Kooperierende Objektsysteme: Betrieb und Entwicklung

Vom Promotionsausschuß der
Technischen Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
genehmigte Dissertation

von

Gerald Schröder

aus Hamburg

1999

1. Gutachter: Prof. Dr. J.W. Schmidt
2. Gutachter: Prof. Dr. F.H. Vogt
Tag der mündlichen Prüfung: 8. April 1999

Danksagung

An dieser Stelle möchte ich allen danken, die mich während der Promotionsphase unterstützt haben. Insbesondere danke ich Herrn Prof. Dr. J.W. Schmidt, der mir die Promotion ermöglicht hat, Herrn Prof. Dr. F.H. Vogt für das Interesse an meiner Arbeit und Herrn Prof. Dr. G.-F. Mayer-Lindenberg für die freundliche Leitung des Prüfungsausschusses.

Während der schwierigen Phasen haben mich meine Kollegen und Studenten auf vielfältige Weise immer wieder zum Weitermachen ermuntert; besonderer Dank dafür gebührt Claudia Niederée, Andreas Rudloff, Ulrike Steffens, André Wittenburg, Ilona Hachmann, Marc Weikard, Helen Pert, Claudius Haasis und Susanne Möller. Die technische Basis und viele Ideen dieser Arbeit entstanden in Zusammenarbeit und Diskussionen mit Prof. Dr. Florian Matthes, Dr. Bernd Mathiske, Dr. Ingrid Wetzel, Rainer Müller, Andreas Gawecki, Andreas Geisler, Gerd Bremer, Marcel Kornacker und Andreas Piellusch. "Meine" Studienarbeiter und Diplomanden Andreas Bösch, Christian Koch, Oliver Nietsch, Kay Ramme und Nastaran Vaziri haben mit viel Geduld Beiträge zu dieser Arbeit geleistet. Besonders befruchtend für die Klärung meiner Gedanken waren die Diskussionen mit Jens Wahlen und Axel Wienberg. Ohne die von Thomas Sidow und Thomas Rahmlow trotz meiner ständigen Störversuche am Leben erhaltene technische Infrastruktur wäre diese Arbeit nie möglich gewesen. Ulrike Hantschmann hat mir oft den Rücken freigehalten, damit ich meine Arbeit beenden konnte.

Mein herzlichster Dank gilt meinen Eltern, die mich vorbehaltlos unterstützt haben und immer für mich da waren, wenn ich sie brauchte. Ihnen ist diese Arbeit gewidmet.

Inhaltsverzeichnis

1 Einleitung: Überblick und Zielsetzung **6**
1.1 Kooperation . . . 8
1.2 Betrieb . . . 9
1.3 Entwicklung . . . 11
1.4 Zielsetzung und Ergebnisüberblick . . . 11

2 Objektsysteme **14**
2.1 Objekt . . . 15
2.1.1 Datum . . . 16
2.1.2 Funktion . . . 18
2.1.3 Prozeß . . . 23
2.2 Bindung . . . 25
2.3 Repräsentation . . . 26
2.3.1 Interpreter . . . 27
2.3.2 Übersetzer . . . 28
2.4 Objektsystem . . . 29
2.5 Ablaufumgebung und Ablaufsystem . . . 30
2.5.1 Externes Objekt . . . 32
2.5.2 Basisobjekt . . . 32
2.6 Entwicklungsumgebung und Entwicklungssystem . . . 33
2.6.1 Standardobjekt . . . 34
2.6.2 Entwicklungsobjekt . . . 35
2.7 Zusammenfassung . . . 35

3 Betrieb von Objektsystemen **37**
3.1 Ablauf von Objektsystemen . . . 37
3.1.1 Lebenszyklus . . . 37
3.1.2 Ablaufzustände . . . 39
3.1.3 Reflektion . . . 40
3.1.4 Objektfreigabe . . . 41
3.1.5 Generische Ressourcenfreigabe . . . 42
3.2 Persistente Objektsysteme . . . 45
3.2.1 Lebensdauer von Objekten . . . 45
3.2.2 Persistente Repräsentation eines Ablaufzustands . . . 46
3.2.3 Wiederherstellung von Ablaufzuständen . . . 48
3.3 Evolution von Objektsystemen . . . 49
3.3.1 Ablauffähigkeit und Integrität . . . 51
3.3.2 Evolutionsunterstützung . . . 56
3.4 Zusammenfassung . . . 60

4 Kooperation von Objektsystemen in heterogenen Umgebungen **61**
4.1 Kooperationsmodell . . . 62
4.1.1 Ablauf einer Kooperation . . . 63
4.1.2 Kommunikation . . . 64
4.1.3 Kooperationsmechanismen und -strategien . . . 66
4.1.4 Heterogenität und Adaption . . . 67
4.2 Übertragung von Objekten . . . 71
4.2.1 Portierung . . . 72
4.2.2 Repräsentationsumwandlung . . . 74
4.2.3 Übertragung von Bindungen . . . 76
4.3 Bindungen zwischen Objektsystemen . . . 78
4.3.1 Eigenschaften externer Bindungen . . . 79
4.3.2 Operationen auf externen Objekten . . . 81
4.3.3 Objektfreigabe und externe Bindungen . . . 83
4.3.4 Externe Bindungen an Objekte des Ablaufsystems . . . 87
4.3.5 Persistenz externer Objekte . . . 89
4.3.6 Evolution . . . 90
4.4 Zusammenfassung . . . 92

5 Entwicklung kooperierender Objektsysteme **94**
5.1 Modellierung . . . 95
5.1.1 Entwicklungsanforderungen und -mechanismen . . . 96
5.1.2 Anwendungsgerechte und ablaufgerechte Repräsentation . . . 97
5.1.3 Übersetzung der anwendungsgerechten in die ablaufgerechte Repräsentation 99
5.1.4 Entwurf der anwendungsgerechten Repräsentation . . . 101
5.2 Validierung . . . 102
5.2.1 Typisierung . . . 103
5.2.2 Statische Validierung von Kooperationen . . . 106
5.2.3 Dynamische Validierung von Kooperationen . . . 108
5.2.4 Kooperation der Entwicklungssysteme zur Validierung . . . 109
5.2.5 Reflektive Unterstützung der Validierung . . . 112
5.2.6 Versionierung im Typsystem . . . 114
5.3 Generierung . . . 116
5.3.1 Entwicklungsmuster . . . 117
5.3.2 Dynamische Generierung . . . 118
5.3.3 Statische Generierung . . . 119
5.3.4 Anwendung auf die Entwicklung kooperierender Objektsysteme . . . 121
5.4 Zusammenfassung . . . 121

6 Ergebnisse **123**
6.1 Zusammenfassung . . . 123
6.2 Ausblick . . . 125

Abbildungsverzeichnis

1.1 Ablauf-, Kooperations- und Entwicklungsumgebung 8

2.1 Objekte und Bindungen 15
2.2 Basisdaten 16
2.3 Strukturierte Daten 17
2.4 Statische Bindungen in einer Funktion 19
2.5 Ausführung einer Funktion 19
2.6 Dynamische Bindungen in einer Funktion 20
2.7 Funktionen höherer Ordnung 20
2.8 Funktionsabschluß 21
2.9 Gemeinsam durch mehrere Funktionen genutzte Objekte 22
2.10 Prozeß 23
2.11 Repräsentation, Übersetzer und Interpreter 26
2.12 Grafische Repräsentation der Zahl −1 27
2.13 Repräsentationen für ganze Zahlen und Interpreter 28
2.14 Repräsentationen und Übersetzer 29
2.15 Übersetzen einer Funktion 30
2.16 Objektsysteme, Ablauf- und Entwicklungsumgebungen 30
2.17 Beispiel für Objektsysteme und Umgebungen 31
2.18 Ablaufumgebung 31
2.19 Unterschiedliche Ablaufsysteme 32
2.20 Entwicklungsumgebung 33
2.21 Unterschiedliche Entwicklungssysteme 34
2.22 Ablaufumgebung: Szenario 36

3.1 Lebenszyklus eines Objektsystems 38
3.2 Objektsysteme und Ablaufsysteme 39
3.3 Nutzbare und nicht nutzbare Objekte 41
3.4 Speicherungssystem und persistente Repräsentation 46
3.5 Persistenz von Seiteneffekten 49
3.6 Zustands- und Bindungsevolution 50
3.7 Schnittstelle der Zeichenkettenverarbeitung 52
3.8 Implementation der Zeichenkettenverarbeitung 53
3.9 Geänderte Repräsentation der Zeichenkettenverarbeitung 53
3.10 Zeichenkettenverarbeitung mit anderen Indices 55
3.11 Evolution mit Kontexten 58
3.12 Adaption der Zeichenkettenverarbeitung 58

4.1 Kooperation 63
4.2 Aufbau einer Kommunikationsverbindung 65
4.3 Charakteristika von Verbindungen 67
4.4 Übertragung eines Objekts 72
4.5 Migration vs. Kopie eines Objekts 73

4.6 Portierung von Repräsentationen 74
4.7 Portierung von unveränderten oder unveränderlichen Objekten 75
4.8 Übertragung von Bindungen 76
4.9 Neubindung 77
4.10 Externe Bindung 79
4.11 Operationen auf externen Objekten 81
4.12 Rückruf in Kooperationen 83
4.13 Kooperation zwischen Ablaufsystem und Objektsystem über Basisobjekte 88
4.14 Geschachtelte Ablaufumgebungen 89
4.15 Evolution in einer Kooperation 90
4.16 Zwei kooperierende Klienten und ein geänderter Dienstleister 92

5.1 Entwicklungsszenario 95
5.2 Übersetzer 98
5.3 Wohlgeformtheit 104
5.4 Validierer 104
5.5 Subtypprüfer 105
5.6 Linguistische Reflektion 113
5.7 Evolution von Typen bei Übertragung von Objekten 116
5.8 Entwicklungsmuster 117
5.9 Syntaxerweiterungen 120

Kapitel 1

Einleitung: Überblick und Zielsetzung

> [Es] nehmen sowohl die Komplexität der Aufgaben als auch die Spezialisierung der Individuen ständig zu, so daß Kooperationen einerseits unerläßlich werden und andererseits auch Ergebnisse erzielt werden können, die die Summe der von isolierten Individuen erbringbaren Leistungen übertreffen (z.B. durch Synergieeffekte). [Burger 97]

Kooperation zwischen Softwaresystemen tritt heutzutage in verschiedenen Arten auf, wobei Kommunikationsmechanismen wie Aufruf über eine Schnittstelle, gemeinsame Speicherbereiche oder Nachrichtenaustausch über einen Kommunikationskanal vorausgesetzt werden:

- Kooperation zwischen Anwendung und Betriebssystem;
- Kooperation zwischen parallel auf einem Rechner ablaufenden Anwendungen;
- Kooperation zwischen auf mehreren Rechnern ablaufenden Anwendungen.

Durch die zunehmende weltweite Vernetzung der Rechner [Lennon 97] und darauf aufbauende Anwendungen, wie integrierte virtuelle Unternehmen und mobile Informationssysteme [Silberschatz, Zdonik 96], ist insbesondere die Kooperation zwischen auf mehreren Rechnern ablaufenden Anwendungen in den Mittelpunkt des Interesses gerückt. Auf diesen vernetzten Rechnern laufen Objektsysteme[1] ab, welche durch die Vernetzung miteinander kooperieren können. Daraus entstehen

- verteilte Datenverarbeitungssysteme (*distributed data processing systems*) [Enslow 78],
- verteilte Rechensysteme (*distributed computing systems*) [Ananda, Srinivasan 91] und
- globale Rechner (*global computers*) [Cardelli 97a].

Derartige Systeme zeichnen sich durch folgende essentielle Merkmale [Enslow 78; Notkin et al. 86; Nicol et al. 93; Briot et al. 96; Rosenblum, Wolf 97] aus, die dieser Arbeit als Grundlage dienen:

Kooperative Autonomie: Die Objektsysteme sind autonom, also keiner gemeinsamen Kontrollinstanz unterworfen, verhalten sich aber kooperativ, sind also prinzipiell zur Zusammenarbeit mit anderen Objektsystemen fähig.

[1] In dieser Arbeit wird unter dem Begriff "Objektsystem" sowohl Anwendungs- als auch Ablaufsoftware bis hin zum Mikrocode eines Prozessors zusammengefaßt, analog zu dem unifizierenden Ansatz von [Lampson 94] für die Hardwarestruktur vom Prozessor bis zum weltweiten Netzwerk. Der Begriff "Objekt" ist in dieser Arbeit nicht im Sinne von "objektorientierten" Systemen (vgl. beispielsweise [Meyer 88]) zu verstehen, sondern bezeichnet eine bindbare Einheit eines Objektsystems, genauer ein Datum, eine Funktion oder einen Prozeß. Näheres zu diesen Begriffen findet sich in Kapitel 2.

Heterogenität: Die Objektsysteme unterscheiden sich neben ihrem geografischen Standort in Repräsentation, Aufbau, Leistungsfähigkeit, verfügbaren Ressourcen, unterstützten Protokollen, Sicherheitsanforderungen usw.

Diese Arbeit beschreibt den Betrieb und die Entwicklung dieser Klasse von Systemen, die hier als "kooperierende Objektsysteme" bezeichnet werden. Dabei wird schwerpunktartig ein Modell für Objektsysteme entwickelt, das zur Beschreibung essentieller Aspekte des Betriebs und der Entwicklung kooperierender Objektsysteme dient. Die behandelten Aspekte werden Umgebungen zum Ablauf, zur Kooperation und zur Entwicklung zugeordnet (vgl. Abbildung 1.1):

Ablaufumgebung:

- Ausführung: Interpretation eines Objektsystems, die einen Ablauf (*computation*) ergibt;
- Persistenz: Verlängerung der Lebensdauer von Objekten über einen Ablauf hinaus;
- Evolution: Änderung eines Objektsystems während des Ablaufs;
- Sicherheit: Authentisierung, Autorisierung und Verschlüsselung während eines Ablaufs;
- Verhaltensreflektion: anwendungsgesteuerte Änderung des Ablaufverhaltens eines Objektsystems.

Kooperationsumgebung:

- Kommunikation: Verbindung zwischen Objektsystemen, die Kooperation ermöglicht;
- Adaption: Umgang mit heterogenen Objektsystemen;
- Entfernter Aufruf: Auslösen von Operationen auf Objekten anderer Objektsysteme;
- Migration: Übertragung von Objekten zwischen Objektsystemen;
- Entfernte Bindung: Bindungen zwischen Objekten verteilter Objektsysteme.

Entwicklungsumgebung:

- Modellierung: Beschreibung eines Objektsystems und Übersetzung in eine ablauffähige Repräsentation;
- Strukturierung: Aufteilung eines Objektsystems in Teilsysteme und Zusammensetzung eines Objektsystems aus diesen Teilen;
- Validierung: Typisierung der Objekte eines Objektsystems und Prüfung der Typverträglichkeit;
- Generierung: Erzeugung von Teilen eines Objektsystems durch Instanziierung von Mustern;
- Linguistische Reflektion: Behandlung von Entwicklungsaufgaben während der Erzeugung oder des Ablaufs des Objektsystems.

Die in dieser Arbeit erläuterten Konzepte wurden im Rahmen des Tycoon Projektes an der Universität Hamburg und der Technischen Universität Hamburg-Harburg entwickelt. Den Ausgangspunkt dieses Projektes bildet [Matthes 93], der die Modellierung, die Ausführung und die Persistenz von Objektsystemen beschreibt; die Realisierung der Persistenz wird von [Kornacker 95; Müller iV] vertieft. [Mathiske 96] legt die technischen Grundlagen für die Migration von Objekten. Sicherheit wird erschöpfend von [Rudloff iV] erfaßt. Strukturierung wird von [Matthes 93; Koch 97; Wienberg iV] behandelt, die Validierung von Objektsystemen ist Thema von [Matthes 93; Geisler 95]. Die Generierung findet sich auch in [Schröder 93; Wetzel 95]. Der Beitrag dieser Arbeit liegt in der Aufstellung eines allgemeinen Objektmodells zur Beschreibung dieser Konzepte und Technologien, in der Ergänzung um Reflektion, Evolution, entfernte Bindungen sowie weiteren Freiheitsgraden bei der Validierung und Generierung. Die beschriebenen Kooperationskonzepte werden beispielsweise bei der Realisierung von Workflow-Anwendungen [Johannisson 97; Matthes 97] und digitalen Bibliotheken [Schmidt et al. 97] eingesetzt.

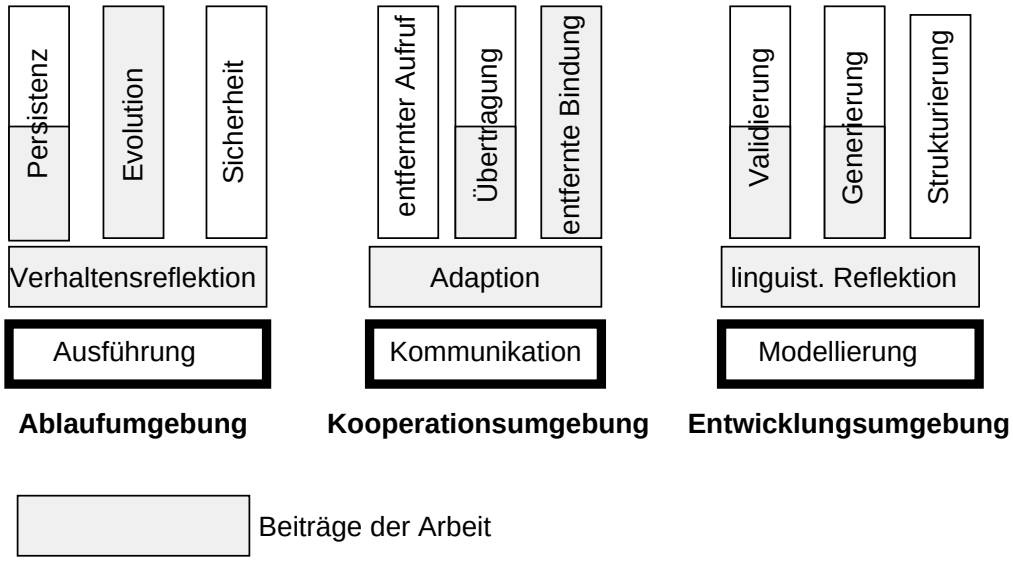

Abbildung 1.1: Ablauf-, Kooperations- und Entwicklungsumgebung

In diesem Abschnitt wird zuerst der in dieser Arbeit verwendete Kooperationsbegriff erläutert (vgl. Abschnitt 1.1). Anschließend wird kurz auf den Betrieb kooperierender Objektsysteme eingegangen (vgl. Abschnitt 1.2). In Abschnitt 1.3 wird die Entwicklung kooperierender Objektsysteme überblicksartig diskutiert. Die Zielsetzung und ein Überblick über die Ergebnisse der Arbeit beschließen dieses einleitende Kapitel (vgl. Abschnitt 1.4).

1.1 Kooperation

Kooperation ist mehr als nur Kommunikation [Nicol et al. 93]: Kooperation bedeutet, daß die Kooperationspartner eine Aufgabe gemeinsam bearbeiten [Burger 97]. Bestimmte Aufgaben können nur in Kooperation bearbeitet werden, beispielsweise wenn einem Kooperationspartner Ressourcen nicht zur Verfügung stehen, die er zur Bearbeitung einer Aufgabe benötigt; andere Aufgaben werden gemeinsam einfacher, schneller oder billiger gelöst, beispielsweise durch Arbeitsteilung unter Ausnutzung von speziellen Fähigkeiten.

Bei der Kooperation von Objektsystemen zur Bearbeitung einer Aufgabe sind folgende Fragen zu beantworten [Baldi et al. 97; Ghezzi, Vigna 97]:

- Wo befinden sich die Ressourcen, beispielsweise die zu verarbeitenden Daten?
- Wo befindet sich das Bearbeitungswissen, die Regeln, repräsentiert durch Funktionen?
- Wo wird die Ausführung vorgenommen, d.h. wo läuft der aufgabenbearbeitende Prozeß ab?
- Was wird zwischen den Kooperationspartnern ausgetauscht (Daten, Funktionen, Prozesse)?

Aus der Beantwortung dieser Fragen für eine Anwendung ergeben sich die anwendungsspezifischen Anforderungen an die Mobilität von Daten, Code und Prozessen [Mathiske et al. 97], welche erst die Kooperation ermöglichen. Zusammengefaßt ergeben sich folgende Kooperationsparadigmen [Picco et al. 97; Carzaniga et al. 97; Baldi et al. 97; Ghezzi, Vigna 97]:

1. **C/S** (*Client/Server*): Ein Klient fordert einen Dienstleister auf, eine Dienstleistung für ihn zu erbringen, also eine Funktion auszuführen, die der Dienstleister anbietet. Es werden nur Daten zwischen den beiden Kooperationspartnern übertragen. Diese Art der Kooperation

wird als entfernter Prozeduraufruf (RPC, *Remote Procedure Call*) [Birrell, Nelson 84] oder entfernter Methodenaufruf (RMI, *Remote Method Invocation*) [Sun 97; Orfali et al. 97] bezeichnet. In der synchronen Version wartet der Klient auf das Ergebnis, während er in der asynchronen Form parallel weiterarbeitet [Liskov, Shriva 88; Walker et al. 90]. Teilweise sind auch Rückrufe des Dienstleisters an den Klienten (*callback*) möglich [Cohrs et al. 88]. Eine Standardierung des C/S Ansatzes im objektorientierten Kontext ist OMA (*Object Management Architecture*) [OMG 97b] bzw. CORBA (*Common Object Request Broker Architecture*) [OMG 98]. Die Art der übertragbaren Parameter ist eingeschränkt auf Daten und Referenzen [OMG 98]. Wenn Funktionen als Parameter und Resultat zugelassen sind (Funktionen höherer Ordnung), ergibt sich das Paradigma REV/COD; wenn auch Prozesse übertragen werden können, ergibt sich MA.

2. **REV** (*Remote EValuation*) und **COD** (*Code On Demand*): Ein Klient überträgt eine Funktion an einen Dienstleister, der sie ausführt (REV), oder ein Klient fordert eine Funktion von einem Dienstleister an, um sie auszuführen (COD). Die Übertragung einer Funktion an einen Dienstleister bietet sich an, wenn Berechnungen auf großen Datenmengen vorgenommen werden sollen und der Berechnungsalgorithmus sich auf einem anderen Rechner als die Daten befindet [Stamos, Gifford 90]. Die Anforderung von Funktionen kommt zum Einsatz, wenn lokal ein Algorithmus nicht zur Verfügung steht, der für eine Berechnung benötigt wird [Diehl 98].

3. **MA** (*Mobile Agents*): Ein Kooperationspartner gibt einen Prozeß, also eine Funktion in Ausführung, an einen anderen Kooperationspartner weiter, damit dieser die Ausführung fortsetzt. In diesem Fall wird nicht nur eine Funktion, sondern auch ein Ausführungszustand, also ein Prozeß, übertragen [Smith 88; Hsieh et al. 93; Nuttall 94; Steensgard, Jul 95; Geib et al. 96; Mathiske 96]. Diese Prozesse werden Agenten [Wayner 95; Bradshaw 96; Caglayan, Harrison 97] genannt, wenn sie selbständig entscheiden, wann und mit welchem Ziel sie den Ausführungskontext wieder verlassen. Migrierende Prozesse nutzen lokale Ressourcen [Hsieh et al. 93] und interagieren mit lokalen Benutzern [Bharat, Cardelli 95].

Bei allen Kooperationsparadigmen sind die Bindungen zu beachten:

- Werden gebundene Objekte mit übertragen oder
- werden die Bindungen in entfernte Bindungen umgewandelt oder
- werden im empfangenden Objektsystem lokale Objekte angebunden?

Als Grundlage für die Diskussion dieser Fragen wird in Kapitel 2 wird ein auf den Begriffen Objekt und Bindung basierendes Objektsystemmodell entworfen.

1.2 Betrieb

Der Betrieb von kooperierenden Objektsystemen erfordert auf Kooperation ausgerichtete Ablaufsysteme[2] [Horn, Cahill 91; Lea et al. 93]. Da die Ablaufsysteme nicht den anwendungsspezifischen Kooperationskontext kennen [Saltzer et al. 84], stellen sie Basismechanismen zur Verfügung, die von der Anwendung gemäß ihrer Kooperationsstrategie eingesetzt werden [Levin et al. 75; Lea et al. 93].

Strukturell handelt es sich damit um Ablaufsysteme mit einem Mikrokern [Lea et al. 93; Cheung, Loong 95; Jacquemot et al. 96]. Die Anwendung erweitert diesen Kern um anwendungsspezifische Objekte, die das Verhalten des Ablaufsystems beim Ablauf der Anwendung spezialisieren [Habert et al. 90]. Diese Objekte reagieren auf Ereignisse, die vom Ablaufsystem erkannt

[2] Im Kontext dieser Arbeit wird statt von "Betriebssystemen" von "Ablaufsystemen" gesprochen, um die Einbeziehung des Prozessors oder Interpreters zu betonen. Vergleiche dazu Kapitel 2.

werden [Menon et al. 93], beispielsweise Umgebungsänderungen. Es handelt sich dabei um eine bewußte Adaption der Anwendung an die Umgebungsänderungen (*application-aware adaption*) im Gegensatz zu einer für die Anwendung transparenten, unter einer Abstraktionsschicht des Ablaufsystems verborgenen Adaption [Noble, Satyanarayanan 95; Satyanarayanan et al. 95; Schill et al. 95]. Dazu gibt die Anwendung ihr Interesse an bestimmten Ereignissen bekannt, indem sie eine ereignisspezifische Funktion beim Ablaufsystem registriert [Cohrs et al. 88; Welling, Badrinath 97]. Das Ablaufsystem aktiviert dann beim Auftreten dieses Ereignisses die Funktion.

Zur Realisierung von objektbasierter Kooperation kollaborieren [Satyanarayanan et al. 95] Anwendung und Ablaufsystem. Der eingesetzte Kollaborationsmechanismus ist Reflektion [Smith 84; Maes 87; Yokoyama et al. 95] gemäß eines Metaobjektprotokolls [Kiszales et al. 91]. Das Verhalten jedes Objekts wird dabei potentiell durch ein Metaobjekt bestimmt; wenn kein Metaobjekt vorhanden ist, wird ein vom Ablaufsystem bestimmtes Standardverhalten implementiert. Es wird also indirekt das Ablaufsystem erweitert. Mit diesem Mechanismus wird beispielsweise die Persistenz [Paepke 90] und Migration [Lux 95; Holder, Ben-Shaul 97] von Objekten realisiert. Dabei zeigt sich, daß Persistenz und Migration auf ähnliche Basismechanismen wie Objektlinearisierung [Herlihy, Liskov 82] zurückgreifen [Chiba 95], so daß ein Synergieeffekt entsteht, wenn diese Basismechanismen herausfaktorisiert werden [Shrivastava et al. 91; Horn, Cahill 91; Mathiske et al. 95]. Diese Mechanismen werden außer im Kontext kooperierender Objektsysteme bespielsweise auch für persistente Anwendungen, Multimediaanwendungen und mobile[3] Anwendungen eingesetzt [Cheung, Loong 95].

Basismechanismen zur Realisierung von Kooperation sind neben den unverzichtbaren Kommunikationsmechanismen portable Repräsentationen [Gosling 95]. Portable Repräsentationen erlauben eine Interoperabilität in heterogenen Umgebungen [Mathiske 96] zur Realisierung der Kooperationsstile REV, COD und MA (vgl. Abschnitt 1.1). Im kommerziellen Sektor werden als Basismechanismen zur Kooperation zunehmend CORBA [OMG 97b; OMG 98] und Java [Java Team et al. 96; Lindholm, Yellin 96] eingesetzt [Redlich 96; Curtis 97; Vogel, Duddy 97; Orfali, Harkey 97; Baker 97; Bösch 97]:

CORBA bietet einen ortsunabhängigen Zugriff auf Objekte und standardisierte Kommunikationsmechanismen zwischen Objektsystemen in heterogenen Umgebungen, allerdings mit der Einschränkung auf das Kooperationsparadigma C/S.

Java realisiert eine plattformunabhängige Coderepräsentation, so daß die Kooperationsparadigmen REV und COD, insbesondere im Kontext des World Wide Web als Ablaufsystem [Black, Walpole 95; Manola 98], implementiert werden.

Weitere Basismechanismen werden in CORBA durch die CORBAservices [OMG 97a] spezifiziert, beispielsweise zur Persistenz [Sessions 96] oder Sicherheit (Authentisierung, Autorisierung, Verschlüsselung) [Birrell, Nelson 84; Stamos, Gifford 90; Doorn et al. 95; Rudloff iV]. Eine endgültige Integration der Mechanismen und eine Implementation des Kooperationsparadigmas MA im CORBA Kontext steht noch aus.

Im Rahmen dieser Arbeit werden in Kapitel 3 grundlegende Mechanismen zum Betrieb von Objektsystemen eingeführt, insbesondere im Bereich der integritätserhaltenden Speicherverwaltung von Objekten und Bindungen. Die (reflektive) Kollaboration von Anwendungen und Ablaufsystem wird am Beispiel der Persistenz und Evolution aufgezeigt.

Die Mechanismen zur Kooperation werden darauf aufbauend in Kapitel 4 betrachtet. Insbesondere wird die Übertragung von Objekten und Bindungen in heterogenen Umgebungen und die Behandlung von Bindungen zwischen Objektsystemen betrachtet.

[3] "Mobil" bedeutet hier, daß das gesamte Objektsystem und sein Ablaufsystem seinen Ausführungskontext wechselt. Dieser Fall tritt auf, wenn beispielsweise ein Laptop ein Netzwerk verläßt [Satyanarayanan et al. 95; Cardelli, Gordon 98]. Im Kontext dieser Arbeit werden die Herausforderungen der Mobilität unter Evolution (Anpassung an veränderte Umgebung) und Kooperation (zeitweise unerreichbare Kooperationspartner) behandelt.

1.3 Entwicklung

Die Entwicklung kooperierender Objektsysteme wird durch eine Vielzahl von Programmiersprachen und -systemen [Briot et al. 96; Mathiske 96; Thorn 97; Carzaniga et al. 97] mit unterschiedlichen Programmierparadigmen unterstützt:

imperativ in Argus [Liskov, Shriva 88; Liskov 88];

funktional in Scheme [Cejtin et al. 95];

objektorientiert in Emerald [Black et al. 87; Jul et al. 88], Distributed Smalltalk [Schelvis, Bledoeg 88; Bennett 90] und DOWL [Achauer 93];

datenflußorientiert in Distributed Oz [Roy et al. 97];

agentenorientiert in Messengers [Fukuda et al. 96; Fukuda et al. 97];

koordinationsorientiert in Laura [Tolksdorf 93; Tolksdorf 95] und ActorSpace [Callsen, Agha 94];

...

Diese Programmierparadigmen nutzen im Kern die in Abschnitt 1.1 eingeführten Kooperationsparadigmen, erweitern sie allerdings teilweise um zusätzliche Leistungen, beispielsweise zum Auffinden von Objekten oder zur Synchronisation von Nebenläufigkeit. Ziel der unterschiedlichen Programmierparadigmen ist vor allem die Modellierung kooperierender Objektsysteme. Darauf aufbauend dient die Validierung von Objektsystemen dazu, Ablauffehler während der Entwickung zu vermeiden oder frühzeitig während des Ablaufs zu erkennen. In Kapitel 5 wird gezeigt, wie linguistische Reflektion [Sheard 90; Stemple et al. 92a; Chiba 95; Masuhara et al. 95] (vgl. dazu auch den Begriff der Entwicklungsobjekte in Kapitel 2) zur statischen (zur Entwicklungszeit angewendeten) und dynamischen (zur Laufzeit ausgewerteten) Validierung von kooperierenden Objektsystemen eingesetzt wird.

Programmierparadigmen für kooperierende Objektsysteme können im Stile von Entwurfsmustern [Gamma et al. 95] auf der Basis einer ausdrucksmächtigen Kernsprache [Matthes, Schmidt 91a; Schmidt, Matthes 93] realisiert werden. Dieser Ansatz bietet den Vorteil, daß die Kernsprache einfach gehalten und eine höhere Interoperabilität erzielt wird, ohne auf mächtige Abstraktionen verzichten zu müssen. Dazu dienen traditionell in der Sprache entwickelte und wiederverwendete Bibliotheken [Liskov et al. 77; Lauer, Satterthwaite 79; Wirth 79] (vgl. dazu Standardobjekte in Kapitel 2). Ergänzend erweitern statische Generatoren (Syntaxerweiterungen) und dynamische Generatoren (Funktionen höherer Ordnung) die Sprache um neue Konstrukte, die in die Kernsprache abgebildet werden [Cheatham 66; Kohlbecker et al. 86; Schröder 93; Cardelli et al. 94] (vgl. Kapitel 5).

1.4 Zielsetzung und Ergebnisüberblick

Der Betrieb[4] und die Entwicklung[5] kooperierender Objektsysteme stellen in methodischer wie technischer Hinsicht hohe Anforderungen an eng aufeinander abgestimmte, durchgängig anwendbare Modelle, Methoden und Technologien, welche innerhalb der Informatik bislang in weitgehend separaten Entwicklungslinien erforscht und realisiert worden sind. Diese Arbeit verfolgt deshalb sowohl integrierende wie auch abstrahierende Ziele mit folgenden Schwerpunkten:

1. Aufstellung eines einheitlichen Rahmens zur Beschreibung kooperierender Objektsysteme.

[4] Der Begriff "Betrieb" umfaßt in dieser Arbeit die gesamte, aufgrund der Persistenz potentiell unendliche Lebensdauer eines Objektsystems (vgl. Kapitel 3).

[5] Der Begriff "Entwicklung" erfaßt in dieser Arbeit sowohl die initiale Erstellung (vgl. Kapitel 5) wie auch die langfristige Evolution (vgl. Abschnitt 3.3) von Objektsystemen.

2. Systematische Ableitung der beim Betrieb und bei der Entwicklung kooperierender Objektsysteme zu lösenden Aufgaben aus diesem Rahmen.

3. Entwicklung von Lösungsbeiträgen für diese Aufgaben.

Bei der systematischen Ausfüllung des vorgelegten Lösungsrahmens werden einerseits neuartige Abstraktionen über bekannte und erprobte Informatikfunktionalität gelegt (beispielsweise wird die Reflektion unter Kooperation neu gefaßt), andererseits werden aber auch Ergänzungen an der Basisfunktionalität für kooperierende Objektsysteme vorgenommen (beispielsweise ein Mechanismus zur flexiblen, autonomiewahrenden Wiederherstellung von Bindungen bei der Übertragung von Objekten). In jedem Fall wird dabei das Ziel verfolgt, auf dem jeweils neuesten Stand der Technik der betroffenen Informatikbereiche aufzusetzen. Dies betrifft inbesondere

- die Evolution persistenter Objektsysteme,
- die Behandlung entfernter und wiederherzustellender Bindungen bei der Kooperation zwischen autonomen Objektsystemen in heterogenen Umgebungen sowie
- die Validierung der Kooperation und die Generierung von Kooperationsmustern zur Entwicklung von kooperierenden Objektsystemen.

Bei der notwendigen Schwerpunktbildung steht die Integrität und Ablauffähigkeit kooperierender Objektsysteme im Mittelpunkt.

Als Grundlage für den Betrieb und die Entwicklung kooperierender Objektsysteme wird in dieser Arbeit ein Objektsystemmodell entwickelt, das die essentiellen Aspekte von Objektsystemen erfaßt. Objektsysteme setzen sich aus Objekten und Bindungen zwischen Objekten zusammen, wobei die Bindungen besondere Bedeutung für die Persistenz, Evolution und Kooperation besitzen. Die Behandlung von Bindungen bildet daher einen Schwerpunkt dieser Arbeit. Objekte werden entlang ihres Verhaltensaspekts als passiv (Daten), aktivierbar (Code) oder aktiv (Prozesse) eingestuft. Die Repräsentationen von Objekten und Bindungen bilden die Basis für den Ablauf, die Persistenz, die Kooperation und die Entwicklung von Objektsystemen. Ablaufsysteme dienen zum Betrieb von Objektsystemen und verleihen ihnen somit eine Ablaufsemantik. Objektsysteme werden mit Hilfe von Entwicklungssystemen erstellt. Zur Realisierung von kooperierenden Objektsystemen ist eine Kooperation der Objektsysteme mit ihren Ablaufsystemen (während des Ablaufs) und ihren Entwicklungssystemen (während der Entwicklung) notwendig, zusammengefaßt unter den Begriffen Verhaltens- und linguistische Reflektion. Ablauf- und Entwicklungssysteme werden daher wiederum als Objektsysteme modelliert, was eine radikale Vereinfachung bedeutet, da nun Reflektion als Spezialfall der Kooperation konzeptuell erfaßt wird (Kapitel 2).

Der Betrieb von Objektsystemen geht vom Ablauf eines Objektsystems in einer Ablaufumgebung aus. Im Mittelpunkt steht dabei die dynamische Belegung und Freigabe von Ressourcen während des Ablaufs, wodurch die Ablauffähigkeit des Objektsystems bestimmt wird. Zur Sicherung der Ablauffähigkeit werden reflektive Techniken herangezogen. Die Forderung nach der Erweiterung der Lebensdauer von Objekten über einen Ablauf hinaus führt zu persistenten Objektsystemen; hierbei wird die persistente Sicherung eines Ablaufzustandes und seine konsistente Wiederherstellung betrachtet. Durch die Langlebigkeit werden Anpassungen von Objektsystemen während ihres Ablaufs notwendig, um veränderten Anforderungen oder Umgebungsänderungen gerecht zu werden. Bei dieser Evolution ist insbesondere zu beachten, daß die Integrität und damit Ablauffähigkeit des Objektsystems gewahrt wird. In dieser Arbeit werden daher ausführlich Evolutionsmechanismen unter dem Aspekt der Sicherung der Ablauffähigkeit diskutiert (Kapitel 3).

Die Kooperation zwischen autonomen Objektsystemen in heterogenen Umgebungen wird aus dem Objektsystemmodell abgeleitet. Grundlegende Kooperationsmechanismen sind die Übertragung und Bindung von Objekten zwischen Objektsystemen. Aufgrund der Autonomie und Heterogenität der Kooperationspartner sind bei der Übertragung insbesondere die Portierung der Repräsentationen und die Erhaltung bzw. Wiederherstellung der Bindungen zu beachten. In dieser Arbeit wird ein flexibler Mechanismus zur Wiederherstellung von Bindungen unter Beachtung

der Autonomie der Kooperationspartner eingeführt. Bindungen zwischen Objektsystemen greifen langfristig (insbesondere bei persistenten Objektsystemen) in die Autonomie der Objektsysteme während des Ablaufs ein. Daher erfordern diese Bindungen eine kontextabhängige Abstimmung der Kooperationsstrategien der kooperierenden Objektsysteme. In dieser Arbeit werden unterschiedliche Strategien diskutiert und insbesondere eine Strategie eingeführt, welche einen Ausgleich zwischen den Autonomiebestrebungen beider Kooperationspartner schafft (Kapitel 4).

Während der Entwicklung von kooperierenden Objektsystemen steht die Sicherung der Ablauffähigkeit durch Validierung und Generierung im Vordergrund. Dabei werden die Entwicklungszeit (statische Validierung und Generierung) und Ablaufzeit (dynamische Validierung und Generierung) betrachtet. Ausgangspunkt bildet die Modellierung der Objektsysteme in einer anwendungsgerechten Repräsentation. Aus dieser anwendungsgerechten Repräsentation wird durch ein Entwicklungssystem eine ablaufgerechte Repräsentation erzeugt, die von einem Ablaufsystem interpretiert wird. Um Ablauffehler während einer Kooperation möglichst auszuschließen oder wenigstens frühzeitig zu erkennen, werden statische und dynamische Validierung eingesetzt. In dieser Arbeit wird gezeigt, wie die dynamische Validierung durch Kooperation des Objektsystems mit dem Entwicklungssystem erreicht wird. Wechselwirkungen mit der Evolution des Entwicklungssystems werden aufgezeigt. Um während der Entwicklung neue Kooperationsmuster zu unterstützen und somit die Gefahr von Entwicklungsfehlern zu verringern, wird statische und dynamische Generierung eingesetzt. In dieser Arbeit wird gezeigt, wie das Entwicklungssystem um statische Generierung erweitert wird und somit flexibel und sicher neuartige Kooperationsmuster unterstützt (Kapitel 5).

Kapitel 2

Objektsysteme

In diesem Kapitel wird ein Modell für Objektsysteme eingeführt. Anhand dieses Modells wird in den folgenden Kapiteln diskutiert,

- wie Objektsysteme betrieben werden, insbesondere in Bezug auf Persistenz und Evolution,
- wie Objektsysteme in heterogenen Umgebungen kooperieren und
- mit welchen Mitteln kooperierende Objektsysteme entwickelt werden.

In dieser Arbeit wird unter dem Begriff "Objektsystem" Anwendungs- und Betriebsssoftware bis hin zum Mikrocode eines Prozessors zusammengefaßt, analog zu dem unifizierenden Ansatz von [Lampson 94] für die Hardwarestruktur vom Prozessor bis zum weltweiten Netzwerk. Auf einem Rechner können prinzipiell mehrere Objektsysteme parallel, ineinander verschachtelt oder hintereinander ablaufen und dabei miteinander kooperieren; weiterhin können auch auf verschiedenen Rechnern ablaufende Objektsysteme miteinander kooperieren, wenn eine Verbindung zwischen diesen Rechnern besteht. Der Betrieb von Objektsystemen wird in Kapitel 3, die Kooperation zwischen Objektsystemen in Kapitel 4 betrachtet.

Der Begriff "Objekt" ist in dieser Arbeit nicht im Sinne von "objektorientierten" Systemen (vgl. beispielsweise [Nygaard, Dahl 81; Goldberg, Robson 85; Meyer 88]) zu verstehen, sondern bezeichnet eine bindbare Einheit eines Objektsystems. Objekte werden entlang ihres Verhaltens in Daten, Funktionen und Prozesse unterschieden und in diesen Kategorien insbesondere unter ihrem Bindungsaspekt diskutiert. In objektorientierten Systemen sind beispielsweise Daten und Funktionen (Methoden) bindbare Einheiten, ohne (im Sinne der Objektorientierung) Objekte zu sein, während in diesem Modell Daten und Funktionen als Objekte angesehen werden; Prozesse werden in objektorientierten Modellen nur implizit betrachtet, während sie hier gleichberechtigt neben Daten und Funktionen repräsentiert werden. Bindungen zwischen Objekten werden explizit eingeführt und behandelt, was in objektorientierten Modellen selten der Fall ist [Shapiro 94]. In verteilten objektorientierten Modellen finden sich daher Hilfskonstrukte, welche diese Einheiten verfügbar machen, ohne sie in das Systemmodell zu integrieren (vgl. beispielsweise [Jul et al. 88; Bennett 90; Cardelli 95; Geib et al. 96]).

Im Mittelpunkt des Modells stehen aus Objekten bestehende *Objektsysteme*, welche ablauffähige Anwendungen realisieren, die ein Problem der Anwendungswelt lösen. Objektsysteme laufen im Kontext von *Ablaufumgebungen* ab, welche die Ablaufsemantik der Objektsysteme bestimmen. Objektsysteme werden mit Hilfe von *Entwicklungsumgebungen* entwickelt; Entwicklungsumgebungen erzeugen ablauffähige Objektsysteme für bestimmte Ablaufumgebungen.

Ablauf- und Entwicklungsumgebungen strukturieren sich in *Ablauf-* und *Entwicklungssysteme*, die wiederum eigenständig ablauffähige Objektsysteme darstellen, sowie weitere Objekte (Standardobjekte, Entwicklungsobjekte, Basisobjekte, externe Objekte).

Zentraler Bestandteil des Ablaufsystems ist der *Interpreter*, der Objektsysteme interpretiert und ihnen somit eine Basissemantik verleiht, die durch Basis- und externe Objekte ergänzt wird.

Externe Objekte sind Objekte, die eine andere Ablaufumgebung als das Objektsystem besitzen und somit nur durch Kooperation erreichbar sind. *Basisobjekte* sind Objekte des Ablaufsystems, welche die Ablaufeigenschaften des Objektsystems beeinflussen.

Entwicklungssysteme werden durch *Übersetzer* geprägt, die Objektsysteme erzeugen sowie bei der Erzeugung Standard- und Entwicklungsobjekte in die Objektsysteme integrieren. *Standardobjekte* sind Objekte, die in verschiedene Objektsysteme integriert werden können. *Entwicklungsobjekte* sind Objekte des Entwicklungssystems, die Meta-Wissen über die Erzeugung des Objektsystems enthalten, so daß das Objektsystem mit ihrer Hilfe über sich selbst Aussagen machen oder während des Ablaufs neue Objekte entwickeln kann, deren Art zur Entwicklungszeit noch nicht bekannt ist.

Die Basis für Objektsysteme, Ablauf- und Entwicklungsumgebungen bilden *Objekte*. Objekte werden entlang ihres Bindungsaspekts grundsätzlich unterschieden in *aggregierte Objekte*, die andere Objekte binden können, und *Basisdaten*, die nur gebunden werden. Aggregierte Objekte werden weiterhin entlang ihres Verhaltensaspekts unterschieden in *Daten* (passive Objekte), *Funktionen* (aktivierbare Objekte) und *Prozesse* (aktivierte Objekte). Zwischen Objekten können *Bindungen* bestehen, wobei ein Objekt jeweils ein (nicht notwendigerweise anderes) Objekt bindet.

Objekte und Bindungen liegen in *Repräsentationen* vor, die durch Interpreter interpretiert werden und somit den Ablauf der Objektsysteme steuern. Sie bilden die Basis für die Persistenz und die Übertragung von Objekten und Bindungen. Die Repräsentationen werden von Übersetzern sowohl verarbeitet als auch erzeugt und umgewandelt.

2.1 Objekt

Objekte sind grundlegende informationstragende Einheiten. Objektsysteme, Ablauf- und Entwicklungsumgebungen sind aus Objekten zusammengesetzt.

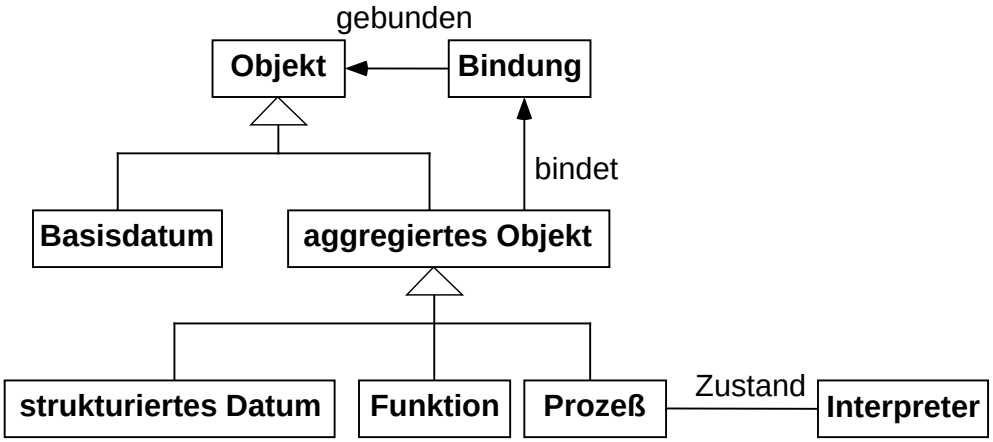

Abbildung 2.1: Objekte und Bindungen

Objekte werden anhand ihrer Rolle in Bindungen unterschieden (vgl. Abbildung 2.1) in aggregierte Objekte, die andere Objekte binden können, und Basisdaten, die nur gebunden werden. Bindungen zwischen Objekten werden in Abschnitt 2.2 behandelt. Entlang ihres Verhaltens werden Objekte unterschieden in (Basis- und strukturierte) Daten (passive Objekte, vgl. Abschnitt 2.1.1), Funktionen (aktivierbare Objekte, vgl. Abschnitt 2.1.2) und Prozesse (aktivierte Objekte, vgl. Abschnitt 2.1.3).

2.1.1 Datum

Daten repräsentieren Informationen. Sie sind passive Objekte, d.h. sie werden durch aktivierbare Objekte (Funktionen) erzeugt, verarbeitet und verändert. Ihr Informationsgehalt erschließt sich durch Funktionen (vgl. Abschnitt 2.1.2), die auf ihnen definiert sind, die Daten verarbeiten und damit interpretieren.

Daten werden anhand ihrer Bindungseigenschaften aufgeteilt in Basisdaten und strukturierte Daten.

Basisdaten (vgl. Abbildung 2.2) stellen einen (Informations-)Wert an sich da. Sie können keine anderen Objekte binden, sondern nur gebunden werden.

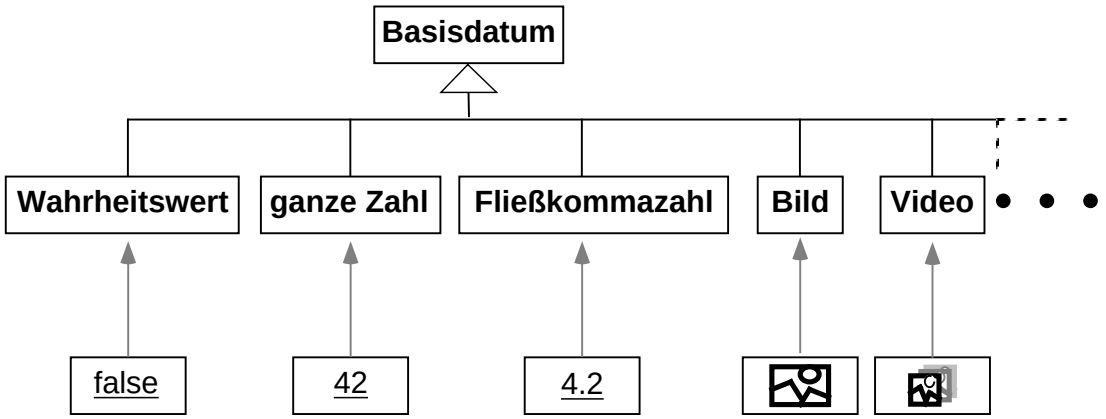

Abbildung 2.2: Basisdaten

Beispielsweise (vgl. Tabelle 2.1) sind auf Wahrheitswerten (wahr und falsch) u.a. die Negation und die UND-Verknüpfung definiert, welche wiederum Wahrheitswerte erzeugen. Diese Beispiele

Datentyp	Operationen
Wahrheitswerte	Negation, UND-Verknüpfung
ganze Zahlen	Absolutwert, Divison
Fließkommazahlen	Runden, Cosinus
Bilder	Anzeigen, Ausschnitt
Videos	Abspielen, Standbild

Tabelle 2.1: Beispiele für Operationen auf Basisdaten

zeigen, daß

- Basisdaten sowohl klein (ein Bit bei Wahrheitswerten) als auch groß (typischerweise Mega- oder Gigabytes bei Videos) sein können,
- Operationen auf den Daten oft neue Daten, auch eines anderen Datentyps, erzeugen (wie die Negation, der Ausschnitt eines Bildes oder das Standbild aus einem Video) und
- Operationen abhängig von dem Inhalt (Informationswert) des Objekts Seiteneffekte auslösen können (das Anzeigen eines Bildes oder Abspielen eines Videos, aber auch Fehler wie eine Divison durch Null).

Im Gegensatz zu Basisdaten bestehen *strukturierte Daten* (vgl. Abbildung 2.3) wiederum aus anderen Objekten, indem sie diese Objekte binden. Sie legen damit eine Struktur über die Objekte. Beispielsweise werden ein Straßenname, eine Hausnummer, eine Postleitzahl und ein Ortsname zu

einem (Adreß-)Tupel aggregiert oder mehrere Personen (repräsentiert durch die sie beschreibenden Daten) zu einer Menge zusammengefaßt. Diese Stukturen (Tupel, Menge usw.) sind generisch, d.h. sie strukturieren (prinzipiell) beliebige Objekte, auf die durch generische, d.h. datenunabhängige, Funktionen zugegriffen wird. Strukturierte Daten können nicht nur andere (Basis- oder strukturierten) Daten, sondern auch Funktionen und Prozesse binden.

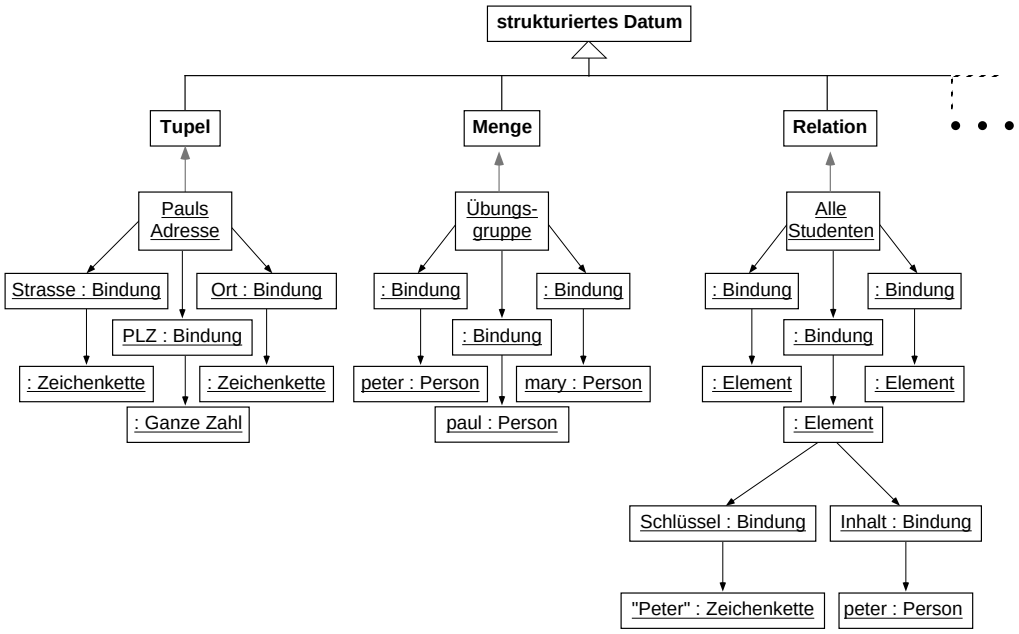

Abbildung 2.3: Strukturierte Daten

Beispielsweise (vgl. Tabelle 2.2) sind auf Tupeln die Aggregation von (beliebigen) Objekten und der Zugriff über einen Feldnamen auf ein gebundenes Objekt als Operationen definiert. Diese

Datentyp	Operationen
Tupel	Aggregieren; Zugriff über einen Feldnamen auf ein Objekt
Menge	Einfügen; Abfrage, ob ein Objekt sich in der Menge befindet
Relation	Zugriff über einen Schlüssel auf ein Objekt

Tabelle 2.2: Beispiele für Operationen auf strukturierten Daten

Beispiele zeigen, daß

- strukturierte Daten eine bei ihrer Erzeugung festgelegte Größe (wie Tupel) oder eine prinzipiell unendliche Größe (wie Relationen) besitzen können,
- die Datenstrukturen sowohl Objekte unterschiedlichen Typs (wie Tupel) oder auch Objekte eines Typs (wie Relationen) enthalten können und
- die Funktionen auf den strukturierten Daten nicht von den Elementen abhängen, die sie enthalten, sondern an die Datenstruktur gekoppelt sind.

Daten können voneinander abhängen, ohne daß sich dies in Bindungen ausdrückt. Beispiele sind ein Index in ein Feld (*array*) oder ein Schlüssel für eine Relation: Sowohl der Index als auch der Schlüssel sind eigenständige Daten, die keine Bindung an das Feld oder die Relation besitzen;

umgekehrt sind auch das Feld und die Relation nicht an den Index oder den Schlüssel gebunden. Trotzdem wird über den Index bzw. den Schlüssel als Parameter von Zugriffsfunktionen auf Elemente des Feldes bzw. der Relation zugegriffen. Also besteht eine implizite Abhängigkeit zwischen den Daten. Ein anderes Beispiel ist eine beschreibende Information wie die Größe eines Bildes, die unabhängig von dem Bild existiert, aber von ihm abgeleitet ist. Es handelt sich hier um semantische, also wertabhängige, Bezüge.

2.1.2 Funktion

Funktionen sind aktivierbare Objekte, d.h. sie beschreiben einen Ablauf oder eine Berechnung.

Das folgende Beispiel zeigt eine *Ablaufbeschreibung*:

```
let f() :Ok =
begin
  let var x = readInteger()
  if x = 0 then writeChar('0') else
    while x >= 0 do
      writeChar('I') end
      x := x-1
    end
  end
end
```

Abläufe bestehen beispielsweise aus Sequenzen (*begin* ... *end*), Schleifen (*while* ... *do* ... *end*) und Verzweigungen (*if* ... *then* ... *end*). Sie können andere Funktionen, neue Datenobjekte erzeugen (*let var x* = ...) oder (gebundene) Datenobjekte manipulieren ($x :=$ ...). Über Seiteneffekte ändern Funktionen ihre Umgebung und werden von ihr beeinflußt:

steuernde Seiteneffekte (Aktoren): Befehle des Interpreters, die Seiteneffekte auf die Umgebung haben (*writeChar*)

aufnehmende Seiteneffekte (Sensoren): Befehle des Interpreters, deren Ausführung vom Umgebungszustand abhängt (*readInt*)

Das folgende Beispiel zeigt eine seiteneffektfreie *Berechnung*:

```
let f() :Real =
  1.0 / 3.0
```

Berechnungen erzeugen ein Ergebnis, das zurückgegeben wird, ohne Seiteneffekte. Mischformen aus Abläufen und Berechnungen sind möglich.

Eine Funktion kann andere Objekte binden (vgl. Abbildung 2.4):

```
fun()
begin
  let x = calcInt()
  if x == y then
    suspend p
  end
end
```

Es können Funktionen (*readInteger*), Daten (y) und Prozesse (p) gebunden werden. Diese Bindungen sind *statisch*, weil sie unabhängig vom Kontext der Ausführung der Funktion sind. Eine Funktion mit ihren statischen Bindungen wird auch *Funktionsabschluß* (*closure*) genannt [Kaplan, Wileden 95b].

Die Ausführung einer Funktion (vgl. Abbildung 2.5) wird durch einen Interpreter vorgenommen, der die Repräsentation der Funktion als Befehle interpretiert. Der Zustand des Interpreters

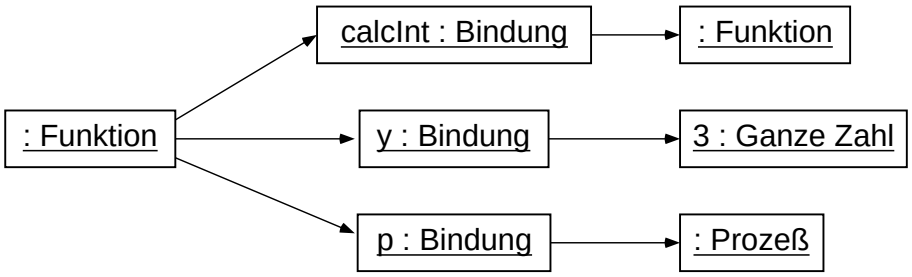

Abbildung 2.4: Statische Bindungen in einer Funktion

wird durch einen Prozeß repräsentiert. Die Ausführung einer Funktion findet daher immer im Kontext eines Prozesses statt. Eine Funktion kann im Kontext verschiedener Prozesse ausgeführt werden; für jede Ausführung gibt es aber genau einen Prozeß, in dessen Kontext diese Ausführung stattfindet.

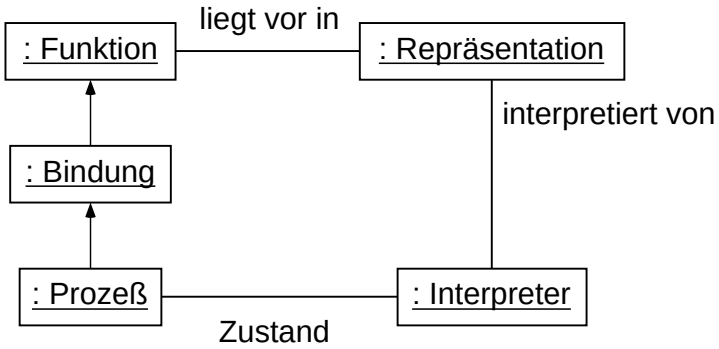

Abbildung 2.5: Ausführung einer Funktion

Bei der Ausführung kann eine Funktion parametrisiert werden (vgl. Abbildung 2.6):

```
let f(i :Int) :Int = i * i
let y = 3
(* -- *)
f(y)
```

Die Funktion nutzt über die Parameter Bindungen des Prozeßobjekts, das die Ausführung anstößt, während ihres Ablaufs. Diese Bindungen werden *dynamisch* genannt, weil sie vom Ausführungskontext abhängen.

Aus der parametrisierten Funktion *f* wird (temporär) beim Ablauf ein (unbenannter) Funktionsabschluß erzeugt (vgl. Abbildung 2.6), der eine statische Bindung an den Parameterwert *3* enthält. Dies erklärt die Erzeugung eines Funktionsabschlusses: dynamische Bindungen werden sukzessive zu statischen Bindungen umgeformt. Bei Funktionen höherer Ordnung wird dieser Funktionsabschluß explizit benannt, beispielsweise bei Funktionen, die Funktionen als Rückgabe zurückliefern (vgl. Abbildung 2.7):

```
let add(x :Int) =
  fun(y :Int) :Int
    x + y
(* -- *)
```

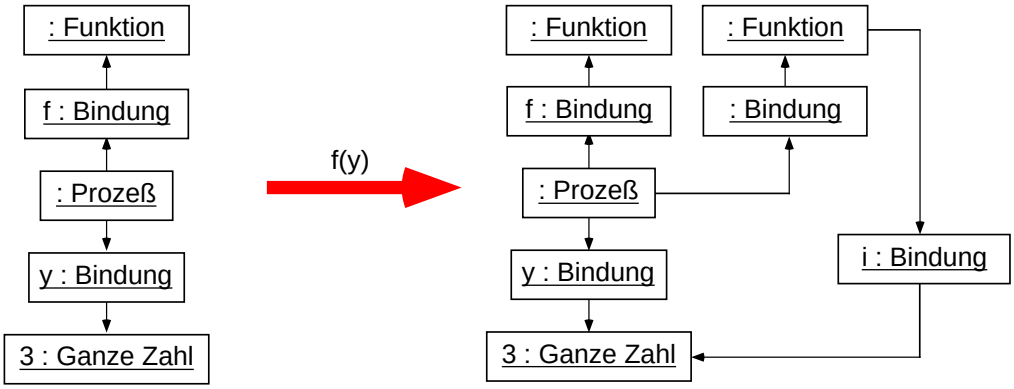

Abbildung 2.6: Dynamische Bindungen in einer Funktion

```
let add3 = add(3)
```

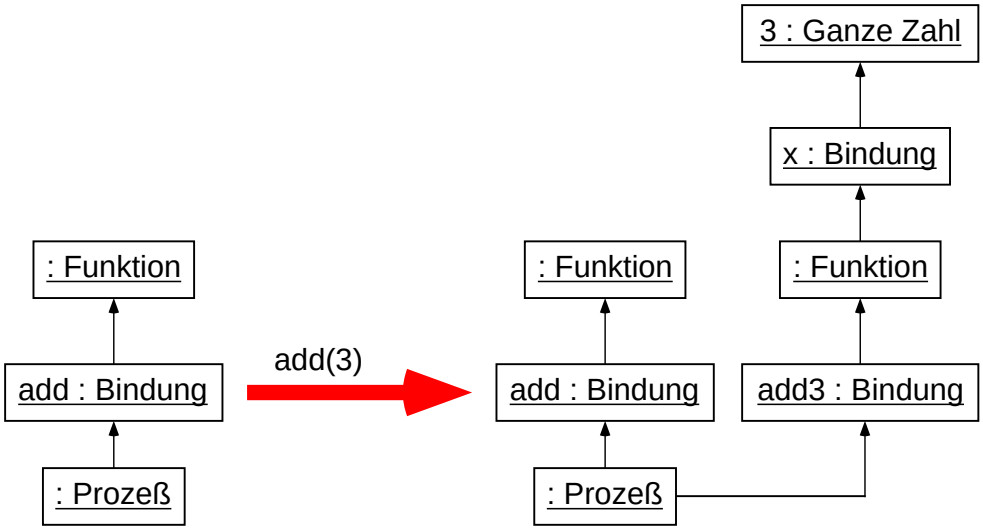

Abbildung 2.7: Funktionen höherer Ordnung

Die Funktion *add3* wird erzeugt, indem die Funktion *add* mit der Zahl *3* parametrisiert wird.

Konzeptuell ist jede Funktion eine Funktion höherer Ordnung, deren dynamische Bindungen zu einem (Binde-)Zeitpunkt im Kontext eines Prozesses zu statischen Bindungen umgewandelt werden, so daß ein (neuer) Funktionsabschluß entsteht [Koch 97] (vgl. Abbildung 2.8):

```
let f_link(readInteger :Fun() :Int  y :Int  p :Process) =
  fun() :Ok
  begin
    let x = readInteger()
    if x == y then
      suspend p
    end
  end
let readInteger' = ...
```

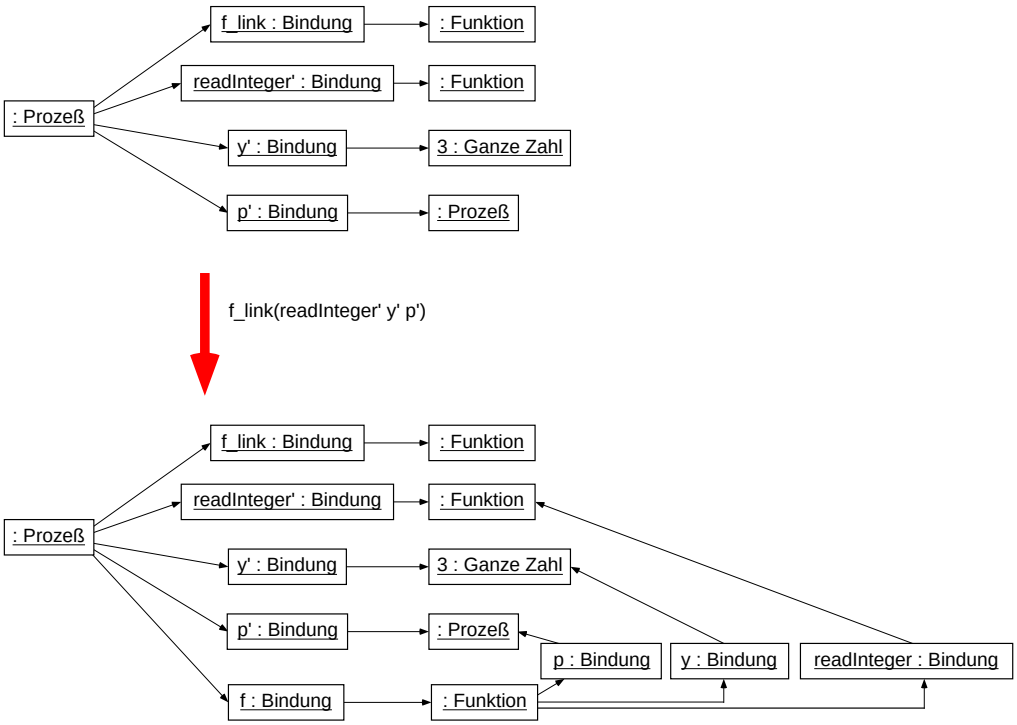

Abbildung 2.8: Funktionsabschluß

```
let y' = ...
let p' = ...
(* -- *)
let f = f_link(readInteger' y' p')
```

Die Funktion *f_link* ist eine Funktion, die keine statischen Bindungen enthält, sondern nur über ihre Parameter dynamische Bindungen. Sie wird im Kontext eines Prozesses aufgerufen, wobei die dynamischen Bindungen durch statische Bindungen ersetzt werden und so der Funktionsabschluß *f* entsteht. Die Funktion *f* kann jetzt wiederum unabhängig von dem sie erzeugenden Prozeß weiterverwendet werden, insbesondere im Kontext anderer Prozesse.

Eine Funktion kann innerhalb eines oder mehrerer Prozesse mehrmals (gleichzeitig und hintereinander) aktiv sein. Das dynamische Binden der Parameter und das Anlegen lokaler Objekte erfolgt über den Prozeß, der die Funktion aktiviert. Bei jedem Aufruf wird ein Funktionsabschluß angelegt, der die Parameter als statische Bindungen enthält. Die gebundenen Objekte werden nicht kopiert, damit Seiteneffekte auf gemeinsam benutzten Objekten (*count*) für alle Funktionen (*f*, *g*) sichtbar sind (vgl. Abbildung 2.9):

```
let link_f(var count :Int) =
  fun() count := count+1
let link_g(var count :Int) =
  fun() count := count-1
let var count' = 0
(* -- *)
let f = link_f(&count')
let g = link_g(&count')
```

Daten werden durch Funktionen, die auf ihnen arbeiten, interpretiert. Funktionen erzeugen und ändern Daten, sie werten Daten aus, binden sie oder lösen die Bindungen wieder auf. Der Ablauf

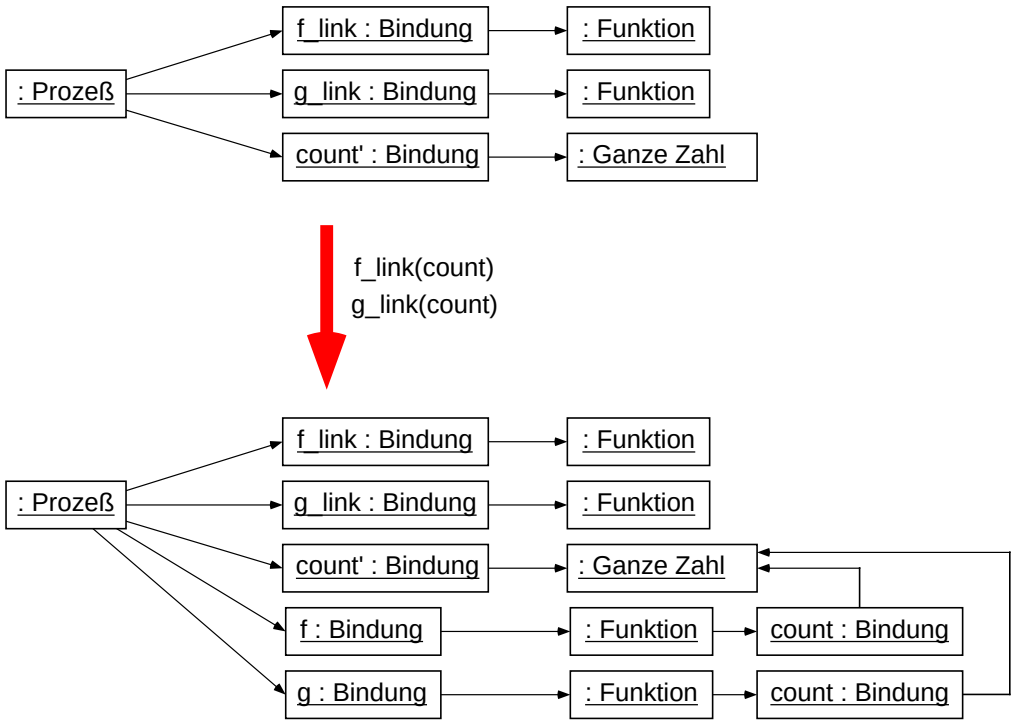

Abbildung 2.9: Gemeinsam durch mehrere Funktionen genutzte Objekte

von Funktionen wird von Daten gesteuert: sie dienen als Parameter, als Zwischenresultate und als Ergebnis eines Funktionsablaufs.

Die Erzeugung von Funktionen ist funktionsgetrieben, d.h. eine Funktion (ein Übersetzer oder Generator) erzeugt den (interpreterabhängigen) Programmcode einer Funktion als Daten. Diese Daten sind von vornherein nicht von "normalen" Daten zu unterscheiden. Erst die Interpretation durch einen Interpreter im Rahmen eines Prozesses gibt diesen Daten die Semantik einer Funktion. Funktionen haben also die besondere Eigenschaft, nicht nur Daten, sondern auch selbst wieder Funktionen erzeugen zu können. Sie können sogar den eigenen Programmcode als Daten betrachten und somit manipulieren, d.h. zur Laufzeit ändern. Dies wird als selbstmodifizierende Funktion bezeichnet und stellt einen Sonderfall der linguistischen Reflektion dar [Sheard 90; Stemple et al. 92b; Geisler 95]. Im folgenden Beispiel wird durch Übersetzung einer Zeichenkette s zur Laufzeit eine Funktion erzeugt, die unter dem Bezeichner f im folgenden (statischen) Programmtext aufgerufen wird.

```
let s = ``fun(x :Int  y :Int) :Int  x+y''
let f = compile(s)
f(1 2)
```

Funktionen werden durch Interpreter ausgeführt. Der Zustand des Interpreters wird als Prozeß repräsentiert. Funktionen können neue Prozesse erzeugen und durch Operationen manipulieren, z.B. suspendieren oder abbrechen. Außerdem kann dann eine Funktion feststellen, in welchem Prozeß sie ausgeführt wird, und somit reflektiv ihre Ausführung beeinflussen, z.B. wie im folgenden Beispiel ihren Ablauf suspendieren.

```
let f() :Ok =
begin
  let p = process_self()
  suspend(p)
```

```
  ...
end
```

2.1.3 Prozeß

Ein Prozeßobjekt enthält die Zustandsbeschreibung des Interpreters für einen Ablauf eines Funktionsobjekts (bzw. mehrerer Funktionsobjekte bei Aufrufen von Subfunktionen in einer Funktion) und den Kontext (Bindungen), in dem diese Funktion abläuft. Es steuert den Ablauf des Funktionsobjekts, insbesondere erzeugt es Funktionsabschlüsse und stellt so die dynamischen Bindungen der Parameter an Objekte her.

Auf Prozessen sind u.a. folgende Operationen definiert:

Erzeugen: Ein Prozeß wird erzeugt, indem ein Interpretationskontext eines Interpreters angelegt wird. Initial benötigt dieser Kontext eine (parameterlose) Funktion, die innerhalb dieses Prozesses vom Interpreter ausgeführt werden soll.

Suspendieren: Die Ausführung der Funktion wird unterbrochen, aber der Prozeßzustand bleibt erhalten.

Aktivieren: Die Ausführung eines suspendierten Prozesses wird an der Stelle fortgesetzt, an der er suspendiert wurde.

Abbrechen: Die Ausführung des Prozesses wird abgebrochen, ohne die Funktion, die der Prozeß gerade ausführt, zu beenden; der Interpretationskontext wird ungültig.

Die Beschreibung des Ablaufzustands hängt von einem Interpreter ab, der die Funktion(en) interpretiert (vgl. Abbildung 2.10):

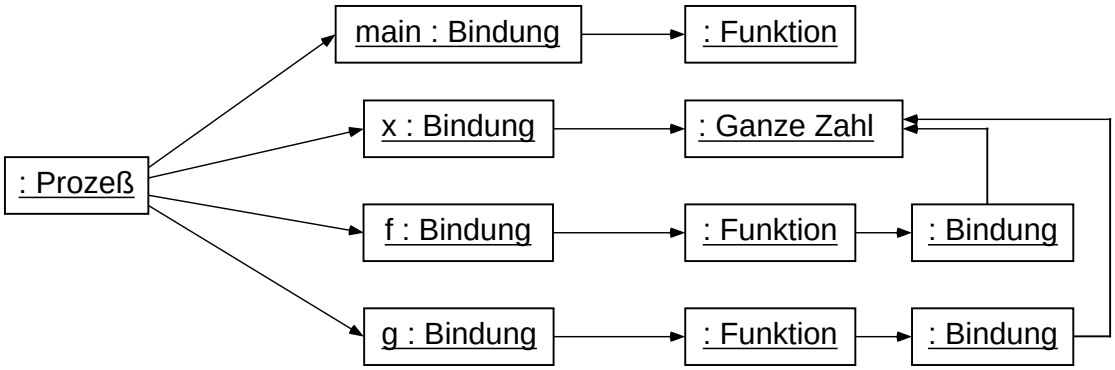

Abbildung 2.10: Prozeß

```
newProcess(
let main() :Ok =
begin
  let var x = 10
  let f() :Ok =
    begin
      x := x+1
    end
  let g() :Ok =
    begin
      f() (* 2 *)
```

```
      f() (* 3 *)
    end
  g() (* 1 *)
  f()
end)
```

Ein neuer Prozeß wird mit einer Funktion *main* initialisiert, die er ausführt, d.h. es wird eine Bindung zwischen der Funktion *main* und dem Prozeß aufgebaut. Der Prozeß beginnt mit der Ausführung der Funktion, indem die einzelnen Anweisungen der Funktion vom Interpreter interpretiert werden. Die Anweisungen führen dazu, daß beispielsweise neue Objekte und Bindungen angelegt (*let var x* = ...), Funktionsabschlüsse durchgeführt (*let f() :Ok* = ...) und Funktionen aufgerufen (*g()*) werden. Ein Prozeß wird beendet, wenn die Funktion abgearbeitet ist, mit der der Prozeß initialisiert wurde, in diesem Fall also *main*.

Im Prozeß ist bei der Ausführung die Information gekapselt, welche Anweisung der gerade ausgeführten Funktion als nächstes zu interpretieren ist (*instruction pointer*) und mit welcher Anweisung nach dem Aufruf einer Unterfunktion fortgesetzt wird (*call stack*). Im Beispiel wird die Funktion *g* von der Funktion *main* aufgerufen und ruft ihrerseits zweimal die Funktion *f* auf. Der dynamische Ablauf sieht also (ausschnittsweise) folgendermaßen aus:

1. Ein Prozeß wird mit der Funktion *main* initialisiert. Bei der Interpretation der Funktion *main* werden ein Datenobjekt, initialialisiert mit *10* und gebunden als *x*, sowie zwei Funktionsabschlüsse, gebunden als *f* und *g*, angelegt.

2. In *main* wird die Funktion *g* aufgerufen und vermerkt, daß die Auswertung der Funktion *main* an der Stelle *(* 1 *)* fortzusetzen ist.

3. In *g* wird die Funktion *f* aufgerufen und vermerkt, daß die Auswertung der Funktion *g* an der Stelle *(* 2 *)* fortzusetzen ist.

4. *f* wird abgearbeitet und an die Stelle *(* 2 *)* in *g* zurückgekehrt.

5. In *g* wird die Funktion *f* aufgerufen und vermerkt, daß die Auswertung der Funktion *g* an der Stelle *(* 3 *)* fortzusetzen ist.

6. *f* wird abgearbeitet und an die Stelle *(* 3 *)* in *g* zurückgekehrt.

7. Die Abarbeitung von *g* ist beendet, also wird an die Stelle *(* 1 *)* in *main* zurückgekehrt.

An diesem Beispiel ist zu sehen, daß ein Prozeß

- eine sich dynamisch ändernde Menge von Bindungen an Objekte beinhaltet;
- in seinem Zustand Bezug auf die interne Struktur der interpretierten Funktionsobjekte, d.h. die in ihnen enthaltenen Anweisungen, nimmt;
- zu einem Zeitpunkt mehrere (geschachtelte) Funktionen im Zustand der Ausführung sein können, wobei eine Funktion aktuell interpretiert wird, während die Interpretation der anderen Funktionen bei einer bestimmten Anweisung fortgesetzt wird, wenn die geschachtelten Funktionsaufrufe abgearbeitet sind.

Wenn mehrere Funktionen nicht geschachtelt, sondern gleichzeitig (parallel) ausgeführt werden sollen, werden mehrere Prozesse angelegt, die diese Funktionen ausführen. Je nach Art der Ablaufumgebung werden sie dabei echt parallel von mehreren Interpretern oder wechselweise von einem Interpreter ausgeführt; der Interpreter wechselt in diesem Fall mit dem Prozeß den Ausführungskontext. Wenn die parallel ausgeführten Prozesse gemeinsame Bindungen beinhalten, werden sie leichtgewichtige Prozesse oder Fäden (*threads*) genannt, weil ein Teil des Prozeßkontextes beim Prozeßwechsel erhalten bleibt.

2.2 Bindung

Zwischen Objekten können *Bindungen* (vgl. Abbildung 2.1 auf Seite 15) bestehen, wobei ein Objekt jeweils ein (nicht notwendigerweise anderes) Objekt bindet. Die an einer Bindung beteiligten Objekte vertreten also zwei unterschiedliche Rollen: ein Objekt bindet, das andere Objekt wird gebunden. Eine Bindung besteht immer zwischen zwei Objekten: dem bindenden und dem gebundenen Objekt. Ein Objekt kann sowohl mehrere andere Objekte binden als auch von mehreren Objekten gebunden werden. Das bindende Objekt hat einen Verweis auf das gebundene Objekt, während dem gebundenen Objekt nicht bekannt ist, durch welche Objekte es gebunden wird. Bidirektionale Bindungen werden durch zwei unidirektionale Bindungen realisiert.

Bindungen zwischen Objekten dienen zum Aufbau komplexer Objektstrukturen aus einzelnen Objekten. Bindungen können zwischen Objekten der gleichen und unterschiedlicher Objektarten bestehen (vgl. Tabelle 2.3). Ein gebundenes Objekt kann wiederum andere Objekte binden. Die

bindend	gebunden	Beispiel
Datum	Datum	Tupel, das Daten aggregiert
Datum	Funktion	Funktion in einem Tupel
Datum	Prozeß	Prozeß in einem Tupel
Funktion	Funktion	Funktionsaufruf
Funktion	Datum	Benutzung einer Variablen in einer Funktion
Funktion	Prozeß	Suspendierung eines Prozesses in einer Funktion
Prozeß	Prozeß	Kenntnis eines Prozesses über den ihn erzeugenden Prozeß
Prozeß	Datum	von einem Prozeß in seinem Ablauf erzeugte Daten
Prozeß	Funktion	Funktionsabschluß während eines Prozeßablaufs

Tabelle 2.3: Bindungsarten

geordnete Liste von Objekten, die, ausgehend von einem Objekt, durch Verfolgen von Bindungen durchlaufen wird, um zu einem bestimmten Objekt zu gelangen, wird *Pfad* genannt. Bindungen können rekursiv sein, d.h. das Objekt kann sich selbst binden oder ein Objekt binden, durch das es (transitiv) gebunden wird. D.h. ein Objekt kann sich in einem Pfad befinden, durch den es erreicht wird. Ein Objekt kann auf verschiedenen Pfaden von einem anderen Objekt erreicht werden.

Bindungen ermöglichen es, unabhängig voneinander entstandene Objekte in unterschiedliche Kontexte zu stellen (z.B. zu aggregieren oder in einem Massendatentyp zu verwalten), ohne sie jeweils zu kopieren. Dabei haben Bindungen im Gegensatz zu Objektkopien die folgenden Eigenschaften:

Sharing: Das gebundene Objekt kann an mehreren Bindungen teilnehmen und ist damit über verschiedene Pfade zu erreichen. Eine Änderung an dem Objekt, die über einen Pfad erfolgt, ist auch über die anderen Pfade sichtbar. Bei Objektkopien finden die Änderungen in den lokalen Objektkopien statt und sind somit nicht für alle bindenden Objekte sichtbar.

Identität: Ein Objekt, das über mehrere Bindungen angesprochen werden kann, behält trotzdem seine Objektidentität. Die Objektidentität ist unabhängig von den Bindungen, in denen das Objekt gebunden wird, und von den Pfaden, über die es erreichbar ist.

Bindungen werden von einem Interpreter erzeugt und manipuliert. Sie liegen wie Objekte in einer Repräsentation vor, die von einem Interpreter verstanden wird. Die Operationen auf Bindungen werden von einem Prozeß bzw. den im Kontext eines Prozesses interpretierten Funktionen gesteuert. Bindungen werden aufgebaut, abgebaut oder verfolgt, um Operationen auf den gebundenen Objekten auszuführen. Ein Interpreter kann nur Operationen auf Objekten durchführen, die in einer Repräsentation vorliegen, die der Interpreter versteht. Der Interpreter kann nur Operationen auf Objekten durchführen, die im Kontext eines Prozesses gebunden sind, denn nur diese Objekte sind für den Interpreter erreichbar.

Eine Bindung ist ein Verweis auf ein Objekt. Die Implementation dieses Verweises hängt vom Bindemechanismus des Interpreters des Laufzeitsystems ab. Die Bindung kann damit unterschiedliche Eigenschaften aufweisen:

Eindeutigkeit: Der Verweis kann eindeutig auf ein Objekt verweisen oder nur die Eigenschaften eines Objekts beschreiben, das beim Verfolgen der Bindung aus einer Menge von Objekten herausgesucht wird.

Existenzzusicherung: Es kann sichergestellt sein, daß zu einem Verweis auch wirklich ein Objekt existiert, d.h. das Verfolgen der Bindung ist immer möglich. Andererseits kann es auch sein, daß zu einem Verweis kein entsprechendes Objekt (mehr) existiert. Der Versuch, die Bindung aufzulösen, führt dann zu einem Laufzeitfehler.

Spätestens in dem Moment, in dem eine Bindung verfolgt, d.h. auf das gebundene Objekt zugegriffen, wird, muß die Bindung an ein konkretes Objekt erfolgen [Morrison et al. 90; Kaplan, Wileden 95a]. Vorher kann die Bindung dynamisch, d.h. nicht auf ein konkretes Objekt bezogen, sondern durch eine Beschreibung des gebundenen Objektes ersetzt sein. Sobald das konkrete Objekt die Beschreibung des gebundenen Objektes ersetzt, wird die Bindung statisch. Auch der umgekehrte Weg ist gangbar, d.h. ein statische Bindung kann durch eine dynamische Bindung ersetzt werden. Außerdem kann eine statische Bindung an ein Objekt durch eine statische Bindung an ein anderes Objekt sowie die dynamische Bindung mittels einer Beschreibung durch eine neue Beschreibung ersetzt werden.

2.3 Repräsentation

Alle maschinellen (digitalen) Objekte und Bindungen werden durch Bitmuster realisiert. Die Interpretation dieser Bitmuster, also der semantische Inhalt, wird mit dem Begriff Repräsentation beschrieben. Objektsysteme, d.h. Objekte und Bindungen, liegen in Repräsentationen vor (vgl.

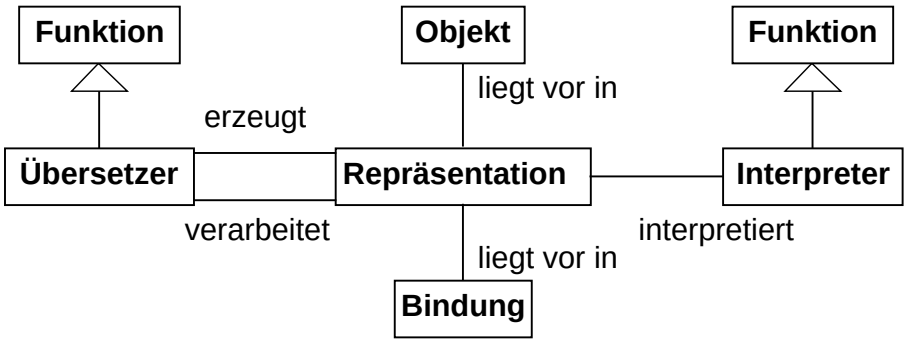

Abbildung 2.11: Repräsentation, Übersetzer und Interpreter

Abbildung 2.11). Einige Repräsentationen können durch Interpreter von Ablaufsystemen interpretiert werden, besitzen also eine Ablaufsemantik. Repräsentationen werden von Übersetzern der Entwicklungssysteme sowohl verarbeitet als auch erzeugt.

Eine Repräsentation wird (direkt) interpretierbar genannt, wenn für sie ein (maschineller) Interpreter existiert, der die Semantik der repräsentierten Objekte und Bindungen erfaßt. In diesem Sinne sind menschengerechte Repräsentationen wie Zeichenketten oder Grafiken nicht direkt interpretierbar, wenn es keinen Interpreter dafür gibt. Erst durch einen Übersetzer werden die in der menschengerechten Repräsentation vorliegenden Objekte und Bindungen umgewandelt in eine maschinengerechte Repräsentation, also eine Repräsentation, für die ein Interpreter vorliegt.

Beispiele für Repräsentationen der ganzen Zahl −1:

- Bitmuster unterschiedlicher Art (*1111 1111* oder *1000 0001*)
- Bitmuster unterschiedlicher Länge (*1000 0001*, *10000 0000 0000 0001*)
- Zeichenkette (*"–1"*, *"–01"*)
- textuelle Beschreibung (*minus eins*)
- Grafik (vgl. Abbildung 2.12)

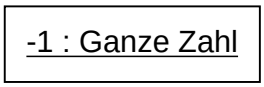

Abbildung 2.12: Grafische Repräsentation der Zahl –1

2.3.1 Interpreter

Ein Interpreter ist eine Funktion eines Ablaufsystems. Er verleiht einem Objektsystem Semantik, indem er die Objekte und Bindungen des Objektsystems interpretiert.

Grundlage des Interpretationsprozesses sind Funktionen. Ein Interpreter erhält initial eine Funktion, die er interpretiert. Eine Funktion zerlegt sich in einzelne Anweisungen an den Interpreter. Die Anweisungen umfassen:

- Kontrollflußsteuerung, z.B. Verzweigungen, Schleifen;
- Aufruf von Funktionen, mit und ohne Parametrisierung, und Rückkehr von solchen Aufrufen, mit oder ohne Rückgabe eines Resultats;
- Erzeugung und Manipulation von Objekten, also Daten, Funktionen und Prozessen;
- Erzeugung, Abbau und Verfolgen von Bindungen zwischen Objekten;
- Ausführung von Operationen auf Objekten, z.B. Addition von Zahlen, Suspendierung eines Prozesses;
- Anstoßen und Aufnehmen von Seiteneffekten, z.B. Anzeigen von Daten oder Entgegennehmen von Eingaben.

Der Zustand der Ausführung einer Funktion durch einen Interpreter wird durch folgende Angaben beschrieben:

- Welche Anweisung ist als nächste auszuführen?
- Welche Bindungen wurden im Prozeßablauf vorgenommen (und noch nicht wieder abgebaut)?
- Welche Funktionen sind aufgerufen und noch nicht beendet worden (Aufrufhierarchie)?
- Wie ist der Zustand der Basisobjekte des Ablaufsystems, die an den Interpreter gebunden sind, beispielsweise von Schaltern, die den Interpretationsvorgang beeinflussen?

Dieser Zustand wird auf der Ebene des Objektsystems durch einen Prozeß repräsentiert. Dadurch kann ein Ausführungszustand von einem Objektsystem gebunden und manipuliert werden.

Jeder Interpreter interpretiert bestimmte Repräsentationen. D.h., nicht jede maschinengerechte Repräsentation ist von allen Interpretern interpretierbar. Bitmuster, die ein in einer vom Interpreter nicht verstandenen Repräsentation vorliegen, kann der Interpreter nicht interpretieren. Er muß sie uninterpretiert lassen, also als Basisdatum ohne Operationen behandeln, oder ihre Interpretation an einen Interpreter delegieren, der sie versteht.

Interpreter unterscheiden sich darin (vgl. Heterogenität in Abschnitt 4.1.4),

1. welche Repräsentationen sie interpretieren;
2. welche Semantik sie den Repräsentationen durch die Interpretation verleihen.

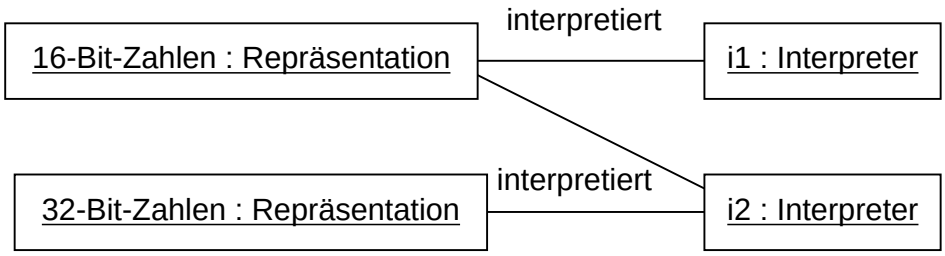

Abbildung 2.13: Repräsentationen für ganze Zahlen und Interpreter

Beispielsweise könnte ein Interpreter *i1* ganze Zahlen mit einer Genauigkeit von 16 Bit interpretieren, während ein anderer Interpreter *i2* 32 Bit unterstützt (vgl. Abbildung 2.13). Ein Interpreter *i2* kann auch beide Repräsentationen unterstützen. Zwei Interpreter *i1* und *i2*, die 16 Bit lange ganze Zahlen unterstützen, können beispielsweise bei der Addition zweier solcher Zahlen ein unterschiedliches Verhalten zeigen, wenn es zu einem Überlauf kommt, beispielsweise die Ergebniszahl abschneiden oder eine Ausnahme auslösen.

Beim Entwurf eines Interpreters ist zu entscheiden, welche Repräsentationen interpretiert werden sollen. Damit wird festgelegt, welche Funktionalität Objektsysteme erbringen können, die von diesem Interpreter interpretiert werden, beispielsweise bei der Genauigkeit der Zahlendarstellungen und Rechenoperationen.

Der Interpreter legt außerdem fest, ob Bindungen (im Rahmen von Kooperationen) an externe Objekte erfolgen können, welche externen Objekte zugänglich sind und welche Operationen auf diesen Objekten ausgeführt werden können. Wenn Zugriff auf externe Objekte besteht, kann die Funktionalität des Interpreters erweitert werden, beispielsweise durch die Anbindung von externen Rechenoperationen höherer Genauigkeit. Allerdings wird dadurch die Portabilität eingeschränkt; vgl. dazu Abschnitt 4.1.4.

Neben dem Zugriff auf externe Objekte regelt der Interpreter auch, welche Basisobjekte, also Objekte des Ablaufsystems, durch die der Interpretationsvorgang beeinflußt wird, zugänglich sind. Durch Basisobjekte können Operationen implementiert werden, die sich nicht in der vom Interpreter interpretierten Repräsentaton befinden, beispielsweise zur Persistenzsicherung (vgl. Abschnitt 3.2): um einen Interpretationszustand persistent abzulegen, könnte es eine Anweisung in der Repräsentation geben, die vom Interpreter direkt interpretiert wird, oder dies könnte durch den Aufruf einer Funktion (als Basisobjekt) erfolgen.

Für den Entwickler eines Interpreters ist also zu entscheiden,

- welche Funktionalität direkt im Interpreter durch die interpretierte Repräsentation implementiert wird,
- in welchem Umfang das Objektsystem Zugang zu externen und Basisobjekten erhält, und darauf aufbauend,
- welche Funktionalität durch externe und Basisobjekte abgedeckt wird und daher nicht direkt im Interpreter bzw. der von ihm interpretierten Repräsentation präsent ist.

2.3.2 Übersetzer

Übersetzer sind Funktionen, die Repräsentationen aufeinander abbilden. Sie finden sich sowohl in Entwicklungssystemen, um aus menschengerechten Repräsentationen semantisch äquivalente

maschinengerechte, interpretierbare Repräsentationen zu erzeugen, als auch in Ablaufsystemen, beispielsweise um eine nicht von einem Interpreter interpretierbare Repräsentation in eine semantisch äquivalente interpretierbare Repräsentation umzuformen.

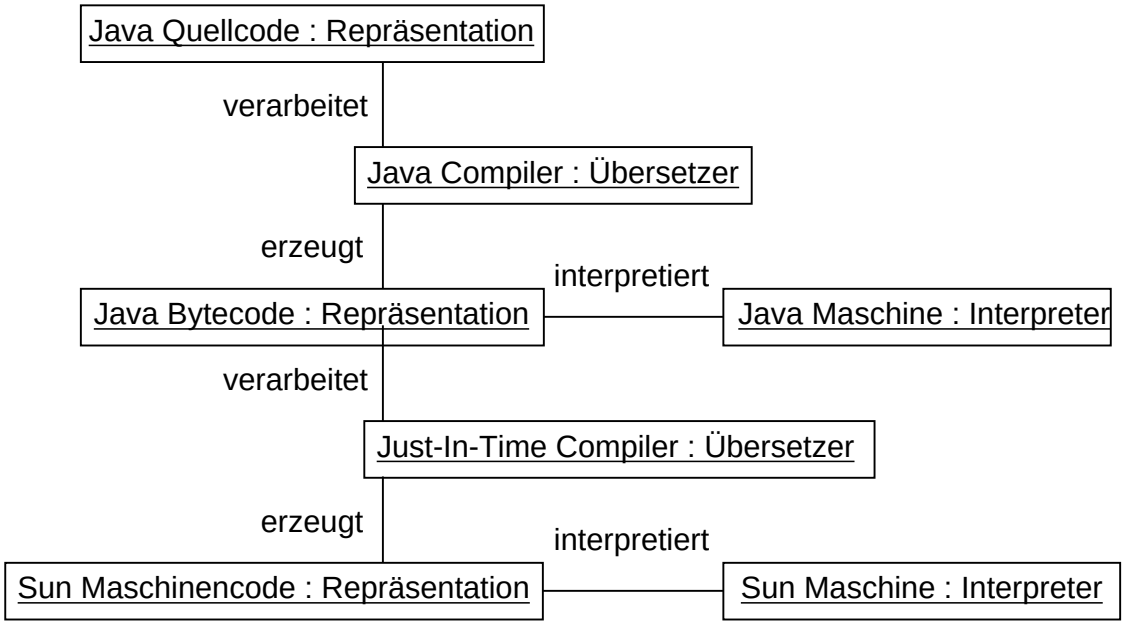

Abbildung 2.14: Repräsentationen und Übersetzer

Beispielsweise bildet ein Java-Übersetzer [Diehl 98] den menschenlesbaren Quellcode von Java-Programmen [Java Team et al. 96] in von einer Java-Maschine [Lindholm, Yellin 96] interpretierbaren Java-Bytecode [Gosling 95] ab (vgl. Abbildung 2.14); diesen Übersetzer findet man typischerweise in einer Entwicklungsumgebung für Java-Programme.

Ein Just-In-Time-Übersetzer [Gosling 95; Plezbert, Cytron 97] bildet den Bytecode in maschinenabhängigen Code ab, beispielsweise in einen Sun-Maschinencode für eine Sun-Maschine als Interpreter; dieser Übersetzer wird auf Sun-Maschinen eingesetzt, um Java-Bytecode direkt vor der Ausführung umzuwandeln. Diese Umwandlung wird evtl. von der Java-Maschine angestoßen, der damit die Interpretation an einen anderen Interpreter abgibt.

Übersetzer kommen für alle Arten von Objekten und für Bindungen zum Einsatz. Sie bilden Objekte und Bindungen in semantisch äquivalente Objekte und Bindungen ab (vgl. Abbildung 2.15). D.h. ein Übersetzer erzeugt aus einem Datenobjekt in einer Repräsentation ein Datenobjekt in einer anderen Repräsentation. Wenn die Objekte und Bindungen nicht direkt abgebildet, sondern umgewandelt oder ergänzt werden, handelt es sich um einen Generator (vgl. Kapitel 5).

2.4 Objektsystem

Objektsysteme sind Sammlungen von Objekten und Bindungen, welche ablauffähige Anwendungen realisieren, die Aufgaben der Anwendungswelt bearbeiten.

Objektsysteme stehen im Mittelpunkt des Modells (vgl. Abbildung 2.16). Objektsysteme laufen im Kontext von Ablaufumgebungen ab, welche die Ablaufsemantik der Objektsysteme bestimmen. Objektsysteme werden mit Hilfe von Entwicklungsumgebungen entwickelt, welche bestimmte Ablaufumgebungen unterstützen und somit Entwicklern Hilfsmittel bereitstellen, mit denen sie Objektsysteme für diese Ablaufumgebungen erstellen. Beispielsweise kann ein Java Objektsystem (vgl. Abbildung 2.17), das mit Hilfe einer Java Entwicklungsumgebung erstellt wird, auf einer Java-Ablaufumgebung betrieben werden. Hingegen könnte ein mit einer C++-Entwicklungsumgebung

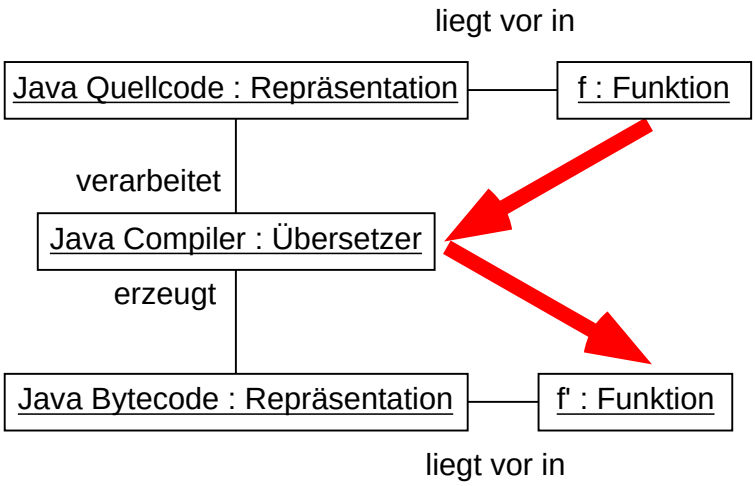

Abbildung 2.15: Übersetzen einer Funktion

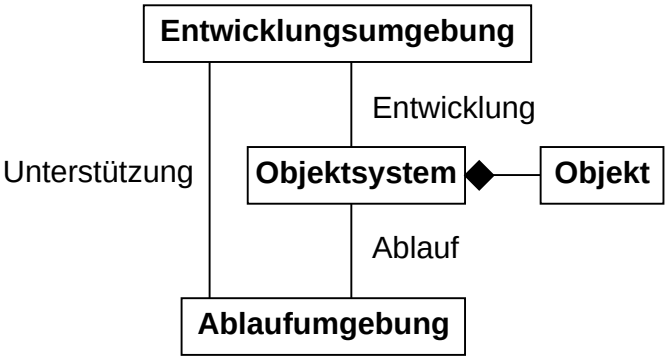

Abbildung 2.16: Objektsysteme, Ablauf- und Entwicklungsumgebungen

erstelltes C++-Objektsystem nicht (ohne weiteres) auf der Java-Ablaufumgebung ablaufen. Das mit der Java-Entwicklungsumgebung für eine Java-Ablaufumgebung erstellte Java-Objektsystem ist andererseits auch nicht auf einer (reinen) Solaris-Ablaufumgebung ablauffähig.

2.5 Ablaufumgebung und Ablaufsystem

Ablaufumgebungen sind Umgebungen, die Objektsystemen eine Ablaufsemantik verleihen, indem sie die Objektsysteme interpretieren und ihnen Funktionalität in Form von Objekten zur Verfügung stellen, welche die Objektsysteme zur Laufzeit nutzen können.

Ablaufumgebungen bestehen aus Ablaufsystemen, die wiederum Objektsysteme darstellen, sowie Basis- und externen Objekten (vgl. Abbildung 2.18).

Ein Ablaufsystem ist ein Objektsystem, das ein anderes Objektsystem interpretiert, ihm also eine Ablaufsemantik verleiht. Eine Ablaufumgebung muß ein Ablaufsystem enthalten, denn ansonsten kann einem Objektsystem keine Semantik verliehen werden, während externe Objekte optional sind.

Zentraler Bestandteil des Ablaufsystems ist der Interpreter (vgl. Abschnitt 2.3.1), der Objektsysteme interpretiert und ihnen somit die Basissemantik verleiht, die durch Basis- und externe

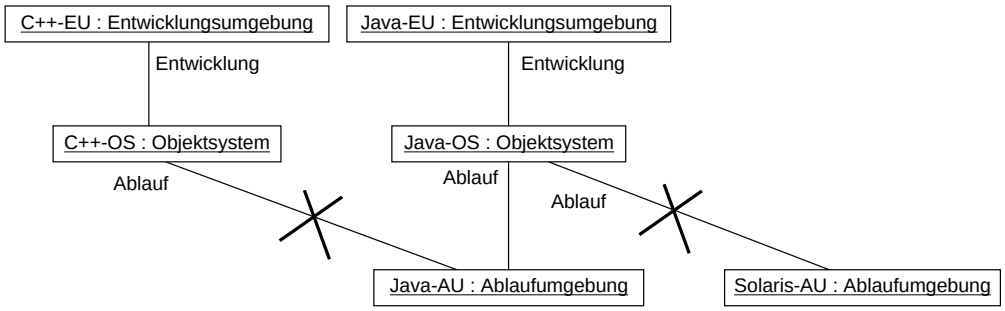

Abbildung 2.17: Beispiel für Objektsysteme und Umgebungen

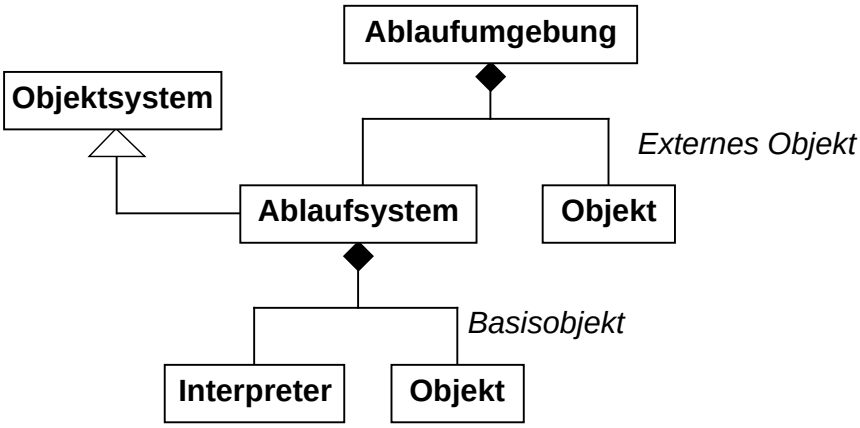

Abbildung 2.18: Ablaufumgebung

Objekte ergänzt wird. Objektsysteme binden mit Hilfe des Ablaufsystems Basis- bzw. externe Objekte und führen Operationen auf ihnen aus. Externe Objekte sind Objekte, die eine andere Ablaufumgebung als das Objektsystem besitzen. Basisobjekte sind Objekte des Ablaufsystems, durch welche die Ablaufeigenschaften des Objektsystems beeinflußt werden. In anderen Arbeiten werden diese Objekte beispielsweise Metaklassen oder Metaobjekte genannt [Smith 84; Maes 87; Kiszales et al. 91].

Ablaufsysteme unterscheiden sich durch die von ihrem Interpreter verarbeiteten Repräsentationen, die Semantik, die der Interpreter diesen Repräsentationen verleiht, und die Basisobjekte, die sie anbieten (vgl. Abbildung 2.19 und Heterogenität in Abschnitt 4.1.4):

Repräsentation: Eine Java-Maschine als Ablaufsystem verarbeitet Java-Bytecode als Repräsentation, während eine Sun-Maschine Sun-Maschinencode verarbeitet. Die Unterschiede können auch subtiler sein, wenn beispielsweise eine Java-Maschine nur eine Untermenge des Java-Bytecodes interpretiert, beispielsweise um Sicherheitslücken zu umgehen, indem sie keinen Zugriff auf externe Objekte bietet.

Semantik: Zwei Java-Maschinen unterschiedlicher Hersteller können demselben Java-Bytecode eine unterschiedliche Semantik zuordnen.

Basisobjekte: Eine persistente Java-Maschine könnte ein Basisobjekt *Persistenz* anbieten, über die ein Objektsystem seine Persistenz kontrollieren kann; ansonsten implementiert die persistente Java-Maschine dieselbe Semantik für den Java-Bytecode wie die nicht-persistente Java-Maschine.

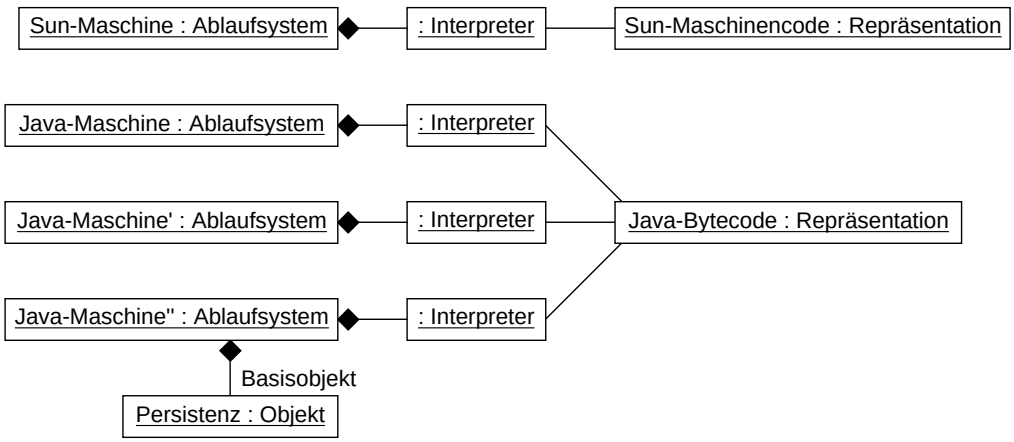

Abbildung 2.19: Unterschiedliche Ablaufsysteme

Basis- und externe Objekte sind entweder statisch in der Ablaufumgebung vorhanden oder werden dynamisch erzeugt und freigegeben.

2.5.1 Externes Objekt

Externe Objekte sind Objekte, die eine andere Ablaufumgebung als das Objektsystem besitzen, d.h. sie werden nicht von dem Interpreter der Ablaufumgebung interpretiert und unterliegen somit nicht den Ablaufregeln, die für das Objektsystem gelten.

Externe Objekte sind ebenso Objekte wie die Objekte des Objektsystems, also existieren Daten, Funktionen und Prozesse sowie Bindungen zwischen diesen Objekten, aber die Operationen auf diesen Objekten können eine andere Semantik besitzen als vergleichbare Operationen innerhalb der Ablaufumgebung. Beispielsweise kann eine Additionsoperation auf ganzen Zahlen in einer Ablaufumgebung zu einem Abbruch führen, wenn ein Überlauf auftritt, während in einer anderen Ablaufumgebung kein Abbruch eintritt.

Externe Objekte können in einer Repräsentation vorliegen, die von dem Interpreter des Ablaufsystems nicht interpretiert werden kann, jedenfalls nicht in ihrer ursprünglichen Semantik. Beispielsweise kann eine als externes Objekt vorliegende Funktion von einem Interpreter als (uninterpretiertes) Bitmuster verarbeitet werden, aber die Ablaufsemantik der Funktion geht dabei verloren, d.h. der Interpreter kann diese Funktion nicht aufrufen, keinen Funktionsabschluß bilden und nicht feststellen, welche Bindungen diese Funktion enthält.

Damit externe Objekte eine über ihre Repräsentation als Bitmuster hinausgehende Funktionalität erhalten, müssen sie entweder durch einen Übersetzer (hier als Teil der Ablaufumgebung) in die Repräsentation umgewandelt werden, die der Interpreter der Ablaufumgebung versteht, oder es muß eine Schnittstelle zu der Ablaufumgebung des externen Objekts existieren, so daß der Interpreter Operationen auf den externen Objekten in ihrer ursprünglichen Ablaufumgebung auslösen kann.

Der Interpreter legt fest, welche externen Objekte dem Objektsystem zugänglich sind und welche Operationen auf externen Objekten definiert sind. Im Basismodell sind keine Bindungen an externe Objekte möglich und es können keine Operationen auf externen Objekten angestoßen werden. Erst durch Kooperation wird dies erreicht (vgl. Kapitel 4).

2.5.2 Basisobjekt

Basisobjekte sind Objekte des Ablaufsystems, durch die die Ablaufeigenschaften des Objektsystems beeinflußt werden. Ein Objektsystem kann (reflektiv) auf Basisobjekte des Ablaufsystems

zugreifen, um Informationen über das Ablaufsystem zu erhalten, z.B. die Größe des verfügbaren Speicherraumes, und um Operationen auszulösen, die den Interpretationsvorgang betreffen oder beeinflussen, z.B. den aktuellen Interpreterzustand persistent ablegen (vgl. Abschnitt 3.1.3)..

Basisobjekte können sowohl externe als interne Objekte sein; je nachdem werden sie von dem Interpreter des Objektsystems interpretiert oder ihre Interpretation von dem Interpreter des Ablaufsystems vorgenommen. Sie können den Interpreter des Objektsystems in seinem Interpretationsvorgang beeinflussen. Beispielsweise kann ein Basisobjekt angesprochen werden, das einen persistenten Sicherungspunkt setzt, auf den der Interpretationsvorgang zurückgesetzt werden kann, oder die Prozeßwechsel können eingeschränkt werden, um einen unteilbar ablaufenden Abschnitt zu implementieren.

Der Interpreter ist ebenfalls ein Basisobjekt, d.h. er ist ein Objekt des Ablaufsystems und interpretiert sich nicht selbst. Wenn der Interpreter dem Objektsystem zugänglich gemacht wird, kann das Objektsystem einen (rekursiven) Interpretationsvorgang starten, parallel oder verschachtelt zu der eigenen Interpretation.

Welche Basisobjekte dem Objektsystem zugänglich gemacht werden und welche Operationen auf diesen Basisobjekten definiert sind, wird vom Interpreter festgelegt. Dieses vom Interpreter festgelegte Protokoll zur Behandlung von Basisobjekten wird Metaobjektprotokoll [Kiszales et al. 91] genannt.

2.6 Entwicklungsumgebung und Entwicklungssystem

Entwicklungsumgebungen sind Umgebungen, mit denen Objektsysteme für solche Ablaufumgebungen erstellt werden, die von den Entwicklungsumgebungen unterstützt werden. Entwicklungsumgebungen stellen Entwicklern Hilfsmittel zur Verfügung, mit denen sie die Objektsysteme erzeugen können. Entwicklungsumgebungen enthalten Funktionalität in Form von Objekten, welche die Entwickler in das Objektsystem integrieren können.

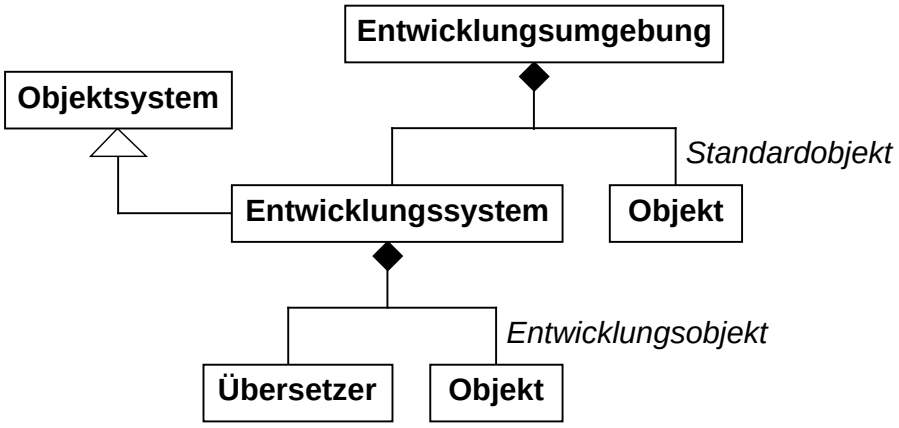

Abbildung 2.20: Entwicklungsumgebung

Entwicklungsumgebungen bestehen aus Entwicklungssystemen, die wiederum Objektsysteme darstellen, sowie Entwicklungs- und Standardobjekten (vgl. Abbildung 2.20).

Ein Entwicklungssystem ist ein Objektsystem, mit dem Objektsysteme entwickelt werden. Entwicklungssysteme werden durch einen Übersetzer (vgl. Abschnitt 2.3.2) geprägt, der Objektsysteme erzeugt sowie bei der Erzeugung Standard- und Entwicklungsobjekte in die Objektsysteme einbindet, d.h. Bindungen an diese Objekte erzeugt. Die eingebundenen Objekte werden zu internen Objekten, wenn sie von dem Interpreter der Ablaufumgebung interpretiert werden, und ansonsten zu externen Objekten.

Entwicklungssysteme unterscheiden sich in den vom Übersetzer verarbeiteten und erzeugten Repräsentationen, der beim Übersetzen angesetzten Semantik und den zur Verfügung gestellten Entwicklungsobjekten (vgl. Abbildung 2.21):

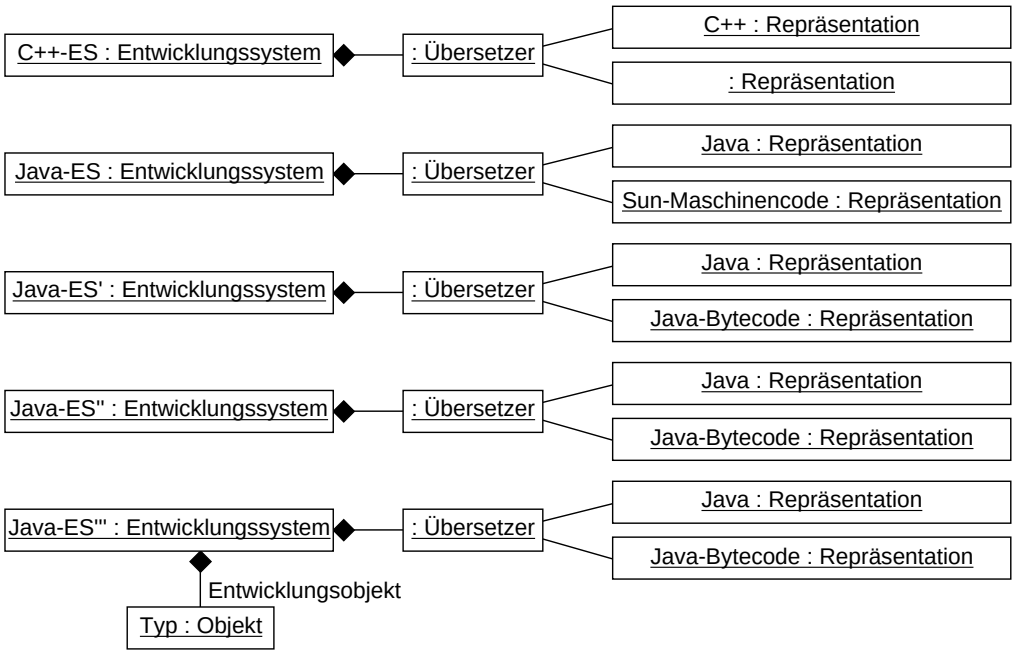

Abbildung 2.21: Unterschiedliche Entwicklungssysteme

Repräsentationen: Während ein C++-Entwicklungssystem C++-Code als Repräsentation verarbeitet, übersetzt ein Java-Entwicklungssystem Java-Code. Die Unterschiede in den übersetzen Repräsentationen können subtiler sein, beispielsweise wenn nur eine Untermenge des Java-Codes übersetzt wird. Java-Entwicklungssysteme können als Zielrepräsentation nicht nur Java-Bytecode, sondern auch beispielsweise Sun-Maschinencode erzeugen.

Semantik: Derselbe Java-Code kann von zwei unterschiedlichen Übersetzern, welche dieselbe Zielrepräsentation (wie Java-Bytecode) erzeugen, trotzdem mit unterschiedlicher Semantik belegt werden und somit zu unterschiedlichem Zielcode führen, z.B. zwecks Optimierung.

Entwicklungsobjekte: Zwei Java-Entwicklungssysteme können aus unterschiedlichen Entwicklungsobjekten bestehen und intern beispielsweise andere Typrepräsentationen benutzen. Je nachdem, ob bzw. wie sie den reflektiven Zugriff auf Entwicklungsobjekte unterstützen, können sich also andere Entwicklungsobjekte in dem erzeugten Objektsystem wiederfinden.

2.6.1 Standardobjekt

Standardobjekte sind Objekte der Entwicklungsumgebung, die in verschiedene Objektsysteme eingebunden werden können. Sie stellen Funktionalität zwecks Wiederverwendung zur Verfügung. Diese (komplexe) Funktionalität wird nur einmalig implementiert, getestet, dokumentiert usw., kann aber beliebig oft verwendet werden. Beispielsweise kann eine Bibliothek mit Rechenoperationen als Standardobjekt angeboten werden. Ein Objektsystem, das diese Rechenoperationen benötigt, bindet dieses Standardobjekt ein.

Ein Standardobjekt wird zu einem Objekt des Objektsystems, wenn es in einer Repräsentation vorliegt, die von dem Ablaufsystem interpretiert werden kann, für das das Objektsystem entwickelt wird. Ansonsten benötigt das Standardobjekt zur Laufzeit eine andere Ablaufumgebung und wird

damit zu einem externen Objekt. In diesem Fall kann der Zugriff auf dieses Objekt durch den Interpreter (vgl. Abschnitt 2.3.1) eingeschränkt sein.

Standardobjekte können Funktionalität kapseln, die in externen oder Basisobjekten implementiert ist. Sie bieten damit eine Schnittstelle zu der Ablaufumgebung, die unabhängig von den vom Interpreter zur Verfügung gestellten Zugriffsmechanismen auf externe und Basisobjekte ist. Für den Entwickler ergibt sich eine Komplexitätsreduktion, da er die Unterschiede zwischen externen, Basis- und internen Objekten nicht beachten muß.

Standardobjekte bilden den Basismechanismus für Abstraktionen aller Art. Standardobjekte stehen allen Entwicklern zur Verfügung, damit sie in Objektsysteme integriert werden können. Dies erhöht die Produktivität und schafft eine gemeinsame (standardisierte) Basis für Teamarbeit. Durch standardisierte Standardobjekte für unterschiedliche Ablaufumgebungen können Objektsysteme von Heterogenität (vgl. Abschnitt 4.1.4) abstrahieren. Durch die Nutzung von Standardobjekten in mehreren Objektsystemen wird die Kooperation (vgl. Abschnitt 4) zwischen Objektsystemen vereinfacht, da diese Objektsysteme somit über eine gemeinsame Basis verfügen.

2.6.2 Entwicklungsobjekt

Entwicklungsobjekte sind Objekte des Entwicklungssystems, welche Meta-Wissen über die Erzeugung des Objektsystems enthalten. Das Objektsystem kann mit ihrer Hilfe über sich selbst Aussagen machen oder sogar während des Ablaufs weitere Objekte entwickeln, beispielsweise die Funktion *compile* auf Seite 22.

Entwicklungsobjekte werden durch den Übersetzer zugänglich gemacht. Dieser Vorgang wird linguistische Reflektion genannt [Sheard 90; Stemple et al. 92b] (vgl. Abschnitt 5.2.5). Welche der Objekte des Entwicklungssystems zugänglich gemacht werden, ist eine Frage des Entwicklungssystems bzw. des Übersetzers.

Entwicklungsobjekte umfassen einerseits die statischen Bestandteile des Entwicklungssystems, also den Übersetzer und weitere (Hilfs-)Objekte, beispielsweise Parser, Typprüfer oder Codegenerator. Diese Objekte können zur Laufzeit des Objektsystems genutzt werden, beispielsweise um zur Laufzeit Objekte in bestimmten Repräsentationen zu erzeugen oder Typrepräsentationen miteinander zu vergleichen.

Andererseits können auch dynamisch während der Entwicklung erzeugte Objekte, beispielsweise Typinformationen oder Zwischenrepräsentationen, als Metainformationen über das entwickelte Objektsystem in das Objektsystem eingebunden werden. Diese Objekte werden zur Laufzeit ausgewertet, um beispielsweise Kompatibilitäten anhand von Typinformationen zu prüfen oder andere Repräsentationen von Objekten zu erzeugen.

2.7 Zusammenfassung

Das in diesem Kapitel eingeführte Modell für Objektsysteme bildet die Grundlage für die weiteren Untersuchungen zum Betrieb und zur Entwicklung kooperierender Objektsysteme in dieser Arbeit.

- Objekte und Bindungen zwischen Objekten bilden die grundlegenden semantischen Kategorien des Modells. Objekte werden anhand ihres Verhaltensaspekts in passive, aktivierbare und aktive Objekte (Daten, Code und Prozesse) unterschieden. Ihre statischen und Ablaufeigenschaften, insbesondere die Bindungs- und Repräsentationsaspekte, werden herausgearbeitet. Diese Eigenschaften sind als Grundlage für die detaillierte Diskussion von Persistenz, Kooperation, Evolution und Entwicklung von Objektsystemen unerläßlich.

- Objektsysteme bilden eine einheitliche, Ablauf- und Entwicklungssysteme einschließende Grundlage für den Betrieb und die Entwicklung von kooperierenden Objektsystemen. Durch die einheitliche Modellierung der Ablauf- und des Entwicklungssysteme als Objektsysteme werden die zum Betrieb und zur Entwicklung zu betrachtenden Fälle verallgemeinert und damit reduziert. Beispielsweise wird die reflektive Bindung eines (Basis-)Objektes des Ablaufsystems durch ein Objektsystem als ein Spezialfall einer Bindung bei der Kooperation

zwischen Objektsystemen betrachtet. Diese Vereinheitlichung ermöglicht es also, Spezialfälle auf allgemeine Fälle zurückzuführen und somit die Diskussion in den folgenden Kapiteln zu fokussieren.

Das Ausgangsszenario dieser Arbeit stellt sich in dem hier eingeführten Objektsystemmodell wie folgt dar (vgl. Abbildung 2.22): Die grundlegenden Bausteine für Objektsysteme, Ablauf- und Entwicklungsumgebungen bilden Objekte und Bindungen. Objekte und Bindungen liegen in Repräsentationen vor, die von Übersetzern als Bestandteil von Entwicklungsumgebungen erzeugt und von in Ablaufumgebungen enthaltenen Interpretern ausgeführt werden. Die Ablaufumgebung eines Objektsystems enthält ein Ablaufsystem, bestehend aus einem Interpreter sowie Basisobjekten. Objekte des Objektsystems können durch Kooperation mit anderen Objektsystemen externe Objekte binden und Operationen auf diesen externen Objekten ausführen.

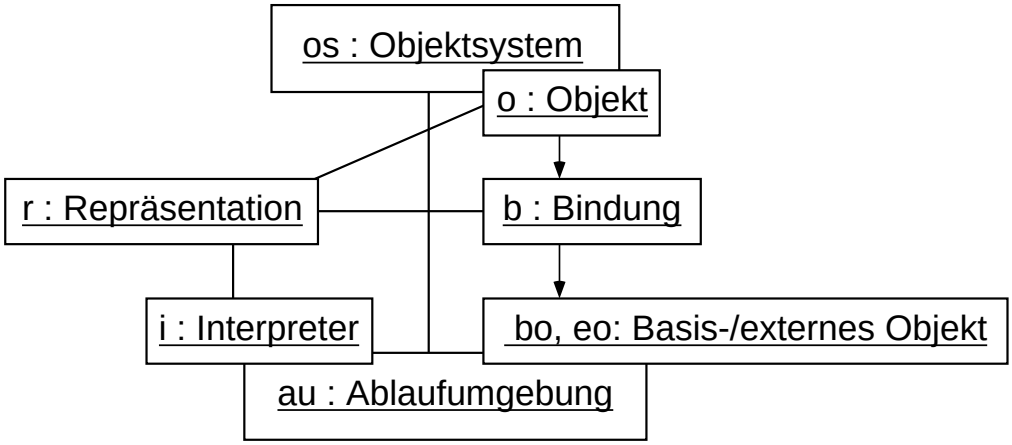

Abbildung 2.22: Ablaufumgebung: Szenario

In diesem Szenario ergeben sich folgende Herausforderungen an den Betrieb und die Entwicklung von Objektsystemen, die anhand des eingeführten Modells in den folgenden Kapiteln behandelt werden:

Persistenz: Ein Ablaufzustand eines Objektsystems, bestehend aus Objekten und Bindungen, wird von der Ablaufumgebung festgehalten, potentiell unendlich lange aufbewahrt und reaktiviert.

Evolution: Ein Objektsystem ändert sich mit der Zeit, um sich an neue Anforderungen anzupassen.

Heterogenität: Verschiedene Ablaufumgebungen unterscheiden sich voneinander, um spezielle Anforderungen der Objektsysteme zu erfüllen.

Kooperation: Unabhängige Objektsysteme kooperieren miteinander, um gemeinsam eine Aufgabe zu lösen.

Heterogenität und Persistenz sind *Eigenschaften* der Ablaufumgebungen für Objektsysteme, wobei Persistenz eine Eigenschaft *einer* Ablaufumgebung ist, während Heterogenität sich auf *mehrere* Ablaufumgebungen bezieht. Kooperation und Evolution sind *Vorgänge*. Evolution ist ein Vorgang innerhalb *eines* Objektsystems, während Kooperation zwischen *mehreren* Objektsystemen stattfindet.

Kapitel 3

Betrieb von Objektsystemen

In diesem Kapitel wird aufbauend auf dem in Kapitel 2 beschriebenen (statischen) Objektmodell der (dynamische) Ablauf von Objektsystemen durch Ablaufsysteme beschrieben, wobei insbesondere die Erhaltung der Ablauffähigkeit mit Hilfe von Reflektion betrachtet wird (vgl. Abschnitt 3.1). Um langlebige Anwendungen zu unterstützen, erhalten persistente Objektsysteme die (insbesondere dynamisch während des Ablaufs erzeugten) Objekte des Objektsystems über die Lebensdauer des Ablaufsystems hinaus (vgl. Abschnitt 3.2). Den in der potentiell unendlich langen Lebenszeit eines Objektsystems auftretenden Fehlern und geänderten Anforderungen wird durch Evolution begegnet (vgl. Abschnitt 3.3). Bei der Darstellung von Ablauf, Persistenz und Evolution wird besonders auf das reflektive Zusammenspiel zwischen Objektsystem und Ablaufsystem eingegangen. Dieses Zusammenspiel bildet die Grundlage für die in Kapitel 4 beschriebene Kooperation zwischen Objektsystemen.

3.1 Ablauf von Objektsystemen

Objektsysteme durchlaufen einen Lebenszyklus, der von der Entwicklung über den Ablauf bis zur Freigabe der durch das Objektsystems belegten Ressourcen verläuft (vgl. Abschnitt 3.1.1). Während des Ablaufs des Objektsystems werden Ablaufzustände eingenommen, die sich aus den Zuständen der Objekte und Bindungen des Objektsystems sowie dem vom Objektsystem durch Seiteneffekte veränderten Zustand der Ablaufumgebung ergeben (vgl. Abschnitt 3.1.2). Das Objektsystem greift reflektiv auf Objekte seines Ablaufsystems zu und registriert beispielsweise Funktionen, die beim Eintreten bestimmter Ablaufzustände (bzw. Ereignisse) vom Ablaufsystem aufgerufen werden (vgl. Abschnitt 3.1.3). Während des Ablaufs werden Objekte, die nicht mehr nutzbar sind, wieder freigegeben, um die Ablauffähigkeit des Objektsystems durch Wiederverwendung des von den Objekten belegten Speichers zu erhalten (vgl. Abschnitt 3.1.4). Die vom Objektsystem in der Ablaufumgebung beanspruchten Ressourcen werden ebenfalls freigegeben, sobald sie nicht mehr benötigt werden; dazu wird ein generischer Freigabemechanismus genutzt, der auf den reflektiven Eigenschaften des Ablaufsystems aufbaut (vgl. Abschnitt 3.1.5).

3.1.1 Lebenszyklus

Objektsysteme durchlaufen folgenden Lebenszyklus (vgl. Abbildung 3.1):

1. Erzeugung des Objektsystems durch ein Entwicklungssystem.
2. Hochfahren einer Instanz des Objektsystems (Inkarnation) durch ein Ablaufsystem.
3. Bearbeitung einer anwendungsabhängigen Aufgabe durch das Objektsystem.
4. Herunterfahren des Objektsystems durch das Ablaufsystem.

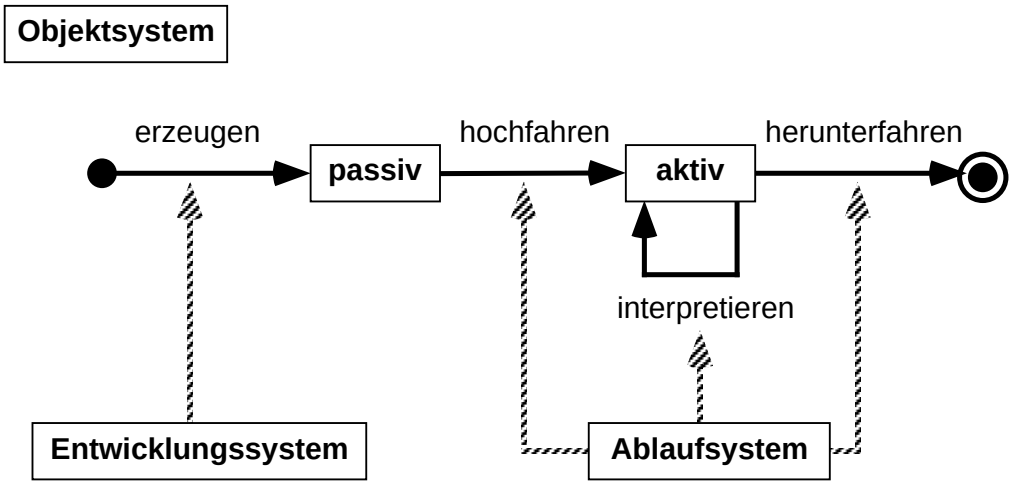

Abbildung 3.1: Lebenszyklus eines Objektsystems

Entwicklungssysteme erzeugen Objektsysteme. Nach der Erzeugung sind das Entwicklungssystem und die Entwicklungsumgebung obsolet, bis auf die Entwicklungs- und Standardobjekte (vgl. Abschnitt 2.6), die vom Objektsystem genutzt werden. Zur Entwicklungszeit muß die Ablaufumgebung des Objektsystems (noch) nicht vorhanden sein. Zur Ausführung wird das Objektsystem von der Entwicklungsumgebung an ein Ablaufsystem der Ablaufumgebung weitergereicht.

Der Übergang von der Entwicklungs- zur Ablaufumgebung besteht beispielsweise aus dem Zusammenbinden (*linking*) des ablauffähigen Objektsystems aus mehreren Objekten, d.h. fehlende Bindungen zwischen den Objekten werden vervollständigt. Dieses Zusammenbinden des Objektsystems zu einer ausführbaren Inkarnation kann ein mehrschrittiger Vorgang sein, der teilweise erst während des Ablaufs erfolgt (*dynamic linking*). Das Zusammenbinden wird beispielsweise durch den Aufruf von Funktionsobjekten implementiert [Koch 97]. Beim Binden können Prüfungen oder Transformationen der Objekte vorgenommen werden, beispielsweise um die Performanz zu erhöhen oder Ablauffehler auszuschließen [Gosling 95].

Objektsysteme werden von Ablaufsystemen interpretiert. Wenn ein Objektsystem einem Ablaufsystem der Ablaufumgebung zur Interpretation übergeben wird, entsteht eine Inkarnation des Objektsystems. Das Ablaufsystem führt Operationen auf den Objekten des Objektsystems aus.

Um eine Aufgabe zu bearbeiten, wird mindestens ein Prozeßobjekt (vgl. Abschnitt 2.1) erzeugt, das dem Interpreter als Ausgangspunkt des Interpretationsvorgangs dient. Dieses Objekt liegt in einer Repräsentation vor, die von dem Interpreter verarbeitet werden kann, also einer ausführbaren Repräsentation. Bei mehrprozeßfähigen (*multi-tasking*) Ablaufsystemen können auch mehrere Prozeßobjekte erzeugt und nebenläufig interpretiert werden.

Das Ablaufsystem interpretiert die Prozeßobjekte, indem es die Anweisungen der mit den Prozeßobjekten verbundenen Funktionsobjekte ausführt. Weitere Prozeßobjekte können erzeugt und aktiviert werden. Prozesse können suspendiert und wieder "aufgeweckt" werden. Laufende Prozesse werden beendet, wenn das mit ihnen verbundene (primäre) Funktionsobjekt abgearbeitet ist oder ein Prozeß explizit abgebrochen wird. Dieser Abbruch kann vom Prozeß selber, von anderen Prozessen oder vom Ablaufsystem ausgelöst werden.

Der Ablauf des Objektsystems endet, wenn kein Prozeß mehr interpretiert wird und auch kein suspendierter Prozeß mehr aufgeweckt werden kann. Das Ablaufsystem fährt das Objektsystem herunter, d.h. es gibt Ressourcen frei, die das Objektsystem belegt, beispielsweise den durch das Objektsystem belegten Speicher (vgl. Abschnitt 3.1.4).

Entwicklungs- und Ablaufsysteme sind wiederum Objektsysteme, die eine Entwicklungs- und Ablaufumgebung benötigen (vgl. Abbildung 3.2). Sie durchlaufen also den gleichen Lebenszyklus.

Initiale Ablaufumgebung ist die Hardware eines Rechensystems inklusive des Hauptprozessors als Ablaufsystem. Dieser Hauptprozessor stellt eine Art persistentes Objektsystem (vgl. Abschnitt 3.2) dar, das bei Einschalten des Stroms in einem definierten Zustand hochgefahren wird und dann durch Hochfahren weiterer Objektsysteme eine geschachtelte Hierarchie von Ablaufumgebungen schafft. Bevor ein Objektsystem hochgefahren werden kann, muß also eine Ablaufumgebung vorhanden sein und ein Ablaufsystem hochgefahren werden. Umgekehrt kann ein Objektsystem nur arbeiten, solange seine Ablaufumgebung existiert und sein Ablaufsystem arbeitet. Sobald das Ablaufsystem eines Objektsystems heruntergefahren wird, wird auch das Objektsystem heruntergefahren. Dieses geschieht spätestens beim Aussschalten des Stromes, was das grundlegende Ablaufsystem, den Hauptprozessor, herunterfährt, und damit auch alle darauf aufbauenden Objekt- und Ablaufsysteme.

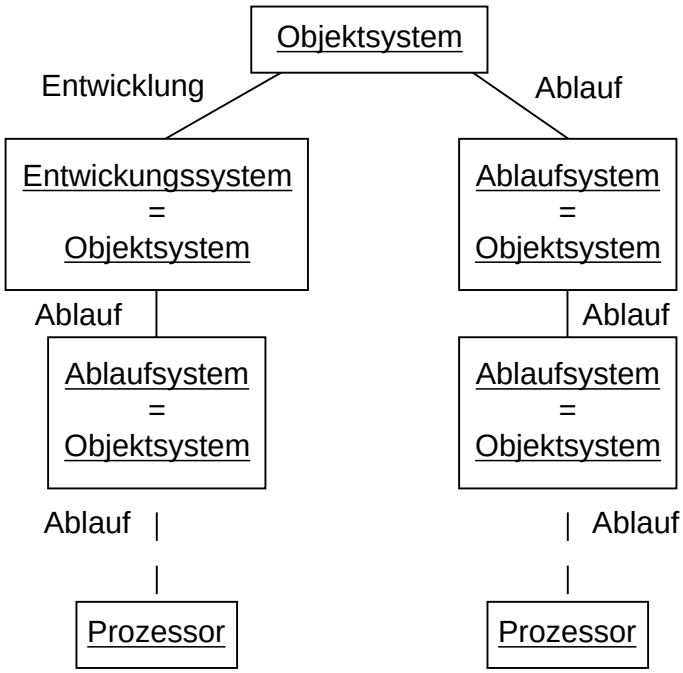

Abbildung 3.2: Objektsysteme und Ablaufsysteme

3.1.2 Ablaufzustände

Während des Ablaufs eines Objektsystems führt der Interpreter des Ablaufsystems Operationen aus, die den Zustand des Objektsystems und der Ablaufumgebung ändern.

Objektsystem: Der Zustand des Objektsystems ergibt sich aus den Zuständen der Objekte und den Bindungen zwischen den Objekten. Zustandsändernde Operationen sind demnach:

- Anlegen eines neuen Objektes im Objektsystem. Für das Objekt werden Ressourcen, zumindest Speicher, reserviert. Das Objekt ist initial an den erzeugenden Prozeß gebunden.
- Ändern des Zustands, also des Bitmusters, eines Objekts des Objektsystems.
- Anlegen oder Auflösen einer Bindung zwischen zwei Objekten eines Objektsystems.
- Freigeben eines Objektes des Objektsystems. Das Objekt ist in dem Objektsystem nicht mehr nutzbar. Seine Ressourcen können freigegeben werden. Zur automatischen

Speicherfreigabe vgl. Abschnitt 3.1.4, zur generischen Freigabe weiterer Ressourcen Abschnitt 3.1.5.

Ablaufumgebung: Der Zustand der Ablaufumgebung setzt sich aus den Zuständen der Objekte und der Objektsysteme, die die Ablaufumgebung bilden, zusammen. Dabei handelt es sich insbesondere um das Ablaufsystem des interpretierten Objektsystems, dessen Ablaufsystem usw. Die Zustandsänderungen in der Ablaufumgebung werden durch die Interpretation seiteneffektbehafteter Operationen durch das Ablaufsystem vorgenommen. Dazu zählen auch solche Operationen wie das Öffnen eines Fensters auf einer grafischen Benutzeroberfläche, wenn dies über eine Operation des Interpreters des Ablaufsystems geschieht. Wenn eine seiteneffektbehaftete Operation an ein anderes Objektsystem weitergereicht wird, handelt es sich dagegen um Kooperation, die in Kapitel 4 behandelt wird.

Ein konkreter Ablaufzustand ergibt sich aus der Folge der seit der Erzeugung des Objektsystems ausgeführten Operationen. Ein Ablaufzustand kann aber nicht unbedingt durch Wiederholung der Operationsfolge wiederhergestellt werden weil zeit- oder seiteneffektabhängiges Verhalten sich durch andere Ausgangsbedingungen und nebenläufige Abläufe ändern kann. Beispielsweise könnten Benutzereingaben, also aufnehmende Seiteneffekte, einen anderen Wert besitzen oder Ressourcenbeschränkungen Abläufe verhindern, welche diese Ressourcen benötigen.

3.1.3 Reflektion

Das Objektsystem greift während des Ablaufs auf Objekte seines Objektsystems zu, beispielsweise um einen ressourcenschonenden Betrieb zu ermöglichen (vgl. Abschnitt 3.1.5), die Persistenz von Prozessen zur sichern (vgl. Abschnitt 3.2) und integritätserhaltende Evolution durchzuführen (vgl. Abschnitt 3.3). Ablaufsysteme, die diesen Zugriff des Objektsystems auf ihre Objekte unterstützen, sind reflektiv. Damit stellt Reflektion eine spezielle Art der Kooperation zwischen Objektsystemen dar (hier: Objektsystem und Ablaufsystem; vgl. auch Kapitel 4).

Reflektive Ablaufsysteme [Smith 84; Maes 87; Watanabe, Yonezawa 88; Ferber 88; Foote, Johnson 89; Kirby et al. 96] werden in ihrer Funktionalität durch die Objektsysteme, die sie interpretieren, erweitert oder beeinflußt. Das Ablaufsystem wird durch Aufrufe in das Objektsystem (*up-call*) erweitert bzw. angepaßt [Cohrs et al. 88; Menon et al. 93; Lea et al. 93] oder teilt dem Objektsystem Ereignisse wie die Änderung der Ablaufumgebung mit [Noble, Satyanarayanan 95; Satyanarayanan et al. 95]. Dazu registriert das Objektsystem beim Ablaufsystem Funktionen, die das Ablaufsystem zu definierten Zeitpunkten aufruft. Diese definierten Zeitpunkte sind "Einhängpunkte" (*hooks*) für die Funktionen. Typische Zeitpunkte sind (im Vorgriff auf die in der Arbeit gezeigten Einsatzbereiche):

- Ein Objekt ist nicht mehr nutzbar und soll freigegeben werden.
- Ein Objekt wird nur noch durch besonders gekennzeichnete, sogenannte "schwache" Bindungen gebunden.
- Eine bestimmte Operation soll auf einem Objekt ausgeführt werden oder ist gerade ausgeführt worden.
- Ein Zeitintervall ist abgelaufen oder ein Zeitpunkt erreicht.
- In der Ablaufumgebung tritt ein Ereignis auf, beispielsweise wird eine Taste gedrückt, ein Mauszeiger bewegt oder ein Gerät ausgeschaltet.
- Bei der Belegung einer Ressource, beispielsweise Speicherplatz, ist eine (untere oder obere) Schranke erreicht oder überschritten.

Die "eingehängten" Funktionen werden zu den vorherbestimmten Zeitpunkten von dem Ablaufsystem in einem Prozeß ausgeführt, d.h. das Ablaufsystem stößt die Ausführung unabhängig vom

Ablaufzustand des Objektsystems an; daher werden die Funktionen auch (asynchrone) "Rückruffunktionen" (*callbacks*) genannt.

Das Objektsystem beeinflußt damit sein eigenes Ablaufverhalten, stellt Berechnungen über sich selbst an und delegiert zwecks Konsistenzerhaltung und Fehlervermeidung das Auslösen von zustandsabhängigem Verhalten an das Ablaufsystem. In objektorientierten Systemen ist diese reflektive Beeinflussung teilweise auch unter den Begriffen Metaklassen oder Metaobjektprotokoll zu finden [Smith 84; Maes 87; Kiszales et al. 91].

3.1.4 Objektfreigabe

Während des (potentiell unendlich langen) Ablaufs eines Objektsystems können beliebig viele Objekte erzeugt werden. Jedes Objekt belegt Speicherplatz. Wenn alle erzeugten Objekte erhalten bleiben, wird mit der Zeit immer mehr des (endlichen) Platzes zur Speicherung von Objekten belegt. Der belegte Speicherplatz wächst ständig, bis eine Grenze, z.B. durch begrenzte Adressierungsräume, erreicht ist. An dieser Grenze endet die Ablauffähigkeit des Objektsystems, weil kein weiterer Platz mehr vorhanden ist.

Normalerweise werden nicht alle der während der Lebenszeit des Objektsystems erzeugten Objekte während des gesamten Ablaufs genutzt. Viele Objekte besitzen eine kürzere Nutzungsdauer und brauchen daher nur für eine begrenzte Zeit vorhanden sein.

Ein Objekt ist nutzbar, wenn es entweder ein aktiver (vom Ablaufsystem interpretierter) Prozeß ist oder von einem aktiven Prozeß über Bindungen zugreifbar ist (vgl. Abbildung 3.3); diese Definition wird für kooperierende Objektsysteme in Kapitel 4 erweitert, um auch die Nutzung von Objekten durch aktive Prozesse anderer Objektsysteme zu erfassen. Inaktive (suspendierte) Prozesse werden nur indirekt einbezogen, wenn sie von einem aktiven Prozeß aus erreichbar sind und somit von ihm aktiviert werden könnten.

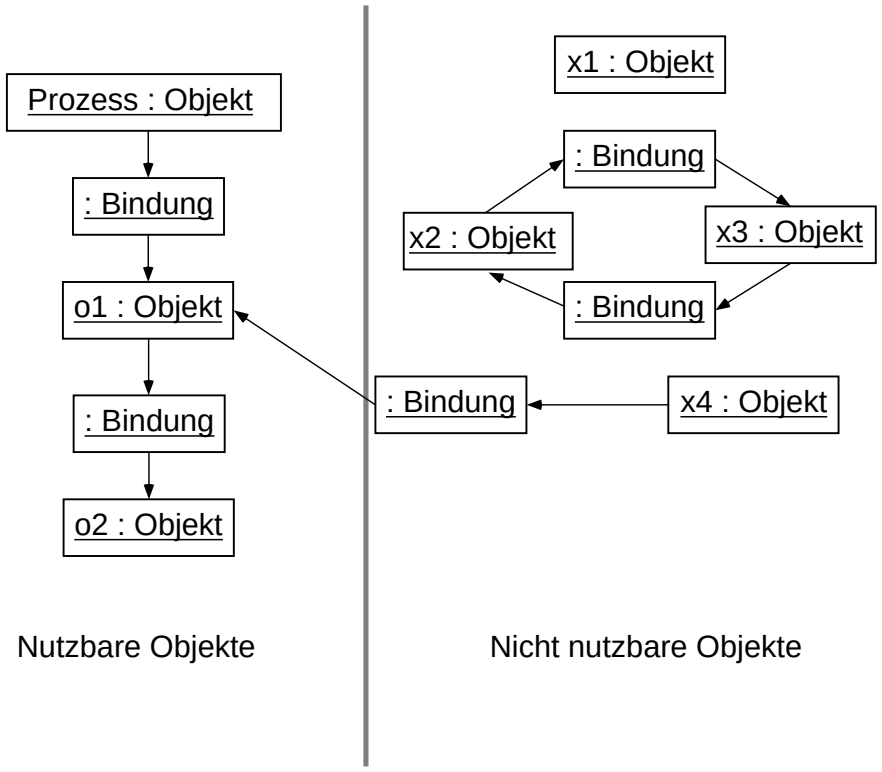

Abbildung 3.3: Nutzbare und nicht nutzbare Objekte

Die Menge der nutzbaren Objekte wird schrittweise durch die transitive Erreichbarkeit über Bindungen zwischen Objekten definiert. Ausgangspunkt bilden die aktiven Prozesse eines Objektsystems (bzw. der aktive Prozeß, wenn das Ablaufsystem nicht mehrprozeßfähig ist). Die Menge wird vervollständigt durch alle Objekte, die über Bindungen transitiv von diesen Objekten aus erreichbar sind. Alle Objekte, die von benutzbaren Objekten gebunden werden, sind ebenfalls erreich- und somit benutzbar. Alle anderen Objekte sind nicht mehr benutzbar (wie *x1, x2, x3, x4*). Wenn zyklische Bindungen zwischen mehreren Objekten existieren (wie zwischen *x2* und *x3*) und keines dieser Objekte von einem nutzbaren Objekt aus erreichbar ist, sind diese Objekte nicht mehr benutzbar. Wenn ein nicht mehr nutzbares Objekt wie *x4* ein nutzbares Objekt bindet, hat dies keine Auswirkungen.

Objekte, die nicht mehr benutzbar sind, können freigegeben werden, d.h. der von ihnen verbrauchte Speicher kann für andere Objekte genutzt werden. Allerdings dürfen keine Objekte freigegeben werden, die noch nutzbar sind. Dies führt zu zwei Forderungen an die Integrität und Ablauffähigkeit von Objektsystemen:

Speichereffizienz: Nicht mehr nutzbare Objekte werden freigegeben.

Konsistenzerhaltung: Nutzbare Objekte werden nicht freigegeben.

Die Freigabe von Objekten erfolgt explizit durch den Entwickler oder automatisch durch eine Freispeicherverwaltung (*garbage collection*) des Ablaufsystems. Die explizite Freigabe kann die aufgestellten Anforderungen nicht erfüllen:

Speicherlecks (*memory leaks*): Es muß eine Bindung an das freizugebende Objekt existieren, damit es freigegeben werden kann. Wenn es keine Bindung des Objektes mehr gibt, kann es nicht freigegeben werden und somit werden die Ressourcen weiter verbraucht und können nicht neu belegt werden.

Inkonsistente Bindungen (*dangling references*): Ein Objekt kann an mehreren Bindungen beteiligt sein. Wenn das Objekt über eine Bindung freigegeben wird, müssen alle Bindungen an das Objekt abgebaut werden. Wenn dies nicht geschieht, führt dies zu einem inkonsistenten Zustand des Objektsystems.

Daher können die Anforderungen durch Ablaufsysteme ohne automatische Speicherverwaltung wie beispielsweise solche der Programmiersprache C [Kernighan, Ritchie 90] nicht gewährleistet werden.

Automatische Speicherfreigabemechismen wie beispielsweise des Ablaufsystems der Programmiersprache Java [Java Team et al. 96; Lindholm, Yellin 96] garantieren die Einhaltung der Forderungen und vermeiden damit die Probleme der expliziten Freigabe. Bei der automatischen Freigabe stellt das Ablaufsystem fest, welche Objekte des Objektsystems von aktiven Prozessen aus erreichbar und daher benutzbar sind. Die nicht benutzbaren Objekte werden automatisch vom Ablaufsystem freigegeben, so daß ihr Speicherplatz wiederverwendet werden kann.

3.1.5 Generische Ressourcenfreigabe

Neben dem Speicherplatz belegen (Prozeß-)Objekte weitere Ressourcen. Dabei handelt es sich um

- objektsysteminterne Ressourcen, die zwischen mehreren Prozessen geteilt werden, beispielsweise Semaphore [Piellusch 96], und
- Ressourcen der Ablaufumgebung, die durch Seiteneffekte der vom Ablaufsystem ausgeführten Operationen belegt werden (vgl. beispielsweise [Bharat, Cardelli 95; Kornacker 95]).

Ressourcen sollten spätestens in dem Moment freigegeben werden, in dem der Prozeß, der sie nutzt, nicht mehr existiert.

Angenommen, ein Prozeß sperrt einen Drucker zur exklusiven Ausgabe; im normalen Ablauf gibt er diesen Drucker auch wieder frei:

```
lockPrinter()
...
unlockPrinter()
```

Allerdings kann es passieren, daß der Drucker nie wieder freigegeben wird, beispielsweise durch folgenden Ablauf:

1. Der Prozeß wird suspendiert.
2. Wenn er durch keinen anderen Prozeß mehr erreichbar ist, kann er nicht mehr aufgeweckt werden.
3. Die Speicherverwaltung erkennt, daß er nicht mehr nutzbar ist und gibt seinen Speicher frei.

Der Drucker wird nicht mehr freigegeben, bleibt also für andere Prozesse gesperrt.

Der Drucker kann freigegeben werden, wenn der Prozeß von der Speicherverwaltung freigegeben wird. Die Freigabe kann (eingeschränkt) *implizit* durch das Ablaufsystem erfolgen, das dazu während des Prozeßablaufs die belegten und freigegebenen Ressourcen vermerkt und bei der Freigabe des Prozeßobjektes durch die Speicherverwaltung die noch von dem Prozeß belegten Ressourcen freigibt. Dazu muß das Ablaufsystem den Ablauf des Prozesses überwachen, um während des Ablaufs die ressourcenbelegenden und -freigebenden Aktionen zu erkennen und zu verzeichnen. Objektsysteminterne Ressourcen und prozeßübergreifende anwendungsabhängige Entwicklungsmuster (vgl. Abschnitt 5.3) können damit nicht abgedeckt werden; beispielsweise ist es nicht möglich, daß ein Prozeß einen Drucker belegt und ein anderer ihn freigibt. Damit stellt die implizite Ressourcenfreigabe (im Gegensatz zur impliziten Speicherfreigabe) eine Einschränkung der Ablauffähigkeit dar.

Daher steuert der Entwickler des Objektsystems *explizit* die Freigabe der Ressourcen, allerdings mit der Unterstützung der automatischen Speicherverwaltung. In diesem Fall stellt die Speicherverwaltung eine Möglichkeit zur Verfügung, zu Prozeßobjekten reflektiv Funktionen als Basisobjekte zu registrieren, die vom Ablaufsystem aufgerufen werden, wenn die Prozesse freigegeben werden sollen. Beispielsweise wird für das aktuelle Prozeßobjekt eine Funktion registriert, die den Drucker freigibt:

```
lockPrinter
registerGCFunction(fun() unlockPrinter)
```

Die Speicherverwaltung ist dafür verantwortlich, daß die Funktion aufgerufen wird, wenn der Prozeß nicht mehr nutzbar ist und daher freigegeben werden soll.

Um mehrere Ressourcen zu belegen und wieder freizugeben, werden entweder mehrere Funktionen registriert, die von der Speicherverwaltung aufgerufen werden, oder eine registrierte Funktion übernimmt die Aufgabe, die Ressourcen freizugeben, beispielsweise indem sie die Funktionen zur Freigabe der einzelnen Ressourcen aufruft. Aus der Sicht des Ablaufsystems reicht die Möglichkeit, eine Funktion zu registrieren, aus, um die gewünschte Funktionalität zu implementieren. Die Komplexität der Verwaltung mehrerer Funktionen wird damit in das Objektsystem verlagert.

Wenn eine Ressource explizit freigegeben wird, sollte die Freigabefunktion deregistriert werden, da ansonsten die Ressource ein weiteres Mal freigegeben wird, wenn das Prozeßobjekt freigegeben wird:

```
lockPrinter
let unlockFun = registerGCFunction(fun() unlockPrinter)
...
unlockPrinter
deregisterGCFunction(unlockFun)
```

Der Bezeichner *unlockFun* für die registrierte Funktion erlaubt eine Identifikation und Deregistrierung der Freigabefunktion. Hier wird davon abstrahiert, ob die Verwaltung mehrerer Freigabefunktionen vom Ablaufsystem oder vom Objektsystem vorgenommen wird.

Die explizite Freigabe bietet die Möglichkeit, komplexe Entwicklungsmuster (vgl. zur Entwicklung auch Kapitel 5) zu realisieren. Beispielsweise soll ein Prozeß an seinem Ende einen suspendierten Prozeß *sleepingProcess* aufwecken. Um dieses Aufwecken sicherzustellen, auch wenn der aufweckende Prozeß freigegeben wird, wird die Freigabefunktion herangezogen:

```
let wakeUpFun = registerGCFunction(self  fun() wakeup(sleepingProcess))
```

Wenn ein Objekt von mehreren Prozessen gemeinsam benutzt wird oder schon vor Ende des Prozesses freigegeben werden soll, wird die Freigabefunktion an das Objekt statt an den erzeugenden Prozeß gebunden. Beispielsweise könnte das Ablaufsystem dem Objektsystem zur Interaktion mit einem Benutzer Zugang zu einer graphischen Benutzeroberfläche über dynamisch erzeugte Fensterobjekte bieten. Normalerweise erfolgt das Öffnen und Schließen des Fensters explizit:

```
let aWindow = openWindow(...)
...
closeWindow(aWindow)
```

Wenn das Fenster von mehreren Prozessen zur Interaktion genutzt wird, sollte es erst geschlossen werden, wenn keiner der Prozesse es mehr benutzt. Umgekehrt kann es auch schon vor dem Ende des erzeugenden Prozesses geschlossen werden, wenn es innerhalb des Prozesses nicht mehr nutzbar ist; dadurch werden Ressourcen in der Ablaufumgebung freigegeben. Dazu wird die Schließfunktion an das interne Objekt *aWindow* gebunden, welche die Ressource im Objektsystem repräsentiert:

```
let aWindow = openWindow(...)
let freeWindow(x :Window) = closeWindow(x)
assignFreeFunction(aWindow freeWindow)
```

Durch *assignFreeFunction* wird dem Objekt *aWindow* eine Funktion *freeWindow* zugeordnet, die das freizugebende Objekt als Parameter erhält und somit die Freigabe der damit verbundenen Ressource erlaubt. Das Objekt wird als Parameter übergeben und somit dynamisch gebunden, weil eine statische Bindung in der Freigabefunktion *freeWindow* wiederum die Freigabe verhindern würde, denn das Objekt wäre dadurch noch gebunden und somit nutzbar.

Eine generische Ressourcenfreigabe initiiert also anwendungsabhängige Funktionen bei der Freigabe von Objekten. An jedes Objekt kann eine Funktion gebunden werden, die vom Speicherfreigabemechanismus aufgerufen wird, wenn das Objekt freigegeben werden soll. Um die Funktion während der Speicherfreigabe ausführen zu können, muß ein Prozeßobjekt existieren bzw. erzeugt werden, das die Funktion ausführt.

Im Kontext dieses Prozesses sind weitere Operationen vor der endgültigen Freigabe des Objekts möglich, beispielsweise eine Benutzerinteraktion, die vor der Freigabe eine Sicherung des Fensterinhalts durchführt.

```
let freeWindow(w :Window) =
begin
  let answer = popup(``Save before closing?'' array ``Yes'' ``No'' end)
  if answer == ``Yes'' then save(getContents(w)) end
  closeWindow(w)
end
```

Durch die Freigabefunktion kann das Objekt auch wieder erreichbar gemacht werden. Beispielsweise könnten alle nicht mehr erreichbaren Fenster gesammelt werden, um sie erst zu einem bestimmten Zeitpunkt gemeinsam zu schließen, weil sie bis dahin noch für den Benutzer wichtige Informationen visualisieren:

```
let freeWindow(w :Window) = registerWindowForClosing(w)
...
foreach w in registeredWindows() do closeWindow(w) end
```

Die Freigabefunktion *freeWindow* wird von der Speicherverwaltung aufgerufen, wenn das Objekt *w* nicht mehr erreichbar, also nicht mehr nutzbar ist. Die Freigabefunktion stellt eine Bindung her, durch die das Objekt wieder erreichbar und somit nutzbar wird. Also darf es dann nicht mehr freigegeben werden. D.h. die Speicherverwaltung prüft nach dem Aufruf der Freigabefunktion, ob das Objekt jetzt immer noch nicht erreichbar ist und gibt es nur in diesem Fall endgültig frei.

3.2 Persistente Objektsysteme

Objektsysteme unterstützen langlebige Vorgänge in der Anwendungswelt. Persistente Objektsysteme erlauben es, die Lebensdauer des Objektsystems der Dauer der Vorgänge in der Anwendungswelt anzupassen [Sessions 96]. Persistenz bezeichnet die verlängerte Lebensdauer von Objekten über die Lebensdauer der Instanz des sie erzeugenden und interpretierenden Ablaufsystems hinaus (vgl. Abschnitt 3.2.1). Realisiert wird Persistenz über Speicherungssysteme, die eine persistente Repräsentation des Objektsystems ablegen (vgl. Abschnitt 3.2.2). Bei der Wiederherstellung von Vorgängen (Prozeßobjekten) sind insbesondere die Seiteneffekte zu behandeln (vgl. Abschnitt 3.2.3).

3.2.1 Lebensdauer von Objekten

Objektsysteme existieren höchstens so lange wie das Ablaufsystem, das sie interpretiert. Sie werden spätestens in dem Moment freigegeben, in dem das Ablaufsystem heruntergefahren wird.

Eine Anforderung an Objektsysteme, insbesondere an Informationssysteme im kommerziellen Umfeld mit langwährenden Geschäftsprozessen [Mathiske 96], ist die Dauerhaftigkeit von Objekten, um sie wiederholt nutzen zu können. Daten werden wiederholt verarbeitet, Funktionen wiederholt abgearbeitet, Prozesse unterbrochen und wieder fortgesetzt. Dabei wird ein Zustand der Objekte festgehalten und zu einem beliebigen Zeitpunkt wiederhergestellt. Die Objekte und die von ihnen belegten Ressourcen können zwischenzeitlich freigegeben werden. Das Ablaufsystem, das die Objekte interpretiert, kann heruntergefahren und durch eine andere Instanz ersetzt werden.

Persistenz bedeutet also, daß ein Objekt beliebig lange leben und damit genutzt werden kann. Inbesondere leben persistente Objekte länger als die Instanz des Ablaufsystems, das sie interpretiert. Die von persistenten Objekten während des Ablaufs genutzten Ressourcen können zwischenzeitlich anderweitig genutzt werden.

Nicht alle Objekte müssen persistent sein. Objekte, deren Lebenszyklus beendet ist und die nicht zur Dokumentation oder Reproduktion o.ä. benötigt werden, sollen keine Ressourcen mehr verbrauchen; sie werden endgültig freigegeben. Beispiele sind Daten, die nur Zwischenergebnisse darstellen, Funktionen, die nur einmalig abgearbeitet werden, oder Prozesse, die abgelaufen sind und ihren Zweck erfüllt haben.

Objekte, deren Lebenszyklus nicht beendet ist, die also für eine Anwendung weiterhin benötigt werden, sollen so lange leben (persistent sein), wie diese Anwendung besteht. Dies gilt für alle Arten von Objekten:

- Daten, auf die später zurückgegriffen werden soll: Wenn diese Daten verloren gingen, müßten sie neu erfaßt oder berechnet werden, wenn dies überhaupt möglich ist. Die Aufnahme von Meßwerten eines Vorgangs kann beispielsweise nicht wiederholt werden, die manuelle Eingabe von Daten ist sehr kostspielig und ohne die Ausgangsdaten, die als Zwischenergebnisse im obigen Sinne freigegeben werden, können Berechnungen nicht wiederholt werden.

- Funktionen, die wiederholt ausgeführt werden sollen: Wenn die ablauffähige Funktion nicht persistent abgelegt ist, muß sie wieder erzeugt werden, wozu eine Entwicklungsumgebung notwendig ist. Wenn die Funktion nicht in einer von einem Entwicklungssystem verarbeiteten Repräsentation vorliegt, muß sie neu implementiert werden, bevor sie wieder ausgeführt werden kann. Dieser Entwicklungsprozeß ist bei komplexen Anwendungen sehr aufwendig, teuer und fehlerträchtig.

- Prozesse, die fortgesetzt werden sollen: Wenn der Prozeß nicht fortgesetzt werden kann, muß er von vorne beginnen. Der gesamte Ablauf des Prozesses muß nachvollzogen werden. Objekte, die von diesem Prozeß gebunden oder erzeugt wurden, müssen wieder gebunden oder neu erzeugt werden. Entscheidungen, die getroffen wurden, müssen erneut getroffen werden. Dabei kann es vorkommen, daß Daten oder Funktionen, die im Laufe des Prozesses zeitweise benutzt wurden, nicht mehr verfügbar sind, und somit Abläufe, Entscheidungen oder Berechnungen nicht nachvollzogen werden können.

Datenbanken werden zur Speicherung von Daten [Lockemann, Schmidt 87] und Filesysteme oder Repositories für die Speicherung von Funktionen [Bernstein 98] benutzt. Orthogonal persistente Objektsysteme bieten auch Persistenz für Prozesse [Matthes, Schmidt 94; Mathiske 96].

3.2.2 Persistente Repräsentation eines Ablaufzustands

Persistenz wird durch Speicherungssysteme realisiert [Matthes et al. 96; Müller iV]. Das Speicherungssystem legt die Objekte in einer persistenten Repräsentation ab (vgl. Abbildung 3.4), z.B. einer Linearisierung eines Objektgraphen [Herlihy, Liskov 82; Matthes et al. 96]. Um ein Objektsystem zu speichern, werden die Objekte und Bindungen aus der ablauffähigen in die persistente Repräsentation überführt. Bei einer Reaktivierung des Objektsystems wird aus der persistenten Repräsentation die ablauffähige Repräsentation wiederhergestellt. Gegenüber der ablauffähigen Repräsentation abstrahiert die persistente Repräsentation von den Eigenschaften der Instanz des Ablaufsystems, welches das Objektsystems interpretiert, beispielsweise von Hauptspeicheradressen.

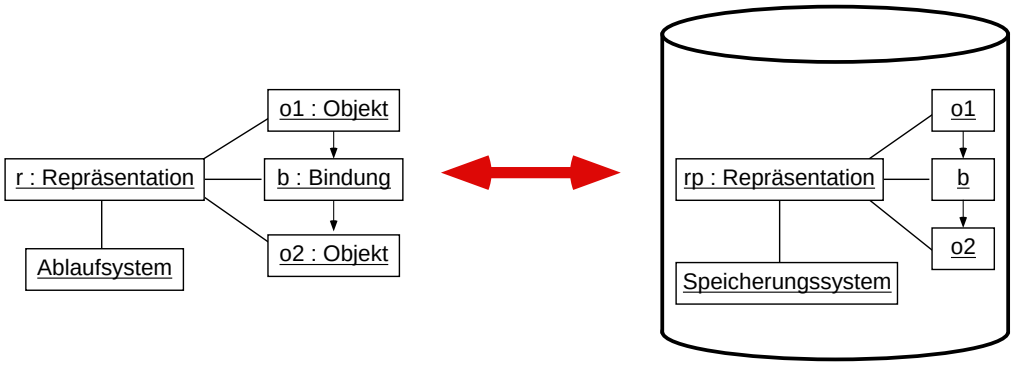

Abbildung 3.4: Speicherungssystem und persistente Repräsentation

Bei der Umwandlung sind Objekte und Bindungen zwischen Objekten zu betrachten [Matthes, Schmidt 94]:

Objekte: Objekte enthalten formatierte Bitmuster, die teilweise implizit voneinander abhängen. Die Bitmuster und semantischen Abhängigkeiten sind verlustfrei wiederherzustellen.

Formate: Ablaufsysteme interpretieren eine prinzipiell beliebig erweiterbare Menge von Bitmustern, beispielsweise für Zahlen in unterschiedlicher Ausprägung (z.B. Ganz-, Fließkomma-, komplexe Zahlen) und Genauigkeit, Zeichen (z.B. Zeichensatz gemäß ASCII oder ISO-LATIN) und Zeichenketten (z.B. Endekennung oder Länge), Funktionen (z.B. Bytecode, Maschinencode) und Prozesse (z.B. Register, Ablaufhistorie). Das Speicherungssystem sollte die Menge an speicherbaren Bitmustern nicht einschränken, beispielsweise durch vorgegebene Formate oder Längenrestriktionen. Allerdings kann bei der Speicherung ein anderes Format verwendet werden, beispielsweise zur Fehlererkennung mit Hilfe von Prüfsummen oder zur Speicherersparnis durch Kompression, solange eine verlustfreie Rückumwandlung gewährleistet ist.

Abhängigkeiten: Objekte enthalten in ihren Bitmustern teilweise implizite Abhängigkeiten von anderen Objekten, beispielsweise ein Objekt, das die Länge eines anderen Objektes vermerkt, oder ein Prozeß, der als Bestandteil einen Verweis auf die nächste auszuführende Anweisung einer Funktion enthält. Diese semantischen Abhängigkeiten gelten nicht in der persistenten Repräsentation, wenn andere Formate zur Speicherung als zum Ablauf verwendet werden. Sie sollten nach der Rückumwandlung der Objekte in die ablauffähige Repräsentation wieder gelten.

Von der Instanz des Ablaufsystems und der ablauffähigen Instanz des Objektsystems abhängige Angaben sind dagegen nicht wiederherstellbar, beispielsweise der interne Prozeßidentifikator des Ablaufsystems für ein Prozeßobjekt oder der Ablageort (Hauptspeicheradresse) eines Objekts.

Es wird hierbei davon ausgegangen, daß die ablauffähige Ausgangsrepräsentation wiederhergestellt wird. Bei heterogenen Ablaufsystemen (vgl. Kapitel 4) wird aus der persistenten Repräsentation unter Umständen eine andere ablauffähige Repräsentation als die Ausgangsrepräsentation wiederhergestellt [Shub 90; Steensgard, Jul 95; Redhead 96]. In diesem Fall ist Portabilität die Hauptanforderung, so daß die Verlustfreiheit nicht unbedingt als Forderung aufrechterhalten wird; beispielsweise werden Genauigkeitsverluste bei Fließkommazahlen toleriert.

Bindungen: Aggregierte Objekte bilden komplexe Bindungsstrukturen. Die Bindungen zwischen den Objekten sind strukturerhaltend wiederherzustellen [Matthes et al. 96].

Bindende und gebundene Objekte: Objekte können an beliebig vielen Bindungen teilnehmen: Sie können beliebig oft gebunden werden; aggregierte Objekte können beliebig viele Objekte binden. Dabei gibt es keine Einschränkungen bezüglich Art, Größe, Format, Inhalt o.ä. der Objekte. Nach der Rückumwandlung dürfen keine ins Leere zeigenden Bindungen (*dangling references*) existieren, d.h. die gebundenen Objekte müssen mit den Bindungen wiederhergestellt werden. Außerdem muß auch nach der Rückumwandlung die Identifikation der Bindungen wieder gelten, beispielsweise über die Reihenfolge der Bindungen, wenn auf diese in Funktionen Bezug genommen wird; dies ist auch eine semantische Abhängigkeit.

Strukturen: Die durch Bindungen gebildeten Strukturen enthalten Muster, die bei der Abbildung und Rückumwandlung besonders zu beachten sind, insbesondere gemeinsam genutzte Objekte und rekursive Bindungen. Bindungen an das gleiche Objekt werden in der persistenten Repräsentation unter Umständen durch mehrere Paare von Objekten ersetzt. Sie sollen auch nach der Rückumwandlung das gleiche Objekt und nicht zwei unterschiedliche, wenn auch gleich aufgebaute Objekte binden. Rekursive Bindungen binden (evtl. transitiv) das bindende Objekt, beispielsweise bei rekursiven Funktionen. Während der Abbildung auf eine persistente Repräsentation wird diese Rekursion beispielsweise bei linearen Repräsentationen [Herlihy, Liskov 82; Huang, Lamb 94; Mathiske 96] aufgebrochen, aber während der Rückumwandlung wiederhergestellt.

Die persistente Repräsentation eines Objektsystems wird dauerhaft abgelegt und zu einem beliebigen Zeitpunkt reaktiviert. Die Reaktivierung kann im Kontext eines anderen Ablaufsystems erfolgen. Die persistente Repräsentation darf daher nicht abhängig von vorübergehenden Eigenschaften der Inkarnationen des Objektsystems und des Ablaufsystems sein, z.B. durch Hauptspeicheradressen, die nach einer Reinkarnation ungültig sind.

Objekte, für die keine persistente Repräsentation existiert, werden transiente Objekte genannt. Für diese Objekte wird eine Beschreibung persistent abgelegt, wie dieses Objekt wiederherzustellen ist, z.B. durch dynamische Erzeugung. In diesem Fall müssen entweder bei der Umwandlung oder spätestens in dem Moment, in dem das Objekt von dem Interpreter benötigt wird, die Aktionen ausgeführt werden, die das Objekt erzeugen.

Objekte können auch getrennt persistent abgelegt werden, beispielsweise in unterschiedlichen Speicherungssystemen für Daten und Funktionen. Dabei gehen dann allerdings die Bindungen

zwischen den Objekten verloren. Sie müssen bei der Wiederherstellung der Objekte oder während des Ablaufs explizit neu aufgebaut werden.

Wenn die Repräsentation, mit der die Ablaufumgebung eines Objektsystems arbeitet, persistent ist, muß keine Umwandlung in einer persistente Repräsentation durchgeführt werden. In diesem Fall wird nur das Objektsystem in seinem gegenwärtigen Zustand persistent abgelegt und kann später reaktiviert werden [Matthes et al. 96; Müller iV].

Wenn eine Umwandlung stattfindet, muß diese eine eindeutige Semantik besitzen. Bei der Umwandlung in die persistente Repräsentation kann es mehrere alternative Repräsentationen geben, aus der eine ausgewählt wird, evtl. kontextabhängig verschiedene. Wenn zwei unterschiedliche Teile der Ausgangsrepräsentation in ein Element der persistenten Repräsentation überführt werden, ist keine eindeutige Rückführung mehr möglich und geht somit Semantik verloren, wenn nicht zusätzlich beschreibende Informationen mit abgespeichert und bei der Reaktivierung (reflektiv) ausgewertet wird.

3.2.3 Wiederherstellung von Ablaufzuständen

Prozeßobjekte beschreiben den Zustand einer vom Ablaufsystem durchgeführten Interpretation von Funktionsobjekten. Der Zustand ergibt sich aus der zeitlichen Abfolge der ausgeführten Operationen. Der Ablaufzustand besteht zumindest aus einem Verweis auf die nächste auszuführende Anweisung einer Funktion, die Aufrufhierarchie von Funktionen mit Verweisen auf die nach der Rückkehr in die einzelnen Funktionen auszuführende nächste Anweisung, alternativ auch Fortsetzungsfunktionen (*continuations*) [Maes 87; Cejtin et al. 95; Gawecki, Matthes 96a], sowie die im Prozeßablauf getätigten Bindungen.

Außerdem gehören zum Prozeßzustand die Zustände der Basisobjekte des Ablaufsystems, welche die Interpretation beeinflussen oder durch das Objektsystem manipuliert oder abgefragt werden können. Dies sind beispielsweise Registerinhalte, Schaltereinstellungen, die das Verhalten des Ablaufsystems beeinflussen, oder die verstrichene Prozeßlaufzeit. Diese Objekte sind ebenfalls persistent abzulegen und zu rekonstruieren.

Während des Ablaufs werden auch Seiteneffekte ausgelöst. Zur Wiederherstellung des Ablaufzustands gehört daher auch eine (teilweise) Wiederholung der Seiteneffekte [Kornacker 95]. Die Seiteneffekte sind während des Ablaufs zu erfassen und nach der Dauer ihrer Gültigkeit bzw. ihres Beitrags zur Ablauffähigkeit des Prozesses zu kategorisieren (vgl. Abbildung 3.5):

Einmalige Seiteneffekte werden danach unterschieden, ob sie eine Wirkung auf den weiteren Ablauf des Prozesses besitzen. Beispielsweise tragen die Ausgabe eines Warntons oder der Ausdruck einer Seite nicht zur Ablauffähigkeit des Ablaufsystems bei und sind daher nicht zu wiederholen. Hingegen ist das Einschalten eines externen Gerätes zu wiederholen, wenn weitere Seiteneffekte darauf Bezug nehmen, denn ansonsten ist der weitere Ablauf des Prozesses nicht gewährleistet.

Paarweise auftretende Seiteneffekte bestehen aus einem öffnenden und einem schließenden Seiteneffekt. Wenn der schließende Seiteneffekt noch nicht aufgetreten ist, muß der öffnende Seiteneffekt wiederholt werden. Sobald der schließende Seiteneffekt auftritt, ist damit der Effekt des öffnenden Seiteneffekts aufgehoben und somit eine Wiederholung der beiden Seiteneffekte sinnlos. Beispielsweise stellt das Belegen einer Ressource einen öffnenden Seiteneffekt dar, die Freigabe den schließenden. Wenn die Freigabe noch nicht erfolgt ist, muß die Ressource beim Wiederherstellen des persistenten Zustands wieder belegt werden.

Folgen von Seiteneffekten ergeben sich, wenn Seiteneffekte in einer bestimmten Reihenfolge auftreten müssen. Dabei kann es sich um einmalige Seiteneffekte mit langfristiger Wirkung und paarweise auftretende Seiteneffekte handeln. Bei der Wiederholung der Seiteneffekte ist die Reihenfolge zu beachten. Paarweise auftretende Seiteneffekte, deren schließender Seiteneffekt schon aufgetreten ist, werden aus der Folge herausgenommen. Wenn die Folge abgeschlossen ist, wird sie nicht wiederholt, d.h. es ergibt sich ein abgeschlossener Gültigkeitsbereich für

alle einmaligen Seiteneffekte in der Folge; beim Schließen der Folge müssen auch alle paarweise auftretenden Seiteneffekte geschlossen sein. Beispielsweise stellen die Interaktionen mit einem Benutzer über eine graphische Benutzerschnittstelle eine Seiteneffektfolge dar, wobei die Reihenfolge der zu zeichnenden Bildschirmelemente einzuhalten ist, aber zwischenzeitlich geöffnete und inzwischen wieder geschlossene Fenster zu ignorieren sind.

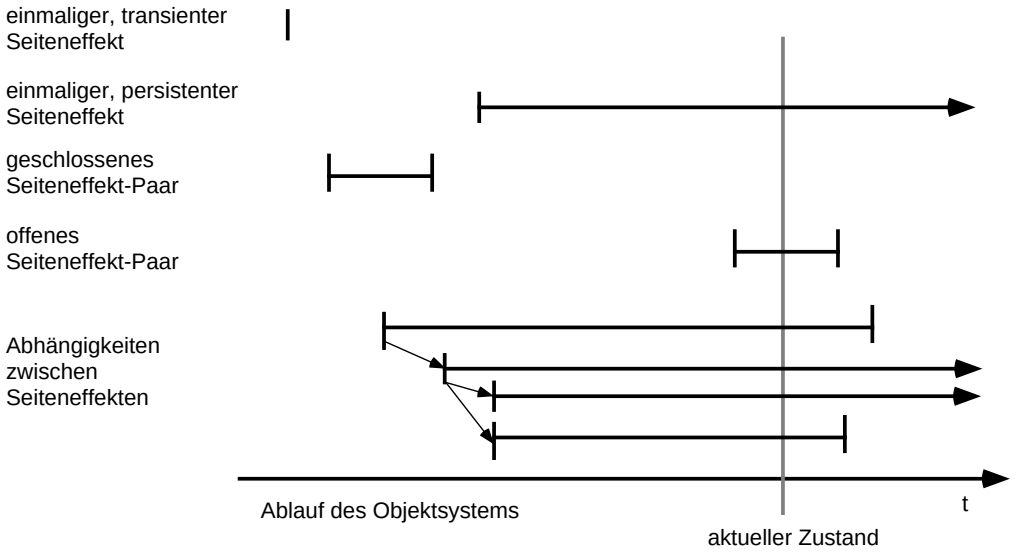

Abbildung 3.5: Persistenz von Seiteneffekten

Um die Seiteneffekte bei der Wiederherstellung eines Prozeßobjektes zu wiederholen, werden während des Ablaufs reflektiv Funktionen als Basiobjekte registriert (und deregistriert bei kompensierenden Seiteneffekten), welche die Seiteneffekte ausführen [Redhead 96]. Diese Funktionen werden persistent abgelegt und während der Wiederherstellung ausgeführt. Dieses Prinzip entspricht dem der Freigabe von Ressourcen, wie in Abschnitt 3.1.4 gezeigt.

3.3 Evolution von Objektsystemen

Objektsysteme leben so lange wie die Anwendungen, die sie unterstützen. Diese Anwendungen sind nicht statisch, sondern ändern sich dynamisch [Hofmeister, Purtilo 93; Nierstrasz, Meijler 95; Shaw, Garlan 96] während der Lebenszeit des Objektsystems, insbesondere bei langlebigen (persistenten) Objektsystemen [Atkinson et al. 93; Kirby et al. 96]. Beispielsweise werden Fehler korrigiert, die Leistung verbessert oder geänderte bzw. erweiterte Anforderungen erfüllt [Müller 97]. Die Änderungen stellen sich als Ergänzung, Streichung oder Austausch von Informationen, Abläufen und Vorgängen einer Anwendung dar [van der Hoek et al. 97]. Dieser Vorgang wird als Evolution der Anwendung bezeichnet, wenn auf eine existierende Anwendung aufgebaut und diese schrittweise angepaßt wird [Sametinger 97]. Im Gegensatz dazu besitzt die Entwicklung einer Anwendung nicht unbedingt einen Bezug zu einer existierenden Anwendung. Evolution bedeutet die Fortsetzung der Entwicklung (vgl. Kapitel 5) während des Ablaufs. Wie bei der Entwicklung ist eine entscheidende Forderung die Sicherstellung der Ablauffähigkeit. Die evolutionäre Weiterentwicklung und Rekonfiguration wird durch Konfigurationsmanagement unterstützt [Warren, Sommerville 95; Zeller 95; van der Hoek et al. 97]. Die Rekonfiguration wird reflektiv durch das Objektsystem angestoßen, wenn es durch das Ablaufsystem Änderungen der Umgebung gemeldet bekommt [Cheung, Loong 95; Noble, Satyanarayanan 95].

Wenn die von einem Objektsystem unterstützte Anwendung sich ändert, ändert sich auch das

Objektsystem. Beispielsweise werden Daten ergänzt oder Funktionen ausgetauscht. Das Objektsystem durchläuft also eine Evolution. Diese Evolution findet während seines Ablaufs statt. Wenn stattdessen der Ablauf des Objektsystems abgebrochen und durch das Hochfahren eines geänderten oder neu entwickelten Objektsystems ersetzt wird, gehen alle dynamisch während des Ablaufs erzeugten Objekte und Bindungen verloren. Der Ablauf wird wiederholt, d.h. Daten müssen neu erfaßt und Berechnungen erneut durchgeführt werden. Der Verlust dieser geleisteten Arbeit ist meistens nicht tragbar, die Fortsetzung der Arbeit mit einem nicht zur Anwendung passenden Objektsystem aber auch nicht. Daher wird das Objektsystem mit der Anwendung im laufenden Betrieb weiterentwickelt [Hofmeister, Purtilo 93].

Evolution bedeutet, daß Objektzustände oder Bindungen zwischen Objekten eines Objektsystems manipuliert werden (vgl Abbildung 3.6).

- Evolution durch Änderung eines Objektzustands erhält die Identität des Objekts und macht die Änderung über alle Bindungen zugreifbar; allerdings bestehen seitens des Ablaufsystems teilweise Einschränkungen, so daß beispielsweise die Art oder Größe des Objekts nicht geändert werden kann.

- Evolution durch Änderung der Bindungen an ein Objekt erlaubt eine partielle Evolution, d.h. nicht alle Bindungen werden geändert, so daß das alte Objekt erhalten und nutzbar bleibt. Diese Änderung der Bindung wird im Kontext maschinennaher Architekturen auch Rebindung (*rebinding*) genannt [Blaaw, Brooks, Jr. 97; Ramme 98].

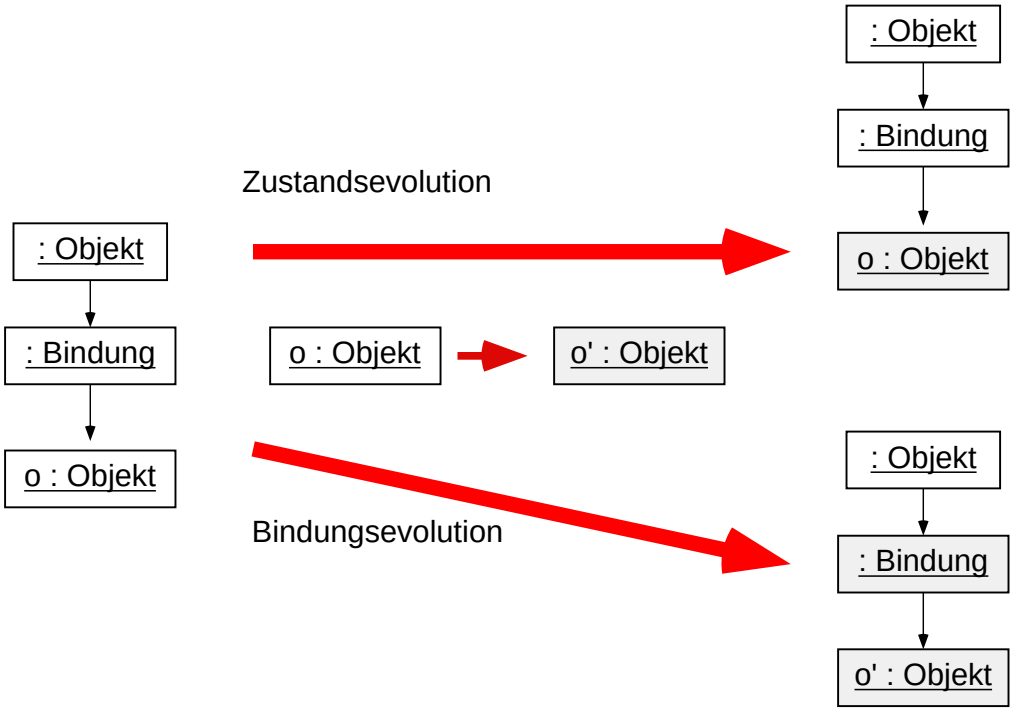

Abbildung 3.6: Zustands- und Bindungsevolution

Die Entwicklung eines Objektsystems hat die Ablauffähigkeit des Objektsystems und die Vermeidung bestimmter Ablauffehler zum Ziel (vgl. Kapitel 5). Das erstellte Objektsystem erfüllt Integritätsbedingungen und sichert ihre Erhaltung während des Ablaufs zu. Evolution darf die Ablauffähigkeit des Objektsystems nicht gefährden und während der Entwicklung zugesicherte Integritätsbedingungen nicht invalidieren (vgl. Abschnitt 3.3.1). Um Evolution zu ermöglichen,

welche die Ablauffähigkeit und Integrität des Objektsystems erhält, werden verschiedene Evolutionsmechanismen eingesetzt (vgl. Abschnitt 3.3.2).

3.3.1 Ablauffähigkeit und Integrität

Die Ablauffähigkeit und Integrität des Objektsystems darf durch Evolution nicht gefährdet werden. Uneingeschränkte Evolution führt potentiell zu Ablauffehlern [Bennett 90; Gosling 95; Kirby et al. 96]. Folgende Faktoren schränken die Substituierbarkeit von Objekten grundsätzlich oder zeitweise ein (vgl. auch [Senivongse, Utting 96]):

Struktur: Durch Evolution darf die Struktur des Objektsystems nicht so verändert werden, daß Operationen auf Objekten ausgeführt werden, auf denen sie nicht definiert sind.

Ablaufzustand: Der aktuelle Ablaufzustand der von einem Prozeß ausgeführten Funktionen darf durch die Evolution nicht geändert werden.

Ablaufhistorie: Die während eines Ablaufs eines Objektsystems durch Seiteneffekte und Objektmanipulationen geschaffenen Bedingungen dürfen durch Evolution nicht invalidiert werden.

Nebenläufigkeit: Prozesse, die unter der Annahme entwickelt werden, daß ihr Ablauf nicht durch nebenläufige Prozesse beeinflußt wird, dürfen nicht durch nebenläufige Evolution beeinträchtigt werden.

Erzeugung: Optimierungen, die das Entwicklungssystem bei der Erzeugung des Objektsystems vornimmt, dürfen nicht durch Evolution ungültig werden.

Entwicklung: Die von einem Entwickler während der Entwicklung getroffenen Annahmen zur semantischen Integrität des Objektsystems dürfen durch Evolution nicht gefährdet werden.

Anwendung: Abhängig von der Semantik des Anwendungsbereiches, den das Objektsystem unterstützt, sind nur bestimmte Evolutionen sinnvoll.

Struktur

Auf Objekten sind Operationen definiert. Die Operationen sind abhängig von der Art des Objektes, beispielsweise ist die Operation "Aufrufen" auf Funktionen, nicht aber auf Daten oder Prozessen definiert. Daher kann ein Objekt einer Art nicht durch ein Objekt einer anderen Art ersetzt werden, beispielsweise ein Funktionsobjekt durch ein Datenobjekt, denn die Operationen sind auf diesem Objekt nicht definiert.

Aber auch innerhalb einer Objektart bestehen Unterschiede zwischen den Objekten. Beispielsweise unterscheiden sich Funktionen nach Anzahl und Art der Parameter und des Rückgabewertes. Daher können auch Objekte einer Art nicht beliebig gegeneinander ausgetauscht werden, beispielsweise eine Funktion, die einen Parameter erwartet, gegen eine, die zwei Parameter erwartet, denn beim Aufruf würde der zweite Parameter undefiniert sein. Andererseits kann es durchaus sein, daß das Ablaufsystem eine Funktion, die einen Parameter erwartet, mit zwei Parametern aufrufen kann, ohne die Ablauffähigkeit einzuschränken; der zweite Parameter wird dann ignoriert. Zur Substituierbarkeit gehören abhängig vom Ablaufsystem noch weitere Einschränkungen, beispielsweise die von einer Funktion ausgelösten Ausnahmen oder die von einem Prozeß akzeptierten Signale.

Das Ablaufsystem schränkt also die Substituierbarkeit von Objekten ein. Diese Einschränkungen werden beispielsweise mit Typsystemen erfaßt und überprüft (vgl. Kapitel 5).

Ablaufzustand

Der Ablaufzustand ergibt sich aus der zu einem Zeitpunkt ausgeführten Funktion und der Aufrufhierarchie von Funktionen bis zur primären Funktion des Prozesses. Diese in Ausführung befindlichen Funktionen können nicht ohne weiteres ersetzt werden.

```
let f() = ...
let g() = begin f() f() end
g()
```

Die Funktion *g* wird in einem Prozeß ausgeführt. Bei jedem Aufruf der Funktion *f* wird in dem Prozeß vermerkt, an welcher Stelle die Funktion *g* fortzusetzen ist. Die Funktion *g* soll nun durch die Funktion *g'* ersetzt werden, die nur noch einen Aufruf der Funktion *f* enthält.

```
let g'() = begin f() end
```

Wenn die Ausführung der Funktion *g* schon über den ersten Aufruf der Funktion *f* hinausgelangt ist, kann die Funktion nicht mehr durch *g'* ersetzt werden, denn eine Fortsetzung von *g'* an einer *g* entsprechenden Stelle ist nicht mehr möglich.

Eine Ersetzung einer gerade von einem Prozeß ausgeführten Funktion ist also nur möglich, wenn beide Funktionen (neben einer passenden Signatur) über einen identischen Präfix der Anweisungssequenz verfügen und die Ausführung des Prozesses sich noch in dieser gemeinsamen Anfangssequenz befindet.

Ablaufhistorie

Die Substituierbarkeit von Objekten ist abhängig von der während des Ablaufs erfolgten Nutzung der Objekte, also der Ablaufhistorie. Als Beispiel dient ein (aggregiertes) Objekt, das eine Zeichenkettenverarbeitung realisiert (vgl. Abbildung 3.7). Zeichenketten sind hier abstrakte Datenobjekte, die dynamisch von der Zeichenkettenverarbeitung erzeugt werden und auf denen Verarbeitungsfunktionen definiert sind.

```
interface String
import int char
export
  T <:Ok
  new(size :int.T  init :char.T) :T
  size(string :T) :int.T
  get(string :T  index :int.T) :char.T
  set(string :T  index :int.T  element :char.T) :Ok
  concat(string1 :T  string2 :T) :T
end
```

Abbildung 3.7: Schnittstelle der Zeichenkettenverarbeitung

In einer Implementation (vgl. Abbildung 3.8) werden Zeichenketten als Datenobjekte repräsentiert, die aus einer Länge und den einzelnen Zeichen der Zeichenkette bestehen. Die einzelnen Zeichen werden über einen Index adressiert, der bei 1 beginnt. Auf dieser Repräsentation sind Funktionen zur Erzeugung und Manipulation der Zeichenketten definiert.

Diese Implementation wird benutzt, um eine Zeichenkette *aString* mit zwei Zeichen zu erzeugen und das zweite Zeichen durch den Buchstaben *'a'* zu ersetzen:

```
let aString = string0.new(2 '_')
string0.set(aString 2 'a')
```

Eine weitere Implementation der Schnittstelle (vgl. Abbildung 3.9) legt die Länge hinter dem Inhalt statt davor ab. Solange noch keine Zeichenkette erzeugt wurde, können die beiden Implementationen gegeneinander ausgetauscht werden, weil sie die gleiche Schnittstelle implementieren. Allerdings darf das Objekt nur als ganzes ausgetauscht werden, mit allen Funktionsobjekten; es dürfen nicht einzelne Funktionsobjekte des aggregierten Objektes anderweitig gebunden bleiben. Dies wird beispielsweise durch Kontexte (vgl. Abschnitt 3.3.2) zugesichert.

```
implementation string0 :String
import int char array
export
  Let T = Tuple size :Int  content :array.T(char.T) end
  let new(size :int.T  init :char.T) :T =
    tuple size array.new(size init) end
  let size(string :T) :int.T = string.size
  let get(string :T  index :int.T) :char.T =
    array.get(string.content int.dec(index))
  let set(string :T  index :int.T  element :char.T) :Ok =
    array.set(string.content int.dec(index) element)
  let concat(string1 :T  string2 :T) :T =
  begin
    let newString = new(int.add(size(string1 size(string2))
    for i = 1 upto size(string1) do
      set(newString i get(string1 i))
    end
    for i = 1 upto size(string2) do
      set(newString int.add(size(string1) i) get(string2 i))
    end
    newString
  end
end
```

Abbildung 3.8: Implementation der Zeichenkettenverarbeitung

```
implementation string1 :String
import int char array
export
  Let T = Tuple content :array.T(char.T)  size :int.T end
  ...
end
```

Abbildung 3.9: Geänderte Repräsentation der Zeichenkettenverarbeitung

Wenn schon Zeichenketten erzeugt wurden, entsteht durch den Austausch des Objektes eine Inkonsistenz. Weil die Repräsentation der Zeichenkette sich ändert, können vorhandene Werte nicht mehr weiterverarbeitet werden. Die Änderung des Repräsentationstyps des abstrakten Datentyps zieht also eine Inkompatibilität nach sich, so daß die beiden Implementationen nicht mehr gegeneinander substituierbar sind, obwohl sie die gleiche Schnittstelle implementieren. In diesem Fall ist eine Transformation der Zeichenketten nötig (vgl. Abschnitt 3.3.2).

Nebenläufigkeit

Evolution findet nebenläufig zum Ablauf des Objektsystems statt. Wie bei nebenläufigen Prozessen sind daher Ablaufabhängigkeiten zwischen dem ändernden und dem geänderten Prozeß zu beachten.

```
let var a = 1
par
  x: begin a := 2  ...  a := a + 1  ...  2 * a  ... end
  y: begin a := 3  ...  a := a - 1  ...  2 * a  ... end
end
```

Die Bindung *a* wird von den beiden nebenläufigen Prozessen *x* und *y* verändert. Je nachdem, in welcher Reihenfolge bzw. Verzahnung diese Prozesse ausgeführt werden, ergeben sich andere Ergebnisse [Jessen, Valk 87]. Um serialisierbare Abfolgen [Jessen, Valk 87; Lockemann, Schmidt 87] zu erreichen, werden Sperren eingesetzt.

```
let var a = 1
par
  x: begin lock(a) ...  unlock(a) end
  y: begin lock(a) ...  unlock(a) end
end
```

Durch die Sperre *lock(a)* kann nur ein Prozeß, *x* oder *y*, die Bindung *a* ändern, so daß der andere auf die Freigabe der Sperre *unlock(a)* warten muß.

Wenn nur ein Prozeß auf Objekten oder Bindungen arbeitet, sind Sperren nicht nötig.

```
let var a = 1
a := 2  ...  a := a + 1  ...  2 * a  ...
```

Da Evolution einen nebenläufigen Prozeß darstellt, der auf die Bindung *a* ändernd zugreifen kann, sind auch hier Sperren notwendig, um eine integritätserhaltende Abarbeitung zu gewährleisten. Dabei ist zu unterscheiden zwischen prinzipiell nutzbaren, also durch eine Bindung zugänglichen, und aktuell genutzten Objekten. Da für einen Prozeß und seine primäre Funktion alle Objekte des Objektsystems nutzbar sind (vgl. Objektfreigabe in Abschnitt 3.1.4), ist die Menge der aktuell genutzten Objekte auf einer niedrigeren Granularititätsebene explizit durch Transaktionsklammern [Lockemann, Schmidt 87] o.ä. festzulegen.

Erzeugung

Zur Prüfung und Optimierung wird während der Entwicklung zwischen veränderlichen und unveränderlichen Bindungen unterschieden. Eine der Evolution zugängliche Bindung nimmt eine Zwischenstellung ein [Ramme 98].

```
let var a = 1
a := 2
let b = 2 * a
```

Die Bindung *a* ist veränderlich und kann mit dem Zuweisungsoperator := geändert werden. Da während der Entwicklung (außer durch Einsatz von Datenflußanalysetechniken) nicht bekannt ist, welcher Wert durch *a* gebunden wird, wird der Ausdruck *2 * a* immer evaluiert. Eine Evolution der Bindung *a* entspricht damit einer (nebenläufigen) Zuweisung.

```
let c = 3
(* c := 4  nicht erlaubt *)
let d = 2 * c
```

Die Bindung c ist unveränderlich, d.h. der Zuweisungsoperator $:=$ kann nicht angewendet werden. Damit ist der Wert von c während der Entwicklung festgelegt. Ein Optimierer kann daher den Wert von d vorausberechnen.

Die Vorausberechnung des Wertes wird ungültig, wenn die Bindung c durch Evolution während des Ablaufs geändert wird. In diesem Fall muß auch der Wert von d neu berechnet werden, d.h. c ist wie eine veränderliche Bindung zu behandeln. Wenn die Optimierung beibehalten werden soll, ist die Evolution der Bindung c nicht möglich, d.h. es werden während der Entwicklung bestimmte Objekte und Bindungen von einer möglichen Evolution ausgeschlossen.

Entwicklung

In einer weiteren Implementation der Zeichenkettenverarbeitung (vgl. Abbildung 3.10) bleibt die Repräsentation erhalten, aber die Semantik der Funktionen, die auf dieser Repräsentation arbeiten, ändert sich: Der Index beginnt nun bei 0. Die Implementation ändert sich nur insofern, als daß Indexmanipulationen wegfallen bzw. hinzukommen.

```
implementation string2 :String
import int char array
export
  ...
  let get(string :T  index :int.T) :char.T =
    array.get(string.content index)
  let set(string :T  index :int.T  element :char.T) :Ok =
    array.set(string.content index element)
  let concat(string1 :T  string2 :T) :T =
  begin
    ...
    for i = 0 upto int.dec(size(string1)) do
      set(newString i get(string1 i))
    end
    for i = 0 upto int.dec(size(string2)) do
      set(newString int.add(size(string1) i) get(string2 i))
    end
    ...
  end
end
```

Abbildung 3.10: Zeichenkettenverarbeitung mit anderen Indices

Da die Repräsentation für Zeichenketten unverändert ist, können vorhandene Datenwerte der Implementation *string0* weiterverwendet werden. Sie werden auch von den neuen Funktionen verarbeitet. Allerdings hat sie sich die Semantik der Indices geändert, so daß in diesem Fall die vorhandenen Indices und ihre Berechnungsvorschriften anzupassen sind. Folgender Aufruf liefert nicht das erwartete Ergebnis:

```
string2.get(aString 2)
```

aString wurde mit der ersten Version erzeugt und der zweite Buchstabe der Zeichenkette auf *'a'* gesetzt. Der zweite Buchstabe hat nun aber nicht mehr den Index 2, sondern 1. Also ist das Ergebnis falsch; der Aufruf würde sogar zu einem Laufzeitfehler führen, denn ein Buchstabe mit dem Index 2 existiert nicht.

Somit müßten alle Indices konvertiert werden: Jeder Index muß dekrementiert werden, wie es auch an der Änderungen der Funktion *concat* zu sehen ist. Dabei reicht es nicht, vorhandene Indices zu dekrementieren, selbst wenn diese als Indices erkannt werden könnten, denn Indices könnten auch durch Berechnungen entstehen. Eine Lösung besteht in dem Einsatz eines Stellvertreterobjekts (vgl. Abschnitt 3.3.2).

Obwohl die Repräsentation des Datentyps und die Signaturen der auf ihm definierten Funktionen sich nicht geändert haben, ist die Implementation *string2* nicht kompatibel zur Implementation *string0*. Die Inkompatibilität liegt allein in der Implementation der Funktionen begründet. Diese Inkompatibilitäten können nicht automatisch festgestellt werden. Sie werden an Versionierungssysteme delegiert (vgl. Abschnitt 3.3.2).

Anwendung

Weiterhin bestehen während der Evolution anwendungsabhängige Einschränkungen. Beispielsweise können zwei Funktionen, die aus der Sicht des Ablaufsystems substituierbar sind, also beispielsweise die gleiche Anzahl und Art von Parametern erwarten, semantisch eine andere Bedeutung tragen.

```
let inc(i :Int) :Int = i+1
```

```
let printInt(i :Int) :Int =
begin
  let s = fmtInt(i)
  printString(s)
  length(s)
end
```

Die Funktionen *inc* und *printInt* sind von vornherein gegeneinander substituierbar. Während *inc* seinen Parameter inkrementiert zurückgibt, wandelt *printInt* ihn in eine Zeichenkette, druckt als Seiteneffekt diese Zeichenkette und gibt die Länge der Zeichenkette zurück. Die Semantik ist so unterschiedlich, daß eine Substitution nicht sinnvoll erscheint; diese Entscheidung ist aber anwendungsabhängig und daher von einem Entwickler zu treffen. Versionierungssysteme (vgl. Abschnitt 3.3.2) unterstützen ihn dabei.

3.3.2 Evolutionsunterstützung

Folgende Mechanismen unterstützen die Evolution (vgl. auch [Herlihy, Liskov 82; Atkinson et al. 93; Kirby et al. 96]):

Versionierung stellt kompatible Objekte in Beziehung, um ihre Eignung für und die Auswirkungen auf die Evolution darzustellen.

Kontexte verzeichnen Bindungen und Objekte, die gemeinsam ersetzt werden müssen.

Stellvertreter vermitteln zwischen Objekten, wenn das ersetzende Objekt nicht kompatibel zum ersetzten Objekt ist.

Transformation beschreibt die regelhafte Umwandlung von Objekten.

Versionierung

Um anwendungsabhängige Substituierbarkeit von Objekten zu erfassen, werden Versionierungssysteme [Skarra, Zdonik 87; Bolinger, Bronson 95] eingesetzt. Während der Entwicklung werden Objekte, die in einer Beziehung stehen, beispielsweise wie *string0* und *string1* einen abstrakten Datentypen implementieren, durch das Versionierungssystem erfaßt [Nagl 96]. Dabei gibt es zwei Arten von Zusammenhängen zwischen Objekten:

Historie: Aufeinanderfolgende Versionen von Objekten stellen einen (entwicklungs-)historischen Zusammenhang dar.

Alternative: Nebeneinanderstehende Versionen von Objekten stellen alternative Implementierungen dar.

Die Versionierung beinhaltet abgestufte Kompatibilitäten. Kompatibel heißt für die Evolution, daß die Objekte gegeneinander ausgetauscht werden können, wobei die Austauschbarkeit vom Ablaufzustand des Objektsystems abhängt, wie in Abschnitt 3.3.1 gezeigt. Kompatibilitäten lassen sich nicht automatisch ableiten, es können nur von vornherein bestimmte Objekte als inkompatibel erkannt werden, beispielsweise durch das Typsystem.

Der Entwickler bestimmt die Kompatibilität von Objekten anhand der Auswirkungen der Änderungen bzw. Unterschiede zwischen den Objekten auf die Integritätsbedingungen aus Abschnitt 3.3.1. Beispielsweise ändert die Fehlerkorrektur an einer Funktion normalerweise nicht die Semantik der Funktion, so daß eine weitgehende Kompatibilität gegeben ist, während unterschiedliche Datenstrukturen zu weitergehenden Änderungen und daher einer niedrigeren Kompatibilität führen.

Kontexte

Die Evolution von Bindungen und Objekten ist kontextabhängig, um die Integrität des Objektsystems zu wahren:

- Ein Objekt ist durch mehrere Bindungen gebunden; bei der Evolution sind mehrere dieser Bindungen gleichzeitig zu ersetzen. Diese Bindungen bilden einen Kontext.
- Mehrere Objekte (bzw. Bindungen an die Objekte) sind bei der Evolution gleichzeitig zu ersetzen. Diese Objekte bzw. Bindungen bilden einen Kontext.

Beispielsweise wird die Zeichenkettenverarbeitung in mehreren Funktionen benutzt (vgl. auch Abbildung 3.11):

```
let a() :string0.T = begin ... string0 ... end
let b() :string0.T = begin ... string0 ... end
let c(s :string0.T) = begin ... string0 ... end
let d(s :string0.T) = begin ... string0 ... end
c(a())
d(b())
```

Die Funktion *c* erhält als Parameter eine von der Funktion *a* erzeugte Zeichenkette, die Funktion *d* eine von der Funktion *b* erzeugte. Die Funktionen *a* und *c* sowie die Funktionen *b* und *d* bilden jeweils einen Kontext der Zeichenkettenverarbeitung *string0*; zwischen den Kontexten werden keine Zeichenketten ausgetauscht, nur innerhalb der Kontexte. Wenn die Bindungen an *string0* durch Bindungen an ein neues Objekt *string1* ersetzt werden sollen, können die beiden Kontexte getrennt betrachtet werden. Alle Bindungen der Objekte eines Kontextes an *string0* müssen gleichzeitig ersetzt werden.

Bei der Entwicklung werden solche Kontexte durch programmiersprachliche Mittel wie Sichtbarkeitsbereiche oder Modulstrukturen mit Import- und Exportbeziehungen gebildet [Koch 97]. Während des Ablaufs entstehen Kontexte durch die dynamisch erzeugten Objekte, die abhängig von anderen Objekten sind, beispielsweise die Zeichenketten, die von der Zeichenkettenverwaltung abhängen.

Stellvertreter

Anstatt eine Bindung eines Objektes an ein anderes Objekt zu ersetzen, wird ein Stellvertreterobjekt eingefügt, das Funktionen zur Weiterleitung bzw. Transformation der Operationen auf dem Objekt einsetzt [Senivongse, Utting 96]. Beispielsweise wird das Objekt *string0* nicht direkt durch *string1* ersetzt, weil die Objekte andere Repräsentationen für die Zeichenketten einsetzen und somit vorhandene Zeichenketten nicht weiterverwendet werden können. Stattdessen wird ein Adapterobjekt (vgl. Abbildung 3.12) eingesetzt, das über die Funktionen Konvertierungen (in beide

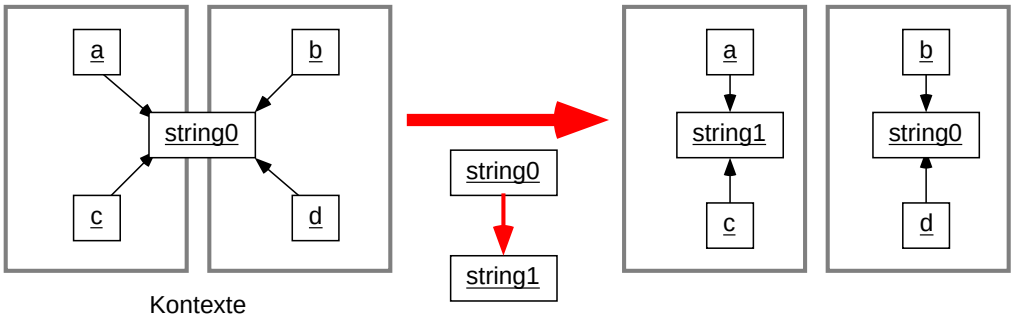

Abbildung 3.11: Evolution mit Kontexten

Richtungen) vornimmt. Es ist darauf zu achten, daß das Objekt *string1* nicht direkt im gleichen Kontext benutzt wird, so daß keine Zeichenketten der neuen Repräsentation entstehen, denn diese sind inkompatibel zu den vorhandenen. Das Objekt *string01Adapt* kann überall dort eingesetzt werden, wo vorher das Objekt *string0* eingesetzt wird. Beide Objekte können auch im gleichen Kontext verwendet werden. In der gleichen Weise kann die Anpassung der Indices zwischen *string0* und *string2* vorgenommen werden.

```
implementation string01Adapt :String
import string0 string1 int char array
export
  Let T = string0.T
  let adapt0To1(s0 :T) :string1.T = ...
  let adapt1To0(s1 :string1.T) :T = ...
  let new(size :int.T  init :char.T) :T =
    adapt1To0(string1.new(size init))
  let size(s :T) :int.T = string1.size(adapt0To1(s))
  let get(s :T  index :int.T) :char.T =
    string1.get(adapt0To1(s) index)
  ...
end
```

Abbildung 3.12: Adaption der Zeichenkettenverarbeitung

Wenn das ersetzte Objekt nicht aus Funktionen besteht, werden die vom Ablaufsystem auf dem Objekt angewendeten Operationen reflektiv durch Funktionen ersetzt [Maes 87]. Beispielsweise werden bei Datenobjekten die Lese- und Schreiboperationen auf dem Datum durch entsprechende Funktionen ersetzt.

```
let peter = tuple let alter = 26 end
```

Das Tupel *peter* mit einem Altersattribut wird durch ein anderes Tupel *peter'* ersetzt, welches das Geburtsdatum enthält.

```
let peter' = tuple let geburtstag = 10-sep-1971 end
```

Die Ersetzung erfolgt nicht direkt, weil dann auch alle Funktionen, die auf das Attribut *alter* zugreifen, geändert werden müßten. Stattdessen wird die Komponente *alter* des Tupels *peter* um eine Zugriffsfunktion ergänzt, die das Alter berechnet:

```
let petersAlter() = peter'.geburtstag - heute
assignRead(peter.alter petersAlter)
```

Der lesende Zugriff auf das Alter des Tupels *peter* wird also durch eine Funktion realisiert, die das Alter aus dem Geburtsdatum des Tupels *peter'* berechnet. Dies setzt voraus, daß die vom Ablaufsystem auf den Objekten angebotenen Operationen bekannt sind und das Ablaufsystem einen reflektiven Mechanismus bereitstellt, sie durch Funktionen des Objektsystems zu ersetzen.

Transformation

Bei der Evolution werden nicht nur Objekte durch andere Objekte ersetzt; teilweise werden auch Regeln angegeben, gemäß derer vorhandene Objekte zu transformieren sind [Atkinson et al. 93; Kirby et al. 96].

Wenn beispielsweise die Zeichenkettenverwaltung *string0* durch *string1* ersetzt wird, müssen alle mit *string0* im Ersetzungskontext erzeugten Zeichenketten umgewandelt werden, um von *string1* verarbeitet werden zu können. Die Transformation wird durch eine Funktion vorgenommen:

```
let transformString0To1(s :string0.T) :string1.T = ...
```

Diese Funktion ist auf alle Zeichenketten des Typs *string0.T* anzuwenden; diese Zeichenketten sind durch die von *transformString0To1* erzeugten Zeichenketten des Typs *string1.T* zu ersetzen. Die Transformation erfolgt einmalig oder bedarfsabhängig:

Einmalige, vollständige Transformation: Alle im Ersetzungskontext von *string0* erzeugten Zeichenketten werden transformiert und ersetzt. Dazu müssen alle betroffenen Zeichenketten bekannt sein.

Bedarfsabhängige, partielle Umwandlung: Die Transformation wird erst in dem Moment vorgenommen, in dem sie wirklich nötig ist, beispielsweise wenn eine Funktion der Zeichenkettenverwaltung *string1* auf einer Zeichenkette des Typs *string0.T* ausgeführt wird. Dies erlaubt eine Koexistenz verschiedener Objekte, also beispielsweise von Zeichenketten des Typs *string0.T* und *string1.T*. Die Zeichenketten der verschiedenen Typen müssen dazu unterschieden werden können. Während des Ablaufs wird wiederholt gezielt überprüft, ob eine Transformation notwendig ist.

Die zu transformierenden Objekte müssen identifiziert werden. Dazu werden sie entweder gekennzeichnet oder verzeichnet:

Kennzeichnen: Jedes (zu transformierende) Objekt wird bei seiner Erzeugung mit einem für das erzeugende Objekt eindeutigen Kennzeichner (*tag*) versehen, beispielsweise durch einen dynamischen Typen (vgl. Kapitel 5). Anhand der Kennzeichnung wird festgestellt, um welches Objekt es sich handelt, beispielsweise ob es sich um eine Zeichenkette des Typs *string0.T* oder *string1.T* handelt.

Bei der vollständigen Transformation werden alle Objekte des Objektsystems durchgegangen und die entsprechend gekennzeichneten Objekte transformiert. Dazu stellt das Ablaufsystem eine Möglichkeit bereit, alle Objekte des Objektsystems oder eines Kontextes durchzugehen. Außerdem werden die gekennzeichneten Objekte und ihre Kennzeichner, die auch wiederum Objekte darstellen, eindeutig von anderen, nicht gekennzeichneten Objekten unterschieden oder es werden von vornherein alle Objekte gekennzeichnet.

Verzeichnen: Jedes (zu transformierende) Objekt wird bei seiner Erzeugung in einem zu dem erzeugenden Objekt bzw. Kontext gehörenden Verzeichnis aufgenommen. Alle verzeichneten Objekte sind zu transformieren. Dies setzt voraus, daß die verzeichneten Objekte nicht kopiert werden, ohne daß die Kopien auch in das Verzeichnis aufgenommen werden. Damit verzeichnete Objekte auch wieder freigegeben werden, wenn sie nicht mehr anderweitig gebunden sind, werden die Bindungen des Verzeichnisses als schwache Bindungen [Birrell et al. 93] realisiert; schwache Bindungen sind Bindungen, die trotzdem die Freigabe des gebundenen Objektes erlauben, allerdings wird vor der Freigabe eine Funktion aufgerufen, welche die Bindung auflöst (vgl. Abschnitt 3.1.4).

3.4 Zusammenfassung

In diesem Kapitel wird der Betrieb von Objektsystemen unter dem Aspekt der langfristigen Sicherung der Ablauffähigkeit betrachtet:

- Reflektive Techniken dienen zur Ressourcenfreigabe, zur Persistenzsicherung und zur Evolutionsunterstützung. Sie werden als Spezialfall der Kooperation von Objektsystemen identifiziert.
- Aus der Forderung nach langlebigen (persistenten) Objektsystemen und geänderten Anforderungen erwächst die Notwendigkeit der Evolution von Objektsystemen. Es werden Evolutionsmechanismen eingeführt, welche die Ablauffähigkeit des Objektsystems sicherstellen.

Objektsysteme werden mit Hilfe von Ablaufsystemem betrieben. Die Ablaufsysteme definieren die Ablaufsemantik der Objektsysteme. Der Ablauf manifestiert sich in einer Folge von Ablaufzuständen. Während des Ablaufs werden Objekte angelegt und Ressourcen belegt. Um die Ablauffähigkeit von Objektsystemen zu erhalten, werden Objekte und Ressourcen freigegeben, wenn diese nicht mehr zum Betrieb des Objektsystems benötigt werden. Zur generischen Ressourcenfreigabe werden reflektive Techniken eingesetzt.

Durch das einheitliche Objektsystemmodell präsentiert sich Reflektion als eine besondere Art der Kooperation, hier zwischen Ablauf- und Objektsystem. Im Kontext kooperierender Objektsysteme werden in Kapitel 4 die reflektiven Eigenschaften der Ablaufsysteme weiter genutzt, beispielsweise um die Freigabe von gebundenen Objekten anderer Objektsysteme, also externen Objekten, zu veranlassen oder zu verhindern. Damit stellt Reflektion gleichzeitig einen Spezialfall der Kooperation und eine Technik zur Realisierung von Kooperation dar.

Aus der Langlebigkeit der durch Objektsysteme unterstützten Anwendungen erwächst die Forderung nach der Persistenz von Objektsystemen. Zur Unterstützung der Persistenz von Ablaufzuständen eines Objektsystems dient eine persistente Repräsentation des Ablaufzustands. Die Wiederherstellung eines persistent abgelegten Ablaufzustands erfordert besondere Maßnahmen in Kooperation zwischen Ablauf- und Objektsystem, also Reflektion, um Zustände außerhalb des Objektsystems wiederherzustellen. Diese Zustände außerhalb des Objektsystems entstehen durch Kooperation mit anderen Objektsystemen. In Kapitel 4 wird vertiefend auf die Persistenz von Bindungen zwischen Objektsystemen eingegangen.

Die Langlebigkeit von Objektsystemen führt wiederum zur Forderung nach Evolution, um geänderten Anforderungen der Anwendungen gerecht zu werden. Evolution setzt die Entwicklung von Objektsystemen während des Ablaufs fort. Bei der Evolution von Objektsystemen ist daher wie bei der Entwicklung die Wahrung der Ablauffähigkeit und Integrität sicherzustellen. Dazu werden Evolutionsmechanismen vorgestellt, welche die Ablauffähigkeit erhalten und kontextabhängig einzusetzen sind. Zur Realisierung der Evolution werden wiederum reflektive Techniken eingesetzt. Wie in Kapitel 4 gezeigt wird, entstehen aus der (unabhängigen) Evolution von Objektsystemen während einer Kooperation besondere Anforderungen an die Kooperationsmechanismen, damit die Objektsysteme sich an die Änderungen der Kooperationspartner anpassen.

Kapitel 4

Kooperation von Objektsystemen in heterogenen Umgebungen

Komplexe Anwendungen, die von Objektsystemen realisiert werden, erfordern die Lösung verschiedener Aufgaben unter Einbeziehung unterschiedlicher Leistungen. Objektsysteme kooperieren miteinander, um gemeinsam die Leistungen zu erbringen, die zur Lösung einer anspruchsvollen Aufgabe benötigt werden.

Die miteinander kooperierenden Objektsysteme sind autonom [Enslow 78], d.h. sie sind prinzipiell eigenständig ablauffähig; dies schließt von den Klienten unabhängige Dienstleister, die nur nach Aufforderung durch einen Klienten aktiv werden, ein. Kooperation ist für autonome Objektsysteme eine temporäre Maßnahme, um eine Anwendungsaufgabe zu lösen, die Lösung effizienter zu gestalten oder andere Objektsystemen bei der Lösung ihrer Aufgaben zu unterstützen [Burger 97]. Die Objektsysteme entstehen unabhängig voneinander, laufen unabhängig voneinander ab und kooperieren explizit während des Ablaufs. Damit unterscheiden sie sich von Systemen, die gemeinsam entwickelt werden und nur gemeinsam ablauffähig sind, beispielsweise Simulationen auf Mehrprozessorsystemen [Enslow 78].

Die Kooperation bietet folgende Vorteile [Smith 88; Hsieh et al. 93; Lux 95]:

Ressourcenverteilung: Ein Objektsystem besitzt nicht die nötigen Ressourcen, um eine Aufgabe zu erfüllen. Es kooperiert daher mit anderen Objektsystemen, die lokal über die Ressourcen verfügen.

Lastverteilung: Ein Objektsystem verfügt nicht über die Kapazitäten, um eine Aufgabe optimal oder innerhalb vorgegebener Grenzen (im Sinne der verbrauchten Ressourcen oder der verbrauchten Zeit) zu erfüllen. Es könnte die Aufgabe also prinzipiell lösen, würde dabei aber unangemessen viele Ressourcen oder zuviel Zeit verbrauchen. Es kooperiert daher mit anderen Objektsystemen, um die Last zu verteilen.

Weiterverwendung von Vorarbeiten: Ein Objektsystem kann durch Kooperation die Ergebnisse von Aufgaben, die andere schon gelöst haben, weiterverwenden. Es vermeidet damit Doppelarbeit und die redundante Erzeugung von Ergebnissen, wenn diese Ergebnisse überhaupt reproduziert werden können. Unter der Annahme, daß das Ergebnis korrekt ist, vermeidet das Objektsystem damit Fehler, die bei der wiederholten Produktion des Ergebnisses auftreten könnten, die im Extremfall sogar zu keinem oder einem falschen Ergebnis führen würden.

Es wäre keine Kooperation zwischen Objektsystemen erforderlich, wenn jedes Objektsystem jede gewünschte Funktionalität mit optimaler Performanz bzw. minimalem Ressourcenverbrauch erbringen kann und alle Informationen, Abläufe und Ablaufzustande aller Objektsysteme jederzeit überall verfügbar wären. Diese Annahme ist unrealistisch:

- Ablaufumgebungen sind aufgrund der Heterogenität (vgl. Abschnitt 4.1.4) für die Erfüllung bestimmter Aufgaben prädestiniert. Objektsysteme, die in diesen Ablaufumgebungen ablaufen, sind auf die optimale Erfüllung dieser Aufgaben spezialisiert. Andere Objektsysteme, welche die Erfüllung einer dieser Aufgaben zur Lösung ihrer Aufgabe benötigen, delegieren diese Aufgabe an das spezialisierte Objektsystem. Sie profitieren von der Spezialisierung, denn das Objektsystem kann damit die Aufgabe überhaupt (wenn die Spezialisierung exklusiv ist) oder mit einer besonderen Performanz erfüllen.

- Objektsysteme unterscheiden sich durch die Objekte, aus denen sie aufgebaut sind. Diese Objekte entstehen während der Entwicklung und während des Ablaufs des Objektsystems. Während Entwicklungsobjekte (vgl. Abschnitt 2.6.2) Bestandteil mehrerer Objektsysteme werden können, sind die dynamisch während eines Ablaufs, zum Beispiel durch Benutzerinteraktion, erzeugten Objekte exklusiver Bestandteil eines Objektsystems. Diese Objekte stellen den (persistenten) Ablaufzustand des Objektsystems dar. Ein Objektsystem kann nicht direkt auf Objekte (Informationen, Abläufe oder Ablaufzustände) eines anderen Objektsystems zugreifen, wenn es diese Objekte zur Lösung seiner Aufgabe benötigt. Nur durch Kooperation mit diesem Objektsystem kann es auf die Objekte des Objektsystems zugreifen und somit seine Aufgabe erfüllen.

Kooperation bedeutet, daß die Autonomie der kooperierenden Objektsysteme eingeschränkt wird. In dem Moment, in dem eine Aufgabe an ein anderes Objektsystem delegiert oder das Ergebnis einer von einem anderen Objektsystem gelösten Aufgabe weiterverwendet werden soll, koordinieren die Objektsysteme sich über ihre Ablaufumgebungen. Sie stellen eine Verbindung untereinander her, binden sich also aneinander, und halten dabei ein gemeinsames Protokoll ein, um sich zu verständigen und auszutauschen. Es besteht die Aufgabe, Protokolle zu finden, die Autonomie und Kooperation anwendungs-, partner- und situationsspezifisch gegeneinander abwägen [Silberschatz, Zdonik 96].

Das in dieser Arbeit aus dem Objektsystemmodell systematisch abgeleitete Kooperationsmodell führt zu autonomen Objektsystemen in heterogenen Umgebungen (vgl. Abschnitt 4.1). Aus der Autonomie und Heterogenität folgern zwei Herausforderungen an die Kooperation zwischen Objektsystemen:

- Die Übertragung von Objekten zwischen Objektsystemen erfordert eine Umwandlung der (heterogenen) Repräsentationen und eine Wiederherstellung von Bindungen unter Wahrung der Autonomie des Empfängers (vgl. Abschnitt 4.2).

- Bindungen zwischen Objektsystemen schränken die Autonomie der Objektsysteme ein und gefährden die Ablauffähigkeit, so daß Kooperationsstrategien zwischen den Kooperationspartnern kontextabhängig ausgehandelt werden müssen (vgl. Abschnitt 4.3).

4.1 Kooperationsmodell

Das Kooperationsmodell wird aus dem in Kapitel 2 entwickelten Objektsystemmodell abgeleitet. Ein Objektsystem kooperiert mit anderen Objektsystemen auf folgende Arten [Picco et al. 97]:

- Es löst Operationen auf Objekten anderer Objektsysteme aus.

- Es überträgt Objekte von/zu anderen Objektsystemen.

- Es bindet Objekte anderer Objektsysteme, wenn eine externe Operation eine Bindung an ein externes Objekt aufbaut oder ein Objekt übertragen wird, das externe Objekte bindet.

Diese grundlegenden Eigenschaften bilden das Kooperationsmodell. Kooperierende Objektsysteme nutzen Kombinationen der Kooperationsarten zur Realisierung unterschiedlicher Anforderungen. Beispielsweise werden Massendaten auf andere Art ausgetauscht als einfache Daten [Birrell et al. 95], während grundsätzlich die Übertragung von Objekten (vgl. Abschnitt 4.2) in allen Kooperationsparadigmen C/S, REV, COD und MA (vgl. Kapitel 1) verwendet wird.

4.1.1 Ablauf einer Kooperation

Die Ausgangssituation einer Kooperation wird von zwei Objektsystemen und zwei Ablaufumgebungen gebildet, in denen jeweils die Objektsysteme ablaufen. Jede Ablaufumgebung verfügt über ein Ablaufsystem mit einem Interpreter, der die Repräsentation interpretieren kann, in der das Objektsystem vorliegt. Jedes Objektsystem kann nur die Operationen ausführen, die der jeweilige Interpreter unterstützt. Die Interpreter stellen grundlegende Kommunikationsmechanismen zur Verfügung, über welche die Kooperation zwischen den Objektsystemen realisiert wird.

Bei Objektsystemen, die auf einem Rechner ablaufen, werden beispielsweise vom lokalen Betriebssystem implementierte "Stecker" (*sockets*) als grundlegender Kommunikationsmechanismus eingesetzt; bei verteilt ablaufenden Objektsystemen kommt ein Netzwerkprotokoll zum Einsatz.

Die Kommunikation kann auch schon als Kooperation realisiert sein, beispielsweise durch die Anbindung von C-Bibliotheken an eine virtuelle Maschine wie die Java-Maschine [Lindholm, Yellin 96]. Da die C-Bibliotheken eine eigene Ablaufumgebung besitzen und damit ein Objektsystem darstellen, das von dem auf der Java-Maschine ablaufenden Objektsystem angesprochen wird, handelt es sich um Kooperation mit allen in diesem Kapitel dargestellten Herausforderungen (Übertragen und Binden von Objekten, Auslösen von Operationen; unabhängige Speicherverwaltung, Evolution, Persistenz usw.). Über Operationen auf den als externe Objekte angebundenen C-Bibliotheken wird die Kooperation zwischen weiteren Objektsystemen realisiert.

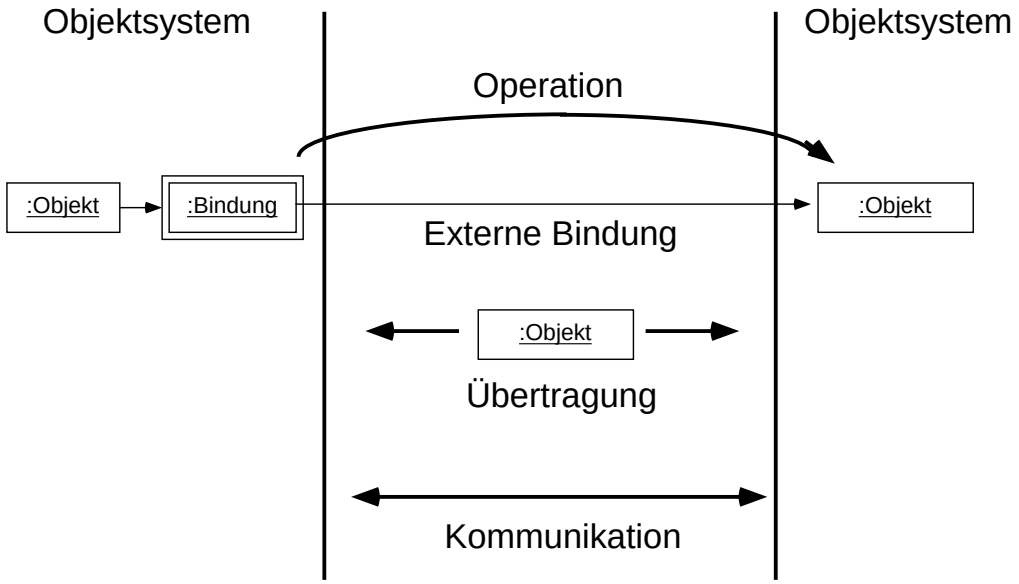

Abbildung 4.1: Kooperation

Die Kooperation setzt sich aus folgenden Teilen zusammen (vgl. Abbildung 4.1):

Kommunikation: Die Objektsysteme kommunizieren miteinander. Nach Aufbau eienr Verbindung kooperieren sie über diese Verbindung und bauen die Verbindung wieder ab. Dieser Aspekt wird kurz in Abschnitt 4.1.2) beleuchtet, stellt aber keinen zentralen Aspekt des Kooperationsmodells dar.

Übertragung: Ein Objektsystem überträgt ein Objekt an das andere Objektsystem. Bei heterogenen Ablaufumgebungen (vgl. Abschnitt 4.1.4) ist dabei eine Portierung des übertragenen Objekts vorzunehmen. Insbesondere sind die Bindungen zu betrachten. Die Übertragung wird in Abschnitt 4.2 ausführlich behandelt.

Externe Objekte: Ein Objektsystem bindet ein Objekt des anderen Objektsystems und löst auf gebundenen (externen) Objekten Operationen aus. Insbesondere ist ein Ausgleich zwischen

der Wahrung der Integrität der Objektsysteme und dem Erhalt ihrer Autonomie zu finden. Externe Objekte sind Thema von Abschnitt 4.3.

Voraussetzung für Kooperation in heterogenen Umgebungen ist eine Adaption an die Fähigkeiten und Eigenschaften des Kooperationspartners. Dieses Thema zieht sich durch alle Abschnitte, beispielsweise die Portierung bei der Übertragung von Objekten oder die Behandlung externer Bindungen.

4.1.2 Kommunikation

Die Objektsysteme *os1* und *os2* (vgl. Abbildung 4.2) möchten miteinander kooperieren. Erster Schritt einer Kooperation ist der Aufbau einer Kommunikationsverbindung zwischen den Objektsystemen. Hier wird exemplarisch ein möglicher Ablauf gezeigt, der zum Aufbau einer Verbindung über ein Kooperationssystem führt; das Kooperationssystem ist entweder Teil des Ablaufsystems, beispielsweise für die Kooperation mit Basisobjekten gemäß eines Metaobjektprotokolls [Kiszales et al. 91], oder ein externes System wie ein ORB (*object request broker*) in CORBA [OMG 98]:

1. Das Objektsystem *os2* zeigt sich kooperativ, d.h. es gibt seine prinzipielle, evtl. eingeschränkte Kooperationsbereitschaft bekannt. Dabei gibt es eine Beschreibung an, die es charakterisiert und über die es von anderen Objektsystemen gefunden werden kann. Diese Beschreibung besteht aus unstrukturierten und/oder strukturierten Informationen, z.B. einer textuellen Beschreibung (unstrukturiert) oder einer Signatur (strukturiert).

2. Das Kooperationssystem verwaltet die Beschreibungen der Objektsysteme, auch über heterogene Ablaufumgebungen hinweg. Die Beschreibungen der Objektsysteme werden dazu über die verschiedenen Ablauf- und Entwicklungsumgebungen hinweg vereinheitlicht.

3. Das Objektsystem *os1* wählt anhand einer eigenen Beschreibung des gesuchten Objektsystems oder durch Inspektion der vom Kooperationssystem verwalteten Beschreibungen von kooperativen Objektsystemen das Objektsystem *os2* als Kooperationspartner aus.

4. Die beiden Objektsysteme handeln ein Protokoll aus, über das sie die Kooperation abwickeln. Die Auswahl des Protokolls hängt davon ab [Saltzer et al. 84; Lampson 94],

   - welche Protokolle beide Objektsysteme (mit Hilfe ihrer Ablaufsysteme) beherrschen;
   - welche Charakteristika die Verbindung aufweist, beispielsweise ob die beiden Ablaufsysteme sich auf dem gleichen Rechner befinden oder in einem Weitverkehrsnetzwerk verteilt sind; und
   - welche Einschränkungen die beiden kooperierenden Objektsysteme bezüglich der Kooperation vornehmen, z.B. Dauer und Art der Kooperation, Zugriffsbeschränkungen, Unterstützung von externen Bindungen oder Migration.

   Das Protokoll kann während der Kooperation gewechselt werden.

Der Zustand des Kooperationsverlaufs spiegelt sich in Objekten und Bindungen, sowohl der Objektsysteme als auch der Ablaufsysteme, wider.

Wenn während der Kooperation die Verbindung verlorengeht, handeln die Kooperationspartner unabhängig voneinander gemäß des vereinbarten Protokolls, beispielsweise indem sie versuchen, die Verbindung wieder aufzubauen, den Zustand der Kooperation persistent ablegen, um ihn zu einem späteren Zeitpunkt zu reaktivieren, oder die Verbindung abbauen.

Jeder der Kooperationspartner kann jederzeit die Kooperation beenden. Die Verbindung wird abgebaut, d.h. der ausgehandelte Kontext über die zu verwendenden Protokolle gilt nicht mehr. Durch Bindungen an externe Objekte kann weiterhin eine (implizite) Verbindung zwischen den Objektsystemen bestehen. Wenn ein Objektsystem Operationen auf diesen externen Objekten durchführen möchte, wird versucht, die Kooperation fortzusetzen, indem die Verbindung erneut aufgebaut wird. Dies entspricht auch dem Verhalten zustandsloser Kooperationsprotokolle wie

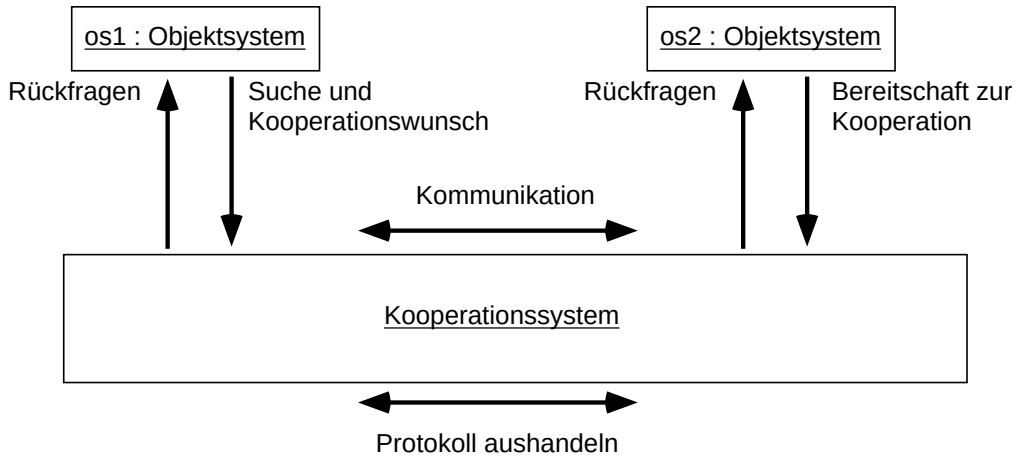

Abbildung 4.2: Aufbau einer Kommunikationsverbindung

HTTP, bei denen externe Bindungen durch URLs (*uniform resource locator*) repräsentiert werden und erst durch den Zugriff auf die URL eine Verbindung zu dem Objektsystem, welches das referenzierte Objekt enthält, aufgebaut wird [Bösch 97; Manola 98].

Das folgende Beispiel zeigt, wie eine Verbindung zwischen den Objektsystemen *os1* und *os2* aufgebaut wird [Mathiske 96]. Das Objektsystem *os2* gibt über die Operation *cooperation.register* seine Bereitschaft zur Kooperation bekannt und identifiziert sich, in diesem Fall durch eine Zeichenkette. Die Funktion *cooperate* soll aufgerufen werden, sobald eine Kooperation zustande kommt; die Funktion erhält als Parameter *c* ein Objekt, über das die Kooperation mit dem anderen Objektsystem als Kooperationspartner abgewickelt wird. Die Funktion *cooperate* wird damit wie die in Abschnitt 3.1 bei der Speicher- bzw. Ressourcenfreigabe gezeigten Funktionen asynchron vom Ablaufsystem in einem Prozeß ausgeführt, wenn eine Kooperation beginnt. Alternativ könnte der laufende Prozeß auch an dieser Stelle blockieren und auf den Beginn einer Kooperation warten. Das Objektsystem kann die möglichen Kooperationspartner, -formen oder -protokolle durch weitere Parameter der Funktion *register* einschränken:

```
let cooperate(c :cooperation.T) :Ok = ...
cooperation.register("os2" cooperate ...)
```

Die Information, daß das Objektsystem *os2* kooperationsbereit ist wird durch das Kooperationssystem an eine vermittelnde Instanz weitergegeben.

Das Objektsystem *os1* fragt über die Operation *cooperation.registered* nach möglichen Koooperationspartnern; über den Parameter *predicate* kommt eine kriteriengesteuerte Suche zum Einsatz, um die Menge der möglichen Kooperationspartner einzuschränken:

```
let predicate(c :cooperation.T) :Bool = ...
let cooperationPartners = cooperation.registered(predicate ...)
let cp = choose(cooperationPartners ...)
```

Als Ergebnis erhält das Objektsystem eine Menge von möglichen Kooperationspartnern *cooperationPartners*, die es auswertet, um einen Partner *c* auszuwählen. Wenn das Objektsystem *os1* einen Kooperationspartner gefunden hat, baut es über die Operation *cooperation.begin* eine Verbindung zu dem Objektsystem auf, mit dem es kooperieren möchte; dabei kann die Art der Kooperation noch weiter spezifiziert werden, beispielsweise ob externe Bindungen, Operationen auf externen Objekten und Migration gewünscht sind oder wie eng die Kopplung sein soll (vgl. Abschnitt 4.3):

```
let c = cooperation.begin(cp ...)
```

Das Kooperationssystem nimmt mit dem Objektsystem *os2* Kontakt auf, um eine Verbindung aufzubauen und ein Protokoll für diese Verbindung auszuhandeln. Dieses Protokoll legt u.a. fest, welche Objekte und Bindungen über die Verbindung in welchen Repräsentationen ausgetauscht werden und welche Operationen auf Objekten möglich sind. Wenn die Verbindung zu den gewünschten Konditionen zustande kommt, erhält das Objektsystem *os1* als Ergebnis der Operation *cooperation.begin* ein Objekt *c*, das die Verbindung zu dem anderen Objektsystem symbolisiert. Über dieses Objekt wird die Kooperation abgewickelt. Analog existiert auf der Seite des Objektsystems *os2* auch ein Objekt *c*, über das die Kooperation abgewickelt wird; das Objektsystem *os2* erhält es als Parameter über den Aufruf der Funktion *cooperate*. Wenn eines der Objektsysteme *os1* oder *os2* die Kooperation beenden möchte, nutzt es dazu die Operation *cooperation.end*:

```
cooperation.end(c)
```

Dieses Beispiel zeigt nur eine Möglichkeit, wie die Verbindung auf- und abgebaut wird. Es existieren viele Variationen, beispielsweise:

- Die Kooperationspartner blockieren, bis eine Kooperation zustande kommt.
- Statt einer gerichteten Kooperation mit einem Objektsystem werden mehrere Kooperationen gleichzeitig durchgeführt (*broadcast*).
- Bei zustandslosen Kooperationsprotokollen ist der Verbindungsabbau implizit; jede Kooperation erfordert einen neuen Verbindungsaufbau.

4.1.3 Kooperationsmechanismen und -strategien

Das Kooperationsmodell abstrahiert entsprechend der Prinzipien der Trennung von Strategie und Mechanismus [Levin et al. 75] sowie des *end-to-end* Arguments [Saltzer et al. 84] von den Kooperationsmechanismen, die Teile des Modells implementieren. Beispielsweise werden

- Bindungen zwischen Objektsystemen durch Einbettung von Objektbezeichnern wie Hauptspeicheradressen oder durch Namensdienste verwaltete symbolische Namen hergestellt [Mathiske 96; OMG 98];
- aus virtuellen Maschinen wie der Java Maschine [Lindholm, Yellin 96] heraus C-Bibliotheken aufgerufen;
- Daten von und zu Datenbanksystemen übertragen [Date, Darwen 97];
- entfernte Prozeduren synchron [Birrell, Nelson 84] oder asynchron [Liskov, Shriva 88; Walker et al. 90] aufgerufen, wobei meist die Lokation der aufgerufenen Prozedur durch Einsatz eines lokalen *proxy* [Shapiro 86] transparent ist, sogar wenn die Prozedur migriert [Black, Artsy 89];
- entfernte Objekte angesprochen [Sun 97; OMG 98] oder
- Objekte (als Applets) übertragen [Brown, Najork 96; Java Team et al. 96].

Das Kooperationsmodell erlaubt unterschiedliche Kooperationsstrategien, die von der Lokation der Objektsysteme und den Charakteristika der Verbindung, insbesondere der Latenzzeit und der Zuverlässigkeit [Lampson 94], abhängen (vgl. Abbildung 4.3). Objektsysteme auf demselben Rechner können sehr schnell miteinander kommunizieren und die Verbindung bricht normalerweise nicht ab. Sobald andere Rechner (Router, Kabel usw.) involviert sind, sinkt die Zuverlässigkeit, weil mehr Einflußfaktoren und Fehlerquellen existieren, und steigt die Latenzzeit, weil Engpässe zu überwinden sind und der Protokollaufwand steigt. Wenn ein Objektsystem sich auf einem Rechner befindet, der nur temporär erreichbar ist, beispielsweise bei mobilen Rechnern, wird die Latenzzeit zeitweise sehr groß und die Zuverlässigkeit der Verbindung sehr klein. Außerdem muß jederzeit, auch bei relativ zuverlässigen Verbindungen, damit gerechnet werden, daß aufgrund eines Fehlers

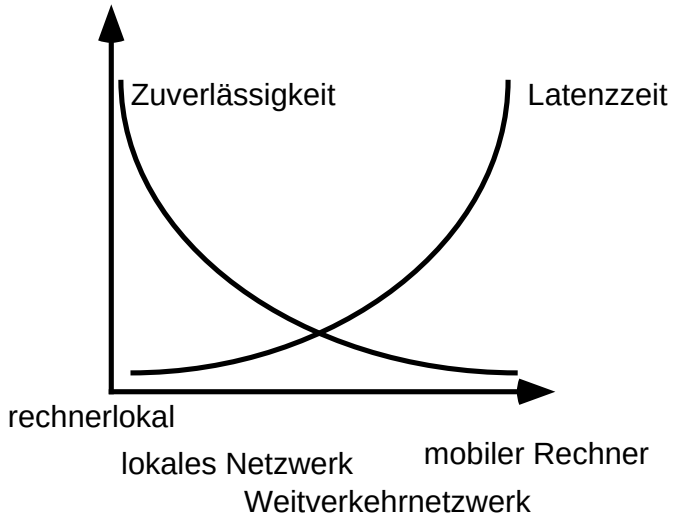

Abbildung 4.3: Charakteristika von Verbindungen

oder auf expliziten Wunsch ein Objektsystem während einer Kooperation zeitweise unerreichbar ist oder heruntergefahren wird.

Das Ablaufsystem implementiert grundlegende Kooperationsmechanismen, ohne die Koooperationssstrategien festzulegen [Lux 95]. Wenn das Ablaufsystem auch die Kooperationsstrategien festlegen würde, müßte das Ablaufsystem für jede neue Kooperationsstrategie, die ein Objektsystem zum Ablauf benötigt, erweitert werden, was zu vielen Änderungen und unhandlich großen Ablaufsystemen führt. Das Ablaufsystem stellt daher nur Basismechanismen bereit, auf deren Grundlage die Kooperation stattfindet. Das Objektsystem wählt die Kooperationsmechanismen und -strategien aus, mit deren Hilfe es mit anderen Objektsystemen kooperiert. Dabei kann es für verschiedene Kooperationsbeziehungen auch unterschiedliche Kooperationsmechanismen und -strategien auswählen. Die Einbindung der Kooperationsmechanismen in die anwendungsabhängigen Kooperationsstrategien erfolgt reflektiv, wie in Kapitel 3 gezeigt.

Um auch eine Erweiterbarkeit auf der Ebene der Kooperationsmechanismen zu bieten und damit den Objektsystemen Zugang zu (potentiell unendlich vielen) Kooperationsmechanismen zu geben, kann das Ablaufsystem einen grundlegenden generischen Kooperationsmechanismus implementieren, beispielsweise die Anbindung von C-Bibliotheken [Bennett 90] oder von CORBA-Objekten [OMG 98]. Dieser Kooperationsmechanismus ist dann vorgegeben (*built-in*). Mit Hilfe dieses Kooperationsmechanismus' werden weitere Kooperationsmechanismen angebunden (*add-on*). Die Aussagen des Kooperationsmodells gelten auch für diesen generischen Kooperationsmechanismus.

4.1.4 Heterogenität und Adaption

Offene Systeme sind heterogen [Nierstrasz, Meijler 95; Notkin et al. 86; Tolksdorf 93; Shaw, Garlan 96; Rosenblum, Wolf 97]. Heterogenität bezeichnet die Uneinheitlichkeit verschiedener Ablaufumgebungen, d.h. der Ablaufsysteme inklusive der Basisobjekte und der in der Ablaufumgebung vorhandenen externen Objekte. Ablaufsysteme unterscheiden sich z.B. bezüglich der interpretierten Repräsentation, der Performanz der Abarbeitung oder der Unterstützung von Reflektion. Externe Objekte bieten beispielsweise Zugang zu Kooperationsmechanismen oder Datenbanksystemen. Objektsysteme, die in diesen verschiedenen Ablaufumgebungen ablaufen, nutzen die speziellen Eigenschaften der Ablaufumgebungen, um ihre Aufgabe zu bearbeiten. Sie passen sich also den Eigenschaften der Ablaufumgebung an (Adaption).

Aus Ablaufsystemen und externen Objekten bestehende Ablaufumgebungen sind in zweierlei Hinsicht heterogen, d.h. unterscheiden sich voneinander:

Umfang: welche unterschiedlichen Systemteile gibt es?

Funktionalität: was leisten äquivalente Systemteile?

Homogenität erleichtert die Kooperation zwischen Objektsystemen [Shub 90; Callsen, Agha 94; Steensgard, Jul 95; Mathiske 96]. Homogene Ablaufumgebungen unterscheiden sich weder in dem Umfang noch in der Funktionalität, d.h. die Systeme verfügen über die gleichen Teile und die äquivalenten Teile der Systeme verfügen über gleiche Eigenschaften. Homogene Ablaufumgebungen sind damit aus Sicht des Objektsystems nicht voneinander unterscheidbar. Ein für diese Ablaufumgebung entwickeltes Objektsystem kann auf jeder dieser homogenen Ablaufumgebungen ablaufen, ohne daß sich ein Unterschied in der Abarbeitung zeigt.

Die Heterogenität von Ablaufumgebungen wird von Nachfragern und Anbietern vorangetrieben,

- auf der Seite der Nachfrager nach Ablaufumgebungen von den verschiedenen, komplex strukturierten und sich weiterentwickelnden Anwendungsbereichen sowie der räumlichen Verteilung,
- auf der Seite der Anbieter von Ablaufumgebungen von den technologischen Entwicklungen und der Konkurrenz verschiedener Anbieter.

Im Folgenden werden die Dimensionen der Heterogenität aufgezeigt. Aus der Diskussion wird deutlich, daß Homogenität eine nicht haltbare Annahme ist und daher ein wachsender Bedarf an Adaption besteht.

Heterogenität durch unterschiedliche Anwendungsbereiche. Auf der Nachfragerseite finden sich, auch innerhalb eines Unternehmens, unterschiedliche Anwendungsbereiche, z.B. Buchhaltung und Maschinensteuerung. Die Anwendungsbereiche definieren Anforderungen an die Objektsysteme, z.B. bezüglich der Persistenz der erzeugten Objekte (in der Buchhaltung) oder der zeitkritischen Ansteuerung externer Systeme (bei Maschinensteuerungen). Die Objektsysteme realisieren mit den von der Ablaufumgebung zur Verfügung gestellten Mitteln diese Anforderungen. Somit werden unterschiedliche Leistungsmerkmale von den Ablaufumgebungen für verschiedene Anwendungsbereiche gefordert, die von anwendungsbereichsspezifischen Ablaufumgebungen erfüllt werden.

Heterogenität durch Anwendungsprofile in einem Anwendungsbereich. Die von den Ablaufumgebungen geforderten Leistungsmerkmale sind abhängig von den Anwendern in einem Anwendungsbereich. Beispielsweise stellt die Buchhaltung für ein multinationales Unternehmen andere (höhere) Anforderungen als die Buchhaltung für einen Ein-Personen-Haushalt; ebenso ist die Steuerung eines Atomkraftwerks ungleich kritischer als die einer Waschmaschine. Ein Anwendungsbereich, beispielsweise die Buchhaltung, kann außerdem in verschiedenen, ansonsten durchaus vergleichbaren Unternehmen, unterschiedlich strukturiert sein oder unterschiedlich eingesetzt werden. Dies führt zu unterschiedlichen Anforderungen an die Objektsysteme und die Ablaufumgebungen in diesem Anwendungsbereich. Somit entstehen in den Anwendungsbereichen unterschiedliche Anforderungsprofile, die von unterschiedlichen Ablaufumgebungen erfüllt werden.

Heterogenität durch komplexe Anwendungsbereiche. Je komplexer die Anwendungsbereiche werden, desto komplexer werden die Anforderungen an die Objektsysteme und ihre Ablaufumgebungen. Komplexe Anwendungsbereiche führen zu widersprüchlichen Anforderungen, z.B. wenn in einem Unternehmen die Buchhaltung als Grundlage sowohl für das Rechnungswesen als auch für Managemententscheidungen dienen soll, wobei für das Rechnungswesen Genauigkeit und Ausfallsicherheit im Vordergrund stehen, während für das Management leichte Bedienbarkeit und

schnelle Aggregationsmöglichkeiten (mit niedriger Genauigkeit) wichtig sind. Diese Anforderungen lassen sich normalerweise nicht von einem System erfüllen, sondern erfordern die Zerlegung in verschiedene, spezialisierte Systeme, die sich auf spezialisierte Ablaufumgebungen (z.B. verschiedene Betriebssysteme und Benutzeroberflächen) stützen. Somit entsteht Heterogenität durch die Komplexität eines Anwendungsbereichs.

Heterogenität durch Evolution eines Anwendungsbereichs. Die Anforderungen eines Anwendungsbereichs entwickeln sich mit der Zeit weiter. Dies führt dazu, daß die in diesem Bereich eingesetzten Objektsysteme und Ablaufumgebungen geändert oder ersetzt werden müssen (vgl. zur Evolution Abschnitt 3.3). Somit entstehen Generationen von Ablaufumgebungen, die zeitweise nebeneinander existieren. Dies führt zu einer Heterogenität der Ablaufumgebungen für einen Anwendungsbereich durch die Evolution des Anwendungsbereichs.

Heterogenität durch lokale Ressourcen. Heterogenität ergibt sich aus lokalen Besonderheiten, wenn Ressourcen nur an bestimmten Stellen vorhanden sind, beispielsweise weil sie teuer, groß (unbeweglich) oder besonders schützenswert sind, oder weil sie besonderen Anforderungen bezüglich Konsistenz oder Wartung unterliegen. Somit unterscheidet sich die Ablaufumgebung an dem Ort, wo die Ressource vorhanden ist, von der Ablaufumgebung an anderen Orten. Ein Objektsystem, das zur Lösung seiner Aufgabe beim Ablauf bestimmte Ressourcen benötigt, muß in einer Ablaufumgebung ablaufen, welche diese Ressourcen zur Verfügung stellt. Falls diese Ressourcen nicht in einer Ablaufumgebung vorliegen, müssen in den verschiedenenen Ablaufumgebungen ablaufende Objektsysteme so miteinander kooperieren, daß die Aufgabe gelöst wird.

Heterogenität durch statische und dynamische Leistungsfähigkeit und Kapazität. Eigenschaften einer Ablaufumgebung sind ihre Leistungsfähigkeit und ihre Kapazität, wobei sich hier statische (d.h. zeitlich unveränderliche) und dynamische (d.h. mit der Zeit variierende) Aspekte unterscheiden lassen. Ansonsten gleichartige Ablaufumgebungen sind unterschiedlich leistungsfähig, z.B. weil die Interpreter (Prozessoren) unterschiedlich schnell (statischer Aspekt) oder ausgelastet (dynamischer Aspekt) sind. Ihre Kapazität hängt beispielsweise vom Speicherausbau (statischer Aspekt) oder vom aktuellen Speicherverbrauch (dynamischer Aspekt) ab. Dadurch ergibt sich Heterogenität, die den Ablauf eines Objektsystem beeinflußt, z.B. weil der Ablauf zu langsam ist, um zeitliche Anforderungen einzuhalten, oder weil während des Ablaufs nicht genügend Speicher zur Verfügung steht, um die Aufgabe zu erfüllen.

Heterogenität durch Leistungsunterschiede der Anbieter. In einem homogenen Markt wären alle Ablaufumgebungen, die die Anforderungen eines Anwendungsbereichs erfüllen, von der Leistung her gleich und daher ununterscheidbar für Objektsysteme, die auf ihnen ablaufen; nur ihr Herstellungsprozeß und ihre Preise wären unterschiedlich. In einem heterogenen Markt grenzen die Anbieter sich auch durch Leistungsunterschiede voneinander ab. Die Leistungen erfüllen die Anforderungen des Anwendungsbereichs in unterschiedlichem Maße. Diese unterschiedlichen Leistungen führen zu Heterogenität.

Heterogenität durch technologischen Fortschritt. Technologischer Fortschritt führt zu Weiterentwicklungen der Leistungsfähigkeit von Ablaufumgebungen. Diese neuen Ablaufumgebungen existieren parallel zu den weniger leistungsfähigen Vorgängern und erzeugen somit Heterogenität.

Heterogenität ergibt sich also aus verschiedenen Gründen. Homogene Ablaufumgebungen führen auf der Nachfragerseite zu den folgenden positiven Auswirkungen:

- Jedes Objektsystem ist prinzipiell überall ablauffähig, d.h. jede Aufgabe kann überall gelöst werden. Es gibt keine Einschränkungen der Art, daß zwar eine Ablaufumgebung vorhanden ist, aber das Objektsystem in dieser Ablaufumgebung nicht läuft. Bei Ausfall einer

Ablaufumgebung kann diese sofort durch eine andere ersetzt werden, da beide die gleichen Eigenschaften besitzen; Ausfallsicherheit durch Redundanz ist also einfach zu erreichen.

- Da es nur eine Art von Ablaufumgebung gibt, wird der Entwicklungs-, Wartungs- und Lernaufwand geringer, d.h. die Komplexität und damit die Kosten werden reduziert.

Heterogenität führt gegenüber einer homogenen Umgebung zu wachsender Komplexität und steigenden Kosten. Eine Homogenisierung verspricht also sinkende Komplexität und Kosten.

Homogenisierung aller Anwendungsbereiche. Homogenität der Anforderungen der Anwendungsbereiche ergibt sich, wenn eine Obermenge der Anforderungen aller Anwendungsbereiche unter Ausschluß von Widersprüchen gebildet wird. Dies ist aufgrund der Vielzahl der Anwendungsbereiche mit ihren komplexen, teilweise sich gegenseitig ausschließenden Anforderungen nicht zu erwarten. Also wird es weiterhin verschiedene Ablaufumgebungen für verschiedene Anwendungsbereiche geben.

Homogenisierung einzelner Anwendungsbereiche. Die Homogenisierung einzelner Anwendungsbereiche zeigt sich in Standardisierungsprozessen für diese Bereiche. In einem Anwendungsbereich wird durch die Standardisierung Homogenität der Anforderungen erreicht. Aufbauend auf einer stabilen Menge von Anforderungen für einen Anwendungsbereich werden die Leistungen einer Ablaufumgebung definiert, die diese Anforderungen erfüllen.

Bei der Homogenisierung sind die Optimierung der Leistung und die Stabilität der Anforderungen zu beachten.

Optimierung der Leistung. Bei der Bestimmung der Leistungen einer Ablaufumgebung zu einer gegebenen Menge von Anforderungen ist ein Optimierungsproblem zu lösen, wenn die Anforderungen voneinander abhängig und teilweise gegenläufig sind, z.B. Maximierung der Geschwindigkeit vs. Maximierung der Ausfallsicherheit vs. Minimierung des Preises. Durch unterschiedliche Lösungen dieses Optimierungsproblems ergibt sich wiederum Heterogenität.

Stabilität der Anforderungen. Damit eine Ablaufumgebung die aufgestellten Anforderungen erfüllt, darf die festgelegte Menge von Anforderungen für einen Anwendungsbereich nicht mehr variiert werden. Sobald die Anforderungen sich ändern, z.B. durch spezielle Anforderungen eines Nachfragers, beginnt der Prozeß wieder und führt zu heterogenen (alten und neuen) Ablaufumgebungen.

Es zeigt sich, daß auch eine Homogenisierung nicht erfolgversprechend ist. Anspruchsvolle Anwendungsbereiche stellen Anforderungen, die nicht von *einer* Ablaufumgebung erfüllt werden (können). In diesem Fall ist es notwendig, verschiedene Ablaufumgebungen einzubeziehen, für jede Ablaufumgebung ein Objektsystem zu implementieren und diese Objektsysteme kooperieren zu lassen. Kooperierende Objektsysteme müssen damit rechnen, daß jedes Objektsystem eine andere Ablaufumgebung besitzt. Für den Entwickler von Objektsystemen ergibt sich die Konsequenz, daß er sich bei der Entwicklung eines Objektsystems der Heterogenität der Ablaufumgebungen bewußt sein muß, indem er sie einerseits ausnutzen kann, aber andererseits auch mit Unterschieden umgehen oder von ihnen abstrahieren muß.

Heterogenität im Objektsystemmodell. Dem in Kapitel 2 aufgestellten Objektsystemmodell folgend, entstehen die Unterschiede zwischen den Ablaufumgebungen entweder durch unterschiedliche Ablaufsysteme, also Interpreter und Basisobjekte, oder unterschiedliche externe Objekte.

Das Ablaufsystem interpretiert das Objektsystem. Die Objekte und Bindungen des Objektsystems liegen in einer Repräsentation vor, die von dem Interpreter des Ablaufsystems interpretiert wird. Wenn die Interpreter verschiedener Ablaufsysteme unterschiedliche Repräsentationen verarbeiten, paßt die Repräsentation der Objekte des Objektsystems nur zu einem der Interpreter. Die Objekte besitzen keine Ablaufsemantik in einem Ablaufsystem, dessen Interpreter sie nicht interpretieren kann.

Interpreter, welche die gleiche Repräsentation verarbeiten, können dieser Repräsentation verschiedene Semantiken durch den Interpretationsvorgang verleihen. D.h. Interpreter unterscheiden sich nicht nur durch die Repräsentation, die sie verarbeiten, sondern auch durch ihre Implementierung.

Das Ablaufsystem enthält Basisobjekte und die Ablaufumgebung externe Objekte, die dem Objektsystem durch den Interpreter zugänglich sind. Ablaufumgebungen unterscheiden sich darin, welche Basis- und externen Objekte sie mit welcher Funktionalität zur Verfügung stellen. D.h. die Ablaufumgebungen stellen Mengen von Basis- und externen Objekten bereit, die disjunkt sind oder sich überschneiden; die von zwei Ablaufumgebungen zur Verfügung gestellten disjunkten Mengen von Basis- und externen Objekte können Objekte mit gleicher, ähnlicher oder unterschiedlicher Funktionalität enthalten.

Zusammengefaßt ergeben sich folgende Dimensionen der Heterogenität:

- Repräsentation, die vom Interpreter des Ablaufsystems interpretiert wird
- Semantik, die der Interpreter der Repräsentation verleiht
- Menge der internen, Basis- und externen Objekte, die ein Objektsystem ansprechen kann
- Funktionalität der internen, Basis- und externen Objekte

Adaption bedeutet in diesem Modell, daß ein Objektsystem über (reflektive) Mechanismen verfügt, sich an diese Unterschiede in den Ablaufumgebungen anzupassen [Hofmeister, Purtilo 93; Lux 95; Noble, Satyanarayanan 95; Satyanarayanan et al. 95]. In dieser Arbeit werden Adaptionsmechanismen für Kooperationsbeziehungen zwischen Objektsystemen eingeführt.

4.2 Übertragung von Objekten

Ein sendendes Objektsystem überträgt ein Objekt an ein empfangendes Objektsystem (vgl. Abbildung 4.4). Bei der Übertragung sind die Repräsentation des Objekts umzuwandeln und die Bindungen des Objekts an andere Objekte zu behandeln [Herlihy, Liskov 82; Vegdahl 86; Jul et al. 88; Stamos, Gifford 90; Bennett 90; Shub 90; Shapiro 94; Cejtin et al. 95; Birrell et al. 95; Lux 95; Holder, Ben-Shaul 97].

Repräsentation: Damit das Ablaufsystem des empfangenden Objektsystems auf dem übertragenen Objekt Operationen ausführen kann, muß die Repräsentation des Objektes in diejenige umgewandelt werden, die von diesem Ablaufsystem verarbeitet wird. Dabei ist eine Abbildung der Repräsentation des Ablaufsystems des sendenden Objektsystems auf diejenige des Ablaufsystems des empfangenden Objektsystems vorzunehmen.

Bindungen: Das übertragene Objekt wird auch nach seiner Übertragung über seine Bindungen auf Objekte zugreifen. Entweder werden die gebundenen Objekte mit übertragen (*deep copy*), die Bindungen werden in Bindungen an externe Objekte (aus Sicht des empfangenden Objektsystems) umgewandelt oder die Bindungen werden auf interne oder externe Objekte des empfangenden Objektsystems umgesetzt. Wenn die Bindungen auf andere Objekte umgesetzt werden sollen, ist zu klären, wie diese Objekte gefunden werden.

Außerdem ist festzulegen, ob das übertragene Objekt migriert oder kopiert werden soll (vgl. Abbildung 4.5).

Migration: Das Objekt wandert, wird also während oder nach der Übertragung aus dem sendenden Objektsystem entfernt. Dazu werden die lokalen Bindungen von anderen Objekten an das Objekt in externe Bindungen umgewandelt, denn das Objekt wird damit zu einem externen Objekt. Externe Bindungen auf das Objekt werden umgesetzt oder weitergeleitet [Day et al. 93].

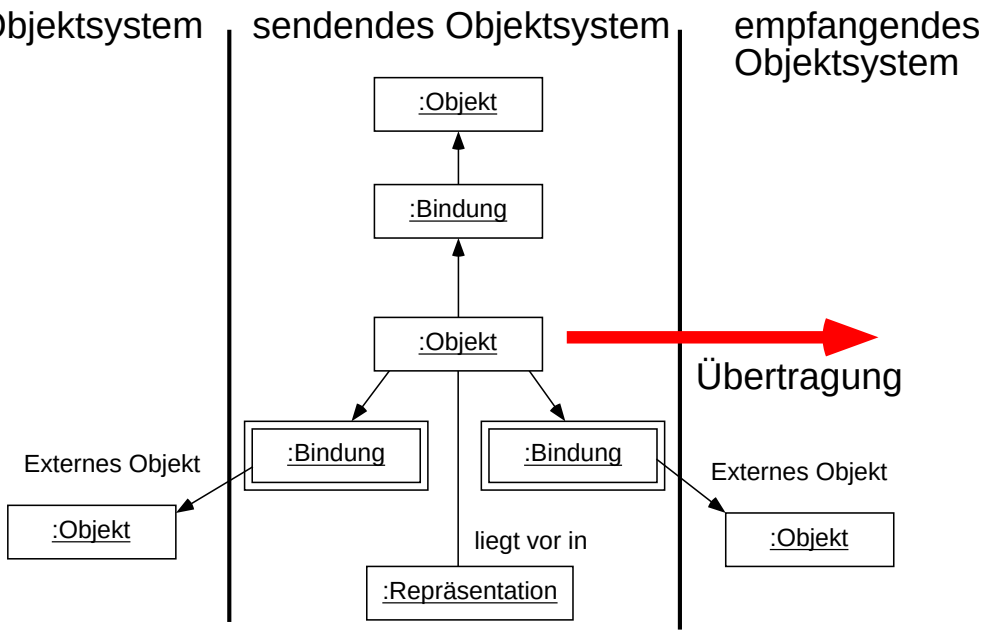

Abbildung 4.4: Übertragung eines Objekts

Kopie: Das Objekt wird kopiert, so daß es nach der Übertragung zweimal, in zwei Objektsystemen, existiert. Es ist zu klären, wie die Objektidentität zu behandeln ist, d.h. ob die Objektidentität bei der Übertragung beibehalten werden kann und soll [Cejtin et al. 95; Cardelli 95; Roy et al. 97]. Inkonsistenzen sind möglich, wenn das Objekt einen Zustand besitzt, der jeweils auf einer Kopie geändert wird, oder wenn nach einer (Rück- oder wiederholten) Übertragung des Objekts bzw. seiner Kopie in den Bereich des gleichen Ablaufsystems ein Vergleich auf Objektidentität zwischen den Objekten (bzw. Kopien) durchgeführt wird.

Bei der Übertragung von Objekten treten vergleichbare Aufgaben wie bei der Persistenzsicherung (vgl. Abschnitt 3.2) auf [Mathiske 96]:

- Semantikerhaltende Abbildung von Repräsentationen und Strukturen.
- Wiederherstellung von seiteneffektbehafteten Ablaufzuständen bei der Übertragung von Prozeßobjekten.

Hier wird auf die besonderen Eigenschaften der Übertragung von Objekten in heterogenen Umgebungen unter besonderer Beachtung der (internen und externen) Bindungen eingegangen.

Objekte beliebigen Typs werden übertragen. Da beide Objektsysteme aufgrund der Autonomie unabhängig voneinander entwickelt werden, kann das empfangende Objektsystem nicht sicher sein, daß das sendende Objektsystem wirklich die erwarteten Objekte schickt. Um die Integrität des empfangenden Objektsystems zu wahren, sind daher zusätzliche Maßnahmen erforderlich, beispielsweise die Übertragung einer Beschreibung der Objekte; dieser Aspekt wird in Kapitel 5 anhand der (dynamischen) Typisierungseigenschaften weiter vertieft.

4.2.1 Portierung

Objekte, welche die Ablaufumgebung wechseln, müssen von einem Entwickler (manuell) an die neue Ablaufumgebung angepaßt werden bzw. sich während des Ablaufs (automatisch) an die neue Ablaufumgebung anpassen. Beide Vorgänge werden als Portierung bezeichnet.

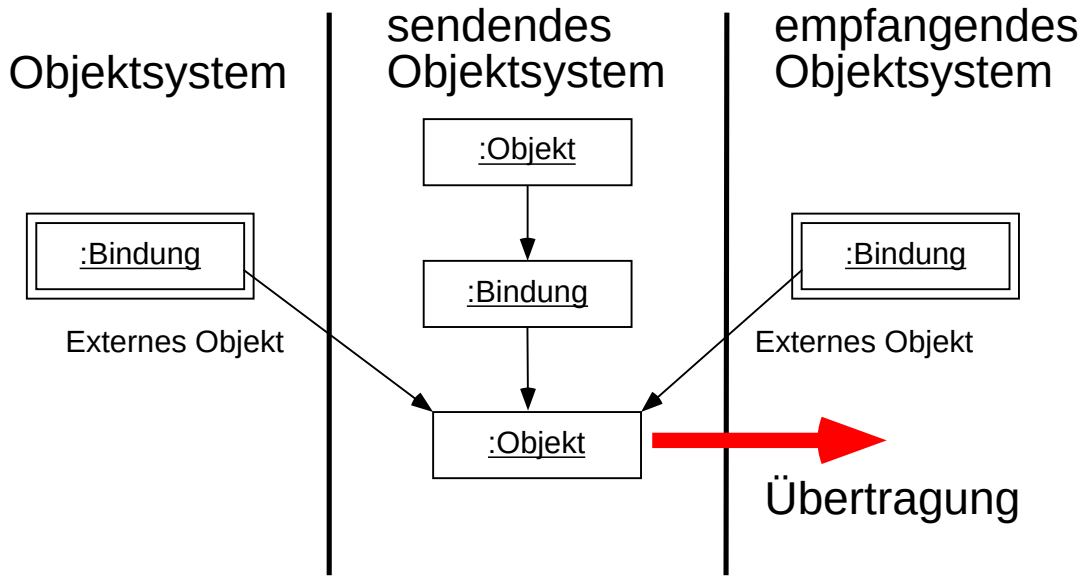

Abbildung 4.5: Migration vs. Kopie eines Objekts

Eine Portierung geht von einer Ablaufumgebung des Objektsystems und der Semantik, die das Objektsystem in dieser Ablaufumgebung besitzt, aus. Nach der Portierung in die neue Ablaufumgebung soll das Objektsystem

- die gleiche Semantik wie in der ursprünglichen Ablaufumgebung besitzen und
- sich im gleichen Ablaufzustand wie vor der Portierung befinden.

Der von einem Entwickler manuell durchgeführte Vorgang der Portierung stellt einen Spezialfall des hier vorgestellten Modells dar. Der Entwickler portiert ein Objektsystem in seinem Initialzustand, d.h. ohne Zustandsänderungen, die während des Ablaufs des Objektsystems stattfinden (vgl. Abschnitt 3.1). Damit können keine Portierungen während der Kooperation zwischen Objektsystemen erfaßt werden, z.B. von dynamisch während des Ablaufs erzeugten Objekten.

Nach der Portierung ist folgende Integritätsbedingung einzuhalten, damit das Objektsystem ablauffähig ist:

Das portierte Objektsystem muß zur Ablaufumgebung passen, d.h.

- Objekte und Bindungen müssen in der vom Interpreter verarbeiteten Repräsentation vorliegen;
- Bindungen dürfen nur in der Ablaufumgebung oder im Objektsystem vorhandene Objekte referenzieren.

Diese Integritätsbedingung kann abgeschwächt werden, wobei dann die Ablauffähigkeit nicht mehr automatisch sicherzustellen ist:

Das Objektsystem darf

- den Interpreter nur Objekte und Bindungen in der von ihm verarbeiteten Repräsentation interpretieren lassen und
- nur Bindungen an vorhandene Objekte nutzen.

D.h. das Objektsystem darf Objekte und Bindungen enthalten, die nicht interpretiert werden, und Bindungen an nicht vorhandene Objekte sind erlaubt, solange sie nicht für die Ausführung relevant werden. Wenn allerdings Objekte interpretiert werden, die nicht in der richtigen Repräsentation vorliegen, oder versucht wird, Bindungen an Objekte zu verfolgen, die nicht vorhanden sind, ist das Ablaufverhalten undefiniert. Per Reflektion können in diesem Fall "just in time" [Plezbert, Cytron 97; Ramme 98] die Objekte in eine interpretierbare Repräsentation umgewandelt oder die fehlenden Objekte angebunden werden.

4.2.2 Repräsentationsumwandlung

Wenn die übertragenen Objekte in einer Repräsentation vorliegen, die von beiden Interpretern verstanden wird, werden sie *portabel* genannt. Die Objekte und Bindungen müssen nicht umgewandelt werden. Dies trifft zum Beispiel auf Java-Applets zu, die in der Repräsentation als Java-Bytecode zwischen verschiedenen Java-Ablaufumgebungen (mit gleicher Java-Maschine als Interpreter im Ablaufsystem) ausgetauscht werden [Gosling 95; Brown, Najork 96].

Wenn die von den Interpretern verarbeiteten Repräsentationen unterschiedlich sind, müssen die Objekte und Bindungen des Objektsystems in die Repräsentation gebracht werden, die von dem neuen Interpreter verarbeitet wird [Herlihy, Liskov 82; Horn, Cahill 91; Huang, Lamb 94; Steensgard, Jul 95] (vgl. Abbildung 4.6). Diese Abbildung sollte semantikerhaltend sein, d.h. die von den Objekten getragene Semantik sollte sich ebenso in der neuen Repräsentation wiederfinden; allerdings kann diese Semantikerhaltung oft nur graduell angegeben werden, beispielsweise wenn Repräsentationen für Bilder unterschiedliche Farbsysteme unterstützen oder Coderepräsentationen für Funktionen (zusammen mit dem verarbeitenden Interpreter) ein anderes Echtzeitverhalten während des Ablaufs zeigen [Horn, Cahill 91].

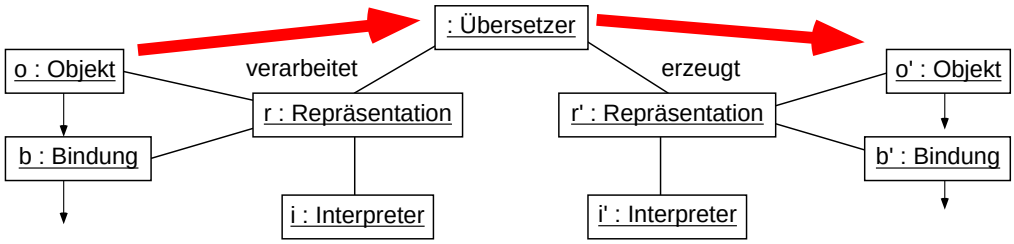

Abbildung 4.6: Portierung von Repräsentationen

Ein Übersetzer (vgl. Abschnitt 2.3.2) bildet die Objekte und Bindungen aus der einen Repräsentation in die andere ab; dabei können noch Zwischenrepräsentationen herangezogen werden, wie folgende Beispiele zeigen:

- Basisdaten: Ein PICT-Bild wird in ein GIF-Bild als Zwischenrepräsentation umgewandelt und dann als JPEG-Bild von dem neuen Interpreter, z.B. einem WWW-Browser, weiterverarbeitet.
- Strukturierte Daten: Eine Relation wird aus einer Oracle-Datenbank der Version 7.3 in eine Datei exportiert und dann in eine Oracle-Datenbank der Version 8.0 eingelesen, so daß Oracle 8.0 als Interpreter diese Daten interpretieren kann.
- Funktionen: Ein in Java-Bytecode vorliegendes Applet wird in Sun-Maschinencode umgewandelt, um von einer Sun-Maschine als Interpreter ausgeführt zu werden.
- Prozesse: Ein auf einer Java-Maschine vorliegender Prozeß wird in eine Repräsentation für eine Sun-Maschine gebracht, so daß diese Maschine den Prozeß interpretieren kann.

Bei der Portierung ist die Forderung nach (anwendungsabhängig graduell abgestufter) Semantikerhaltung zu beachten. Teilweise läßt sich die Semantik nicht wiederherstellen. Wenn beispielsweise das neue Grafikformat weniger Farben vorsieht, geht Semantik verloren. Außerdem sind semantische Abhängigkeiten zwischen den Objekten, die nicht durch Bindungen erfaßt werden, aufzulösen (vgl. Evolution in Abschnitt 3.3):

- Ein Objekt enthält die Größe eines Bildobjekts; durch die Umwandlung des Bildes ändert sich auch die Größe, so daß das Objekt invalidiert wird.
- Ein Feld (*array*) enthält Elemente, auf die durch Indizes zugegriffen wird; wenn die Indizierung des neuen Feldes bei 1 statt bei 0 beginnt, stimmen vorhandene Indizes nicht mehr.
- Ein Prozeß enthält den Abarbeitungszustand einer Funktion, d.h. die Information, welche Anweisung der Funktion als nächstes auszuführen ist. Durch die Umwandlung der Funktion wird diese Information ungültig, wenn die interne Struktur der Funktion (die Sequenz ihrer Anweisungen) sich ändert.

Zustandslose Objekte und zustandsbehaftete Objekte, die seit ihrer Erzeugung nicht verändert wurden, müssen nicht direkt in die neue Repräsentation übersetzt werden [Steensgard, Jul 95]. Stattdessen können auch Objekte herangezogen werden, aus denen diese Objekte durch einen Übersetzungsvorgang, beispielsweise während der Entwicklung (vgl. Kapitel 5), erzeugt wurden (vgl. Abbildung 4.7):

- Eine Java-Funktion, die in der Repräsentation als Java-Bytecode vorliegt, kann aus dem Quelltext direkt in die Repräsentation als Sun-Maschinencode übersetzt werden.
- Ein GIF-Bild, das durch Umwandlung aus einem PICT-Bild entstanden ist, kann in ein JPEG-Bild portiert werden, indem das PICT-Bild in JPEG umgewandelt wird.

Sobald die erzeugten Objekte allerdings nach ihrer Erzeugung verändert werden, sind das Ausgangsobjekt und das erzeugte Objekt nicht mehr äquivalent. Die Änderungen müssen entweder am Ausgangsobjekt oder am neu erzeugten Objekt nachvollzogen werden.

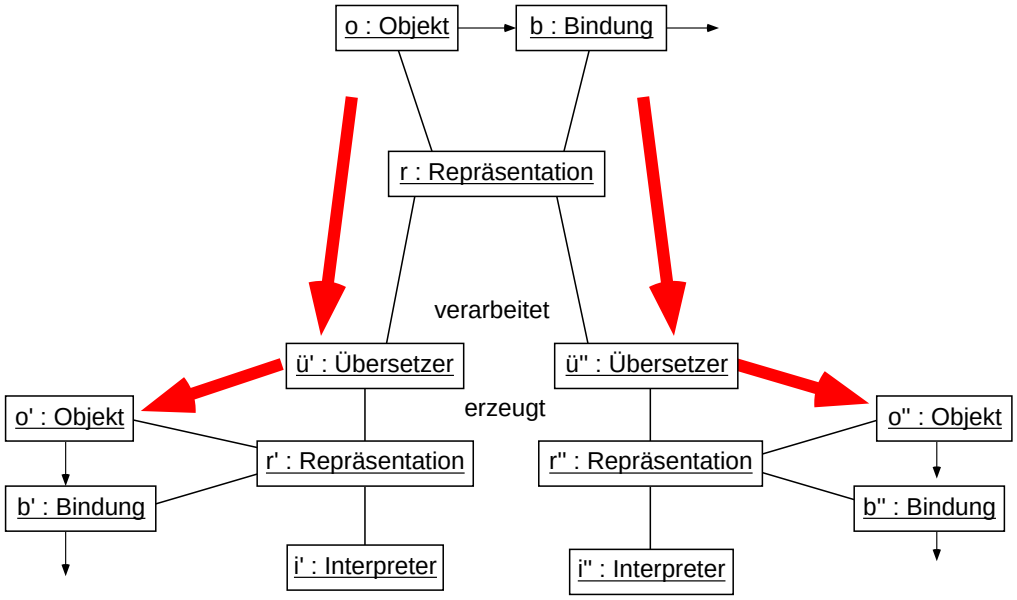

Abbildung 4.7: Portierung von unveränderten oder unveränderlichen Objekten

Die passende Repräsentation kann auch dynamisch in dem Moment erzeugt werden, in dem der Interpreter sie benötigt [Plezbert, Cytron 97]. In diesem Fall muß das Ablaufsystem über einen Übersetzer verfügen, der aus der Repräsentation der Objekte eine für den Interpreter des Objektsystems passende Repräsentation erzeugt [Adl-Tabatabai et al. 96]. Außerdem muß der Interpreter darauf vorbereitet sein, auf Objekte in Repräsentationen zu treffen, die er nicht direkt interpretieren kann, und dann den entsprechenden Übersetzer aufrufen, um eine Repräsentation des Objekts zu erhalten, die er interpretieren kann. Dies setzt eine Kennzeichnung des Objekts und Reflektion voraus (vgl. Abschnitt 3.3).

4.2.3 Übertragung von Bindungen

Bei der Übertragung eines aggregierten Objekts ist zu entscheiden, wie die Bindungen des Objekts behandelt werden sollen; dabei sind sowohl interne als auch externe Bindungen zu betrachten.

Für interne Bindungen ist zu entscheiden, wie mit der Bindung zu verfahren ist (vgl. Abbildung 4.8):

Tiefe Kopie: Das gebundene Objekt wird auch übertragen und die Bindung an dieses Objekt im empfangenden Objektsystem wiederhergestellt. Dabei sind wie bei der Persistenzsicherung (vgl. Abschnitt 3.2) die gemeinsame Nutzung von Objekten und rekursive Strukturen zu beachten [Herlihy, Liskov 82; Mathiske 96].

Umwandlung in externe Bindung: Die Bindung wird in eine externe Bindung (aus Sicht des empfangenden Objektsystems) umgewandelt [Cardelli 95].

Neubindung (auch: Rebindung): Die Bindung wird im empfangenden Objektsystem erneuert, also durch die Bindung an ein anderes Objekt ersetzt [Bennett 90; Horn, Cahill 91; Ramme 98]. Dazu wird die Bindung beim Senden durch eine Beschreibung des neu zu bindenden Objekts ersetzt. Diese Beschreibung wird beim Empfang des Objektes ausgewertet und so das neu zu bindende Objekt gefunden.

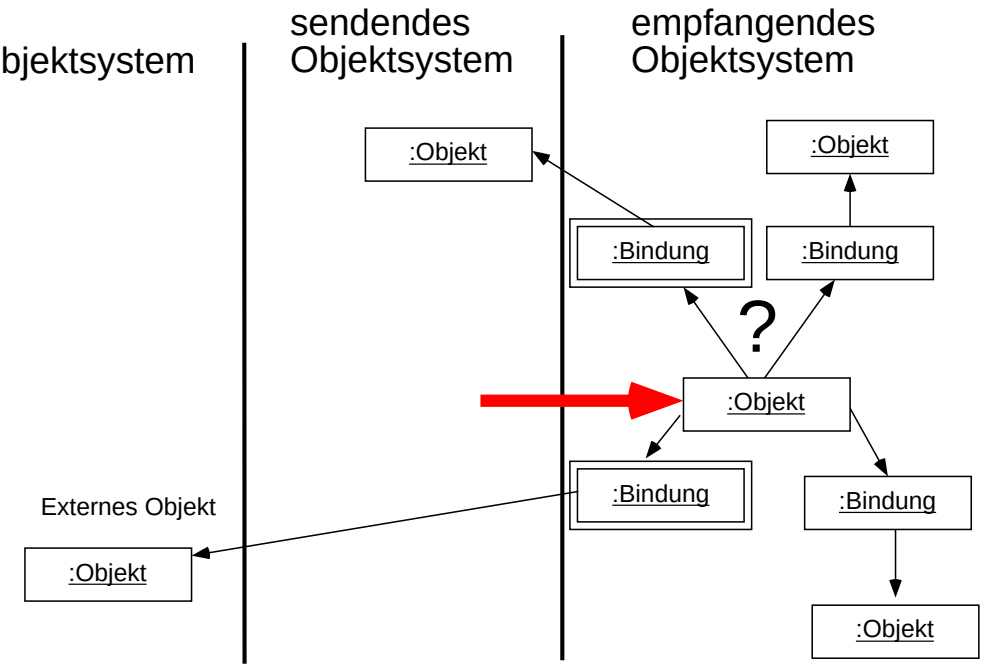

Abbildung 4.8: Übertragung von Bindungen

Bei externen Bindungen sind folgende Fälle zu unterscheiden (vgl. Abbildung 4.9).

1: Umwandlung in interne Bindung. Das gebundene externe Objekt ist ein Objekt des empfangenden Objektsystems (vgl. Fall 1 in Abbildung 4.9). In diesem Fall wird die externe Bindung beim Empfang in eine interne Bindung umgewandelt.

2: Externe Bindung wird beibehalten. Wenn das gebundene externe Objekt auch über Kooperationsbeziehungen des empfangenden Objektsystems zugänglich ist, wird es weiter benutzt (vgl. Fall 2 in Abbildung 4.9). Abhängig von der Kooperationsstrategie für externe Bindungen (vgl. Abschnitt 4.3) muß unter Umständen die Bindung an dieses Objekt erneuert werden, weil eine Kopie der Bindung erfolgt ist.

3: Äquivalentes Objekt wird gebunden. Wenn das gebundene Objekt von dem empfangenden Objektsystem aus nicht zugreifbar ist, müssen zumindest die Bindungen an dieses Objekt geändert werden. Falls im empfangenden Objektsystem oder seiner Ablaufumgebung ein zu dem Ausgangsobjekt äquivalentes Objekt existiert, d.h. ein Objekt, das die gleiche Semantik wie das Ausgangsobjekt besitzt, kann die Bindung an dieses Objekt erfolgen (vgl. Fall 3 in Abbildung 4.9).

4: Semantisch verwandtes Objekt wird über Stellvertreter gebunden. Falls ein äquivalentes Objekt nicht zur Verfügung steht, aber andere Objekte mit (zusammengenommen) ähnlicher Semantik, werden Stellvertreterobjekte (vgl. Abschnitt 3.3) eingesetzt, die eine Abbildung der Semantik auf diese ähnlichen Objekte des empfangenden Objektsystems bzw. seiner Ablaufumgebung vornehmen (vgl. Fall 4 in Abbildung 4.9).

Wenn keine der Bedingungen zutrifft, d.h. die Semantik des externen Objekts nicht erhalten, ersetzt oder abgebildet werden kann, müssen Teile des Objektsystems, welche die nicht ersetzbaren externen Objekte binden, komplett gegen äquivalente Teile ausgetauscht werden, d.h. das Problem des Bindens wird um eine Ebene zurückverlagert auf das Objekt *o*.

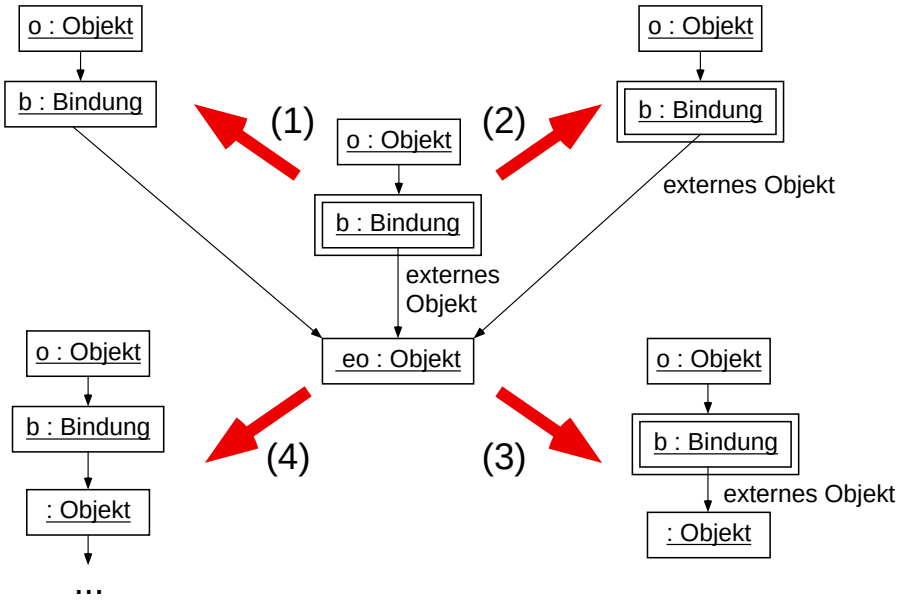

Abbildung 4.9: Neubindung

Für jede Bindung ist festzulegen, wie sie übertragen werden soll, beispielsweise um Adaption an andere Umgebungen zu ermöglichen [Noble, Satyanarayanan 95; Satyanarayanan et al. 95; Holder, Ben-Shaul 97]. Alternativ wird statt der einzelnen Bindungen an ein Objekt für alle

Bindungen an dieses Objekt festgelegt, wie sie zu behandeln sind. Damit ist diese Möglichkeit nicht so flexibel, weil nicht abhängig von dem Kontext, in dem Objekt gebunden ist, eine andere Übertragungsstrategie genutzt werden kann. Dieser Freiheitsgrad kann wiederum erreicht werden, wenn eine Funktion als Entscheidungsgrundlage dient [Ramme 98]: Die Funktion erhält als Parameter beispielsweise das bindende Objekt sowie den Kooperationskontext, der auch das empfangende Objektsystem beschreibt, und entscheidet daraufhin, welche Übertragungsart zu wählen ist.

Dies korrespondiert mit einer flexiblen Art der Neubindung [Ramme 98]: Als Beschreibung der Bindung bzw. des zu bindenden Objekts wird eine Funktion übertragen, die auf der Seite des Empfängers ausgeführt wird; diese Funktion erhält als Parameter einen Zugang zu den auf der Empfängerseite zugänglichen Objekten und liefert das zu bindende Objekt zurück.

Die vorhandenen und bindbaren Objekte werden dazu in einem Repository [Bernstein 98] mit beschreibenden Informationen erfaßt, die eine Auswahl erlauben, beispielsweise über den Namen, den Typ, die Version und eine Semantikbeschreibung. Dieses Repository wird während der Entwicklung (vgl. Kapitel 5) aufgebaut und durch Kooperation mit anderen Objektsystemen erweitert. Ein (einfaches) Beispiel für so ein Repository sind die Klassenbibliotheken von Java, die über den Klassenlader angesprochen werden [Java Team et al. 96].

Diese Funktion kann auch Anpassungen vornehmen, beispielsweise um Versionsunterschiede auszugleichen (vgl. Abschnitt 3.3) oder um verschiedene Plattformen zu adaptieren (vgl. Abschnitt 4.1.4). Wenn die Bindung an ein Objekt nicht erhalten wird, sind wie bei der Evolution (vgl. Abschnitt 3.3) (semantische) Abhängigkeiten zwischen den Objekten zu beachten.

Wenn die Funktion kein passendes Objekt findet, kann sie auch eine weitere Übertragung eines passenden Objekts anstoßen, sofern das empfangende Objektsystem dies durch die Übergabe entsprechender Parameter gestattet, oder eine externe Bindung an das ursprüngliche Objekt realisieren.

Die Bindung muß nicht statisch sein, d.h. die gebundenen Objekte müssen noch nicht existieren. Sie können auch reflektiv in dem Moment erzeugt oder angebunden werden, in dem die Bindung während des Ablaufs des Objektsystems benutzt wird, d.h. das Objekt benötigt wird, um darauf Operationen auszuführen [Shapiro 94]. Dies wird dynamische Bindung genannt; vergleiche dazu auch Abschnitt 3.1.

Die Übertragung einer Funktion, die eine Bindung realisiert, statt einer beschreibenden Information [Kaplan, Wileden 95a; Java Team et al. 96; Mathiske 96] bietet eine erweiterte Flexibilität für die Entwickler von kooperierenden Objektsystemen. Der Sender gibt Strategien für die Wiederherstellung einer Bindung vor und der Empfänger schränkt durch die Übergabe von Parametern an die bindende Funktion diese Strategien auf das von ihm tolerierte Maß ein [Cardelli 95]; somit bleibt die Autonomie beider Partner gewahrt. Sowohl Evolution als auch Heterogenität werden mit dieser Methode beherrschbar.

Diese Methode stellt ein weiteres Beispiel für Reflektion dar: Das Ablaufsystem greift auf eine vom Objektsystem zur Verfügung gestellte Funktion zurück, um eine Funktionalität, hier das Finden eines zu bindenden Objekts, zu realisieren. Damit wird eine höhere Flexibilität erreicht, als wenn diese Funktionalität im Ablaufsystem realisiert wäre. Umgekehrt könnte das Objektsystem ohne die Mithilfe des Ablaufsystems diese Funktionalität nicht realisieren.

4.3 Bindungen zwischen Objektsystemen

Ein (importierendes) Objektsystem bindet ein Objekt eines anderen (exportierenden) Objektsystems (vgl. Abbildung 4.10). Bindungen an externe Objekte werden beispielsweise eingesetzt, wenn entfernte Objektsysteme Zugriff auf eine lokale Ressource bekommen sollen oder ein migrierendes Objekt beim Verlassen eines Objektsystems eine Bindung an ein immobiles lokales Objekt hält [Finkelstein et al. 94; Mathiske 96], um aus der Ferne oder nach der Rückkehr darauf zuzugreifen. Im Kontext des World Wide Web stellen Hyperlinks in Form von URLs externe Bindungen dar [Lennon 97]; in persistenten verteilten Objektsystemen finden sich weitere Implementierungen [Kaplan, Wileden 95a; Mathiske 96]. Externe Bindungen können sowohl ortsabhängig als auch ortsunabhängig sein [Day et al. 93; Auerbach et al. 94].

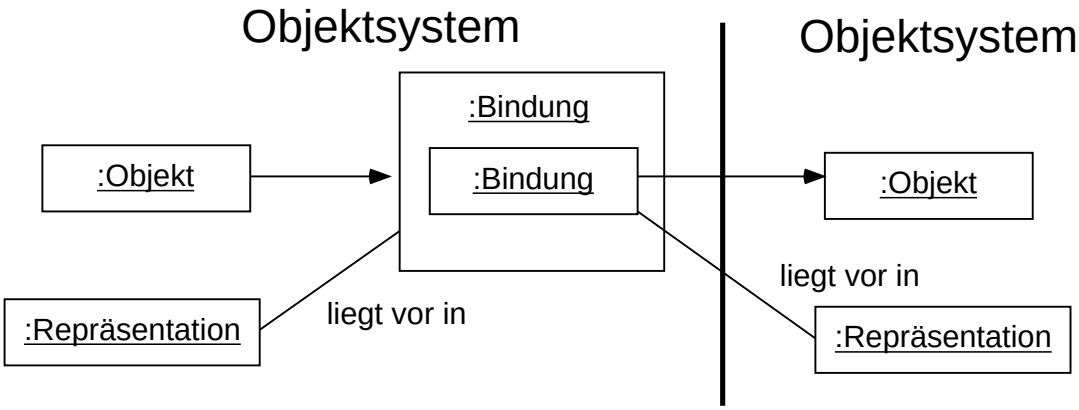

Abbildung 4.10: Externe Bindung

Folgende Rollen sind zu unterscheiden:

Importierendes Objektsystem: Das importierende Objektsystem bindet ein externes Objekt und stößt auf ihm Operationen an.

Importierendes Ablaufsystem: Das Ablaufsystem des importierenden Objektsystems sieht eine Bindung an ein externes Objekt, auf dem es keine Operationen ausführen kann.

Exportierendes Objektsystem: Das exportierende Objektsystem gibt eine Bindung an eines seiner Objekte heraus, das damit zum externen Objekt wird.

Exportierendes Ablaufsystem: Das exportierende Ablaufsystem führt nach Anforderung von außen, d.h. vom importierenden Objektsystem oder Ablaufsystem, Operationen auf dem externen Objekt aus.

Das exportierende Ablaufsystem exportiert eine Bindung an das beschriebene Objekt in einer von ihm interpretierbaren Repräsentation. Das importierende Ablaufsystem bettet diese Bindung in seine Repräsentation ein, so daß sie nicht als eine Bindung an ein internes Objekt interpretiert wird. Diese Bindungen stellen aus Sicht des importierenden Objektsystems und seines Ablaufsystems Bindungen an externe Objekte dar. Die externen Objekte stehen nicht unter der Kontrolle des importierenden Ablaufsystems. Das importierende Ablaufsystem kann nicht direkt Operationen auf den externen Objekten ausführen, sondern delegiert die Ausführung dieser Operationen an das exportierende Ablaufsystem (vgl. Abschnitt 4.3.2). Nur durch Kooperation mit dem exportierenden Ablaufsystem kann das importierende Ablaufsystem sicherstellen, daß die Bindungen an externe Objekte gültig bleiben, d.h. daß das exportierende Ablaufsystem die Objekte nicht freigibt, solange noch externe Bindungen auf die Objekte existieren (vgl. Abschnitt 4.3.3).

Da beliebige Objekte durch externe Bindungen gebunden werden können, kann nicht statisch festgelegt werden, welche Art Objekt durch eine externe Bindung gebunden ist. Dieser (dynamische) Typisierungsaspekt wird in Kapitel 5 vertieft.

4.3.1 Eigenschaften externer Bindungen

Damit ein Objekt in einer externen Bindung gebunden werden kann, muß eine exportierbare Bindung existieren. Externe Bindungen heben sich durch drei Eigenschaften von anderen Bindungen ab:

Stabilität: Eine exportierbare Bindung wird an andere Objektsysteme übertragen, d.h. sie verläßt den Einflußbereich des exportierenden Ablaufsystems. Bei der Rückübertragung wird anhand

der Bindung das gebundene Objekt identifiziert. Die Bindung sollte daher langfristig stabil sein, d.h. sie sollte sich nicht mit der Zeit ändern, z.B. durch lokale Speicherrückgewinnungsoperationen.

Lebensdauer: Das durch eine exportierte Bindung gebundene Objekt darf nicht (ohne weiteres) freigegeben werden, auch wenn das lokale Objektsystem das Objekt nicht mehr bindet und es daher freigegeben werden könnte, ohne die Integrität des lokalen Objektsystems zu verletzen. Wenn die Integrität der importierenden Objektsysteme maximal zugesichert werden soll, darf das Objekt nie mehr freigegeben werden, denn es könnte immer noch irgendwo eine externe Bindung auf das Objekt existieren. Um Speicher zurückgewinnen zu können, sollte das exportierende Ablaufsystem die Lebensdauer der externen Bindung einschränken können.

Operationen: Auf den durch externe Bindungen gebundenen Objekten sind Operationen definiert, die andere Ablaufsysteme im exportierenden Ablaufsystem auf dem gebundenen Objekt auslösen, beispielsweise das Suspendieren eines Prozesses oder der Aufruf einer Funktion. Die Operationen sollten abhängig von dem aufrufenden Ablaufsystem oder dem gebundenen Objekt durch das exportierende Objektsystem eingeschränkt werden können. Beispielsweise könnte ein reflektiver Rückruf an das exportierende Objektsystem erfolgen, wenn eine Operation auf dem exportieren Objekt ausgelöst wird.

Die externe Bindung wird vom exportierenden Ablaufsystem erzeugt, entweder automatisch oder nach Aufforderung durch das Objektsystem. Bei der Erzeugung können die Lebensdauer und die Rechte (bezüglich Weitergabe und Operationen auf dem gebundenen Objekt) eingeschränkt werden, wenn die Kooperationsstrategie dies unterstützt.

Eine externe Bindung kann von dem importierenden Ablaufsystem nicht verfolgt werden, weil es die Repräsentation nicht interpretieren kann, d.h. sie ist für das Ablaufsystem opaque. Das importierende Ablaufsystem bettet die externe Bindung also uninterpretiert in seine Repräsentation für Bindungen ein. Das importierende Ablaufsystem muß einen Verweis auf das exportierende Ablaufsystem behalten, damit die externe Bindung zugeordnet werden kann.

Die externe Bindung stammt von dem exportierenden Ablaufsystem, in dessen Kontext sich das Objekt befindet, das gebunden wird. Dieses Ablaufsystem kennt die Repräsentation der Bindung, kann sie also interpretieren. Bei der Rückübertragung einer exportierten Bindung muß dem exportierenden Ablaufsystem mitgeteilt werden oder es muß automatisch erkennen, daß ein übertragenes Bitmuster eine exportierte Bindung auf eines seiner Objekte beinhaltet; insbesondere muß diese Bindung von Bindungen unterschieden werden, die von anderen Ablaufsystemen exportiert werden. Ansonsten kann das Ablaufsystem die Bindungen nicht als solche interpretieren.

Das importierende Ablaufsystem kann keine Annahmen über die Gültigkeit der externen Bindungen machen, da das exportierende Ablaufsystem autonom ist und somit nicht erreichbar sein, den Zugriff verweigern oder das gebundene Objekt freigeben kann. Das Kooperationsprotokoll sollte diese Fälle abdecken und das importierende Ablaufsystem eine Behandlung dieser Fälle vorsehen.

Das exportierende Ablaufsystem weiß nie, ob außerhalb seines Einflußbereichs noch eine externe Bindung auf ein Objekt seines Objektsystems existiert, wenn die externe Bindung einmal seinen Einflußbereich verlassen hat. Es können also Zugriffe auf Objekte erfolgen, die von dem Ablaufsystem freigegeben wurden. Das exportierende Ablaufsystem sollte mit diesen Fällen rechnen und sie geeignet behandeln, z.B. indem es keine externe Bindung zweimal vergibt.

Ein Objektsystem importiert externe Bindungen durch sein Ablaufsystem. Da das gebundene Objekt nicht dem Einfluß des Ablaufsystems des importierenden Objektsystems unterliegt, muß das Objektsystem mit einem anderen Verhalten als bei lokalen Objekten rechnen; beispielsweise kann das Verfolgen einer Bindung fehlschlagen, was bei lokalen Bindungen nicht passieren kann. Das Objektsystem sollte daher mit externen Objekten anders umgehen und geeignet auf Ausnahmen reagieren, auch wenn in der Entwicklung der Zugriff transparent ist.

Externe Objekte können vom exportierenden Ablaufsystem freigegeben werden, obwohl eine externe Bindung auf das Objekt existiert. Dementsprechend kann die externe Bindung ungültig

werden. Wenn das importierende Ablaufsystem merkt, daß eine Bindung ungültig ist, kann es das Objektsystem darüber informieren, beispielsweise durch einen reflektiven Rückruf. Ansonsten wird spätestens beim Zugriff auf dieses Objekt, beispielsweise wenn eine Operation darauf ausgeführt werden soll, eine Ausnahme auftreten. Diese Ausnahmen treten nicht im Fall lokaler Bindungen auf, weil das Ablaufsystem für lokale Bindungen garantiert, daß gebundene Objekte nicht freigegeben werden.

Auf externen Objekten stehen unter Umständen andere Operationen zur Verfügung als auf lokalen Objekten, weil die Ablaufsysteme sich unterscheiden (vgl. Heterogenität in Abschnitt 4.1.4). Außerdem kann das exportierende Ablaufsystem die Operationen weiter einschränken, beispielsweise aus Sicherheitserwägungen oder auf Wunsch des exportierenden Objektsystems.

4.3.2 Operationen auf externen Objekten

Das importierende Objektsystem löst auf Objekten des exportierenden Objektsystems Operationen aus (vgl. Abbildung 4.11), beispielsweise das Suspendieren eines Prozesses oder den Aufruf einer Funktion [Birrell, Nelson 84; Cejtin et al. 95]. Diese Objekte sind durch externe Bindungen gebunden. Das importierende Ablaufsystem kann die Bindungen und die Objekte nicht interpretieren. Die Operationen werden daher an das exportierende Ablaufsystem delegiert [Orfali, Harkey 97; Tolksdorf 93; Tolksdorf 95].

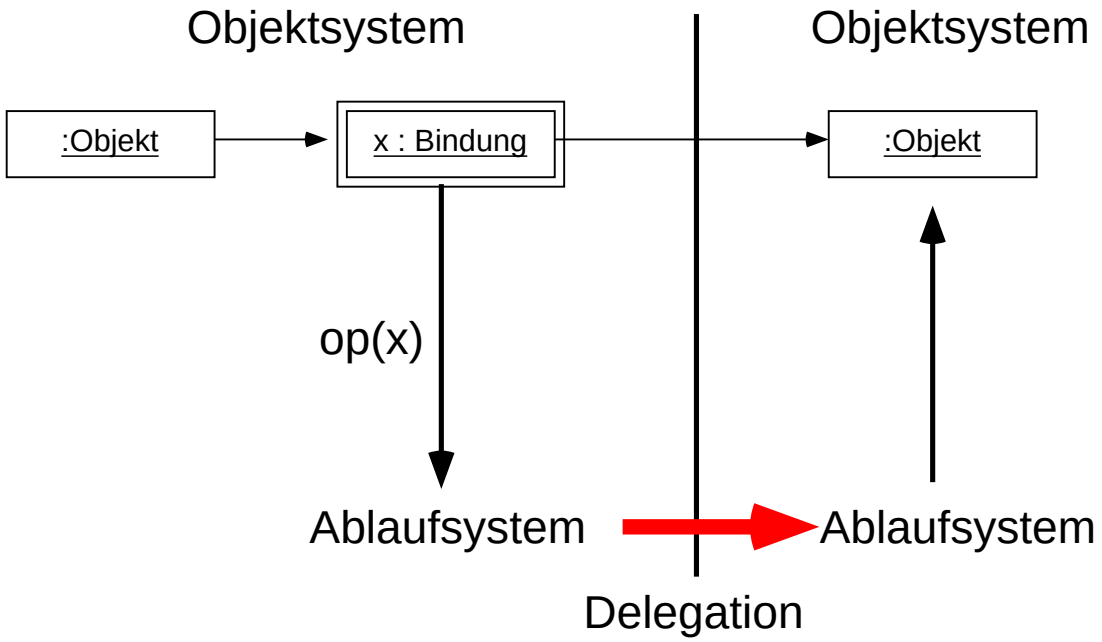

Abbildung 4.11: Operationen auf externen Objekten

Die Delegation kann auf mehrere Arten durchgeführt werden:

Explizit: Das Objektsystem stößt explizit die gewünschten Operationen auf dem externen Objekt an. Beispielsweise wird explizit die Inkrementoperation auf dem externen Objekt *extOp* angestoßen.

```
let extObj = bindExternalInteger()
execExternalOp(incOp extObj)
```

Dies setzt voraus, daß das Objektsystem (bzw. sein Entwickler) sich bewußt ist, daß es sich bei *extObj* um ein externes Objekt handelt.

Reflektiv: Das Objektsystem weist dem externen Objekt *extObj* reflektiv eine Funktion zu, die das Ablaufsystem aufruft, wenn es Operationen wie *incOp* auf dem Objekt ausführt:

```
let extObj = bindExternalInteger()
reflectiveOp(incOp extObj  fun(o) execExternalOp(incOp o)
extObj++
```

In diesem Fall weiß das Objektsystem nur bei der Erzeugung des externen Objektes *extObj* von seinem Sonderstatus, kann es aber anschließend wie ein internes Objekt behandeln.

Implizit: Das Ablaufsystem setzt automatisch die Operationen auf dem externen Objekt *extObj* um:

```
let extObj = bindExternalInteger()
extObj++
```

Das Objektsystem weiß in diesem Fall nichts von dem Sonderstatus des externen Objekts *extObj*.

Dabei ist zu beachten, daß aufgrund der Heterogenität das exportierende Ablaufsystem andere Operationen auf den Objekten unterstützen kann als das importierende Ablaufsystem, so daß es keine eindeutige Entsprechung der Operationen auf den Objekten gibt. Außerdem kann die Menge der exportierten, also von außen auslösbaren Operationen durch die Schnittstelle des exportierenden Ablaufsystems oder durch das exportierende Objektsystem eingeschränkt sein.

Einige Operationen verlangen Parameter, die mit übertragen werden; Bindungen können in externe Bindungen umgewandelt, die gebundenen Objekte kopiert, durch andere Objekte substituiert oder neu gebunden werden [Kono et al. 94; Mira da Silva et al. 96]. Bindungen an externe Objekte werden in der dafür vorgesehenen Repräsentation übermittelt; dabei hängt es vom Herausgeber der externen Bindung und dem empfangenden Ablaufsystem ab, welche Operationen auf diesen Bindungen möglich sind. Bei Bindungen an interne Objekte kann eine exportierbare Bindung erzeugt werden, die übertragen wird. Wenn statt einer externen Bindung an ein Objekt das Objekt inklusive seiner Bindungen übermittelt werden soll, kommen die Mechanismen zum Austausch von Objekten (vgl. Abschnitt 4.2) zum Einsatz. Analoges gilt für Ergebnisse der Operationen.

Das folgende Beispiel zeigt, wie im *lokalen* Fall schrittweise das Alter einer Person *peter* ausgelesen, inkrementiert, neu gesetzt und ausgegeben wird, wobei die Ausgabe erfolgt, indem die Funktion *printInt* an eine Funktion *doAge* übergeben wird, die wiederum *printInt* mit dem Argument *age* aufruft.

```
let printInt(i :Integer) = ...
let peter = tuple
  let age = 23
  let setAge(x :Int) age := x
  let doAge(f :Fun(:Int) :Ok) = f(age)
end
let petersAge = peter.age
inc(peter.age)
peter.setAge(35)
peter.doAge(printInt)
```

Wenn ein Objektsystem dagegen eine *externe* Bindung *importedPeter* an das Objekt *peter* hält, werden die Operationen weitergeleitet. Das Auslesen des Alters wird in eine *dotOp* umgewandelt, die auf der externen Bindung *importedPeter* arbeitet; es wird davon ausgegangen, daß *peter* das Alter *age* als erste Bindung enthält. Zurückgeschickt wird eine externe Bindung *importedPetersAge*, die eine Bindung an das Alter enthält. Dieses Objekt wird inkrementiert.

```
let importedPetersAge = cooperation.dotOp(importedPeter 0)
execExternalOp(incOp importedPetersAge)
```

Das Setzen des Alters auf den neuen Wert erfordert zwei Operationen: Zuerst wird mit einer *dotOp* eine externe Bindung *importedPetersSetAge* an die Funktion *setAge* des Objekts *peter* erzeugt. D.h. der Rückgabewert dieser *dotOp* Operation ist eine externe Bindung. Diese externe Bindung kann dann in einer *callOp* benutzt werden, um einen Funktionsaufruf der Funktion *setAge* auszulösen. Der Funktionsaufuf erhält als Parameter das Basisobjekt *35*.

```
let importedPetersSetAge = cooperation.dotOp(importedPeter 1)
cooperation.callOp(importedPetersSetAge 35)
```

Für die Ausgabe des Alters wird zuerst eine exportierbare Bindung *exportedPrintInt* an die lokale Funktion *printInt* erzeugt. Nachdem die externe Bindung *importedPetersDoAge* an die Funktion *doAge* zur Verfügung steht, wird diese Funktion aufgerufen und erhält als Argument die exportierte Bindung *exportedPrintInt*.

```
let exportedPrintInt = export.new(printInt)
let importedPetersDoAge = cooperation.dotOp(importedPeter 2)
cooperation.callOp(importedPetersDoAge exportedPrintInt)
```

Bei der Ausführung der Funktion *doAge* wird die (externe) Funktion *printInt* aufgerufen, d.h. der Programmfluß wechselt zwischen den beiden Objektsystemen [Cohrs et al. 88] (vgl Abbildung 4.12) Dies entspricht in expliziter Form dem in Kapitel 3 gezeigten Rückruf des Ablaufsystems in das Objektsystem, beispielsweise bei der Freigabe von Objekten. Die dort gezeigten Rückrufe sind Spezialfälle dieses Falls; der Spezialfall zeichnet sich dadurch aus, daß ein Objektsystem und sein Ablaufsystem kooperieren.

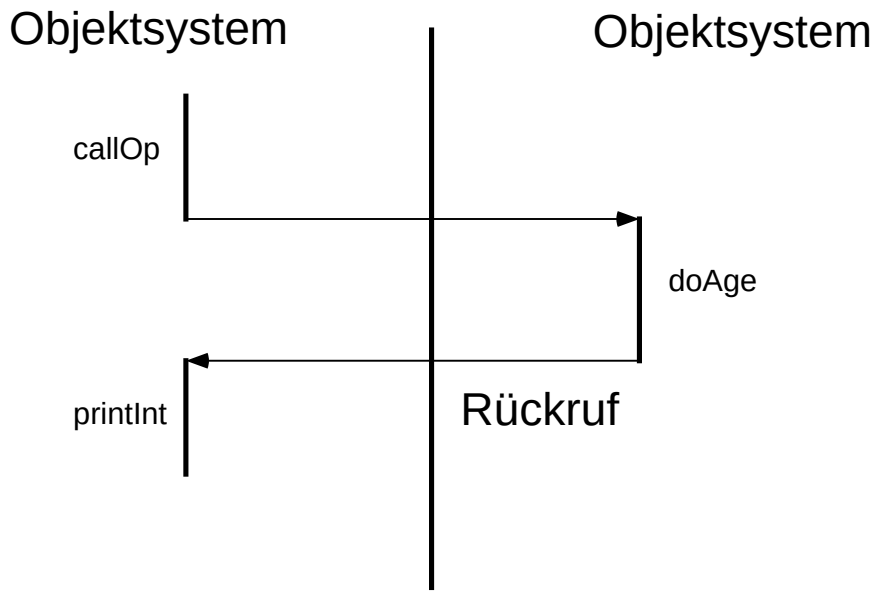

Abbildung 4.12: Rückruf in Kooperationen

4.3.3 Objektfreigabe und externe Bindungen

Grundlegende Implementierungen verteilter Speicherverwaltungen unter der Annahme der hohen Verfügbarkeit beschreiben [Birrell et al. 93; Piquer 96; Maheshwari, Liskov 97c; Maheshwari,

Liskov 97a; Maheshwari, Liskov 97b], insbesondere die Freigabe von nicht mehr erreichbaren, verteilten, zyklischen Bindungen. Aufbauend auf diesen Arbeiten wird hier der Fokus auf autonome, persistente Objektsysteme in heterogenen Umgebungen gelegt, in denen die Objektsysteme nicht ständig erreichbar sind und einen wesentlichen Einfluß auf die Kooperationsstrategie ausüben.

Durch externe Bindungen wird die Autonomie der Objekt- und Ablaufsysteme eingeschränkt:

- Sie binden Objekte, die nicht in ihrem Einflußbereich liegen.
- Ihre Objekte werden durch andere Objektsysteme gebunden.

Insbesondere wird die Objektfreigabe (vgl. Abschnitt 3.1.4) eingeschränkt [Birrell et al. 93]:

- eine zu frühe Objektfreigabe gefährdet die Integrität des importierenden Objektsystems,
- eine zu späte Freigabe schränkt die Ressourcen des exportierenden Objektsystems ein.

Daher sind Kooperationsstrategien bzw. -protokolle zu finden, die den Autonomiebestrebungen beider Kooperationspartner gerecht werden; eine Festlegung ist aufgrund der applikationsabhängig unterschiedlichen Anforderungen nicht sinnvoll [Saltzer et al. 84; Noble, Satyanarayanan 95]. Daher ist eine Offenheit für applikationsspezifische Anpassungen zu wahren, beispielsweise durch reflektive Mechanismen [Schill et al. 95]. Aufgrund unsicherer Netzverbindungen und der prinzipiellen Autonomie der Objektsysteme ist insbesondere in heterogenen Umgebungen nicht gewährleistet, daß die kooperierenden Objektsysteme jederzeit erreichbar sind und bleiben [Birrell et al. 93; Rosenblum, Wolf 97]; diesen Umstand müssen die Protokolle einbeziehen. Im Folgenden werden exemplarisch solche Protokolle diskutiert; durch anwendungsabhängige Kooperationskontexte ergeben sich weitere Protokolle.

Anforderungen

Aufgrund der Autonomie der Objekt- und Ablaufsysteme kann ein importierendes Ablaufsystem keine Annahmen über interne Objekte auf externe Objekte übertragen, d.h. es kann weder das gebundene externe Objekt freigeben (vgl. Abschnitt 3.1.4) noch sicherstellen, daß das exportierende Ablaufsystem das gebundene Objekt nicht freigibt oder Operationen auf diesem Objekt erlaubt (vgl. Abschnitt 4.3.2). In Erweiterung der Integritätsbedingungen aus Abschnitt 3.1.4 sollte daher folgendes gelten, um die Ablauffähigkeit des Objektsystems zu wahren:

> Ein gebundenes Objekt darf nicht freigegeben werden, solange noch eine *interne oder externe* Bindung existiert, die es bindet.

Objekte können freigegeben werden, sobald es keine internen oder externen Bindungen gibt, die sie binden. Dem Ablaufsystem des gebundenen Objekts ist aufgrund der Autonomie der Objekt- und Ablaufsysteme nicht bekannt, ob ein Objekt in externen Bindungen gebunden wird. Während ein Objekt, das nicht mehr lokal gebunden ist, das somit nicht mehr zugreifbar ist, von der Freispeicherverwaltung freigegeben werden kann, sollte dies nicht passieren, wenn noch externe Bindungen auf dieses Objekt existieren. Es stellt sich die Frage, wann ein Objekt freigegeben werden kann. Bei der Wahl einer Lösung sind folgende Kriterien gegeneinander abzuwägen:

Kooperationsfähigkeit: Um kooperationsfähig zu sein, sollten externe Bindungen zwischen Objektsystemen existieren können. Ein importierendes Objektsystem sollte alle Operationen auf externen Objekten ausführen können, die auch lokal für diese Objekte definiert sind, und jederzeit auf diese Objekte zugreifen können, als wären sie Bestandteil des Objektsystems.

Autonomie: Ein Objektsystem sollte über alle seine Objekte frei verfügen können. Ein exportierendes Objektsystem sollte darüber bestimmen können, welche Objekte per externer Bindung zugreifbar sind, welche Operationen auf diesen Objekten möglich sind und welche Objekte es dem externen Zugriff entzieht oder durch andere Objekte ersetzt, beispielsweise wenn das gebundene Objekt veraltet ist oder eine neue Version existiert (vgl. Abschnitt 3.3).

Speicherrückgewinnung: Da der Speicherbereich eines Objektsystems durch die technischen Eigenschaften des Ablaufsystems begrenzt wird, sollten nicht mehr benutzte (d.h. ungebundene) Objekte freigegeben werden, um somit Speicher zurückzugewinnen.

Fehlervermeidung: Ein Zugriff auf ein nicht existierendes oder nicht erreichbares Objekt per externer Bindung ist ein Fehler, der bei lokalen Bindungen nicht auftreten kann, wenn ein Ablaufsystem die Integritätsbedingungen aus Abschnitt 3.1.4 einhält.

Diese Kriterien führen zu widersprüchlichen Anforderungen: Maximale Autonomie schließt Kooperationsfähigkeit aus, denn jede Kooperation schränkt die Autonomie ein; Speicherrückgewinnung führt potentiell zu Fehlern, wenn Objekte freigegeben werden, solange noch externe Bindungen auf das Objekt existieren.

Durch Protokollvereinbarungen zwischen den Ablaufsystemen der kooperierenden Objektsysteme soll ein Ausgleich zwischen Kooperationsfähigkeit und Autonomie einerseits sowie Speicherrückgewinnung und Fehlervermeidung andererseits erreicht werden (vgl. auch das *end-to-end* Argument in [Saltzer et al. 84]). Im Folgenden werden Basismechanismen vorgestellt, aus denen solche Protokolle aufgebaut sind. Ein konkretes Kooperationsprotokoll besteht aus mehreren dieser Basismechanismen, die zusammenwirken und der Kooperation einen anwendungsabhängigen Charakter verleihen, indem sie beispielsweise die Autonomie des exportierenden Objektsystems oder die Fehlervermeidung betonen.

Exportierende und importierende Objekt- bzw. Ablaufsysteme verfolgen gegensätzliche Ziele bezüglich der externen Bindungen:

- Das exportierende Ablaufsystem möchte das durch die externe Bindung gebundene Objekt freigeben, wenn es nicht mehr von seinem Objektsystem benutzt wird, um Speicher zurückzugewinnen. Außerdem möchte sein Objektsystem unter Umständen kontrollieren, welches Objektsystem die externe Bindung erhält und wie lange die Bindung gültig bleibt.

- Das importierende Ablaufsystem möchte die externe Bindung aufrechterhalten, solange sein Objektsystem noch darauf zugreifen kann, um Fehler zu vermeiden. Außerdem möchte das Objektsystem unter Umständen die externe Bindung kopieren und in anderen Kooperationsbeziehungen weitergeben.

Die beiden Ablaufsysteme müssen kooperieren, damit beide ihre Ziele erreichen. Die im Folgenden vorgestellten Kooperationsstrategien erreichen diese Ziele in unterschiedlichem Maße. Es werden zwei Ansätze vorgestellt, die jeweils die Autonomie eines der beiden Kooperationspartner betonen, aber insbesondere bei Verbindungsfehlern zu unerwünschtem Verhalten führen können. Daraus wird ein gemeinsamer Ansatz abgeleitet, der einen Kompromiß zwischen Autonomie, Fehlervermeidung und Speicherrückgewinnung für beide Partner bietet. Aus dem anwendungsabhängigen Kooperationskontext heraus muß eine passende Kooperationsstrategie ausgewählt werden.

Maximale Autonomie des importierenden Ablaufsystems

Das exportierende Ablaufsystem zählt, wie oft eine externe Bindung herausgegeben wurde. Dies setzt voraus, daß das Ablaufsystem die Herausgabe einer externen Bindung während einer Kooperation der Objektsysteme feststellen kann. Die importierenden Ablaufsysteme melden dem exportierenden Ablaufsystem, wenn sie die externe Bindung kopieren und wenn sie diese nicht mehr benötigen, weil das importierende Objektsystem sie nicht mehr erreichen kann. Sobald alle herausgegebenen externen Bindungen und ihre Kopien nicht mehr benötigt werden, kann das gebundene Objekt freigegeben werden.

Auf der Seite des importierenden Ablaufsystems wird zu der externen Bindung reflektiv eine Funktion registriert, die von der Speicherverwaltung aufgerufen wird, wenn das Objekt freigegeben werden soll (vgl. Abschnitt 3.1.4). In diesem Fall wird dem exportierenden Ablaufsystem eine Nachricht geschickt, daß die externe Bindung *exportedPeter* (hier als *importedPeter* abgelegt) nicht mehr benötigt wird, sobald sie nicht mehr erreicht werden kann:

```
let cleanUpImport(i :import.T) :Ok =
  import.free(i)
gc.register(importedPeter cleanUpImport)
```

Dieser Ansatz beschränkt die Autonomie des exportierenden Ablaufsystems, denn es ist auf die Kooperation des importierenden Ablaufsystems angewiesen, damit dieses die Bindung an das Objekt freigibt. Außerdem baut es auf der Zuverlässigkeit der Verbindung zwischen den beiden Ablaufsystemen auf. Wenn diese Verbindung unzuverlässig ist, ergeben sich folgende Probleme:

- Wenn eine herausgegebene externe Bindung nicht vom importierenden Objektsystem registriert wird oder wenn die Meldung der Freigabe einer externen Bindung nicht beim exportierenden Objektsystem ankommt, bleibt das Objekt bestehen, obwohl keine externen Bindungen mehr auf das Objekt existieren.
- Wenn ein Ablaufsystem die externe Bindung doppelt erhält, obwohl nur eine herausgegeben wurde, werden mehr externe Bindungen freigegeben, als registriert sind. Dadurch wird das Objekt freigegeben, obwohl noch externe Bindungen auf das Objekt existieren.

Maximale Autonomie des exportierenden Ablaufsystems

Das exportierende Ablaufsystem kennzeichnet externe Bindungen durch ein Verfallsdatum bzw. eine Lebensdauer [Doorn et al. 95]. Es sichert damit zu, daß das importierende Ablaufsystem innerhalb der Lebensdauer auf die externe Bindung zugreifen kann. Die Lebensdauer kann verlängert werden, wenn das exportierende Ablaufsystem dies unterstützt. Das importierende Ablaufsystem ist dafür verantwortlich, die Lebensdauer rechtzeitig zu verlängern, wenn sein Objektsystem die externe Bindung noch benötigt. Wenn die Lebensdauer abgelaufen ist, sollte es Operationen auf diesen Objekten von vornherein zurückweisen oder das importierende Objektsystem reflektiv davon in Kenntnis setzen.

Das folgende Beispiel zeigt, wie auf der Seite des importierenden Ablaufsystems eine automatische Verlängerung der Lebensdauer der externen Bindung *importedPeter* eingerichtet wird. Dazu wird ein Dienst *EventTimer* des Ablaufsystems herangezogen, der die mit *extendLife* erzeugte Funktion aufruft, wenn die Lebensdauer der externen Bindung *importedPeter* fast abgelaufen ist. Über *latency* wird eine Latenzzeit zum Benachrichtigen des exportierenden Ablaufsystems in Rechnung gestellt. Die Lebensdauer wird um die Zeitspanne *duration* verlängert. Falls die Verlängerung fehlschlägt, wird das Objektsystem über die Funktion *notifyExpiration* davon in Kenntnis gesetzt. Außerdem wird die automatische Verlängerung abgebrochen, wenn das Objekt *importedPeter* freigegeben, die externe Bindung also nicht mehr vom Objektsystem benötigt wird. Dabei ist zu beachten, daß die Bindungen an *importedPeter*, die von dem Objekt *importedPeterLifeExtender* ausgehen, nicht gezählt werden dürfen, denn ansonsten wird das Objekt nie freigegeben; hier kommen die in Kapitel 3 eingeführten schwachen Bindungen [Birrell et al. 93] zum Einsatz.

```
let extendLife(i :import.T  duration :TimeInterval
    notifyExpiration :Fun(:import.T) :Ok)(self :eventTimer.T) :Ok =
  begin
    let until = expiration(i)+duration
    let prolongAccepted = import.prolong(i until)
    if prolongAccepted then
      eventTimer.restart(self until-latency)
    else
      notifyExpiration(i)
    end
  end
let notify(i :import.T) :Ok = ...
let importedPeterLifeExtender =
  eventTimer.new(import.expiration(importedPeter)-latency
    extendLife(weakRef(importedPeter) duration notify))
```

```
cleanUp.register(importedPeter
  fun(:import.T) eventTimer.free(importedPeterLifeExtender)
```

Dieser Ansatz schränkt die Autonomie des importierenden Ablaufsystems ein. Das exportierende Ablaufsystem kann das gebundene Objekt freigeben, obwohl es noch von dem importierenden Objektsystem erreichbar ist, wenn die Verlängerung nicht rechtzeitig erfolgt oder verweigert wird. Die Integrität des importierenden Objektsystems ist somit nicht sichergestellt.

Dieser Ansatz ist abhängig von einer zuverlässigen Verbindung zwischen den beiden Objektsystemen, damit die Verlängerung rechtzeitig stattfinden kann.

Eingeschränkte Autonomie des exportierenden und des importierenden Ablaufsystems

Diese Strategie bildet einen Kompromiß zwischen den Autonomie- und den Integritätsanforderungen der exportierenden und importierenden Ablaufsysteme (vgl. auch [Birrell et al. 93]). Das exportierende Ablaufsystem führt für jede externe Bindung

- eine Liste der Ablaufsysteme, welche die externe Bindung importiert haben und
- eine Lebensdauer für jede externe Bindung.

Das exportierende Ablaufsystem erhält die Bindung mindestens so lange, wie die Lebensdauer angibt. Die Lebensdauer kann von den importierenden Ablaufsystemen verlängert werden. Wenn das gebundene Objekt freigegeben werden könnte, weil es nicht von dem Objektsystem lokal gebunden wird, und die Lebensdauer abgelaufen ist, bemüht das exportierende Ablaufsystem sich, bei den importierenden Ablaufsystemen nachzufragen, ob die Bindung noch benötigt wird und daher die Lebensdauer verlängert werden soll. Wenn kein importierendes Ablaufsystem eine Verlängerung wünscht, kann das Objekt freigegeben werden. Sollte eines der importierenden Ablaufsysteme nicht innerhalb einer bestimmten Zeitspanne oder einer gegebenen Anzahl von Wiederholungen erreicht werden, wird das Objekt freigegeben. Wenn Fehler ausgeschlossen werden sollen, muß in diesem Fall die Zeitspanne "unendlich" gelten, weil nicht mit Sicherheit ausgeschlossen werden kann, daß das Ablaufsystem doch irgendwann wieder erreichbar ist und noch eine externe Bindung hält. Dies gilt insbesondere für persistente Objektsysteme, die zu einem späteren Zeitpunkt reaktiviert werden.

Das importierende Ablaufsystem erhält die externe Bindung mit einer garantierten Lebensdauer. Wenn das Ablaufsystem die Bindung nicht mehr benötigt, informiert es das exportierende Ablaufsystem und wird somit aus der Liste der importierenden Ablaufsysteme gestrichen. Wenn die Lebensdauer einer externen Bindung sich dem Ende nähert, sollte das Ablaufsystem eine Verlängerung bei dem exportierenden Ablaufsystem beantragen.

Damit das exportierende Ablaufsystem Kenntnis über die importierenden Ablaufsysteme erhält, darf die externe Bindung von einem importierenden Ablaufsystem nicht einfach an andere Ablaufsysteme weitergegeben werden. Stattdessen muß die Weitergabe dem exportierenden Ablaufsystem gemeldet werden. Das exportierende Ablaufsystem kann aber die Weitergabe einer externen Bindung ablehnen, beispielsweise aus Gründen der Sicherheit, d.h. die Kopie ist dann ungültig, es sind evtl. keine Operationen darauf möglich und ihre Lebensdauer ist nicht garantiert.

4.3.4 Externe Bindungen an Objekte des Ablaufsystems

Basisobjekte sind Objekte des Ablaufsystems (vgl. Abschnitt 2.5.2). Aus Sicht des Objektsystems kann es sich dabei um interne oder externe Objekte handeln. Da das Ablaufsystem auch ein Objektsystem ist, besitzen die Basisobjekte auch ein Ablaufsystem, das ihnen Semantik verleiht (vgl. Abbildung 4.13). Dieses Ablaufsystem legt fest, welche Basisobjekte gebunden werden können, welche Operationen möglich sind usw. Im Unterschied zu anderen externen Objekten sind Basisobjekte trivialerweise während des Ablaufs des Objektsystems vorhanden, da es sonst (ohne das Ablaufsystem) nicht ablaufen könnte.

Die Schnittstelle zwischen Objektsystem und Ablaufsystem zur Behandlung externer Objekte kann über den Interpreter in der von ihm implementierten Repräsentation oder über Basisobjekte des Ablaufsystems implementiert werden. Da Basisobjekte aber auch nur (spezielle) externe Objekte sind, müssen einige Teile der Schnittstelle zumindest für Basisobjekte in der vom Interpreter interpretierten Repräsentation implementiert sein. Umgekehrt können Basisobjekte wie externe Objekte angebunden werden, wenn ein generischer Mechanismus dafür vom Interpreter zur Verfügung gestellt wird. Das Ablaufsystem tritt dabei gleichzeitig als exportierendes Objektsystem auf.

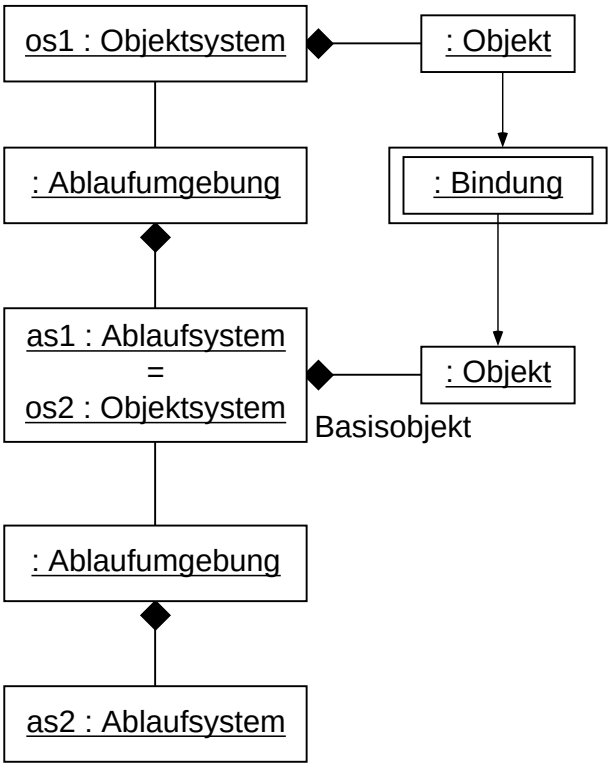

Abbildung 4.13: Kooperation zwischen Ablaufsystem und Objektsystem über Basisobjekte

Die Kooperation zwischen Objektsystemen läuft über die Ablaufsysteme der Objektsysteme ab. Wenn ein Objektsystem herunterfährt, d.h. alle Prozesse beendet sind, kann es nicht mehr aktiv kooperieren. Aber es kann trotzdem noch passiv an einer Kooperation teilnehmen, beispielsweise wenn ein Objekt des Objektsystems durch eine externe Bindung von einem anderen Objektsystem gebunden ist.

Andererseits kann die Kooperation zwischen zwei Objektsystemen nur so lange stattfinden, wie die Ablaufsysteme beider Objektsysteme hochgefahren sind. Sobald ein Ablaufsystem herunterfährt, ist die Kooperation beendet (außer bei persistenten Objektsystemen, vgl. Abschnitt 3.2). Insbesondere sind die externen Bindungen an Objekte des Objektsystems des heruntergefahrenen Ablaufsystems ungültig. Auf den externen Objekten können keine Operationen mehr ausgeführt werden. Objekte können nicht mehr zwischen den Objektsystemen migrieren.

Die externen Objekte und ihr Ablaufsystem sind prinzipiell unabhängig von dem Objektsystem, das diese Objekte bindet, und seinem Ablaufsystem (vgl. Abbildung 4.14). Wenn beispielsweise Operationen auf einem externen Objekt einer parallel zu einer Java-Maschine existierenden LISP-Maschine ausgeführt werden sollen, muß diese hochgefahren sein.

Allerdings kann das Ablaufsystem der externen Objekte ebenfalls das Ablaufsystem für das Ablaufsystem des Objektsystems bilden, wie es beispielsweise bei einer Sun-Maschine als Grund-

lage für eine Java-Maschine ist. Damit ist das Ablaufsystem der externen Objekte, hier die Sun-Maschine, mindestens solange hochgefahren wie das Ablaufsystem des Objektsystems, hier die Java-Maschine. In diesem Fall vereinfacht sich die Kooperation, da das Ablaufsystem der externen Objekte mindestens so lange existiert wie das Ablaufsystem des Objektsystems und damit wie das Objektsystem selber. Die externen Objekte sind somit prinzipiell während der gesamten Laufzeit des Objektsystems erreichbar, Operationen auf diesen Objekten sind möglich und die Objekte können zwischen den Ablaufsystemen migrieren. Diese Situation tritt beispielsweise bei C-Bibliotheken auf, die an virtuelle Maschinen gebunden werden [Lindholm, Yellin 96].

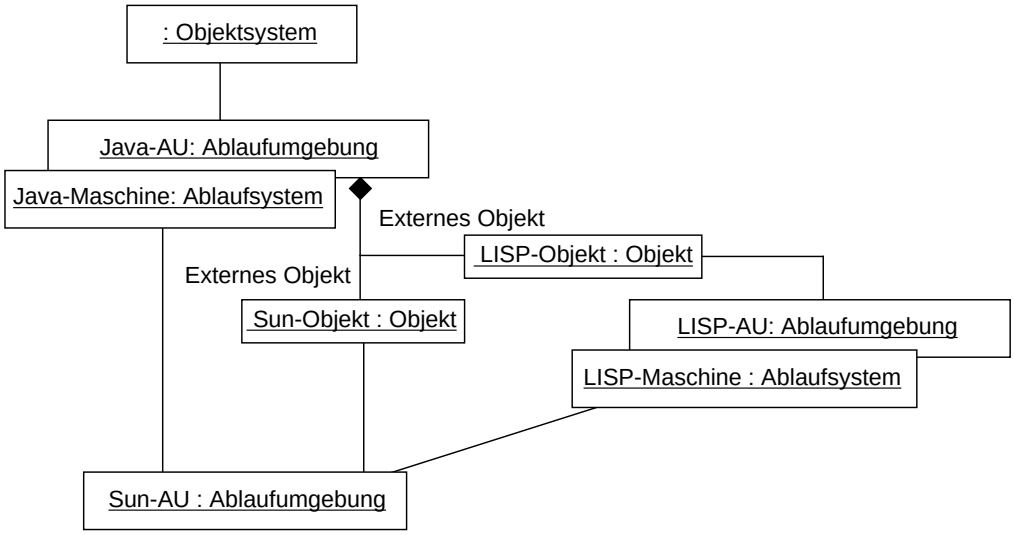

Abbildung 4.14: Geschachtelte Ablaufumgebungen

4.3.5 Persistenz externer Objekte

Objektsysteme sind autonom und an beliebig vielen (inhaltlich disjunkten) Kooperationsbeziehungen beteiligt. Es ist nicht sinnvoll, inhaltlich nicht zusammenhängende Kooperationsbeziehungen persistent abzulegen. Daher kann nicht davon ausgegangen werden, daß ein Ablaufzustand aller an einer Kooperation beteiligten Objektsysteme persistent abgelegt werden kann, wenn eines der beteiligten Objektsysteme dies wünscht.

Die Persistenzsicherung eines Objektsystems ist aus den zwei Rollen, als importierendes und exportierendes Objektsystem, zu betrachten.

Als importierendes Objektsystem werden die Bindungen an externe Objekte mit dem Objektsystem persistent abgelegt. Allerdings ist die Lebenszeit der Bindungen zu beachten. Wenn nicht bekannt ist, wann das Objektsystem wieder aktiviert wird, muß sie unendlich sein, damit die Reaktivierbarkeit sichergestellt ist. In der Kooperationsstrategie sollte daher diese Gültigkeitsdauer vorgesehen werden, die dann explizit bei der Persistenzsicherung eines Objektsystems angefordert wird. Insbesondere externe Bindungen an Objekte des Ablaufsystems besitzen eine geringere Gültigkeitsdauer. In diesem Fall ist die Liste der Seiteneffekte, die beim Hochfahren des Objektsystems zu wiederholen sind (vgl. Abschnitt 3.2), um die Wiederherstellung der externen Bindungen zu ergänzen. Auch hier ist unter Umständem eine Reihenfolge einzuhalten. Wenn das Objektsystem nach der Persistenzsicherung weiterläuft, dürfen keine externen Objekte freigegeben werden, die noch durch die persistente Version des Objektsystems gebunden sind. D.h. die persistente Version ist wie eine Kopie des Objektsystems zu betrachten und es sind daher die gleichen Aktionen auszuführen, die auch bei einer Kopie der externen Bindung erfolgt wären.

Als exportierendes Objektsystem ist damit zu rechnen, daß ein Objektsystem, das eine externe

Bindung hält, zeitweise nicht zu erreichen ist, weil es nur in einer persistenten (inaktiven) Version vorliegt. Trotzdem kann das Objektsystem jederzeit wieder aktiviert werden. Daher sollten keine vorzeitigen Freigaben extern gebundener Objekte erfolgen. Außerdem sollte das exportierende Objektsystem es erlauben, Kopien der externen Bindungen zu erzeugen und mit einer unendlichen Lebensdauer zu versehen.

4.3.6 Evolution

Die Evolution von kooperierenden Objektsystemen baut auf der in Abschnitt 3.3 beschriebenen Evolution von Objektsystemen auf. Ausgangspunkt bildet die Annahme, daß die Systeme langlebig sind und über persistente (verteilte) Zustände mit Konsistenzannahmen verfügen. Um sich geänderten oder neuen Anforderungen anzupassen, werden die Systeme einem Evolutionsprozeß unterworfen, der ihre Ablauffähigkeit nicht gefährden soll. Daher sind besondere Maßnahmen zu treffen, um die Konsistenz und damit Ablauffähigkeit der kooperierenden Objektsysteme zu gewährleisten. Im Gegensatz zur lokalen Evolution liegen dabei die von einer Evolution betroffenen Objekte nicht ausschließlich im Einflußbereich des lokalen Ablaufsystems. Daher werden die Kooperationsmechanismen um kooperative Evolutionsmechanismen ergänzt.

An einer Kooperation sind (mindestens) zwei Objektsysteme mit ihren Ablaufsystemen beteiligt, wobei jedes der beteiligten Systeme sich ändern kann.

- Die Evolution eines Objektsystems wirkt sich auf den Kooperationspartner aus. Wenn die Änderungen nicht durch Adapter abgefangen werden, muß der Kooperationspartner sich an die Änderungen anpassen (vgl. Abschnitt 3.3).

- Die Evolution eines Ablaufsystems hat Auswirkungen auf das Objektsystem, das dieses Ablaufsystem für den Ablauf benötigt, und das Ablaufsystem des Kooperationspartners, da beide Ablaufsysteme zusammen die Kooperation ermöglichen.

  Dies ist ein Spezialfall der allgemeinen Evolution. Er tritt beispielsweise auf, wenn im Kontext von CORBA sich eine ORB-Implementierung oder die CORBA Spezifikation ändert [Soley 92; OMG 98].

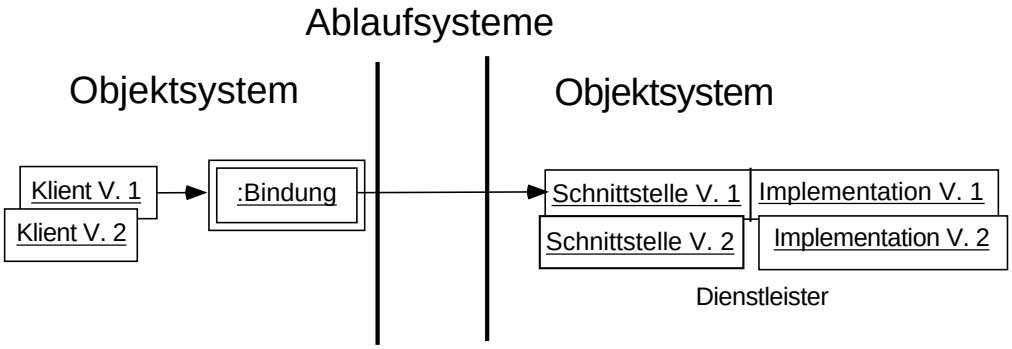

Abbildung 4.15: Evolution in einer Kooperation

Das Ausgangsszenario bilden zwei Objekte, wobei ein Objekt die Rolle des Klienten und das andere die Rolle des Dienstleisters (vgl. [OMG 98]) übernimmt (vgl. Abbildung 4.15); symmetrische Kooperationsbeziehungen werden auf zwei dieser Kienten/Dienstleister-Paare abgebildet. Der Klient wird gegen eine Schnittstelle des Dienstleisters entwickelt. Der Dienstleister implementiert diese Schnittstelle. Während des Ablaufs findet eine Bindung des Klienten an eine Implementation der Schnittstelle statt. In diesem Szenario können sich der Klient, die Schnittstelle und die Implementation der Schnittstelle ändern. Idealerweise sollen Änderungen an dem Klienten oder Dienstleister

keine Auswirkungen auf den Kooperationspartner zeigen [Soley 92], was nur bei eingeschränkten Änderungen möglich ist.

Eine Änderung des Klienten zeigt keine Auswirkungen auf den Dienstleister, solange er keine Erwartungen an den Klienten stellt, beispielsweise bzgl. der Benachrichtigung, daß die Bindung an den Dienstleister nicht mehr benötigt wird. Wenn der Dienstleister während der Kooperation solche Dienste des Klienten benötigt, wechselt er damit die Rolle und wird zum Klienten, so daß die folgende Diskussion der Auswirkungen von Änderungen des Dienstleisters auf ihn anzuwenden ist.

Als Änderungszeitpunkte sind aus Sicht des Klienten zu unterscheiden:

1. Entwicklung
   (a) vor der Entwicklung des Klienten
   (b) während der Entwicklung des Klienten
   (c) nach der Entwicklung des Klienten
2. Ablauf
   (a) vor dem Ablauf des Klienten (nach seiner Entwicklung)
   (b) während des Ablaufs des Klienten, vor der Benutzung[1] des Dienstleisters
   (c) während der Benutzung des Dienstleisters durch den Klienten
   (d) nach einer Benutzung des Dienstleisters durch den Klienten, wobei eine weitere Benutzung (in einem anderen Kontext) möglich ist
   (e) nach einem Ablauf des Klienten (vor dem nächsten Ablauf)

Im Unterschied zur lokalen Evolution (vgl. Abschnitt 3.3) laufen die Entwicklung, die Evolution und der Ablauf von Klient und Dienstleister (prinzipiell) unabhängig voneinander ab, so daß sich ein Koordinationsbedarf für die Entwicklungs- und Ablaufumgebungen ergibt. Beispielsweise ist zu beachten, daß ein Dienstleister nach der Entwicklung, aber vor dem Ablauf des Klienten, oder zwischen zwei Abläufen des Klienten geändert werden kann.

Mögliche Arten von Änderungen des Dienstleisters sind aus Sicht des Klienten (vgl. [Soley 92]):

1. Die Schnittstelle des Dienstleisters ändert sich.
2. Die Implementierung des Dienstleisters ändert sich.
3. Die Lokation des Dienstleisters ändert sich.

Die Änderungen besitzen einen unterschiedlichen Umfang, der sich beispielsweise in Subtypbeziehungen bei Schnittstellen ausdrückt [Matthes 93]. Im Unterschied zur lokalen Evolution (vgl. Abschnitt 3.3) kommt die Lokationsänderung hinzu.

Je nach Zeitpunkt, Art und Umfang der Änderung sind unterschiedliche Maßnahmen zur Sicherung der Ablauffähigkeit zu treffen, die im wesentlichen denen der lokalen Evolution entsprechen (vgl. Abschnitt 3.3). Insbesondere sind auch übergreifende Kontexte zu beachten, also die Änderung mehrerer voneinander abhängiger Dienstleister oder Klienten. Wenn beispielsweise zwei Klienten auf einen Dienstleister zugreifen und auch untereinander kooperieren (vgl. Abbildung 4.16), ist eine Anpassung beider Klienten bei der Änderung des Dienstleisters notwendig, um Ablaufsicherheit zu gewährleisten.

Die Entwicklungsumgebungen des Klienten und des Dienstleisters sind miteinander zu koordinieren, so daß insbesondere Änderungen der Schnittstelle des Dienstleisters in die Entwicklung des Klienten einbezogen werden. Mit der CORBA-IDL (*Interface Definition Language*) und dem CORBA `Interface Repository` liegen Basismechanismen vor, die in die Entwicklungsumgebungen zu integrieren sind.

[1] "Benutzung" ist hier eine in sich abgeschlossene Folge voneinander abhängiger Operationen, die der Klient beim Dienstleister anfordert.

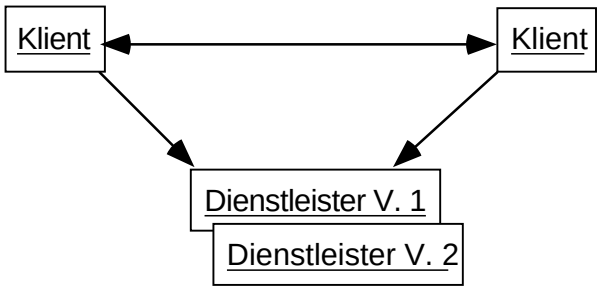

Abbildung 4.16: Zwei kooperierende Klienten und ein geänderter Dienstleister

Während des Ablaufs bietet sich die Nutzung standardisierter Dienste wie der CORBAservices [OMG 97a] an, um die Evolution über unterschiedliche Ablaufumgebungen hinweg zu unterstützen:[2]

- Sowohl Schnittstellen als auch Implementationen werden versioniert. Beim Zugriff des Klienten auf den Dienstleister werden die Versionen abgeglichen und geeignete Maßnahmen, beispielsweise zur Adaption an ältere Versionen, eingeleitet. Das CORBA `InterfaceRepository` bietet initiale Semantik für die Schnittstellenverwaltung, insbesondere Versionsbezeichnungen. Das CORBA `ImplementationRepository` und der CORBA `Portable Object Adapter` bieten einen Ansatzpunkt zur Unterstützung der Evolution auf Implementationsseite.

- Um die Adaption durch den Dienstleister bzw. dessen Entwicklungsumgebung zu unterstützen, bietet die Ablaufumgebung reflektive Eingriffsmechanismen in den entfernten Zugriff an, beispielsweise durch den CORBA `Interceptor`.

- Die Bindungen zwischen den kooperierenden Objekten werden erfaßt, so daß Anpassungen während des Ablaufs möglich sind. Dazu können beispielsweise die CORBAservices `Lifecycle` und `Relationship` eingesetzt werden.

- Um Klienten von einer Änderung des Dienstleisters zu unterrichten und ihnen somit die Möglichkeit der Anpassung zu geben, registrieren die Klienten sich beim Dienstleister und werden informiert, sobald sich der Dienstleister ändert. Dazu kann der CORBAservice `Event` genutzt werden.

- Die für den Klienten transparente Änderung der Lokation von Objekten wird durch Namensdienste und Kooperation auf der Ebene der Ablaufsysteme [Mathiske 96] bewerkstelligt.

Dies setzt voraus, daß sowohl die Entwicklungs- als auch die Ablaufumgebungen kooperierender Objektsysteme auf die Evolution ausgerichtet sind, beispielsweise durch die Unterstützung reflektiver Techniken (vgl. Kapitel 3). Die kooperierenden Objektsysteme sind mit dem Ziel zu entwickeln, Evolution auch in Kooperationen zu unterstützen. Die Entwicklung geht damit über den funktionalen Aspekt der Unterstützung einer Anwendung hinaus, wie in Kapitel 5 vertieft wird.

4.4 Zusammenfassung

Nur durch die Kooperation von Objektsystemen in heterogenen Umgebungen lassen sich komplexe Anwendungsaufgaben lösen, welche die Erbringung unterschiedlicher Leistungen erfordern, die von

[2] In [Soley 92] wird eine Unterstützung für Evolution durch CORBA gefordert, in [OMG 95] eine Anforderungsliste für einen `Change Management Service` vorgelegt und in [OMG 97a] dieser Service angekündigt; eine Spezifikation liegt bisher nicht vor.

einem Objektsystem allein nicht erbracht werden können. Anwendungsabhängige Kooperationsstrategien werden durch grundlegende Kooperationsmechanismen unterstützt:

- Ein Objektsystem stellt eine Kommunikationsverbindung zu einem Kooperationspartner her.
- Ein Objektsystem bindet Objekte eines Kooperationspartners.
- Ein Objektsystem stößt die Ausführung von Operationen auf Objekten eines Kooperationspartners an.
- Ein Objektsystem überträgt Objekte an einen Kooperationspartner.

Besondere Herausforderungen bedeuten die Autonomie und Heterogenität der Kooperationspartner.

Die Heterogenität der Objektsysteme bzw. Ablaufumgebungen wird durch Adaption mit Hilfe reflektiver Mechanismen überbrückt. Objekte werden portiert, Repräsentationen umgewandelt, Bindungen angepaßt oder erneuert. Ein besonderer Beitrag liegt in der Erweiterung der Flexibilität bei der Wiederherstellung von Bindungen durch die Übertragung von Funktionen, die das Binden im Zielsystem unter den vom Zielsystem vorgegebenen Randbedingungen vornehmen. Somit kann eine anwendungsabhängige Kooperationsstrategie ausgewählt werden, welche die Autonomie beider Kooperationspartner wahrt.

Die Autonomie von kooperierenden Objektsystemen wird durch externe Bindungen eingeschränkt, insbesondere bei persistenten Systemen. Kooperationsstrategien zur Speicherverwaltung ermöglichen einen anwendungsabhängigen Ausgleich zwischen Integritätssicherung und Ressourcenschonung. Dabei bilden reflektive Mechanismen, die auf der Kooperation zwischen Objekt- und Ablaufsystem beruhen, die Grundlage für die Realisierung und für eine flexible Erweiterbarkeit um neue Kooperationsmechanismen.

Kooperationsmechanismen werden in der Entwicklung (vgl. Kapitel 5) durch Entwicklungsmuster unterstützt. Um die Ablauffähigkeit von kooperierenden Objektsystemen zu erhalten, werden während der Entwicklung Validierungsmechanismen eingeführt, die eine Überprüfung der Ablaufsicherheit dynamisch während des Ablaufs ermöglichen.

Kapitel 5

Entwicklung kooperierender Objektsysteme

Die Entwicklung kooperierender Objektsysteme bettet sich ein in den Softwareentwicklungsprozeß aus Analyse, Entwurf und Implementierung [Rumbaugh et al. 91; Sommerville 96; Ruble 97]. Hier wird der Fokus auf Entwurf und Implementierung kooperierender Objektsysteme unter besonderer Beachtung der Integritätssicherung gesetzt.

Ergebnis der Entwicklung ist ein ablauffähiges Objektsystem, also ein Objektsystem in einer Repräsentation, die von einem Interpreter ausgeführt werden kann. Für das entwickelte Objektsystem gelten folgende externe Qualitätskriterien [Dunn 93]:

Effizienz. Das Objektsystem realisiert eine ressourcenschonende Lösung des Anwendungsfalls und nutzt in der Ablaufumgebung vorhandene Ressourcen, die zur Lösung des Anwendungsfalls beitragen, effizient aus.

Fehlerfreiheit. Das Objektsystem vermeidet fehlerhafte Abläufe, die zu Fehlinterpretationen, Informationsverlusten oder Schäden führen.

Effizienz und Fehlervermeidung sind nicht unabhängig voneinander. Insbesondere führen Fehler zu verminderter Effizienz, weshalb Fehlervermeidung ein zentrales Anliegen der Entwicklung ist.

Beispielsweise ist die wiederholte manuelle Erfassung von Daten nicht effizient, weil die Ressource "Zeit" übermäßig verbraucht wird, und außerdem auch fehlerträchtig, weil die Daten unterschiedlich erfaßt werden könnten. Daher sollte ein Objektsystem während des Ablaufs erzeugte Objekte, die noch benötigt werden, persistent ablegen und mit anderen Objektsystemen kooperieren, die diese Objekte ebenfalls benötigen. Andererseits treten durch Persistenz und Kooperation neue Fehlerquellen auf, insbesondere im Zusammenhang mit Evolution und Heterogenität. An die Entwicklungsumgebung für kooperierende, persistente, sich weiterentwickelnde Objektsysteme in heterogenen Umgebungen werden erweiterte Anforderungen im Vergleich mit Entwicklungsumgebungen für eigenständige, transiente, statische Objektsysteme gestellt. Insbesondere ist die Vermeidung von Fehlern zu betrachten. In diesem Kapitel werden daher Entwicklungsmechanismen zur Modellierung, Validierung und Generierung vorgestellt, die zur Fehlervermeidung beitragen.

In Begriffen des Objektsystemmodells aus Kapitel 2 wird der Entwicklungsprozeß folgendermaßen beschrieben: Die Entwicklung wird durch ein Entwicklungssystem unterstützt. Das Entwicklungssystem ist auf ein Ablaufsystem ausgerichtet, d.h. es erzeugt Objektsysteme für dieses Ablaufsystem. Das Ablaufsystem bietet eine ablaufgerechte Repräsentation und Basisobjekte, welche zusammen auf eine effiziente Ausführung unter Ausnutzung von Persistenz und Kooperation ausgerichtet sind (vgl. Kapitel 3 und 4). Ziel des Entwicklungsprozesses ist es, diese Effizienz zu nutzen und zugleich weitgehende Fehlerfreiheit zu garantieren. Entscheidender Beitrag dazu ist die Nutzung einer anwendungsgerechten Repräsentation zur Modellierung des Objektsystems, aus der durch einen Übersetzungsvorgang die ablaufgerechte Repräsentation erzeugt wird (vgl. Abschnitt 5.1). Validierer prüfen statisch (zur Entwicklungszeit) oder dynamisch (zur Ablaufzeit)

die Einhaltung von Integritätsbedingungen durch das Objektsystem und verhindern somit (gewisse) fehlerhafte Abläufe (vgl. Abschnitt 5.2). Generatoren erzeugen statisch oder dynamisch Teile des Objektsystems und garantieren dabei die Einhaltung von Integritätsbedingungen (vgl. Abschnitt 5.3).

Das Ausgangsszenario der Entwicklung stellt sich folgendermaßen dar (vgl. Abbildung 5.1): Ein Anwendungsfall soll durch ein Objektsystem unterstützt werden. Das Objektsystem wird in einer anwendungsgerechten Repräsentation modelliert, also beispielsweise in einer textuellen (Quelltexte einer höheren Programmiersprache) oder grafischen (Diagramme einer Modellnotation) Repräsentation. Aus dieser Beschreibung erzeugt ein Übersetzer das ablaufgerechte Objektsystem, dessen Ablauf die Anwendung realisiert. Sowohl während der Übersetzung als auch während des Ablaufs gelangen Validierungs- und Generierungsmechanismen zum Einsatz. Während der Übersetzung werden diese (Validierungs- und Generierungs-)Mechanismen als "statisch" bezeichnet, weil sie nur von den zur Entwicklungszeit zur Verfügung stehenden Informationen gesteuert werden und einen festen Teil des Objektsystems während der Entwicklung prüfen bzw. erzeugen, während die während des Ablaufs "dynamisch" auftretenden Mechanismen den Ablaufzustand des Objektsystems einbeziehen und somit zustandsabhängig unterschiedliche Teile des Objektsystems prüfen bzw. erzeugen.

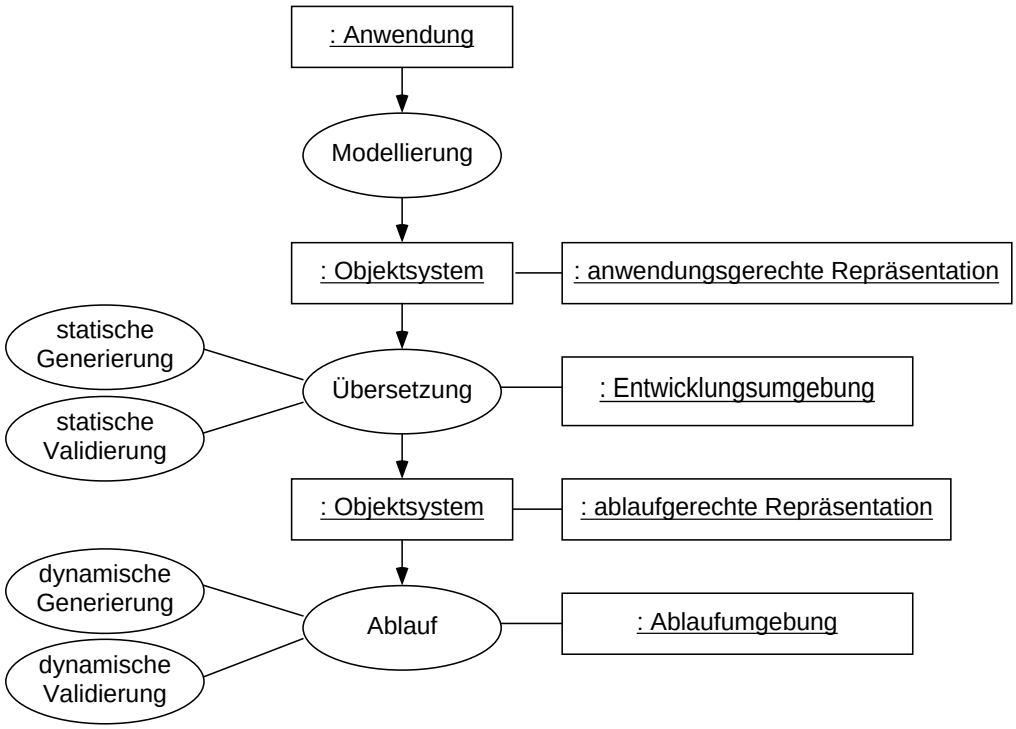

Abbildung 5.1: Entwicklungsszenario

5.1 Modellierung

Die Modellierung eines Objektsystems unterliegt Entwicklungsanforderungen, denen mit Hilfe von Entwicklungsmechanismen begegnet wird (vgl. Abschnitt 5.1.1). Das Objektsystem wird in einer anwendungsgerechten Repräsentation modelliert, aus der eine ablaufgerechte Repräsentation erzeugt wird (vgl. Abschnitt 5.1.2). Die automatische Erzeugung der ablaufgerechten aus der anwendungsgerechten Repräsentation übernimmt ein Übersetzer (vgl. Abschnitt 5.1.3). Der Entwurf

einer anwendungsgerechten Repräsentation unterliegt Kriterien, die vom Entwicklungsprozeß bestimmt werden (vgl. Abschnitt 5.1.4).

5.1.1 Entwicklungsanforderungen und -mechanismen

Die Entwicklungsumgebung unterstützt den Entwickler dabei, ein die Anwendung realisierendes Objektsystem für eine bestimmte Ablaufumgebung zu erzeugen. Die Entwicklung persistenter, kooperierender, sich weiterentwickelnder Objektsysteme in heterogenen Umgebungen stellt besondere Anforderungen an die Entwicklungsumgebung.

Persistenz: Die Entwicklungsumgebung unterstützt den Entwickler dabei, den Zustand eines Objektsystems persistent abzulegen und wieder zu rekonstruieren.

Persistenzsicherung verlangt (vgl. Abschnitt 3.2), daß jedes Objekt in einer persistenten Repräsentation vorliegt bzw. in diese Repräsentation umgewandelt wird. Die Entwicklungsumgebung erzeugt die persistenten Repräsentationen oder bindet Funktionen ein, die auf Anforderung eine persistente Repräsentation erzeugen. Mit Hilfe reflektiver Mechanismen des Ablaufsystems wird die Reaktivierbarkeit eines persistenten Objektsystems unterstützt, beispielsweise die Wiederholung von Seiteneffekten. Bei kooperierenden Objektsystemen gleicht die Entwicklungsumgebung die unterschiedlichen Persistenzmodelle aneinander an.

Kooperation: Die Entwicklungsumgebung unterstützt den Entwickler dabei, ein Objektsystem mit anderen Objektsystemen kooperieren zu lassen.

Die Entwicklungsumgebung legt die Kooperationsstrategie (vgl. Kapitel 4) nach den Erfordernissen der Anwendung fest und implementiert sie mit Hilfe der Kooperationsmechanismen der Ablaufumgebung. Sie ermöglicht die Verbindungsaufnahme, den Austausch von Objekten, die Bindung an externe Objekte, den Export von Objekten, die Unterstützung von Operationen auf importierten und exportierten Objekten usw. Sie vermeidet (soweit möglich) und behandelt Fehler, die während einer Kooperationsbeziehung aufgrund der Autonomie der Partner auftauchen. Sie sorgt dafür, daß reflektiv Basisobjekte erzeugt oder registriert werden, die beispielsweise die Verlängerung der Gültigkeit von externen Bindungen vornehmen oder das Objektsystem über nicht mehr gültige Bindungen informieren.

Adaption: Die Entwicklungsumgebung unterstützt den Entwickler dabei, ein Objektsystem portabel zu halten, zwischen Ablaufumgebungen zu portieren und somit an Ablaufumgebungen zu adaptieren.

In heterogenen Umgebungen (vgl. Kapitel 4) werden unterschiedliche Ablaufrepräsentationen verwendet und stehen unterschiedliche externe Objekte zur Verfügung. Die Entwicklungsumgebung sichert die Portabilität des Objektsystems, indem sie eine portable Repräsentation oder Übersetzer zwischen den Repräsentationen vorhält sowie die externen Objekte so verwaltet, daß sie ausgetauscht werden können. Bei kooperierenden Objektsystemen tritt dieser Fall ein, wenn Objekte zwischen den Objektsystemen ausgetauscht werden und sich an die neue Ablaufumgebung adaptieren müssen.

Evolution: Die Entwicklungsumgebung unterstützt den Entwickler dabei, ein Objektsystem weiter zu entwickeln.

Die Entwicklungsumgebung stellt sicher, daß die Evolution (vgl. Abschnitt 3.3) die Ablauffähigkeit des Objektsystems nicht einschränkt, und daß alle betroffenen Teile des Objektsystems geändert werden. Bei kooperierenden Objektsystemen beachtet sie insbesondere auch externe Bindungen und setzt Versionierung ein, um auch Kooperationspartner zu unterstützen, die sich auf einem anderen Entwicklungsstand befinden.

Diese Anforderungen werden durch Entwicklungsmechanismen des Entwicklungssystem erfüllt, die durch Funktionsobjekte implementiert werden:

Übersetzer bilden zwei Repräsentationen aufeinander ab.

- Abbildung einer anwendungsgerechten auf eine ablaufgerechte Repräsentation, wie sie von Compilern realisiert wird [Aho et al. 88].
- Abbildung von ablaufgerechten Repräsentationen unterschiedlicher Ablaufsysteme aufeinander oder Abbildung einer portablen Repräsentation in eine auf einem bestimmten Ablaufsystem ablauffähige Repräsentation [Gosling 95; Diehl 98]) bzw. umgekehrt [Steensgard, Jul 95; Mathiske 96].
- Abbildung einer transienten auf eine persistente Repräsentation bzw. umgekehrt [Matthes, Schmidt 94; Matthes et al. 96].

Validierer prüfen die Eigenschaften von in einer Repräsentation vorliegenden Objekten (vgl. Abschnitt 5.2).

- Prüfung von Typeinschränkungen auf Typobjekten durch Typprüfer (*type checker*) [Cardelli 97b].
- Prüfung eines in einer ablauffähigen Repräsentation vorliegenden Funktionsobjektes auf Nutzung sicherheitsbedenklicher Anweisungen (*bytecode verifier* [Lindholm, Yellin 96; Stata, Abadi 98]).
- Prüfung von explizit definierten Integritätsbedingungen, beispielsweise im Vertragsmodell [Meyer 88].

Generatoren erzeugen in einem Entwicklungsmuster angeordnete Objekte (vgl. Abschnitt 5.3).

- Generierung von Implementationsmustern [Gamma et al. 95].
- Generierung von *stubs* [Birrell et al. 95] oder *proxies* [Shapiro 86].
- Generierung von Protokollimplementierungen für Kooperationsstrategien [Mathiske 96].

5.1.2 Anwendungsgerechte und ablaufgerechte Repräsentation

Ziel der Entwicklung ist es, ein ablauffähiges Objektsystem zu erzeugen. Dazu muß das Objektsystem in einer Repräsentation vorliegen, die von einem Interpreter ausgeführt werden kann, der ablaufgerechten Repräsentation; diese Repräsentation ist also auf die Ausführung durch einen Interpreter, "die Maschine", ausgerichtet. Die Modellierung von Anwendungen findet interpreterunabhängig in einer anwendungsnahen Repräsentation statt, die anwendungsnahe Abstraktionen bietet; diese Repräsentation ist auf die Modellierung einer Anwendung durch einen Entwickler, "den Menschen" ausgerichtet, beispielsweise eine textuelle Modellierungssprache [Sebesta 89] oder eine grafische Modellierungssprache wie UML [Fowler, Scott 97].

Zwischen der anwendungsgerechten und der ablaufgerechten Repräsentation findet eine Abbildung statt, in der ein in einer anwendungsgerechten Repräsentation modelliertes Objektsystem in ein ablauffähiges Objektsystem in einer ablaufgerechten Repräsentation übersetzt wird. Es finden also zwei gegenläufige Vorgänge statt:

- Von der ablaufgerechten Repräsentation wird abstrahiert, bis eine anwendungsgerechte Repräsentation erreicht ist.
- Die anwendungsgerechte Repräsentation wird in eine ablaufgerechte Repräsentation abgebildet.

Das Objektsystem wird von seinen Entwicklern in einer anwendungsgerechten Repräsentation modelliert. Aus dieser Repräsentation erzeugt ein Übersetzer (vgl. Abbildung 5.2) eine ablaufgerechte Repräsentation, die von einem Interpreter eines Ablaufsystems ausgeführt wird [Aho et al. 88]. Die beiden Repräsentationen können sich entsprechen, d.h. der Entwickler arbeitet mit der ablaufgerechten Repräsentation bzw. der Interpreter führt die anwendungsgerechte Repräsentation aus.

Anwendungs- und ablaufgerechte Repräsentationen erfüllen unterschiedliche Anforderungen:

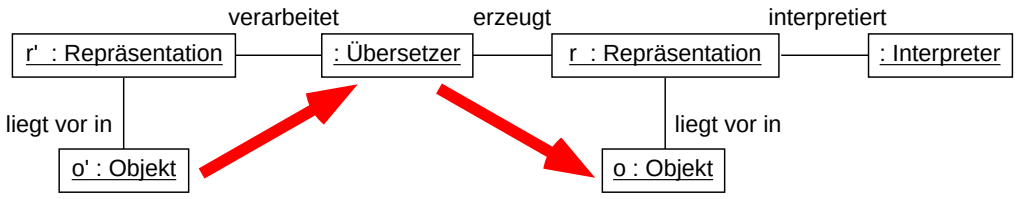

Abbildung 5.2: Übersetzer

Wesentliche Aspekte einer *anwendungsgerechten Repräsentation* sind Modellierungsmächtigkeit, Versteh- und Prüfbarkeit [Sebesta 89; Pratt, Zelkowitz 96]. Aus diesem Grunde weist diese Repräsentation Redundanz auf. Es existiert keine eineindeutige Abbildung zwischen anwendungsgerechter und ablaufgerechter Repräsentation, sondern ein Modellierungskonstrukt wird in mehrere Konstrukte der Ablaufrepräsentation abgebildet (vgl. Generierung in Abschnitt 5.3) und ein Konstrukt der Ablaufrepräsentation von mehreren Modellierungskonstrukten genutzt, so daß überlappende bzw. alternative Modellierungskonstrukte existieren. Über Redundanz in der Modellierung werden Widersprüche festgestellt, beispielsweise durch die Angabe von Typen (vgl. Abschnitt 5.2). Befehle und Bezeichner sind symbolisch durch Zeichenketten repräsentiert, statt eine möglichst kurze Schreibweise zu wählen, z.B. als Bitmuster oder de Bruijn Indices [de Bruijn 72]. Außerdem unterstützt die anwendungsgerechte Repräsentation Entwicklungsparadigmen, beispielsweise funktional, imperativ, deklarativ oder objektorientiert [Pratt, Zelkowitz 96], durch Abstraktion von Eigenschaften des Interpreters und des Ablaufsystems sowie Generierung von Ausführungsmustern (vgl. Abschnitt 5.3).

Die *ablaufgerechte Repräsentation* ist auf die effiziente Ausführung durch den Interpreter abgestimmt [Gosling 95]. Sie ist kurz, redundanzfrei und spiegelt die Eigenschaften des Interpreters direkt wider. Die Befehle und Bezeichner werden daher numerisch repräsentiert. Die verfügbaren Befehle sind durch die effiziente Ausführbarkeit bestimmt. Dabei kann eine Präferenz für bestimmte Anwendungen bzw. Programmierparadigmen bestehen, beispielsweise durch Befehle, die genau diese Anwendungen besonders effizient unterstützen, während sie ansonsten durch Befehlsfolgen emuliert werden müßten. Diese Repräsentation ist normalerweise nicht anwendungsspezifisch, sondern auf optimale Ausführbarkeit für beliebige Anwendungen ausgelegt. Sie besteht beispielsweise aus wenigen grundlegenden Befehlen, die sich einfach und somit schnell ausführen lassen. Der Befehlssatz ist begrenzt, so daß Redundanz vermieden wird und die Komplexität des Interpreters sinkt.

Der Übersetzer von einer anwendungsgerechten in eine ablaufgerechte Repräsentation hat die Aufgabe, die semantische Lücke zwischen den beiden Repräsentationen zu überwinden. Somit kann ein Entwickler ein Objektsystem in einer anwendungsgerechten Repräsentation modellieren. Das Objektsystem liegt nach der Übersetzung in einer ablaufgerechten Repräsentation vor, die effizient ausgeführt wird.

Beispielsweise wird ein Aufruf der Funktion *f* mit Übergabe der Parameter *param1* und *param2* sowie der Rückgabe eines Ergebnisses in einer anwendungsgerechten Repräsentation folgendermaßen modelliert:

```
let result = f(param1 param2)
```

In der ablaufgerechten Repräsentation werden beispielsweise die Parameter auf einem Stapelspeicher abgelegt und dann die Funktion aufgerufen; anschließend wird das Resultat vom Stapelspeicher geholt und abgelegt:

```
LOAD param1
PUSH
```

```
LOAD param2
PUSH
CALL f
POP
STORE result
```

Auf der Gegenseite, also der aufgerufenen Funktion *f*, werden die Parameter innerhalb der Funktion benutzt, um das Ergebnis zu errechnen:

```
let f(p1 p2) = p1-p2
```

Die Parameter müssen vom Stack geholt werden, neben der Rücksprungadresse, die ebenfalls auf dem Stack liegt; außerdem muß das Resultat auf den Stack gelegt werden, bevor zurückgesprungen wird:

```
f:
POP
STORE returnAddress
POP
STORE p2
POP
STORE p1
LOAD p1
SUB p2
PUSH
LOAD returnAddress
PUSH
RET
```

Abstraktion und Übersetzung sind zusammen zu betrachten:

- In der anwendungsgerechten Repräsentation wird von der konkreten Realisierung des Funktionsaufrufs über einen Stapelspeicher abstrahiert. Es wäre auch eine andere Realisierung möglich, ohne daß die Modellierung der Anwendung geändert werden müßte.
- Die in der ablaufgerechten Repräsentation erzeugte Funktion und der Funktionsaufruf erfüllen das gleiche Protokoll, d.h. die Reihenfolge, in der die Parameter abgelegt werden, daß die Funktion den Stapelspeicher "leerräumt" und das Ergebnis ablegt usw. Wenn eine Seite ein anderes Protokoll verwendet, beispielsweise bezüglich der Reihenfolge der Parameter, würde der Aufruf nicht korrekt ablaufen.

5.1.3 Übersetzung der anwendungsgerechten in die ablaufgerechte Repräsentation

Die Übersetzung ist der Prozeß, in dem aus der anwendungsgerechten Repräsentation die ablaufgerechte Repräsentation eines Objektsystems erzeugt wird. Übersetzer werden in unterschiedlichen Kontexten verwendet.

- Es existieren verschiedene anwendungsgerechte Repräsentationen, die von Übersetzern in die gleiche ablaufgerechte Repräsentation übersetzt werden. Beispielsweise werden unterschiedliche höhere Programmiersprachen wie C, Ada und Java in Maschinencode für das Ablaufsystem Sun übersetzt.
- Eine anwendungsgerechte Repräsentation wird von verschiedenen Übersetzern in unterschiedliche ablaufgerechte Repräsentationen übersetzt. Beispielsweise wird die höhere Programmiersprache Java in Maschinencode für verschiedene Ablaufsysteme wie Sun oder Intel übersetzt [Diehl 98].

- Bei der Übersetzung werden unter Umständen Zwischenrepräsentationen erzeugt und weiterverarbeitet; diese Zwischenrepräsentation können, müssen aber nicht, ausführbar sein [Steensgard, Jul 95; Gosling 95; Gawecki, Matthes 96a; Lindholm, Yellin 96].
- Der Zeitpunkt der Erzeugung der ausführbaren Repräsentation ist variabel. Spätestens in dem Moment, in dem der Interpreter die Interpretation eines Objekts vornimmt, muß dieses Objekt in der von dem Interpreter verstandenen Repräsentation vorliegen. Die Erzeugung findet zu einem beliebigen Zeitpunkt vor der Interpretation statt. Insbesondere können Zwischenrepräsentationen erzeugt werden, die erst zu einem späteren Zeitpunkt in eine ausführbare Repräsentation übersetzt werden, um beispielsweise Portabilität von Objekten zu erreichen und gleichzeitig eine performante Ausführung zu gewährleisten [Steensgard, Jul 95; Plezbert, Cytron 97].

Die Aufgaben eines Übersetzers stellen sich wie folgt dar [Aho et al. 88; Schröder 93]:

Scanning. Der Übersetzer liest die Ausgangsrepräsentation, üblicherweise einen Zeichenstrom. Dieser Zeichenstrom wird lexikalisch in Symbole zerlegt, beispielsweise Schlüsselworte wie "begin" oder "end", Sonderzeichen wie "+" oder "(", Trennzeichen wie Leerzeichen oder Zeilenumbrüche.

Parsing. Nicht alle Symbolfolgen sind sinnvolle Sätze der Sprache. Daher wird die Menge der Symbolfolgen weiter eingeschränkt. Ein erster Schritt ist die Prüfung der Syntax, d.h. der Anordnung der Symbole. Beispielsweise könnte die Syntax festlegen, daß auf eine öffnende Klammer eine schließende Klammer zu folgen hat und sich zwischen diesen beiden Klammern nur bestimmte andere Symbole befinden dürfen. Die Spezifikation der Syntax erfolgt mit Hilfe kontextfreier Grammatiken oder eingeschränkten Unterklassen davon [Aho et al. 88]. Aus diesen kontextfreien Grammatiken lassen sich automatisch ablauffähige Parser generieren (vgl. Abschnitt 5.3), die beispielsweise einen deterministischen und endlichen Parsevorgang garantieren.

Scoping. In den gemäß einer Grammatik korrekten Sätzen existieren Kontextabhängigkeiten durch symbolische Bezeichner, beispielsweise bei Sprachen mit statischen Sichtbarkeitsregeln:

```
let i = 1
let f(j) =
  begin
    let i = 2
    i * j * k
  end
f(l)
```

Obwohl diese Ausdrücke syntaktisch korrekt sind, fehlt die Definition der Bezeichner k und l. Außerdem ist aus der Repräsentation allein nicht ersichtlich, an welche Definition von i das referenzierende Auftreten von i im Ausdruck $i * j * k$ gebunden ist. Der Übersetzer ordnet jedem referenzierenden Auftreten eines Bezeichners genau eine Definition zu. Wenn keine Definition gefunden wird, ist dies ein Fehler.

Codegenerierung. Aus der anwendungsgerechten Repräsentation wird eine ablaufgerechte Repräsentation erzeugt.

Ein Übersetzer übernimmt also alle Aufgaben, um aus der anwendungsgerechten Ausgangsrepräsentation eine ablaufgerechte Zielrepräsentation zu erzeugen. Bestimmte Teile der Aufgaben können aus dem Übersetzer in Hilfswerkzeuge wie Struktureditoren [Reps, Teitelbaum 89] ausgelagert werden. Aus grafischen Beschreibungen lassen sich Prototypen erzeugen und zur Validierung nutzen [Nietsch 97]. Außerdem werden in Übersetzer weitere Werkzeuge eingebunden, beispielsweise Typprüfer [Matthes 93; Cardelli 97b] (vgl. Abschnitt 5.2), welche die Abwesenheit bestimmter

Laufzeitfehler zusichern, oder Optimierer [Gawecki, Matthes 96a], welche die Laufzeitperformanz des ablauffähigen Codes erhöhen.

5.1.4 Entwurf der anwendungsgerechten Repräsentation

Die Gestaltung der anwendungsgerechten Repräsentation unterliegt mehreren Einflußfaktoren, beispielsweise:

Der Anwendungsbereich bildet den Rahmem, in dem die entwickelte Anwendung zur Lösung eines Problems beiträgt. Beispielsweise wird im kommerziellen Sektor die Datenmodellierung [Rumbaugh et al. 91; Blaha, Premerlani 97] und bei eingebetteten Echtzeitanwendungen die Zustands-/Ereignismodellierung [Harel 87] betont.

Der Projektkontext legt anhand der Verwendung der Anwendung fest, wie die Anwendung zu entwickeln ist. Beispielsweise sind länger andauernde, iterative Entwicklungsprojekte mit mehreren Entwicklern, die zu einem komplex strukturierten und langfristig gewarteten Produkt führen [Balzert 82], zu unterscheiden von kurzen Projekten eines Entwicklers, die schnell und einmalig zu einem "Wegwerf-"Prototypen führen [Ruble 97].

Das Entwicklungsparadigma beschreibt eine strukturierte Denkweise, die beim Entwickeln einer Anwendung zugrunde gelegt wird. Beispiele sind das imperative, funktionale, logische oder objektorientierte Paradigma [Pratt, Zelkowitz 96].

Je nach Ausprägung und Zusammenstellung der Einflußfaktoren entstehen andere anwendungsgerechte Repräsentationen. Durch die Übersetzung in die (gleiche) ablaufgerechte Repräsentation sind die mit den unterschiedlichen anwendungsgerechten Repräsentationen entwickelten Objektsysteme trotzdem auf dem gleichen Ablaufsystem ablauffähig und können eng miteinander kooperieren. Wenn keine gemeinsame ablaufgerechte Repräsentation vorliegt, hängt die Kooperationsform von dem Vorhandensein von Übersetzern zur Portierung bei der Übertragung von Objekten ab.

Ein Übersetzer ist durch die ablaufgerechte Repräsentation, die er erzeugt, eingeschränkt. Er kann keine Objektsysteme erzeugen, die Eigenschaften aufweisen, die nicht von dem Interpreter des Ablaufsystems unterstützt werden. Beispielsweise kann er keine Operationen zum Suspendieren von Prozessen oder zur Bindung externer Objekte erzeugen, wenn der Interpreter diese Operationen nicht unterstützt.

Umgekehrt muß sich nicht jede Eigenschaft des Interpreters in der anwendungsgerechten Repräsentation wiederfinden oder von dem Übersetzer ausgenutzt werden. Wenn beispielsweise der Interpreter Prozesse suspendieren und externe Objekte binden kann, muß eine Programmiersprache trotzdem nicht über Konstrukte verfügen, die diese Eigenschaften ausnutzen. Der Entwickler ist dann in dieser Programmiersprache eingeschränkt und kann nicht die volle Mächtigkeit des Ablaufsystems ausnutzen. Dies ist sinnvoll, wenn für bestimmte Anwendungen oder Teile eines Objektsystems nicht die volle Mächtigkeit benötigt wird und daher die mit dieser Mächtigkeit verbundene Komplexität versteckt oder mögliche Fehler bei der Benutzung dieser Eigenschaften von vornherein ausgeschlossen werden sollen.

Neben der vom Interpreter erbrachten Basisfunktionalität, die sich in der ablaufgerechten Repräsentation widerspiegelt, bietet das Ablaufsystem zusätzlich operationale Funktionalität durch Basisobjekte an (vgl. Abschnitt 2.5). Operationale Funktionalität kann sowohl durch den Interpreter als auch durch Basisobjekte des Ablaufsystems erbracht werden. Der Entwickler des Ablaufsystems legt fest, welche Möglichkeiten in welcher Form zur Verfügung stehen. Die Übersetzer der Entwicklungsumgebung sind durch diese Entscheidungen gebunden. Wenn benötigte operationale Funktionalität nicht direkt unterstützt wird, emulieren Übersetzer oder Generatoren (vgl. Abschnitt 5.3) die Funktionalität mit den vom Ablaufsystem angebotenen Mitteln. Voraussetzung dafür ist, daß eine minimale Basisfunktionalität bereitsteht. Wenn beispielsweise kein Zugriff auf persistente Objekte wie Dateien besteht, kann Persistenz nicht realisiert werden.

Anwendungsbereich, Projektkontext oder Entwicklungsparadigma variieren typischerweise zwischen einzelnen Entwicklungsprojekten [Sommerville 96]. Wenn für jedes Projekt ein anderes, spezialisiertes Entwicklungssystem eingesetzt wird, steigen die Entwicklungskosten, denn Ergebnisse

vorheriger Projekte können nur schwer, wenn überhaupt, weiterverwendet werden [Kueffmann 94; Sametinger 97] und der Einarbeitungsaufwand für die Entwickler wird größer [Sommerville 96]. Daher sollte ein Entwicklungssystem für mehrere Entwicklungsprojekte eingesetzt werden. Es werden zwei Arten von Entwicklungssystemen anhand der Konfiguration für ein Entwicklungsprojekt unterschieden [Matthes, Schmidt 91a]:

Obermengenansatz: Das Entwicklungssystem enthält eine Obermenge aller in den Entwicklungsprojekten benötigten Konzepte und deren Kombinationen. Die Konfiguration für ein Entwicklungsprojekt ergibt sich durch *Herausstreichen* nicht benötigter Konzepte und Kombinationen.

- Vorteile: Das Entwicklungssystem ist maximal vorkonfiguriert. Die vorhandenen Kombinationen sind konsolidiert und standardisiert. Optimierungen sind möglich.
- Nachteile: Naturgemäß ist der Umfang der angebotenen Konzepte und Kombinationen begrenzt, also sind die Entwicklungsprojekte darauf eingeschränkt. Je mehr Konzepte und Kombinationen angeboten werden, desto komplexer wird das Entwicklungssystem.

Differenzmengenansatz: Das Entwicklungssystem enthält disjunkte, orthogonal kombinierbare Konzepte. Für jedes Entwicklungsprojekt wird das Entwicklungssystem *erweitert*, indem die Konzepte neu kombiniert bzw. Kombinationen aus anderen Projekten übernommen werden.

- Vorteile: Das Entwicklungssystem weist eine geringe Komplexität auf. Die Anzahl und Art der Kombinationen ist nicht begrenzt. Durch iterative Auslese ergeben sich konsolidierte, standardisierungsfähige Kombinationen.
- Nachteil: Die Konfiguration der Umgebung erhöht die Komplexität für die Entwickler. Optimierungen sind nicht möglich.

Moderne Entwicklungssysteme verfolgen den Differenzmengenansatz, indem sie generische Dienste des Ablaufsystems integrieren und kombinieren. Die anwendungsgerechte Repräsentation bleibt dabei weitgehend unabhängig von Anwendungsbereich, Projektkontext und Entwicklungsparadigma. Um das Entwicklungssystem für ein Projekt zu erweitern, werden Generatoren eingesetzt [Schröder 93; Wetzel 95]. Um die Validierung der entwicklten Objektsysteme zuzusichern, werden Typsysteme höherer Ordnung benötigt [Matthes, Schmidt 91a; Matthes 93; Cardelli 97b].

5.2 Validierung

Um fehlerhafte Abläufe eines Objektsystems weitgehend schon während der Entwicklung auszuschließen, insbesondere bei großen Systemen mit getrennter Entwicklung der Systemteile [Lauer, Satterthwaite 79], werden beschreibende Informationen (Typen) eingesetzt, die durch Validierer (Typprüfer) ausgewertet werden (vgl. Abschnitt 5.2.1). Der Validierer entscheidet anhand der Typregeln eines Typsystems, ob ein Ausdruck wohlgeformt ist und garantiert damit die Abwesenheit bestimmter Fehler in dem Objektsystem.

Wenn Objektsysteme kooperieren, werden die Validierungen objektsystemübergreifend vorgenommen (vgl. Abschnitt 5.2.2). Da Objektsysteme in heterogenen Umgebungen mit unterschiedlichen Entwicklungssystemen erstellt werden, kooperieren auch die Entwicklungssysteme. Insbesondere tauschen sie beschreibende Informationen über die Objektsysteme aus. Dazu ziehen sie ein gemeinsames Typsystem heran.

Bei einer Kooperation mit einem generischen Objektsystem stehen die Typen der ausgetauschten Objekte nicht statisch fest. Daher werden beschreibende Objekte des Entwicklungssystems (Entwicklungsobjekte) in das Objektsystem eingebunden (vgl. Abschnitt 5.2.3). Während des Ablaufs werden diese Objekte zwischen Objektsystemen ausgetauscht und (dynamische) Validierungen vorgenommen. Zur Einbindung von Entwicklungsobjekten während der Entwicklung, genauer: während der Übersetzung des Objektsystems, wird linguistische Reflektion (vgl. Abschnitt 5.2.5) eingesetzt.

Evolution in kooperierenden Objektsystemen führt dazu, daß Objekte und damit ihre Typen sich ändern. Daher werden zusätzlich zu den Objekten auch die Typen versioniert [Skarra, Zdonik 87] (vgl. Abschnitt 5.2.6)

5.2.1 Typisierung

Ein Ziel der Entwicklung ist, fehlerhafte Abläufe des Objektsystems zu vermeiden. Beispielsweise führt die Ausführung folgender Ausdrücke zu fehlerhaftem Verhalten:

- Eine Zahl kann nicht zu einer Zeichenkette addiert werden.

  ```
  1+"hello"
  ```

- Eine Zahl kann nicht als Funktion aufgerufen werden.

  ```
  let i = 1
  i(2)
  ```

- Eine Funktion, die einen ganzzahligen Parameter erwartet, kann nicht ohne Parameter oder mit einer Zeichenkette als Parameter aufgerufen werden.

  ```
  let f(i) = i * i
  f()
  f("hello")
  ```

Die Ausdrücke sind syntaktisch korrekt, so daß auch eine (prinzipiell) ablauffähige Repräsentation erzeugt wird. Da sie aber semantisch nicht korrekt sind, sind die Ergebnisse der Abläufe unbestimmt, was zu Fehlschlüssen, Programmabbrüchen oder Schäden führt.

Um solche fehlerhaften Abläufe von vornherein auszuschließen, werden den Objekten beschreibende Informationen hinzugefügt. Diese Informationen sind insofern redundant, als daß sie für die Ausführung keine Auswirkungen besitzen, die zur Lösung des Anwendungsproblems beitragen. Anhand der beschreibenden Information wird eine Validierung vorgenommen, welche die Abwesenheit bestimmter Fehler während des Ablaufs des Ausdrucks garantiert.

Die Validierung sichert also zu, daß an bestimmten Stellen des Ablaufs definierte Bedingungen gelten. Im Vertragsmodell [Meyer 88] werden diese Bedingungen explizit in Prädikatenlogik ausformuliert. Nur wenn die Zusicherungen (*assertions*) erfüllt sind, wird der Ablauf fortgesetzt. Beispielsweise sichert die folgende Bedingung zu, daß nie durch 0 geteilt wird:

```
let f(i) =
  begin
    assert(i # 0)
    1 / i
  end
```

Wenn die Funktion mit dem Parameter 0 aufgerufen wird, schlägt die Zusicherung fehl und die Funktion wird mit einer Ausnahme beendet. D.h. ein Fehler wird frühzeitig (hier beim Eintritt in die Funktion) während des Ablaufs erkannt und somit die fehlerhafte Auswertung eines Ausdrucks verhindert.

Prädikate, die nie erfüllbar sind, also während eines Ablaufs immer zu einem Abbruch führen, sollten schon zur Entwicklungszeit erkannt werden.

```
assert(1 hasType Int)
assert("hello" hasType Int)
1+"hello"
```

In diesem Fall kann die Addition nie ausgeführt werden, weil die Zeichenkette *"hello"* nie den Typ *Int* haben wird und damit diese Bedingung immer zu einem Abbruch führt. Es handelt sich hier um einen statischen Fehler, also einen Fehler, der nicht vom (sich dynamisch ändernden) Ablaufzustand abhängt, sondern (statisch) in der Form des Ausdrucks begründet ist. Ausdrücke, deren Auswertung aufgrund statischer Eigenschaften immer zu einem Fehler führt, sind nicht semantisch wohlgeformt (vgl. Abbildung 5.3), auch wenn sie syntaktisch wohlgeformt sind.

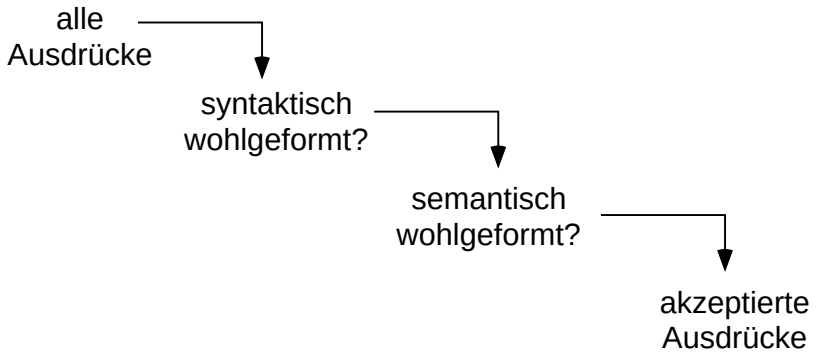

Abbildung 5.3: Wohlgeformtheit

Es ist für uneingeschränkte Prädikate unentscheidbar, ob sie jemals erfüllbar sind. Daher werden statische Informationen über die Form eines Ausdrucks gesondert erfaßt. Diese Informationen erlauben einem Validierer (vgl. Abbildung 5.4), zur Entwicklungszeit Aussagen über die semantische Wohlgeformtheit der Ausdrücke zu treffen. Der Validierer akzeptiert bestimmte Ausdrücke anhand der Beschreibungen als semantisch wohlgeformt und lehnt andere ab; je nach Mächtigkeit des Beschreibungssystems kann er auch für einige Ausdrücke keine Aussage treffen [Matthes 93]. Durch einen Validierer wird also die Menge der syntaktisch wohlgeformten Ausdrücke eingeschränkt auf die gemäß einer statischen Semantik wohlgeformten Ausdrücke.

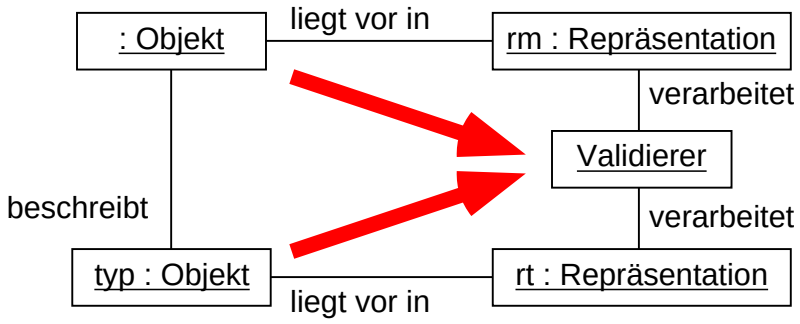

Abbildung 5.4: Validierer

Semantisch wohlgeformte Ausdrücke garantieren die Abwesenheit bestimmter Ablauffehler.

- Datenobjekte (hier: Zahlen und Zeichenketten) sowie die auf den Datenobjekten definierten Funktionsobjekte (hier die Additionsfunktion) werden mit Typinformationen versehen. Nur wenn die Typen zusammenpassen, ist der Ausdruck wohlgeformt. Beispielsweise addiert die Funktion *add* zwei Zahlen *Int*, so daß eine Zeichenkette *String* nicht vom Validierer akzeptiert wird.

  ```
  let add(i1, i2 :Int) :Int = ...
  ```

```
let s :String = "hello"
add(1 s) (* Value of type Int expected, value s of type String found. *)
```

- Objekte werden mit Informationen über ihre Art versehen; die auf die Objekte anwendbaren Operationen sind von der Art des Objekts abhängig. Beispielsweise ist auf Datenobjekten wie ganzen Zahlen der Funktionsaufruf nicht definiert:

```
let i :Int = 1
i(2) (* Function object expected, value i of type Int found. *)
```

- Funktionen können nur aufgerufen werden, wenn die Anzahl und die Art der Parameter stimmt:

```
let f(i :Int) :Int = i * i
f() (* Missing parameter. *)
f(1.1) (* Value of type Int expected, value 1.1 of type Real found. *)
```

Wie diese Beispiele zeigen, ist eine zentrale Aufgabe eines Validierers der Vergleich zweier Typen auf Kompatibilität (vgl. Abbildung 5.5). In den Beispielen werden u.a. die Typen *Int* und *String* miteinander verglichen. Nur wenn die Typen gemäß der Regeln des Typsystems [Matthes 93; Cardelli 97b] kompatibel zueinander sind, ist der Ausdruck, den diese Objekte bilden, wohlgeformt.

Um trotz der Typprüfung eine maximale Flexibilität zu erreichen, werden komplexe Typsysteme, beispielsweise mit parametrischem Polymorphismus [Cardelli, Wegner 85; Myers et al. 97; Odersky, Wadler 97], oder *type matching* [Gawecki, Matthes 96b] eingesetzt.

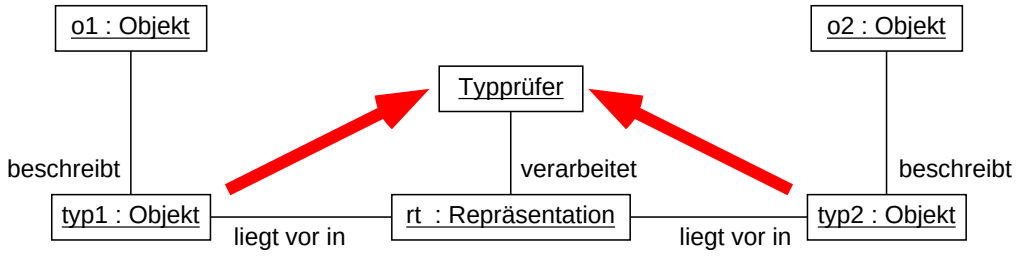

Abbildung 5.5: Subtypprüfer

Zusammenfassend ist die statische Validierung also folgendermaßen in die Entwicklung integriert:

- Jedem Objekt werden beschreibende Objekte, Typen genannt, zugeordnet.
- Über den beschreibenden Objekten existieren Aussagen (das Typsystem).
- Nur wenn ein Typprüfer anhand der beschreibenden Objekte die Richtigkeit der Aussagen zeigt, ist ein Ausdruck wohlgeformt.

Statische Validierungen über ein Typsystem können zur Entwicklungszeit nur stattfinden, wenn die Typinformationen verfügbar sind. Bei Kooperationsbeziehungen zwischen Objektsystemen stehen dagegen die Typinformationen teilweise erst während des Ablaufs zur Verfügung. In diesem Fall werden dynamische Validierungen eingesetzt, die während des Ablaufs die Wohlgeformtheit zusichern.

Es bleibt festzustellen, daß mit Hilfe eines Typsystems nur die Wohlgeformtheit gemäß einer statischen Semantik zugesichert werden kann. Beispielsweise kann ein Typsystem nicht zusichern, daß zur Laufzeit keine Division durch 0 auftritt:

```
let f(i :Int) :Int = 1 / i
f(0)
```

Die Erkennung solcher Fehler bleibt dynamischen Validierungen außerhalb des Typsystems, beispielsweise nach dem Vertragsmodell, vorbehalten. Als halbautomatische Validierungsalternative bieten sich Prototypen an [Nietsch 97].

5.2.2 Statische Validierung von Kooperationen

In Kooperationsbeziehungen verfügen Objektsysteme über externe Bindungen an Objekte anderer Objektsystem und lösen Operationen auf diesen Objekten aus. Wie bei lokalen Objekten werden fehlerhafte Abläufe vermieden, indem den externen Objekten und Operationen Typen zugeordnet und während der Entwicklung validiert werden [Stamos, Gifford 90].

Als Beispiel dient der Aufruf einer externen Funktion:

```
let f = bindExternalObject(otherObjectSystem theFunction)
f(1 "hello")
```

Die Bindung wird statisch oder dynamisch vorgenommen [Morrison et al. 90; OMG 98]:

Statische Bindung: Das Entwicklungssystem stellt während der Entwicklung über das Ablaufsystem eine Bindung an das externe Objekt her. Voraussetzungen dafür sind:

- Das exportierende Objektsystem ist bekannt.
- Das zu bindende Objekt ist bekannt.
- Das Entwicklungssystem kann über das Ablaufsystem externe Bindungen vornehmen.
- Auf der Gegenseite ist das Objektsystem bekannt und wird das Objekt von diesem Objektsystem exportiert.

Die externe Bindung wird mit Hilfe des Ablaufsystems während der Entwicklung aufgebaut und in das Objektsystem integriert. Ab diesem Zeitpunkt ist das Ablaufsystem für die Aufrechterhaltung der Bindung zuständig. Falls die Bindung ungültig wird, informiert das Ablaufsystem während der Entwicklung das Entwicklungssystem und während des Ablaufs das Objektsystem. Das Entwicklungssystem kann anhand statisch vorhandener Informationen über das Objekt statische Validierungen vornehmen.

Dynamische Bindung: Bei dynamischen Bindungen kann das Entwicklungssystem keine Bindung vornehmen. Dynamische Bindungen treten ein, wenn eine der folgenden Bedingungen gilt:

- Das exportierende Objektsystem ist (noch) nicht bekannt.
- Das zu bindende Objekt ist (noch) nicht bekannt.
- Das Entwicklungssystem kann über das Ablaufsystem keine externen Bindungen vornehmen.
- Auf der Gegenseite existiert das exportierende Objektsystem (noch) nicht oder das zu bindende Objekt wird (noch) nicht exportiert.

Das Entwicklungssystem kann nur eingeschränkt statische Validierungen vornehmen, indem es Annahmen über die dynamische Bindung trifft, die dann während des Ablaufs dynamisch validiert werden.

Im Vergleich zu einem lokalen Objekt existieren bei der Bindung an ein externes Objekt folgende Fehlerquellen während des Ablaufs des Objektsystems:

Bindung ungültig: Die Bindung an das Objekt ist ungültig. Im lokalen Fall wird vom Ablaufsystem zugesichert, daß dieser Fall nicht auftritt (vgl. Kapitel 3). Diese Aufgabe übernimmt das Ablaufsystem auch bei externen Objekten (vgl. Kapitel 4), allerdings mit der Einschränkung, daß die Bindung ungültig werden kann. Wenn bei statischen Bindungen die Bindung zur Entwicklungszeit nicht vorgenommen werden kann, stellt dies einen Entwicklungsfehler dar. Wenn bei dynamischen Bindungen die Bindung während des Ablaufs nicht vorgenommen werden kann, ist dies ein Ablauffehler.

Operation nicht verfügbar: Das Ablaufsystem des kooperierenden Objektsystems erlaubt es nicht, die gewünschte Operation auf dem Objekt auszuführen, beispielsweise einen Funktionsaufruf auf dem Objekt durchzuführen, oder die Operation wird fehlerhaft aufgerufen, beispielsweise bei einem Funktionsaufruf mit der falschen Anzahl von Parametern. Im lokalen Fall erzeugt das Entwicklungssystem keine Operationen auf Objekten, die vom Ablaufsystem nicht ausgeführt werden. Bei statischen Bindungen benötigt der Validierer des Entwicklungssystems also Informationen darüber, welche Operationen in welcher Form auf dem externen Objekt unterstützt werden. Wenn die gewünschte Operation nicht zur Verfügung steht oder fehlerhaft benutzt wird, stellt dies einen Entwicklungsfehler dar. Wenn die Menge der Operationen auf dem Objekt sich ändert, beispielsweise durch Evolution oder eine andere Sicherheitsstrategie, muß dies während der Entwicklung dem Entwicklungssystem und während des Ablaufs dem Objektsystem mitgeteilt werden, damit die Validierung wiederholt bzw. das Objektsystem geändert werden kann. Bei dynamischen Bindungen ergibt sich dementsprechend ein Ablauffehler, wenn die gewünschte Operation auf dem Objekt nicht verfügbar ist oder falsch genutzt wird.

Objekt oder Bindung nicht übertragbar: Ein Objekt oder eine Bindung kann zwischen den Objektsystemen nicht übertragen werden. Einige Operationen verlangen, daß ein Objekt oder eine Bindung an ein Objekt übertragen wird, beispielsweise als Parameter einer Funktion oder als Rückgabewert. Die Übertragung impliziert bei heterogenen Ablaufsystemen, daß das Objekt konvertiert wird. Wenn die Konvertierung nicht möglich ist, schlägt die Operation fehl. Ebenso muß eine Bindung, die übertragen wird, exportierbar sein. Es stellt also bei statischen Bindungen einen Entwicklungsfehler bzw. bei dynamischen Bindungen einen Ablauffehler dar, wenn ein Objekt oder eine Bindung nicht übertragen werden kann.

Das Entwicklungssystem des importierenden Objektsystems benötigt also zur statischen Validierung folgende Informationen:

- Welche Objekte werden vom Kooperationspartner exportiert?
- Welcher Art ist das exportierte Objekt, d.h. welche Operationen sind in welcher Form auf ihm definiert?
- Welche Objekte bzw. Bindungen sind übertragbar?

Die Anbindung der externen Objekte und die möglichen Operationen werden dem Entwicklungssystem in "Stummeln" (*stubs*) zur Verfügung gestellt [Birrell et al. 95]. Dabei handelt es sich um die Typbeschreibungen der externen Objekte:

```
let f :Fun(:Int :String) :Int = stub(otherObjectSystem theFunction)
```

In diesem Fall wird *f* der Typ einer Funktion zugeordnet. D.h. es wird angenommen, daß das während des Ablaufs extern angebundene Objekt den Typ einer Funktion besitzt, der Funktionsaufruf als Operation also unterstützt wird, und die Funktion die angegebenen Parameter erwartet. Der Funktionsaufruf wird an das Ablaufsystem des Objektsystems *otherObjectSystem*, das die Funktion exportiert, weitergeleitet. Dabei werden die Parameter übertragen und evtl. in ein anderes Format übersetzt (*marshalling*). Ebenso wird der Rückgabewert übertragen. Aus Sicht des Objektsystems stellt sich die Funktion *f* wie eine interne Funktion dar. Im Gegensatz zu lokalen

Funktionen kann es allerdings sein, daß der Aufruf der Funktion nicht möglich ist, weil das Objekt nicht zugreifbar ist oder die Operationen auf dem Objekt nicht mehr unterstützt werden.

Zu beachten ist dabei, daß es sich hier um eine Art Ad-Hoc Polymorphismus [Cardelli, Wegner 85] handelt. Die Funktion *f* ist nicht zu unterscheiden von einer lokalen Funktion, aber das Verhalten beim Funktionsaufruf ist ein anderes. Insbesondere können andere Fehler während des Aufrufs auftreten als im lokalen Fall.

Auch Operationen auf externen Objekten, die keine Funktionsobjekte sind, werden für das Entwicklungssystem oft als Funktionen dargestellt, d.h. die Stummel bestehen nur aus Funktionen.

```
let get_i :Fun() :Int = stubOp(otherObjectSystem getOp i)
let set_i :Fun(:Int) :Ok = stubOp(otherObjectSystem setOp i)
```

Beispielsweise exportiert das Objektsystem *otherObjectSystem* das ganzzahlige Basisobjekt *i*. Auf Basisobjekten seien die Operationen *getOp* zum Lesen des Wertes und *setOp* zum Setzen des Wertes definiert. Lokal werden die Funktionen *get_i* und *set_i* definiert, welche die Operationen zum Lesen und Setzen des Wertes auslösen. Damit steht das externe Basisobjekt *i* nicht gleichberechtigt neben lokalen Basisobjekten, denn es wird kein Ad-Hoc-Polymorphismus unterstützt. Diese Art der Einbindung ist vor allem nötig, wenn das lokale und das entfernte Objektsystem über unterschiedliche Operationen auf gleichen Objektarten verfügen. Wie in Kapitel 3 kann über Reflektion des Ablaufsystems eine Art Ad-Hoc-Polymorphismus realisiert werden.

Es bleibt die Frage, ob beispielsweise die Operationen *getOp* und *setOp* auf *i* unterstützt werden, also allgemein, welche Operationen auf welchen Objekten definiert sind. Um diese Frage zu beantworten, kooperieren die beiden Entwicklungssysteme: Das Entwicklungssystem des exportierenden Objektsystems stellt eine Beschreibung der exportierten Objekte zur Verfügung, die das Entwicklungssystem des importierenden Objektsystems nutzt, um Validierungen vorzunehmen und beispielsweise Funktionsrümpfe zu generieren.

- Das Entwicklungssystem des exportierenden Objektsystems sichert zu, daß eine Bindung an ein Objekt dieser Art exportiert wird.
- Das Entwicklungssystem des importierenden Objektsystems sichert zu, daß auf dem extern gebundenen Objekt nur die für Objekte dieser Art zulässigen Operationen angewendet werden, evtl. sogar noch weiter eingeschränkt.

Da die beiden Objektsysteme sowie ihre Ablauf- und Entwicklungssysteme autonom sind, bleiben mögliche Fehlerquellen:

- Das Ablaufsystem des exportierenden Objektsystems stellt die Operation(en) auf dem Objekt nicht zur Verfügung, beispielsweise aus Sicherheitsgründen.
- Das exportierende Objektsystem exportiert ein Objekt einer anderen Art, so daß die Operationen nicht anwendbar sind, beispielsweise weil es im Zuge der Evolution weiterentwickelt wurde.
- Das importierende Objektsystem nimmt einen falschen Typ für das Objekt an und spricht so die Operationen fehlerhaft an, beispielsweise indem es versucht, ein Datenobjekt als Funktionsobjekt anzusprechen oder beim Aufruf eines Funktionsobjekts die falschen Parameter überträgt.

Um diese Fälle abzufangen, werden dynamische Validierungen eingesetzt (vgl. Abschnitt 5.2.3).

5.2.3 Dynamische Validierung von Kooperationen

Nicht alle Typeinschränkungen können statisch validiert werden [Abadi et al. 89; Birrell et al. 95]. Bei generischen Operationen zwischen Objektsystemen, beispielsweise dem Lesen und Schreiben auf einer Datenbank, können prinzipiell beliebige Objekte übertragen werden. Bei der Entwicklung eines Objektsystems werden Annahmen darüber getroffen, welcher Art die gebundenen oder übertragenen Objekte sind, und das Objektsystem anhand dieser Annahmen validiert.

Während des Ablaufs ist sicherzustellen, daß die während der Entwicklung getroffenen Annahmen noch gültig sind, d.h. daß die gebundenen oder übertragenen Objekte von der angenommenen Art sind. Dazu werden die Objekte mit ihren Typbeschreibungen gekapselt und die Typbeschreibungen zur dynamischen Validierung während des Ablaufs ausgewertet [Herlihy, Liskov 82; Abadi et al. 89], beispielsweise mit *typecodes* in CORBA [Vogel, Duddy 97].

Wenn beispielsweise ein Objekt zwischen zwei Objektsystemen übertragen wird, kann bei der Entwicklung des empfangenden Objektsystems nicht statisch zugesichert werden, daß ein Objekt des erwarteten Typs übertragen wird.

```
let i :Int = receiveObject(otherObjectSystem)
```

In diesem Beispiel wird implizit angenommen, daß eine ganze Zahl übertragen wird, und dem übertragenen Objekt daher der lokale Typ *Int* zugeordnet. Dieser Ansatz kann zu Fehlern führen, wenn das kooperierende Objektsystem von anderen Annahmen ausgeht und daher beispielsweise eine reelle Zahl überträgt. Selbst wenn beide Objektsysteme zum Entwicklungszeitpunkt über eine gemeinsame Annahme verfügen, kann diese Annahme durch Evolution (vgl. Abschnitt 3.3) nach der Entwicklung invalidiert werden.

Um trotzdem eine Validierung der Annahmen während des Ablaufs zu erlauben, gibt der Sender den Typ des Objekts dem Objekt mit und der Empfänger vergleicht diesen Typ mit seiner Annahme.

```
let o = receiveObject(otherObjectSystem)
if areCompatible(typeOf(o) typeRepOf(Int)) then
  let i :Int = valueOf(o)
end
```

Dies setzt folgendes voraus:

- Die beiden kooperierenden Objektsysteme müssen über eine gemeinsame Typrepräsentation (evtl. mit zwischengeschalteten Übersetzern [Horn, Cahill 91]) verfügen, um einen Austausch der Typen zu erlauben. D.h. die Entwicklungsumgebungen der beiden Objektsysteme müssen aufeinander abgestimmt sein. Nur so kann das empfangende Objektsystem mit *typeOf* die Typrepräsentation des übertragenen Objekts extrahieren.[1]

- Die Typen müssen sowohl beim Sender als auch beim Empfänger zur Laufzeit verfügbar sein (dynamische Typen). Dies setzt Reflektion in der Entwicklungsumgebung voraus (vgl. Abschnitt 5.2.5). Im Beispiel erzeugt die Operation *typeRepOf* eine Typrepräsentation des Typs *Int*, die in das erzeugte Objektsystem eingebunden wird. Dies bedeutet also auch, daß der Validierer in einen Übersetzer integriert sein muß.

- Es muß während des Ablaufs eine Validierungsfunktion des Typprüfers zur Verfügung stehen, welche die beiden Typen miteinander vergleicht. Diese Funktion kann ebenfalls durch Reflektion eingebunden werden, sie kann aber auch explizit als Standardobjekt zur Verfügung stehen. Im Beispiel handelt es sich um die Funktion *areCompatible*, die zwei Typrepräsentationen auf Kompatibilität prüft.

5.2.4 Kooperation der Entwicklungssysteme zur Validierung

Die Informationen über die exportierten Objekte liegen nicht primär in dem Entwicklungssystem des importierenden Objektsystems vor, sondern nur in den Entwicklungssystemen der exportierenden Objektsysteme. Die Entwicklungssysteme kooperieren, indem sie beschreibende Informationen über die externen Objekte und die verfügbaren Operationen austauschen [Auerbach et al. 94; Mainetto et al. 95; Birrell et al. 95; van der Hoek et al. 97]. Aufgrund der Heterogenität können sich die beteiligten Ablauf- und Entwicklungssysteme erheblich unterscheiden, so daß unterschiedliche

[1] Ein hier nicht weiter betrachteter Sonderfall der Kooperation ist, daß ein Objektsystem Objekte nicht interpretiert, sondern uninterpretiert weiterreicht. In diesem Fall wird kein gemeinsames Typsystem benötigt.

Typsysteme vorliegen [Senivongse, Utting 96]. Es stellt sich insbesondere die Frage, welche Typen der Entwicklungssysteme kompatibel sind, d.h. ob sie über vergleichbare Operationen verfügen oder ob die von ihnen beschriebenen Objekte ineinander übersetzt werden können, wenn sie übertragen werden [Puder, Geihs 97]. Beispielsweise können ganze Zahlen des Typs *Int* in einem System mit 32 Bit und in einem anderen mit 64 Bit repräsentiert werden; in diesem Fall sind die beiden Objekte nicht direkt kompatibel, wenn nicht das übertragene Objekt mit Hilfe eines Übersetzers umgewandelt wird. Außerdem können unterschiedliche Operationen auf diesen Zahlen definiert sein. Es ist daher möglich, daß keine eindeutige Abbildung der Typsysteme aufeinander existiert:

- Es gibt keinen äquivalenten Typ in dem anderen Entwicklungssystem, d.h. keinen Typ, der diese Informationen enthält oder diese Operationen aufweist.
- Operationen, die lokal auf Objekten eines Typs gelten, sind nicht in der gleichen Weise auf externe Objekte dieses Typs anwendbar.
- Lokale Objekte eines Typs können nicht (vollständig) in Objekte des äquivalenten Typs eines anderen Entwicklungssystem umgewandelt werden.

Um eine Kooperation zu ermöglichen, werden folgende Maßnahmen ergriffen:

- Fehlende Typen und Operationen werden emuliert, d.h. neue Typen werden eingeführt, auf denen nur externe Operationen möglich sind.
- Operationsaufrufe werden angepaßt, d.h. das Verhalten wird durch weitere Operationen, entweder interne oder externe, so verändert, daß es wie das Verhalten der internen Operationen wirkt.
- Die Abbildungen zwischen den Objekten werden unvollständig, d.h. entweder geht Semantik verloren oder bestimmte Situationen werden durch dynamische Validierungen abgefangen und führen so zu kontrollierten Ausnahmesituationen.

Bei Datenstrukturen, den in jeder Kooperationsform übertragenen Objekten (vgl. Kapitel 1) stellt sich die Frage, welche Kompatibilitätsregeln gelten (vgl. hierzu auch [Matthes 93]). Angenommen, in beiden Systemen existiert ein Personentyp und ein Objekt dieses Typs wird übertragen: wie stellt das empfangende Objektsystem fest, ob das Objekt kompatibel zum lokalen Personentyp ist?

- Wenn die Typidentität nur lokal definiert, d.h. es keine globale Instanz gibt, die Typidentitäten verwaltet, können die beiden lokalen Typen nicht auf Identität getestet werden, weil sie unabhängig voneinander entstanden sind und somit immer unterschiedliche Identitäten besitzen.
- Die Namen der beiden Typen werden miteinander verglichen (Namensäquivalenz). Diese Methode ist unsicher, denn die Namen können durchaus unterschiedlich definiert sein, beispielsweise:

```
Let Person =              Let Person =
Tuple                     Tuple
  name :String              surname :String
  age :Int                  familyname :String
end                         address :String
                          end
```

  In diesem Fall sind die beiden Typen *Person* nicht kompatibel.
- Allerdings kann es sein, daß die Namen zwar unterschiedlich sind, beispielsweise *Person* und *PersonType*, sich aber strukturell voll entsprechen, also beide folgendermaßen definiert sind:

```
Tuple
  name :String
  age :Int
end
```

In diesem Fall sind die beiden Typen *Person* und *PersonType* miteinander kompatibel (strukturell äquivalent).

- Es stellt sich die Frage, ob auch folgender Typ strukturell äquivalent ist:

```
Let Person =
Tuple
  theName :String
  theAge :Int
end
```

  In diesem Fall sind die Namen der einzelnen Felder zwar unterschiedlich, aber ihren Typen sind kompatibel.

- Wenn die Namen keine Rolle mehr spielen, fallen alle kompatiblen Typen auf folgendes Schema zusammen:

```
Tuple
  :String
  :Int
end
```

  Insbesondere fällt auch folgender Typ darunter:

```
Let Car =
Tuple
  name :String
  maxSpeed :Int
end
```

  Es ist wahrscheinlich nicht gewünscht, daß dieser Typ zu dem Typ *Person* kompatibel ist.

- Um die beiden Typen *Person* und *Car* voneinander zu unterscheiden, werden abstrakte Datentypen[2] eingeführt:

```
Let Person = Abstract Tuple ...
Let Car = Abstract Tuple ...
```

  In diesem Fall wird die Strukturäquivalenz der beiden Typen nicht aufrechterhalten, sondern es wird wieder auf die Identität zurückgeschaltet. Damit ist es aber nicht mehr möglich, ein Objekt dieses Typs typsicher zu einem anderen Objektsystem zu übertragen, wenn dieser Typ nur eine lokale Identität besitzt.

- Das bedeutet, es muß eine den beiden Objektsystemen übergeordnete Instanz existieren, welche die Typidentitäten verwaltet, beispielsweise in einem gemeinsamen Namensraum [Kaplan, Wileden 95a; Kaplan, Wileden 95b; OMG 98]. Die Entwicklung der Objektsysteme ist nicht mehr unabhängig voneinander. Diese gemeinsame Instanz findet sich meistens in der gemeinsamen Repräsentation für Typen wieder, in der einige Typen festgelegt sind. In diesem Fall ist es aber nicht möglich, daß kooperierende Objektsysteme weitere gemeinsame abstrakte Datentypen besitzen. Eine solche Standardisierung findet beispielsweise durch MIME Typen im Kontext des Internet statt [Cardelli 97a]. Mit Hilfe so eines gemeinsamen Typsystems wird der Heterogenität der Ablauf- und Entwicklungsumgebungen begegnet.

Die Kooperation der Entwicklungssysteme bildet also die Grundlage für die Validierung der Kooperation von Objektsystemen. Die Entwicklungssysteme sichern jeweils für das von ihnen erzeugte Objektsystem bestimmte Bedingungen zu, die während des Ablaufs eingehalten werden. Sie verfügen über ein zumindest in Teilen gemeinsames Typsystem, über das sie sich verständigen.

[2] [Matthes 93] zeigt eine andere Möglichkeit zur Definition abstrakter Datentypen.

5.2.5 Reflektive Unterstützung der Validierung

Teile des Entwicklungssystems werden auch während des Ablaufs eines Objektsystems eingesetzt, beispielsweise um den Ablauf dynamisch zu validieren (vgl. Abschnitt 5.2.3) oder plattformabhängigen Code zu erzeugen. Während des Ablaufs des Objektsystems steht das Entwicklungssystem normalerweise nicht mehr zur Verfügung. Es stellt sich daher die Frage, wie diese Teile des Entwicklungssystems in das Objektsystem eingebunden werden, so daß sie während des Ablaufs zur Verfügung stehen. Bei der linguistischen Reflektion [Kirby et al. 96] werden Objekte des Entwicklungssystems während der Übersetzung reflektiv in das entwickelte Objektsystem eingebunden, d.h. es werden Objekte des Entwicklungssystems, mit dem das Objektsystem entwickelt wird, in das Objektsystem eingebunden. Es handelt sich dabei sowohl um statische als auch um dynamische Objekte des Entwicklungssystems:

Statische Entwicklungsobjekte sind Teile des Entwicklungssystems und damit unabhängig vom entwickelten Objektsystem. Es handelt sich dabei beispielsweise um Funktionen wie den Typprüfer, den Optimierer oder den Codeerzeuger, die während des Ablaufs des Objektsystems aufgerufen werden. Während einer Kooperation werden Objekte inklusive ihrer beschreibenden Typobjekte ausgetauscht; zur Entwicklungszeit getroffene Annahmen über die Objekte werden anhand der Typobjekte durch den Typprüfer validiert. Ein Optimierer kann während des Ablaufs des Objektsystems Optimierungen vornehmen, wenn dynamische Bindungen statisch geworden sind [Gawecki, Matthes 96a]. Ein Codeerzeuger kann zur Laufzeit auf das Ablaufsystem angepaßten Code aus portablem Code erzeugen [Steensgard, Jul 95; Diehl 98].

Dynamische Entwicklungsobjekte sind vom Objektsystem abhängige Objekte, die das Entwicklungssystem während der Entwicklung erzeugt. Beispielsweise werden Typobjekte in das Objektsystem eingebunden, um während des Ablaufs dynamische Validierungen bei der Kooperation mit anderen Objektsystemen durchzuführen. Zwischenrepräsentationen der Objekte dienen als portable Repräsentationen, die in anderen Objektsystemen auf anderen Ablaufsystemen mit Hilfe von Teilen des dortigen Entwicklungssystems, die reflektiv eingebunden wurden, optimiert oder ablauffähig gemacht werden.

Die Einbindung der Entwicklungsobjekte in das Objektsystem durch linguistische Reflektion hängt von der Art des Entwicklungssystems ab (vgl. Abbildung 5.6).

- Wenn das Entwicklungssystem auf demselben Ablaufsystem aufbaut wie das Objektsystem, können die Entwicklungsobjekte als normale Objekte eingebunden werden. Dies ist der Fall, wenn das Entwicklungssystem mit sich selbst entwickelt wurde; dieser Vorgang wird *bootstrap* genannt [Lecarme, Peyrolle-Thomas 78]. Es sind evtl. Mechanismen notwendig, um sie aus dem Entwicklungssystem zu lösen, beispielsweise eine Kopieroperation. Wenn das Entwicklungssystem ein persistentes Objektsystem ist, wird dieser Vorgang durch die Speicherverwaltung (vgl. Abschnitt 3.1.4) unterstützt [Wienberg 97]. Während des Ablaufs des Objektsystems sind die Entwicklungsobjekte voll in das Objektsystem integriert und nicht von Objekten des Objektsystems zu unterscheiden.

- Wenn das Entwicklungssystem auf einem anderen Ablaufsystem als das Objektsystem aufbaut, werden die Entwicklungsobjekte über externe Bindungen in das Objektsystem eingebunden oder in interne Objekte konvertiert. Dies ist beispielsweise der Fall bei einem in C++ entwickelten Java-Entwicklungssystem, das nicht die Java-Maschine als Ablaufsystem nutzt. Die gebundenen Entwicklungsobjekte müssen auch während des Ablaufs des Objektsystems noch existieren, d.h. sie müssen persistent sein, wenn das erzeugte Objektsystem persistent ist (vgl. Abschnitt 3.2).

Linguistische Reflektion besteht aus folgenden Mechanismen des Entwicklungssystems, jeweils für statische und dynamische Objekte:

- Identifizierung der Objekte, d.h. der Entwickler muß die einzubindenden Objekte benennen können.

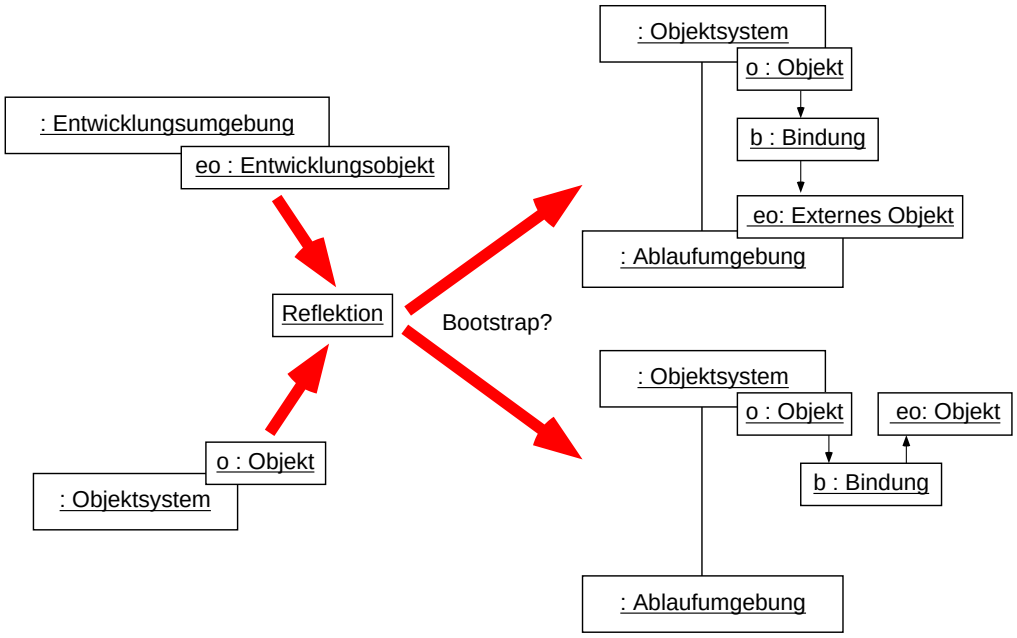

Abbildung 5.6: Linguistische Reflektion

- Persistente Repräsentation der Objekte, d.h. die Objekte müssen in einer Form verfügbar sein, in der sie auch während des Ablaufs des Objektsystems angesprochen werden können; transiente Entwicklungsobjekte können nicht reflektiv eingebunden werden.
- Bindung an die Objekte, d.h. je nach Art des Entwicklungssystems muß eine interne oder eine externe Bindung erzeugt werden.

Diese Mechanismen sind in den Übersetzer (vgl. Abschnitt 2.3.2) integriert.

Beispielsweise wird linguistische Reflektion genutzt, um Typinformationen, also Typobjekte des Entwicklungssystems, in das Objektsystem einzubinden.

```
let i :Int = 1
let type_i = typeOf(i)
```

Hier wird ein Objekt *type_i* mit den Typinformationen zu dem Objekt *i* erzeugt, hier also dem Typ *Int*. Während der Übersetzung wird *typeOf(i)* durch eine Bindung an das Typobjekt, oder allgemeiner einen Generator für das Typobjekt [Geisler 95], ersetzt. Zur Laufzeit steht zu dem Objekt *i* der Typ als beschreibende Information in dem Objekt *type_i* bereit, um ausgewertet zu werden.

```
if areCompatible(type_i type_j) then
  compare(i j)
end
```

Hier wird das Typobjekt *type_i* mit dem Typobjekt *type_j* auf Kompatibilität geprüft; wenn beide Typen kompatibel sind, werden die zugehörigen Objekte *i* und *j* miteinander verglichen. Um die Kompatibilität der Typen zu prüfen, wird eine Funktion *areCompatible* eingesetzt. Diese Funktion ist auf den Typobjekten definiert. Da die Typobjekte Objekte des Entwicklungssystems sind, handelt es sich dabei auch um eine Funktion des Entwicklungssystems, die per linguistischer Reflektion eingebunden wird.

```
let areCompatible = reflect("type compatibility checker")
```

Die Einbindung dieses statischen Entwicklungsobjekts erfolgt über *reflect* [Geisler 95]. Das einzubindende Objekt wird über eine Zeichenkette beschrieben. Statische Objekte des Entwicklungssystems, die in das Objektsystem reflektiv eingebunden werden können, werden unter einer eindeutigen Beschreibung registriert [Koch 97].

Zu beachten ist insbesondere bei der Reflektion die Evolution von Objektsystemen, in diesem Fall für Entwicklungssysteme: Beispielsweise werden (die Repräsentation der) Typobjekte und die Prüffunktionen weiterentwickelt. In diesem Moment sind sie nicht mehr kompatibel zu älteren (vom alten Entwicklungssystem erzeugten) Typobjekten, weil sich der Typ der Typobjekte geändert hat, es sich also um eine neue Version handelt. Ältere Typobjekte können nicht mit den neueren Prüffunktionen bzw. umgekehrt neuere Typobjekte nicht mit den alten Prüffunktionen verglichen werden. D.h. die Typobjekte selber brauchen beschreibende Typobjekte, damit festgestellt werden kann, ob sie überhaupt verglichen werden können bzw. mit welchen Funktionen sie bearbeitet werden können. Da bekannt ist, daß der Typ eines Typobjekts wieder ein Typobjekt ist, reicht hier die Versionsinformation aus, um die Kompatibilität zu prüfen. Es wird also zu den Typobjekten vermerkt, zu welcher Version des Typsystems sie gehören.

Wenn Reflektion bei der Entwicklung von Entwicklungssystemen, dem *bootstrap*, genutzt wird, führt dies potentiell zu Typunsicherheiten. Angenommen, es existiert eine Typvergleichsfunktion, die reflektiv exportiert werden soll und daher mit ihrer Beschreibung und ihrem Typ registriert wird:

```
registerReflectObject(typeCompare typeOf(typeCompare) "type checker")
```

Sie wird mit einem Typobjekt des Entwicklungssystems n, mit dem das Entwicklungssystem $n+1$ entwickelt wird, abgelegt. D.h. in dem erzeugten Entwicklungssystem $n+1$ existiert ein Typobjekt, das von dem Typprüfer des Entwicklungssystems n akzeptiert wird. Wenn das Typsystem sich weiterentwickelt hat, kann der neue Typprüfer des Entwicklungssystems $n+1$ diesen Typ nicht verarbeiten. Angenommen, nun bindet ein mit diesem Entwicklungssystem $n+1$ entwickeltes Objektsystem diese Funktion ein.

```
let areCompatible = reflect("type checker")
```

Diese Einbindung ist typunsicher, denn der Typ von *areCompatible* ist nicht bekannt. Um die Typsicherheit zu erhalten, muß dem Bezeichner *areCompatible* ein Typ gegegeben werden und es muß geprüft werden, ob dieser Typ dem Typ der reflektiv eingebundenen Funktion entspricht.

```
let areCompatible(:Type :Type) :Bool =
  reflect("type checker" typeRepOf(Fun(:Type :Type) :Bool))
```

Hier legt das Entwicklungssystem $n+1$ ein Typobjekt für den Typ *Fun(:Type :Type) :Bool* an. Dieses Typobjekt kann nicht mit dem für die Funktion abgelegten Typobjekt verglichen werden, weil beide von verschiedenen Entwicklungssystemen stammen. Erst wenn das Entwicklungssystem $n+1$ (ohne Änderungen) ein Entwicklungssystem $n+2$ erzeugt und dabei ein Typobjekt für die Funktion angelegt hat, kann ein Vergleich durchgeführt werden.

Um diesen Fall zu vermeiden, muß das reflektiv registrierte Objekt mit einem explizit im neuen Typsystem erzeugten Typobjekt registriert werden:

```
let typeObject = ...
registerReflectObject(typeCompare typeObject "type checker")
```

Hier wird explizit ein Typobjekt *typeObject* des Entwicklungssystem $n+1$ erzeugt. Allerdings kann nun nicht das Entwicklungssystem n genutzt werden, um sicherzustellen, daß dieses Typobjekt wirklich die Funktion *typeCompare* beschreibt; diese Zusicherung muß der Entwickler übernehmen.

5.2.6 Versionierung im Typsystem

Durch Evolution wird das Typsystem um eine Dimension erweitert: die Versionen [Herlihy, Liskov 82]. Ein Typ wird geändert, d.h. es werden beispielsweise Attribute hinzugefügt, entfernt oder

verändert.

```
Let Person.1 =          Let Person.2 =
Tuple                   Tuple
  name :String            name :String
  age :Int                birthday :Date
end                     end
```

Beispielsweise wird beim Typ *Person* das Attribut *age* durch ein Attribut *birthday* ersetzt. Dies hat folgende Auswirkungen:

- Es existieren Objekte des gleichen Typs in verschiedenen Versionen:

  ```
  let peter :Person.1 = tuple "Peter" 27 end
  let paul :Person.2 = tuple "Paul" 13-FEB-1970 end
  ```

  peter und *paul* sind zwar beide Personen, aber gemäß unterschiedlicher Versionen des Typs *Person*.

- Um die Objekte der verschiedenen Versionen zu verarbeiten, existieren Operationen, die auf dem gleichen Typ in unterschiedlichen Versionen arbeiten:

  ```
  let isOlder.1(p1, p2 :Person.1) :Bool = p1.age > p2.age
  let isOlder.2(p1, p2 :Person.2) :Bool = date.before(p1.birthday  p2.birthday)
  ```

  Eine Funktion *isOlder*, die feststellt, ob eine Person älter ist als die andere, ist unterschiedlich für die beiden Typen *Person.1* und *Person.2* realisiert.

Eine Version eines Typs kann nur eleminiert werden, wenn (vgl. Abschnitt 3.3)

- alle Objekte dieses Typs in Objekte des neuen Typs umgewandelt werden und
- alle Operationen, die auf Objekten des alten Typs arbeiten, durch Operationen ersetzt werden, die auf dem neuen Typ definiert sind.

Bei kooperierenden Objektsystemen müßten diese Umwandlungen bzw. Ersetzungen in allen Objektsystemen erfolgen, die miteinander kooperieren bzw. jemals kooperieren werden. Dies kann aufgrund der Autonomie der einzelnen Objektsysteme meistens nicht gewährleistet werden. Es ist dann anzunehmen, daß noch Objekte des alten Typs oder Operationen, die auf dem alten Typ arbeiten, in anderen Objektsystemen existieren. Daher wird die Version eines Typs in das Typsystem aufgenommen und beim Übertragen von Objekten geprüft.

Dies gilt auch für lokale Typen, d.h. Typen von Objekten, die nur in einem Objektsystem erzeugt und bearbeitet werden, wenn diese Objekte übertragen werden (vgl. Abbildung 5.7). Wenn das Objektsystem mit anderen Objektsystemen kooperiert und diese Objekte überträgt, verlassen sie den Einflußbereich des Objektsystems. Wenn sie später, nach einer Evolution, zurückkehren, muß sichergestellt sein, daß keine falschen Typannahmen getroffen werden.

```
Let Person = Tuple name :String  age :Int  end
let peter :Person = tuple "Peter" 27 end
sendObject(otherObjectSystem peter)
Let Person = Tuple name :String  birthday :Date end
let o = receiveObject(otherObjectSystem)
isCompatible(typeOf(o) typeRepOf(Person))
```

Das Objekt *peter* wird gemäß der ersten Version des Typs *Person* konstruiert und an ein anderes Objektsystem übertragen. Anschließend wird der Typ *Person* lokal neu definiert. Wenn im gleichen Zuge alle Objekte und Operationen dieses Typs umgewandelt werden, ist der alte Typ überflüssig.

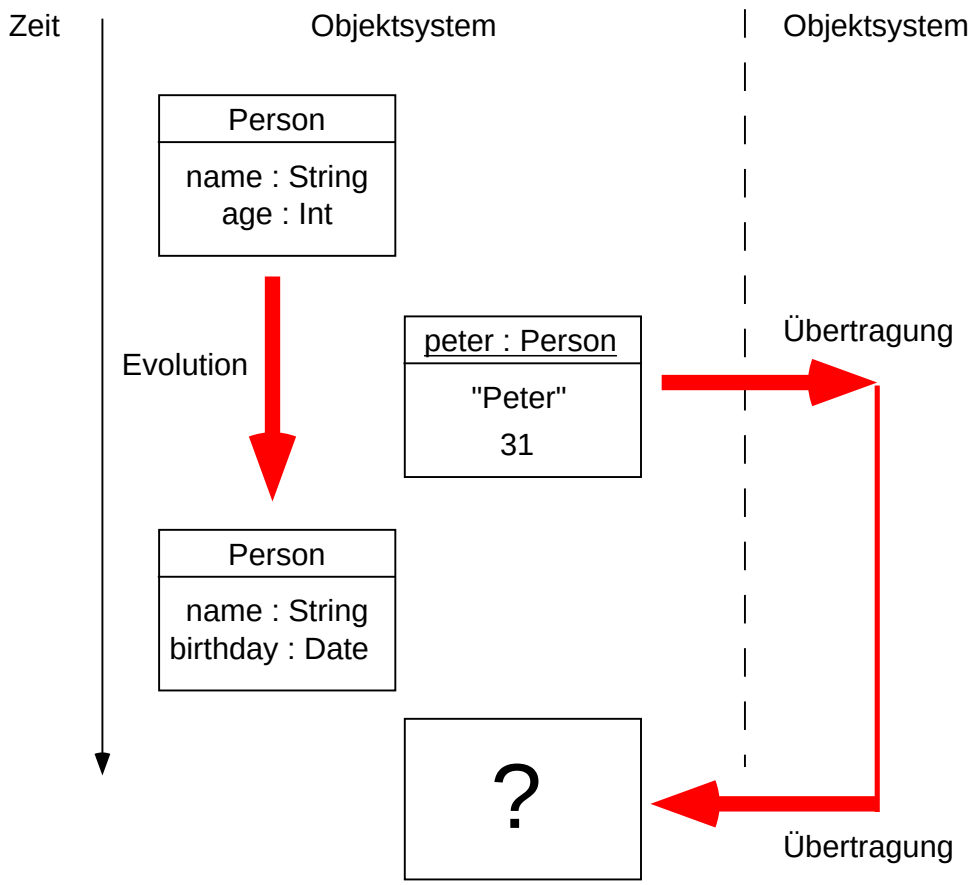

Abbildung 5.7: Evolution von Typen bei Übertragung von Objekten

Wenn jetzt aber das Objekt zurückübertragen wird, besitzt es nicht den (aktuellen) Typ *Person*, sondern die vorherige Version. Es kann nur mit den Operationen bearbeitet werden, die auf der alten Version von *Person* definiert sind.

Daher versioniert eine Entwicklungsumgebung für kooperierende Objektsysteme Typen sowie die auf diesen Typen definierten Objekte und Operationen.

5.3 Generierung

Ziel der Entwicklung (bzw. Weiterentwicklung) ist die Erstellung (bzw. Änderung) eines Objektsystems, das ablauffähig ist, seine Objekte persistent ablegt und mit anderen Objektsystemen kooperiert. Daher liegen die Objekte des Objektsystems in Repräsentationen vor, die Eigenschaften wie *ablauffähig*, *portabel* oder *persistent* aufweisen, und es existieren Übersetzer, die Repräsentationen mit den gewünschten Eigenschaften aus anderen Repräsentationen erzeugen [Feldmann 79; Feldmann 88; Koch 96; Koch 97].

Während der Entwicklung durch die Modellierung von Objektsystemen treten Muster (*patterns*) im Sinne ähnlicher Zusammenstellungen von Objekten auf [Gamma et al. 95; Sametinger 97], beispielsweise in Form von Softwarearchitekturen für kooperierende Systeme [Shaw, Garlan 96; Lamb 96] oder Kooperationsstrategien [Herlihy, Liskov 82; Auerbach et al. 94; Wolff, Löhr 96]. Muster zeichnen sich durch Gleichartigkeit aus, die in Regeln niedergelegt ist, deren Verletzung einen Fehler darstellt. Diese Fehler werden von der Validierung nur erkannt, wenn die Regeln bei

der Entwicklung des Entwicklungssystems bekannt sind und die anwendungsgerechte Repräsentation die Entwicklungsmuster unterstützt. Da Entwicklungssysteme aber größtenteils den Anspruch verfolgen, allgemein verwendbar zu sein, implementieren sie nur wenige Entwicklungsmuster, beispielsweise Funktionsaufrufe oder Schleifen. Von Entwicklern nach Fertigstellung des Entwicklungssystems identifizierte anwendungsabhängige Muster werden dagegen nicht direkt durch die Modellierung des Entwicklungssystems unterstützt.

5.3.1 Entwicklungsmuster

Um Fehler bei der Entwicklung zu vermeiden, sollten die Muster dem Entwickler so zur Verfügung gestellt werden, daß sie automatisch gemäß der Regeln instanziiert werden [Gamma et al. 95; Pree 95; Coplien, Schmidt 95; Blaha, Premerlani 97]. Ein Entwicklungsmuster besteht aus folgenden Teilen (vgl. Abbildung 5.8):

- Rahmenobjekte, die ein Rahmenwerk (*framework*) für das Muster bilden.
- Benutzerdefinierte Objekte, die das Rahmenwerk füllen.
- Steuerungsobjekte, die angeben, wie das Rahmenwerk mit den benutzerdefinierten Objekten zu füllen ist.
- Ein Generator, der das Rahmenwerk gemäß der Regeln und gesteuert von den Steuerungsobjekten mit den benutzerdefinierten Objekten füllt.

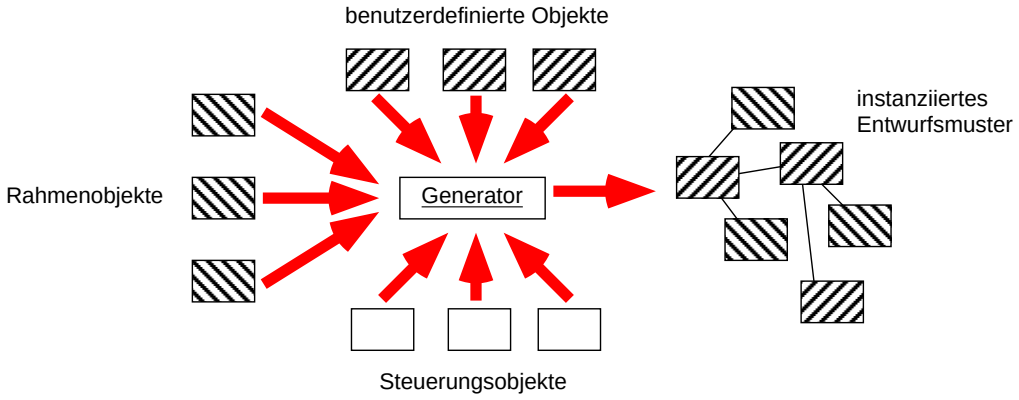

Abbildung 5.8: Entwicklungsmuster

Gemäß dieses Schemas wird beispielsweise eine Transaktion als ein Entwicklungsmuster betrachtet, welches das Zwei-Phasen-Sperrprotokoll [Lockemann, Schmidt 87] implementiert.

- Standardobjekte sind hier die Sperrtabellen eines Datenbanksystems, die von der Transaktion genutzt werden.
- Benutzerdefinierte Objekte sind die zu sperrenden Objekte und der Funktionskörper, welche die Funktionalität der Transaktion implementiert und auf die gesperrten Objekte zugreift.
- Steuerungsobjekt ist beispielsweise eine Auswahl der Strategie, die beim Sperren anzuwenden ist (striktes oder nicht-striktes Zwei-Phasen-Protokoll).
- Der Generator erzeugt eine Funktion, welche die Transaktionseigenschaften erfüllt, indem die von dem Funktionskörper genutzten Objekte gemäß der ausgewählten Strategie gesperrt werden.

Um Entwicklungsmuster des Datenbankbereichs zu unterstützen, wird bei Datenbankprogrammiersprachen [Atkinson, Bunemann 87] die Repräsentation einer Programmiersprache um datenbankspezifische Elemente ergänzt. Der Übersetzer wird so erweitert, daß aus der Beschreibung einer Transaktion eine korrekte Instanziierung mit Implementation des Zwei-Phasen-Sperrprotokolls abgeleitet wird. Beispielsweise wird im DBPL System [Schmidt, Matthes 94; Schröder 91] die Syntax der Sprache Modula-2 [Wirth 88] um eine Repräsentation für Transaktionen erweitert:

```
TRANSACTION t(param1: INTEGER; param2: ARRAY OF CHAR);
BEGIN
...
END t;
```

Durch einen Eingriff in das Entwicklungssystem wird also der Übersetzer so erweitert, daß er eine anwendungsgerechte Repräsentation akzeptiert, die das Entwicklungsmuster enthält. Der Generator wird in den Übersetzer integriert und die Standardobjekte stehen im Entwicklungssystem als Entwicklungsobjekte zur Verfügung. Das Entwicklungssystem wird anschließend neu erzeugt. Alle Möglichkeiten des Übersetzers können ausgenutzt werden, beispielsweise um die Wohlgeformtheit der benutzerdefinierten Objekte zu prüfen und eine anwendungsgerechte Repräsentation des Entwicklungsmusters zu gewährleisten. Dies erfordert detaillierte Kenntnisse über sowie direkten Zugriff auf das Entwicklungssystem und dessen Erzeugung (vgl. Abschnitt 5.2.5).

5.3.2 Dynamische Generierung

Ohne Eingriff in das Entwicklungssystem werden Entwicklungsmuster durch Funktionen, meist höherer Ordnung, implementiert. Die Funktionen erhalten die benutzerdefinierten und Steuerungsobjekte als Parameter. Sie enthalten den Generator und binden die Rahmenobjekte an.

Die Funktionen dienen als Generator. An die Funktion sind die Rahmenobjekte gebunden, während die benutzerdefinierten und Steuerungsobjekte als Parameter an die Funktion übergeben werden. Beispielsweise werden im Tycoon System [Matthes 93] Transaktionen durch Funktionen höherer Ordnung erzeugt [Niederée 92]:

```
let f(i :Int) :Int = ...
let t_f :Fun(i :Int) :Int = transaction.generate1(f ...)
```

Die Funktion *f* wird durch die Generatorfunktion *transaction.generate1* zur Transaktion *t_f*. Die Transaktion *t_f* ist vom Typ her nicht von der Funktion *f* zu unterscheiden, aber sie implementiert zusätzlich zu den Ablaufeigenschaften von *f* eine Transaktionssemantik.

Die generierenden Funktionen werden während des Ablaufs ausgeführt, d.h. Prüfungen, die über Typprüfungen hinausgehen, finden erst während des Ablaufs ab und Fehler werden erst in diesem Moment erkannt. Außerdem stellt in statischen Typsystemen die Einschränkung auf endlich lange, statisch typisierte Parameterlisten bei Funktionen eine Beschränkung der Mächtigkeit dar [Schröder 93].

```
let generate(P1 <:Ok  R <:Ok  f :Fun(:P1) :R) :Fun(:P1) :R = ...
```

Der Generator *generate1* ist spezialisiert auf Funktionen, die einen Parameter erhalten. Funktionen mit anderen Parameterlisten müssen durch andere Generatoren erzeugt werden.

Ein Transaktionsgenerator für Funktionen mit beliebigen Parameterlisten läßt sich durch linguistische Reflektion implementieren [Stemple et al. 92b; Geisler 95]. Der Übersetzer wird reflektiv (vgl. Abschnitt 5.2.5) in das Objektsystem eingebunden. Die Transaktion wird während des Ablaufs in einer anwendungsgerechten oder Zwischenrepräsentation erzeugt, in die (Bindungen an) Objekte in der ablaufgerechten Repräsentation eingebettet sind. Der Übersetzer erzeugt daraus eine Transaktion in einer ablaufgerechten Repräsentation. Das Rahmenwerk besteht in der anwendungsgerechten Repräsentation; die benutzerdefinierten Objekte werden eingebettet; die Steuerungsobjekte wertet die Funktion aus, welche die beiden Repräsentation mischt, während der Übersetzer als Teil des Generators genutzt wird.

Dieser Ansatz erlaubt nur sehr eingeschränkte statische Validierungen. Stattdessen werden die Validierungen durch den (reflektiv eingebundenen) Übersetzer während des Ablaufs vorgenommen. Entwicklungsfehler bleiben somit während der Entwicklung unerkannt und werden erst während des Ablaufs gemeldet.

5.3.3 Statische Generierung

Eine Lösung, die flexibler ist als die Nutzung von Funktionen höherer Ordnung, Entwicklungsfehler nicht erst während des Ablaufs meldet und keine Neuentwicklung des Entwicklungssystems erfordert, sind additive Repräsentationserweiterungen [Kohlbecker et al. 86; Schröder 93; Cardelli et al. 94], kombiniert mit linguistischer Reflektion [Sheard 90; Stemple et al. 92a; Kirby et al. 96] und speziellen Metaobjektprotokollen [Chiba 95; Masuhara et al. 95].

Das Entwicklungssystem bietet reflektive Erweiterungsmöglichkeiten (vgl. Abbildung 5.9), in diesem Fall im Bereich der Erkennung (*parsing*) der anwendungsgerechten Repräsentation. Der Erkennungsalgorithmus kann erweitert bzw. durch einen neuen ersetzt werden, ohne daß das Entwicklungssystem neu erzeugt wird. Der neue Erkennungsalgorithmus erkennt eine anwendungsgerechte Repräsentation, die das Entwicklungsmuster enthält. Aus dieser Repräsentation wird die ablaufgerechte Repräsentation (oder eine Vorstufe davon) erzeugt. D.h. der größte Teil des Entwicklungssystems, insbesondere das Typsystem, bleibt unangetastet, so daß keine Evolution vorliegt, sondern eine Parametrisierung des Übersetzers mit einem anderen Erkennungsalgorithmus. Dabei wird die Phasenstruktur des Übersetzungsvorgangs [Aho et al. 88] ausgenutzt.

Der Erkennungsalgorithmus wird mit Hilfe eines inkrementellen Parsergenerators [Heering et al. 89] erzeugt. Der Parsergenerator verarbeitet eine anwendungsgerechte Repräsentation, in der die Erweiterungen beschrieben werden. Er prüft die Erweiterungen darauf, ob das Entwicklungsmuster eindeutig erkannt werden kann, und erzeugt einen Erkennungsalgorithmus, der das Entwicklungsmuster in eine Repräsentation umsetzt, die von dem Übersetzer weiterverarbeitet wird. Beispielsweise lautet die Beschreibung des Entwicklungsmusters für Transaktionen [Schröder 93]:

```
transactionG :== "transaction" "(" sigs = sigsG ")" body = valueG
=>
fun(sigs)
begin
  let t = transaction.new()
  let result = body
  t.commit()
  result
end
```

- Der erste Teil (bis =>) enthält eine Beschreibung der anwendungsgerechten Repräsentation (Syntax) des Entwicklungsmusters. Beispielsweise ist dies eine Anwendung des Musters:

  ```
  let insertPerson = transaction(p :Person)
  begin
    noOfPersons := noOfPersons+1
    persons.insert(p)
  end
  ```

- Der zweite Teil beschreibt die Umsetzung des Entwicklungsmusters in eine vom Übersetzer verarbeitete Repräsentation. Das Beispiel wird folgendermaßen umgesetzt:

  ```
  fun(p :Person)
  begin
    let t = transaction.new()
    let result =
      begin
  ```

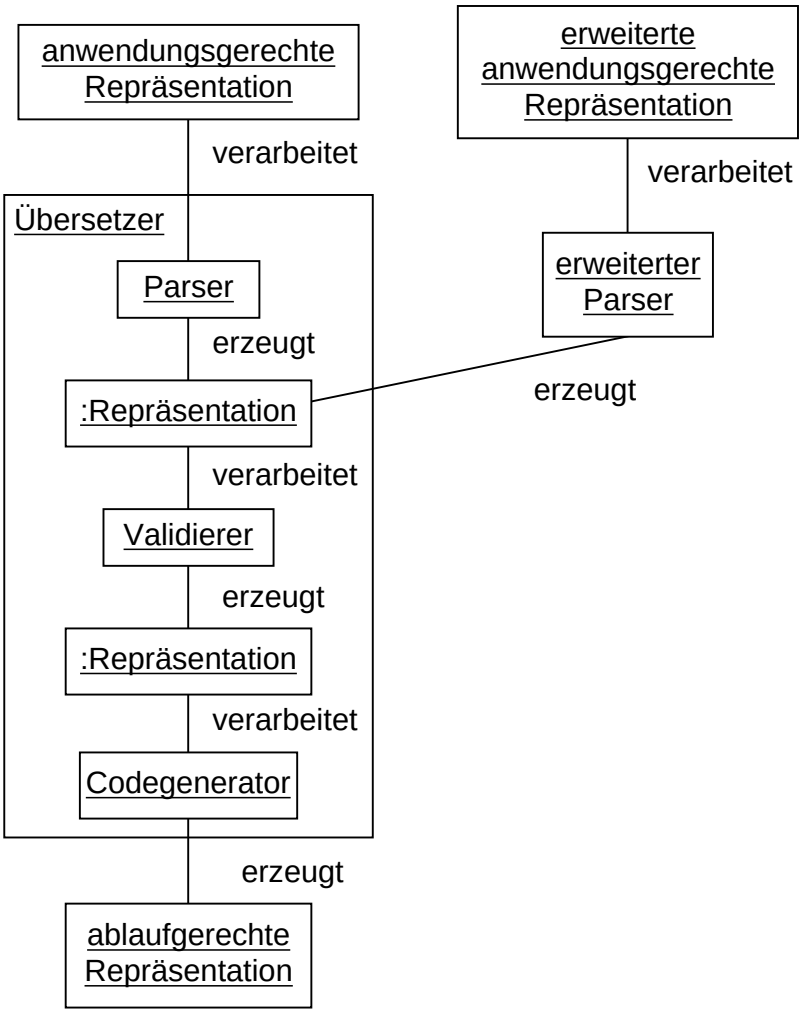

Abbildung 5.9: Syntaxerweiterungen

```
      noOfPersons := noOfPersons+1
      persons.insert(p)
    end
  t.commit()
  result
end
```

In den Erkennungsalgorithmus können reflektiv auch weitere Funktionen eingebunden werden. Sie werden während des Erkennungsvorgangs aufgerufen, um Bedingungen zu prüfen, Teile des Musters zu verarbeiten oder die Ausgabe anzureichern. Beispielsweise können so auch Teile des Entwicklungssystems in den Erkennungsalgorithmus eingebunden und zu Prüfungen herangezogen werden.

Additive Repräsentationserweiterungen unterstützen im Tycoon System Entwicklungsmuster für folgende Bereiche, ohne daß das Entwicklungssystem auf diese Bereiche spezialisiert ist:

- Objektorientierung [Matthes 93]
- Vertragsorientierte Programmierung mit Vor- und Nachbedingungen [Jung 95]

- Nebenläufige Programmierung [Piellusch 96]
- Migration [Mathiske 96]
- Kooperationsstrategien [Mathiske 96]
- Agentensysteme [Johannisson 97]
- Strukturierung komplexer Systeme [Koch 97]

5.3.4 Anwendung auf die Entwicklung kooperierender Objektsysteme

Die statische Generierung von Entwicklungsmustern wird beispielsweise eingesetzt, um das Kooperationsparadigma MA (Mobile Agents, vgl. Kapitel 1) in einer Add-On-Form [Matthes, Schmidt 91b] zu implementieren [Mathiske 96; Redhead 96].

Die gewünschte Syntax für migrierende Agenten lautet folgendermaßen:

```
migrate to gruppenLeiter with remote budgetDB :BudgetDB do
  ueberpruefe(budgetDB abrechnung.summe)
end
```

Der Prozeß soll also zu einem Ort *gruppenLeiter* migrieren und eine nur dort vorhandene lokale Ressource *budgetDB* nutzen, um eine Abrechnungssumme zu überprüfen.

Dieses Entwicklungsmuster wird in folgende Anweisungssequenz umgesetzt:

```
begin
  let budgetDB = gate.migrateTo(:BudgetDB gruppenLeiter)
  ueberpruefe(budgetDB abrechnung.summe)
end
```

Die Funktion *gate.migrateTo* wird also als Standardobjekt genutzt. Sie realisiert die Migration des Agenten.

Das Entwicklungsmuster wird durch folgende Syntaxerweiterung unterstützt:

```
migrateG :== "migrate" "to" destination = valueG
             "with" "remote" localId = ideG ":" type = typeG
             "do" sequence = valuesG "end"
=>
begin
  let localId = gate.migrateTo(:type destination)
  sequence
end
```

Diese Technik kann beliebig ausgebaut werden, beispielsweise um anwendungsgerechte Syntax zum Behandeln immobiler Objekte durch Rebinden [Bharat, Cardelli 95; Mathiske 96; Ramme 98] zu erreichen, ohne das Entwicklungssystem oder das Ablaufsystem zu verändern.

Eine Form von Repräsentationserweiterungen durch statische Generierung wird auch bei der Kooperation von Entwicklungssystemen (vgl. Abschnitt 5.2.4) eingesetzt: Für exportierte Objekte werden aus der anwendungsgerechten Repräsentation der Objekte sowohl die ablaufgerechte Repräsentation als auch weitere Repräsentationen für den Export der Objekte, die Stummel für importierende Objektsysteme usw. generiert [Auerbach et al. 94; OMG 98].

5.4 Zusammenfassung

Dieses Kapitel zeigt, mit welchen Mitteln kooperierende persistente Objektsysteme in heterogenen Umgebungen entwickelt werden, wobei insbesondere der Aspekt der Fehlervermeidung beachtet wird. Dabei sind zwei Zeitpunkte zu unterscheiden:

- Während der Entwicklung wird *statisch* validiert und generiert, d.h. es kann eine statische Validität bestimmter Eigenschaften des Objektsystems garantiert werden.
- Um diese Validität während des Ablaufs zu erhalten, werden *dynamisch* während des Ablaufs weitere Validierungen und Generierungen vorgenommen.

Um die Entwicklungstechniken der Validerung und Generierung zu beiden Zeitpunkten einsetzen zu können, werden in dieser Arbeit die dynamische Validierung durch Einbindung von Entwicklungsobjekten in das Objektsystem und die statische Generierung durch Erweiterung des Entwicklungssystems um Entwicklungsmuster eingeführt. Dies wird möglich durch das in dieser Arbeit entwickelte einheitliche Objektsystemmodell, das die Beschreibung der Kooperation zwischen Objektsystem und Entwicklungssystem erlaubt. Dabei wird auch die Weiterentwicklung von Objektsystemen, die Evolution (vgl. Abschnitt 3.3), einbezogen.

Der Entwicklungsprozeß kooperierender Objektsysteme stellt sich somit folgendermaßen dar:

Die **Modellierung** bildet die Basis der Entwicklung, aus der ein ablauffähiges Objektsystem entsteht. Die Objektsysteme werden in einer anwendungsgerechten Repräsentation modelliert. Ein Übersetzer erzeugt aus der anwendungsgerechten Repräsentation eine ablaufgerechte Repräsentation der Objektsysteme. Die Freiheitsgrade und Einschränkungen bei dem Entwurf der anwendungs- und ablaufgerechten Repräsentation werden aufgezeigt.

Die **Validierung** trägt dazu bei, fehlerhafte Abläufe von Objektsystemen früh, sowohl statisch zur Entwicklungszeit als auch dynamisch zur Ablaufzeit, zu erkennen, indem beschreibende Informationen (insbesondere Typinformationen) angelegt, aufbewahrt und von Validierern (insbesondere Typprüfern) ausgewertet werden.

- Um zum Zeitpunkt der Entwicklung die Abwesenheit bestimmter Fehler zu garantieren, verfügt das Entwicklungssystem über ein statisches Typsystem; statische Eigenschaften der Objektsysteme werden in dem Typsystem beschrieben und validiert.
- Um Prüfungen während des Ablaufs des Objektsystems, insbesondere bei Kooperation mit anderen Objektsystemen, zu erlauben, erzeugt das Entwicklungssystem dynamische Typen und integriert sie zusammen mit Prüfalgorithmen des Entwicklungssystems über linguistische Reflektion in das Objektsystem.
- Um zwischen kooperierenden Objektsystemen ein gemeinsames Verständnis über die ausgetauschten Objekte aufzubauen und somit Mißverständnisse zu vermeiden, verfügen die Entwicklungsumgebungen der Objektsysteme über ein gemeinsames Typsystem.
- Um Evolution in kooperierenden Objektsystemen zu unterstützen, sind die Typen des Typsystems versioniert.

Durch **Generierung** wird die Modellierung um anwendungsgerechte Entwicklungsmuster ergänzt, wobei der Generator die Fehlermöglichkeiten einschränkt und somit insbesondere bei der statischen Generierung zu einer höheren Validität des Objektsystems beiträgt.

Entwicklungsmuster unterstützen die Anpassung eines Entwicklungssystems an unterschiedliche Anwendungsbereiche. Sie halten Invarianten ein und tragen somit zur Validierung des Objektsystems bei. Eine Implementierung über additive Repräsentationserweiterungen stellt einen Ausgleich dar zwischen der invasiven Erweiterung des Entwicklungssystems und der eingeschränkten Nutzung von dynamischen Generatorfunktionen.

Kapitel 6

Ergebnisse

In diesem Kapitel werden die einzelnen Beiträge der Arbeit bewertend zusammengefaßt (vgl. Abschnitt 6.1) und ein Ausblick auf darauf aufbauende Forschungsarbeiten gegeben (vgl. Abschnitt 6.2).

6.1 Zusammenfassung

Diese Arbeit stellt einen einheitlichen Rahmen zur Beschreibung von kooperierenden Objektsystemen auf, leitet die beim Betrieb und bei der Entwicklung kooperierender Objektsysteme auftretenden Aufgaben systematisch aus diesem Rahmen ab und entwickelt zentrale Lösungsbeiträge für diese Aufgaben. Hintergrund des Rahmens bilden Zeit und Raum, in denen Objekte sich bewegen und gebunden werden:

Zeit: Objektsysteme werden entwickelt (mit Hilfe von Objektsystemen, hier Entwicklungssysteme genannt), potentiell unendlich lange (persistent) betrieben (mit Hilfe von Objektsystemen, hier Ablaufsysteme genannt) und während des Betriebs durch Evolution weiterentwickelt. Zentrale Aufgaben sind die Erzeugung, Verarbeitung, Speicherung und Interpretation von Objektrepräsentationen sowie die Herstellung und Auflösung von Bindungen.

Raum: Objektsysteme kooperieren miteinander, indem sie Objekte anderer Objektsysteme binden und Objekte zwischen den Objektsystemen übertragen; insbesondere kooperieren Objektsysteme reflektiv mit ihren Entwicklungs- und Ablaufsystemen. Zentrale Aufgaben sind die Übertragung und Umwandlung von Objektrepräsentationen sowie die Erhaltung und Wiederherstellung von Bindungen.

Objektrepräsentationen und -bindungen spielen also eine zentrale Rolle in Zeit und Raum. Durch den in dieser Arbeit entwickelten Rahmen gelingt eine konzise und einheitliche Beschreibung der Phänomene, die bei zeitlich und räumlich kooperierenden Objektsystemen auftreten.

Der einheitliche Rahmen wird durch das Objektsystemmodell gebildet. Das Modell beschränkt sich auf die für den Betrieb und die Entwicklung von Objektsystemen essentiellen Aspekte. Insbesondere die Verhaltensaspekte von Objekten bestimmen wesentlich die für Persistenz, Evolution und Kooperation in heterogenen Umgebungen zu betrachtenden Fälle. Daher werden die Objekte entlang ihres Verhaltensaspekts in passiv, aktivierbar und aktiv (Daten, Code und Prozesse) unterschieden.

Bindungen zwischen Objekten werden explizit modelliert und ermöglichen damit eine detaillierte Betrachtung von Persistenzsicherung (rekursive Bindungen, Bindung eines Objektes in mehreren Bindungen), Evolution (Ersetzen einer oder mehrerer Bindungen an ein Objekt) und Kooperation (Bindungen zwischen Objektsystemen, Bindungen von zwischen Objektsystemen übertragenen Objekten).

Ablauf- und Entwicklungsumgebungen bestehen aus Objektsystemen (Ablauf- und Entwicklungssystemen) und Objekten. Sie werden daher mit den gleichen Begriffen wie Objektsysteme

modelliert. Durch diese Modellierung werden Spezialfälle, wie die Bindung zwischen Objekten des Objektsystems und des Ablaufsystems, auf einen allgemeinen Fall, die Bindung zwischen (kooperierenden) Objektsystemen, zurückgeführt, so daß die im allgemeinen Fall gewonnenen Erkenntnisse auf den Spezialfall übertragen werden können, beispielsweise zur Erneuerung von Bindungen in Zeit und Raum.

Die Ablauffähigkeit von Objektsystemen wird wesentlich durch die Belegung und Freigabe von Ressourcen bestimmt. Es zeigt sich, daß ohne den Einsatz reflektiver Techniken die Ablauffähigkeit des Objektsystems eingeschränkt wird. Reflektive Techniken gelangen auch bei der Persistenzsicherung, Evolution und Kooperation zum Einsatz, stellen also eine grundlegende Eigenschaft von Ablaufumgebungen für Objektsysteme dar. Reflektive Techniken werden als eine spezielle Art der Kooperation, hier zwischen Ablauf- und Objektsystem, identifiziert und sind somit implizit bei der Kooperation von Objektsystemen mit erfaßt. Die über kooperierende Objektsysteme gewonnenen Erkenntnisse sind somit ebenfalls direkt auf die Reflektion anwendbar.

Aufgrund der Forderung nach Langlebigkeit der Objekte werden Objektsysteme um die Eigenschaft der Persistenz orthogonal erweitert. Persistenz zieht Evolution als Technik zur Anpassung an geänderte Anforderungen oder Umgebungen nach sich. Evolution erfordert eine differenzierte Auseinandersetzung mit der Integrität eines Objektsystems während des Ablaufs. Abhängig vom Ablaufzustand und den betrachteten Objekten gelangen unterschiedliche Evolutionsmechanismen unter Beachtung der Ablauffähigkeit zum Einsatz.

Autonome Objektsysteme kooperieren in heterogenen Umgebungen, indem sie Objekte austauschen, Objekte des anderen Objektsystems binden und auf diesen gebundenen Objekten Operationen auslösen. Zur Übertragung von Objekten sind Repräsentationen umzuwandeln und Bindungen zu behandeln, beispielsweise durch eine erneute Bindung von Objekten nach der Übertragung. Diese Wiederherstellung einer Bindung wird insbesondere durch die Heterogenität und Evolution der kooperierenden Objektsysteme erschwert und erfordert daher einen adaptiven Mechanismus, der die Autonomieanforderungen der Kooperationspartner wahrt. Der in dieser Arbeit vorgestellte reflektive Mechanismus zur Wiederherstellung von Bindungen über eine bindende Funktion bietet die geforderte Flexibilität.

Entfernte Bindungen schränken die Autonomie der Kooperationspartner ein und stellen eine Gefährdung der Ablauffähigkeit der beteiligten Objektsysteme dar. Daher nimmt in dieser Arbeit die Abstimmung der Kooperationsstrategien der beteiligten Kooperationspartner breiten Raum ein. Als Spezialfall entfernter Bindungen wird die Bindung zwischen Objekten des Ablaufsystems und des Objektsystems identifiziert, die als reflektive Technik in vielen Bereichen Anwendung findet. Die Wechselwirkungen zwischen Persistenz und entfernten Bindungen führen zu erweiterten Anforderungen an die Kooperationsstrategien.

Als zentrale Anforderung der Entwicklung von kooperierenden Objektsystemen werden, aufbauend auf der Modellierung, die dynamische Validierung sowie die statische Generierung diskutiert. Die dynamische Validierung ist unerläßlich zur Sicherung der Ablauffähigkeit kooperierender Objektsysteme und schränkt wiederum die Autonomie der Objektsysteme ein, insbesondere in Bezug auf die Evolution der Entwicklungssysteme. Die statische Generierung unterstützt Kooperationsmuster und sichert somit die Validität der entwickelten Objektsysteme. Sie dient damit als erweiterbares Abstraktionsmittel oberhalb der Basisabstraktionen einer Programmiersprache als anwendungsnaher Repräsentation. Die (reflektive) Übernahme von Objekten des Entwicklungssystems in das entwickelte Objektsystem wird durch das einheitliche Rahmenmodell erfaßt und stellt wiederum einen Spezialfall der Kooperation zwischen Objektsystemen dar.

Zusammenfassend liegen die Beiträge dieser Arbeit in

- der Aufstellung eines einheitlichen Rahmenmodells zur Diskussion des Betriebs und der Entwicklung kooperierender Objektsysteme;
- der Diskussion der Evolution persistenter Objektsysteme unter Sicherung der Ablauffähigkeit;
- der Entwicklung von Kooperationsstrategien für entfernte Bindungen und zur Wiederherstellung von Bindungen bei der Übertragung von Objekten während der Kooperation zwischen

autonomen Objektsystemen in heterogenen Umgebungen sowie

- der Weiterentwicklung von Techniken zur dynamische Validierung von Kooperationen sowie zur statischen Generierung von Kooperationsmustern bei der Entwicklung von kooperierenden Objektsystemen.

Es zeigt sich, daß ein einheitlicher Rahmen zur Beschreibung von Objektsystemen (inklusive Ablauf- und Entwicklungssystemen) in Zeit und Raum sowie die Klärung der bei der Evolution von Objektsystemen und der Kooperation zwischen Objektsystemen zu lösenden Aufgaben essentielle Grundlagen sowohl für den Betrieb als auch für die Entwicklung von Objektsystemen darstellen.

6.2 Ausblick

In dieser Arbeit werden die bei Betrieb und Entwicklung kooperierender Objektsysteme auftretenden Aufgaben und Lösungsansätze entlang eines umfassenden Objektsystemmodells beschrieben. Dieses Modell abstrahiert weitgehend von einer Implementierung. Kommerzielle Kooperationsstandards wie CORBA [OMG 98] oder Ablauf- und Entwicklungssysteme für kooperierende Systeme wie Java [Java Team et al. 96; Lindholm, Yellin 96] stehen zu diesem Objektsystemmodell und weiteren Schwerpunkten dieser Arbeit in enger Beziehung; im folgenden werden diese Beziehungen konkretisiert, zukunftsweisende Entwicklungen aufgezeigt und evaluiert:

- Die vorhandenen Systeme bieten noch kein vom Entwicklungsparadigma (bei CORBA und Java dem objektorientierten Paradigma) abstrahierendes Objektsystemmodell. Daher sind Code und Prozesse bei ihnen nicht als Objekte erster Klasse vorhanden. Somit werden die in Abschnitt 1.1 eingeführten Kooperationsparadigmen, insbesondere das Paradigma MA, nicht voll unterstützt. Die Kooperation über unterschiedliche Paradigmen hinweg wird schwierig bis unmöglich. Es fehlt eine Verallgemeinerung im Stile von CORBA ohne die Festlegung auf ein Paradigma, so wie die paradigmenneutrale Modellierung in [Matthes 93].

- Reflektion ist als Basismechanismus für den Betrieb von kooperierenden Objektsystemen zur Verfügung zu stellen und konsequent in allen Bereichen der Systeme zu nutzen, also zur Unterstützung der Ressourcenverwaltung, Persistenzsicherung, Evolution und Kooperation.

- Die Evolution von Objektsystemen ist noch nicht als elementarer Bestandteil des Ablaufs von Objektsystemen erkannt und gewürdigt. Standardisierte Versionierungen sind nötig, um die Kooperation autonomer und unabhängig (weiter-)entwicklter Objektsysteme zu unterstützen. Die Evolution muß außerdem schon bei der Entwicklung eines Objektsystems einbezogen werden, wie in Abschnitt 3.3 gezeigt. Entsprechende Evolutionsmuster sind wie die in Abschnitt 5.3 gezeigten Kooperationsmuster zu entwerfen und reflektiv zu implementieren.

- Bei der Übertragung von Objekten zwischen Objektsystemen sollten weitere Freiheitsgrade bei der Repräsentationsumwandlung entlang der in Abschnitt 4.2 gezeigten Linien verfolgt werden. Außerdem ist die Erneuerung von Bindungen bei der Übertragung von Objekten in der in dieser Arbeit gezeigten Allgemeinheit noch nicht in gängigen Systemen zu finden. Java bietet mit dem Klassenlader einen Schritt in dieser Richtung. Eine Schnittstelle, die den unterschiedlichen Anforderungen nach Sicherheit und Flexibilität autonomer kooperierender Objektsysteme gerecht wird, ist in die Kooperationsstandards zu integrieren.

- Die Behandlung entfernter Bindungen zwischen Objektsystemen ist weiter auszubauen, wie in Abschnitt 4.3 gezeigt. Dabei sind sowohl die Gemeinsamkeiten unterschiedlicher Arten von entfernten Bindungen, beispielsweise in andere Objektsysteme oder in das Ablaufsystem, auszunutzen wie auch Flexibilitätsgrade in Form unterschiedlicher, mit den Anforderungen der autonomen Objektsysteme skalierender Kooperationsstrategien einzubeziehen.

- Kooperationsstandards müssen die dynamische Validierung einbeziehen. Dabei ist die Evolution der Entwicklungssysteme zu beachten.
- Entwicklungssysteme sollten die statische Generierung von Entwicklungsmustern unterstützen. Sowohl für die Kooperation als auch für die Evolution sind Muster zu finden und den Entwicklern in geeigneter Form zur Verfügung zu stellen.

In dem Maße, in dem die breite Basis des in dieser Arbeit entwickelten und systematisch ausgefüllten Rahmens modellmäßig, methodisch und technisch auch kommerziell unterstützt und in die Standardisierung einbezogen wird, werden der Betrieb und die Entwicklung kooperierender Objektsysteme in heterogenen Umgebungen den gleichen Stand erreichen wie heute der Betrieb und die Entwicklung von (nicht-kooperierenden) Objektsystemen.

Literaturverzeichnis

Abadi et al. 89: Abadi, Martín, Cardelli, Luca, Pierce, Benjamin, und Plotkin, Gordon. *Dynamic Typing in a Statically-Typed Language.* In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL '89).* ACM Press, 1989, S. 213–227.

Achauer 93: Achauer, Bruno. *The DOWL Distributed Object-Oriented Language.* Communications of the ACM, Jg. 36, September 1993, Nr. 9, S. 48–55.

Adl-Tabatabai et al. 96: Adl-Tabatabai, Ali-Reza, Langdale, Geoff, Lucco, Steven, und Wahbe, Robert. *Efficient and Language-Independent Mobile Programs.* In: *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation '96.* ACM Press, 1996, S. 127–136.

Aho et al. 88: Aho, A. V., Sethi, R., und Ullman, J. D. *Compilerbau.* Addison-Wesley Publishing Company, 1988.

Ananda, Srinivasan 91: Ananda, A.L. und Srinivasan, B. (Hrsg.). *Distributed Computing Systems: Concepts and Structure.* IEEE Society Press, 1991.

Atkinson et al. 93: Atkinson, M.P., Sjøberg, D.I.K., und Morrison, R. *Managing Change in Persistent Object Systems.* In: Nishio und Yonezawa [93], S. 315–338.

Atkinson et al. 95: Atkinson, M., Maier, D., und Benzaken, V. (Hrsg.). *Persistent Object Systems, Tarascon 1994.* Springer Verlag, 1995.

Atkinson, Bunemann 87: Atkinson, M. P. und Bunemann, P. *Types and Persistence in Database Programming Languages.* ACM Computing Surveys, Jg. 19, Juni 1987, Nr. 2, S. 105–190.

Auerbach et al. 94: Auerbach, J.S., Gopal, A.S., Kennedy, M.T., und Russell, J.R. *Concert/C: Supporting Distributed Programming with Language Extensions and a Portable Multiprotocol Routine.* In: *Proceedings of the 14th IEEE International Conference on Distributed Computer Systems ICDS '94.* IEEE Society Press, 1994, S. 152–159.

Baker 97: Baker, Sean. *CORBA Distributed Objects Using ORBIX.* Addison-Wesley Publishing Company, 1997.

Baldi et al. 97: Baldi, M., Gai, S., und Picco, G. *Exploting Code Mobility in Decentralized and Flexible Network Management.* In: Rothermel und Popescu-Zeletin [97], S. 13–26.

Balzert 82: Balzert, H. *Die Entwicklung von Software-Systemen.* Bibliographisches Institut, 1982.

Bennett 90: Bennett, John K. *Experience with Distributed Smalltalk.* IEEE Software Practice and Experience, Jg. 20, Februar 1990, Nr. 2, S. 157–180.

Bernstein 98: Bernstein, Philip A. *Repositories and Object Oriented Databases.* SIGMOD Record, Jg. 27, Maerz 1998, Nr. 1, S. 88–96.

Bharat, Cardelli 95: Bharat, Krishna A. und Cardelli, Luca. *Migratory Applications.* In: *Proceedings of te ACM Symposium on User Interface Software and Technology '95*, 1995, S. 133–142.

Birrell et al. 93: Birrell, Andrew, Evers, David, Nelson, Greg, Orwicki, Susan, und Wobber, Edward. *Distributed Garbage Collection for Network Objects.* Technical report, Digital Equipment Corporation Systems Research Center, Dezember 1993.

Birrell et al. 95: Birrell, Andrew, Nelson, Greg, und Owicki, Susan. *Network Objects.* IEEE Software Practice and Experience, Jg. 25, Dezember 1995, Nr. S4, S. 87–130.

Birrell, Nelson 84: Birrell, Andrew D. und Nelson, Bruce Jay. *Implementing Remote Proceduce Calls.* ACM Transactions on Computer Systems, Jg. 2, Februar 1984, Nr. 1, S. 39–59.

Blaaw, Brooks, Jr. 97: Blaaw, G.A. und Brooks, Jr., F.P. *Computer Architecture: Concepts and Evolution.* Addison-Wesley Publishing Company, 1997.

Black, Artsy 89: Black, Andrew P. und Artsy, Yeshayahu. *Implementing Location Independent Invocation.* In: *Proceedings of the 9th IEEE International Conference on Distributed Computer Systems ICDS '89.* IEEE Society Press, 1989, S. 550–559.

Black et al. 87: Black, Andrew, Hutchinson, Norman, Jul, Eric, Levy, Henry, und Carter, Larry. *Distribution and Abstract Types in Emerald.* IEEE Transactions on Software Engineering, Jg. 13, 1987, Nr. 1, S. 65–76.

Black, Walpole 95: Black, Andrew P. und Walpole, Jonathan. *Object to the rescue! or: httpd: the next generation operating system.* ACM SIGOPS Operating Systems Review, Jg. 29, 1995, Nr. 1, S. 91–95.

Blaha, Premerlani 97: Blaha, Michael und Premerlani, William. *Object-Oriented Modeling and Design for Database Applications.* Prentice Hall International, 1997.

Bolinger, Bronson 95: Bolinger, Don und Bronson, Tan. *Applying RCS and SCCS.* O'Reilly & and Associates, 1995.

Bösch 97: Bösch, Andreas. *Das WWW als Informationssystem: Konzepte, Dienste, Defizite, Entwicklungen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, September 1997.

Bradshaw 96: Bradshaw, Jeffrey (Hrsg.). *Software Agents.* MIT Press, 1996.

Briot et al. 96: Briot, J.-P., Geib, J.-M., und Yonezawa, A. (Hrsg.). *Object-Based Parallel and Distributed Computation '95.* Springer Verlag, 1996.

Brown, Najork 96: Brown, Marc H. und Najork, Marc A. *Distributed Active Objects.* In: *Proceeding of the 5th International WWW Conference*, 1996.

Burger 97: Burger, Cora. *Groupware — Kooperationsunterstützung für verteilte Anwendungen.* dpunkt Verlag, 1997.

Caglayan, Harrison 97: Caglayan, Alper und Harrison, Colin. *Agent Sourcebook: A Complete Guide to Desktop, Internet and Intranet Agents.* John Wiley & Sons, 1997.

Callsen, Agha 94: Callsen, Christian J. und Agha, Gul. *Open Heterogeneous Computing in ActorSpace.* Journal of Parallel and Distributed Computing, Jg. 21, 1994, S. 289–300.

Cardelli et al. 94: Cardelli, Luca, Matthes, Florian, und Abadi, Martín. *Extensible Grammars for Language Specialization.* In: Beeri, C., Ohori, A., und Shasha, D.E. (Hrsg.). *Database Programming Languages (DBPL-4).* Springer Verlag, 1994.

Cardelli, Gordon 98: Cardelli, Luca und Gordon, Andrew D. *Mobile Ambients.* In: *ETAPS '98*, 1998.

Cardelli, Wegner 85: Cardelli, L. und Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* In: *Computing Surveys, Jahrg. 17, Nr. 4, New York*, 1985, S. 471–522.

Cardelli 95: Cardelli, Luca. *A Language with Distributed Scope.* In: *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 1995, S. 286–297.

Cardelli 97a: Cardelli, Luca. *Global Computation.* ACM SIGPLAN Notices, Jg. 32, Januar 1997, Nr. 1, S. 66–68.

Cardelli 97b: Cardelli, Luca. *Type Systems.* In: Tucker, Allen B. (Hrsg.). *The Computer Science and Engineering Handbook.* CRC Press, 1997, S. 2208–2236.

Carzaniga et al. 97: Carzaniga, Antonio, Picco, Gian Pietro, und Vigna, Giovanni. *Designing Distributed Applications with Mobile Code Paradigms.* In: *Proceedings of the 1997 International Conference on Software Engineering (ICSE '97).* IEEE/ACM Press, 1997, S. 22–32.

Cejtin et al. 95: Cejtin, Henry, Jagannathan, Suresh, und Kelsey, Richard. *Higher-Order Distributed Objects.* ACM Transactions on Programming Languages and Systems, Jg. 17, September 1995, Nr. 5, S. 704–739.

Cheatham 66: Cheatham, T. E., Jr. *The Introduction of Definitional Facilities into Higher Level Programming Languages.* In: *AFIPS Proceedings of the Joint Computer Conference*, Bd. 29, 1966, S. 623–637.

Cheung, Loong 95: Cheung, W.H. und Loong, Anthony H.S. *Exploring Issues of Operating Systems Structuring: From Microkernel to Extensible Systems.* ACM SIGOPS Operating Systems Review, Jg. 28, 1995, Nr. 4, S. 4–16.

Chiba 95: Chiba, Shigeru. *A Metaobject Protocol for C++.* In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '95)*, 1995, S. 285–299.

Ciancarini et al. 95: Ciancarini, P., Nierstrasz, O., und Yonezawa, A. (Hrsg.). *Object-Bases Models and Languages for Concurrent Systems.* Springer Verlag, 1995.

Ciancarini, Hankin 96: Ciancarini, P. und Hankin, C. (Hrsg.). *Coordination Languages and Models.* Springer Verlag, 1996.

Cohrs et al. 88: Cohrs, David L., Miller, Barton P., und Call, Lisa A. *Distributed Upcalls: A Mechanism for Layering Asynchronous Abstractions.* In: *Proceedings of the 8th IEEE International Conference on Distributed Computer Systems ICDS '88.* IEEE Society Press, 1988, S. 55–62.

Coplien, Schmidt 95: Coplien, J. O. und Schmidt, D. C. (Hrsg.). *Pattern Languages of Program Design (PLoP).* Addison-Wesley Publishing Company, 1995.

Curtis 97: Curtis, David. *Java, RMI and CORBA.* White paper, Object Management Group, 1997.

Date, Darwen 97: Date, C. und Darwen, H. *A Guide to the SQL Standard.* Addison-Wesley Publishing Company, 4 edition, 1997. SQL-3?!? nicht gelesen!

Day et al. 93: Day, Mark, Liskov, Barbara, Maheshwari, Umesh, und Myers, Andrew C. *References to Remote Objects in Thor.* ACM Letters on Programming Languages and Systems, Jg. 2, Maerz 1993, Nr. 1–4, S. 115–126.

de Bruijn 72: de Bruijn, N. G. *Lambda-calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem.* Indag. Math., Jg. 34, 1972, Nr. 5, S. 381–392.

Diehl 98: Diehl, Stephan. *A Formal Introduction to the Compilation of Java.* IEEE Software Practice and Experience, Jg. 28, Maerz 1998, Nr. 3, S. 297–328.

Doorn et al. 95: Doorn, Leendert van, Abadi, Martin, Burrows, Mike, und Wobber, Edward. *Secure Network Objects.* Technical report, Vrije Universiteit Amsterdam, 1995.

Dunn 93: Dunn, Robert H. *Softwarequalität.* Hanser Verlag, 1993.

Enslow 78: Enslow, Philip. H., Jr. *What is a "Distributed" Data Processing System.* IEEE Computer, Jg. 11, Januar 1978, Nr. 1, S. 13–21.

Feldmann 79: Feldmann, Stuart I. *Make – A Program for Maintaining Computer Programs.* IEEE Software Practice and Experience, Jg. 9, 1979, S. 255–265.

Feldmann 88: Feldmann, Stuart I. *Evolution of MAKE.* In: Winkler, Jürgen F. H. (Hrsg.). *International Workshop on Software Version and Configuration Control*, Bd. 30, 1988, S. 413–416.

Ferber 88: Ferber, Jacques. *Computational Reflection in Class-Based Object-Oriented Languages.* In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '89).* ACM Press, 1988, S. 317–326.

Finkelstein et al. 94: Finkelstein, David, Acton, Donald, Coatta, Terry, Hutchinson, Norm, und Neufeld, Gerald. *Object Properties in the Raven System.* In: *Proceedings of the 14th IEEE International Conference on Distributed Computer Systems ICDS '94.* IEEE Society Press, 1994, S. 502–509.

Foote, Johnson 89: Foote, Brian und Johnson, Ralph E. *Reflective Capabilities in Smalltalk-80.* In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '89).* ACM Press, 1989, S. 327–335.

Fowler, Scott 97: Fowler, Martin und Scott, Kendall. *UML Distilled.* Addison-Wesley Publishing Company, 1997.

Fukuda et al. 96: Fukuda, Munehiro, Bic, Lubomir F., Dillencourt, Michael B., und Merchant, Fehmina. *Intra- and Inter-Object Cooordination with MESSENGERS.* In: Ciancarini und Hankin [96], S. 179–196.

Fukuda et al. 97: Fukuda, Munehiro, Bic, Lubomir F., Dillencourt, Michael B., und Merchant, Fehmina. *A Hierarchical Mapping Scheme for Mobile Agent Systems.* In: *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing.* IEEE Society Press, 1997, S. 66–71.

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

Gawecki, Matthes 96a: Gawecki, A. und Matthes, F. *Exploiting Persistent Intermediate Code Representations in Open Database Environments.* In: *Proceedings of the 5th Conference on Extending Database Technology, EDBT'96*, Avignon, France, Maerz 1996, S. 403–423. Springer Verlag.

Gawecki, Matthes 96b: Gawecki, A. und Matthes, F. *Integrating Subtyping, Matching and Type Quantification: A Practical Perspective.* In: *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, Linz, Austria, Juli 1996, S. 26–47. Springer Verlag.

Geib et al. 96: Geib, Jean-Marc, Gransart, Christophe, Grenot, Chrystel, und Merle, Philippe. *(Thread and Object)-Oriented Distributed Programming.* In: Briot et al. [96], S. 83–103.

Geisler 95: Geisler, Andreas. *Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Juni 1995.

Ghezzi, Vigna 97: Ghezzi, Carlo und Vigna, Giovanni. *Mobile Code Paradigms and Technologies: A Case Study.* In: Rothermel und Popescu-Zeletin [97], S. 39–49.

Goldberg, Robson 85: Goldberg, Adele und Robson, David. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1985.

Gosling 95: Gosling, James. *Java Intermediate Bytecodes.* In: *ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, 1995, S. 111–118.

Gutknecht 94: Gutknecht, J. (Hrsg.). *Programming Languages and System Architectures.* Springer Verlag, 1994.

Habert et al. 90: Habert, Sabine, Mosseri, Laurence, und Abrossimov, Vadim. *COOL: Kernel Support for Object-Oriented Environments.* In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '90) and of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '90)*, 1990, S. 269–277.

Harel 87: Harel, D. *Statecharts: A Visual Formalism for Complex Systems.* In: *Science of Computer Programming, Nr. 8, Amsterdam*, 1987, S. 231–274.

Heering et al. 89: Heering, J., Kint, P., und Rekers, J. *Incremental Generation of Parsers.* ACM SIGPLAN Notices, Jg. 24, Juli 1989, Nr. 7, S. 179–191.

Herlihy, Liskov 82: Herlihy, M. und Liskov, B. *A Value Transmission Method for Abstract Data Types.* ACM Transactions on Programming Languages and Systems, Jg. 4, Oktober 1982, Nr. 4, S. 527–551.

Hofmeister, Purtilo 93: Hofmeister, Christine und Purtilo, James. *Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement.* In: *Proceedings of the 13th IEEE International Conference on Distributed Computer Systems ICDS '93.* IEEE Society Press, 1993, S. 101–110.

Holder, Ben-Shaul 97: Holder, Ophir und Ben-Shaul, Israel. *A Reflective Model for Mobile Software Objects.* In: *Proceedings of the 17th IEEE International Conference on Distributed Computer Systems ICDS '97.* IEEE Society Press, 1997, S. 339–346.

Horn, Cahill 91: Horn, Chris und Cahill, Vinny. *Supporting Distributed Applications in the Amadeus Environment.* computer communications, Jg. 14, Juli 1991, Nr. 6, S. 358–365.

Hsieh et al. 93: Hsieh, Wilson C., Wang, Paul, und Weihl, William E. *Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems.* In: *4th ACM Conference on Principles and Practice of Parallel Programming.* ACM Press, 1993, S. 239–248.

Huang, Lamb 94: Huang, Lin und Lamb, David Alex. *Communicating Abstract Data Type Values in Heterogeneous Distributed Programs.* In: *Proceedings of the 14th IEEE International Conference on Distributed Computer Systems ICDS '94.* IEEE Society Press, 1994, S. 458–465.

Jacquemot et al. 96: Jacquemot, Christian, Strarup, Peter, und Carrez, Stephane. *CHORUS/COOL: CHORUS Object Oriented Technology.* In: Briot et al. [96], S. 187–204.

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

Jazayeri, Schauer 97: Jazayeri, Mehdi und Schauer, Helmut (Hrsg.). *Software Engineering — ESEC/FSE '97.* Springer Verlag, 1997.

Jessen, Valk 87: Jessen, Eike und Valk, Rüdiger. *Rechensysteme - Grundlagen der Modellbildung.* Springer, 1987.

Johannisson 97: Johannisson, Nico. *Eine Umgebung für mobile Agenten: Agentenbasierte verteilte Datenbanken am Beispiel der Kopplung autonomer „Internet Web Site Profiler“.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, April 1997.

Jul et al. 88: Jul, Eric, Levy, Henry, Hutchinson, Norman, und Black, Andrew. *Fine-Grained Mobility in the Emerald System.* ACM Transactions on Computer Systems, Jg. 6, Februar 1988, Nr. 1, S. 109–133.

Jung 95: Jung, Tanja. *Softwarebibliotheken: Entwurf und Dokumentation am Beispiel von Tycoon.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, November 1995.

Kaplan, Wileden 95a: Kaplan, Alan und Wileden, Jack C. *Conch: Experimenting with Enhanced Name Management for Persistent Object Systems.* In: Atkinson et al. [95], S. 318–331.

Kaplan, Wileden 95b: Kaplan, Alan und Wileden, Jack C. *Formalization and Application of a Unifying Model for Name Management.* In: *Proceedings of SIGSOFT '95.* ACM Press, 1995, S. 161–172.

Kernighan, Ritchie 90: Kernighan, B. W. und Ritchie, D. M. *Programmieren in C.* Hanser, 2. edition, 1990.

Kirby et al. 96: Kirby, G.N.C., Connor, R.C.H., Morrison, R., und Stemple, D. *Using Reflection to Support Type-Safe Evolution in Persistent Systems.* Technical report, University of St Andrews, 1996.

Kiszales et al. 91: Kiszales, G., Riviéres, J. des, und Bobrow, D. G. *The Art of the Metaobject Protocol.* MIT Press, 1991.

Koch 96: Koch, Christian. *Generisches und typsicheres Modulmanagement.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, Maerz 1996.

Koch 97: Koch, Christian. *Entwicklungsunterstützung für persistente strukturierte Systeme.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Juni 1997.

Kohlbecker et al. 86: Kohlbecker, E. E., Friedman, D. P., Felleisen, M., und Duba, B. *Hygienic Macro Expansion.* In: *Conference on Lisp and Functional Programming*, 1986, S. 151–159.

Kono et al. 94: Kono, Kenji, Kato, Kazuhiko, und Masuda, Takashi. *Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers.* In: *Proceedings of the 14th IEEE International Conference on Distributed Computer Systems ICDS '94.* IEEE Society Press, 1994, S. 142–151.

Kornacker 95: Kornacker, Marcel. *Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, August 1995.

Kueffmann 94: Kueffmann, Karin. *Software-Wiederverwendung.* Vieweg Verlag, 1994.

Lamb 96: Lamb, D.A. (Hrsg.). *Studies of Software Design. ICSE '93 Workshop.* Springer Verlag, 1996.

Lampson 94: Lampson, Butler W. *Interconnecting Computers: Architecture, Technology and Economics.* In: Gutknecht [94], S. 1–20.

Lauer, Satterthwaite 79: Lauer, Hugh C. und Satterthwaite, Edwin H. *The Impact of Mesa on System design.* In: *Proceedings of the 4th International Conference on Software Engineering*, September 1979, S. 174–182.

Lea et al. 93: Lea, Rodger, Jacqernot, Christian, und Pilleresse, Eric. *COOL: System Support for Distributed Programming.* Communications of the ACM, Jg. 36, September 1993, Nr. 9, S. 37–46.

Lecarme, Peyrolle-Thomas 78: Lecarme, Olivier und Peyrolle-Thomas, Marie-Claude. *Self-Compiling Compilers: An Appraisal of their Implementation and Portability.* IEEE Software Practice and Experience, Jg. 8, 1978, Nr. 2, S. 149–170.

Lennon 97: Lennon, Jennifer A. *Hypermedia Systems and Application: World Wide Web and Beyond.* Springer Verlag, 1997.

Levin et al. 75: Levin, R., Cohen, E., Corwin, W., Pollack, F., und Wulf, W. *Policy/Mechanism Separation in Hydra.* ACM SIGOPS Operating Systems Review, Jg. 9, November 1975, Nr. 5, S. 132–140.

Lindholm, Yellin 96: Lindholm, Tim und Yellin, Frank. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, 1996.

Liskov et al. 77: Liskov, Barbara, Snyder, Alan, Atkinson, Russell R., und Schaffert, Craig. *Abstraction Mechanisms in CLU.* Communications of the ACM, Jg. 20, August 1977, Nr. 8, S. 564–576.

Liskov, Shriva 88: Liskov, Barbara und Shriva, Liuba. *Promises: Linguistic Support for Efficient Asychronous Procedure Calls in Distributed Systems.* In: *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation '88.* ACM Press, 1988, S. 260–267.

Liskov 88: Liskov, Barbara. *Distributed Programming in Argus.* Communications of the ACM, Jg. 31, Maerz 1988, Nr. 3, S. 300–312.

Lockemann, Schmidt 87: Lockemann, P. C. und Schmidt, J. W. (Hrsg.). *Datenbank-Handbuch.* Springer Verlag, 1987.

Lux 95: Lux, Wolfgang. *Adaptable Object Migration: Concept and Implementation.* ACM SIGOPS Operating Systems Review, Jg. 29, 1995, Nr. 2, S. 54–69.

Maes 87: Maes, Pattie. *Concepts and Experiments in Computational Reflection.* In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '87)*, 1987, S. 147–155.

Maheshwari, Liskov 97a: Maheshwari, Umesh und Liskov, Barbara. *Collecting Distributed Garbage by Controlled Migration.* Distributed Computing, Jg. 10, 1997, Nr. 2, S. 79–86.

Maheshwari, Liskov 97b: Maheshwari, Umesh und Liskov, Barbara. *Collecting Distributed Garbage Cycles by Back Tracing.* In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '97)*, August 1997, S. 239–248.

Maheshwari, Liskov 97c: Maheshwari, Umesh und Liskov, Barbara. *Partitioned Garbage Collection of a Large Object Store.* In: *SIGMOD Conference '97*, 1997, S. 313–323.

Mainetto et al. 95: Mainetto, G., Giacomo, M. Di, und Vinciotti, L. *Distributed Galileo: a Persistent Programming Language with Transactions.* In: Atkinson et al. [95], S. 503–525.

Manola 98: Manola, Frabk. *Towards a Richer Web Object Model.* SIGMOD Record, Jg. 27, Maerz 1998, Nr. 1, S. 76–80.

Masuhara et al. 95: Masuhara, Hidehiko, Matsuoka, Satoshi, Asai, Kenichi, und Yonezawa, Akinori. *Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation.* In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '95)*, 1995, S. 300–315.

Mathiske et al. 95: Mathiske, B., Matthes, F., und Schmidt, J.W. *Scaling Database Languages to Higher-Order Distributed Programming.* In: *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, September 1995.

Mathiske et al. 97: Mathiske, B., Matthes, F., und Schmidt, J.W. *On Migrating Threads.* Journal of Intelligent Information Systems, Jg. 8, 1997, Nr. 2, S. 167–191.

Mathiske 96: Mathiske, Bernd. *Mobilität in persistenten Objektsystemen.* Dissertation, Fachbereich Informatik, Universität Hamburg, Oktober 1996.

Matthes et al. 96: Matthes, F., Müller, R., und Schmidt, J.W. *Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol.* In: *Advances in Databases and Information Systems (ADBIS'96), Proceedings of the Third International Workshop of the Moscow ACM SIGMOD Chapter*, 1996.

Matthes, Schmidt 91a: Matthes, F. und Schmidt, J. W. *Bulk Types: Built-In or Add-On?* In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*, September 1991.

Matthes, Schmidt 91b: Matthes, F. und Schmidt, J.W. *Bulk Types: Built-In or Add-On?* In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece.* Morgan Kaufmann Publishers, September 1991.

Matthes, Schmidt 94: Matthes, F. und Schmidt, J.W. *Persistent Threads.* In: *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, Santiago, Chile, September 1994, S. 403–414.

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer Verlag, 1993.

Matthes 97: Matthes, Florian. *Mobile Processes in Cooperative Information Systems.* In: *Proceedings STJA'97 (Smalltalk und Java in Industrie und Ausbildung)*, Erfurt, Germany, September 1997. Springer Verlag.

Menon et al. 93: Menon, Sathis, Dasgupta, Partha, und LeBlanc, Jr., Richard. *Asynchronous Event Handling in Distributed Object-Based Systems.* In: *Proceedings of the 13th IEEE International Conference on Distributed Computer Systems ICDS '93.* IEEE Society Press, 1993, S. 383–390.

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

Mira da Silva et al. 96: Mira da Silva, M., Atkinson, M.P., und Black, A.P. *Semantics for Parameter Passing in a Type-Complete Persistent RPC.* In: *Proceedings of the 16th IEEE International Conference on Distributed Computer Systems ICDS '96*, 1996, S. 411–418.

Morrison et al. 90: Morrison, R., Atkinson, M.P., Brown, A.L., und Dearle, A. *On the Classification of Binding Mechanisms.* Information Processing Letters, Jg. 34, 1990, Nr. 1, S. 51–55.

Müller 97: Müller, B. *Reengineering - Eine Einführung.* Teubner Verlag, 1997.

Müller iV: Müller, Rainer. *Objektspeicherungssysteme.* Dissertation, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, in Vorbereitung.

Myers et al. 97: Myers, Andrew C., Bank, Joseph A., und Liskov, Barbara. *Parameterized Types for Java.* In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 1997, S. 132–145.

Nagl 96: Nagl, M. (Hrsg.). *Building Tightly Integrated Software Development Environments: The IPSEN Approach.* Springer, 1996.

Nicol et al. 93: Nicol, John R., Wilkes, C. Thomas, und Manda, Frank A. *Object Orientation in Heterogeneous Distributed Computing Systems.* IEEE Computer, Jg. 26, Juni 1993, Nr. 6, S. 57–67.

Niederée 92: Niederée, Claudia. *Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, November 1992.

Nierstrasz, Meijler 95: Nierstrasz, O. und Meijler, T.D. *Requirements for a Composition Language.* In: Ciancarini et al. [95], S. 147–161.

Nietsch 97: Nietsch, Oliver. *Modellierung und Validierung dynamischen Systemverhaltens am Beispiel von Tycoon und StP/OMT.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, April 1997.

Nishio, Yonezawa 93: Nishio, Shojiro und Yonezawa, Akinori (Hrsg.). *Object Technologies for Advanced Software.* Springer Verlag, 1993.

Noble, Satyanarayanan 95: Noble, Brian D. und Satyanarayanan, M. *A Research Status Report on Adaption for Mobile Data Access.* SIGMOD Record, Jg. 24, Dezember 1995, Nr. 4, S. 10–15.

Notkin et al. 86: Notkin, David, Hutchinson, Norm, Sanislo, Jan, und Schwartz, Michael. *Report on the ACM SIGOPS Workshop on Accomodating Heterogeneity.* ACM SIGOPS Operating Systems Review, Jg. 20, April 1986, Nr. 2, S. 9–24.

Nuttall 94: Nuttall, Mark. *A brief survey of systems providing process or object migration facilities.* ACM SIGOPS Operating Systems Review, Jg. 28, 1994, Nr. 4, S. 64–80.

Nygaard, Dahl 81: Nygaard, K. und Dahl, O.-J. *Simula 67.* In: Wexelblat, R. W. (Hrsg.). *History of Programming Languages.* acm press, 1981.

Odersky, Wadler 97: Odersky, Martin und Wadler, Philip. *Pizza into Java: Translating Theory into Practice.* In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 1997, S. 146–159.

OMG 95: OMG. *Object Service RFP5/6.* http://www.omg.org/docs/1995/95-10-13.ps, 1995.

OMG 97a: OMG. *CORBAservices: Common Object Services Specification.* Object Management Group, 1997.

OMG 97b: OMG. *OMA: Object Management Architecture.* Object Management Group, 1997.

OMG 98: OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, 2.2 edition, 1998.

Orfali et al. 97: Orfali, R., Harkey, D., und Edwards, J. *Instant CORBA.* John Wiley & Sons, 1997.

Orfali, Harkey 97: Orfali, R. und Harkey, D. *Client/Server Programming with JAVA and CORBA.* John Wiley & Sons, 1997.

Paepke 90: Paepke, Andreas. *PCLOS: Stress Testing CLOS. Experiencing the Metobject Protocol.* In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '90) and of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '90).* Springer Verlag, 1990, S. 194–211.

Picco et al. 97: Picco, Gian Pietro, Roman, Grnia-Catalin, und McCann, Peter J. *Expressing Code Mobility in Mobile Unity.* In: Jazayeri und Schauer [97].

Piellusch 96: Piellusch, Andreas. *Synchronisation langlebiger Aktivitäten in persistenten Objektsystemen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Juni 1996.

Piquer 96: Piquer, José M. *Indirect Distributed Garbage Collection.* ACM Transactions on Programming Languages and Systems, Jg. 18, September 1996, Nr. 5, S. 615–647.

Plezbert, Cytron 97: Plezbert, Michael P. und Cytron, Ron K. *Does "Just in Time" = "Better Late than Never"?* In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 1997, S. 120–131.

Pratt, Zelkowitz 96: Pratt, T.W. und Zelkowitz, M.V. *Programming Languages: Design and Implementation.* Prentice Hall International, 3rd edition, 1996.

Pree 95: Pree, W. *Design Pattern for Object-Oriented Software Development.* Addison-Wesley Publishing Company, 1995.

Puder, Geihs 97: Puder, A. und Geihs, K. *Meta-Level Service Type Specifications.* In: Rolia et al. [97], S. 74–84.

Ramme 98: Ramme, Kay. *Dynamisches Rebinden in kooperierenden Objektsystemen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Februar 1998.

Redhead 96: Redhead, Tim. *A High-Level Process Checkpointing and Migration Schema for Heterogeneous Distributed Systems.* In: Schill et al. [96], S. 272–284.

Redlich 96: Redlich, Jens-Peter. *Corba 2.0. Praktische Einfuehrung fuer C++ und Java.* Addison-Wesley Publishing Company, 1996.

Reps, Teitelbaum 89: Reps, T. W. und Teitelbaum, T. *The Synthesizer Generator.* Springer Verlag, 1989.

Rolia et al. 97: Rolia, Jerome, Slonim, Jacob, und Bitsford, John (Hrsg.). *Open Distributed Processing and Distributed Platforms.* Chapman & Hall, 1997.

Rosenblum, Wolf 97: Rosenblum, David S. und Wolf, Alexander L. *A Design Framework for Internet-Scale Event Observation and Notification.* In: Jazayeri und Schauer [97].

Rothermel, Popescu-Zeletin 97: Rothermel, Kurt und Popescu-Zeletin, Radu (Hrsg.). *Mobile Agents.* Springer Verlag, 1997.

Roy et al. 97: Roy, Peter van, Haridi, Seif, Smolka, Gert, Mehl, Michael, und Scheidhauer, Rolf. *Mobile Objects in Distributed Oz.* ACM Transactions on Programming Languages and Systems, Jg. 19, September 1997, Nr. 5, S. 804–851.

Ruble 97: Ruble, David A. *Practical Analysis and Design for Client/Server and GUI Systems.* Prentice Hall International, 1997.

Rudloff iV: Rudloff, Andreas. *Nutzungsrechte in offenen Dienstinfrastrukturen.* Dissertation, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, in Vorbereitung.

Rumbaugh et al. 91: Rumbaugh, James, Blaha, Michael, Premerlani, William, Eddy, Frederick, und Lorensen, William. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

Saltzer et al. 84: Saltzer, J. H., Reed, D. P., und Clark, D. D. *End-To-End Arguments in System Design.* ACM Transactions on Computer Systems, Jg. 2, November 1984, Nr. 4, S. 277–288.

Sametinger 97: Sametinger, J. *Software Engineering with Reusable Components.* Springer Verlag, 1997.

Satyanarayanan et al. 95: Satyanarayanan, M., Noble, Brian, Kumar, Puneet, und Price, Morgan. *Application-Aware Adaption for Mobile Computing.* ACM SIGOPS Operating Systems Review, Jg. 29, Januar 1995, Nr. 1, S. 52–55.

Schaefer, Botella 95: Schaefer, W. und Botella, P. (Hrsg.). *Software Engineering - ESEC '95.* Springer Verlag, 1995.

Schelvis, Bledoeg 88: Schelvis, Marcel und Bledoeg, Eddy. *The Implementation of a Distributed Smalltalk.* In: Gjessing, S. und Nygaard, K. (Hrsg.). *ECOOP '88.* Springer Verlag, 1988, S. 212–232.

Schill et al. 95: Schill, A., Bellmann, B., Böhmak, W., und Kümmel, S. *Infrastructure Support for Cooperative Mobile Environments.* In: *Proceedings of the 4th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises WET ICE '95.* IEEE Society Press, 1995, S. 171–178.

Schill et al. 96: Schill, A., Mittasch, C., Spaniol, O., und Popien, C. (Hrsg.). *Distributed Platforms - ICDP 1996.* Chapman & Hall, 1996.

Schmidt et al. 97: Schmidt, J.W., Schröder, G., Niederée, C., und Matthes, F. *Linguistic and Architectural Requirements for Personalized Digital Libraries.* International Journal of Digital Libraries, Jg. 1, 1997, Nr. 1.

Schmidt, Matthes 93: Schmidt, J.W. und Matthes, F. *Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems.* In: *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering RIDE '93 Vienna, Austria*, April 1993, S. 2–17.

Schmidt, Matthes 94: Schmidt, J.W. und Matthes, F. *The DBPL Project: Advances in Modular Database Programming.* Information Systems, Jg. 19, 1994, Nr. 2, S. 121–140.

Schröder 91: Schröder, Gerald. *Die Standardisierung von Modula-2.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, November 1991.

Schröder 93: Schröder, Gerald. *Syntaktische Erweiterbarkeit von Programmiersprachen unter Benennungs-, Bindungs- und Typisierungsinvarianten.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, August 1993.

Sebesta 89: Sebesta, R. W. *Concepts of Programming Languages.* Benjamin/Cummings, 1989.

Senivongse, Utting 96: Senivongse, T. und Utting, I.A. *A Model for Evolution of Services in Distributed Systems.* In: Schill et al. [96], S. 373–385.

Sessions 96: Sessions, Roger. *Object Persistence — Beyond Object Oriented Databases.* Prentice Hall International, 1996.

Shapiro 86: Shapiro, Marc. *Structure and Encapsulation in Distributed Systems: The Proxy Principle.* In: *Proceedings of the 6th IEEE International Conference on Distributed Computer Systems ICDS '86.* IEEE Society Press, 1986, S. 198–204.

Shapiro 94: Shapiro, Marc. *A Binding Protocol for Distributed Shared Objects.* In: *Proceedings of the 14th IEEE International Conference on Distributed Computer Systems ICDS '94.* IEEE Society Press, 1994, S. 134–141.

Shaw, Garlan 96: Shaw, Mary und Garlan, David. *Software Architectures: Perspectives on an Emerging Discipline.* Prentice Hall International, 1996.

Sheard 90: Sheard, T. *Compile-time Reflection: A New Approach to Polymorphism* , 1990.

Shrivastava et al. 91: Shrivastava, Santosh K., Dixon, Graeme N., und Parrington, Graham D. *An Overview of the Arjuna Distributed Programming System.* IEEE Software, Januar 1991, S. 66–73.

Shub 90: Shub, Charles M. *Native Code Process-Originated Migration in a Heterogeneous Environment.* In: *CSC '90. Proceedings of the 1990 ACM eighteenth annual computer science conference on Cooperation.* ACM Press, 1990, S. 266–270.

Silberschatz, Zdonik 96: Silberschatz, Avi und Zdonik, Stan. *Strategic Directions in Database Systems — Breaking Out of the Box.* ACM Computing Surveys, Jg. 28, Dezember 1996, Nr. 4, S. 764–778.

Skarra, Zdonik 87: Skarra, Andrea H. und Zdonik, Stanley B. *Type Evolution in an Object-Oriented Database.* In: Shriver, Bruce und Wegner, Peter (Hrsg.). *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

Smith 84: Smith, Brian Cantwell. *Reflection and Semantics in Lisp.* In: *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages (POPL '84).* ACM Press, 1984, S. 23–35.

Smith 88: Smith, Jonathan A. *A Survey of Process Migration Mechanisms.* ACM SIGOPS Operating Systems Review, Jg. 22, Juli 1988, Nr. 3, S. 28–40.

Soley 92: Soley, Richard Mark. *Object Management Architecture Guide.* Technical report, Object Management Group, 1992.

Sommerville 96: Sommerville, Ian. *Software Engineering.* Addison-Wesley Publishing Company, 5th edition, 1996.

Stamos, Gifford 90: Stamos, James W. und Gifford, David K. *Remote Evaluation.* ACM Transactions on Programming Languages and Systems, Jg. 12, Oktober 1990, Nr. 4, S. 537–564.

Stata, Abadi 98: Stata, Raymie und Abadi, Martín. *A Type System for Java Bytecode Subroutines.* In: *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 1998.

Steensgard, Jul 95: Steensgard, Bjarne und Jul, Eric. *Object and Native Code Thread Mobility Among Heterogeneous Computers.* In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Bd. 29. ACM Press, 1995, S. 68–78.

Stemple et al. 92a: Stemple, D., Sheard, T., und Fegaras, L. *Reflection: A Bridge from Programming to Database Languages.* In: *Proceedings HICSS, Hawaii*, 1992, S. 46–55.

Stemple et al. 92b: Stemple, D., Stanton, R. B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G. N. C., Fegaras, L., Cooper, R. L., Connor, R. C. H., Atkinson, M. P., und Alagic, S. *Type-Safe Linguistic Reflection: A Generator Technology.* Technical report, University of St. Andrews, 1992.

Sun 97: Sun. *Java Remote Method Invocation Specification* , Februar 1997.

Thorn 97: Thorn, Tommy. *Programming Languages for Mobile Code.* ACM Computing Surveys, Jg. 29, September 1997, Nr. 3, S. 213–239.

Tolksdorf 93: Tolksdorf, Robert. *Laura: A Coordination Language for Open Distributed Systems.* In: *Proceedings of the 13th IEEE International Conference on Distributed Computer Systems ICDS '93*, 1993, S. 39–46.

Tolksdorf 95: Tolksdorf, Robert. *Coordination in Open Distributed Systems.* VDI Verlag, 1995.

van der Hoek et al. 97: Hoek, André van der, Hall, Richard S., Heimbigner, Dennis, und Wolf, Alexander L. *Software Release Management.* In: Jazayeri und Schauer [97].

Vegdahl 86: Vegdahl, Steven R. *Moving Structures between Smalltalk Images.* In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '86).* ACM Press, 1986, S. 466–471.

Vogel, Duddy 97: Vogel, Andreas und Duddy, Keith. *Java Programming with CORBA.* John Wiley & Sons, 1997.

Walker et al. 90: Walker, Edward F., Floyd, Richard, und Neves, Paul. *Asynchronous Remote Operation Execution in Distributed Systems.* In: *Proceedings of the 10th IEEE International Conference on Distributed Computer Systems ICDS '90.* IEEE Society Press, 1990, S. 253–259.

Warren, Sommerville 95: Warren, I. und Sommerville, I. *Dynamic Configuration Abstraction.* In: Schaefer und Botella [95], S. 173–190.

Watanabe, Yonezawa 88: Watanabe, Takuo und Yonezawa, Akinori. *Reflection in an Object-Oriented Concurrent Language.* In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages and System Architectures (OOPSLA '89).* ACM Press, 1988, S. 306–315.

Wayner 95: Wayner, P. *Agents unleashed. A public domain look at agent technology.* AP Professional, 1995.

Welling, Badrinath 97: Welling, Girish und Badrinath, B.R. *A Framework for Environment Aware Mobile Applications.* In: *Proceedings of the 17th IEEE International Conference on Distributed Computer Systems ICDS '97.* IEEE Society Press, 1997, S. 384–391.

Wetzel 95: Wetzel, I. *Programmieren mit STYLE: Über die systematische Entwicklung von Programmierumgebungen.* Lang, 1995.

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

Wienberg iV: Wienberg, Axel. *Komponenten in einer objektorientierten Programmiersprache: Modell, sprachliche Umsetzung und Visualisierung.* Diplomarbeit, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, in Vorbereitung.

Wirth 79: Wirth, Niklaus. *The Module: A System Structuring Facility in High-Level Programming Languages.* In: Tobias, Jeffrey M. (Hrsg.). *Language Design and Programming Methodology.* Springer Verlag, September 1979, S. 1–24.

Wirth 88: Wirth, Niklaus. *Programming in Modula-2.* Springer Verlag, 4. edition, 1988.

Wolff, Löhr 96: Wolff, T. und Löhr, K.-P. *Transparently Programming Heterogeneous Distributed Systems.* In: Schill et al. [96], S. 399–412.

Yokoyama et al. 95: Yokoyama, Takanori, Saito, Masahiko, Shimada, Masaru, und Murata, Satoru. *An Object-Based Distributed Computing Environment Based on a Reflective Architecture.* In: *Proceedings of the 15th IEEE International Conference on Distributed Computer Systems ICDS '95.* IEEE Society Press, 1995, S. 263–270.

Zeller 95: Zeller, Andreas. *A Unified Version Model for Configuration Management.* In: *Proceedings of SIGSOFT '95.* ACM Press, 1995, S. 151–160.