht in Hinsicht auf eine Hypothese oder Fragestellung. Zur Unterstützung muß ein Visualisierungswerkzeug Mechanismen zur Abstraktion und Auswahl zur Verfügung stellen, um den Benutzer nicht mit für sein Untersuchungsziel irrelevanten Details zu belasten.

In diesem Abschnitt soll ein Überblick über die in existierenden Visualisierungswerkzeugen eingesetzten Reduktionstechniken gegeben werden. Der Schwerpunkt liegt dabei auf der Visualisierung der dynamischen Ebene. Den beschriebenen Techniken wird die Vergröberung anhand dynamischer Komponenten gegenübergestellt.

6.2.1. Quantitative Zusammenfassung

Das Zählen gleichartiger Phänomene wie etwa in [DePauw et al. 94; Sefika et al. 96] bietet eine Möglichkeit der Abstraktion, die zum einen Aussagen über die Kopplung von Subsystemen gestattet und zum anderen die Aufmerksamkeit auf Phänomene lenkt, die zu selten oder zu häufig auftreten. Dadurch können u.a. Performanzprobleme identifiziert werden. Die quantitative Bewertung kann grafisch als Abstand dargestellt und so direkt wahrgenommen werden.

Bei den gezählten Phänomenen kann es sich zum Beispiel um die Exemplare einer Klasse oder um die zwischen verschiedenen Subsystemen versendeten Nachrichten handeln.

Ein Nachteil der quantitativen Zusammenfassung besteht darin, daß auftretende Muster in dieser Form nicht erkennbar sind; man erhält Information über das "wie sehr" und "wie viel", aber nicht über das "wie".

6.2.2. Gruppierung dynamischer Phänomene anhand von statischer Information

Das in [Sefika et al. 96] beschriebene System erlaubt die Gruppierung von Objekten in Abhängigkeit von ihrer Klasse. So kann etwa die Kommunikation beliebiger Exemplare der Klasse A mit beliebigen Exemplaren der Klasse B gezählt werden. Allgemeiner kann auch die Kommunikation aller Exemplare von zu einer bestimmten Softwarekomponente gehörigen Klassen erfaßt werden.

In [Lange, Nakamura 95] wird die Verknüpfung mit statischen Aspekten des Programms in noch allgemeinerer Form vorgeführt. Im Unterschied zu den bisher betrachteten Werkzeugen entstehen als Ergebnis gefilterte Objekt- und Sequenzdiagramme und nicht quantitative Darstellungen.

So können zum Beispiel die in einem Sequenzdiagramm darzustellenden Interaktionen auf in bestimmten Klassen definierte Nachrichten eingeschränkt werden. Die Darstellung kann auch auf Exemplare bestimmter Klassen beschränkt werden, wobei die Klassen etwa aus einer bestimmten Softwarekomponente stammen.

Als Grundlage dient für [Lange, Nakamura 95] eine Prolog-Wissensbank, die sowohl Informationen der statischen als auch der dynamische Ebene enthält. Zur Spezifikation einer Anfrage steht die volle Mächtigkeit einer logischen Programmiersprache zur Verfügung. Die Autoren merken aber an, daß die Programmierung von Anfragen auch von der eigentlichen Untersuchung ablenken kann. Zum Beispiel kann auch das Anfrage-Programm Fehler enthalten und muß dann wiederum untersucht werden. Ihr System stellt daher vorgefertigte Anfragen und Filter und die Präsentation ihrer Ergebnisse durch eine grafische Oberfläche zur Verfügung.

Wie in Abschnitt 2.3.2 bei der Betrachtung von Lokalität und Interaktion festgestellt, ist die Unterteilung nach der statischen Zugehörigkeit im allgemeinen nicht geeignet, auf

der dynamischen Ebene zusammengehörige Objekte zu erkennen. Wird eine statische Softwarekomponente in verschiedenen dynamischen Zusammenhängen eingesetzt, so sind ihre komplexen Exemplare durch statische Kriterien nicht unterscheidbar.

Mit der Interaktion zwischen Komponenten soll hier im Gegensatz zu den zitierten Systemen daher immer die Interaktion dynamischer Komponenten gemeint sein.

6.2.3. Vergröberung der Aufrufhierarchie

In [Koskimies, Mössenböck 96] wird ein Werkzeug beschrieben, das unter anderem die automatische Generierung von Sequenzdiagrammen (*event trace diagram, scenario diagram*) erlaubt. Eine übersichtliche Darstellung wird dadurch erreicht, daß die während der Ausführung einer Methode stattfindenden weiteren Aufrufe zunächst weggefaltet werden. Auf Benutzerwunsch können sie ebenenweise ausgeklappt werden. Dadurch findet ein vom Benutzer gesteuerter Abstieg entlang der Aufrufhierarchie statt.

Diese Vorgehensweise entspricht einer klassischen prozeduralen Sichtweise. Einheit des Laufzeitverhaltens ist die Ausführung einer Prozedur, nicht aber kommunizierende Objekte. Der Ansatz ist daher auch nicht ohne weiteres auf nebenläufige oder reentrante Interaktion anwendbar, und führt zu Problemen bei der Darstellung von typisch objektorientierten Interaktionsformen wie z.B. des *Beobachter-Musters* [Gamma et al. 95].

6.2.4. Vergröberung Dynamischer Komponenten

Die Gruppierung von Objekten zu dynamischen Komponenten läßt sich auf zweierlei Arten für die Programmvisualisierung einsetzen: Zum einen für die Visualisierung des Systemzustandes und zum anderen für die Visualisierung von Interaktion.

Bei der Visualisierung des Systemzustandes kann die hierarchische Struktur der dynamischen Komponenten ausgenutzt werden. Subkomponenten eines Objektes werden vom Visualisierungswerkzeug zunächst nicht dargestellt. Erst dann, wenn der Benutzer sich für das Innere der dynamischen Komponente interessiert, werden auch die enthaltenen Objekte angezeigt. Wie in den Beispielen in Kapitel 2 können die Subkomponenten zu demselben Diagramm hinzugefügt werden. Dadurch gelangt man zu einer Darstellung des Systemzustandes auf wählbarer Granularitätsebene.

Im Hinblick auf Interaktion kann von Kommunikation innerhalb einer dynamischen Komponente abstrahiert werden. Dabei wird ausgenutzt, daß in der objektorientierten Programmierung jeder Nachricht und jedem Ergebnis ein sendendes und ein empfangendes Objekt zugeordnet werden kann.

Der Benutzer wählt die dynamischen Komponenten aus, deren Interaktion ihn interessiert, und kann so den zu beobachtenden Systemausschnitt und die Granularitätsebene der Beobachtung bestimmen. Kommunikation, die vollständig außerhalb des ausgewählten Bereichs abläuft, wird genauso weggelassen wie Kommunikation, die vollständig innerhalb einer ausgewählten dynamischen Komponente abläuft (Abbildung 6.1).

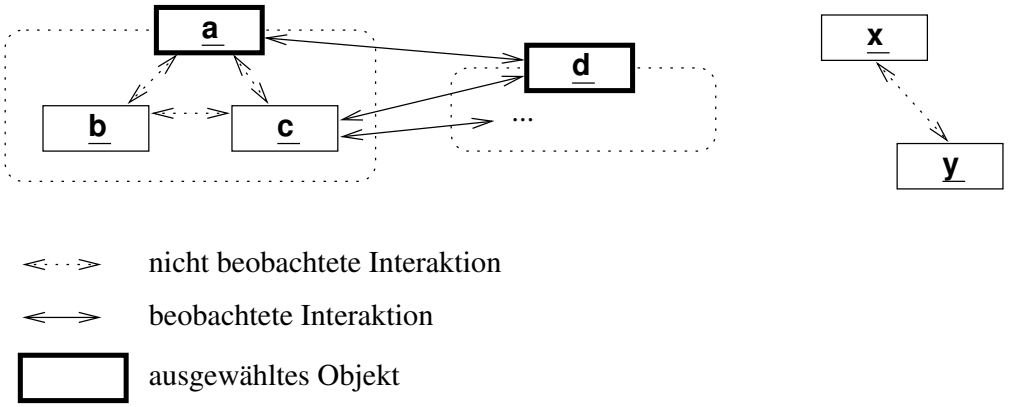

Abbildung 6.1.: Vergröberung der Interaktion

Prinzipiell kann jede Kommunikation einer dynamischen Komponente mit ihrer Umwelt einen Informationsaustausch mit einer anderen dynamischen Komponente bedeuten (etwa indirekt durch einen nicht beobachteten toten Briefkasten wie in Abbildung 6.2). In der Nachrichtenabfolge eindeutig zu erkennen ist Kommunikation aber nur dann, wenn ein Kontrollfluß von einer dynamischen Komponente in eine andere stattfindet: Eine Nachricht oder ein Ergebnis verläßt eine ausgewählte dynamische Komponente, und eine Nachricht oder ein Ergebnis desselben Threads betritt später eine andere ausgewählte dynamische Komponente. Als zusätzliche Filterung kann sich die Visualisierung auf diese Fälle beschränken (Abbildung 6.3).

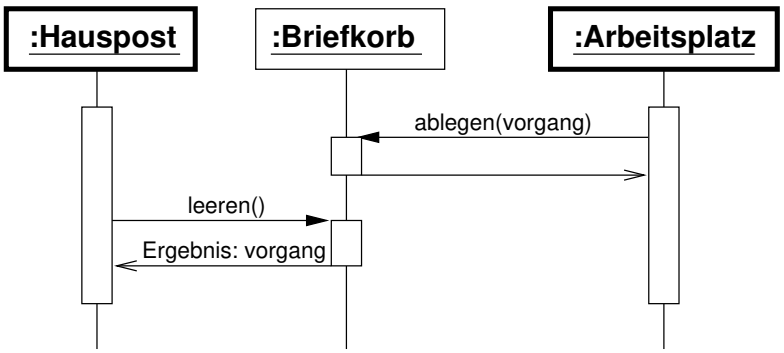

Abbildung 6.2.: Interaktion ohne Kontrollfluß: ein toter Briefkasten

Liegt eine ausgewählte dynamische Komponente innerhalb einer anderen ausgewählten, so erhält der Benutzer die Möglichkeit, Interaktion nicht nur horizontal zwischen Komponenten eines Systems, sondern auch vertikal zwischen Komponente und Subkomponente zu untersuchen. Diese Möglichkeit ist besonders wichtig, wenn eine ausgewählte dynamische Komponente in eine andere hineinmigriert (Abbildung 6.4).

Zur Auswahl der zu beobachtenden dynamischen Komponenten ist jede Anfrage denk-

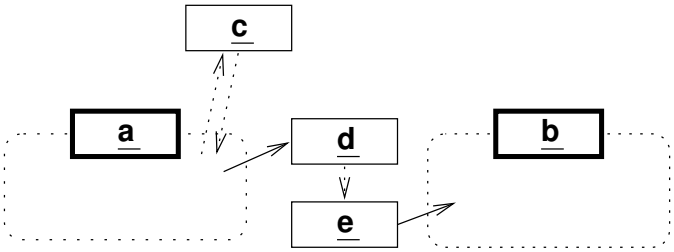

Abbildung 6.3.: Kontrollfluß durch nicht ausgewählte Objekte

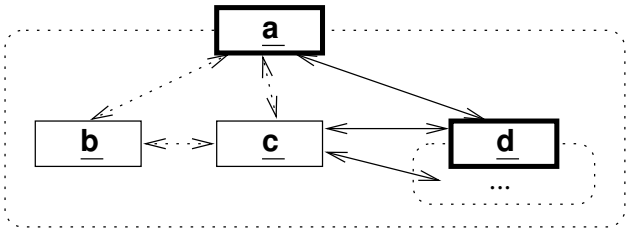

Abbildung 6.4.: Verschachtelte ausgewählte Objekte

bar, die ein Objekt oder auch eine Menge von Objekten liefert. Dabei kann auch eine Verknüpfung mit Information der statischen Ebene sinnvoll sein.

6.3. Visualisierung unter Verwendung Dynamischer Komponenten

In diesem Abschnitt wird ein im Tycoon-2-System prototypisch implementiertes Visualisierungswerkzeug beschrieben, das die reflektiv vom System bereitgestellte Information über dynamische Komponenten für die Darstellung von Objekten und Interaktion ausnutzt. Die Interaktion wird in Form von Sequenzdiagrammen dargestellt. Wesentliches Ziel des Prototypen ist es, dynamische Komponenten sichtbar zu machen.

In einem praktisch einsetzbaren Visualisierungswerkzeug müßten zur Darstellung der dynamischen Ebene auch damit verknüpfte Sichtweisen der statischen hinzukommen. Diese umzusetzen gehört jedoch nicht zum Thema dieser Arbeit. Darstellungen der statischen Ebene sind in dem implementierten Prototypen daher nur ansatzweise vorhanden.

6.3.1. Technische Randbedingungen

Das Tycoon-2-System verfügt über keine eigene grafische Benutzeroberfläche. Zwei Wege stehen offen: Entweder setzt man den im Tycoon-2-System vorfügbaren HTTP-*web-server* und dynamisch erzeugtes HTML ein (STML, [Matthes, Wienberg 97]) und überläßt die

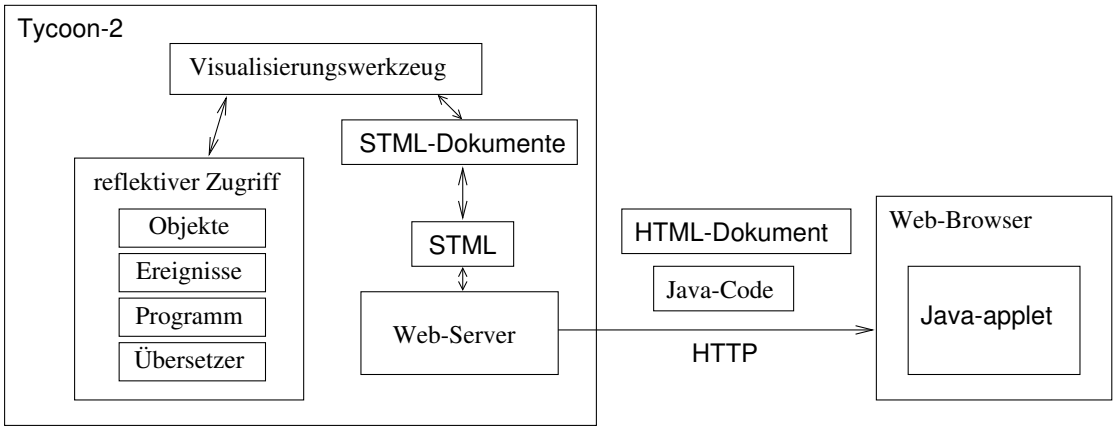

Abbildung 6.5.: Einbettung des Visualisierungswerkzeugs

eigentliche Darstellung einem *web browser*, oder man nimmt den aufwendigeren Weg und koppelt eine Java-Applikation über das Netzwerk direkt an Tycoon-2 [Schneider 98].

Da die Implementierung einer Benutzerschnittstelle nicht zentrales Thema dieser Arbeit ist, wurde der einfachere Weg gewählt: Der Prototyp des Visualisierungswerkzeugs ist mit Hilfe von STML implementiert und produziert HTML-Dokumente. Wo die Möglichkeiten von HTML nicht ausreichen, werden parametrisierte Java-*applets* zur Präsentation der Inhalte eingesetzt. Die Dokumente werden dynamisch generiert [Matthes, Wienberg 97] und spiegeln bei jedem Zugriff den Zustand des laufenden Systems wider.

Aufgrund der Einschränkungen durch die HTTP-Technologie können die Dokumente aber nicht vom Tycoon-2-System aus aktualisiert werden. Eine Animation des aktuellen Zustandes (etwa eine Fortschreibung der Sequenzdiagramme) ist daher in diesem Rahmen genausowenig möglich, wie die nachträgliche Ergänzung eines Dokuments um neu erfragte Informationen (etwa der schrittweise Aufbau eines Objektdiagramms). Auch ist HTML keine Grundlage für eine grafische Benutzeroberfläche. Eine praktisch einsetzbare Lösung sollte die grafische Oberfläche daher als eigenständige Java-Applikation implementieren.

Die im implementierten System an der Visualisierung beteiligten Komponenten sind in Abbildung 6.5 dargestellt.

6.3.2. Visualisierung des Objektspeichers

Wie in Abschnitt 4.6 beschrieben, enthält jedes Objekt im Tycoon-2-System eine Bindung für jede Exemplarvariable seiner Klasse. Um den Zustand eines Objekts darzustellen, reicht es daher, den Inhalt jeder Bindung darzustellen.

Die Darstellung von Objekten in Form der in dieser Arbeit bisher verwendeten Objektdiagramme wäre wünschenswert, um die automatisch generierten Diagramme mit der

Entwurfsnotation vergleichbar zu machen. Diese Darstellung ist jedoch technisch sehr aufwendig, besonders unter den gegebenen Randbedingungen.

Stattdessen werden die Subkomponenten eines betrachteten Objekts in einer zeilen- und einrückungsorientierten Baumform mit wahlweise versteckten Unterbäumen angezeigt. Das verwendete grafische Element ist zur Zeit unter dem Namen *Explorer-Hierarchie* bekannt (Abbildung 6.6).

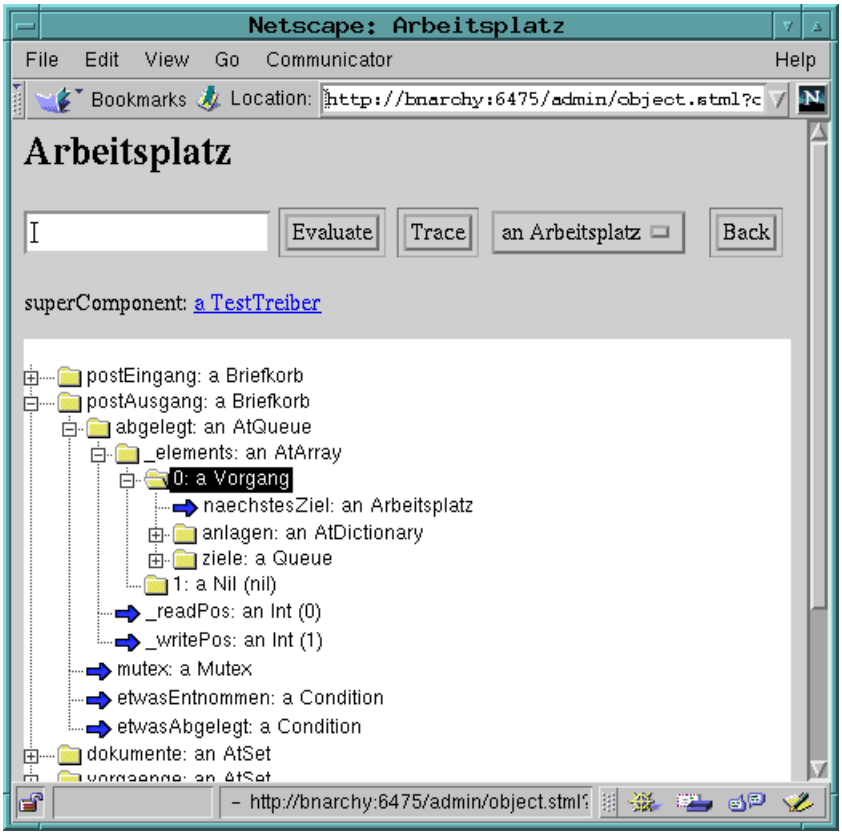

Abbildung 6.6.: Darstellung eines Objekts mit Subkomponenten

Referenzbindungen werden vom Prototypen als anwählbarer Hypertext dargestellt. Wird einer Referenzbindung gefolgt, so wird die Darstellung des aktuellen Objekts durch die des Zieles der Referenz ersetzt.

Eine ähnliche, baumförmige Darstellung von Objektgraphen findet sich auch in vielen kommerziellen Entwicklungsumgebungen (etwa Metrowerks, VisualAge, Visual Basic). Dort kann allerdings nicht zwischen Referenzbindung und Komponentenbindung unterschieden werden, was dazu führt, daß ein Objekt an mehreren Stellen des Baumes expandiert werden kann. Die Objektidentität geht in der Darstellung also verloren. Bei der Visualisierung einer zyklischen Datenstrukturen kann durch schrittweise Verfeinerung ein beliebig tiefer Baum entstehen, obwohl u.U. nur eine sehr kleine Zahl von Objekten beteiligt ist.

Da auch Subkomponenten eine eigene Objektidentität haben, und so über Referenzbindungen unabhängig von ihrer Superkomponente erreicht werden können, bietet der Prototyp eine Hypertext-Verbindung zur Superkomponente (wenn vorhanden). Dies erleichtert die Orientierung im Objektspeicher.

Neben der schrittweisen Navigation kann ein Objekt auch durch eine Anfrage erreicht werden. Im Kontext eines Objekts kann ein beliebiger Tycoon-2-Ausdruck eingegeben und ausgewertet werden, dessen Ergebnis (oder Ausnahme) dann wieder visualisiert wird.

Diese Möglichkeit wird durch den als Objekt im System vorhandenen Übersetzer erschlossen. Programmtext kann so übersetzt und dann reflektiv ausgeführt werden.

Die Darstellung der Objekte erfolgt ebenfalls reflektiv. Vom Klassenobjekt erfährt das Visualisierungswerkzeug Anordnung, Namen und Sorten der Exemplarvariablen. Mit Hilfe der Methode `__basicAt` kann dann positional der aktuelle Inhalt der Bindungen des Objekts erfragt werden.

6.3.3. Interaktion zwischen dynamischen Komponenten

Interaktion zwischen Tycoon-2-Objekten nimmt, wie in Abschnitt 4.6 beschrieben, drei Formen an: Nachricht, Ergebnis und Ausnahme. Eine Nachricht geht von einer Aktivierung an ein Objekt, das Ergebnis geht von der erzeugten Aktivierung zurück zu der aufrufenden Aktivierung. Eine Ausnahme geht ebenfalls von einer Aktivierung aus, bricht aber alle aufrufenden Aktivierungen ab, bis sie abgefangen wird. Bis auf ausgelöste Ausnahmen entsprechen diese Interaktionen genau den Elementen eines UML-Sequenzdiagramms.

Die Auswahl, welche dynamischen Komponenten den Lebenslinien des Sequenzdiagramms zuzuordnen sind, wird in zwei Schritten getroffen. Zunächst werden alle zu beobachtenden dynamischen Komponenten programmiersprachlich durch Aufruf der Methode `isTracedComponent` markiert. Von den ausgewählten dynamischen Komponenten erhalten jedoch nur diejenigen eine Lebenslinie, die während der beobachteten Ausführung als Quelle oder Ziel einer Interaktion auftreten.

Als zu beobachtende Ausführung wird ein beliebiger Tycoon-2-Ausdruck ausgewertet. Die im Verlauf der Ausführung auftretenden Interaktionen werden beobachtet, anhand der ausgewählten dynamischen Komponenten gefiltert und gegebenenfalls aufgezeichnet. Die Filterung verwendet den in Abschnitt 6.2.4 beschriebenen Algorithmus. Die aufgezeichneten Interaktionen werden schließlich dem Benutzer als Sequenzdiagramm präsentiert. Parallel zur grafischen Darstellung werden als Hypertext das genaue Sender- und Empfänger-Objekt innerhalb der jeweiligen dynamischen Komponente sowie die in der Nachricht übertragenen Argumente ausgegeben (Abbildung 6.7).

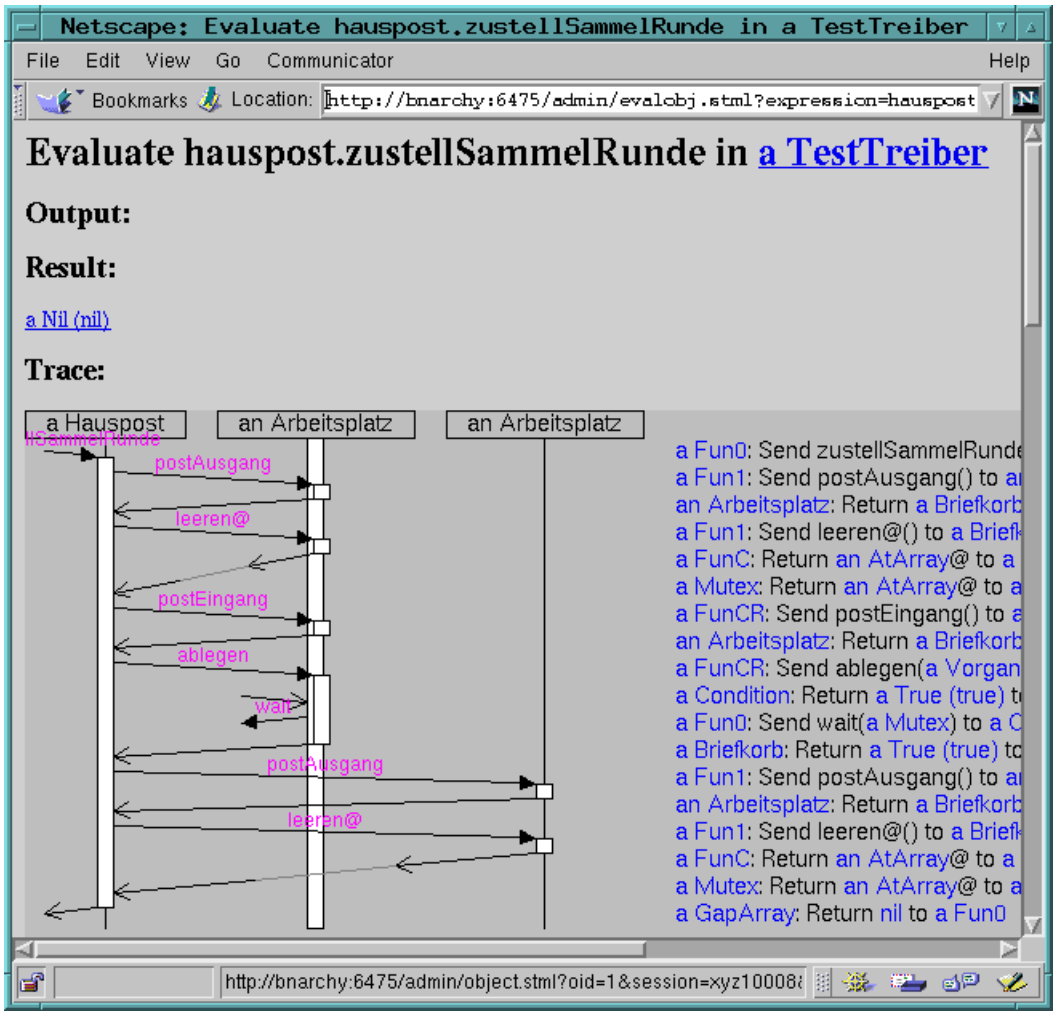

Abbildung 6.7.: Ein automatisch generiertes Sequenzdiagramm

6.3.4. Zugriff auf Interaktionsinformation

Um die Interaktions-Information verarbeiten zu können, muß sie zunächst einmal beobachtet werden. Dazu gibt es im wesentlichen zwei Möglichkeiten: Instrumentierung und Verhaltensreflektion. Instrumentierung verändert den Programmtext, während Verhaltensreflektion das dynamische Verhalten verändert.

Den Begriff Instrumentierung kann man daraus erklären, daß im Programmtext Meßinstrumente angebracht werden. Wann immer die Kontrolle ein solches Meßinstrument erreicht, wird ein Ereignis gemeldet. Typische Positionen für Meßinstrumente sind am Anfang und Ende einer Methode sowie vor und hinter einem Aufruf.

Ein Vorteil der Instrumentierung besteht darin, daß sie prinzipiell portabel ist – als Programmtransformation verläßt sie sich nur auf die Semantik der Programmiersprache. Ein

weiterer Vorteil ist, daß auch instrumentierter Programmtext noch übersetzt und relativ effizient ausgeführt werden kann. Als Nachteil sind die gesteigerte Programmgröße (auch in übersetzter Form) und gesteigerte Übersetzungszeit zu nennen. Instrumentierung kann automatisch durchgeführt werden, so daß für den Programmierer im allgemeinen kein Mehraufwand entsteht.

Der für die Visualisierung dynamischer Komponenten entscheidende Nachteil ist jedoch, daß Instrumentierung sich an der statischen Programmstruktur orientiert. Ereignisse werden nur aus den Programmteilen gemeldet, die instrumentiert worden sind. Zur Überwachung der Grenze einer dynamischen Komponente müssen aber alle Nachrichten sichtbar sein, die die dynamische Komponente verlassen oder betreten, egal in welcher Klasse die aufgerufene Methode implementiert ist.

Den gesamte Programmtext inklusive Standardbibliothek zu instrumentieren ist im Tycoon-2-System aufgrund seiner reflektiven Struktur nicht möglich. Das Visualisierungswerkzeug läuft in demselben System wie sein Untersuchungsobjekt und teilt dessen Standardbibliothek. Es ist sogar vorstellbar, daß das Visualisierungswerkzeug das Verhalten eines anderen Exemplars seiner selbst visualisiert.

Als geeigneter erweist sich die traditionell in Testhilfen (*debugger*) eingesetzte Verhaltensreflektion. Ein Prozeß wird unter Kontrolle eines anderen Prozesses ausgeführt. Wann immer in dem kontrollierten Prozeß ein interessantes Ereignis stattfindet, wird der kontrollierte Prozeß angehalten und der kontrollierende Prozeß informiert. Dieser erhält nun Gelegenheit, den Zustand des kontrollierten Prozesses zu untersuchen und evtl. auch zu verändern, bevor er die Ausführung fortsetzen läßt. Was dabei unter einem interessanten Ereignis zu verstehen ist, kann dem Reflektionsmechanismus als Parameter mitgeteilt werden. Da die aufgetretenen Ereignisse im kontrollierenden Prozeß noch weiter gefiltert werden können, ist die Spezifikation der interessanten Ereignisse hauptsächlich als Mittel der Effizienzsteigerung zu verstehen.

Das Visualisierungswerkzeug verwendet daher eine auf der Ebene der virtuellen Maschine implementierte Verhaltensreflektion. Die Implementierung der Verhaltensreflektion ist auf die Persistenz des Tycoon-2-Systems hin ausgelegt, so daß auch während einer laufenden Programmuntersuchung ein persistenter Sicherungspunkt angelegt werden kann.[1]

Die Reflektion ist nicht vollständig. Es können zwar Nachrichten, Ergebnisse und Ausnahmen erkannt werden und auch das aktive Objekt eines Prozesses ist feststellbar. Aktivierungen sind jedoch durch die Reflektionsschnittstelle nicht identifizierbar, da sie, wie in Abschnitt 5.4.1 erwähnt, nicht als eigenständige Objekte im Objektspeicher vorliegen. Für eine vollständige Darstellung im Sequenzdiagramm muß die Information über vorhandene Aktivierungen daher anhand der Interaktionsfolge rekonstruiert werden.

[1] Dies erwies sich als relativ verzwickt und führte zur Aufdeckung kleiner Fehler im Persistenzmechanismus.

7. Schluß

Zum Abschluß der Arbeit sollen die Ergebnisse zusammengefaßt, eine Bewertung des Erreichten vorgenommen sowie ein Ausblick auf offen gebliebene und sich neu ergebende Fragen gegeben werden.

7.1. Ergebnisse

In dieser Arbeit wurde das Konzept der dynamischen Komponenten in Begriffen der objektorientierten Modellierung entwickelt und als graphische Notation in der Modellierung sowie als Erweiterung der Programmiersprache Tycoon-2 umgesetzt. Die erweiterte Programmiersprache wurde im Tycoon-2-System implementiert und die sich daraus ergebende Strukturierung des Objektspeichers für beliebige Anwendungen reflektiv zur Verfügung gestellt. Dadurch wurde die Machbarkeit des Konzepts erwiesen.

Aufbauend darauf wurde der Prototyp eines Visualisierungswerkzeugs geschaffen, das die Komponentenstruktur ausnutzt, um den Systemzustand und die stattfindende Interaktion auf wählbarer Granularitätsebene darzustellen.

Bei der Untersuchung der in Programmiersprachen vorhandenen Bindungsarten fiel auf, daß hierarchische Objektstrukturen dort hauptsächlich Effizienzgründen dienen und daher mit Einschränkungen verknüpft sind, die nur mit Kenntnis der Maschinenrepräsentation verständlich werden.

Im Gegensatz dazu ist die Komponentenbindung durch die Konzentration auf den wesentlichen Aspekt in der Lage, ein sehr breites Spektrum von Aggregation auszudrücken. Die Komponentenbindung wurde als spezielle Art der Aggregation charakterisiert, die zwischen den Objekten zur Laufzeit jederzeit eine hierarchische Struktur bildet.

In der programmiersprachlichen Umsetzung wurden Variablen zur Unterscheidung zwischen Referenzbindung und Komponentenbindung statisch in zwei Sorten unterteilt, die jeweils verschiedene Operationen zulassen. Als Operation, die Komponentenbindungen ändert, dabei aber die hierarchische Struktur aufrechterhält, wurde das destruktive Lesen ausgewählt. Destruktives Lesen ersetzt die traditionelle Kopiersemantik der Zuweisung durch Bewegung des zugewiesenen Objekts und erhält so die Objektidentität.

Durch das Visualisierungswerkzeug wurden die entstehenden dynamischen Komponenten sichtbar gemacht, wodurch die Implementierung des Konzepts überprüft werden konnte.

7.2. Bewertung

Mit der Komponentenbindung steht erstmals eine Art der Aggregation durchgängig von der Modellierung bis zur programmiersprachlichen Umsetzung zur Verfügung. Obwohl sich die Komponentenbindung an der Konzeptebene und nicht an der Maschinenebene orientiert, läßt sie sich mit vertretbarem Laufzeit- und Speicheraufwand implementieren. Es wurde eine funktionierende Implementierung erreicht, auf deren Grundlage weitere Untersuchungen möglich werden. Durch die überblicksartige Visualisierung wurde eine erste neu erschlossene Möglichkeit demonstriert.

Das durch dynamische Komponenten unterstützte Prinzip der hierarchischen Strukturierung tritt in der Modellierung sehr natürlich auf, weshalb dynamische Komponenten in der Programmierung vielfach eingesetzt werden können. Die änderbare, hierarchische Strukturierung des Systemzustandes eröffnet gegenüber einer reinen Referenzsemantik vielversprechende Möglichkeiten:

Granularitätsebenen Die hierarchische Struktur kann verwendet werden, um von ganzen Unterbäumen und den darin stattfindenden Ereignissen zu abstrahieren. Diese Möglichkeit wird vom Visualisierungswerkzeug ausgenutzt.

Lokalität Die hierarchische Struktur gibt ein Maß für den Abstand zwischen zwei Objekten. Der Programmierer kann sie bewußt einsetzen, um Lokalität auszudrücken, was in einem verteilten System durch verbesserte Lastabschätzung zu verringertem Kommunikationsbedarf zwischen verschiedenen Rechnerknoten führt. Im Hauptspeicher eines Rechners wird Zugriffslokalität durch die verschiedenen *caches* belohnt: zwischen Prozessor und Hauptspeicher, zwischen Hauptspeicher und Festplatte etc.

Tiefe Kopie und Serialisierung Häufig ist es Aufgabe des Programmierers, durch einen Algorithmus auszudrücken, welche Objekte gemeinsam zu serialisieren oder zu kopieren sind. Stattdessen kann die deklarativ definierte dynamischen Komponente als Objektgruppe benutzt werden. Die Methode *copy@* der Standardbibliothek verwendet diese Möglichkeit bereits.

Migration Als Kombination von Lokalität und Serialisierung ist Migration eine natürliche Anwendung. Die Objekte einer dynamischen Komponente befinden sich alle auf demselben Rechner. Eine Subkomponente kann durch Änderung einer Komponentenbindung abgespalten und an eine andere dynamische Komponente übergeben werden, die sich potentiell auf einem anderen Rechner befindet.

Überprüfung von Modelleigenschaften Da der Programmierer das System durch die Auszeichnung einer Komponentenbindung über die intendierten Eigenschaften dieser Bindung informiert, kann das System die Eigenschaften überprüfen. Dies geschieht soweit wie möglich statisch am Programmtext und optional auch als Überwachung zur Laufzeit.

Die noch nicht umgesetzten Punkte (Lokalität, Serialisierung und Migration) wären auf der Grundlage des vorhandenen Systems zu untersuchen, um das volle Potential der dynamischen Komponenten auszuloten.

Der durch dynamische Komponenten entstehende Mehraufwand läßt sich an die beabsichtigte Anwendung anpassen. Drei Grade lassen sich unterscheiden:

1. Der geringste Zusatzaufwand entsteht, wenn Komponentenbindungen nur statisch ausgezeichnet und überprüft werden. Komponentenbindungen sind dadurch von Referenzbindungen unterscheidbar, und zur Laufzeit können die Subkomponenten eines gegebenen Objekts bestimmt werden. Statisch ist sichergestellt, daß jedes Objekt höchstens eine Superkomponente hat. Diese Möglichkeiten sind ausreichend, um tiefe Kopie, Serialisierung und Migration umsetzen zu können. Mehraufwand entsteht nur durch die Operation des destruktiven Lesens, die zusätzlich zum einfachen Lesezugriff einen Schreibzugriff erfordert.

2. Das System kann die Superkomponente jedes Objekts speichern und jederzeit auf dem neusten Stand halten. Dies erlaubt die vergröberte Visualisierung von Objektinteraktion. Außerdem kann bei der Rekonstruktion einer Komponentenbindung aus einer Referenzbindung sichergestellt werden, daß das referenzierte Objekt nicht bereits in einer Komponentenbindung steht; diese Zusicherung des Programmierers kann also zur Laufzeit geprüft werden. Mehraufwand entsteht an folgenden Stellen:

   (a) Für jedes Objekt muß ein Verweis auf die Superkomponente gespeichert werden. Im implementierten System verursacht dies einen Zuwachs des Speicherbedarfs von zehn Prozent.

   (b) Der Verweis muß nach jeder Speicherbereinigung daraufhin überprüft werden, ob die Superkomponente freigegeben wurde. Der entstehende Laufzeitaufwand ist pro Speicherbereinigung proportional zur Anzahl überlebender Objekte.

   (c) Bei jeder Zuweisung an eine Komponentenvariable muß geprüft werden, ob sie bereits ein Objekt enthält, das ggf. ungebunden wird. Der Mehraufwand beträgt pro Zuweisung einen Lesezugriff, einen Test auf *nil* und ggf. einen Schreibzugriff.

   (d) Bei Zuweisung und destruktivem Lesen auf Komponenten-Exemplarvariablen muß ggf. der Superkomponenten-Verweis des bewegten Objekts aktualisiert werden. Der Mehraufwand beträgt pro Zugriff einen Test auf *nil* und ggf. einen Schreibzugriff.

   (e) Bei Beendigung einer Aktivierung durch Ergebnis oder Ausnahme muß der Inhalt lokaler Komponentenvariablen ungebunden werden. Der Mehraufwand ist schwer zu quantifizieren, läßt sich aber durch einen optimierenden Übersetzer vermutlich sehr gering halten.

   Der Laufzeitaufwand ist also durch einen konstanten Faktor begrenzt. Da die genannten Einsatzmöglichkeiten der Superkomponenten-Information in der Programmentwicklung liegen, ist der Aufwand vertretbar. Bis auf den gewachsenen

Speicherbedarf und die geringfügig verlangsamte Speicherbereinigung tritt Zusatzaufwand insbesondere nur an den Stellen auf, an denen Komponentenbindungen tatsächlich verwendet werden. Programmteile, die nur mit Referenzbindung arbeiten, werden so kaum beeinflußt.

3. Durch Laufzeittests kann sichergestellt werden, daß keine zyklischen Objektstrukturen entstehen. Hier kann also eine weitere Zusicherung des Programmierers zur Laufzeit überwacht werden. Dazu muß bei jeder Zuweisung an eine Komponenten-Exemplarvariable die Kette der Superkomponenten-Verweise des Zielobjekts bis zur Wurzel verfolgt werden. Der Aufwand ist proportional zur Schachtelungstiefe des Zielobjekts und somit statisch i.A. nicht begrenzt, auch wenn er durch geeignete Programmierung niedrig gehalten werden kann. Diese Möglichkeit kann daher unabhängig von der Verwaltung der Superkomponenteninformation deaktiviert werden.

Eine quantitative Bewertung des Laufzeitmehraufwandes steht noch aus. Aussagekräftige Werte lassen sich erst dann bestimmen, wenn eine realistische Anwendung unter Verwendung dynamischer Komponenten implementiert ist. Dies geht jedoch über den Umfang der Diplomarbeit hinaus.

7.3. Ausblick

Die vom Visualisierungswerkzeug generierten, unter Verwendung dynamischer Komponenten gefilterten Sequenzdiagramme stellen einen neuen, erfolgversprechenden Ansatz dar, das Verhalten eines komplexen objektorientierten Systems übersichtlich darzustellen. Durch eine im Prototypen nicht implementierte Verknüpfung mit statischen Programmaspekten ließen sich noch problemadäquatere Filterungen erreichen.

Die Nützlichkeit des Visualisierungswerkzeugs läßt sich ohne eine vollständige Visualisierungsumgebung und ohne den Einsatz in einem realen Projekt nicht abschließend beurteilen; das Visualisierungswerkzeug wäre dafür zu vervollständigen, bzw. in anderer Technologie (echtem Java statt HTML) neu zu entwickeln. Da die Grundlagen der Spracherweiterung unabhängig von der Wirtssprache sind, wäre es auch vorstellbar, vollständig auf Java umzusteigen.

In der ersten Version meldete der Prototyp des Visualisierungswerkzeugs jede Kommunikation, die die Grenze einer ausgewählten dynamischen Komponente kreuzte. Ziel war es, auch indirekte Kommunikation verschiedener ausgewählter Komponenten erkennbar zu machen, weshalb jede Kommunikation mit der Umwelt dargestellt werden mußte. Dadurch wurde jedoch auch jede Kommunikation mit Basiswerten wie z.B. Ganzzahl-Objekten (`Int`) angezeigt, da dem Visualisierungswerkzeug nicht bekannt war, daß diese Objekte keinen änderbaren Zustand besitzen und daher nicht als toter Briefkasten in Frage kommen.

Die in der Arbeit beschriebene Version umgeht das Problem, indem sie die Anforderungen abschwächt und nur Kontrollfluß, aber nicht jeden Datenfluß zwischen ausgewählten dynamischen Komponenten darstellt. Eine fundamentalere Lösung hätte in der Einführung von Wertobjekten à la Emerald [Hutchinson 87] bestanden, die über keine Objektidentität verfügen und damit auch keiner bestimmten Lokalität zuzuordnen sind. Kommunikation mit einem Wertobjekt wird dann nicht als Kommunikation mit der Außenwelt gewertet. Wertobjekte sind sowohl in der Visualisierung als auch in der Migration nützlich.

Eine Vergröberung von Interaktion findet durch den in dieser Arbeit beschriebenen Ansatz nicht statt, Nachrichten werden gefiltert, nicht zusammengefaßt. Auf der statischen Ebene erlaubt die UML-Notation die Beschreibung von Verhalten in hierarchisch verfeinerten endlichen Automaten; dies ließe sich eventuell auch auf die dynamische Ebene übertragen.

Neben Objekten und Interaktion sind auch Bindungen ein Phänomen der dynamischen Ebene. [Kristensen 94] schlägt vor, Assoziationen und dadurch auch Bindungen zu Gruppen zusammenzufassen. Dieser Ansatz ist auch eine Konsequenz des im relationalen Datenmodell vertretenen *semantischen Relativismus* [Brodie 84]: Im relationalen Datenmodell kann der Beobachter denselben Sachverhalt sowohl als Relation als auch als Objekt deuten. Dadurch wird die Gruppierung von Objekten aus einem anderen Blickwinkel zur Gruppierung von Bindungen.

Bei der Visualisierung fiel auf, daß die Zuordnung von Funktionsabschlüssen zu dynamischen Komponenten unklar ist. Einerseits ist ein Funktionsabschluß ein Objekt mit Objektidentität, an das auch Komponentenbindungen existieren können; andererseits besitzt es durch die Identität des bei der Erzeugung umgebenden Objekts schon eine ausgezeichnete Beziehung. In der Darstellung wurden die besten Ergebnisse erreicht, wenn ein Funktionsabschluß als Subkomponente des umgebenden Objekts betrachtet wurde. Diese Deutung entspricht aber nicht der Anforderung, daß eine Subkomponente von der Superkomponente aus erreichbar sein muß. Es wäre auch vorstellbar, Funktionsabschlüsse generell als Wertobjekte zu behandeln – die Rolle der Funktionsabschlüsse bleibt zu klären.

Funktionsabschlüsse führten auch im Zusammenhang mit Komponentenparametern zu Implementierungsproblemen, da das Tycoon-2-System für jeden möglichen Funktions-Selektor eine eigene, fest definierte Klasse benötigt. Statt die Menge der Funktions-Selektoren statisch festzulegen, wäre es eventuell geschickter, Funktionsausdrücke als Variante der Klassendefinition anzusehen. Ein Beispiel dafür geben die *inner classes* von Java [Gosling et al. 96].

Weiter wäre zu untersuchen, ob es möglich ist, generische Behälterklassen mit der Sorte (Komponente oder Referenz) der gespeicherten Bindungen zu parametrisieren, ähnlich wie es mit dem Typ der gespeicherten Bindungen schon geschieht. Die zu erwartenden Vorteile entsprechen denen der Einführung von Typparametern: Code muß nur einmal geschrieben werden und kann einmalig für alle möglichen Parametrisierungen überprüft

werden. Um dies zu ermöglichen, müßte eine gemeinsame operationale Grundlage für Referenz- und Komponentenbindungen gefunden werden.

Eine Alternative bestände darin, die Elemente eines Behälters immer als Subkomponenten zu behandeln; wird eine Referenz benötigt, so muß diese durch ein eigenes Objekt der Klasse `Reference(T)` dargestellt werden. Da eine Referenz eigentlich keine Objektidentität besitzt, könnte dieser Ansatz mit dem Konzept der Wertobjekte verknüpft werden. Die entstehende Lösung wäre den traditionellen Zeiger-Typen allerdings sehr ähnlich.

Die Parametrisierung der Bindungssorte könnte sich an [Noble et al. 98] orientieren. Der dortige Ansatz unterscheidet zwischen den Elementen eines Behälters und den zur Verwaltung der Elemente benötigten Datenstrukturen. Beide Beziehungen werden in dem in dieser Arbeit beschriebenen System jedoch uniform durch Komponentenbindung ausgedrückt. Hier geht also eine Unterscheidung verloren, was in der Visualisierung dazu führt, daß der Benutzer mit Implementierungsdetails der Behälterobjekte konfrontiert wird. Es wäre zu untersuchen, ob sich dieses Manko durch Parametrisierung beheben läßt.

Insgesamt bieten die programmiersprachliche Umsetzung von Aggregation und das Konzept der dynamischen Komponenten eine fruchtbare Grundlage für weitere Untersuchungen.

Danksagung

Die Arbeit wäre nicht möglich gewesen ohne die Diskussionen mit Florian Matthes, die stets so lange dauerten, bis im kreativen Chaos wieder eine Struktur sichtbar wurde. Auch meinem Bruder Frank Wienberg bin ich für viele hilfreiche Kommentare, Anregung und Ermutigung zu Dank verpflichtet.

Erst durch die konzeptuelle Klarheit des Tycoon-Systems wurde es möglich, über die Umsetzung neuer Konzepte nachzudenken. Andreas Gawecki war ein Vorbild dafür, wie man eine Idee konsequent umsetzt. Auch den anderen Kollegen bei Higher-Order möchte ich für fruchtbare Diskussionen danken, allen voran Marc Weikard, Matthias Ernst und Daniel Schneider.

Marko Boger rezensierte ein Exposé und verwies mich auf die Programmiersprachen Eiffel und Emerald. Den Bezug zum *occur check* in Prolog stellte Olaf Kummer her.

Birgit Slomski ertrug die Geisteshaltung, in die mich stundenlanges Feilen an Implementierung und Text brachte, und stand mir in der Diplomarbeit mit stilistischen und syntaktischen Tips zur Seite.

Mein letztes (aber nicht geringstes) Dankeschön geht an Gerald Schröder und Frank Schimmel, die jeweils 100 Seiten Diplomarbeit Korrektur lasen, und auch an meine Gutachter, die diese Aufgabe noch vor sich haben.

A. Erweiterte Grammatik von Tycoon-2

In diesem Abschnitt wird die vollständige Grammatik der um dynamische Komponenten erweiterten Programmiersprache Tycoon-2 wiedergegeben. Die Erweiterung basiert auf der Fassung 1.0 des Tycoon-2-Sprachreports [Gawecki, Wienberg 98].

Zur Definition der Meta-Syntax wird folgende Notation benutzt: *Id* bezeichnet ein nicht-terminales Symbol, *A* und *B* bezeichnen syntaktische Ausdrücke.

Id ::= *A*	das nicht-terminalen Symbol *Id* ist definiert als *A*
Id	ein nicht-terminales Symbol
x	das terminale Symbol mit Zeichenfolge *x* (Schlüsselwort)
"x"	das terminale Symbol mit Zeichenfolge *x*
(A)	bedeutet *A*
A B	bedeutet *A* gefolgt von *B* (stärkste Bindung)
A | *B*	bedeutet *A* oder *B*
[A]	bedeutet (| A)
{ A }	bedeutet [A { A }]
{ A , }	bedeutet [A { "," A }]

Die Lexis hat sich gegenüber der in [Gawecki, Wienberg 98] angegebenen nur in einem Punkt verändert: Zu den im dortigen Abschnitt 2.6 angegebenen *Other Tokens* ist das Symbol "@" hinzugekommen. Da das Symbol nie zu den vorhandenen Operatoren in Konflikt kommt, muß keine Präzedenz definiert werden. Größere Änderungen ergeben sich nur an der EBNF-Grammatik aus Abschnitt 8 von [Gawecki, Wienberg 98]. Die aktualisierte Grammatik wird im folgenden wiedergegeben.

A.1. Werte

Value ::=
    self
    | *string* | *char* | *Number* | *symbol*
    | "(" *Sequence* ")"
    | *Value BinOp Value*
    | *Value LazyBinOp Value*
    | *UnaryOp Value*
    | *(Value* | **super***)* "." *Selector* [*Arguments*] ["@"]
    | *identifier* [*Arguments*] ["@"]
    | *(Value* | **super***)* "." *Selector* ":=" *Value*
    | *identifier* ":=" *Value*
    | *Value* "[" *PositionalArguments* "]" ["@"]
    | *Value* "[" *PositionalArguments* "]" ":=" *Value*
    | [**fun** "(" { *ValueSignature* , } ")" [("@:" | ":") *Type*]] "{" *Sequence* "}"
    | "#(" { *Value* , } ")"
Sequence ::=
    { *Value* | *Binding* }
Number ::=
    ["-"](*int* | *long* | *real*)
Selector ::=
    identifier | *string*
Arguments ::=
    "(" *PositionalArguments KeywordArguments* ")"
PositionalArguments ::=
    { *PositionalArgument* , }
PositionalArgument ::=
    Value | *Type*
KeywordArgument ::=
    Keyword Value
Keyword ::=
    identifier ":"
Binding ::=
    identifier [("@:" | ":") *Type*] ":=" *Value* |
    identifier ("@:" | ":") *Type*
LazyBinOp ::=
    "&&" | "||"
BinOp ::=
    "=" | "==" | "!=" | "!==" |
    ">" | ">=" | ">>" |
    "<" | "<=" | "<<" |
    "↑" | "&" | "|"

A.2. Typen

Type ::=
    Self |
    identifier ["(" { *Type* , } ")"] |
    Fun "(" { *ValueSignature* , } ")" ("@:" | ":") *Type*

A.3. Klassen und Methoden

Class ::=
    class *ClassNameDomain*
    [**super** { *Type* , }]
    [*documentationString*]
    [*SelfSignature*]
    [**meta** *Type*]
    "{" { *MethodDefinition* | *SlotDefinition* }
        [**private** { *MethodDefinition* | *SlotDefinition* }]
    "}"
ClassNameDomain ::=
    identifier ["(" { *TypeSignature* , } ")"]
SelfSignature ::=
    Self "<:" *Type* |
    Self "=" *Type*
SlotDefinition ::=
    Selector ("@:" | ":") *Type*
      [*documentationString*]
MethodDefinition ::=
    Selector ["(" *Signatures KeywordParameters* ")"] ("@:" | ":") *Type*
      [*documentationString*]
      [**require** *Value*]
      [**ensure** *Value*]
      MethodBody
KeywordParameters ::=
    { *Keyword ValueSignature* [":=" *Value*] }
MethodBody ::=
    "{" *Sequence* "}" |
    builtin ["{" *Sequence* "}"] |
    deferred |
    extern *ForeignName*
ForeignName ::=
    string

A.4. Signaturen

Signature ::=
    ValueSignature | *TypeSignature*
ValueSignature ::=
    [*identifier*] ("@:" | ":") *Type*
TypeSignature ::=
    identifier "<:" *Type* |
    identifier "=" *Type*
Signatures ::=
    { *Signature* , }

Literaturverzeichnis

Achauer 93: Bruno Achauer. „The DOWL Distributed Object-Oriented Language“. *Communications of the ACM*, 36(9):48–55, 1993.

Almeida 97: Paulo Sérgio Almeida. „Balloon Types: Controlling Sharing of State in Data Types“. In: Mehmet Aksit und Satoshi Matsuoka, Hrsg., *ECOOP '97 – Object-Oriented Programming. 11th European Conference Jyväskylä, Finland*, Nummer 1241 in Lecture Notes in Computer Science, Seite 32–59. Springer-Verlag, Juni 1997.

Barret et al. 96: Kim Barret, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, und P.Tucker Withington. „A Monotonic Superclass Linearization for Dylan“. In: OOPSLA [96], Seite 69–82.

Blake, Cook 87: Edwin Blake und Steve Cook. „On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk“. In: J. Bézivin, J.-M. Hullot, P. Cointe, und H. Lieberman, Hrsg., *ECOOP '87 – European Conference on Object-Oriented Programming*, Nummer 276 in Lecture Notes in Computer Science, Seite 41–50. Springer-Verlag, Juni 1987. Paris, France.

Brodie 84: Michael L. Brodie. „On the Development of Data Models“. In: Michael L. Brodie, John Mylopoulos, und Joachim W. Schmidt, Hrsg., *On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Kapitel 2, Seite 19–47. Springer-Verlag, 1984.

Cardelli, Gordon 98: Luca Cardelli und Andrew D. Gordon. „Mobile Ambients“. In: Maurice Nivat, Hrsg., *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998)*, Band 1378, *Lecture Notes in Computer Science*, Seite 140–155. Springer-Verlag, 1998.

Civello 93: F. Civello. „Roles for composite objects in object-oriented analysis and design“. In: OOPSLA [93], Seite 376–393.

DePauw et al. 94: Wim DePauw, Doug Kimelman, und John Vlissides. „Modeling Object-Oriented Program Execution“. In: M. Tokoro und R. Pareschi, Hrsg., *Proceedings of the 8th European Conference on Object-Oriented Programming, ECOOP '94, Bologna, Italy*, Seite 163–182. Springer-Verlag, Berlin, Germany, Juli 1994.

Ernst 98: Matthias Ernst. „Typüberprüfung in einer polymorphen objektorientierten Programmiersprache. Analyse, Design und Implementierung eines Typprüfers für Tycoon-2“. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Deutschland, Juni 1998.

Forrester 72: Jay W. Forrester. *Grundzüge einer Systemtheorie.* Gabler Wiesbaden, 1972.

Fowler, Scott 97: Martin Fowler und Kendal Scott. *UML Distilled: Applying the Standard Objekt Modeling Language.* Addison-Wesley, 1997.

Gamma et al. 95: E. Gamma, R. Helm, R. Johnson, und J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

Gawecki, Wienberg 98: Andreas Gawecki und Axel Wienberg. „Report on the Tycoon-2 Programming Language Version 1.0“. Technical documentation of Higher-Order GmbH, Hamburg, Februar 1998.

Gil, Lorenz 96: Joseph Gil und David H. Lorenz. „Environmental Acquisition – A New Inheritance-Like Abstraction Mechanism“. In: OOPSLA [96], Seite 214–231.

Goldberg, Robson 83: A. Goldberg und D. Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

Gosling et al. 96: James Gosling, Bill Joy, und Guy Steele. *The Java Language Specification.* Addison-Wesley, August 1996.

Griffel 98: Frank Griffel. *Componentware : Konzepte und Techniken eines Softwareparadigmas.* dpunkt-Verlag, Heidelberg, 1998.

Hamer 97: Keno Hamer. „Visualisierung von Objektinteraktionen“. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Deutschland, Juli 1997.

Harms, Weide 91: Douglas E. Harms und Bruce W. Weide. „Copying and Swapping: Influences on the Design of Reusable Sortware Components“. *IEEE Transactions on Software Engineering*, Seite 424–434, Mai 1991.

Hoare, Wirth 73: C. A. R. Hoare und Niklaus Wirth. „An Axiomatic Definition of the Programming Language PASCAL“. *Acta Informatica*, 2:335–355, 1973.

Hogg 91: John Hogg. „Islands: Aliasing Protection In Object-Oriented Languages“. In: *Proceedings of OOPSLA '91*, Seite 271–285, Oktober 1991.

Hopcroft, Ullman 79: John E. Hopcroft und Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

Hutchinson 87: N. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* Dissertation, Department of Computer Science, University of Washington, 1987.

Jul 89: E. Jul. *Object Mobility in a Distributed Object-Oriented System.* Dissertation, Department of Computer Science, University of Washington, 1989.

Kent, Maung 95: Stuart Kent und Ian Maung. „Encapsulation and Aggregation". In: *TOOLS Pacific '95 – Technology of Object-Oriented Languages and Systems.* Prentice Hall, 1995.

Kernighan, Ritchie 78: Brian W. Kernighan und Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1978.

Kernighan, Ritchie 88: Brian W. Kernighan und Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2. Auflage, 1988.

Kim et al. 87: Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, und Darrel Woelk. „Composite object support in an object-oriented database system". In: OOPSLA [87], Seite 118–125.

Koskimies, Mössenböck 96: Kai Koskimies und Hanspeter Mössenböck. „Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs". In: *International Conference on Software Engineering (ICSE '96), Berlin*, 1996.

Kristensen 94: Bent Bruun Kristensen. „Complex Associations: Abstractions in Object-Oriented Modeling.". In: *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages & Applications*, Seite 272–286, Portland, Oregon, USA, Oktober 1994. ACM.

Lange, Nakamura 95: D. B. Lange und Y. Nakamura. „Interactive Visualization of Design Patterns Can Help in Framework Understanding". In: *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages & Applications*, Seite 342–357, Austin, Texas, USA, Oktober 1995. ACM.

Madsen et al. 93: Ole Lehrmann Madsen, Birger Møller-Pedersen, und Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

Matthes, Wienberg 97: F. Matthes und A. Wienberg. „Visualizing Persistent Objects using Higher-Order Functions in SGML". Technischer Bericht, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, Germany, September 1997.

Meyer 88: Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

Meyer 92: Bertrand Meyer. *Eiffel: the Language.* Prentice Hall Object-Oriented Series, 1992.

Meyer 97: Bertrand Meyer. *Object-oriented Software Construction, 2nd edition.* Prentice Hall, 1997.

Minsky 96: Naftaly H. Minsky. „Towards Alias-Free Pointers“. In: Pierre Cointe, Hrsg., *ECOOP '96 – Object-Oriented Programming*, Nummer 1098 in Lecture Notes in Computer Science, Seite 189–209. Rutgers University, New Brunswick NJ 08903 USA, Springer-Verlag, Juli 1996. 10th European Conference Linz, Austria.

Noble et al. 98: James Noble, Jan Vitek, und John Potter. „Flexible Alias Protection“. In: Eric Jul, Hrsg., *ECOOP '98 – Object-Oriented Programming. 12th European Conference Brussels, Belgium*, Nummer 1445 in Lecture Notes in Computer Science, Seite 158–185. Springer-Verlag, Juli 1998.

OOPSLA 87: ACM. *OOPSLA '87*, Orlando, Florida, USA, Oktober 1987.

OOPSLA 93: ACM. *OOPSLA '93*, San Jose, California, Oktober 1993.

OOPSLA 96: ACM. *Proceedings of OOPSLA '96*, San Jose, California, Oktober 1996.

Petre et al. 98: M. Petre, A. F. Blackwell, und T. R. G. Green. „Cognitive Questions in Software Visualisation“. In: J. Stasko, J. Domingue, B. Price, und M. Brown, Hrsg., *Software Visualization: Programming as a Multi-Media Experience*. MIT Press, Januar 1998.

Schneider 98: Daniel Schneider. „Eine bidirektionale typisierte Kommunikation zwischen zwei reflexiven, objektorientierten Sprachen“. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Deutschland, Januar 1998.

Sefika et al. 96: Mohlalefi Sefika, Aamod Sane, und Roy H. Campbell. „Architecture-Oriented Visualization“. In: OOPSLA [96], Seite 389–405.

Stephenson 92: Neal Stephenson. *Snow Crash*. Bantam Books, Juni 1992.

Stoutamire 97: David Petrie Stoutamire. *Portable, Modular Expression of Locality*. Dissertation, University of California at Berkeley, Dezember 1997.

Stroustrup 86: Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

Stroustrup 92: B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2. Auflage, 1992.

Szyperski 98: Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

Ungar, Smith 87: David Ungar und R. B. Smith. „Self: The Power of Simplicity“. In: OOPSLA [87], Seite 227–243.

Wahlen 98: Jens Wahlen. „Entwurf einer objektorientierten Sprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen“. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Deutschland, Juni 1998.

Weikard 98: Marc Weikard. „Entwurf und Implementierung einer portablen multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache“. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Deutschland, Juli 1998.

Wienberg 97: Axel Wienberg. „Bootstrap einer persistenten objektorientierten Programmierumgebung“. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Deutschland, August 1997.

Wirth 71: Niklaus Wirth. „The Programming Language PASCAL“. *Acta Informatica*, 1, 1971.

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 30. März 1999

(Axel Wienberg)