Persistenz in objekt-orientierten Programmiersprachen am Beispiel von CLOS

Diplomarbeit

Abgabetermin: 7. September 1994

Heiko Kirschke

Betreuung:
Prof. Dr. Leonie Dreschler-Fischer
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Kognitive Systeme
und
Dr. Volker Haarslev
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Kognitive Systeme

Universität Hamburg • Fachbereich Informatik • Vogt-Kölln-Straße 30 • 22527 Hamburg

Zusammenfassung

Die bisher gebräuchliche Methode zum Speichern von langlebigen, strukturierten Daten ist die Benutzung einer Datenbank, die den BenutzerInnen Datentypen und Operationen über die abgelegten Daten zur Verfügung stellt. Neben der Anfragesprache bietet eine Datenbank oft eine Schnittstelle zu einer Programmiersprache an. Diese Schnittstelle stellt lediglich die Funktionalität der Datenbank zur Verfügung, richtet sich im allgemeinen jedoch nicht nach den Konzepten der Programmiersprache, die die Datenbank verwendet. Eine Lösung dieses Problems besteht darin, die Trennung zwischen Datenbank und Programmiersprache aufzuheben, d.h. die Datenbankkonzepte möglichst vollständig in eine Programmiersprache zu integrieren. In diesem Bericht wurde der Ansatz untersucht und realisiert, das Common LISP Object System um Persistenz zu erweitern.

Abstract

The common approach used for storing longlived structured data is to employ a relational database which provides its users with types for structuring data and operations to manipulate them. Along with its data definition and manipulation language, a database often offers an interface to a general programming language. This interface is a possibility to access the database's functionality, but it neglects the concepts of the programming language using the database. A solution of this problem is to remove the dichotomy between a database and a programming language, i.e. to integrate the concepts of a database into the general programming language. This report presents research done on persistency and a realization of extending the Common LISP Object System by persistency.

Schlüsselwörter – Persistenz, objekt-orientierte Datenbank, persistente Objekte, persistente CLOS-Objekte.
Index Terms – Persistency, object-oriented database, persistent objects, persistent CLOS objects.

Eine leicht gekürzte Fassung dieser Diplomarbeit ist als Bericht FBI-HH-B-179/95 beim Fachbereich Informatik der Universität Hamburg erhältlich.

EMail-Adresse des Autors: `kirschke@informatik.uni-hamburg.de`
World Wide Web: `http://lki-www.informatik.uni-hamburg.de/~kirschke/home.html`

Es ist fast mit der Mathematik wie mit der Theologie. So wie die der letzteren Beflissenen, zumal wenn sie in Ämtern stehen, Anspruch auf einen besonderen Kredit von Heiligkeit und eine nähere Verwandschaft mit Gott machen (obgleich sehr viele darunter wahre Taugenichtse sind), so verlangt sehr oft der sogenannte Mathematiker für einen tiefen Denker gehalten zu werden, ob es gleich darunter die größten Plunderköpfe gibt, die man nur finden kann, untauglich zu irgendeinem Geschäft, das Nachdenken erfordert, wenn es nicht unmittelbar durch jene leichte Verbindung von Zeichen geschehen kann, die mehr das Werk der Routine als des Denkens sind.

— Georg Christoph Lichtenberg

Die Erstellung dieser Arbeit wäre ohne die Liebe und das Vertrauen meiner Frau Nicola nicht möglich gewesen; ihr ist diese Arbeit gewidmet.

Danken möchte ich meiner Betreuerin Frau Prof. Dr. Leonie Dreschler-Fischer für die Anregung zum Thema dieser Arbeit; mein Dank gilt Herrn Dr. Volker Haarslev für die Übernahme der Zweitbetreuung.

Herr Carsten Schröder hat mir durch seine kompetente Unterstützung sehr geholfen; dies begann mit den ersten Hinweisen auf interessante Literatur und setzte sich fort mit der Unterstützung bei der Einarbeitung in das Thema und die verwendeten Systeme. Seine konstruktiven Ratschläge flossen an vielen Stellen sowohl in die schriftliche Darstellung meiner Konzepte in dieser Arbeit als auch in deren Realisierung ein. Dafür bin ich ihm zu großem Dank verpflichtet.

Viele andere Mitarbeiter des Fachbereichs Informatik haben mich ebenfalls unterstützt; danken möchte ich Herrn Harald Lange für die Durchsicht meines ersten Entwurfs und Herrn Ralf Möller für die Hinweise, die er mir zukommen ließ. Last but not least möchte ich mich bei Frau Ingeborg Heer-Mück und Herrn Jörn Tellkamp für den reibungslosen Betrieb der Rechner des Arbeitsbereiches Kognitive Systeme bedanken.

Inhaltsverzeichnis

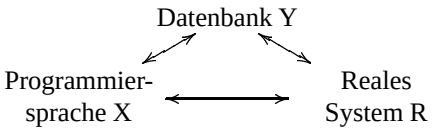
Abbildungsverzeichnis

Kapitel 1

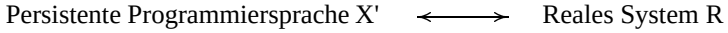
Einleitung

Die Lebensdauer (*extent*) von Daten und Objekten ist bei den meisten Systemen auf die Lebensdauer des Prozesses eingeschränkt, der sie erzeugt hat, d.h. insbesondere wenn dieser Prozeß terminiert, werden alle Daten freigegeben (*flüchtige* oder *transiente* Objekte). Ein persistentes Objekt ist ein Objekt, dessen Lebensdauer unabhängig von dem Prozeß ist, der dieses Objekt erzeugt hat, d.h. ein einmal erzeugtes Objekt kann zu einem anderen Zeitpunkt von einem anderen, späteren Prozeß wiederbenutzt werden [Atkinson et al. 89, S. 75]. In [Morrison et al. 89a, S. 4] wird Persistenz von Daten als der Zeitraum definiert, während der die Daten existieren und benutzbar (*usable*) sind; anzustreben sind Systeme, bei denen Persistenz und Benutzung von Daten getrennt werden.

Die bisher gebräuchliche Methode zum Speichern von langlebigen Daten ist die Benutzung einer Datenbank, die den BenutzerInnen Datentypen und Operationen über die abgelegten Daten zur Verfügung stellt. Viele Datenbanken verfügen über eigene Anfragesprachen, die einen vollständigen Zugriff auf die Möglichkeiten der Datenbank bieten; meist sind diese Anfragesprachen aber nicht für das Erstellen kompletter Applikationen geeignet. Daher wird neben der Anfragesprache oft eine Schnittstelle zu einer Programmiersprache angeboten. Diese Schnittstelle stellt lediglich die Funktionalität der Datenbank zur Verfügung, richtet sich im allgemeinen jedoch nicht nach den Konzepten der Programmiersprache, die die Datenbank verwendet.[1]



Eine Lösung dieses Problems besteht darin, die Trennung zwischen Datenbank und Programmiersprache aufzuheben, d.h. die Datenbankkonzepte möglichst vollständig in eine Programmiersprache zu integrieren:[1]



Es bietet sich an, entweder eine völlig neue Sprache zu erstellen oder aber diese Integration für

[1] Abbildungen nach [Atkinson et al. 89, S. 81]

eine vorhandene Sprache durchzuführen. In dieser Arbeit wurde der zweite Ansatz untersucht und realisiert: das Common LISP Object System (CLOS) wurde um Persistenz erweitert.

1.1 Hintergrund

Das OSCAR (*Open Skies for Conventional Arms Reduction*) Projekt im Fachbereich Informatik der Universität Hamburg befaßt sich mit der Auswertung von Satellitenbildern zur Verifikation von Abrüstungsvereinbarungen. Dazu werden Änderungen an Rüstungsanlagen halbautomatisch überwacht, indem ein Modell einer solchen Anlage im Rechner abgelegt wird und dieses Modell mit den Satellitenbildern der gleichen Anlage verglichen wird; die Aufgabe von OSCAR besteht darin, abrüstungstechnisch relevante Änderungen zwischen Rechnermodell und Satellitenbild zu detektieren und diese Änderungen zur weiteren Verarbeitung kenntlich zu machen. Das im Rechner abgelegte Modell entspricht einem Soll-Zustand; die Satellitenbilder der Anlage geben den Ist-Zustand wieder. Die zu überwachenden Anlagen sind geografischer Natur, d.h. im Gelände feststehende Objekte mit einer gewissen Größe, wie z.B. Flughäfen, Kasernen und Hafenanlagen.

Die für die Verifikation relevanten Anlagen werden als Objekte bestehend aus Teilkomponenten modelliert; so besteht ein Flughafen z.B. aus ein- oder mehreren Hangars, einem Radarturm, den Start- und Landebahnen, den Verbindungswegen, auf denen die Flugzeuge bewegt werden, usw. Die Abhängigkeiten zwischen den Teilkomponenten müssen ebenfalls berücksichtigt werden, z.B. wird man auf einem Flughafen immer einen Verbindungsweg zwischen den Hangars und den Start- und Landebahnen finden. Das allgemeine Konzept einer solchen Anlage wird als generisches Modell abgelegt, welches die einzelnen Teilkomponenten spezifiziert und die Abhängigkeiten zwischen ihnen festlegt. Im generischen Modell wird lediglich qualitativ festgelegt, aus welchen Teilen die gesamte Anlage besteht; erst durch Instanziierung eines generischen Modells wird eine konkrete Anlage erzeugt.

In einem nächsten Schritt ließe sich feststellen, ob beliebige Satellitenbilder Rüstungsanlagen enthalten; dazu würde im Bild nach Strukturen gesucht werden, die die Projektion einer Instanziierung eines generischen Modells auf das Bild sein könnten.

In diesem Zusammenhang ergibt es sich, daß die generischen Modelle sowie ihre Instanziierungen persistent, d.h. über einen längeren Zeitraum, gespeichert werden müssen. Aus den oben genannten Abhängigkeiten zwischen den Teilkomponenten ergibt sich die Notwendigkeit, auf diese Komponenten assoziativ zugreifen zu müssen. Da es sich bei den Instanziierungen um räumliche Daten handelt, muß ebenfalls ein Zugriff mit mehrdimensionalen Schlüsseln möglich sein. Wegen der vom Projekt OSCAR betrachteten Objekte bietet sich für die Realisierung die objekt-orientierte Programmierung an, bei der die Teilkomponenten als Instanzen repräsentiert werden.

1.2 Aufgabenstellung

Die Aufgabenstellung besteht darin, für das im OSCAR Projekt eingesetzte LISP-System eine Möglichkeit zu schaffen, Datenobjekte transparent beliebig lange speichern zu können. Dabei müssen folgende Punkte beachtet werden:

- Das persistente System muß sehr große Objekte handhaben können, wie z.B. Satellitenbilder mit hoher Auflösung.

- Die Anzahl der in einem LISP-System von einem Objekt referenzierten Instanzen kann sehr hoch sein; es sollte möglich sein, ihre Anzahl bezüglich Persistenz einzuschränken, indem sie beispielsweise in ‚interessante' und ‚uninteressante' Instanzen eingeteilt werden und lediglich die ‚interessanten' Objekte persistent gehalten werden. Eine Speicherrückgewinnung sollte den durch nicht mehr benötigte Objekte belegten Speicherplatz wieder freigeben.

- Die Integration des persistenten Systems in vorhandene Systeme soll sehr einfach sein und möglichst wenig Änderungen nach sich ziehen.

- Gefordert sind ebenfalls Datenbankfunktionalitäten, wie Transaktionen und assoziativer mehrdimensionaler Zugriff auf Objekte.

1.3 Übersicht

In Kapitel 2 werden verschiedene Ansätze für Persistenz und ihre Integration in Programmiersprachen vorgestellt. Kapitel 3 behandelt die Identität von Objekten unter dem Gesichtspunkt der Persistenz und die sich daraus ergebenen Konsequenzen für die Realisierung eines persistenten Systems. Verschiedene Architekturen von persistenten Objektsystemen werden in Kapitel 4 erläutert. Kapitel 5 beschäftigt sich mit verschiedenen Programmierkonzepten, die in Bezug auf Persistenz wichtig sind. In Kapitel 6 wird die Realisierung des persistenten Systems SOH beschrieben und analysiert. Kapitel 7 beschreibt das von mir realisierte System für persistente Objekte in Common LISP. Kapitel 8 enthält eine Zusammenfassung der Arbeit und gibt Hinweise für die Weiterentwicklung des in Kapitel 7 vorgestellten Systems.

1.4 Voraussetzungen

Voraussetzung für das Verständnis dieses Textes sind Grundkenntnisse in Common LISP; eine gute Einführung in Common LISP mit dem Schwerpunkt auf praktische Beispiele ist [Norvig 92]. Eine allgemeine Darstellung von LISP findet sich in [Winston et al. 89]. In [AMOP, S. 243–255] werden die Konzepte des Common LISP Object System (CLOS) im Vergleich zu denen von Smalltalk-80 und C++ kurz dargestellt.

Für das Verständnis der beschriebenen Realisierungen von persistenten Systemen sind Kenntnisse des CLOS Metaobjekt-Protokolls (MOP) überwiegend im Bereich der Beeinflussung der Repräsentation von CLOS-Instanzen hilfreich; dies wird in [AMOP, S. 13–34, 47–51, 72–78, 85–90, *96–106*, 137–139, 146–149, 154–158] dargestellt.

1.5 Schreibweisen

> Man spricht deutsch. Si parla Italiano. English spoken. American understood.
>
> — Schild an einem Geschäft im Ernst-Lubitsch-Film ‚Bluebard's Eigth Wife'

Der folgende Text enthält, soweit möglich, die amerikanischen Fachausdrücke in übersetzter Form. Zur Klarstellung, wie der deutsche Begriff zu verstehen ist, wird bei der ersten Benutzung eines von mir übersetzten Begriffes dahinter der amerikanische Originalbegriff in runde

Klammern eingeschlossen angegeben, z.B.: Nach Ablauf einer bestimmten Wartezeit (*time out*) wird ein Fehler signalisiert. Viele dieser Begriffe werden im Glossar (S. 153) erklärt.

Begriffe, deren sinngemäße Übersetzung nicht möglich ist, wie z.B. *Slot* oder *Cache*, wurden von mir im Original verwendet. Bei Zusammensetzungen mit solchen Begriffen habe ich nicht ein Wort gebildet, sondern die Einzelwörter durch einen Bindestrich getrennt, z.B. *Slot*-Zustand statt ‚*Slot*zustand'.

1.6 Begriffe

Altes Lautgedicht
H H H H H H H H H H H H
H H H
H H H H H H
A A A
O la la la O A O A la la
Plinius (i.J. 1847.)

— Kurt Schwitters

Die in diesem Abschnitt definierten Begriffe spiegeln teilweise den Objektbegriff von CLOS wider; für eine allgemeinere Darstellung verweise ich auf [Wegner 90].

1.6.1 Objektbegriff

Ein *Objekt* ist die Abstraktion einer realen Entität; es besitzt eine *Identität* und hat einen *Zustand* und ein *Verhalten*.

Die Identität eines Objektes zeichnet es als einzigartig unter allen Objekten aus. *Werte* sind Objekte, deren Identität durch ihren Zustand festgelegt wird; weiter einschränkend werden Werte als Basisklassen-Instanzen definiert, die grundsätzlich keine weiteren Objekte referenzieren können, wie beispielsweise Zahlen, Zeichen, Zeichenketten und Bitvektoren. Bei (‚echten') Objekten ist die Identität unabhängig von ihrem Zustand [Schewe et al. 92, S. 3]. Mit einer Repräsentationsform für Identität kann ein Objekt referenziert werden. Ein Wert, dessen Zustand direkt in einer Identitätsrepräsentation repräsentiert werden kann, wird als *Immediate* bezeichnet.

Die Struktur des Zustands und das Verhalten eines Objektes wird durch seine *Klasse* festgelegt. Die Struktur des Objektzustands ist aus *Slots* zusammengesetzt; ein *Slot-Zustand* ist der in einem *Slot* enthaltene Teilzustand des Objektes und referenziert einen Wert oder ein Objekt. Das Verhalten des Objektes wird durch seine *Methoden* gebildet; jede einzelne Methode eines Objektes definiert einen Teil seines Verhaltens. Klassen werden durch *Klassen-Metaobjekte* repräsentiert, *Slots* durch *Slot-Metaobjekte* und Methoden durch *Methoden-Metaobjekte*.

Eine Klasse entsteht durch *Spezialisierung* einer oder mehrerer Superklassen, von denen sie Struktur und Verhalten erbt und gegebenenfalls modifiziert oder erweitert. Die aus einer solchen Modifikation hervorgegangene Methode ist eine *spezialisierte Methode*.

Ein *transientes Objekt* ist ein Objekt, dessen Zustand in einem transienten Speicher repräsentiert wird; der Zustand eines *persistenten Objektes* wird in einem persistenten Speicher repräsentiert. Die *transiente Repräsentation eines persistenten Objektes* ist der im persistenten Speicher repräsentierte Zustand des Objektes, der vollständig oder teilweise in eine Repräsentation im transienten Speicher kopiert wurde. Das Verhalten eines persistenten Objektes wird durch das Verhalten seiner transienten Repräsentation bestimmt.

Ein *Objektsystem* ist eine Sammlung von Objekten; es bietet Hilfsmittel zur Manipulation von Objekten an, wie Editoren zum Erstellen von Klassen- und Methoden-Definitionen, Funktionen zur Erzeugung von Klassen, Objekten usw. Innerhalb eines Objektsystems gibt es *Basisklassen*, die sich im Vergleich zu allgemeinen Klassen durch eingeschränkte Möglichkeiten auszeichnen; so können sie beispielsweise nicht spezialisiert werden.

Persistenz ist *orthogonal*, wenn sie eine grundlegende Eigenschaft eines Datums ist und nicht von anderen Eigenschaften des Datums, wie z.B. der Zugehörigkeit zu einem bestimmten Typ, abhängt. Bei *transparenter* Persistenz ergeben sich aus der Sicht der BenutzerInnen keine Änderungen der Schnittstelle zum Objektsystem; die Objekte erhalten unter Beibehaltung ihrer Schnittstelle zusätzlich die Eigenschaft der Persistenz. Intern erfolgt die Einbindung von transparenter Persistenz über spezialisierte Methoden von im System bereits vohandenen (generischen) Funktionen. Bei *intransparenter* Persistenz müssen die für Persistenz erstellten (generischen) Funktionen von den BenutzerInnen explizit benutzt werden.

Die Begriffe Objekt, Instanz und Datum werden synonym verwendet; dies gilt auch für die Begriffe Typ und Klasse.

1.6.2 Begriffe des MOP

Bei der Bezeichnung der Klassen, Objekte und Methoden des MOP habe ich mich an den in [AMOP, S. 74–75, 137, 140] definierten Standard gehalten.

- Die Klassen **class**, **slot-definition**, **generic-function**, **method** und **method-combination** werden als *Basis-Metaobjekt-Klassen* bezeichnet. Eine *Metaobjekt-Klasse* ist eine Subklasse genau einer dieser Klassen.

- Die Klassen **standard-class**, **standard-direct-slot-definition**, **standard-effective-slot-definition**, **standard-method**, **standard-reader-method**, **standard-writer-method** und **standard-generic-function** werden als *Standard-Metaobjekt-Klassen* bezeichnet. Ergänzend fasse ich Subklassen der Klasse **standard-accessor-method** unter dem Begriff *Zugriffs-Methoden-Metaobjekt-Klassen* zusammen. Eine von BenutzerInnen definierte Subklasse einer Standard-Metaobjekt-Klasse wird als *spezialisierte Metaobjekt-Klasse* bezeichnet.

- Auf Standard-Metaobjekt-Klassen spezialisierte Methoden werden als *Standard-Methoden* bezeichnet. Auf spezialisierte Metaobjekt-Klassen spezialisierte Methoden werden als *spezialisierte Methoden* bezeichnet.

- Instanzen einer Metaobjekt-Klasse werden als *Metaobjekte* bezeichnet. Ein *Standard-Metaobjekt* ist die Instanz einer Standard-Metaobjekt-Klasse. Ein *spezialisiertes Metaobjekt* ist die Instanz einer spezialisierten Metaobjekt-Klasse.

  - Ein *Klassen-Metaobjekt* ist eine Instanz der Klasse **standard-class** oder einer ihrer Subklassen.
  - Ein *Slot-Metaobjekt* ist eine Instanz der Klasse **standard-direct-slot-definition** oder **standard-effective-slot-definition** oder einer ihrer jeweiligen Subklassen.
  - Ein *generisches Funktions-Metaobjekt* ist eine Instanz der Klasse **standard-generic-function** oder einer ihrer Subklassen.
  - Ein *Methoden-Metaobjekt* ist eine Instanz der Klasse **standard-method** oder einer ihrer Subklassen.

- Der (ungenaue) Begriff ‚Metaklasse' wird nicht verwendet; statt dessen wird eine Klasse wie z.B. **standard-class** explizit als *Klassen-Metaobjekt-Klasse* bezeichnet (Abbildung 1.3).

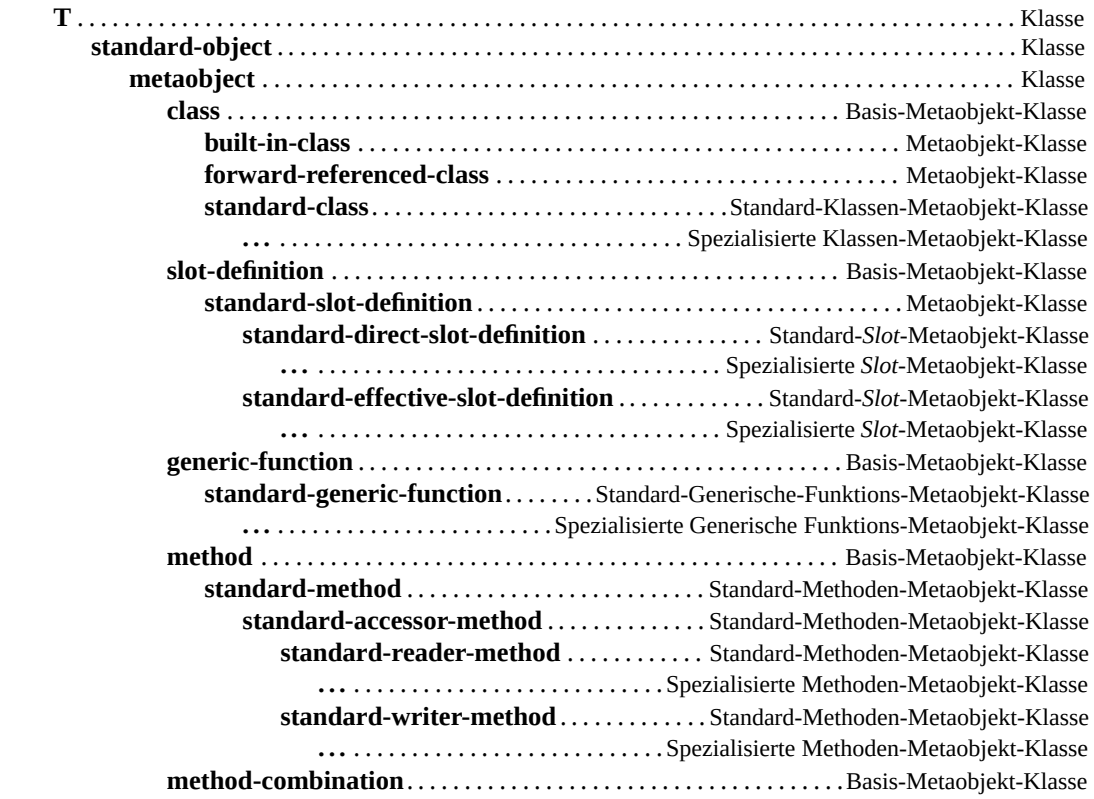


Abbildung 1.3: Metaobjekt-Klassenhierarchie und Klassenbezeichnungen

Kapitel 2

Persistente Systeme

Dieser Abschnitt erläutert verschiedene bisherige Ansätze für persistente Systeme und ihre Integration in Programmiersprachen.

2.1 Relationale Datenbanken

Relationale Datenbanken wurden von Codd Anfang 1970 in erster Linie definiert, damit deren BenutzerInnen besser von Details der Datenrepräsentation abstrahieren können. Eine Datenbank ist genau dann relational, wenn sie mehrere von Codd aufgestellte Kriterien erfüllt [Codd 79, S. 398]; an dieser Stelle gehe ich nur auf die in Zusammenhang mit persistenten Objekten wichtigen Aspekte ein. Die hier wiedergegebenen Definitionen stammen aus [Codd 79, S. 399] und [Schlageter et al. 83, S. 81].

Ein *Bereich* (*domain*) umfaßt alle Werte mit ähnlichem Typ; ein Bereich ist *einfach*, wenn seine Werte atomar sind, d.h. von einer Datenbank nicht weiter zerlegt werden können. Einfache Bereiche sind im allgemeinen Zahlen, Datumsangaben, Zeichen, Zeichenketten usw. D_1, $D_2, \ldots, D_n$ mit $n > 0$ seien nicht notwendigerweise unterschiedliche Bereiche; das kartesische Produkt $D_1 \times D_2 \times \ldots \times D_n$ ist die Menge aller n-Tupel $\langle t_1, t_2, \ldots, t_n \rangle$ mit $t_i \in D_i$ für $1 \leq i \leq n$. Eine Teilmenge R dieses kartesischen Produktes ist eine n-stellige Relation. Ein n-Tupel $x = \langle x_1, x_2, \ldots, x_n \rangle$ mit $x \in R$, $x_i \in D_i$, $1 \leq i \leq n$ wird als *Tupel der Relation R* bezeichnet.

Anstatt der Indexmenge $\{1, 2, \ldots, n\}$ kann auch eine andere ungeordnete Menge verwendet werden, vorausgesetzt, daß zu jeder Tupelkomponente nicht nur sein Bereich, sondern auch sein jeweiliger Index assoziiert wird; die Elemente dieser Menge heißen *Attribute*.

Repräsentiert werden Relationen durch Tabellen; jede Spalte der Tabelle entspricht einem Attribut und jede Zeile einem Tupel (Abbildung 2.1). Relationen sind Mengen; daher müssen die Tupel einer Relation eindeutig sein. Ebenso ist durch den Mengencharakter die Reihenfolge der Tupel und Attribute als beliebig anzunehmen; die in einer real existierenden relationalen Datenbank etablierte Ordnung der Tupel wird ausschließlich aus Effizienzgründen vorgenommen.

Für relationale Datenbanken wird gefordert, daß die Bereiche der Attribute immer einfach sein müssen; nur dann lassen sich die ‚angenehmen' Eigenschaften von relationalen Datenbanken, wie die Existenz von Normalformen zur Vermeidung von Redundanzen und Anomalien,

Relation **Angestellter** ↘ Attribut ↓

	ANG-NR	**NAME**	**GEHALT**	**ABTEILUNG**
Tupel →	4711	Schmoller	3000	Marzipankartoffeln
	4712	Klöbner	6000	Geschäftsführung

Abbildung 2.1: Beispiel Relation **Angestellter**

beweisen. Nicht-einfache Bereiche werden auf Relationen abgebildet. Die Relationenalgebra ermöglicht es, auf das Konzept einer (expliziten) Referenz, so wie sie in allgemeinen Programmiersprachen zur Verfügung steht, zu verzichten; zwei Tupel werden durch den jeweils in einem Attribut enthaltenen gleichen Wert kommutativ in Beziehung gesetzt.

Die grundlegende Eigenschaft einer relationalen Datenbank ist Persistenz aller in ihr enthaltenen Daten; Persistenz ist damit orthogonal als auch transparent. Die Relationenalgebra kann gewisse Eigenschaften einer relationalen Datenbank garantieren. Damit führtenÜberlegungen über persistente CLOS-Objekte bei den AutorInnen einiger persistenter Systeme zu dem Ansatz, das relationale Modell auch für Persistenz von CLOS-Objekten zu benutzen; inwieweit dieser Ansatz sinnvoll ist, wird unter anderem in Kapitel 4 (S. 35) analysiert.

2.1.1 Eingebettete Datenbank-Sprachen

Da relationale Datenbanken normalerweise über keine Datenmanipulationssprache (*DML*) verfügen, die mächtig genug für eine Erstellung von Applikationen ist, bieten sie eine Schnittstelle zu einer allgemeinen Programmiersprache an. Die Leistung dieser Schnittstellen erschöpft sich meist in einer syntaktischen Umsetzung der im Programm eingefügten Datenbank-Anfragen in die Datenmanipulationssprache, ohne daß die Konzepte der Programmiersprache berücksichtigt werden; ebensowenig finden sich in der Programmiersprache geeignete Konzepte für die Verarbeitung von relationalen Daten. Der bei Verwendung einer solchen Schnittstelle entstehende Code besteht damit meist aus einem Gemisch zweier Sprachen, der Datenmanipulationssprache und der allgemeinen Programmiersprache (siehe z.B. *Embedded SQL* [O'Neil 94, S. 201–283] oder die *LIBPQ*-Schnittstelle von POSTGRES [Wensel 93, S. 112–126]). Persistenz über eingebettete Datenbank-Sprachen ist weder orthogonal noch transparent.

2.1.2 PASCAL/R

In PASCAL/R [Schmidt et al. 80] [Schmidt 77] [Clamen 91, S. 5–6] wurde eine relationale Datenbank in PASCAL integriert. Die Ergänzung besteht zum einen in der Erweiterung um die Typen **relation** und **database**, deren Instanzen Relationen bzw. relationale Datenbanken sind, und zum anderen in der Bereitstellung von Konstrukten zum Traversieren und Manipulieren von Relationen, die vollständig in die Sprache integriert wurden. Da in PASCAL/R ausschließlich Tupel persistent gehalten werden können, ist Persistenz nicht orthogonal. Durch die Auswahl und die erfolgte Integration der Manipulationsmöglichkeiten in PASCAL ist die für Tupel angebotene Persistenz weitgehend transparent. Die Weiterentwicklung von PASCAL/R führte zu der auf Modula-2 basierenden relationalen Programmiersprache DBPL [Matthes 92a] [Matthes et al. 92b].

2.2 Nicht-relationale Ansätze

Da die Kopplung einer relationalen Datenbank mit einer allgemeinen Programmiersprache wegen der unterschiedlichen Datenmodellierung im allgemeinen dazu führt, daß die Möglichkeiten beider Systeme nicht zufriedenstellend genutzt werden können, begann ab 1980 eine Entwicklung, bei der nicht das relationale Modell in eine allgemeine Programmiersprache integriert wurde, sondern umgekehrt eine Programmiersprache um Persistenz erweitert wurde.

2.2.1 PS/Algol

In PS/Algol [Clamen 91, S. 16–22] [Nikhil 88, S. 55] wird Persistenz erstmals als grundlegende Eigenschaft eines Datums unabhängig vom Typ angesehen; es handelt sich damit um die erste Programmiersprache mit orthogonaler Persistenz. Anders als in PASCAL/R wurden persistente Objekte nicht nach dem relationalen Modell realisiert, sondern ihre Zustände werden in einem persistenten *Heap* abgelegt; innerhalb dieses *Heaps* werden Referenzen explizit repräsentiert. Für die Lokalisierung von Objekten wird die Möglichkeit geboten, sie unter einem Namen ablegen und adressieren zu können. Der Nachfolger von PS/Algol ist die persistente Programmiersprache Napier [Morrison et al. 89b] [Morrison et al. 89a]; dort werden zusätzlich Prozeduren als persistente Objekte aufgefaßt.

2.3 Tycoon

Tycoon (Typed Communicating Objects in Open Environments) [Matthes et al. 93] [Matthes et al. 92c] baut auf den mit PASCAL/R und DBPL gemachten Erfahrungen auf. Die Tycoon Umgebung sieht Persistenz wie PS/Algol als grundlegende Eigenschaft eines Datums an. Sie ist in Tycoon vollständig und transparent von vornherein in die Sprache integriert worden. Durch die orthogonale Sichtweise von Persistenz wird auf eine Modellierung des verwendeten persistenten Speichers und einem direkten Zugriff auf die in ihm realisierten Manipulationsmöglichkeiten innerhalb der Sprache verzichtet; statt dessen werden in Tycoon die gewünschten Datenbankkonstrukte mittels der Sprache selbst realisiert.

2.4 Fazit

Das relationale Modell ist konzeptionell sehr elegant und einfach; es bietet mächtige Manipulationsmöglichkeiten und mathematisch beweisbare‚angenehme‘ Eigenschaften. Wie jedoch in Abschnitt 7.2.5 (S. 83) gezeigt wird, führen die durch das Modell vorgegebenen Kriterien zu Einschränkungen, die gegen die Verwendung des relationalen Modells für persistente CLOS-Objekte im Rahmen dieser Arbeit sprechen. Statt dessen vertrete ich den Ansatz, für Persistenz den durch PS/Algol vorgezeigten Lösungsweg weiter zu verfolgen.

Kapitel 3

Identität

> Heute betrachtet sich das Vieh darin [im Spiegelbild eines Flusses, Anm. d. Verf.] und sieht sich schwimmen, wenn es daran entlangläuft, und Hänfling und Buchfink verdoppeln sich darin, ohne ihre Einzigkeit zu verlieren. Sie entstehen darin wieder, ohne zu sterben, und wundern sich, daß ihnen in solch kaltem Nest im Nu Junge ausschlüpfen, die genauso groß sind wie sie selber.
>
> — Cyrano De Bergerac: Herzstiche

Die Objekte innerhalb eines Objektsystems stehen nicht für sich alleine, sondern sind durch Referenzen untereinander verbunden. Eine Referenz stellt einen eineindeutigen Bezug zur referenzierten Instanz her. Ein Objekt wird durch seine Identität unterscheidbar von allen anderen Objekten; Eineindeutigkeit für eine Referenz ist daher dann gewährleistet, wenn sie aus der Identität des Objektes bestimmt wird. Dieses Kapitel beschäftigt sich deswegen mit dem Identitätsbegriff allgemein und spezieller mit den möglichen Repräsentationsformen für Identität von transienten und persistenten Objekten zum Aufbau von Referenzen zwischen Instanzen.

Mit Identität wird die Eigenschaft bezeichnet, daß ein Ding oder ein Sachverhalt nur mit sich selbst in allen Merkmalen (*properties*) übereinstimmt [Klaus et al. 76, S. 543] [Zdonik et al. 90, S. 9].[1] Laut *Identitätsprinzip* sind zwei Dinge *x* und *y* genau dann identisch, wenn alle Merkmale *P* zwischen diesen beiden Dingen äquivalent sind:

$$(x \equiv y) \stackrel{def}{=} \bigwedge_{P} P(x) \leftrightarrow P(y) \tag{3.1}$$

In der hier vorliegenden Form kann das Identitätsprinzip (lediglich) verwendet werden, um ein Individuum gegen andere abzugrenzen, indem z.B. alle Individuen zu einem bestimmten Zeitpunkt betrachtet werden; das Identitätsprinzip ermöglicht in diesem Fall die Bestimmung einer Identität für jedes einzelne Individuum, da es sich von allen anderen in mindestens einem Merkmal unterscheidet. Diese sogenannte *abstrakte Identität* ist die triviale Feststellung, daß jedes Individuum genau mit sich selbst identisch ist und wird auch als $x = x$ formuliert.

[1] Dagegen ist Gleichheit lediglich die Übereinstimmung in wesentlichen Merkmalen zweier nicht-identischer Dinge [Schmidt 82, S. 303].

Der nicht-triviale Fall besteht darin, daß sich Merkmale eines Individuums ändern können. Bei einer Änderung bleibt die abstrakte Identität gegenüber allen anderen Individuen erhalten, aber das Identitätsprinzip gilt in der oben genannten Form nicht mehr für das Individuum selbst vor und nach einer Änderung. Trotzdem bleibt das Individuum mit sich selbst gleich, es ist *konkret-identisch*. Die Merkmale eines Individuums lassen sich in konstante und variable Merkmale einteilen; da die sich ergebenden Änderungen der variablen Merkmale im allgemeinen nicht vorhersehbar sind, kann ein eingeschränktes Identitätsprinzip zur Identitätsfeststellung eines Individuums verwendet werden, wobei die Einschränkung darin besteht, nicht alle, sondern ausschließlich die konstanten Merkmale zu betrachten. Die Einteilung in konstante und variable Merkmale muß je nach Verwendungszweck angemessen vorgenommen werden, so daß die konkrete Identität eines Individuums hinreichend feststellbar wird.

Beispielsweise sind variable Merkmale bei einer Person die Kleidung oder die Umgebung, in der sich die Person befindet, wie die aktuelle Postanschrift; ein oft konstantes Merkmal ist der Name; immer konstante Merkmale sind Geburtstag und -ort. Bei Immobilien ist hingegen die Ortsangabe ein konstantes Merkmal, während die aktuellen BesitzerInnen oder BewohnerInnen eines solchen Objektes variable Merkmale sind.

Die in diesem Zusammenhang interessierenden Individuen sind Daten im Sinne der (objektorientierten) Programmierung. Programmierung verfolgt die Absicht, einen Teil der realen Welt als Modell nachzubilden. Dazu wird über die realen Gegebenheiten abstrahiert. Aus den einzelnen konkreten Sachverhalten werden allgemeingültige Gesetzmäßigkeiten abgeleitet und in eine Modellwelt übertragen; unwesentliche Dinge werden im Modell weggelassen [Wand 89, S. 538]. Ein Datenobjekt ist die Abbildung eines Dings oder Sachverhaltes der realen Welt in die Modellwelt. Ein reales Ding besitzt eine Identität; deswegen ist es sinnvoll, den Identitätsbegriff der realen Welt auch auf die durch die Programmierung geschaffene Modellwelt zu übertragen.

3.1 Identitätsrepräsentation

Identität ist eine zu jedem Individuum gehörige Eigenschaft; wenn ein System die Identität der in ihm enthaltenen Individuen widerspiegelt, muß die Identität der einzelnen Individuen durch eine geeignete Form repräsentiert werden. Diese Form kann zur Identifikation (lat. *identitas* ‚Wesenseinheit' und *facere* ‚machen' [Wahrig 68]) eines Individuums benutzt werden. Für Individuen, Identität und Repräsentationsformen für Identität gelten folgende Grundsätze:

1. Es gibt Individuen; jedes Individuum hat eine eigene, genau ihm zugehörige Identität [Paton et al. 88, S. 284].

2. Es gibt keine Identität ohne ein dazugehöriges Individuum.

3. Die Identität eines Individuums kann durch eine geeignete Form repräsentiert werden.

Der Rest dieses Kapitels beschäftigt sich mit Identitätsrepräsentationen und geht insbesondere der Frage nach, was eine ‚geeignete Form' ist.

3.1.1 Referenzen zwischen Individuen

Eine Identitätsrepräsentation kann als Stellvertreter für das Individuum selbst benutzt werden. Damit wird es erst möglich, Referenzen zwischen Individuen aufzubauen. Umgekehrt macht

es das Fehlen einer solchen Repräsentation unmöglich, Referenzen zwischen Objekten zu etablieren; in diesem Fall kann ein Objekt selbst lediglich in einem‚umfassenden' Objekt als Wert enthalten sein, da es nicht referenzierbar ist.

Abbildung 3.1 zeigt als Beispiel den Ausschnitt einer Personenkartei. Es sind insgesamt

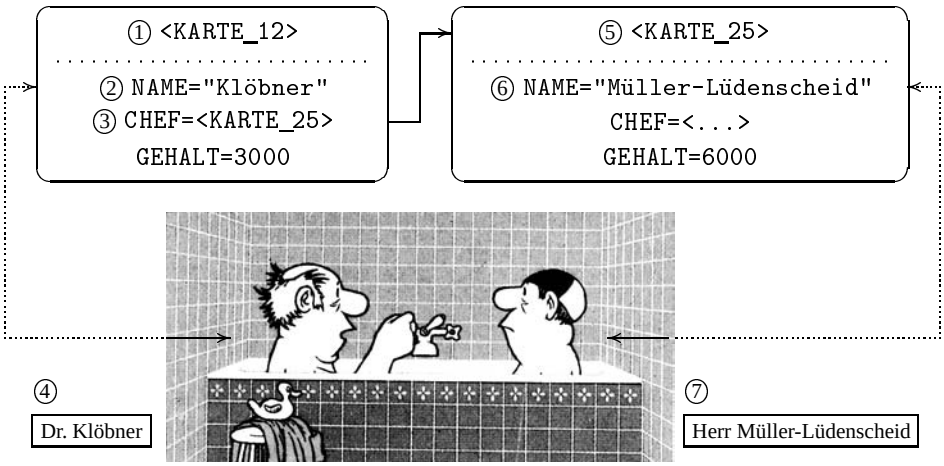


Abbildung 3.1: Identitäten, Identitätsrepräsentationen und Referenzen in einer Personenkartei

vier Individuen beteiligt: zwei Personen (Dr. Klöbner ④ und Herr Müller-Lüdenscheid ⑦) sowie zwei Karteikarten (Karten ① und ⑤). Jedes dieser Individuen hat eine eigene Identität. Die Identitäten der Individuen werden über ihre Namen repräsentiert (Tabelle 3.2).Über die-

$\downarrow^a$ **wird repräsentiert durch den**	**Namen** $\downarrow$
Person mit dem Namen ‚Klöbner'	`"Klöbner"`
Person mit dem Namen ‚Müller-Lüdenscheid'	`"Müller-Lüdenscheid"`
Karte, die die Person mit dem Namen ‚Klöbner' repräsentiert	`<KARTE_12>`
Karte, die die Person mit dem Namen ‚Müller-Lüdenscheid' repräsentiert	`<KARTE_25>`

[a] Zu lesen: Person mit dem Namen Klöbner wird repräsentiert durch den Namen `"Klöbner"`

Tabelle 3.2: Identitätsrepräsentation in einer Kartei

se Repräsentationsformen können die in Abbildung 3.1 eingezeichneten Referenzen aufgebaut werden. Da die Referenz zwischen Person und dazugehöriger Karteikarte über den Namen der Person aufgebaut wird, darf er sich nicht ändern, da sonst die Auflösung der Referenz nicht mehr möglich ist. Ebenso muß der Name eindeutig sein.

3.1.2 Bedingungen für eine Identitätsrepräsentation

Folgende Bedingungen gelten für eine Identitätsrepräsentation von Datenobjekten:

Eindeutigkeit Sie muß eineindeutig sein. Jedes Datum muß eine eigene Identitätsrepräsentation erhalten, die sich von allen anderen gleichzeitig erreichbaren Datenobjekten unterscheidet, da sonst keine eindeutige Identifikation über diese Repräsentationsform möglich ist.

Unabhängigkeit vom aktuellen Zustand Sie darf nicht von Bestandteilen des Datenobjektes abhängig sein, deren Zustand sich während seiner Lebensdauer ändern könnten, da sonst Referenzen auf das Datenobjekt ungültig werden.

Unabhängigkeit von der Umgebung Reale Objekte behalten ihre Identität bei einem Umgebungswechsel; analog dazu behält auch ein Datenobjekt unabhängig von der aktuellen Umgebung seine Identität. Für die gewählte Repräsentationsform von Identität folgt daraus, daß sie davon unabhängig sein muß, falls sich diese während der Lebensdauer des Objektes ändern könnte.

Zeitliche Unabhängigkeit Identität von realen Objekten ist invariant gegenüber Veränderung der Zeit während ihrer Lebensdauer; gleiches gilt für Datenobjekte. Daher muß die Repräsentationsform für Identität konstant bleiben über die Zeitdauer, während der das Datenobjekt existiert.

Das Umfeld, in dem sich die zusammengehörigen Objekte befinden, bestimmt, welche Repräsentation für Identität gewählt werden kann, damit diese Bedingungen erfüllt sind. Wenn beispielsweise alle Objekte ausschließlich in einem nicht-verteilten abgeschlossenem System enthalten sind, genügt es, wenn die Identitätsrepräsentation innerhalb dieses Systems eindeutig ist. Bei verteilten Systemen ist es notwendig, daß die Repräsentation eindeutig für alle beteiligten Prozesse ist.

3.1.3 Forderungen an eine Identitätsrepräsentation

Außer diesen notwendigen Bedingungen sollte eine Identitätsrepräsentation zusätzliche Forderungen erfüllen.

Effiziente Realisierung Bei Systemen mit sehr vielen Objekten sollte die Identitätsrepräsentation möglichst kurz sein; sie sollte insbesondere in einem vernünftigen Verhältnis zur mittleren Größe der Datenobjekte selbst stehen.

Möglichst hoher lokaler Informationsgehalt Eine Identitätsrepräsentation sollte soviel Information wie möglich über das sie repräsentierende Objekt enthalten [Heiler et al. 89, S. 244]. Die Art der Information ist abhängig vom beabsichtigten Verwendungszweck der Objekte. Wenn eine Identitätsrepräsentation z.B. Komponenten enthält, die eine Lokalisierung des dazugehörigen Datenobjektes ermöglichen, kann der Zugriff auf den Objektzustand effizienter realisiert werden. Bei Systemen mit Typbindung der Datenobjekte zur Laufzeit ist es für Typprüfungen von Vorteil, wenn bereits die Identitätsrepräsentation Rückschlüsse auf den Typ des Datenobjektes zuläßt, da dann kein Zugriff auf die Typinformation des Datenobjektes selbst notwendig ist. Je mehr Information in einer Identitätsrepräsentation enthalten ist, um so größer ist sie; damit bedingt ein hoher Informationsgehalt eine geringere Platzeffizienz.

Objekte werden durch Identitätsrepräsentationen referenziert; für sie ergibt sich bei Identitätsrepräsentationen mit Informationsgehalt die Einschränkung, daß sich diese, das Objekt betreffenden Informationen nicht ändern dürfen, weil sonst die in allen das Objekt referenzierenden Identitätsrepräsentationen enthaltenen Informationen ungültig werden. Eine Prüfung, ob dieser Fall eingetreten ist mit anschließender Korrektur der Informationen in der Identitätsrepräsentation ist nicht sinnvoll zu realisieren, da die Prüfung selbst weitere Dereferenzierungen (z.B. des Objektzustands) nach sich zieht und damit der Effizienzvorteil verloren geht.

Kanonische Identitätsrepräsentation Eine kanonische Identitätsrepräsentation ermöglicht eine einheitliche Verarbeitung aller im System auftretenden Identitätsrepräsentationen unabhängig vom Typ des Objektes, das sie repräsentieren. Damit wird die Erstellung eines solchen Systems vereinfacht und die Übersichtlichkeit erhöht. Eine kanonische Form kann bei Datenobjekten von einfachen, skalaren Typen dazu führen, daß die Repräsentationsform mehr Platz verbraucht als das Objekt, das sie repräsentiert; zur Vermeidung dessen dürfen in Common LISP Daten von einfachen Typen (z.B. Daten der Typen **fixnum** [CLtLII, S. 288] oder **character** [CLtLII, S. 371]) nicht-kanonisch repräsentiert werden.

Ein Beispiel für eine kanonische Form ist die in diesem Kapitel verwendete Notation für die Identitätsrepräsentation von Objekten. Sie besteht aus der Konkatenation einer öffnenden spitzen Klammer mit dem Namen des Objekt-Typs, einem Unterstrich, einer beliebig gewählten und innerhalb des Typs eindeutigen Nummer und einer schließenden spitzen Klammer (z.B. `<ANGESTELLTER_23>`). Aus dieser kanonischen Form läßt sich damit zusätzlich bereits der Typ des durch sie repräsentierten Objektes ableiten.

Spätes ‚Materialisieren' des Datenobjektes Eine Identitätsrepräsentation dient unter anderem als Referenz auf das dazugehörige Datenobjekt. Eine Dereferenzierung besteht daraus, eine Abbildung zu evaluieren, die mit der Identitätsrepräsentation als Argument den Zugriff auf den Zustand des sie repräsentierenden Objektes ermöglicht. Spätestens zu diesem Moment muß das Datenobjekt ‚materialisiert' sein, d.h. in einer direkt dereferenzierbaren Repräsentation (z.B. im transienten Speicher) vorliegen [Heiler et al. 89, S. 240]. Vorher besteht diese Notwendigkeit nicht; es ist also ausreichend, bis zur ersten Dereferenzierung lediglich die Identitätsrepräsentation eines Datenobjektes als Stellvertreter für die persistente Repräsentation zu benutzen, die dann beim (ersten) Zugriff den Aufbau einer direkt referenzierbaren (transienten) Repräsentation nach sich zieht.

Spätes Materialisieren läßt sich demzufolge nur dann erreichen, wenn zwischen Identitätsrepräsentation und Datenobjekt eine ‚schwache' Kopplung existiert.

3.2 Übersicht über Identitätsrepräsentationen

Abbildung 3.3 zeigt eine Übersicht für verschiedene Realisierungen von Identitätsrepräsentationen, die zum einen in vorhandenen transienten Systemen verwendet werden (Adressen) und die sich zum anderen für persistente Umgebungen eignen (Bezeichner und Surrogate). In den folgenden Abschnitten werden die einzelnen Repräsentationen erläutert und bewertet.

3.2.1 Adressen

Eine der einfachsten Möglichkeiten zur Identitätsrepräsentation eines Datums ist seine Adresse. Bei direkten Adressen reicht eine einmalige Dereferenzierung für den Zugriff auf den Zustand des dazugehörigen Datenobjektes aus (Abbildung 3.4). Indirekte Adressen referenzieren einen sogenannten *Forwarder*. Ein *Forwarder* ist eine Datenstruktur mit meist fester Größe, die eine direkte Referenz auf das Datenobjekt sowie gegebenenfalls weitere Informationen über das Datenobjekt enthält; indirekte Adressen müssen damit für den Zugriff auf den Objektzustand doppelt dereferenziert werden. Oft werden alle *Forwarder* in einer Tabelle zusammengefaßt; eine indirekte Adresse kann dann als Index in diese Tabelle realisiert werden.

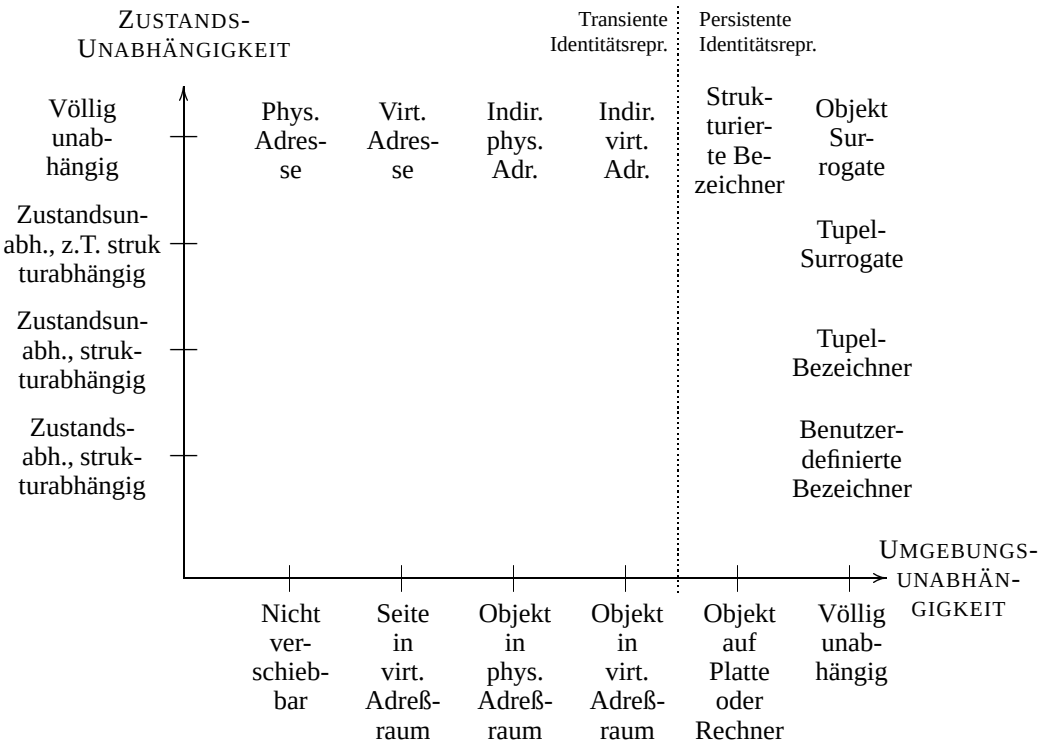


Abbildung 3.3: Übersicht über Identitätsrepräsentationen (nach [Khoshafian et al. 90, S. 43])

Die Interpretation einer Adresse erfolgt auf der *Hardware*-Ebene und damit auf einer sehr niedrigen Abstraktionsebene.

Eindeutigkeit, Unabhängigkeit vom aktuellen Objektzustand

Sofern die Allozierung des die Daten enthaltenen Speicherbereichs korrekt erfolgt, sind Adressen innerhalb ihres Gültigkeitsbereichs immer eindeutig. Sie bleiben bei Zustandsänderungen von Datenobjekten konstant.

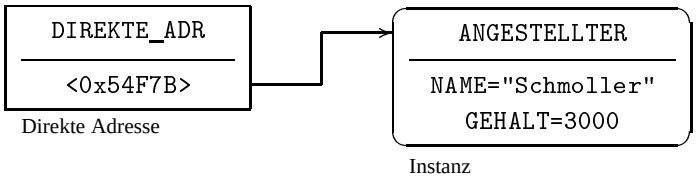


Abbildung 3.4: Direkte Adressen

Umgebungsunabhängigkeit

Indirekte Adressen (Abbildung 3.5) ermöglichen eine Relozierung der referenzierten Objekte innerhalb des Adreßraumes. Beim Wechsel der Umgebung von Objekten aus ihrem Adreßraum heraus wird eine Identitätsrepräsentation über die Adresse entweder nicht mehr möglich (z.B. weil die neue Umgebung keine vergleichbaren [Hauptspeicher-]Adressen hat, wie bei der Übertragung von Objekten in Dateien), oder es kann nicht sichergestellt werden, daß die Objekte in

der neuen Umgebung die gleiche Adresse wie in der vorherigen Umgebung haben (z.B. Transfer von Daten auf eine andere Maschine). Bei der Verwendung von Adressen als Identitätsrepräsentation ist Umgebungsunabhängigkeit damit nicht gegeben.

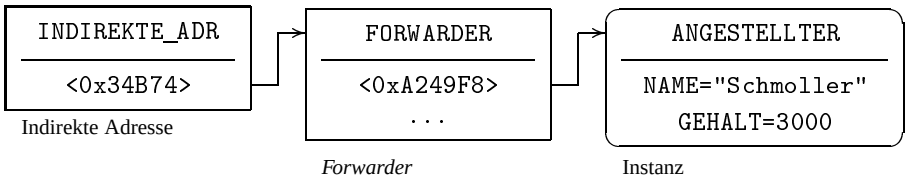


Abbildung 3.5: Indirekte Adressen

Zeitliche Unabhängigkeit

Damit Daten, die sich innerhalb eines transienten Speichers befinden, persistent werden, müssen sie in eine nicht-flüchtige Umgebung transferiert werden. Damit treten für persistente Daten die im letzten Absatz geschilderten Probleme auf; zeitliche Unabhängigkeit ist also bei Adressen als Identitätsrepräsentation für persistente Objekte nicht gegeben.

Sofern indirekte Adressen wie oben genannt als Index in eine Tabelle mit *Forwardern* realisiert werden, läßt sich das Problem der Umgebungs- und Zeitunabhängigkeit dadurch lösen, daß in der persistenten Umgebung außer den Instanzen die Tabelle aller *Forwarder* ebenfalls persistent abgelegt wird; die für die Referenz zwischen *Forwarder* und seiner dazugehörigen Instanz verwendete Identitätsrepräsentation muß dann jeweils der Umgebung angepaßt werden, in der sich das Objekt befindet. Ein Beispiel dafür ist die in GemStone verwendete Adressierung über eine Objekttabelle (*object table*), die eine indirekte Adresse auf den physikalischen Ort eines Objektes abbildet [Maier et al. 87, S. 370]. In LOOM (*Large Object-Oriented Memory*) werden für Referenzen zwischen Objekten indirekte Adressen benutzt, deren Repräsentationsform davon abhängt, ob sich das Objekt im transienten oder persistenten Speicher befindet; bei Übertragung eines Objektes zwischen den Speicherarten werden die indirekten Adressen konvertiert [Kaehler et al. 90, S. 299].

Effizienz

Adressierung von Daten wird auf allen Rechnern bis auf niedrigste Ebene (*Hardware*) unterstützt und ist damit als Identitätsrepräsentation für Datenobjekte, die im transienten Speicher liegen, sehr effizient. Der Zugriff auf den Objektzustand über eine indirekte Adresse ist wegen der doppelten Dereferenzierung langsamer als bei einer direkten Adresse.

Informationsgehalt

Wieviel Informationen bereits eine Adresse über das von ihr dereferenzierte Objekt enthalten kann, ist von der verwendeten Rechnerarchitektur abhängig. Rechner mit einfachen Prozessoren können in einer Adresse meist keine Informationen über das dereferenzierte Objekt ablegen; in diesem Fall müssen die benötigten Informationen immer über eine Dereferenzierung der Adresse besorgt werden.

Bei anderen Rechnern bestehen Adressen aus der Speicheradresse und einer Menge von Markierungsbits (*tag bits*), in denen begrenzt Informationen über das dereferenzierte Objekt

abgelegt werden können; damit können diese Informationen ohne eine Dereferenzierung des Objektes selbst ermittelt werden. Wenn das mit der Adresse repräsentierte Objekt ein referenzierbares Objekt ist, dürfen sich die in den Markierungsbits abgelegten Informationen über das repräsentierte Objekt nicht ändern, da sonst die in allen das Objekt referenzierenden Adressen enthaltenen Informationen ungültig werden.

Meist enthalten die Markierungsbits Informationen über den Typ des referenzierten Objektes; aus der oben genannten Einschränkung folgt, daß sich in diesem Fall der Typ des Objektes nicht ändern darf.[2] Eine mögliche Markierung wäre z.B., daß eine Adresse selbst als Wert zu interpretieren ist.

Bei indirekten Adressen lassen sich im *Forwarder* zusätzliche Informationen über das referenzierte Datenobjekt ablegen. Um diese Informationen zu erhalten, reicht damit eine einmalige Dereferenzierung der indirekten Adresse aus; das Objekt selbst braucht nicht dereferenziert zu werden. Die Ablage von Informationen im *Forwarder* führt dazu, daß ein Teil der Informationen vom Objekt weg in den *Forwarder* verlagert wird; damit müssen die Informationen im *Forwarder* durch das System besonders berücksichtigt werden, z.B. müssen sie beim Aufbau des *Forwarders* aus dem Objekt bestimmt werden und in den *Forwarder* geschrieben werden.

Kanonische Form

Direkte und indirekte Adressen als Identitätsrepräsentation sind dann kanonisch, wenn jede Adresse unabhängig von Merkmalen (wie z.B. dem Typ) des referenzierten Datenobjektes die gleiche Form hat. Aus Effizienzgründen wird oft eine nicht-kanonische Form gewählt, bei der z.B. die Anzahl der verwendeten Markierungsbits vom Typ des Datenobjektes abhängt. In diesem Fall ist die Repräsentation speicherplatz-sparend, verlangt aber einen höheren Aufwand bei der Verarbeitung, weil jeder Typ spezialisierte Methoden benötigt.

Spätes Materialisieren

Eine direkte Adresse verweist immer auf ein existierendes Datenobjekt; es gibt keine direkten Adressen von (noch) nicht existierenden Datenobjekten. Damit die Identität eines Datums über seine direkte Adresse repräsentiert werden kann, muß also das Datum selbst im Speicher alloziert werden. Dies gilt ebenso für virtuellen Speicher, der allerdings nach erfolgter Allozierung die Möglichkeit bietet, den physikalischen Hauptspeicher durch Aus- und Einlagerung von Daten (*paging*) zu entlasten; damit findet eine späte Materialisierung des Objektes für die nach der Allozierung folgenden Zugriffe auf den Objektzustand statt.

Eine indirekte Adresse setzt für ihre Existenz nicht die Allozierung des Speicherplatzes für das durch sie repräsentierte Datenobjekt, sondern lediglich die Allozierung eines *Forwarders* voraus. Damit ließe sich ein spätes Materialisieren dadurch erreichen, daß innerhalb des *Forwarders* Informationen abgelegt werden, wie das Objekt materialisiert werden kann; das Objekt selbst wird dann beim ersten Zugriff auf den Objektzustand nach der Dereferenzierung des *Forwarders* materialisert.

[2] In diesem Zusammenhang sei auf den Unterschied zwischen dem Typ eines Objektes (der in diesem Fall nicht geändert werden darf) und dem Typ selbst (der durchaus geändert werden könnte, sofern er während der‚Lebenszeit' der Objekte zur Verfügung steht) hingewiesen.

Bewertung

Adressen ermöglichen eindeutige und effiziente Referenzen innerhalb eines transienten physikalischen oder virtuellen Adreßraumes; für persistente Objekte sind sie nicht geeignet, da ihre Gültigkeit an den Adreßraum selbst und seine im allgemeinen endliche Lebensdauer gebunden ist. Da die Effizienz von Adressen extrem hoch ist, kann es sich lohnen, ein persistentes Objekt in den transienten Adreßraum zu übertragen; im Verlauf derÜbertragung müßte dann unter anderem eine Konvertierung der persistenten Identitätsrepräsentation in eine Adresse vorgenommen werden (*Pointer-Swizzling*).

3.2.2 Strukturierte Bezeichner

Strukturierte Bezeichner (*structured identifier*) sind Identitätsrepräsentationen, die Informationen über die jeweils repräsentierten Objekte enthalten. Ein Teil der die Objekte betreffenden Informationen wird von den Objekten weg in die strukturierten Bezeichner verlagert. Die Strukturierung wird so gewählt, daß diese Informationen relativ einfach aus den Bezeichnern ermittelt werden können. Die Interpretation eines strukturierten Bezeichners erfolgt auf einer höheren Abstraktionsebene als bei einer Adresse.

Arten von strukturierten Bezeichnern

In persistenten Systemen realisierte strukturierte Bezeichner lassen sich in wertstrukturierte, typstrukturierte und ortsstrukturierte Bezeichner einteilen.

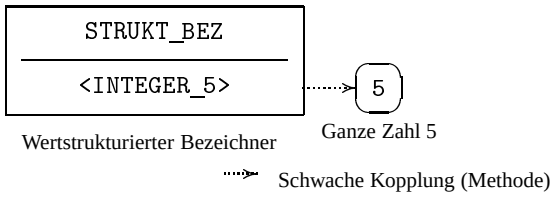


Abbildung 3.6: Wertstrukturierter Bezeichner

Wertstrukturierte Bezeichner Der Zustand eines Wertes läßt sich vollständig in einem wertstrukturierten Bezeichner ablegen (Abbildung 3.6). Die repräsentierten Instanzen werden nicht als tatsächlich allozierte Objekte im Speicher abgelegt, da jeweils ein gesamter Objektzustand im Bezeichner enthalten ist; damit besteht keine Notwendigkeit, zusätzlich explizite Speicherrepräsentationen für die Objekte vorzusehen.

Die in den Bezeichnern enthaltenen Informationen bestehen aus dem Typ und dem Zustand der Objekte (in diesem Beispiel dem Typ `INTEGER` und dem Zustand `5`). Bei strukturierten Bezeichnern für Instanzen von Basistypen kann oft auf die Typinformation verzichtet werden, da der Typ implizit aus dem repräsentierten Zustand selbst hervorgeht; so wäre in diesem Beispiel die Identitätsrepräsentation `<5>` für die ganze Zahl 5 ausreichend.

Typstrukturierte Bezeichner Typstrukturierte Bezeichner enthalten Informationen über den Typ der Objekte, die sie repräsentieren (Abbildung 3.7). Genauso wie bei Adressen mit Markierungsbits können diese Informationen aus den Bezeichnern ausgelesen werden, ohne das dazugehörige Datenobjekt zu dereferenzieren. Auch gilt hier die Einschränkung, daß sich die in

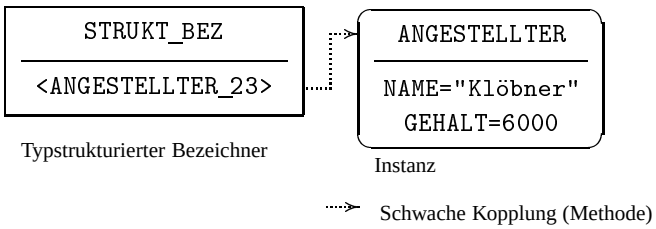


Abbildung 3.7: Typstrukturierter Bezeichner

den Bezeichnern enthaltenen Informationen über die Objekte nicht ändern dürfen, da sonst die in allen ein Objekt referenzierenden Bezeichnern enthaltenen Informationen ungültig werden; in diesem Beispiel darf sich also der Typ eines Objektes nicht ändern.

Außer dem Typ enthält der Bezeichner Informationen zur Lokalisierung des Objektes (in diesem Beispiel die Information 23), die durch die Zugriffsmethoden in eine direkte Referenz auf das Objekt umgewandelt wird (Abbildung 3.8).

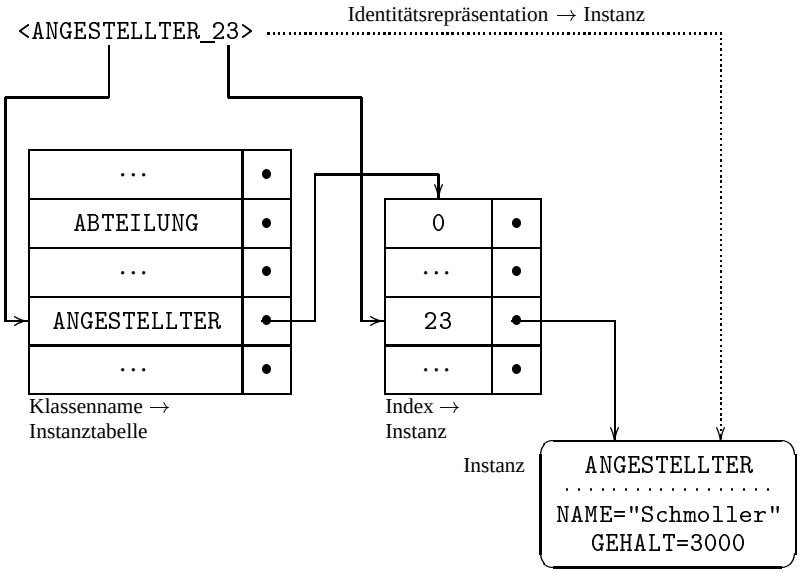


Abbildung 3.8: Abbildung eines typstrukturierten Bezeichners auf ein Datenobjekt

Ortsstrukturierte Bezeichner Die Identitätsrepräsentation mit ortsstrukturierten Bezeichnern (Abbildung 3.9) kann in verteilten Systemen verwendet werden; sie entspricht vom Konzept her einer Erweiterung der indirekten Adressierung. Ein *Client* kann über einen strukturierten Bezeichner eine Referenz auf ein Objekt herstellen, das sich auf einem *Server* befindet. Hier entspricht ein strukturierter Bezeichner dem bei der indirekten Adressierung verwendeten *Forwarder*. Er besteht aus einer zur Umgebung der Instanz (d.h. dem *Server*) lokalen Identitätsrepräsentation und einer Information, mit der die Umgebung des Objektes bestimmt werden kann. Damit können vom *Client* ‚externe' Objekte referenziert werden.

Eindeutigkeit, Unabhängigkeit vom aktuellen Objektzustand

Beides muß für strukturierte Bezeichner durch die gewählte Realisierung sichergestellt werden.

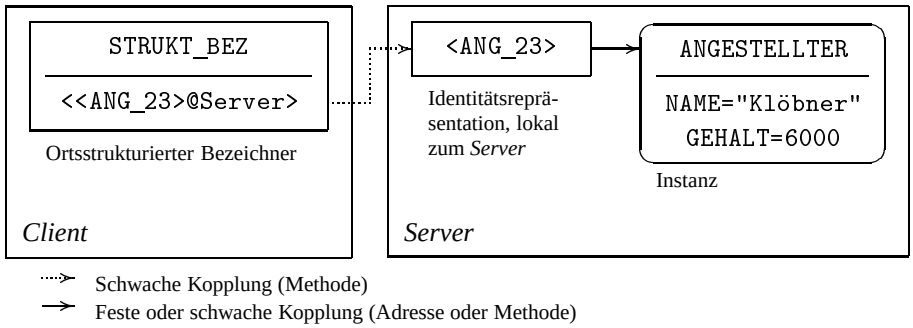


Abbildung 3.9: Ortsstrukturierter Bezeichner

Umgebungsunabhängigkeit, Zeitliche Unabhängigkeit

Wertstrukturierte Bezeichner enthalten implizit das Objekt, das sie repräsentieren; sie bilden eine abgeschlossene atomare Einheit. Zur‚Materialisierung' des repräsentierten Objektes ist nur die Anwendung einer Methode über den Bezeichner notwendig, die ohne weitere Seiteneffekte auskommt; damit sind wertstrukturierte Bezeichner unabhängig von Umgebung und Zeit.

Bei typ- und ortsstrukturierten Bezeichnern muß Umgebungs- und Zeitunabhängigkeit durch die gewählte Realisierung erfolgen.

Effizienz, kanonische Form

Strukturierte Bezeichner können für einfache, kleine Objekte sehr groß werden, insbesondere wenn eine kanonische Repräsentation verwendet wird. Für den Zugriff auf den Objektzustand müssen aus dem Bezeichner die für die Lokalisierung notwendigen Informationen durch eine Methode ermittelt werden; damit sind strukturierte Bezeichner ineffektiver als Adressen.

Informationsgehalt

Die Absicht bei der Verwendung von strukturierten Bezeichnern ist die Ablage von möglichst viel Informationen über ein Objekt in seiner Identitätsrepräsentation.

Spätes Materialisieren

Anders als bei Adressen gibt es bei Bezeichnern keine‚feste' Kopplung zwischen Identitätsrepräsentation und dem Speicherplatz, den das repräsentierte Datum belegt, sondern Bezeichner bilden eine ‚schwache' Kopplung an das Datenobjekt über Methoden; daher ließe sich in diese Methoden ein spätes Materialisieren integrieren. Die für die Materialisierung notwendigen Informationen müssen aus dem Bezeichner ableitbar sein.

Bewertung

Die Auflösung einer Referenz über einen strukturierten Bezeichner bedingt seine Interpretation auf einer relativ hohen Abstraktionsebene. Damit sind strukturierte Bezeichner ineffizienter als Adressen, bieten aber durch eine geeignet realisierte Interpretation die Möglichkeit der Unabhängigkeit von Umgebung und Zeit und sind damit abhängig von ihrer Realisierung für persistente Objekte geeignet. Da sich die in ihnen enthaltenen Informationen nicht ändern dürfen,

sind bei ihrer Verwendung in Common LISP gewisse Restriktionen zu beachten; so sollten z.B. keine typstrukturierten Bezeichner für die Repräsentation der Identität von CLOS-Objekten verwendet werden, da die Klasse einer CLOS-Instanz geändert werden kann.

3.2.3 Benutzer-definierte Bezeichner

> Ich habe keinen Namen
> Dafür! [...]
> Name ist Schall und Rauch,
> Umnebelnd Himmelsglut.
>
> — Goethe: Faust

Wie bereits bei den wertstrukturierten Bezeichnern gezeigt, kann der gesamte Zustand eines konstanten Datenobjektes zur Identitätsrepräsentation genutzt werden. Bei nicht-konstanten Datenobjekten gibt es eine vergleichbare Möglichkeit, indem zur Identitätsrepräsentation ein Teilzustand des Datenobjektes ausgewählt wird und dafür implizit angenommen wird, daß er konstant bleibt. Der Teilzustand wird im allgemeinen so gewählt, daß er genau einen Bezeichner enthält, der zur Identitätsrepräsentation des gesamten Datenobjektes benutzt wird. Anders als bei den wertstrukturierten Bezeichnern gibt es zu jeder Identitätsrepräsentation ein explizit vorhandenes Datenobjekt.

In rein relationalen Datenbanken wird aus externer Sicht diese Art der Identitätsrepräsentation verwendet; aus einem Tupel wird von den BenutzerInnen als Teilzustand ein Attribut ausgewählt, dessen Wert die Identität des Tupels repräsentiert (Abbildung 3.10). Rein relationa-

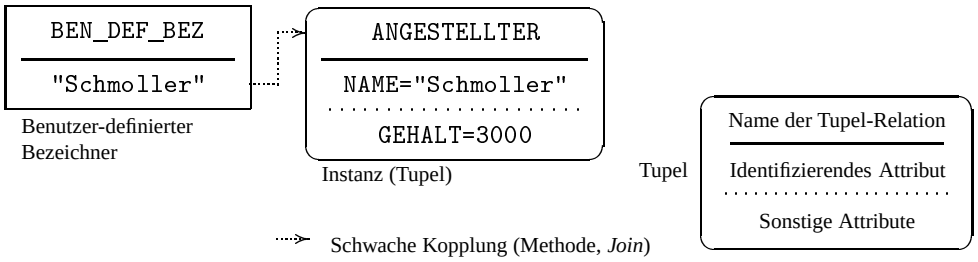


Abbildung 3.10: Benutzer-definierte Bezeichner

le Datenbanken haben keine ‚eingebaute' Repräsentation für Identität der in ihnen enthaltenen Daten.

Eine genauere Betrachtung von Identität und Identitätsrepräsentation in relationalen Datenbanken folgt in Abschnitt 3.4.

Eindeutigkeit

Der Wert dieses Attributes wird von den BenutzerInnen vergeben; damit müssen sie auch die Eindeutigkeit sicherstellen.

Unabhängigkeit vom aktuellen Objektzustand

Der Wert des ausgewählten Attributes ist ein Teilzustand des Tupels und korrespondiert mit einem Merkmalswert des durch den Tupel modellierten realen Objektes; beiÄnderungen dieses Teilzustands ändert sich damit auch die Repräsentation für die Identität des Tupels.

Zudem ist die Gültigkeit des Wertes als Identitätsrepräsentation von Tupeln auf eine Relation eingeschränkt; zur Benutzung des Wertes als Identitätsrepräsentation müssen die BenutzerInnen diesen Gültigkeitsbereich kennen.

Umgebungsunabhängigkeit, Zeitliche Unabhängigkeit

Identitätsrepräsentierende Bezeichner sind unabhängig vom Ort, an dem sich die repräsentierten Objekte befinden. Sie bleiben auch über einen beliebig langen Zeitraum gültig.

Effizienz

Referenzen zwischen Tupeln werden von den BenutzerInnen über *Joins* aufgelöst; die Effizienz dieser *Joins* wird in erster Linie durch den in der relationalen Datenbank realisierten Anfragen-Optimierer (*query optimizer*) bestimmt. Die BenutzerInnen können die Datenbank veranlassen, die Werte der identitätsrepräsentierenden Attribute geordnet abzulegen, um einen schnelleren assoziativen Zugriff zu ermöglichen.

Spätes Materialisieren

Zwischen Bezeichner und Tupel besteht eine schwache Kopplung; damit kann prinzipiell eine späte Materialisierung erfolgen. Der Zeitpunkt wird durch die Realisation innerhalb der Datenbank bestimmt.

Kanonische Form, Informationsgehalt

Inwieweit diese Forderungen erfüllt werden, hängt von der durch die BenutzerInnen gewählten Form der Bezeichner ab; die sich daraus ergebenen Informationen über die repräsentierten Objekte stehen ausschließlich den BenutzerInnen (und nicht der Datenbank) zur Verfügung.

Bewertung

Die durch benutzer-definierte Bezeichner realisierten Eigenschaften und Möglichkeiten stehen ausschließlich den BenutzerInnen zur Verfügung; das persistente System repräsentiert zwar die Bezeichner, hat aber kein Wissen um ihre Bedeutung. Dieses fehlende Wissen wirkt sich unter anderem darin aus, daß Referenzen zwischen Objekten nicht innerhalb der Datenbank aufgelöst werden können; damit lassen sich bestimmte, für persistente Objektsysteme wichtige Konzepte, wie Traversieren der von einem Objekt referenzierten Instanzen oder Speicherrückgewinnung über Erreichbarkeit, nicht innerhalb des persistenten Systems realisieren. Im Rahmen dieser Arbeit sind benutzer-definierte Bezeichner daher zur Identitätsrepräsentation ungeeignet.

3.2.4 Tupel-Bezeichner und Tupel-Surrogate

Innerhalb einer relationalen Datenbanken wird für viele Operationen eine Referenzierung von Tupeln benötigt:

Transaktionen Während einer laufenden Transaktion werden die Zustandsänderungen an Tupeln festgehalten, um bei Abbruch einer Transaktion den Zustand der geänderten Tupel wieder auf den Stand bei Beginn der Transaktion zurückzusetzen (*rollback*). Aus dieser Protokollierung müssen sich daher Referenzen auf die modifizierten Tupel aufbauen lassen.

***Update*, Selektion, Aggregate**[3] Während der internen Verarbeitung durch eine Datenbank werden in einer unteren Schicht aus der Datenbank Tupel ausgewählt und in einer höheren Schicht verarbeitet (z.B. Iteration über diese Tupel mitÄndern, Ausgeben oder Aufsummieren von Attributwerten). Schreibende Zugriffe auf Tupel sind nur möglich, wenn die Datenbank eine Referenz auf die in ihr enthaltenen Tupel aufbauen kann.

Projektion Für schreibenden Zugriff müssen die Referenzen auf die zu einer Projektion gehörigen Tupel in der Datenbank abgelegt werden.

Damit muß die Identität von Tupeln innerhalb der Datenbank repräsentiert werden. Realisiert wird die Identitätsrepräsentation in einigen Datenbanken durch generierte Tupel-Bezeichner, die innerhalb einer Relation eindeutig sind; durch die Einschränkung der Eindeutigkeit auf eine Relation sind sie nicht unabhängig von ihrer Umgebung.

Tupel-Surrogate unterscheiden sich von Tupel-Bezeichnern darin, daß sie über alle Relationen eindeutig sind. In POSTGRES werden beispielsweise Tupel durch eineindeutige Tupel-Surrogate (*IID*) identifiziert [Stonebraker et al. 86a, S. 18].

Tupel-Bezeichner und -Surrogate werden ausschließlich innerhalb der Datenbank verwendet; den Datenbank-BenutzerInnen ist diese Identitätsrepräsentation nicht zugänglich. Zudem wird nur die Identität von Tupeln repräsentiert; für die in den Attributen befindlichen Instanzen gibt es keine Identitätsrepräsentation.

Bewertung

Tupel-Bezeichner und Tupel-Surrogate sind sehr spezielle, für relationale Datenbanken geeignete Identitätsrepräsentationen. Im Rahmen dieser Arbeit wird eine Identitätsrepräsentation benötigt, die für alle in einem Objektsystem auftretenden Instanzen benutzt werden kann; da Tupel-Bezeichner und -Surrogate lediglich die Identität von Tupeln repräsentieren, sind sie damit für den hier beabsichtigten Zweck nicht geeignet.

3.2.5 Surrogate

Surrogate sind globale Identitätsrepräsentationen, die selbst keine direkten Informationen über die jeweils repräsentierten Instanzen enthalten (Abbildung 3.11).

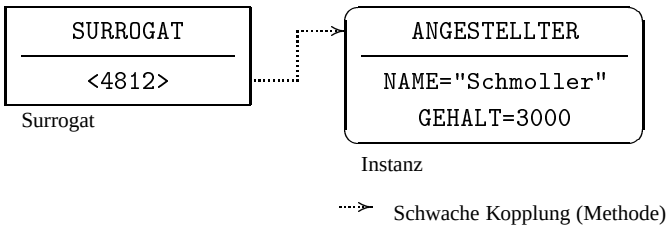


Abbildung 3.11: Surrogate

[3] Ein Aggregat ist eine Funktion über alle Werte eines Attributs einer Relation, z.B. Summenbildung, Mittelwert.

Eindeutigkeit, Unabhängigkeit vom aktuellen Zustand, Umgebungsunabhängigkeit, Zeitliche Unabhängigkeit

Diese Bedingungen können durch eine Realisierung relativ leicht eingehalten werden, indem die Surrogate z.B. von einem globalen Prozeß generiert werden.

Kanonische Form

Eine nicht-kanonische Form ist bei allen Identitätsrepräsentationen ein Hinweis darauf, daß die Form selbst bereits direkte Informationen über das repräsentierte Objekt enthält; daraus folgt für Surrogate, daß sie eine kanonische Form haben müssen.

Informationsgehalt

In Surrogaten selbst sind keine Informationen über die repräsentierten Objekte abgelegt; eine Realisierung muß daher diese Informationen z.B. in internen Tabellen halten, die über Surrogate indiziert werden und als Ergebnis der Indizierung die benötigten Objektinformationen ergeben.

Das Fehlen jeglicher Information über Objekte in Surrogaten hat den Vorteil, daß ein Surrogat immer seine Gültigkeit als Identitätsrepräsentation behält;Änderungen von Objektinformationen wirken sich nicht auf Surrogate aus.

Effizienz

Der in Surrogaten fehlende direkte Informationsgehalt hat folgende Auswirkungen:

Zugriffe auf den Objektzustand Surrogate erfordern im Vergleich zu den bisherigen Identitätsrepräsentationen einen höheren Aufwand für Zugriffe auf den Objektzustand.

Ablage von *Immediates* Der Zustand eines *Immediates* läst sich nicht als Information in einem Surrogat ablegen. Eine Realisierung muß daher intern auch für *Immediates* Speicher allozieren. Viele Realisierungen benutzen aus diesem Grund Surrogate nur für explizit repräsentierte Objekte und nicht für *Immediates*.

Identität von *Immediates* Identische *Immediates* müssen auch identische Surrogate erhalten; die Lösung dieses Problems ist nicht-trivial.

Spätes Materialisieren

Wegen der schwachen Kopplung über Methoden zwischen Surrogaten und repräsentierten Objekten ist ein spätes Materialisieren möglich.

Bewertung

Surrogate sind völlig unabhängig von Umgebung, Zeit und Zustand des Objektes, dessen Identität sie repräsentieren; damit sind sie sehr gut als Identitätsrepräsentation für persistente Objekte geeignet. Problematisch ist die Behandlung von *Immediates*; eine konkrete Realisierung kann dieses Problem aber dadurch lösen, daß ‚abgeschwächte' Surrogate mit einer nicht vollständig kanonischen Form realisiert werden.

3.3 Objekt-Identifikation (*objid*)

> [...] Zeichen oder Zeichen von Zeichen benutzen wir nur, solange wir keinen Zugang zu den Dingen selbst haben.
>
> — Umberto Eco: Der Name der Rose

Die Identitätsrepräsentation eines Datenobjektes wird als seine *objid* bezeichnet. In einer Programmiersprache werden Datenobjekte durch unevaluierte Ausdrücke, sogenannte *Handles*, referenziert. Die Evaluierung eines *Handle* hat als Ergebnis das referenzierte Datenobjekt selbst; damit entspricht sie einer Dereferenzierung des sie repräsentierenden Objektes [Heiler et al. 89, S. 238] (Beispiele siehe Tabelle 3.12). Die Identität der als Ergebnis der Evaluierung entste-

In ↓	**ergibt *Handle* ↓**	**bei Evaluierung Referenz auf ↓ ...**
SQL Datenbank	`SELECT A.ALL WHERE A.GEHALT > 2000 & A IN ANGESTELLTER`	... ein (temporäres) Mengenobjekt, das alle Tupelobjekte enthält, die Angestellte mit einem Gehalt größer 2000 repräsentieren
CLOS	`(find-class 'standard-class)`	... das Standard Klassen-Metaobjekt der Klasse **standard-class**
Persistentes CLOS	`(get-object "FOO_15")`	... das Objekt, dessen Identität durch die Zeichenkette `"FOO_15"` repräsentiert wird

Tabelle 3.12: Beispiele für *Handle* und durch Evaluierung referenzierte Objekte

henden Objekte wird durch einen Objektidentifikator (*objid*) repräsentiert. Die Bestimmung der *objid* eines Objektes ist eine bijektive polymorphe Abbildung von der Menge der Objekte O in die Menge der Objekt-Identifikatoren I:

$$\textit{get-objid} : O \rightarrow I \tag{3.2}$$

Je nach Vorgehensweise konstruiert oder generiert diese Abbildung eine Repräsentationsform für die Identität eines Objektes. Die Umkehrabbildung $\textit{get-objid}^{-1}$, im folgenden *get-object* genannt, bildet einen Objektidentifikator auf das dazugehörige Datenobjekt ab:

$$\textit{get-objid}^{-1} \equiv \textit{get-object} : I \rightarrow O \tag{3.3}$$

Da die *objid* die Identität eines Datenobjektes eineindeutig repräsentiert, ergibt sich daraus:

1. Zwei Objekte o_1 und o_2 sind genau dann identisch, wenn ihre jeweiligen *objids* gleich sind:
$$\textit{identical}(o_1, o_2) \leftrightarrow \textit{get-objid}(o_1) = \textit{get-objid}(o_2) \tag{3.4}$$
2. Die *objid* dient als ‚Stellvertreter' für das Objekt selbst und kann benutzt werden, um Referenzen auf ein Objekt aufzubauen. Die Abbildung *get-object* ermöglicht den Zugriff auf den Zustand des Objektes.

3.4 Identität in verschiedenen Systemen

Jedes in einem System existierende Datum hat eine Identität; Programmiersprachen und Systeme unterscheiden sich aber stark in ihrer Auffassung davon, ob diese Identität überhaupt repräsentiert werden soll und wie diese Repräsentation aussieht.

3.4.1 Identität in statisch typisierten Programmiersprachen

GO IN GREEN

Go in green. Go in, case. Go in green.
Go in green. go in. in, go in green.
Go in green.
Go in green.

— Gertrude Stein

In den klassischen Programmiersprachen wie C und PASCAL werden typisierte Variablen zur Repräsentation von Daten benutzt. Eine Variable wird über einen benutzer-definierten Bezeichner (Variablennamen) adressiert. Das Datum selbst enthält keinerlei Informationen über seinen Typ; er wird lediglich an den Bezeichner des Datums gebunden und steht nur während des Übersetzerlaufs zur Verfügung.

Ein Konzept für Datenidentität gibt es in diesen Sprachen nicht; daraus folgt, daß eine Repräsentationsform für die Identität eines Datums ebenso fehlt. Der Variablenname ermöglicht nur die Adressierung eines Datums; dessen Identität wird durch ihn nicht repräsentiert. Bei diesem Ansatz werden die Konzepte von Addressierung und Identität vermischt. Die Adresse eines Datums ist eine externe Eigenschaft, die es ermöglicht, auf dieses Datum zuzugreifen, während die Identität die innere Eigenschaft der Einzigartigkeit eines Datums widerspiegelt. Die Adresse ist zudem abhängig von der Umgebung, in der sich das Datum befindet [Khoshafian et al. 90, S. 38]. Kennzeichnend für diese Sprachen ist das Fehlen eines Operators, mit dem auf Identität geprüft werden kann; lediglich ein Test auf Gleichheit ist möglich (siehe z.B. [Wirth 83, S. 23]). Dieser Test gibt keinerlei Aufschluß darüber, ob zwei Variablen sich auf ein und dasselbe Datenobjekt beziehen, da jeweils nur die Zustände der Datenobjekte verglichen werden (Abbildung 3.13).

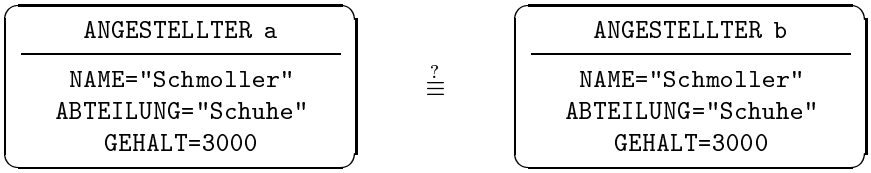


Abbildung 3.13: Identität in ‚klassischen' Programmiersprachen

Oft wird in diesen Sprachen die Adresse eines Datums als seine Identitätsrepräsentation benutzt; damit ist die Identitätsrepräsentation aber nicht in der Sprache selbst realisiert, sondern wird explizit von den BenutzerInnen gewählt. In einigen Sprachen kann auch diese Möglichkeit nicht oder nur eingeschränkt genutzt werden, da sie keine Operatoren zur Ermittlung der Adresse eines Datums anbieten, wie z.B. PASCAL. Die Adresse eines Datums in PASCAL ist nur dann bekannt, wenn für das Datum eine darauf verweisende Zeigervariable deklariert wird und das Datum selbst im *Heap* abgelegt wird; die Adresse des Datums wird bei der Speicherallozііerung in der Zeigervariablen abgelegt. Die Typinformation über das von der Zeigervariablen dereferenzierte Datum ist an den Namen der Zeigervariablen gebunden. Ein Test auf Identität von zwei Daten durch Vergleich ihrer Adressen kann in PASCAL nur erfolgen, wenn beide Daten vom gleichen Typ sind. Ein Identitätstest zwischen Daten verschiedenen Typs ist nicht möglich, da der Übersetzer keinen Vergleich von Zeigervariablen zuläßt, die auf unterschiedliche Typen verweisen.

Durch die Verwendung von Bezeichnern für Objekte in Systemen ohne Repräsentationsfor-

men für Identität tritt noch ein weiteres Problem auf. Auf dasselbe Objekt kann auf verschiedene Art und Weise zugegriffen werden; wird das jeweils durch den Zugriff adressierte Objekt an verschiedene Bezeichner gebunden, ist es bei fehlender Repräsentationsform für Identität nicht möglich festzustellen, ob sich diese Bezeichner auf ein und dasselbe Objekt beziehen. Dieser Effekt wird als *aliasing* bezeichnet.

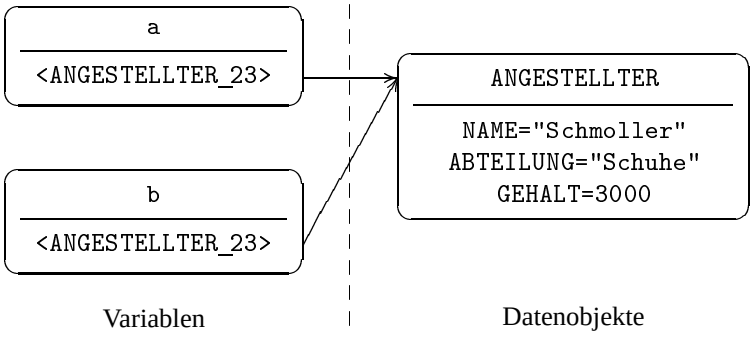


Abbildung 3.14: Identität in objekt-orientierten Programmiersprachen

3.4.2 Identität in dynamisch typisierten Programmiersprachen

In dynamisch typisierten Programmiersprachen werden nicht Variablen, sondern die Datenobjekte selbst typisiert; Variablen dienen lediglich zur Referenzierung der Objekte. Common LISP und viele der objekt-orientierten Programmiersprachen wie Smalltalk-80 und CLOS sind dynamisch typisierte Sprachen.

Die dynamisch typisierten Programmiersprachen benutzen Konstrukte, die Datenidentität berücksichtigen; so repräsentieren diese Systeme die Identitäten der in ihnen enthaltenen Daten. Aus der internen Sicht dieser Sprachen enthält eine Variable keinen direkten Wert, sondern eine Identitätsrepräsentation für ein Datenobjekt, mittels der dann auf den Objektzustand zugegriffen werden kann. Es findet also eine Trennung zwischen dem Objekt als Individuum und seiner Adressierung statt (Abbildung 3.14). Variablen enthalten aus interner Sicht ausschließlich Referenzen auf Datenobjekte und nicht mehr die Objekte selbst wie in den‚klassischen' Programmiersprachen.[4] Aus externer Sicht, also aus der Sicht der BenutzerInnen, wird eine Variable an ein eigenständiges Objekt gebunden; es ergeben sich zweiÄnderungen gegenüber den klassischen Programmiersprachen:

1. Da Objekte Repräsentationsformen für ihre Identität haben und Variablen diese Formen und nicht mehr die Objekte selbst enthalten, kann festgestellt werden, ob sich zwei Variablen auf ein und dasselbe Objekt beziehen; zusätzlich zum Test auf Gleichheit gibt es jetzt noch einen Test auf Identität.

2. Bei einer Zuweisung an eine Variable ändert sich nicht der Zustand des Objektes, sondern die Variable wird an das zugewiesene Objekt gebunden; die vorherige Bindung wird aufgehoben. Der Zugriff auf den Objektzustand erfolgt durch Aufrufe entsprechender Zugriffsfunktionen durch die BenutzerInnen.

[4] Natürlich gibt es auch hier Ausnahmen aus Effizienzgründen, wie z.B. Objekte der Klasse **SmallInt** in Smalltalk-80 oder Instanzen der Klasse **fixnum** in CLOS.

3.4.3 Identität in Common LISP und CLOS

Common LISP und CLOS repräsentieren die Identität von Datenobjekten. In [CLtLII] wird zwar keine bestimmte Identitätsrepräsentation vorgeschrieben, es wird aber angenommen, daß die Identität eines Objektes über seine Adresse repräsentiert wird [CLtLII, S. 103, 104]. Aus Effizienzgründen wird für die Realisierung eines Common LISP-Systems empfohlen, Werte der Klassen **short-float** [CLtLII, S. 22] und **character** [CLtLII, S. 371] als *Immediates* anzusehen; daraus folgt eine nicht-kanonische Identitätsrepräsentation. Für Werte der Klasse **fixnum** wird ebenfalls angenommen, daß sie effizienter als Instanzen der allgemeineren Klasse **integer** repräsentiert werden [CLtLII, S. 16]. Numerische Werte können ebenfalls nicht-kanonisch repräsentiert werden [CLtLII, S. 288]. Die meisten Common LISP-Systeme halten sich an diese Empfehlungen; in dem zur Realisierung von persistenten Objektsystemen in dieser Arbeit verwendetem LISPWORKS Common LISP werden Instanzen der Klassen **short-float**, **character** und **fixnum** als *Immediates* angesehen, deren Identität nicht-kanonisch repräsentiert wird; bei allen anderen Objekten wird deren jeweilige Identität über ihre Adresse repräsentiert.

3.4.4 Identität in relationalen Datenbanken

Die in relationalen Datenbanken enthaltenen persistenten Datenobjekte sind wie nicht-persistente Daten Dinge oder Sachverhalte der realen Welt in abstrakter Form. Bei der Modellbildung werden reale Dinge abstrahiert und anhand gleichartiger Merkmale klassifiziert; jede gefundene Klasse wird in der Datenbank als Relation abgelegt. Die Merkmale der realen Dinge einer Klasse werden als Attribute der Relation definiert. Ein reales Objekt wird in der Datenbank als Tupel der zu seiner Klasse gehörigen Relation abgelegt.

Relationale Datenbanken lassen sich in rein relationale, erweiterte und objekt-orientierte Datenbanken einteilen. In rein relationalen und erweiterten relationalen Datenbanken wird eine Relation als Tabelle verstanden, bei der für jedes Attribut eine Spalte existiert; ein Tupel ist eine Zeile einer solchen Tabelle. In die objekt-orientierten Datenbanken wurden Konzepte der objekt-orientierten Programmierung integriert; so werden Relationen als vererbbare Klassen, Tupel als Instanzen mit Identität und Attribute als *Slots* aufgefaßt. Das bedeutet jedoch nicht, daß sie in der Lage sind, Datenobjekte im Sinne der objekt-orientierten Programmierung (wie z.B. CLOS-Objekte) direkt zu speichern.

Rein relationale Datenbanken

Sie erfüllen die von Codd [Codd 79, S. 399] aufgestellte Bedingung, daß ein Tupel sich in mindestens einem Attributwert von allen anderen Tupeln der gleichen Relation unterscheidet [O'Neil 94, S. 37]; damit gilt für jedes Tupel das Identitätsprinzip (es wird als individuelles, unterscheidbares Datenobjekt aufgefaßt). Trotz dieser Auffassung von Datenidentität fehlt in den rein relationalen Datenbanken nach Codd eine datenbank-interne Identitätsrepräsentation; statt dessen wurde dafür eine Konvention definiert, die die Herstellung von Identität den Datenbank-BenutzerInnen überläßt (Abbildung 3.15).

Identitätsrepräsentation in relationalen Datenbanken Die Identitätsrepräsentation in relationalen Datenbanken besteht daraus, daß innerhalb einer Relation von den BenutzerInnen ein oder mehrere Attribute ausgewählt werden, deren Werte die Identität eines Tupels widerspiegeln (Primärschlüsselattribute); sie bilden somit die *objid* des Tupels. Aus den Bedingungen für Identitätsrepräsentation folgt, daß die beteiligten Attribute ihre Werte

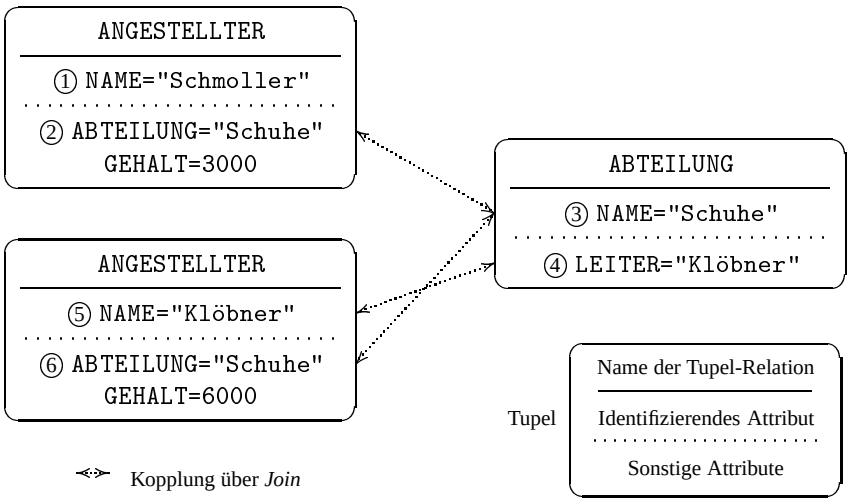


Abbildung 3.15: Identitäten und Referenzen in relationalen Datenbanken

innerhalb eines Tupels nicht ändern dürfen und mindestens innerhalb der Relation eindeutig sein müssen. Die korrekte Vergabe der Attributwerte liegt vollständig in der Hand der BenutzerInnen.

Die BenutzerInnen der Relation, die diese *objid* für den Aufbau von Referenzen ausnutzen möchten, müssen wissen, welche Attribute der Relation für diesen Zweck festgelegt wurden (Abbildung 3.15, Attribute ①, ③ und ⑤) [Schlageter et al. 83, S. 84].

Referenzen zwischen Datenobjekten Referenzen zwischen Datenobjekten werden aufgebaut, indem der Wert des identitäts-repräsentierenden Attributes des dereferenzierten Tupels in ein Attribut im referenzierenden Tupel eingetragen wird. Im referenzierenden Tupel wird dieses Attribut per Konvention von den BenutzerInnen nicht als‚wertetragend‘, sondern als ‚dereferenzierend‘ festgelegt (Sekundärschlüssel oder Fremdschlüsselattribute, Abbildung 3.15, Referenzen zwischen ② ↔ ③, ③ ↔ ⑥ und ④ ↔ ⑤).

Auflösung der Referenzen Die Auflösung der Referenzen erfolgt über Selektion des dereferenzierten Objektes über das identitätsrepräsentierende Attribut. Dazu ist der Aufbau eines *Joins* zwischen referenzierendem und dereferenziertem Tupel nötig. Die Referenz ist symmetrisch; da der *Join* kommutativ ist [Schlageter et al. 83, S. 85, 174], kann durch Umstrukturierung ebenso eine Referenz in Gegenrichtung aufgelöst werden.

Die in Abbildung 3.15 eingezeichneten Referenzen zwischen den Objekten sind nicht explizit in der Datenbank enthalten.

Die korrekte Verwaltung sowohl der identitäts-repräsentierenden Attribute als auch der Referenzen, d.h. der Schlüsselintegrität, liegt in der Hand der BenutzerInnen, da eine rein relationale Datenbank kein ‚Wissen‘ über diese getroffenen Konventionen hat.

Das Problem der Identitätsrepräsentation über einen Attributwert ist seine doppelte Bedeutung. Zum einen dient er als *objid* und zum anderen ist er ein Wert innerhalb des Tupels. Die Lösung innerhalb einer rein relationalen Datenbank besteht darin, diese beiden Bedeutungen aufzutrennen und für die Identitätsrepräsentation ein zusätzliches Attribut **OBJID** festzulegen. Beim Anlegen eines neuen Tupels muß es dann von den BenutzerInnen mit einem beliebigen aber innerhalb der Relation eindeutigen Wert belegt werden. Der gewählte Wert kann mit einem

Merkmal des im Tupel modellierten realen Objektes korrespondieren, sofern dieses Merkmal die Bedingungen für Identitätsrepräsentation einhält (z.B. Personalausweisnummer als *objid* für Tupel der Relation **ANGESTELLTER**).

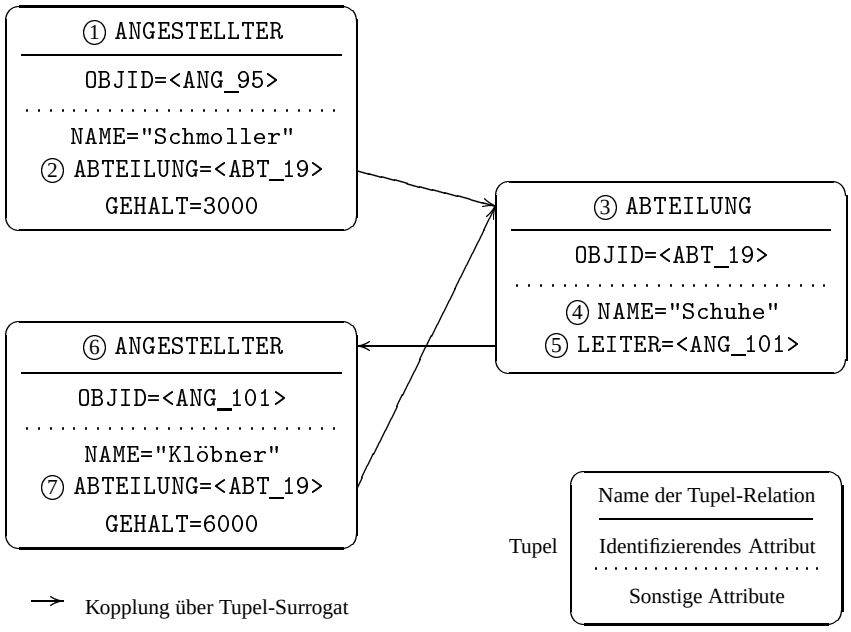


Abbildung 3.16: Identitäten und Referenzen in objekt-orientierten relationalen Datenbanken

Erweiterte relationale Datenbanken

Als mögliches Hilfskonstrukt bieten sie den BenutzerInnen zur Einhaltung der Bedingungen für eine Identitätsrepräsentation sogenannte *Trigger* [O'Neil 94, S. 407] [Valduriez et al. 89, S. 49] (Synonym *Alerter* [Nikhil 88, S. 53]) an. Ein *Trigger* ist eine Prozedur, die bei Eintritt eines Ereignisses oder Erfüllung einer Vorbedingung aktiviert wird (vergleichbar dem *Dämon*-Konzept). Er kann beispielsweise benutzt werden, um beim Ereignis ‚Einfügen eines Tupels' die dazugehörige *objid* zu generieren und in das identitäts-repräsentierende Attribut **OBJID** einzutragen; ebenso kann er bei Zugriff auf Attribute aktiviert werden und dabei die Gültigkeit von Referenzen überwachen.

Bei neueren Systemen gibt es die Möglichkeit, Integritätsbedingungen zu spezifizieren, die von der relationalen Datenbank überwacht werden; damit kann beispielsweise die Schlüsselintegrität automatisch überprüft werden. Problematisch ist aber, daß die entsprechenden Bedingungen auch für den ‚einfachen' Fall von Referenzen zwischen Objekten spezifiziert werden müssen, obwohl die Bedingung der Schlüsselintegrität bereits inhärent in der Referenz enthalten ist.

Objekt-orientierte relationale Datenbanken

Sie erweitern das Typkonzept und lassen Datenbank-Relationen als Typen für Attribute zu (Abbildung 3.16). Aus der Sicht der BenutzerInnen ist der Wert eines derartigen Attributes ein Tupel der typbildenden Relation; intern wird ähnlich wie bei Variablen in objekt-orientierten Programmiersprachen das Tupel durch seine *objid* repräsentiert. Dadurch kann eine direkte Referenz von

einem Attribut auf ein Tupel aufgebaut werden. Die in Abbildung 3.16 eingezeichneten Referenzen werden durch die Datenbank und nicht durch die BenutzerInnen realisiert; für den Zugriff ist daher auch kein *Join* mehr nötig. Die Referenz kann häufig nur in einer Richtung aufgelöst werden.

Als referenzierbare Objekte werden in einigen Systemen ausschließlich Tupel angesehen (z.B. POSTGRES [Stonebraker et al. 86a]). Die Identität aller anderen in der Datenbank enthaltenen Objekte wird in diesen Systemen nicht repräsentiert; damit sind keine Referenzen auf diese Daten möglich.

3.4.5 Identität in persistenten Objektsystemen

Entitäten sollten nicht unnötig vervielfacht werden.

— ‚Ockhams Rasiermesser'

In persistenten Systemen, die auf Maschinen mit flüchtigem Speicher realisiert werden, wird zur Ablage von persistenten Objekten ein Sekundärspeicher benutzt. Nach Erzeugung eines Objektes im flüchtigen Speicher legt das persistente System das Objekt entweder explizit (z.B. durch Veranlassung von außen, wie Anwendung einer Methode durch die BenutzerInnen, die dafür sorgt, daß das Objekt persistent wird) oder implizit (z.B. wird für jedes neu erzeugte Objekt angenommen, daß es persistent sein soll) im nicht-flüchtigen Speicher ab. Bei Zugriffen auf die Objekte von einem späteren oder anderen Prozeß können die referenzierten persistenten Objekte aus Effizienzgründen vom persistenten System aus der persistenten in die transiente Umgebung kopiert werden und dort direkt von den BenutzerInnen manipuliert werden; in diesem Fall gibt es von einem Datum zwei ‚Ausgaben', eine im flüchtigen Speicher und eine im nicht-flüchtigen Speicher (Abbildung 3.17). Für die BenutzerInnen der obersten Schicht eines

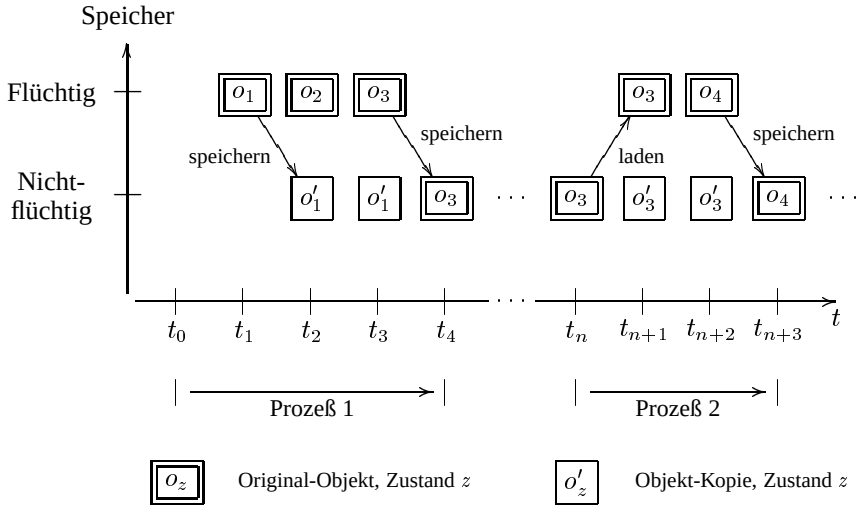


Abbildung 3.17: Identität eines Objektes in einem persistenten System

persistenten Systems sollte diese Dualität nicht sichtbar sein, sondern sie arbeiten lediglich mit dem einen Objekt, das sie direkt im Zugriff haben; das persistente System sorgt für die Verwaltung des Objektes, wie Transfer zwischen flüchtigem und nicht-flüchtigem Speicher, Regelung von Zugriffskonflikten von mehreren BenutzerInnen auf ein Objekt usw.

Die Identität eines Objektes ist damit abhängig von der Schicht eines persistenten Systems. In der obersten Schicht gibt es für jedes Objekt eine Identität; in den darunter liegenden Schichten existiert das Objekt in einer oder mehreren zusammengehörigen‚Ausgaben‘, denen man in diesen Schichten eine eigene Identität geben kann. Bei Transfer eines Objektes zwischen den Schichten müssen dann die jeweiligen Identitätsrepräsentationen umgeformt werden.

3.5 Zusammenfassung

Identität ist eine jedem Datenobjekt zugehörige Eigenschaft. Sie zeichnet ein Datenobjekt als eigenständig aus, d.h. es unterscheidet sich von allen anderen Datenobjekten. Dies muß in einem System von Datenobjekten durch eine geeignete Repräsentationsform wiedergegeben werden. Sie dient dazu, Referenzen auf das dazugehörige Objekt aufzubauen. Eine Repräsentationsform muß dabei bestimmten Bedingungen genügen; sie muß eindeutig und unabhängig von Zeit, aktuellem Zustand und der Umgebung des Objektes sein, dessen Identität repräsentiert werden soll. Für transiente und persistente Umgebungen sind die in diesem Kapitel diskutierten Identitätsrepräsentationen unterschiedlich geeignet. CLOS-Systeme repräsentieren die Identität der in ihnen enthaltenen Objekte im allgemeinen mit nicht-kanonischen Adressen, deren Verwendung als Identitätsrepräsentation in einer persistenten Umgebung damit ausscheidet. Für persistente Objekte muß daher eine geeignete Identitätsrepräsentation innerhalb der persistenten Umgebung ausgewählt und realisiert werden; zu diesem Zweck sind Surrogate sehr gut geeignet, da sie die Bedingungen für eine Identitätsrepräsentationform am Besten erfüllen. Sofern ein persistentes Objekt seine Umgebung wechseln kann, muß bei jedem Wechsel unter anderem seine Identitätsrepräsentation konvertiert werden; bei der Verwendung von Surrogaten bedeutet dies konkret, daß beim Wechsel von persistenten Objekten zwischen der persistenten und der transienten Umgebung unter anderem eine Konvertierung zwischen Surrogaten und Adressen erfolgen muß.

Kapitel 4

Architekturkonzepte von persistenten Objektsystemen

> Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.
>
> — A. J. Perlis: Epigrams on Programming

In diesem Kapitel werden verschiedene Architekturkonzepte von persistenten Objektsystemen erläutert. Die verschiedenen Möglichkeiten zum Gesamtaufbau eines persistenten Systems werden erklärt und es werden die letztendlich im weiteren Verlauf der Arbeit betrachteten Architekturen von persistenten Objektsystemen in CLOS festgelegt; für diese Architekturen werden verschiedene, im Zusammenhang mit Common LISP wichtige Konzepte vorgestellt.

4.1 Systemarchitektur

Für die Systemarchitektur von persistenten Systemen gibt es je nach Anforderung verschiedene Realisierungsansätze.

4.1.1 Verwendung von getrennten Subsystemen

Der oft verwendete Weg zur Herstellung von Persistenz besteht darin, innerhalb einer Applikation zusätzlich zu einem transienten Objektsystem einen persistenten Speicher einzubinden (Abbildung 4.1). Dabei werden für die Komponenten der untersten Schicht bereits vorhandene Systeme verwendet. Die komplette Verwaltung der persistenten Datenobjekte liegt in der Applikationsschicht. Dort müssen die beiden Subsysteme integriert werden und erfordern im allgemeinen für jede einzelne Applikation eine aufwendige spezielle Lösung, insbesondere da sich oft die Konzepte von Objektsystem und Applikation einerseits und persistentem Speicher andererseits nicht decken.

Der Vorteil dieser Lösung ist die schnelle Verfügbarkeit der verwendeten unteren Komponenten, da sie bereits vorhanden sind. Die Qualität der in der Applikationsschicht vorzunehmenden Integration der beiden Subsysteme ist in erster Linie von der Erfahrung des Applikationser-

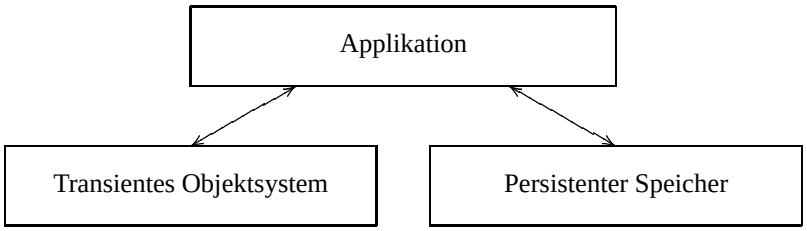


Abbildung 4.1: Aufbau eines persistenten Systems mit getrennten Subsystemen

stellers abhängig. Ebenso ist sie sowohl auf die jeweilige Applikation als auch auf das für den persistenten Speicher verwendete Subsystem spezialisiert.

Beispiele für diese Art von persistenten Systemen sind die Verwendung von C++ oder CLOS als transientes Objektsystem und einer Datei oder Datenbank als persistenten Speicher. In Programmiersprachen eingebettete Datenbanksprachen (*embedded database languages*) fallen ebenfalls unter diese Kategorie; sie bieten aber bereits eine teilweise Integration in die verwendete Programmiersprache an.

In den meisten unter UNIX realisierten Common LISP-Systemen gibt es als Hilfsmittel für Persistenz die Möglichkeit, den aktuellen Zustand des laufenden Common LISP-Prozesses in einer (meist sogar direkt ausführbaren) Datei zu sichern (*workspace dump*); in dieser Datei befinden sich sämtliche Objekte des gesicherten Prozesses im Zustand zum Zeitpunkt der Sicherung. Diese Datei kann dann zu einem späteren Zeitpunkt wieder geladen werden; damit werden auch die gesicherten Objekte wiederhergestellt. Diese Art der Sicherung umfaßt immer alle im laufenden Prozeß vorhandene Objekte; einzelne Speicherungen sind nicht möglich. Selektionsmöglichkeiten über Objekte in dieser Datei sind nicht gegeben.

Die durch Verwendung von getrennten Subsystemen erreichte Persistenz ist weder orthogonal noch transparent; Funktionalitäten über persistente Objekte müssen für jede Applikation neu erstellt werden. Insgesamt gesehen scheidet damit diese Art der Persistenz für den im Rahmen dieser Arbeit beabsichtigten Zweck aus.

4.1.2 Vollständig integrierte Systeme

Vollständig integrierte Systeme (Abbildung 4.2) bieten Persistenz auf einem hohen Niveau. Eine auf einem solchen System erstellte Applikation ist von oberer Sicht aus nicht auf spezielle Hilfsmittel für persistente Objekte angewiessen. Die Verwaltung der persistenten Objekte be-

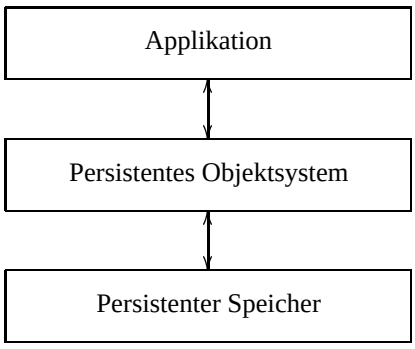


Abbildung 4.2: Aufbau eines vollständig integrierten persistenten Systems

findet sich nicht mehr wie beim ersten Ansatz in der Applikationsschicht, sondern wurde in die unteren beiden Schichten verlagert. Damit entfällt auch die Notwendigkeit, für jede Applikation eine spezielle Lösung zu erstellen.

Ein Nachteil ist der hohe Aufwand für die Erstellung eines vollständig persistenten Objektsystems. Eine effiziente Realisierung wird im allgemeinen auch nur dann erreicht, wenn auf Persistenz spezialisierte Rechnerarchitekturen verwendet werden.

Um den Aufwand für die Erstellung eines solchen Systems zu verringern, könnte beispielsweise ein transientes Objektsystem für eine Rechnerarchitektur mit nicht-flüchtigem Speicher (wie CMOS-RAM mit autarker Stromversorgung) angepaßt werden. Ebenso ließe sich in ein transientes System ein persistenter Objektspeicher integrieren (siehe z.B. [Müller 91, S. 12]). Da viele der bisher realisierten Objektsysteme nicht für Persistenz ausgelegt sind, ist ein optimales System nur durch einen kompletten Neuentwurf möglich.

Damit scheidet dieser Ansatz ebenfalls aus, da ein bereits vorhandenes CLOS-System um Persistenz erweitert werden soll; die vollständige Integration von Persistenz wie in [Müller 91, S. 12] geschildert ist hier nicht durchführbar, da ein derartiger Ansatz die Modifikation der unteren Schichten des Systems voraussetzt, in das Persistenz integriert werden soll, und dies für das vorgegebene CLOS-System nicht möglich ist.

4.1.3 Teilweise integrierte Systeme

Teilweise integrierte Systeme verbinden die beiden ersten Ansätze. Sie bestehen aus einem persistenten Objektsystem, das seinerseits wieder auf einem transienten Objektsystem beruht. Als Hilfsmittel zur Herstellung von Persistenz wird ein nicht-flüchtiger Speicher benutzt (Abbildung 4.3). Das persistente Objektsystem ist kein eigenständiges System, sondern erweitert das

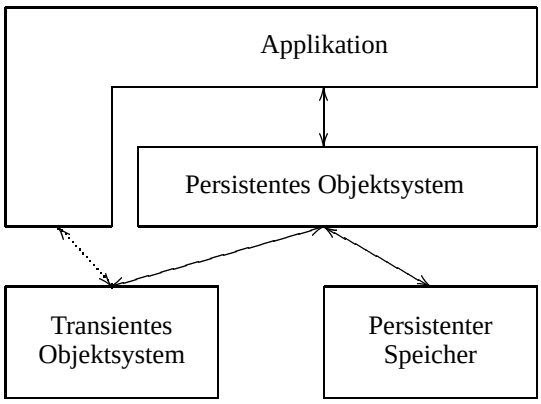


Abbildung 4.3: Aufbau eines teilweise integrierten persistenten Systems

vorhandene transiente System. Deswegen kann für flüchtige Objekte von einer Applikation weiterhin direkt das transiente Objektsystem verwendet werden. Ebenso stehen die im transienten Objektsystem realisierten Konzepte auch für persistente Objekte zur Verfügung. Die Verwaltung der persistenten Objekte erfolgt wie beim zweiten Ansatz nicht in der Applikationsschicht, sondern im persistenten Objektsystem.

Es ergeben sich für teilweise integrierte Systeme die gleichen Vorteile wie bei den vollständig integrierten Systemen. Zusätzlich läßt sich der Aufwand für die Realisierung eines solchen Systems relativ klein halten, da man für das transiente Objektsystem und den persistenten Speicher bereits vorhandene Komponenten verwenden kann. Um eine hohe Orthogonalität

zwischen dem transienten und persistenten Objektsystem zu erhalten, wird das persistente System meist in der gleichen Programmiersprache wie das transiente System realisiert.

Alle in dieser Arbeit betrachteten Systeme beruhen auf diesem Ansatz. Als transientes Objektsystem wird CLOS verwendet. Das persistente Objektsystem ist ebenfalls in CLOS realisiert; einige Systeme benutzen hier zusätzlich in anderen Programmiersprachen verfaßte Subsysteme. Der persistente Speicher ist je nach System unterschiedlich; einige benutzen komplexe Subsysteme, wie relationale und objekt-orientierte relationale Datenbanken (wie beispielsweise PCLOS [Paepke 91a] und SOH [Rowe 87]), andere verwenden Objektspeicher und einfache Dateien (wie beispielsweise WOOD [St. Clair 93] und POB! [S. 81]).

4.2 Persistenter Speicher

Die im Rahmen dieser Arbeit betrachteten persistenten Systeme verwenden entweder eine relationale Datenbank oder einen Objektspeicher als persistenten Speicher.

4.2.1 Relationale Datenbank als persistenter Speicher

Relationale Datenbanken sind im Bereich der Speicherung und Manipulation von Daten sehr mächtig; viele Realisierungen von persistenten Objektsystemen vertreten daher den Ansatz, als persistenten Speicher eine relationale Datenbank zu verwenden (Abbildung 4.4).

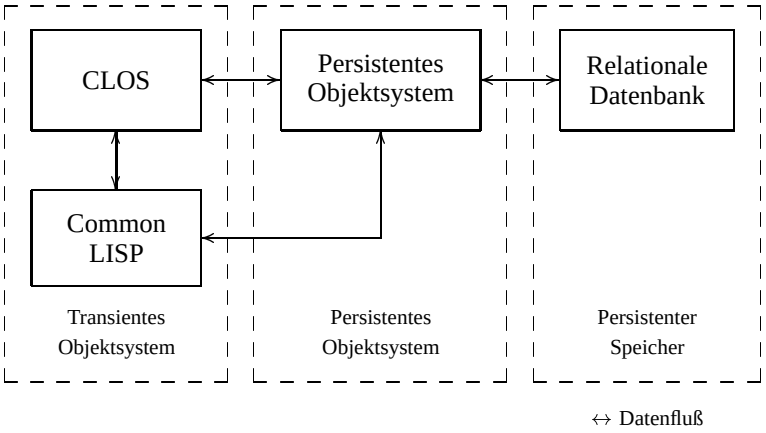


Abbildung 4.4: Persistentes Objektsystem mit relationaler Datenbank

Im Rest dieses Abschnittes werden die Argumente für und gegen diese Vorgehensweise diskutiert. Die hier getroffenen Aussagen sind generell gehalten; auf spezielle Aspekte wird später in diesem Kapitel eingegangen.

Einfache Abbildung zwischen Objektsystem- und Datenbank-Konzepten

Die Hoffnung bei dieser Vorgehensweise ist zum einen, die im transienten Objektsystem realisierten Konzepte möglichst einfach auf entsprechende Datenbank-Konzepte abbilden zu können und zum anderen die von der Datenbank angebotenen Möglichkeiten auch für das persistente Objektsystem verfügbar zu machen.

Die Konzepte von CLOS einerseits und relationalen Datenbanken andererseits unterscheiden sich stark. Aus den in Abschnitt 2.1 (S. 7) geschilderten Eigenheiten einer relationalen

Datenbank folgt, daß die Übertragung des Objektmodells von CLOS auf das relationale Datenmodell Schwierigkeiten bieten wird, sofern orthogonale Persistenz angestrebt wird.

Reale Entitäten werden in relationalen Datenbanken zu Relationen und in CLOS zu Klassen abstrahiert; damit ergeben sich die in Tabelle 4.5 dargestellten Zusammenhänge. Der *Slot*-

Objektsystem	**Relationale Datenbank**
Klasse	Relation
Instanz	Tupel
Effektiver *Slot*	Attribut

Tabelle 4.5: Datenstrukturen in Objektsystem und Datenbank

Zustand eines CLOS-Objektes kann einen Wert oder ein dynamisch typisiertes Objekt referenzieren; in relationalen Datenbanken ist nur genau ein einfacher, auf Werte festgelegter Bereich für ein Attribut erlaubt. Deswegen sollte der *Slot*-Zustand so in einem Attribut repräsentiert werden, daß aus der Repräsentation dynamisch der Typ des repräsentierten Objektes hervorgeht und so unter anderem eine Unterscheidung zwischen referenziertem Wert und referenzierter Instanz möglich wird. Eine derartige Repräsentationsform kann dazu führen, daß der Zustand des repräsentierten Objektes so konvertiert werden muß, daß er von der relationalen Datenbank in seiner ursprünglichen Bedeutung nicht mehr erfaßt werden kann und deswegen ihre Manipulationsmöglichkeiten für persistente Objekte nicht vernünftig genutzt werden können. Dies kann nur vermieden werden, wenn der Objektzustand selbst und keine aus einer Konvertierung hervorgangene Repräsentation dafür in der Datenbank abgelegt wird. Da Attribute statisch auf einfache Bereiche festgelegt werden müssen, wirkt sich dies auf persistente Objekte rückwirkend dahingehend aus, daß ihre *Slots* ebenfalls statisch auf einen einfachen Basistyp typisiert werden müssen, der zu einem einfachen Bereich des Datenbanksystems korrespondiert; damit geht allerdings die Orthogonalität des persistenten Systems verloren.

Eine Abbildung der Konzepte ist entweder nur mit sehr hohem Aufwand möglich oder zieht Einschränkungen nach sich, die für einzelne Applikationen dazu führen können, daß die Verwendung eines derartig realisierten persistenten Objektsystems ausscheidet [Brössler et al. 89, S. 305] [Nikhil 88, S. 52].

Verlagerung des *impedance mismatch*

In jeder Applikation, die eine relationale Datenbank direkt benutzt, muß ein Teil des Codes nur erstellt werden, um den *impedance mismatch* zwischen Applikation einerseits und relationaler Datenbank andererseits auszugleichen. Bei der Verwendung eines persistenten Objektsystems unter Benutzung einer relationalen Datenbank als persistenten Speicher wird der *impedance mismatch* von allen Applikationen in das persistente Objektsystem verlagert, so daß er für die BenutzerInnen des persistenten Systems weitestgehend verschwindet; lediglich die AutorInnen des persistenten Objektsystems müssen sich um eine Lösung des Problems bemühen.

Eine hinreichend generelle Lösung dieses Problems innerhalb des persistenten Objektsystems muß sowohl definiert als auch (möglichst effizient) realisiert werden; der Aufwand dafür kann sehr hoch werden.

Effizienz

Die Entwicklung von relationalen Datenbanken ist weit fortgeschritten; inzwischen sind relationale Datenbanken erhältlich, die zum einen der Definition nach Codd genügen und außerdem

über effiziente Möglichkeiten der Daten-Haltung und -Manipulation verfügen. Damit wird zum einen die Realisierung eines persistenten Objektsystems beschleunigt (da eine vorhandene Datenbank benutzt wird) und zum anderen können die persistenten Instanzen effektiv manipuliert werden.

Dies funktioniert nur, wenn die oben genannten Einschränkungen akzeptabel sind, die aus der Abbildung zwischen CLOS- und Datenbank-Konzepten resultieren.

4.2.2 Objektspeicher und Datei als persistenter Speicher

Ein Objektspeicher bietet Persistenz auf ‚niedrigem' Niveau (Abbildung 4.6). Er ist lediglich

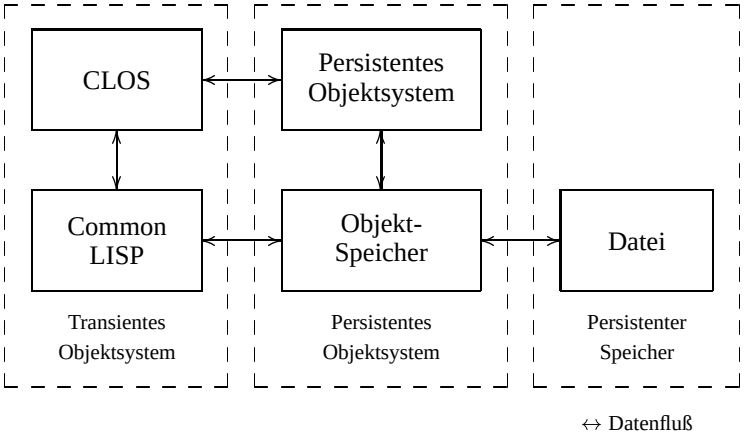


Abbildung 4.6: Persistentes Objektsystem mit Objektspeicher

in der Lage, Common LISP-Daten in einer Datei abzulegen und auf diese mit im Vergleich zu einer relationalen Datenbank sehr bescheidenen Konstrukten zuzugreifen. Die Möglichkeit zur Typbildung wie in relationalen Datenbanken ist nicht vorhanden; die im Objektspeicher enthaltenen Daten sind auf Objektspeicherebene nicht typisiert. Ein Objektspeicher wird oft auch als persistenter *Heap* bezeichnet, da er ähnlich wie ein transienter *Heap* realisiert wird.

Referenzen zwischen Objekten

Innerhalb des Objektspeichers werden Objekte durch Datensätze repräsentiert. Dem Objektspeicher wird mitgeteilt, welche Bestandteile eines Datensatzes als Referenz auf (andere) Datensätze und welche als Werte zu interpretieren sind; die Aufteilung in Referenzen und Werte wird aus der Strukturbeschreibung des Objektes abgeleitet. Damit sind wie in transienten Objektsystemen Referenzen zwischen denen durch die Datensätze repräsentierten Objekte innerhalb des Objektspeichers möglich. Durch die explizite Repräsentation von Referenzen genügt ein Objektspeicher nicht dem relationalen Modell.

Effizienz

Der Aufwand für die Konvertierung eines Objektzustands in eine Form, die vom Objektspeicher verarbeitet werden kann, ist im allgemeinen gering. Objektzustände können teilweise unverändert zwischen Objektspeicher und persistentem Objektsystem ausgetauscht werden; lediglich für Referenzen müssen Konvertierungen vorgenommen werden.

Die im Objektspeicher abgelegten Daten werden bis auf die Referenzen nicht weiter interpretiert oder besonders verarbeitet.

Wahl der Strukturen im Objektspeicher

Da die Zustände der Datensätze vom Objektspeicher nicht weiter interpretiert werden, lassen sie sich mit beliebigen Strukturen anlegen; insbesondere können die Strukturen so gewählt werden, daß sie sich an dem transientem System orientieren, das den Objektspeicher benutzt. Damit kann der *impedance mismatch* bezüglich der Datenrepräsentation zwischen CLOS und Objektspeicher vermieden werden, da für Zustände innerhalb des Objektspeichers die jeweilige Repräsentation des Common LISP-Systems verwendet werden kann. Eine direkteÜbernahme der Zustände bedeutet aber, daß ebenfalls die Interpretation der Zustände, so wie sie im verwendeten Common LISP-System festgelegt ist, implizit mit übernommen wird; wenn an den persistenten Speicher verschiedene persistente Objektsysteme angebunden werden sollen, muß die jeweilige Interpretation des Zustands den verschiedenen Systemen bekannt sein.

Speicherrückgewinnung

Da Objektspeicher Referenzen zwischen den in ihnen enthaltenen Datensätzen ermöglichen, läßt sich innerhalb eines Objektspeichers eine auf Erreichbarkeit basierende Speicherrückgewinnung realisieren (siehe Abschnitt 4.3.2).

Typfreiheit der Daten innerhalb des Objektspeichers

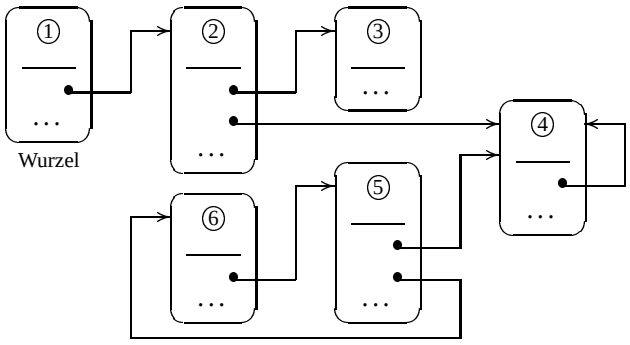
Ein Objektspeicher schreibt dem benutzenden System keine bestimmten Typisierungskonzepte vor. Die fehlenden Konzepte können auf einer höheren Ebene (hier im persistenten Objektsystem) realisiert werden, die dann dem benutzenden System (hier CLOS) angepaßt werden können.

4.3 Lebensdauer von Objekten

Die Definition der Lebensdauer von Objekten erfolgt analog zu der Festlegung in Common LISP [CLtLII, S. 43], daß jedes Objekt eine unendlich lange Lebensdauer hat. Wenn das (persistente) System beweisen kann, daß ein Objekt nicht transitiv von einem Wurzelobjekt aus erreichbar ist, darf es gelöscht werden (Abbildung 4.7). Ein erreichbares Objekt‚wird noch gebraucht‘ und darf vom System nicht gelöscht werden.

4.3.1 Lebensdauer in relationalen Datenbanken

Innerhalb einer rein relationalen Datenbank kann die Lebensdauer von Daten nicht über ihre Erreichbarkeit definiert werden, da die Datenbank die in ihr enthaltenen Referenzen zwischen den Tupeln nicht explizit repräsentiert. Das Löschen von Tupeln in relationalen Datenbanken ist nur über explizite Aufrufe entsprechender Funktionen durch die BenutzerInnen möglich; zusätzlich muß dabei von den BenutzerInnen sichergestellt werden, daß keine ungültigen Referenzen (*dangling references*) auf bereits gelöschte Objekte entstehen. Beispielsweise können in dem auf einer relationalen Datenbank basierendem persistenten Objektsystem PCLOS Objekte explizit gelöscht werden, das System überprüft aber nicht, ob auf das gelöschte Objekt noch Referenzen existieren [Paepke 91a, S. 18].



①: Wurzelobjekt
②, ③, ④: Erreichbare Objekte ⑤, ⑥: Nicht erreichbare Objekte

Abbildung 4.7: Erreichbarkeit von Objekten (nach [Thatte 90, S. 245])

4.3.2 Speicherrückgewinnung

Don: I didn't know you had a cousin Penelope, Bill! Was she pretty?
W. C.: Well, her face was so wrinkled it looked like seven miles of bad road. She had so many gold teeth, Don, she use *(sic!)* to have to sleep with her head in a safe. She died in Bolivia.
Don: Oh Bill, it must be hard to lose a relative.
W. C.: It's almost impossible.

— W. C. Fields: „The Further Adventures of Larson E. Whipsnade and other Tarradiddles"

Die Speicherrückgewinnung hat die Aufgabe, nicht erreichbare Objekte zu finden und den von ihnen allozierten Speicherplatz freizugeben. Aktiviert wird sie durch Eintritt einer Vorbedingung, wie z.B. bei einem bestimmten Füllungsgrad des Speichers oder während Freizeiten (*idle times*) des Systems. Damit ein System feststellen kann, ob ein Objekt erreichbar ist oder nicht und damit gelöscht werden kann, muß es Referenzen zwischen den in ihm enthaltenen Objekten explizit repräsentieren können. Bei der Begrenzung der Lebensdauer von Objekten über ihre Erreichbarkeit sollte zur Vermeidung von inkonsistenten Referenzen auf eine nach außen zur Verfügung gestellte explizite Löschoperation für Objekte verzichtet werden; ein Objekt wird durch nicht-referenzieren für das Objektsystem als zu löschendes Objekt gekennzeichnet.

Eine Speicherrückgewinnung für persistente Objekte muß innerhalb des persistenten Objektsystems oder des persistenten Speichers realisiert werden, da im allgemeinen die Speicherrückgewinnung des transienten Objektsystems nicht für persistente Objekte genutzt werden kann.

Die in dieser Arbeit betrachteten persistenten Objektsysteme haben entweder keine Speicherrückgewinnung oder realisieren sie ebenfalls mittels Erreichbarkeit. Realisierte Vorgehensweisen für Rückgewinnung von persistentem Speicher lassen sich nach dem Zeitpunkt ihres Geschehens in zwei Gruppen einteilen. Bei der ersten Gruppe findet eine Rückgewinnung von persistentem Speicher zur *on-line* Zeit statt, während der der persistente Speicher durch ein persistentes Objektsystem benutzt wird. Die zweite Gruppe führt eine Rückgewinnung zur *off-line* Zeit durch, während der der persistente Speicher nicht genutzt wird.

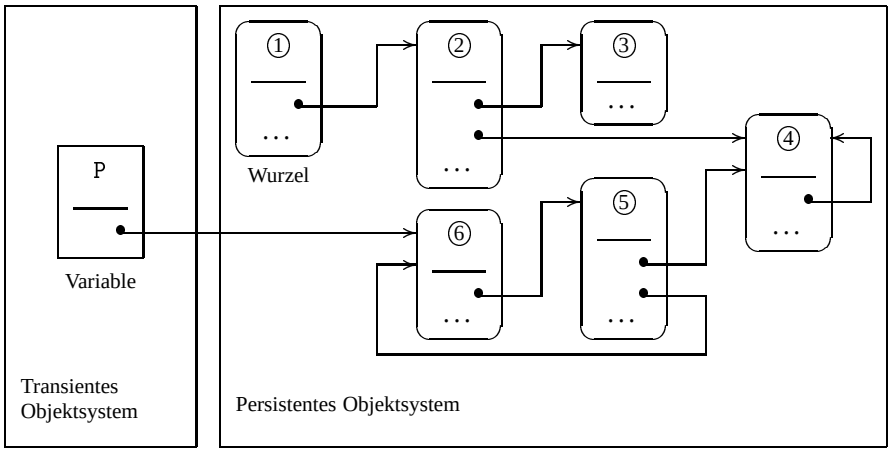
4.3.3 Keine Rückgewinnung von persistentem Speicher

Das Fehlen einer Speicherrückgewinnung ist oft durch die Konzepte des zur Realisierung verwendeten persistenten Speichers begründet. Bei Verwendung einer relationalen Datenbank als persistenten Speicher stellt sich das Problem, daß die Datenbank keine Referenzen der in ihr enthaltenen Daten explizit repräsentieren kann; damit müssen Referenzen zwischen persistenten Objekten auf einer höheren Ebene innerhalb des persistenten Objektsystems realisiert werden. Dafür werden Konstrukte gewählt, die zwar sehr effizient ein persistentes Objekt mit allen von ihm referenzierten Objekten materialisieren können, bei denen aber die Ermittlung der von einem persistenten Objekt lediglich referenzierten Objekte eben genau diese Materialisierung bedingt. Eine Rückgewinnung von persistentem Speicher wurde aus diesen Gründen bei den in dieser Arbeit betrachteten Systemen, die eine Datenbank als persistenten Speicher nutzen, nicht realisiert.

4.3.4 Rückgewinnung von persistentem Speicher zur *on-line* Zeit

Eine Rückgewinnung zur Laufzeit einer Applikation hat den Vorteil einer hohen Verfügbarkeit des persistenten Speichers, da es keine Ausfallzeiten gibt. Nicht erreichbare Objekte werden noch zur Laufzeit als freier Speicher erkannt; damit wird eine gute Ausnutzung des persistenten Speichers erreicht.

Persistente Objektsysteme lassen aus Effizienzgründen oft direkte Referenzen von Variablen oder Objekten des transienten Systems auf persistente Objekte zu; damit kann es bei einer Speicherrückgewinnung zu Problemen kommen, wenn ein persistentes Objekt zwar von einer transienten Variablen bzw. einem transienten Objekt referenziert wird, diese Rerefenz aber nicht im Wurzelobjekt ‚eingetragen' ist (Abbildung 4.8). Eine vom persistenten Objektsystem durch-



①: Wurzelobjekt
②, ③, ④: Erreichbare Objekte ⑤, ⑥: Von der transienten Variable `P` erreichbare Objekte

Abbildung 4.8: Erreichbarkeit eines persistenten Objektes über eine Variable des transienten Systems

geführte Rückgewinnung von persistentem Speicher würde in diesem Beispiel die Objekte ⑤ und ⑥ als nicht erreichbare Objekte markieren und als freien Speicher erneut zur Verfügung stellen, obwohl sie von der (transienten) Variablen `P` referenziert werden.

Zur Vermeidung einer solchen Situation muß das persistente Objektsystem eine Konvention oder ein Protokoll für seine Speicherrückgewinnung festlegen. Eine Konvention könnte so aussehen, daß solche Referenzen unberücksichtigt bleiben. Ein Protokoll würde z.B. festlegen, daß die Speicherrückgewinnung explizit und synchron durch die BenutzerInnen aktiviert wird; in diesem Beispiel führt eine Rückgewinnung von persistentem Speicher zur *on-line* Zeit durch das von den BenutzerInen einzuhaltende Protokoll zu einer verringert transparenten Persistenz.

4.3.5 Rückgewinnung von persistentem Speicher zur *off-line* Zeit

Diese Art der Speicherrückgewinnung kann nur dann durchgeführt werden, wenn keine Applikation den persistenten Speicher benutzt. Dies setzt voraus, daß es solche Zeiten überhaupt gibt. Außerdem ist der persistente Speicher während der Speicherrückgewinnung nicht verfügbar.

Nicht erreichbare Objekte werden zur Laufzeit einer Applikation nicht als freier persistenter Speicher zur Verfügung gestellt; damit kann ein Überlauf des persistenten Speichers auftreten, obwohl genügend nicht erreichbare Objekte vorhanden sein könnten.

Vorteilhaft ist, daß bei dieser Art der Speicherrückgewinnung keine Probleme mit Referenzen des transienten Systems auf persistente Objekte auftreten können; ein Protokoll zwischen transientem und persistentem Objektsystem ist nicht notwendig. Der Zeitpunkt der Speicherrückgewinnung kann so gelegt werden, daß die zur Verfügung stehenden Ressourcen besser oder intensiver genutzt werden können. Insbesondere verbraucht sie dann keine Ressourcen, die ansonsten einer Applikation zugute kommen würden. Damit kann sie neben ihrer eigentlichen Aufgabe auch Optimierungen innerhalb des persistenten Speichers vornehmen, wie z.B. nahe Gruppierung von Objekten bei ihren referenzierten Objekten (*clustering*).

4.4 Typisierung der *objid*

Für *Immediates* kann eine *objid* den gesamten Objektzustand selbst enthalten. Für alle anderen Objekte existiert eine *objid* nur in Verbindung mit einem Datenobjekt, da sie ausschließlich zur Identitätsrepräsentation des dazugehörigen Objektes dient. In beiden Fällen hat eine *objid* selbst den Charakter eines Wertes,[1] allerdings keines ‚gewöhnlichen' Wertes im Sinne einer Instanz der in einem Objektsystem angebotenen Typen oder Klassen, sondern eine *objid* wird als Repräsentation für die Identität eines Datums interpretiert. Diese Eigenschaft sollte sich soweit möglich in der tatsächlich gewählten Repräsentation für *objids* selbst wiederfinden. Im Sinne eines Objektsystems mit Identitätsrepräsentation sollte damit der Typ für *objids*‚neben' allen anderen Typen bzw. Klassen stehen. In [Beech 87, S. 325] wird der Benutzung von *objids* die Bedeutung einer ‚Meta-Ebene' (*metalevel*) zugewiesen.

Bei Realisierung eines persistenten Systems mittels eines vorhandenen transienten Systems ist man für die Wahl einer geeigneten Repräsentation für *objids* selbst auf die Möglichkeiten des transienten Systems angewiesen. Ein solches System bietet zum einen‚eingebaute' Basistypen und -klassen, deren Instanzen z.T. Werte oder *Immediates* sind, und zum anderen die Möglichkeit, ‚höhere' Typen oder Klassen zu definieren.

Kennzeichnend für Basisklassen, deren Instanzen als Werte oder *Immediates* angesehen werden, ist, daß jede von ihnen für ihre Instanzen eine eigene, sehr spezielle Datenrepräsentation hat. Ebenso fehlt den Basisklassen im Gegensatz zu den‚höheren' Klassen die Möglichkeit zur

[1] Als Identitätsrepräsentation nehme man beispielsweise die Speicheradresse eines Objektes an. Der erste Fall entspricht der Situation, daß ein *Immediate* keine Adresse hat. Im zweiten Fall wird ausgesagt, daß nur existierende Objekte eine Adresse haben und daß diese Adresse selbst keine Adresse hat.

Subklassenbildung. Vorteilhaft ist die hohe Effizienz ihrer Instanzen bezüglich Speicherplatz und Verarbeitung. Bei allen ‚höheren' Klassen hingegen werden einander ähnliche Instanzrepräsentationen verwendet, bei denen ein Teil allen Instanzen aller Klassen gemeinsam ist und der restliche Teil aus der Stellung der jeweiligen Klasse in der Klassenhierarchie abgeleitet wird.

Objids sollten möglichst effizient realisiert werden. Damit bietet es sich an, den Typ für *objids* innerhalb des persistenten Objektsystems aus den bereits vorhandenen Basistypen für Werte oder *Immediates* des transienten Systems abzuleiten. Da eine Subklassenbildung nicht in Frage kommt, wird aus den Basistypen eine möglichst einfache Struktur als Typ für *objids* festgelegt. Die aus diesem Typ gebildeten *objids* müssen für das Objektsystem als solche erkennbar sein, d.h. sie müssen sich entweder von ‚normalen' Werten von Basistypen unterscheiden oder aus dem Kontext, in dem sich der Wert befindet, muß hervorgehen, daß er als *objid* oder als ‚normaler' Wert verstanden werden soll. Wenn als Typ für *objids* z.B. Zeichenketten verwendet werden (siehe auch letztes Beispiel in Tabelle 3.12, S. 26), muß entweder aus einer Zeichenkette hervorgehen, ob sie als ‚normale' Zeichenkette oder als *objid* verstanden werden soll oder die Zeichenkette muß sich in einem Kontext befinden, in dem ihre Bedeutung festgelegt ist.

4.5 Strukturbeschreibungen

Um den Zustand eines Objektes im persistenten Speicher abzulegen, muß die Struktur des Objektes bekannt sein. Für möglichst transparente Persistenz sollte diese Information nicht beim Ablegen des Objektes im persistenten Speicher für jede Instanz explizit angegeben werden müssen, sondern sie sollte aus der die Struktur des Objektes definierenden Klasse bestimmt werden. Bei Realisierung eines persistenten Systems, das auf einem vorhandenem transientem System beruht, sollten daher Informationen über die Klasse der persistent zu haltenden Objekte verfügbar sein. Programmiersysteme lassen sich nach Verfügbarkeit und Art von Strukturbeschreibungen einteilen.

4.5.1 Keine Verfügbarkeit der Strukturbeschreibungen

In den klassischen Programmiersprachen wie PASCAL und C sowie den darauf zum Teil basierenden objekt-orientierten Sprachen wie C++ und Eiffel werden Strukturbeschreibungen ausschließlich vom Übersetzer (*compiler*) generiert und benutzt. Zur Laufzeit kommen diese transienten Systeme mit sehr marginalen Typmerkmalen aus, wie z.B. der Größe der Instanzen für Allozierung von Objekten auf dem *Heap*; weitere Informationen sind nicht notwendig und werden demzufolge auch nicht repräsentiert.

4.5.2 Statische Strukturbeschreibungen

Statische Strukturbeschreibungen werden während derÜbersetzung erzeugt. Sie werden hauptsächlich in Objektsystemen verwendet, die eine der oben genannten klassischen Sprachen mit typisierten Variablen und nur eingeschränkt typisierten Datenobjekten um Persistenz erweitern.

Einige Übersetzer bieten die Möglichkeit, die intern erzeugten Strukturbeschreibungen ‚nach außen hin' verfügbar zu machen und zusätzlich die Datenobjekte zu typisieren. Die generierten Strukturbeschreibungen werden dann zur Laufzeit vom persistenten Objektsystem eingelesen und ermöglichen in Verbindung mit der Typisierung der Datenobjekte eine Ermittlung ihrer Strukturbeschreibung und damit eine Zerlegung der Instanzen in ihre Komponenten. Ein

Beispiel für diese Vorgehensweise ist das Texas-System [Singhal et al. 93, S. 147], das von GNU C++ erzeugte Typinformationen und typisierte Datenobjekte benutzt.

Diese Erweiterung ermöglicht zwar typisierte Datenobjekte und Strukturbeschreibungen, gleichzeitig treten damit aber Inkonsistenzen mit der ursprünglich definierten Sprachsemantik auf. Der Typ eines Objektes geht in diesem Fall nicht mehr wie bisher aus der Variablendeklaration zur Übersetzungszeit, sondern aus der Typinformation des Datenobjektes zur Laufzeit hervor; diese Bedeutungsänderung wird von solchen Erweiterungen im allgemeinen nicht berücksichtigt.

Andere auf den oben genannten Sprachen basierende persistente Objektsysteme verwenden (Vor-)Übersetzer (*(pre-)compiler*), die die benötigten Strukturbeschreibungen aus dem Quellcode ermitteln. Zusätzlich modifizieren oder erweitern sie den Quellcode für den nachfolgenden Übersetzerlauf so, daß die in diesen Sprachen ursprünglich fehlende Laufzeitunterstützung für Persistenz durch das persistente Objektsystem möglich wird. Beispiele für diese Art von Persistenz sind *E* [Richardson et al. 89, S. 175, 181] und OBST [Uhl et al. 93].

Kennzeichnend bei diesen Systemen ist, daß die statischen Strukturbeschreibungen nicht in die verwendete Programmiersprache integriert sind; in vielen persistenten Systemen werden die Strukturbeschreibungen auch nicht zur Laufzeit explizit benutzt oder repräsentiert, sondern finden sich lediglich implizit wieder, z.B. durch von einem Vorübersetzer vorgenommeneÄnderungen oder Erweiterungen des Quellcodes.

Die so erreichte Persistenz verlangt eine strenge Disziplin beim Erstellen von Quellcodes, die das persistente Objektsystem benutzen, da sich aus der Verwendung von statisch typisierten Variablenbezeichern zur Repräsentation von dynamisch typisierten Datenobjekten Widersprüche oder Versionskonflikte ergeben können. Je nach verwendeter Programmiersprache dürfen einige Konstrukte für persistente Daten nicht benutzt werden, da sie die für das persistente Objektsystem ermittelten statischen Typ-Informationen invalidieren würden (z.B. Typwandlungen (*type casts*) und variante Verbunde (*unions*) in derartig um Persistenz erweitertem C++; diese Konstrukte ermöglichen es einem Objekt, bei statisch ermittelter Typinformation den Typ zur Laufzeit zu wechseln und sind damit widersprüchlich).

4.5.3 Dynamische Strukturbeschreibungen

In Systemen mit typisierten Datenobjekten werden oft auch ihre Klassen explizit repräsentiert, da sie zur Laufzeit die Realisierung von Sprachkonzepten (wie Polymorphie,Änderungen innerhalb der Klassenhierarchie zur Laufzeit) unterstützen oder sonstige Vorteile bieten, wie z.B. einfachere Introspektionsmöglichkeiten für Datenobjekte. Ein persistentes Objektsystem kann damit die vom transienten System verwalteten Typinformationen benutzen, um den Zustand von Instanzen in seine Bestandteile zu zerlegen.

4.5.4 Strukturbeschreibungen von Common LISP Daten

In reinem Common LISP sind zwar die Datenobjekte typisiert, im Gegensatz zu CLOS liegen aber keine expliziten Strukturbeschreibungen von Common LISP Daten vor, sondern sie sind in das jeweilige System ‚eingebaut'. Direkte Informationen über diese Strukturbeschreibungen sind damit im Common LISP-System nicht vorhanden; dies bedeutet, daß sie anderweitig beschafft werden müssen, z.B. durch entsprechende Common LISP Spezifikationen wie in [CLtLII], Auswertung der Funktionen für den Zugriff auf den Objektzustand, Offenlegung der Strukturbeschreibungen durch die AutorInnen des Common LISP-Systems oder, schlimmstenfalls, durch Untersuchung einer der verwendeten Repräsentationsformen von Instanzen eines

Typs mit Ableitung der benötigten Informationen daraus. Beispiele für verschiedene Common LISP Daten und ihre Strukturbeschreibungen sind:

***cons*-Zelle** Eine *cons*-Zelle besteht aus zwei Referenzen *car* und *cdr*.

Symbol Ein Symbol besteht aus einer Referenz auf seine externe Repräsentation (Namen), einer Referenz auf seinen Wert, einer Referenz auf sein Funktionsobjekt und einer Referenz auf seine *Property*-Liste.

Felder Fast alle relevanten Informationen über Felder (*arrays*) können über Feld-Informationsfunktionen (*array information functions*) erfragt werden; aus den von diesen Funktionen zurückgegebenen Werten kann daher implizit eine Strukturbeschreibung für Felder generiert werden.

Die derartig erlangten Strukturbeschreibungen von reinen Common LISP Daten sind statisch; da nicht zu erwarten ist, daß sich die in [CLtLII] gegebenen Definitionen von Basistypen signifikant ändern werden, sind keine Inkompatibilitäten der Sturkturbeschreibungen über die Zeit zu befürchten.

4.5.5 Strukturbeschreibungen von CLOS-Instanzen

Die Klassen von CLOS-Instanzen werden zur Laufzeit durch Klassen-Metaobjekte repräsentiert, von denen alle notwendigen Informationen für die Ablage ihrer Instanzen im Objektspeicher erhalten werden können. Klassen-Metaobjekte entsprechen dynamischen Strukturbeschreibungen.

4.6 Zusammenfassung

Für die Integration von Persistenz in CLOS gibt es die beiden Ansätze, als persistenten Speicher eine relationale Datenbank oder einen Objektspeicher zu verwenden.

Bei Verwendung einer relationalen Datenbank erhält das persistente System bei entsprechender Realisierung mächtige Möglichkeiten zur Datenmanipulation; bei der Realisierung eines persistenten Systems ist aber zu beachten, daß die von der Datenbank vorgegebene Datenmodellierung insbesondere durch die statische Festlegung der Attribute auf einfache Bereiche und die fehlende explizite Repräsentation von Referenzen gegenüber LISP eine Einschränkung darstellt. Das persistente System sollte intern entsprechende Konvertierungen realisieren, die allerdings dazu führen können, daß die Manipulationsmöglichkeiten der Datenbank nur teilweise genutzt werden können.

Objektspeicher ermöglichen für die Realisierung eines persistenten Systems die Übernahme der in Common LISP verwendeten Datenmodellierung in den persistenten Speicher. Ein Objektspeicher bietet im allgemeinen nur einfachste Möglichkeiten zur Datenmanipulation; ein persistentes System sollte je nach Einsatzgebiet zusätzliche Möglichkeiten realisieren.

Um ein transientes Objekt in einem persistentem Speicher ablegen zu können, muß die Struktur des Objektes bekannt sein. In Common LISP sind Strukturbeschreibungen für Objekte der Basisklassen statisch; für CLOS-Instanzen kann aus ihrer Klasse dynamisch eine Strukturbeschreibung bestimmt werden.

Kapitel 5

Programmierkonzepte in persistenten CLOS-Systemen

In diesem Kapitel werden Programmierkonzepte erläutert, die für ein persistentes Objektsystem wichtig sind und dementsprechend auch realisiert werden sollten. Ebenso werden hier Realisierungskonzepte erklärt, die allen auf CLOS basierenden persistenten Objektsystemen gemeinsam sind; die aus diesen Konzepten resultierenden Einschränkungen werden auch diskutiert. Dieser Abschnitt bildet damit die Grundlage für die in den späteren Kapiteln beschriebenen Realisierungen von persistenten Objektsystemen.

Die allgemeinen Konzepte von objekt-orientierten Datenbanken werden sehr ausführlich in [Zdonik et al. 90, S. 1–32] geschildert.

5.1 Der *Cache*

Da der persistente Speicher eines persistenten Objektsystems normalerweise relativ langsam ist, wird ein *Cache* eingesetzt, der die transienten Repräsentationen aller persistenten Objekte referenziert, deren Zustand zwischen transientem und persistentem Speicher übertragen wurde (Abbildung 5.1). Der *Cache* assoziiert zu einer persistenten die transiente Identitätsrepräsentation,

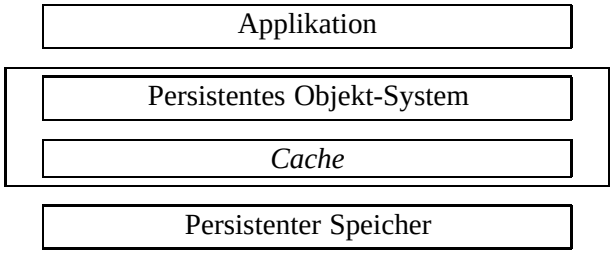


Abbildung 5.1: Schichtenmodell persistentes Objekt-System mit *Cache*

über die dann der Zustand der transienten Repräsentation referenziert werden kann. Oft wird mindestens für Instanzen von Basistypen auch die Umkehrabbildung realisiert.

5.1.1 Effizienzsteigerung

Bei mehrmaligem Zugriff auf den Zustand der transienten Repräsentation eines persistenten Objektes können so Zugriffe auf den persistenten Speicher eingespart werden. Es wird zunächst geprüft, ob die transiente Repräsentation bereits vom Objekt-*Cache* referenziert wird; wenn ja, erfolgt der Zugriff auf den Objektzustand über die aus dem *Cache* gelesene Referenz, andernfalls wird der Zustand des persistenten Objektes in eine neu allozierte transiente Repräsentation kopiert, die Referenz darauf im *Cache* zur *objid* assoziiert und zurückgegeben.

Die Verwendung eines *Cache* kann beim Laden sehr vieler Objekte dazu führen, daß entsprechend viele transiente Repräsentationen alloziert werden und damit entsprechend viel Speicherresourcen verbraucht werden.

5.1.2 Speicherrückgewinnung des transienten Systems

Für jede transiente Repräsentation gibt es jeweils eine Referenz aus dem *Cache*. Dies führt dazu, daß alle vom *Cache* referenzierten transienten Repräsentationen bei einer Speicherrückgewinnung des transienten Systems erreichbar sind und nicht gelöscht werden, auch wenn die Applikation keine Referenzen mehr auf eine solche transiente Repräsentation hat. Um die Speicherrückgewinnung nicht zu blockieren, müssen entweder die Referenzen aus dem *Cache* entfernt werden oder das verwendete Common LISP-System muß sogenannte schwache Referenzen (*weak pointer*) kennen, d.h. Referenzen, die bei der Speicherrückgewinung nicht berücksichtigt werden.

5.2 Datentypen

Common LISP bietet bereits sehr viele verschiedene Basistypen (*built-in types*) an [CLtLII, S. 12–36]; zusätzlich gibt es noch die standardisierten CLOS Klassen sowie systemabängige vordefinierte Klassen.

Der (Gesamt-)Zustand eines Objektes setzt sich aus den in den *Slots* des Objektes enthaltenen (Teil-)Zuständen zusammen. Sie lassen sich in zwei Arten einteilen; sie können entweder einfache Werte oder andere Objekte referenzieren [Schewe et al. 92, S. 7]. Aus der Forderung nach möglichst großer Orthogonalität folgt daraus, daß ein persistentes System beide Arten handhaben können muß.

5.2.1 Werte

Die für diese Arbeit betrachteten Systeme verarbeiten meist nicht alle Werte oder schränken ihre Verwendung ein, z.B. indem Zeichenketten eine bestimmte Länge nicht überschreiten dürfen. Bei Systemen, die eine relationale Datenbank als persistenten Speicher verwenden, leiten sich die von diesen Systemen handhabbaren Werte oft direkt aus den in der relationalen Datenbank angebotenen Basistypen ab.

5.2.2 Objekte und Klassen

A rose is a rose is a rose is a rose.

— Gertrude Stein

```
(defun is-a (the-object the-class-object)
  (when (eq (class-of the-object) the-class-object)
    the-class-object))
(is-a (is-a (is-a (find-class 'standard-class)
                  (find-class 'standard-class))
            (find-class 'standard-class))
      (find-class 'standard-class))
```

— CLOS

Struktur und Verhalten von Objekten wird in CLOS durch ihre Klasse festgelegt. Für orthogonale persistente Systeme sollten daher ebenso innerhalb des persistenten Speichers persistente Objekte mit einer geeigneten persistenten Klassen-Repräsentation abgelegt werden.

Selbstbeschreibende Systeme

Die Struktur und das Verhalten der im persistenten Speicher enthaltenen Objekte werden durch eine persistente Klassen-Repräsentation explizit festgelegt; damit werden die im persistenten System enthaltenen Objekte unabhängig von der Kopplung an das benutzte transiente System selbstbeschreibend (*self-descriptive*). Diese Eigenschaft ist besonders nützlich, wenn unterschiedliche transiente Systeme an den persistenten Speicher angekoppelt werden sollen.

Strukturbeschreibungen von persistenten CLOS-Objekten

Zur transienten Repräsentation von persistenten Objekten werden im allgemeinen Instanzen der im transienten System vorgefundenen Klasse benutzt. Damit muß zu jedem persistenten Objekt zumindestens soviel Information abgelegt werden, daß die Klasse im transienten System lokalisiert werden kann, z.B. über den Klassennamen. Zusätzlich sollten noch weitere Informationen über die Struktur des persistenten Objektes mit abgelegt werden, da CLOSÄnderungen an Klassendefinitionen zuläßt; ein Vergleich der persistenten mit der transienten Strukturinformation kann dann zur Erkennung einer Änderung der transienten Klassendefinition benutzt werden, um dann diese Änderung auf die persistente Strukturinformation und die betroffenen persistenten Instanzen zu propagieren (Schemaentwicklung).

5.2.3 Funktionen

Funktionen enthalten ausführbaren maschinenabhängigen Binärcode, der zudem abhängig vom aktuellen Common LISP Prozeß ist; eine Standardisierung von Funktionscode erfolgte bisher nicht. Ein Speichern bzw. Laden von Code bedeutet daher, eine umfangreiche Transformation des Codes zwischen seiner Repräsentation im transienten und im persistenten Speicher durchzuführen. Beim Speichern müssen die im Code enthaltenen Referenzen in ein prozeßunabhängiges Format gebracht werden, beim Laden des Codes muß dieses prozeßunabhängige Format wieder in das prozeßabhängige Format des transienten Systems umgewandelt werden. Beide Vorgänge greifen in die Interna des benutzten Common LISP ein und setzen damit entsprechende Kenntnisse voraus, die im allgemeinen nicht erhältlich sind. Wegen dieser nicht uner-

heblichen Schwierigkeiten behandeln die in dieser Arbeit betrachteten Systeme kein Speichern oder Laden von Funktionscode, obwohl dies dem Gedanken der orthogonalen Persistenz widerspricht.

5.2.4 Nicht speicherbare Objekte

Sorry, Dave, I can't do that.

— HAL 9001

Wie im letzten Abschnitt gezeigt, kann es Objekte geben, die aus welchem Grund auch immer nicht speicherbar sind. Ein persistentes System sollte auch mit diesen Fällen kontrolliert umgehen können, d.h. insbesondere, daß diese Objekte nicht zu einem unkontrolliertem Fehlerabbruch führen, falls sie z.B. in Folge der Ablage eines speicherbaren Objektes durch die transitiv referenzierte Hülle des Objektes ebenfalls im persistenten Speicher abgelegt werden sollen.

5.2.5 Extern persistente Objekte

Extern persistente Objekte sind Objekte, die nicht durch das persistente System selbst, sondern entweder durch ihre grundsätzlichen Eigenschaften oder durch externen Einfluß (extern bezüglich des persistenten Systems) die Eigenschaft der Persistenz erhalten. Damit erübrigt es sich für das persistente System, diese Objekte nochmals persistent abzulegen, da sie diese Eigenschaft bereits haben; es genügt, eine Referenz auf diese Objekte im persistenten Speicher abzulegen [Heiler et al. 89, S. 239]. Diese Referenz muß über eine Identitätsrepräsentation hergestellt werden, die den in Abschnitt 3.1 (S. 12) dargestellten Bedingungen genügt. Dateien sind z.B. extern persistente Objekte, auf die über den Dateinamen eine Referenz aufgebaut werden kann.

In diesem Zusammenhang wird ein Objekt dann und nur dann als extern persistent angesehen, wenn sein Gesamtzustand ausschließlich aus Teilzuständen besteht, die entweder Werte sind oder weitere extern persistente Objekte referenzieren.

Bestimmte (CLOS) Objekte innerhalb des transienten Common LISP-Systems können bzw. müssen als extern persistent angesehen werden. Die Eigenschaft der Persistenz erhalten diese Objekte dadurch, daß sie vom Common LISP-System nach jedem Start des Systems so restauriert werden, so daß sie über die Zeit ihre Identität behalten. Vorbedingung für die Referenzierung von extern-persistenten Objekten ist die Existenz einer zur Bezugnahme auf diese Objekte geeigneten Identitätsrepräsentation.

Optional extern persistente Objekte Als extern persistent können beispielsweise Klassen-Metaobjekte angesehen werden, deren Zustand in einer Programm-Modul-Datei persistent gehalten wird und deren transiente Repräsentation als Folge des Ladevorgangs aufgebaut wird. Zur Bezugnahme auf ein Klassen-Metaobjekt kann dann bei vorausgesetzter Eindeutigkeit der Name der Klasse verwendet werden.

Sofern sich ein persistentes Objekt auf ein derartiges Objekt bezieht, muß es spätestens bei der Dereferenzierung des extern persistenten Objektes in einer geeigneten dereferenzierbaren Repräsentation (z.B. als transientes CLOS-Objekt) vorliegen; sie kann auch im Verlauf der Dereferenzierung selbst durch das persistente System erzeugt werden.

System-interne Objekte Bestimmte vom Common LISP-System vordefinierte transiente Objekte sollten nur gelesen aber niemals überschrieben werden; ferner wird ihr Zustand durch das Common LISP-System selbst im Verlauf des Systemstarts wiederhergestellt (*environmental objects* [Ford et al. 88, S. 33]). Wegen des ausschließlich lesenden Zugriffs auf diese Objekte lohnt es sich nicht, sie im persistenten Speicher abzulegen, da ihr transienter Zustand nicht durch den im persistenten Speicher abgelegten Zustand überschrieben werden darf; zudem sind diese Objekte in Bezug auf Persistenz ‚uninteressant', da sie entweder lediglich Teilzustände des Common LISP von systeminterner Bedeutung enthalten oder aber in ihrer Bedeutung so festgelegt sind, daß sie als konstant angenommen werden können.

In diese Kategorie fallen beispielsweise alle Standard-Metaobjekte der in [AMOP] definierten Standard-Metaobjekt-Klassen. Redefinitionen der Standard-Metaobjekt-Klassen sind nicht erlaubt [AMOP, S. 144]; daraus folgt, daß die zur Repräsentation verwendeten Standard-Metaobjekte ebenfalls nicht direkt modifiziert werden dürfen. Für Referenzen von persistenten Objekten auf die Standard-Metaobjekte können z.B. für Standard-Klassen-Metaobjekt-Klassen ihre Namen verwendet werden.

5.3 Objektrepräsentation

Je nach Umgebung eines persistenten Objektes bieten sich unterschiedliche Repräsentationsformen für seinen Zustand an.

5.3.1 Objekte in relationalen Datenbanken

Um die von einer relationalen Datenbank angebotenen Funktionalitäten für persistente Objekte verfügbar zu machen, müssen die dort definierten Strukturen benutzt werden; die in dieser Arbeit betrachteten Systeme für persistente Objekte, die als Sekundärspeicher eine relationale Datenbank benutzen, arbeiten daher nach folgendem Prinzip:

- Eine Klasse mit persistenten Instanzen wird auf eine Relation abgebildet (1. Zeile in Tabelle 4.5, S. 39); der Name der Relation wird im allgemeinen aus dem Namen der Klasse abgeleitet. Die Relation wird vor dem ersten Speichern einer persistenten Instanz in der Datenbank angelegt.

- Für jeden effektiven persistenten Slot einer solchen Klasse wird ein Attribut innerhalb der zur Klasse gehörigen Relation definiert (3. Zeile in Tabelle 4.5); der Name des Attributes wird im allgemeinen aus dem Namen des *Slots* abgeleitet. In den meisten relationalen Datenbanken muß ein Attribut statisch typisiert werden; damit ist der Zustand eines *Slots* eines persistenten Objektes auf einen Typ festgelegt. Eine Folge dieser Typisierung kann sein, daß der *Slot* nur Werte und keine persistenten Objekte referenzieren kann.

- Der Zustand eines persistenten Objektes wird als Tupel in der Relation abgelegt, die zur Klasse des Objektes gehört (2. Zeile in Tabelle 4.5). Die *objid* des persistenten Objektes wird ebenfalls in einem zusätzlichen Attribut abgelegt.

Die hier beschriebene Art der Repräsentation führt mindestens innerhalb des persistenten Systems zu einer unterschiedlichen Sichtweise von Metaobjekten und‚normalen' CLOS-Instanzen; Klassen-Metaobjekte werden durch Relationen, *Slot*-Metaobjekte durch Attribute und‚normale'

Instanzen durch Tupel repräsentiert. Die Rückabbildung auf ein Metaobjekt kann je nach verwendeter Datenbank schwierig werden. Einige Datenbanken bieten Zugriff auf die in systeminternen Relationen (*system catalogs*) [O'Neil 94, S. 430] abgelegten Informationen über die verwendeten Repräsentationen; beispielsweise werden Relationen in der objekt-orientierten relationalen Datenbank POSTGRES in der systeminternen Relation **pg type** durch einen Tupel repräsentiert [Wensel 93, S. 41] und können von dort auch gelesen werden.

5.3.2 Objekte in Objektspeichern

Da Objektspeicher keine bestimmten Typisierungskonzepte vorschreiben, können die Strukturen zur Repräsentation von Instanzen dort frei gewählt werden. Der einfachste Weg besteht darin, sie ähnlich wie die Strukturen des transienten Systems festzulegen; zwischen transienten und persistenten Objekten kann dann eine direkte Abbildung durchgeführt werden.

5.3.3 Repräsentation von persistenten Objekten im transienten Speicher

Um überhaupt Zugriff auf den Zustand eines persistenten Objektes zu erhalten, muß es im transienten Speicher repräsentiert werden; der Umfang der Repräsentation kann unterschiedlich groß sein:

- Eine minimale Repräsentation würde lediglich die Informationen enthalten, die für die Adressierung der Instanz im persistenten Speicher notwendig wären, also seine *objid*. Der Zugriff auf den Objektzustand ist dann ausschließlich über Methoden möglich, die die Repräsentation des Objektes im persistenten Speicher referenzieren.

- Eine maximale Repräsentation würde den gesamten Zustand des Objektes in den transienten Speicher kopieren; damit wäre ein Zugriff mit den üblichen Funktionen möglich. Ein Problem bereitet in diesem Fall der Abgleich des Zustands der transienten und persistenten Repräsentation. Dabei wird unter Umständen viel Speicherplatz belegt.

Die in dieser Arbeit betrachteten persistenten Systeme bieten teilweise Mischformen zwischen beiden Repräsentationsarten an, indem z.B. beim Laden eines persistenten Objektes zunächst eine minimale Repräsentation erzeugt wird, die bei der ersten Dereferenzierung des Objektzustands die maximale Repräsentation auf *Slot*- oder Objekt-Ebene alloziert und nachlädt (Beispiel siehe Abschnitt 6.4.3, S. 74).

5.4 Aktive und passive Objekte

Die in einem Common LISP-System vorhandenen Objekte werden von mir in Anlehnung an [Ford et al. 88, S. 26] nach der grundsätzlich gegebenen Art derÄnderung oder Erweiterung ihrer Semantik (im Sinne von [AMOP, S. 1]) in aktive und passive Objekte eingeteilt. Im Rahmen dieser Arbeit wird die Erweiterung der Objektsemantik um Persistenz betrachtet.

5.4.1 Aktive Instanzen

Bei *aktiven Instanzen* kann eine Semantikerweiterung oder -änderung (Persistenz) grundsätzlich mittels der *vorhandenen*, in [CLtLII] und [AMOP] definierten generischen Standard-Funktionen eingebunden werden. Alle Instanzen zeigen somit unabhängig von ihrer Lebensdauer

nach außen das gleiche Verhalten; persistente Instanzen unterscheiden sich lediglich durch ihre verlängerte Lebensdauer von transienten Instanzen. Aktive Instanzen sind die Objekte der gekapselten Klassen und damit insbesondere in Common LISP die Instanzen der CLOS Klassen. Eine Einbindung betrifft die Methoden der generischen Standard-Funktionen und kann auf folgende Arten erfolgen:

Ersetzen der Standard-Methoden

Der in [CLtLII] und [AMOP] definierte Standard läßt ein Überschreiben der für Persistenz relevanten Standard-Methoden zu; damit können sie durch Methoden des persistenten Systems ersetzt werden, die beispielsweise alle *Slot*-Zugriffe auf eine im persistenten Speicher abgelegte Repräsentation umleiten.

Mit dieser Ersetzung würden ausnahmslos alle CLOS-Instanzen persistent werden. In einer konkreten Realisierung dieses Ansatzes kann es Probleme geben, da das LISP-System sich unter Umständen nicht vollständig an den in [CLtLII] und [AMOP] definierten Standard hält, beispielsweise durch den Zugriff auf den Objektzustand über Funktionen der unteren Schichten des Objektsystems; ebenso kann man davon ausgehen, daß die Standard-Methoden optimiert sind und eine Ersetzung zu erheblichen Effizienzverlusten führen kann. Wegen dieser unkalkulierbaren Seiteneffekte werden die Standard-Methoden daher von den meisten persistenten Systemen ähnlich wie systemprimitive Funktionen (Abschnitt 5.6.2, S. 60) angesehen, die besser nicht überschrieben werden.

Nachteilig ist auch, daß diese Vorgehensweise lediglich das Verhalten von CLOS-Instanzen modifiziert; ihre transiente Struktur bleibt unverändert. Damit können bestimmte Eigenschaften, die sich in Verbindung mit einem persistenten System ergeben würden (wie beispielsweise ausschließliche Repräsentation eines *Slot*-Zustands im persistenten Speicher), nicht realisiert werden.

Die durch diesen Ansatz realisierte Persistenz ließe sich als radikal-orthogonal bezeichnen, da alle CLOS-Instanzen grundsätzlich persistent gehalten werden; aus pragmatischen Gründen sollte ein persistentes System aber eher annehmen, daß eine CLOS-Instanz transient ist, bis es eine gegenteilige Information erhält.

Erweiterung der Standard-Methoden

Ebenso ist eine Erweiterung der in [CLtLII] und [AMOP] definierten für Persistenz relevanten Standard-Methoden zulässig, sofern ihre Funktionalität nicht eingeschränkt wird [AMOP, S. 144]. Die bei einer Ersetzung entstehenden Nachteile lassen sich zwar zum Teil vermeiden, indem für rein transiente Objekte die Standard-Methoden und für persistente Objekte Funktionen des persistenten Systems aufgerufen werden; da ein großer Teil der aufgerufenen generischen Funktionen auch bei Verwendung eines persistenten Systems transiente Objekte referenziert, wird auch bei einer Erweiterung ein Effizienzverlust auftreten.

Wie bei der Ersetzung kann die Struktur der CLOS-Instanzen nicht verändert werden.

Spezialisierte Klassen-Metaobjekt-Klassen und spezialisierte Methoden

Durch das in [AMOP] festgelegte Protokoll kann Verhalten und Struktur von aktiven Objekten in CLOS durch Spezialisierung ihrer Klassen-Metaobjekt-Klasse in Bezug auf Persistenz modifiziert werden:

- Die Strukturdefinition von aktiven Objekten kann im Verlauf der Initialisierung des die Klasse repräsentierenden Klassen-Metaobjektes verändert werden.

- Das MOP legt fest, daß die für Persistenz relevanten (generischen) Standard-Funktionen immer eine generische Funktion mit der Klassen-Metaobjekt-Klasse der CLOS-Instanz aufrufen; die auf die Klassen-Metaobjekt-Klasse spezialisierten Methoden legen das ‚eigentliche' persistente Verhalten der Instanz fest. Auf eine spezialisierte Klassen-Metaobjekt-Klasse spezialisierte Methoden können somit das Verhalten eines CLOS-Objektes modifizieren.

Über die Spezialisierung lassen sich die Nachteile der Ersetzung bzw. Erweiterung nur der Methoden vermeiden, da sie lediglich die persistenten Objekte beeinflußt; Verhalten und Struktur der transienten Objekte bleiben unverändert. Die spezialisierte Klassen-Metaobjekt-Klasse wird vom persistenten System realisiert und über eine Klassenoption von den BenutzerInnen in deren Klassen eingebunden. Das persistente System kann annehmen, daß Instanzen mit einer derartigen spezialisierten Klassen-Metaobjekt-Klasse grundsätzlich persistent zu halten sind, da andernfalls die spezialisierte Klassen-Metaobjekt-Klasse nicht benutzt worden wäre.

Ein weiterer Vorteil in der Verwendung von auf Persistenz spezialisierten Klassen-Metaobjekt-Klassen liegt darin, daß im Verlauf der Initialisierung des Klassen-Metaobjektes zusätzliche Klassen- und *Slot*-Optionen ausgewertet werden können, um beispielsweise eine bestimmte Repräsentationsform (Abschnitt 5.3.3, S. 54) oder einen Schnitt (Abschnitt 5.5, S. 57) auf *Slot*- oder Klassen-Ebene zu deklarieren; konkrete Beispiele dafür werden in Abschnitt 7.6.5 (S. 111) gegeben.

Eine CLOS-Instanz, deren Klassen-Metaobjekt-Klasse auf Persistenz spezialisiert ist, wird im folgenden Text als *aktiv-persistente Instanz* bezeichnet; alle anderen Objekte sind *passiv-persistente Objekte*. Analog dazu sind die jeweiligen Klassen der Objekte *aktiv-persistente* bzw. *passiv-persistente Klassen*.

In den in dieser Arbeit betrachteten persistenten Systemen werden intern in den unteren Schichten alle Objekte als passiv-persistent angesehen; aktiv-persistente Objekte entstehen erst durch das in höheren Schichten mit spezialisierten Methoden realisierte Verhalten.

5.4.2 Passive Objekte

Alle nicht-aktiven Instanzen sind *passive Objekte*, d.h. die in [CLtLII] und [AMOP] definierten Standard-Funktionen bzw. die Methoden der generischen Standard-Funktionen können nicht erweitert werden; Persistenz kann entweder durch Übertragung der Objektzustände zwischen einer transienten und einer persistenten Repräsentation (*Swizzling*) oder durch Realisierung von (zusätzlichen) generischen Funktionen und Methoden zur Verfügung gestellt werden. In LISP sind die Instanzen der Basis- und Struktur-Klassen passive Objekte.

5.4.3 Kopplung von Persistenz an aktiv-persistente Objekte

Ein persistentes System sollte Persistenz als Möglichkeit anbieten, die, wenn sie genutzt wird, als Folgerung aus der orthogonalen Sichtweise grundsätzlich allen Objekten unabhängig von ihrer Klasse zur Verfügung steht. Bei den für diese Arbeit betrachteten Systemen, die eine relationale Datenbank als persistenten Speicher verwenden, besteht die Möglichkeit der Persistenz lediglich für aktiv-persistente Objekte und für passiv-persistente Instanzen einiger weniger Basistypen mit entsprechenden Basistypen in der relationalen Datenbank; damit wird die

Möglichkeit der Persistenz für CLOS-Instanzen an die Art ihrer Einbindung gekoppelt. In diesem Fall wird durch die eingeschränkte Persistenz eine nicht-orthogonale Sichtweise vertreten. Problematisch bei diesem Ansatz sind Referenzen der *Slots* von aktiv-persistenten Objekten auf aktive Objekte; da aktive Objekte nicht persistent werden können, wird in diesen Systemen während des Speicherns eines aktiv-persistenten Objektes bei einer solchen Referenz im allgemeinen ein Fehler angezeigt. Bei einer orthogonalen Sichtweise von Persistenz sollten aktive Objekte als passiv-persistent verarbeitet werden.

5.5 Schnitt zwischen Objekten

```
H i n w e i s !
Alle Kurzprogramme werden automatisch
mit textilgerechtem Endschleudern
ohne Spülstop beendet.
```

— Bedienungsanleitung der Waschmaschine
‚Siemens Siwamat Plus 3600‘

Ein transientes Objekt kann sehr viele Referenzen auf andere transiente Objekte enthalten; beim Speichern eines solchen Objektes kann es passieren, daß das persistente System beim Speichern der transitiv referenzierten Objekte einen nicht unerheblichen Teil des Gesamtzustands des Common LISP-Systems im persistenten Speicher ablegt [Ford et al. 88, S. 26]. Beim Laden eines derartigen Objektes würde ein großer Teil des Systemzustands entweder überschrieben werden oder er würde mehrmals repräsentiert werden; ersteres birgt die Gefahr, daß auch systeminterne Objekte unerwartet für das Common LISP-System verändert werden, während letzteres die Speicherresourcen des transienten Systems unnötig belastet. Beispiele für derartige Objekte sind Klassen-Metaobjekte, die jeweils eine Liste ihrer Sub- und Superklassen enthalten. Das Speichern eines Klassen-Metaobjektes würde daher über die transitive Hülle alle Klassen-Metaobjekte im persistenten Speicher ablegen; das Laden eines Klassen-Metaobjektes würde dementsprechend entweder die vorhandenen Klassen-Metaobjekte überschreiben oder die komplette Klassenhierarchie ‚neben‘ der bestehenden zusätzlich aufbauen.

Ein persistentes System sollte daher abhängig von einem Kriterium beim Speichern eines Objektes einen ‚Schnitt‘ machen, ab dem die weiter transitiv referenzierten Objekte nicht mehr gespeichert werden. Dieser Schnitt betrifft einen Teilzustand des zu speichernden transienten Objektes, der während des Speichervorgangs nicht weiter traversiert wird. Da die Objekte jenseits des Schnitts nicht im persistenten Speicher repräsentiert werden, dürfen von dort keine Referenzen auf persistente Objekte existieren, da die persistenten Objekte unter Umständen nicht dereferenzierbar sind; dies ist z.B. der Fall, wenn das persistente System nicht geladen wurde.

5.5.1 Extern-persistente Objekte

Wegen der in Abschnitt 5.2.5 (S. 52) genannten Eigenschaften und der Abgeschlossenheit von extern persistenten Objekten kann bei der Anforderung, ein extern persistentes Objekt im persistenten Speicher abzulegen, ein Schnitt durchgeführt werden; der Zustand des extern persistenten Objektes braucht nicht gespeichert zu werden.

In diesem Fall kann die Referenz auf das extern persistente Objekt durch seine Identitätsrepräsentation hergestellt werden, d.h. die Referenzen des persistenten Objektes bleiben erhalten.

5.5.2 Deklaration eines Schnitts

Da sich eine (automatische) Bestimmung, ob ein Schnitt durchgeführt werden soll oder nicht, als schwierig erweist, besteht eine praktikable Lösung darin, dem persistenten System Hinweise auf die Situationen zu geben, in denen ein Schnitt durchgeführt werden soll[1]. Diese Hinweise können beispielsweise darin bestehen, daß bestimmte Teilzustände von Objekten grundsätzlich nicht persistent gehalten werden sollen.

Damit werden im persistenten Objekt Referenzen auf andere Objekte weggelassen; die transiente und persistente Repräsentation des Objektes unterscheiden sich also.

5.5.3 Redundante Referenzen

Sofern die Bedeutung eines transienten Objektes bekannt ist, können redundante Referenzen bei der Übertragung des Objektzustands vom transienten in den persistenten Speicher weggelassen werden und bei der Rückübertragung des Zustands wieder hinzugefügt werden. Ein Beispiel dafür ist die in einem Klassen-Metaobjekt enthaltene Liste der Subklassen. Ein Weglassen dieser Liste verhindert, daß in Verbindung mit der Liste der Superklassen alle Klassen-Metaobjekte übertragen werden; bei der Übertragung des Zustands aus dem persistenten in den transienten Speicher ist die Liste der Subklassen entweder leer, wenn die repräsentierte Klasse noch nicht exisitierte, oder sie kann aus dem existierenden transienten Klassen-Metaobjekt übernommen werden.

5.6 Protokolle

In diesem Abschnitt werden allgemeine Konzepte für das Verhalten im Zusammenhang mit persistenten Klassen und Objekten erläutert; spezielle Protokolle werden in den Abschnitten 6.4.5 (S. 75) und 7.6.5 (S. 111) für die in den Kapiteln 6 und 7 beschriebenen persistenten Systeme erklärt.

5.6.1 Definition von aktiv-persistenten Klassen

Aktiv-persistente Klassen werden in der Klassendefinition im Makro `defclass` mit der Klassenoption `:metaclass` realisiert, deren Argument der Name einer innerhalb des persistenten Systems implementierten auf Persistenz spezialisierten Klassen-Metaobjekt-Klasse ist. Ihre Instanzen repräsentieren Klassen mit aktiv-persistenten Objekten. Die Auswertung der Klassenoption `:metaclass` wird durch das MOP festgelegt; Details zur Vorgehensweise des MOP finden sich in [Paepcke 91b]. Das Protokoll 7.19 **(defclass)** (S. 111) erläutert die Vorgehensweise speziell für das in Kapitel 7 (S. 81) beschriebene persistente System. An dieser Stelle werden nur kurz die Möglichkeiten beispielhaft erklärt, die sich aus der Definition von aktiv-persistenten Klassen ergeben; inwieweit sie genutzt werden, hängt von der konkreten Realisierung des persistenten Systems ab.

Zusätzliche Klassenoptionen

Zusätzlich zu den in [CLtLII, S. 823] definierten Standard-Klassenoptionen können durch die spezialisierte Klassen-Metaobjekt-Klasse weitere Klassenoptionen ausgewertet werden, die sich

[1] Analog ist z.B. das Vorgehen des Common LISP Übersetzers, Deklarationen zur Code-Optimierung auszuwerten.

auf Verhalten oder Struktur der Instanzen der definierten Klasse auswirken. Beispiele dafür sind:

- Die Lebensdauer, die standardmäßig für die *Slot*-Zustände von Instanzen der definierten Klasse benutzt werden soll, kann als Klassenoption deklariert werden.

- Die Vorgehensweise bei Änderung der Klassendefinition seit der letzten Ablage des die Klasse repräsentierenden Klassen-Metaobjektes kann ausgewählt werden (Art der Schemaentwicklung).

Zusätzliche *Slot*-Optionen

Ebenso kann die spezialisierte Klassen-Metaobjekt-Klasse außer den Standard-*Slot*-Optionen [CLtLII, S. 822] weitere *Slot*-Optionen auswerten, die sich auf Repräsentation und Verhalten eines *Slots* innerhalb der Klasse beziehen:

- Die Lebensdauer des *Slot*-Zustands der Instanzen kann deklarativ vorgegeben werden, z.B. indem er als transient deklariert wird; dann muß der *Slot*-Zustand beim Speichern der transienten Instanz nicht weiter referenziert werden und ermöglicht so einen Schnitt.

- Für den *Slot*-Zustand der Instanzen kann eine bestimmte Repräsentationsform im transienten oder persistenten Speicher ausgewählt werden. Dies beeinflußt im allgemeinen auch die Zugriffsmethoden auf den *Slot*-Zustand.

Spezialisiertes Verhalten

Durch die spezialisierte Klassen-Metaobjekt-Klasse kann auch das Verhalten von aktiv-persistenten Instanzen beeinflußt werden; für Persistenz interessiert in erster Linie das Verhalten beim Erzeugen einer (zunächst) transienten aktiv-persistenten Instanz sowie der Zugriff auf die *Slot*-Zustände.

- Beim Erzeugen einer (transienten) Instanz einer aktiv-persistenten Klasse kann durch eine spezialisierte Methode des persistenten Systems die persistente Repräsentation des Objektes gleich mit erzeugt werden. Begründet wäre dieses Vorgehen durch die Sichtweise, daß Instanzen einer aktiv-persistenten Klasse immer persistent sein sollen.

  Alternativ dazu kann die spezialisierte Methode auch die Erzeugung der transienten Repräsentation im Umfang einschränken und einen Teil des Objektes im persistenten Speicher allozieren; die Zugriffsmethoden müssen dann entsprechend die persistente Repräsentation des Objektes referenzieren.

- Die für den Zugriff auf den *Slot*-Zustand spezialisierten Methoden können für die Verarbeitung der verschiedenen Repräsentationsformen des Zustands genutzt werden.

5.6.2 Zugriff auf den Objektzustand

Dieser Abschnitt erläutert verschiedene Ansätze für den Zugriff auf den Zustand von passiv-persistenten und aktiv-persistenten Objekten.

Passiv-persistente Objekte

Die Manipulation von einfachen Daten in Common LISP erfolgt mit z.T. systemprimitiven Funktionen, die ausschließlich auf der festgelegten transienten Repräsentation des Datums arbeiten können. Die Forderung nach möglichst großer Orthogonalität würde angewendet auf die Manipulationsmöglichkeiten von einfachen Daten bedeuten, daß die gleichen Funktionen sowohl auf der transienten als auch auf der persistenten Repräsentation eines Objektes arbeiten können sollten. Praktisch würde dies bedeuten, diesen Funktionen eine Komponente für die Verarbeitung der persistenten Repräsentation hinzuzufügen, was im allgemeinen für systemprimitive Funktionen durch das LISP-System verwehrt wird; außerdem könnte ein derartiger Eingriff im Fehlerfall die Funktionsfähigkeit des kompletten Systems beeinträchtigen. Praktisch verwendbar sind Objekt-*Swizzling* oder die direkte Manipulation der persistenten Repräsentation.

Objekt-*Swizzling* *Swizzling* bezeichnet die Vorgehensweise, für den Zugriff auf ein persistentes Objekt zusätzlich zur persistenten eine transiente Repräsentation zu allozieren und den Zustand des persistenten Objektes in seine transiente Repräsentation zu kopieren. Referenzen zwischen persistenten Objekten im persistenten Speicher werden beim Kopiervorgang in die Identitätsrepräsentation des transienten Systems transformiert. Manipuliert wird direkt und ausschließlich die nach außen zur Verfügung gestellte transiente Repräsentation des Objektes. Zu ‚geeigneten Zeitpunkten' wird der Zustand der persistenten Repräsentation auf den Stand der transienten gebracht.

Zeitpunkt des Abgleichs Der Zeitpunkt dieses Abgleichs muß festgelegt werden; er sollte möglichst nach Bedarf, d.h. nach einer tatsächlich erfolgten Änderung des Objektzustands erfolgen. Eine Änderung ist nur schwer zu detektieren, da zum einen die Manipulations-Funktionen nicht modifiziert werden können und damit ein Markieren des Objektes als Folge einer Änderung nicht möglich ist;[2] zum anderen scheidet ein Vergleich des Zustands der transienten und der persistenten Repräsentation aus Effizienzgründen meist aus. Deswegen wird der Abgleich meist an andere Vorgänge gebunden; wenn das Objekt beispielsweise von einer anderen Instanz referenziert wird, wird es immer dann gespeichert, wenn auch die referenzierende Instanz gespeichert wird. Eine andere Möglichkeit besteht darin, den Abgleich nur auf expliziten von außen erfolgenden Aufruf einer entsprechenden Funktion durchzuführen [Cardelli et al. 88, S. 35].

Persistente und transiente Identität Das Objekt erhält zusätzlich zu seiner persistenten eine transiente Identität mit eigener Repräsentation; für den Abgleich muß die transiente Identitätsrepräsentation wieder auf die persistente abgebildet werden, damit der Abgleich für genau das persistente Objekt durchgeführt wird, aus dem die transiente Repräsentation entstanden ist.

Effizienz Die Manipulation der Objekte erfolgt nach wie vor sehr effizient mit den vorhandenen Funktionen auf den transienten Repräsentationen.

Direkte Manipulation der persistenten Repräsentation Als zweite Möglichkeit zum Zugriff auf den Zustand von passiv-persistenten Objekten gibt es die direkte Manipulation der

[2] Falls dies möglich wäre, könnte statt des Setzens der Markierung ebenso der Abgleich selbst durchgeführt werden.

persistenten Repräsentation. Das persistente System bietet eigene Funktionen an, die direkt auf den persistenten Objekten im persistenten Speicher arbeiten.

Transiente Repräsentation Sofern eine transiente Repräsentation des persistenten Objektes existiert, sollten Änderungen des im persistenten Speicher abgelegten Zustands in die transiente Repräsentation propagiert werden; bei fehlender transienter Repräsentation entfällt der Abgleich.

Explizite Angabe der persistenten Identitätsrepräsentation Für die direkte Manipulation der persistenten Repräsentation muß die Identitätsrepräsentation des persistenten Objektes von den BenutzerInnen explizit angegeben werden; damit ist zwar das zu manipulierende Objekt eindeutig identifiziert, aber die Benutzung des persistenten Systems wird durch die direkte Verwendung der Identitätsrepräsentationen und der direkt darauf arbeitenden Funktionen für die BenutzerInnen intransparent.

Effizienzverlust Eine direkte Manipulation beinhaltet im allgemeinen den Transfer von Daten zwischen dem transienten und dem persistenten Speicher; die Folge davon kann ein erheblicher Effizienzverlust sein.

Aktiv-persistente Instanzen

Die Manipulation von aktiv-persistenten Instanzen erfolgt ausschließlich über Methoden, die anders als die Manipulations-Methoden für passiv-persistente Objekte erweitert oder modifiziert werden können. Damit kann sowohl ein kontrolliertes *Swizzling* als auch die direkte Manipulation der persistenten Repräsentation von Instanzen realisiert werden.

5.7 Lokalisierung von Objekten

Nachdem ein Objekt persistent gespeichert wurde, muß es möglich sein, das Objekt zu lokalisieren. Für den Prozeß, der das persistente Objekt erzeugt hat, stellt dies im allgemeinen zunächst keine Schwierigkeit dar, wenn er beispielsweise die *objids* der von ihm erzeugten persistenten Instanzen in (transienten) Variablen ablegt hat und darüber adressiert.

Für Prozesse, die bereits existierende persistente Objekte lokalisieren wollen, steht dies nicht zur Verfügung; persistente Objekte müssen über bestimmte Kriterien lokalisierbar sein. Das Ergebnis einer Lokalisierung ist die Menge der zum Kriterium passenden persistenten Objekte.

5.7.1 Lokalisierung über einen Pfad

Die Lokalisierung über einen Pfad findet sich bei einigen persistenten Systemen, die einen Objektspeicher als persistenten Speicher benutzen (z.B. WOOD [St. Clair 93]). Das einzige direkt addressierbare Objekt in einem Objektspeicher ist das Wurzelobjekt; innerhalb des Wurzelobjektes gibt es mindestens einen *Slot*, über den alle von den BenutzerInnen gespeicherten persistenten Objekte erreichbar sein müssen. Erreichbarkeit muß dabei durch den Zustand der von diesem *Slot* referenzierten persistenten Objekte von den BenutzerInnen sichergestellt werden.

Die Lokalisierung über einen Pfad besteht darin, mit dem Wurzelobjekt zu beginnen und als erstes den oben genannten *Slot* zu dereferenzieren; dieser Vorgang wird rekursiv mit dem dereferenzierten persistenten Objekt sowie einem‚richtigen' *Slot* innerhalb dieses Objektes wiederholt, bis ein ‚Schlußobjekt' erreicht wird; dieses ‚Schlußobjekt' ist das über den Pfad lokalisierte Objekt (Abbildung 5.2). Diese Art der Lokalisierung hat den Vorteil, sehr schnell zum Ziel zu

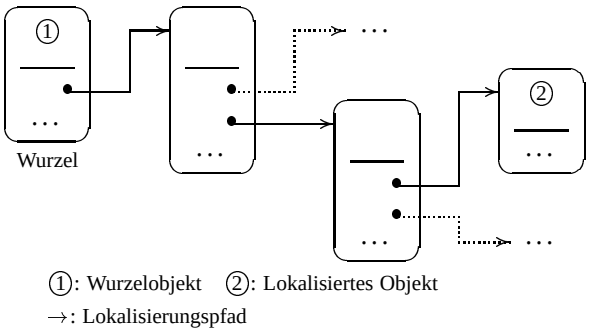


Abbildung 5.2: Lokalisierung eines persistenten Objektes über einen Pfad

führen, da lediglich wenige Objekte traversiert werden müssen. Nachteilig ist, daß die komplette Verwaltung des Objektgraphen in der Hand der BenutzerInnen liegt; das Traversieren (d.h. die Wahl des ‚richtigen' *Slots*) verlangt Kenntnisse über die Strukturen der transitiv vom Wurzelobjekt referenzierten Objekte; ebenso muß ein Kriterium festgelegt werden, wann ein ‚Schlußobjekt' erreicht worden ist und damit die Suche beendet werden kann.

Diese Art der Lokalisierung entspricht vom Konzept her der Adressierung in hierarchischen und netzwerk-orientierten Datenbanken. Sie sollte besser in den unteren Schichten eines persistenten Systems verborgen bleiben; höheren Schichten sollte eine Lokalisierungmöglichkeit geboten werden, die sich mehr an das durch relationale Datenbanken eingeführte Konzept der Adressierung über die Assoziation eines Objektes an einen Wert orientiert und die zudem Erreichbarkeit möglichst einfach sicherstellt.

5.7.2 Lokalisierung über Namen

In LISP kann ein Name (repräsentiert durch eine Zeichenkette) über ein Symbol auf einen Wert abgebildet werden [CLtLII, S. 13, 27]. Eine mehr LISP-gemäße Lokalisierung für persistente Objekte wäre daher die Abbildung eines Namens auf ein persistentes Objekt, eventuell auch unter Benutzung eines persistenten Symbols.

Relationale Datenbank als persistenter Speicher

Bei Verwendung einer relationalen Datenbank als persistenten Speicher läß sich eine solche Abbildung sehr einfach durch eine Relation mit zwei Attributen realisieren, die den Namen auf das dazugehörige persistente Objekt abbildet.

Objektspeicher als persistenter Speicher

In einem Objektspeicher müßte eine persistente Namenstabelle geführt werden, die die entsprechende Abbildung durchführt (Abbildung 5.3; die Objekte ② und ③ können entweder direkt die über die Namen referenzierbaren Objekte sein oder persistente Symbole, deren *Slot* mit dem

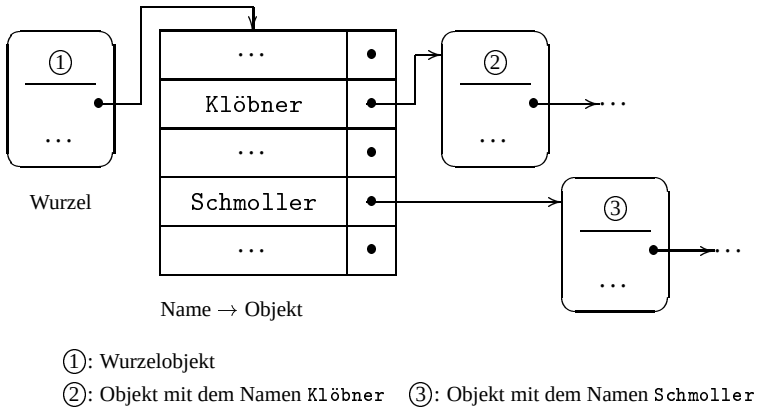


Abbildung 5.3: Lokalisierung eines persistenten Objektes über einen Namen

Wert auf das mit Namen versehene persistente Objekt verweist). Da die Namenstabelle ebenfalls persistent gehalten wird, ist sie vom Wurzelobjekt aus erreichbar; damit sind auch die von der Namenstabelle referenzierten persistenten Objekte immer erreichbar.

Ein Beispiel für die Lokalisierung über Namen findet sich im persistenten Objektsystem ZEITGEIST [Ford et al. 88, S. 28]; dort muß beim Speichern eines transienten Objektes eine Zeichenkette angegeben werden, mit der das aus der Speicherung hervorgegangene persistente Objekt lokalisiert werden kann. In dem von mir realisierten persistentem Objektsystem können persistente Objekte an ein persistentes Symbol gebunden werden (S. 129); eine Lokalisierung besteht im Suchen des persistenten Symbols anhand seines Namens und der Dereferenzierung des Symbol-*Slots*, der den Wert enthält.

5.7.3 Inhaltsorientierte Lokalisierung

Bei der inhaltsorientierten Lokalisierung werden persistente Objekte einer Klasse über den Zustand oder Zustandsbereich eines *Slots* ausgewählt.

Relationale Datenbank als persistenter Speicher

Bei persistenten Objektsystemen mit einer relationalen Datenbank als persistenten Speicher ist die Verwendung der Datenbank mit der Absicht verbunden, die dort realisierte Funktionalität auch für persistente Objekte zur Verfügung zu stellen (siehe z.B. Argumentation in [Rowe 87, S. 2]); deswegen werden auch die Strukturen der persistenten Repräsentationen wie in Tabelle 4.5 (S. 39) gewählt, damit die von der Datenbank angebotenen Funktionen zur Auswahl von Tupeln auch für die persistenten Repräsentationen der persistenten Objekte direkt benutzt werden können. Eine inhaltsorientierte Lokalisierung von Objekten ist damit bereits auf der untersten Ebene des persistenten Objektsystems gegeben.

Objektspeicher als persistenter Speicher

Objektspeicher selbst bieten meist keine inhaltsorientierten Lokalisierungsmöglichkeiten an; statt dessen müssen die entsprechenden Indextabellen innerhalb des persistenten Systems realisiert und gepflegt werden (Abbildung 5.4). Wie die Abbildung zeigt, wird der *Slot*-Zustand zweimal abgelegt: in der Indextabelle und im persistenten Objekt. Das persistente System muß

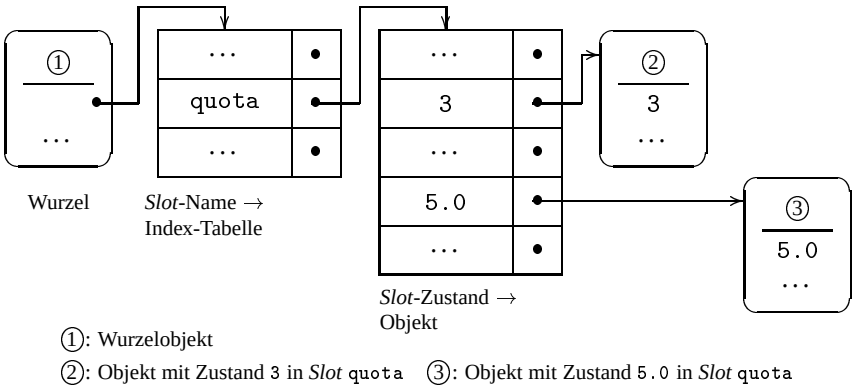


Abbildung 5.4: Inhaltsorientierte Lokalisierung eines persistenten Objektes

daher sicherstellen, daß die Änderung eines *Slot*-Zustands und der Indextabelle konsistent und unteilbar erfolgen. Damit muß ein persistentes System zumindestens Transaktionen realisieren; sofern konkurrenter Zugriff auf den persistenten Speicher möglich ist, muß zusätzlich Unteilbarkeit durch Schreibsperren gewährleistet werden.

Ähnlich wie bei der Lokalisierung über Namen kann auch bei der inhaltsorientierten Lokalisierung Erreichbarkeit durch die Indextabelle gewährleistet werden.

5.8 Datenbank-Konzepte in persistenten Systemen

Dieser Abschnitt beschreibt wünschenswerte Erweiterungen aus der Datenbankwelt, die möglichst in ein persistentes Objektsystem integriert werden sollten. Die hier aufgeführten Punkte betreffen in erster Linie persistente Systeme, die einen Objektspeicher als persistenten Speicher benutzen; Systeme mit relationalen Datenbanken benutzen die entsprechenden von der Datenbank angebotenen Funktionalitäten.

5.8.1 Konsistenter Gesamtzustand

Der Gesamtzustand aller im persistenten Speicher befindlichen Objekte sollte konsistent sein; Zustandsübergänge sollten immer von einem konsistenten Zustand in einen nächsten konsistenten Zustand führen. Da eine automatische Bestimmung eines konsistenten Gesamtzustands recht aufwendig werden kann, überläßt man dessen Festlegung den BenutzerInnen; der Übergang findet innerhalb einer von den BenutzerInnen aktivierten Transaktion statt, die dafür sorgt, daß entweder bei Ende der Transaktion der neue von den BenutzerInnen als konsistent definierte Zustand erreicht wird oder bei einem Abbruch der alte Zustand wiederhergestellt wird [Gray et al. 93, S. 159] (Abbildung 5.5). Ausschließlich innerhalb einer aktiven Transaktion ist ein temporär inkonsistenter Gesamtzustand erlaubt.

5.8.2 Konkurrenter Zugriff

Bei konkurrentem Zugriff muß zusätzlich Serialisierbarkeit gewährleistet sein. Eine einfache Realisierung sind Zwei-Phasen-Transaktionen [Jessen et al. 87, S. 163] in Verbindung mit einem Objekt-Sperrprotokoll. In der ersten Phase werden die persistenten Objekte, auf die zugegriffen werden soll, durch die zugreifende Transaktion für alle konkurrenten Transaktionen

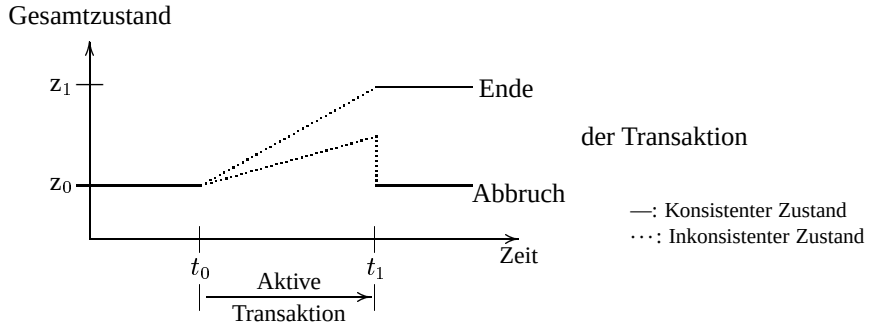


Abbildung 5.5: Zustandsübergänge bei Ende bzw. Abbruch einer Transaktion

gesperrt; die zweite Phase, in der für die zugreifende Transaktion keine neuen Sperren gesetzt werden dürfen, beginnt mit der ersten Aufhebung einer ihrer Sperren. In [Jessen et al. 87, S. 163] wird gezeigt, daß so Serialisierbarkeit sichergestellt ist.

5.8.3 Komplexe Anfragen

Für komplexe Anfragen bieten persistente Objektsysteme mit einer relationalen Datenbank als persistentem Speicher höheren Schichten eine Schnittstelle mit LISP-ähnlicher Syntax, die relativ einfach in Datenbank-Anfragen überführt werden kann. Je nach Qualität der Schnittstelle wird auch die Semantik von LISP berücksichtigt.

Bei Systemen mit Objektspeichern sind die entsprechenden Möglichkeiten meist nicht vorhanden und müssen innerhalb des persistenten Systems realisiert werden.

5.8.4 Schemaentwicklung

In CLOS können Klassendefinitionen geändert werden; ein persistentes System sollte daher Möglichkeiten vorsehen, ein geändertes Klassen-Metaobjekt erneut zu speichern und die persistenten Instanzen der Klasse auf die durch die neue Definition gegebene Struktur zu transformieren.

5.9 Zusammenfassung

Dieses Kapitel erläutert verschiedene Programmierkonzepte, die in die Realisierung eines persistenten Systems in CLOS einfließen können.

Zur Effizienzsteigerung bietet es sich an, die transienten Repräsentationen von persistenten Objekten über einen *Cache* zu referenzieren; damit können Zugriffe auf den persistenten Speicher eingespart werden.

Die Realisierung von orthogonaler Persistenz verlangt, daß ein persistentes System alle in transientem Common LISP und CLOS vorkommenden Instanzen unabhängig vom Typ speichern können sollte. Um ein Objekt zu speichern, muß seine Struktur bekannt sein. Für Instanzen von Basisklassen geht ihre Struktur implizit aus ihrer Definition (z.B. in [CLtLII]) hervor; die Struktur von CLOS-Instanzen einer Klasse wird explizit durch ihr Klassen-Metaobjekt repräsentiert.

Die Informationen über die Struktur eines transienten Objektes wird benutzt, um eine persistente Repräsentation aufzubauen und den Zustand des transienten Objektes dorthin zu übertra-

gen. Bei der Verwendung einer relationalen Datenbank als persistenten Speicher werden durch die relationale Datenmodellierung bestimmte, nicht LISP-gemäße Repräsentationsformen vorgeschrieben; ein Objektspeicher bietet größere Freiheit bei der Wahl der Repräsentation der in ihm enthaltenen Objekte.

Für die Verarbeitung von Persistenz über Objekte gibt es die beiden Sichtweisen der aktiven und der passiven Persistenz. Bei aktiver Persistenz werden für vorhandene generische Funktionen weitere Methoden spezialisiert; damit wird die Einbindung von Persistenz transparent. Für passiv-persistente Objekte kann intransparente Persistenz über *Swizzling* oder über eine direkte Manipulation der persistenten Repräsentation angeboten werden.

Nachdem Objekte gespeichert wurden, sollte eine Lokalisierung der Objekte möglich sein; für die Lokalisierung sollten datenbank-ähnliche Kriterien, wie Auswahl der persistenten Objekte über Namen oder über bestimmte *Slot*-Zustände realisiert werden. Persistente Speicher mit relationalen Datenbanken bieten diese Funktionalität bereits auf niedriger Ebene; bei Verwendung eines Objektspeichers müssen die entsprechenden Auswahlmöglichkeiten meist in einer höheren Schicht zusätzlich realisiert werden.

Die inhaltsorientierte Lokalisierung setzt die zusätzliche Realisierung bestimmter Datenbankfunktionalitäten, wie Transaktionen und gegebenenfalls Sperrprotokolle, für auf Objektspeichern basierende Systeme voraus. Sofern ein derartiges System eine inhaltsorientierte Lokalisierung realisiert, sollten diese zusätzlich realisierten Funktionalitäten nicht nur intern genutzt werden, sondern auch den BenutzerInnen des Systems zur Verfügung gestellt werden.

Kapitel 6

Shared Object Hierarchy

Die Zielsetzung im Rahmen dieser Arbeit bestand darin, zu prüfen, ob die vorliegende Realisierung des persistenten Objektsystems Shared Object Hierarchy (SOH) die in Abschnitt 1.2 (S. 2) gestellte Aufgabenstellung lösen könnte. Der in [Rowe 87] geschilderte Entwurf sprach zunächst dafür; ebenso gab die Verwendung der objekt-orientierten relationalen Datenbank POSTGRES als persistenter Speicher Anlaß zu der Hoffnung, daß die dort realisierten Konzepte auch in SOH wiederzufinden seien. Eine konkrete Überprüfung der Realisierung war nur durch eine Anpassung des vorliegenden Quellcodes an das verwendete LISP-System möglich; dabei wurde auch die Schnittstelle zu der SOH-benutzenden Schicht an die sich durch das MOP ergebenden Möglichkeiten angepaßt.

Das System SOH [Rowe 87] wurde von 1986 bis ca. 1989 von Lawrence A. Rowe in Berkeley an der Universität Kalifornien konzipiert und unter seiner Leitung realisiert. Die Konzeption wird in [Rowe 87] dargestellt; die Dokumentation der Realisierung besteht in einer kurzen Funktionsbeschreibung in [Schank et al. 90, S. 5-74 – 5-79] und aus dem Quellcode.

Die Zielsetzung von SOH besteht darin, eine große Anzahl von Objekten persistent zu speichern und mehreren BenutzerInnen den gemeinsamen Zugriff auf diese Objekte zu ermöglichen [Rowe 87, S. 1]. Als persistenter Speicher dient die objekt-orientierte relationale Datenbank POSTGRES, deren Funktionalität für persistente Objekte genutzt werden soll [Rowe 87, S. 2].

Im folgenden wird die vorliegende Implementation von SOH beschrieben. Es handelt sich dabei um die Version 1.1 mit Stand vom 4.8.1991, entnommen aus dem System PICASSO Version 2.0 [Schank et al. 90]; diese Version wird im folgenden Text als Originalversion bezeichnet. Die Originalversion wurde von mir nach LISPWORKS Common LISP 3.1.1 portiert. Ferner benutzte die vorliegende SOH Version die objekt-orientierte relationale Datenbank POSTGRES 1.0 als persistenten Speicher; die Schnittstelle wurde von mir nach POSTGRES 3.1 umgeschrieben. Die aus der portierten Originalversion entstandene Realisierung wird als portierte Version bezeichnet.

6.1 Architektur

Abbildung 6.1 zeigt ein Schichtenmodell des Systems SOH. Die 1. Schicht wird durch die ob-

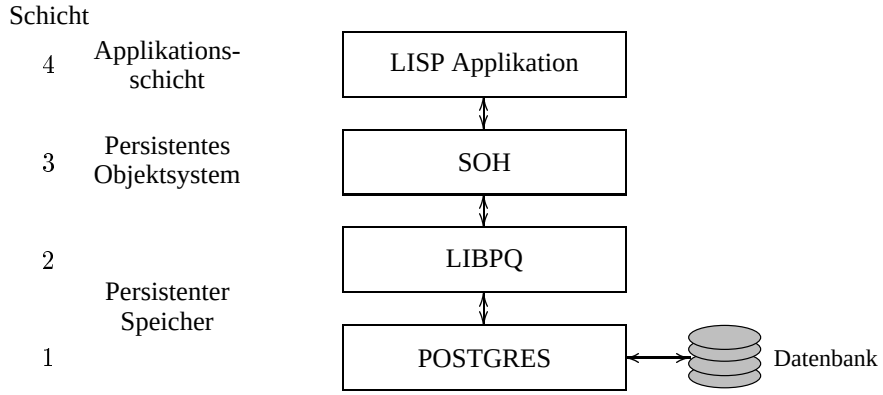


Abbildung 6.1: SOH Schichtenmodell

jekt-orientierte relationale Datenbank POSTGRES gebildet. Schicht 2 ist die für die Anbindung von POSTGRES an LISP verwendete Bibliothek `libpq`. Schicht 3 ist das eigentliche System SOH; es realisiert Persistenz für die Objekte der 4. Schicht.

6.2 Schicht 1: Die objekt-orientierte relationale Datenbank POSTGRES

Die Konzeption für die objekt-orientierte relationale [Stonebraker et al. 89, S. 11] Datenbank POSTGRES [Stonebraker et al. 86a] entstand um 1986 als Nachfolger der Datenbank INGRES in Berkeley an der Universität Kalifornien. Eine Beta-Version war 1987 funktionsfähig; Version 1 wurde ab 1989 ausgeliefert. Mitte 1990 folgte Version 2. Die bei der Anpassung im Rahmen dieser Arbeit verwendete Version 3.1 wurde 1991 veröffentlicht. Die momentan aktuelle Version 4.1 erschien 1993 (alle Angaben nach [Rhein et al. 93, S. 2]). Inzwischen wird POSTGRES unter dem Namen *Montage*[1] in der Version 1.00 kommerziell angeboten [O'Neil 94, S. 179].

Außer den üblichen Eigenschaften einer relationalen Datenbank wurden in POSTGRES die nachfolgend beschriebenen Konzepte realisiert, die zwar z.T. Eingang in die Konzeption von SOH fanden, in der Originalversion aber nicht realisiert wurden, obwohl dies im Rahmen dieser Arbeit nützlich gewesen wäre.

6.2.1 Realisierung von Konzepten der objekt-orientierten Programmierung

In POSTGRES werden Relationen als Klassen und Attribute als *Slots* im Sinne der objekt-orientierten Programmierung aufgefaßt, die an Sub-‚Klassen' (d.h. Relationen in POSTGRES) vererbt werden können [Rhein et al. 93, S. 8] (siehe auch Tabelle 4.5, S. 39). Zusätzlich können Attribute nicht nur auf Basistypen, sondern auch auf Relationen typisiert werden [Rhein et al. 93, S. 10].

Kritik

Wünschenswert wäre eine Aufhebung der aus externer Sicht auch in POSTGRES existierenden Trennung zwischen Tupeln und Instanzen von Basistypen (siehe auch Aufzählung der Typisie-

[1] Ich nehme an, es handelt sich *nicht* um den Plural von Montag.

rungskonzepte von *Montage* in [O'Neil 94, S. 179]).

Die Vererbungsregeln lassen sich nicht beeinflussen; hier fehlt noch eine dem MOP vergleichbare Möglichkeit.

6.2.2 Tupel als Objekte Erster Klasse

Mit Tupeln als Objekte Erster Klasse kann in auf Relationen typisierten Attributen eine Referenz auf einen Tupel innerhalb der Datenbank repräsentiert werden [O'Neil 94, S. 184].

Kritik

Nach wie vor müssen in POSTGRES die Attribute statisch typisiert werden; die Tupel selbst sind nicht typisiert. Ein auf eine Relation typisiertes Attribut kann nur auf ein Tupel der entsprechenden Relation verweisen.

6.2.3 Speicherung und Ausführung von Methoden in der Datenbank

In der Datenbank können Methoden gespeichert und auch ausgeführt werden [Rhein et al. 93, S. 10] [Wensel 93, S. 69]. Der Aufruf einer Methode kann an die Erzeugung bzw. den Zugriff auf den Zustand einer Instanz gebunden werden [Wensel 93, S. 82].

Kritik

Der Code von Methoden ist auf aus C übersetztem Objektcode sowie auf die POSTGRES-eigene Datenmanipulationssprache POSTQUEL festgelegt, d.h. die direkte Ablage oder Ausführung von LISP Code ist nicht möglich.

6.2.4 Definition eigener Basistypen und spezialisierter Methoden

In POSTGRES können von den BenutzerInnen zusätzliche Basistypen in die Datenbank eingebunden werden [Rhein et al. 93, S. 12] [Wensel 93, S. 85] [Stonebraker 86b]. Ferner kann ihr Verhalten durch 1- und 2-stellige Methoden spezialisiert werden [Wensel 93, S. 79][2], insbesondere im Hinblick auf die Verwendung der von POSTGRES angebotenen Indexverwaltung [Rhein et al. 93, S. 27–34]. Speziell für diese Arbeit wäre es möglich gewesen, für geografische Daten entsprechende Basistypen und Zugriffsmethoden zu realisieren, die die von POSTGRES angebotene Indexverwaltung für mehrdimensionale Schlüssel nutzen.

Kritik

Die realisierte Schnittstelle zur Einbindung eines eigenen Basistyps ist extrem umständlich. Dies hat meiner Meinung nach zwei Gründe:

- In POSTGRES wurde ein eigenes Objektsystem realisiert; die oben genannten 1- und 2-stelligen Methoden für die Instanzen dieses Objektsystems zur Nutzung der Indexverwaltung müssen in Form von übersetztem C Code eingebunden werden. Um eine möglichst hohe Performanz zu erreichen, baut die Schnittstelle direkt auf diesem Objektsystem auf. Damit wird bei den BenutzerInnen viel Wissen über die Interna der Datenbank vorausgesetzt.

[2] In POSTGRES wird statt ‚n-stelliger Methode' mit $1 \leq n \leq 2$ die Bezeichnung ‚Operator' verwendet.

- POSTGRES strebt intern eine möglichst hohe Optimierung der Indexverwaltung an; dazu müssen für einen selbstdefinierten Typ der Datenbank deklarativ sehr spezielle Hinweise gegeben werden, die intern das Verhalten der Indexverwaltung für Instanzen des definierten Typs festlegen.

6.2.5 Speicherung von großen Objekten

POSTORE unterstützt die Speicherung von großen Objekten (*large objects*) [Rhein et al. 93, S. 23] [Wensel 93, S. 128–137]. Es werden zwei Formate angeboten, von denen das eine auf Geschwindigkeit und das andere auf Datensicherheit optimiert ist.

Kritik

Ein Objekt in POSTGRES ist genau dann groß, wenn für die Repräsentation seines Zustands mehr als 8 KByte Speicher benötigt werden; diese Größe wird unter Umständen relativ schnell von LISP Objekten (beispielsweise durch Vektoren oder Felder) erreicht. Die Betonung, daß zusätzlich ‚große' Objekte unterstützt werden, läßt den Schluß zu, daß ihre Verarbeitung im Vergleich zu ‚kleinen' Objekten ineffizienter sein wird.

6.2.6 Einbindung der Konzepte in SOH

Einige der oben genannt Konzepte wurden zwar in die Konzeption von SOH übernommen, deren Realisierung wurde in der Originalversion aber nicht durchgeführt; statt POSTGRES hätte ebenso gut eine rein relationale Datenbank verwendet werden können. Der Grund ist unter anderem darin zu sehen, daß diese erweiterten Konzepte noch zu datenbank-speziell realisiert wurden; eine Übertragung auf eine allgemeine Programmiersprache (insbesondere LISP) bereitet damit noch erhebliche Schwierigkeiten.

6.3 Schicht 2: Die LIBPQ Schicht

Schicht 1 und die Schichten 2–4 laufen jeweils in einem eigenen UNIX Prozeß. Der Schicht 1 zugeordnete Prozeß wird als *Backend*-Prozeß bezeichnet, der zur Schicht 2–4 gehörige heißt *Frontend*-Prozeß [Wensel 93, S. 30]. Durch diese Entkopplung können zum einen an den *Backend*-Prozeß gerichtete Datenbankanfragen des *Frontend*-Prozesses parallel zu diesem verarbeitet werden; zum anderen führt ein Abbruch des *Frontend*-Prozesses nicht automatisch zum Abbruch des auf die Datenbank zugreifenden *Backend*-Prozesses, der statt dessen kontrolliert terminieren kann.

Die Aufgabe der 2. Schicht besteht darin, den Start des *Backend*-Prozesses zu veranlassen und zwischen dem *Frontend*- und dem *Backend*-Prozeß eine Verbindung über einen UNIX *Socket* [man socket] herzustellen.

6.4 Schicht 3: Die SOH Schicht

Die in [Rowe 87] gegebene Konzeption und die Originalversion von SOH unterscheiden sich zum Teil extrem; auf signifikante Differenzen wird im folgenden Text hingewiesen. Sofern im folgenden Text nur Eigenschaften der Originalversion beschrieben werden, gelten dieselben Eigenschaften auch für die portierte Version.

6.4.1 Der *Cache*

Der *Cache* ist als assoziativer transienter Zwei-Wege-Speicher in der 3. Schicht realisiert; er bildet sowohl die innerhalb des persistenten Speichers zur Identitatsrepräsentation verwendeten *objids* auf die transiente Identitätsrepräsentation eines persistenten Objektes ab als auch umgekehrt. In der Konzeption [Rowe 87, S. 14–21] werden verschiedene Strategien zur Verwaltung des *Cache* erläutert, speziell im Hinblick auf Konsistenzerhaltung des *Cache* bei Mehr-BenutzerInnen-Betrieb; realisiert wurde keiner der dort angegebenen Algorithmen.

Terminierung für Objektgraphen mit Zyklen

Bei einer Übertragung des Zustands eines Objektes zwischen dem transienten und persistenten Speicher ergibt sich aus der transitiv referenzierten Hülle des Objektes ein gerichteter Graph, dessen Knoten aus den zu übertragenen Objekten und dessen Kanten durch die in den *Slots* enthaltenen Referenzen zwischen den Objekten gebildet werden. Dieser Graph muß nicht zyklenfrei sein; um in diesem Fall die Übertragung der Zustände der zum Graphen gehörigen Objekte zu terminieren, werden bereits übertragene Objekte als ‚gespeichert' markiert. Wird beim Traversieren des Graphen ein als ‚gespeichert' markiertes Objekt aus einem *Slot* referenziert, wurde dessen Zustand bereits übertragen; andernfalls wird das referenzierte Objekt als ‚gespeichert' markiert und sein Zustand wird übertragen.

Realisiert wurde in der Originalversion die Markierung ‚gespeichert' als Eintrag in den *Cache*, d.h. wenn die Instanz vom *Cache* referenziert wird, gilt sie als bereits gespeichert. Damit werden nach dem ersten Speichern eines Objektes die Zustände der transitiv vom gespeicherten Objekt referenzierten Instanzen im aktuellen LISP Prozeß immer nur ein einziges Mal gespeichert, auch wenn sich ihre Zustände geändert haben sollten.

6.4.2 Spezialisierte Metaobjekt-Klassen

In der Konzeption werden die Relationen **DBObject** und **DBClass** definiert [Rowe 87, S. 10]. Die Relation **DBObject** repräsentiert die Zustände der *Slots*, die sich aus der an alle aktiv-persistenten Klassen vererbten abstrakten Superklasse **dbobject** ergeben. Die Relation **DBClass** repräsentiert die Klassen-Metaobjekte von aktiv-persistenten Klassen.

In der Originalversion fand dies seinen Niederschlag in der Klasse **dbobject**, die im Verlauf der Definition einer aktiv-persistenten Klasse immer an die definierte Klasse vererbt wird, und der spezialisierten Klassen-Metaobjekt-Klasse **dbclass** zur Repräsentation von aktiv-persistenten Klassen.

In der portierten Version werden die direkten bzw. effektiven *Slots* von aktiv-persistenten Klassen durch *Slot*-Metaobjekte der spezialisierten *Slot*-Metaobjekt-Klassen **dbclass-standard-direct-slot-definition** bzw. **dbclass-standard-effective-slot-definition** repräsentiert.

6.4.3 Objektrepräsentation

Objekte in der POSTGRES Datenbank

Passiv-persistente Objekte Nach der in [Rowe 87, S. 12] angegebenen Konzeption soll zwischen LISP- und POSTGRES-Typen eine Abbildung festgelegt werden können. Diese Abbildung sollte persistent in der Datenbank gehalten werden; für die beschriebenen Konvertierungen war geplant, typabhängig die Namen von zwei Funktionen abzulegen, die Objektzustände zwischen den Repräsentationsformen von Common LISP und POSTGRES konvertieren (Tabelle 6.2).

Common LISP	**POSTGRES**	**Beschreibung**
fixnum	int4	4 Byte Ganzzahl.
float	float	4 Byte Fließkommazahl.
(simple-array 'char)	char[]	Zeichenkette variabler Länge.
Symbol	char[]	Eine Zeichenkette, deren Evaluierung das Symbol ergibt (z.B. `'x` für das Symbol `x`).
(passiv-persistente)[a] Objekte	char[]	Eine Zeichenkette, deren Evaluierung das Objekt ergibt.

[a] In SOH werden passiv-persistente Objekte als lokale Objekte bezeichnet.

Tabelle 6.2: Beispiele für die Abbildung zwischen Common LISP- und POSTGRES-Typen (nach [Rowe 87, S. 12])

In der Konzeption wird als Begründung für die Verwendung einer relationalen Datenbank als persistenten Speicher unter anderem ausgeführt [Rowe 87, S. 2]:

> A beneficial side-effect is that programs written in a conventional programming language can [...] access the data stored in the object hierarchy.

Der Zugriff auf den Zustand von Daten, die durch Evaluierung einer Zeichenkette generiert werden, bedingt die Existenz einer evaluierenden Instanz, die in konventionellen Programmiersprachen im allgemeinen nicht verfügbar ist. Die einzige Möglichkeit, konventionellen Programmiersprachen Zugriff auf den Zustand von Daten zu verschaffen, besteht darin, die Daten selbst und keine sie generierenden Ausdrücke im persistenten Speicher zu repräsentieren.

Realisiert wurde in der Originalversion für Instanzen von Basistypen nur die allgemeinste Möglichkeit der Repräsentation als Zeichenkette, deren Evaluierung das Datum generiert (*Handle*, S. 26). Die Notwendigkeit dieser Vorgehensweise resultiert aus dem Gegensatz der geforderten statischen Typisierung der Attribute zu der dynamischen Typisierung von LISP Objekten. Durch die Typisierung auf Zeichenketten wird erreicht, daß das Attribut beliebige S-Ausdrücke enthalten kann; die Evaluierung eines S-Ausdrucks kann dann ein dynamisch typisiertes LISP Objekt in der transienten LISP Umgebung erzeugen. In diese Repräsentationsform sind auch implizit die Referenzen auf andere Objekte eingebettet; die in relationalen Datenbanken übliche Art und Weise der Referenz zwischen zwei Objekten über den in jeweils einem Attribut enthaltenem gleichen Wert wurde nicht realisiert.

Aktiv-persistente Instanzen Aktiv-persistente Objekte in SOH werden wie bereits in Abschnitt 5.3.1 (S. 53) beschrieben repräsentiert. Die die *Slots* repräsentierenden Attribute werden in der Originalversion grundsätzlich auf den Typ **Zeichenkette** typisiert; der *Slot*-Zustand eines persistenten Objektes wird in der POSTGRES Datenbank in einer Zeichenkette gespeichert, deren Evaluierung den *Slot*-Zustand im transienten Speicher wiederherstellt.

Im transienten Speicher wird die Identität eines aktiv-persistenten Objektes durch ein Tupel (<*Relationsname*> <*objid*>) repräsentiert [Rowe 87, S. 10]. Der <*Relationsname*> ist der Name der für die Speicherung der aktiv-persistenten Instanzen angelegten Datenbankrelation. Die <*objid*> wird entweder beim Speichern des aktiv-persistenten Objektes vergeben oder auf die *objid* gesetzt, mit der das aktiv-persistente Objekt aus der Datenbank geladen wurde. SOH verwendet damit typstrukturierte Bezeichner zur Identitätsrepräsentation von aktiv-persistenten Instanzen (Abschnitt 3.2.2, S. 19) und berücksichtigt damit nicht die von CLOS gegebene Möglichkeit der Änderung der Klassenzugehörigkeit eines Objektes.

Listen, die von einem aktiv-persistenten Objekt referenziert werden Listen werden in der Originalversion je nach Art ihrer Referenzierung unterschiedlich gespeichert. Eine Liste, die vom *Slot* eines aktiv-persistenten Objektes referenziert wird, wird als Zeichenkette in ihrer Druckrepräsentation (*print representation*) in das den *Slot*-Zustand repräsentierende Attribut geschrieben.

Aktive Objekte und Listen, die nicht von einem aktiv-persistenten Objekt referenziert werden In der Konzeption von SOH wurde ursprünglich festgelegt, daß lediglich aktiv-persistente Instanzen und Objekte einiger Basisklassen persistent werden können [Rowe 87, S. 9, 12]. In der Originalversion wurde SOH um das nur zum Teil in SOH integrierte Subsystem `store` erweitert, um aktive Objekte und Listen, die nicht von einem aktiv-persistenten Objekt referenziert werden, persistent zu halten. Das Subsystem `store` führt eine eigene, von SOH unabhängige Verwaltung von persistenten Listen und Objekten durch. Eine Konsequenz daraus ist, daß Referenzen innerhalb von Listen oder aktiven Objekten auf aktiv-persistente Objekte nicht identitätserhaltend gespeichert werden können. Sowohl die Repräsentationsform in der Datenbank als auch die Identitätsrepräsentation unterscheidet sich von der für aktiv-persistente Objekte. Die Repräsentationen aller Instanzen, die vom Subsystem `store` gespeichert werden, befinden sich in der Relation **local**.

LISP Ausdruck	loid	make
	CONS4711	(let ((c (list 1))) (register-object 'CONS4711 c) (rplacd c (load-object 'CONS4712)) c)
	CONS4712	(let ((c (list 2))) (register-object 'CONS4712 c) c)
(make-instance 'foo :slot-1 'a :slot-2 '(1 2))	FOO4713	(let ((o (make-instance 'foo))) (register-object 'FOO4713 o) (setf (slot-value o 'slot-1) 'a) (setf (slot-value o 'slot-2) (load-object 'CONS4711)) o)

Tabelle 6.3: Beispiele Relation **local**

Die Relation local In der ausschließlich vom Subsystem `store` genutzten Relation **local** entspricht ein Tupel genau einem gespeichertem referenzierbarem Common LISP Datum; sie hat zwei Attribute.

Attribut loid Das Attribut **loid** (*local objid*) enthält die vom Subsystem `store` benutzte Identitätsrepräsentation des persistenten Objektes. Sie besteht aus einer Konkatenation des Klassennamens des persistenten Objektes mit einer Nummer (typstrukturierter Bezeichner, Abschnitt 3.2.2, S. 19). In der Realisierung wird für die Generierung der *local objid* die Funktion **gensym** [CLtLII, S. 245] benutzt, die keine Eindeutigkeit über mehrere LISP Prozesse hinweg garantiert.

Attribut make Im Attribut **make** wird ein S-Ausdruck (*S-expression*) abgelegt, dessen Evaluierung die transiente Repräsentation des persistenten Objektes regeneriert. Der S-

Ausdruck ist so aufgebaut, daß die transienten Repräsentationen der transitiv referenzierten Objekte im Verlauf der Evaluierung geladen und anschließend in das referenzierende Objekt eingetragen werden.

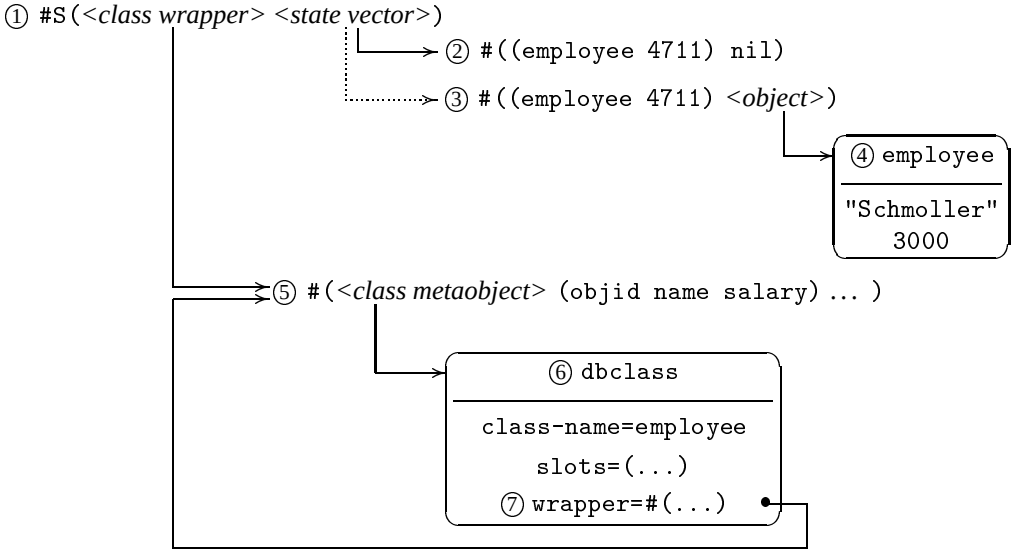
Das Beispiel in Tabelle 6.3 zeigt, wie eine aktive Instanz der Klasse **foo** in der Relation **local** gespeichert wird. Die Funktion **load-object** läd die transiente Repräsentation eines persistenten Objektes entweder aus dem *Cache* oder, wenn sie dort nicht gefunden wurde, aus der Relation **local**; ein Aufruf der Funktion **register-object** trägt die transiente Repräsentation in den *Cache* ein.

Sonstige Daten Alle Instanzen von bisher nicht erwähnten Typen werden von SOH in der POSTGRES Datenbank durch einen S-Ausdruck repräsentiert, dessen Evaluierung die transiente Repräsentation regeneriert (siehe auch die letzte Zeile von Tabelle 6.2, S. 72).

Repräsentation von persistenten Objekten im transienten Speicher

Die Konzeption macht keine Angaben zur Repräsentation von persistenten Objekten im transienten Speicher.

Aktiv-persistente Objekte In der Originalversion greift die Repräsentation von aktiv-persistenten Objekten im transienten Speicher in die Interna des Objektsystems ein (Abbildung 6.4). Instanzen werden in CLOS durch ein Tupel (<*class wrapper*> <*state vector*>) repräsentiert.



①: Objekt mit den Referenzen auf den *Class Wrapper* ⑤ und den Zustandsvektor ② bzw. ③
②: Zustandsvektor mit der *objid* des Objektes direkt nach dem Laden
③: Zustandsvektor mit *objid* und nachgeladener Repräsentation ④ nach dem ersten *Slot*-Zugriff
⑤: *Class Wrapper* mit den Referenzen auf das Klassen-Metaobjekt ⑥ und die Liste der *Slot*-Namen

Abbildung 6.4: Transiente Repräsentation von aktiv-persistenten Objekten

Der <*state vector*> enthält den Zustand des Objektes; die Länge des Vektors entspricht der Anzahl der *Slots*. Um Änderungen der durch das Klassen-Metaobjekt festgelegten Struktur einer

Instanz detektieren zu können, verweist die Instanz ① nicht direkt auf das Klassen-Metaobjekt ⑥, sondern auf einen *Class Wrapper* ⑤, der einen Verweis auf das Klassen-Metaobjekt sowie eine minimale Strukturdefinition bestehend aus einer Liste der *Slot*-Namen enthält. Bei einer Änderung der im Klassen-Metaobjekt ⑥ enthaltenen Strukturdefinition wird von CLOS im Klassen-Metaobjekt ein neuer *Class Wrapper* eingetragen ⑦; eineÄnderung der Strukturdefinition wird beim nächsten Zugriff auf den Objektzustand dadurch detektiert, daß der von der Instanz referenzierte *Class Wrapper* ⑤ und der im Klassen-Metaobjekt abgelegte *Class Wrapper* ⑦ nicht identisch sind. Die im *Class Wrapper* der Instanz enthaltene minimale Strukturdefinition wird benutzt, um den Zustandsvektor des Objektes an die neue Strukturdefinition anzupassen.

In SOH wird die Allozierungs-Methode für aktiv-persistente Objekte spezialisiert; sie ersetzt den Zustandsvektor durch einen Vektor mit konstanter Länge, der unter anderem die *objid* sowie ein zunächst auf `nil` gesetztes Element enthält ②. Beim ersten Zugriff auf den Objektzustand wird von den in SOH spezialisierten Zugriffs-Methoden die transiente Repräsentation des Objektes ④ nachgeladen und in dieses Element eingetragen; die Zugriffe auf den Objekzustand werden auf die nachgeladene Repräsentation ‚umgelenkt'. Der *Slot* mit der *objid* wird nicht in der nachgeladenen Instanz repräsentiert; die spezialisierten Zugriffs-Methoden referenzieren für diesen *Slot* die im Zustandsvektor ② bzw. ③ enthaltene *objid*.

Diese Lösung nutzt die Interna des zur Realisierung von SOH benutzten Objektsystems PCL aus; sie ist daher weder portabel noch MOP-konform. Die Portierung bereitete aber kaum Probleme, da CLOS aus der Weiterentwicklung des Objektsystems PCL hervorging und sich die Interna nur relativ wenig geändert haben. Inzwischen bietet das MOP vollständig die Möglichkeit, die Instanzdaten eines Objektes selbst zu verwalten [AMOP, S. 99], ohne Interna des Objektsystems benutzen zu müssen; diese Möglichkeit wurde in der portierten SOH Version nicht realisiert.

6.4.4 Schnitt zwischen Objekten

Ein Schnitt war in der ursprünglichen Konzeption nicht vorgesehen und wurde in der Originalversion nicht realisiert; um zumindestens das Speichern zuvieler transienter Klassen-Metaobjekte zu verhindern, werden in der portierten Version beim Start von SOH alle Klassen-Metaobjekte, die nicht aktiv-persistente Klassen repräsentieren, in den *Cache* eingetragen und damit als ‚gespeichert' markiert (Abschnitt 6.4.1, S. 71).

6.4.5 Protokolle

Definition von aktiv-persistenten Klassen

In der Konzeption wird auf die Art und Weise der Definition von aktiv-persistenten Klassen nicht eingegangen. In der Originalversion von SOH werden aktiv-persistente Klassen durch das Makro **defdbclass** [Schank et al. 90, S. 5-77] definiert; die dort realisierte Funktionalität wurde in der portierten Version in die Initialisierung des die aktiv-persistente Klasse repräsentierenden Klassen-Metaobjektes verschoben. Die Definition einer aktiv-persistenten Klasse erfolgt jetzt über die Klassenoption `(:metaclass dbclass)`.

Im Verlauf der Initialisierung des Klassen-Metaobjektes wird es in der Datenbank gespeichert, d.h. es wird die sich aus der Klassendefinition ergebene Relation zur Ablage der Instanzen in der Datenbank angelegt. Aus dem vorliegenden Code der Originalversion ist ersichtlich, daß dieser Vorgang über den an das Makro **defdbclass** übergebenen Schlüsselwort-Parameter

`creation-mode` beeinflußbar ist; in der portierten Version sollte diese Aufgabe die zusätzliche Klassenoption `:creation-mode` übernehmen.

Ursprünglich von mir geplant war, daß in der portierten Version bei der Initialisierung eines Klassen-Metaobjektes einer aktiv-persistenten Klasse außer den Standard-Klassen- und *Slot*-Optionen [CLtLII, S. 822, 823] von SOH zusätzliche Optionen ausgewertet werden. Da bereits im Verlauf der Portierung klar wurde, daß SOH die gestellten Anforderungen nicht erfüllen kann, wurde zwar eine Auswertung von zusätzlichen Klassen- und *Slot*-Optionen realisiert, die Ergebnisse der Auswertung werden aber nicht weiter verwendet; die hier für die Auswertung erstellten spezialisierten Methoden werden in modifizierter Form in dem von mir realisierten persistenten System im Protokoll 7.19 **(defclass)** (S. 111) eingesetzt. Die folgenden beiden Abschnitte erläutern trotzdem die zusätzlich geplanten Optionen.

Zusätzliche Klassenoption `:creation-mode` Mit der Klassenoption `:creation-mode` könnte die Art der Schemaentwicklung für aktiv-persistente Klassen ausgewählt werden:

Klassenoption `(:creation-mode always-create)` Bei der Initialisierung des transienten Klassen-Metaobjektes wird die Relation zur Speicherung der persistenten Repräsentationen von Instanzen der aktiv-persistenten Klasse immer in der Datenbank angelegt; eine eventuell bereits vorhandene Relation wird überschrieben.

Klassenoption `(:creation-mode never-recreate)` Bei der Initialisierung des transienten Klassen-Metaobjektes wird die aus dem Klassen-Metaobjekt bestimmte Relation zur Speicherung der persistenten Repräsentationen von Instanzen der aktiv-persistenten Klasse mit der in der Datenbank vorgefundenen Relation gleichen Namens verglichen; sind sie nicht gleich, wird ein Fehler signalisiert.

Klassenoption `(:creation-mode rename-create)` Die Relation wird wie bei der Klassenoption `(:creation-mode always-create)` erläutert angelegt, eine eventuell bereits vorhandene Relation wird aber nicht gelöscht sondern umbenannt.

Zusätzliche *Slot*-Option `:transient` Mit dieser Option könnte für den definierten *Slot* sein Zustand für alle Instanzen der aktiv-persistenten Klasse als transient deklariert werden, d.h. der *Slot*-Zustand wäre nicht in der Datenbank repräsentiert worden.

Erzeugen von aktiv-persistenten Instanzen

In der Originalversion wird für eine aktiv-persistente Instanz bei ihrer Erzeugung im transienten Speicher eine minimale transiente Repräsentation alloziert, die zunächst nur SOH-interne Informationen zur Referenzierung der persistenten Repräsentation des Objektes sowie einige an das Objekt gebundene Statusinformationen enthält. Wurde die transiente Repräsentation im Verlauf einer Lokalisierung des Objektes (Abschnitt 6.4.6, S. 78) erzeugt, exisitiert bereits eine persistente Repräsentation des Objektes; in diesem Fall wird in die transiente Repräsentation die *objid* der persistenten Repräsentation eingetragen. Entstand die transiente Repräsentation nicht im Verlauf einer Lokalisierung, wird abhängig vom dem bei der Erzeugung angegebenen Repräsentationsmodus (*update mode*) gegebenenfalls eine persistente Repräsentation alloziert und dessen *objid* in die transiente Repräsentation eingetragen.

Repräsentationsmodus von aktiv-persistenten Instanzen In der Konzeption wurde festgelegt, daß bei der Erzeugung einer aktiv-persistenten Instanz aus mehreren Möglichkeiten für ihre Repräsentationsform und das daran gekoppelte Verhalten bei Zugriff auf den Zustand ausgewählt werden kann [Rowe 87, S. 17]; dies wurde in der Originalversion auch realisiert. Der Repräsentationsmodus bezieht sich auf den gesamten Zustand genau eines Objektes; er kann nicht auf Klassenebene für den Zustand aller Instanzen einer aktiv-persistenten Klasse oder für die Zustände einzelner *Slots* angegeben werden.

Modus `local-copy` Der Zustand des Objektes wird nur im transienten Speicher repräsentiert; es wird keine persistente Repräsentation alloziert. Zustandsänderungen wirken sich daher nur auf die transiente Repräsentation aus.

Modus `direct-update` Der Zustand des Objektes wird im transienten und im persistenten Speicher repräsentiert; Zustandsänderungen der transienten Repräsentation werden sofort in die persistente Repräsentation propagiert. In der Konzeption war vorgesehen, daß Zustandsänderungen der persistenten Repräsentation durch andere Prozesse ebenfalls in die transiente Repräsentation propagiert werden; eine Realisierung dessen erfolgte nicht.

Modus `deferred-update` Der Zustand des Objektes wird im transienten und im persistenten Speicher repräsentiert; Zustandsänderungen wirken sich zunächst nur auf die transiente Repräsentation aus, bis die BenutzerInnen explizit eineÜbertragung des Zustands von der transienten in die persistente Repräsentation anfordern.

An die objekt-erzeugende Funktion **make-instance** wird der Repräsentationsmodus als das Initialisierungs-Schlüsselwort-Argument (*initarg*) `:dbmode` übergeben.

Zugriff auf den Objektzustand

In der Konzeption wird auf den Zugriff auf den Zustand von persistenten Objekten nicht eingegangen.

Passiv-persistente Objekte In der Originalversion wird auf passiv-persistente Objekte mit *Swizzling* zugegriffen. Die transiente Repräsentation eines passiv-persistenten Objektes wird entweder explizit durch einen Aufruf einer entsprechenden Funktion geladen oder implizit erzeugt, wenn das passiv-persistente Objekt von einer zu ladenden Instanz referenziert wird. Auf die transienten Repräsentationen kann mit den üblichen Common LISP Funktionen zugegriffen werden. Der Abgleich zwischen der transienten und persistenten Repräsentation eines passiv-persistenten Objektes erfolgt explizit durch Aufruf einer entsprechenden Funktion.

Aktiv-persistente Instanzen Für Instanzen von aktiv-persistenten Klassen erfolgt der Zugriff auf den Objektzustand in der Originalversion über Methoden der generischen Funktionen **slot-...-using-class**, die auf die Klassen-Metaobjekt-Klasse **dbclass** spezialisiert sind. Die transiente Repräsentation mit dem Zustand des Objektes wird beim ersten Zugriff auf den Objektzustand alloziert; für lokalisierte Objekte wird der Zustand aus der persistenten in die transiente Repräsentation kopiert (Abschnitt 6.4.3, S. 74). Die spezialisierten Zugriffs-Methoden referenzieren je nach Repräsentations-Modus der Instanz die transiente oder persistente Repräsentation.

6.4.6 Lokalisierung

Passiv-persistente Objekte

In der Originalversion können lediglich die vom Subsystem `store` verwalteten passiv-persistenten Objekte mit der dort vergebenen *objid* lokalisiert werden.

Aktiv-persistente Instanzen

In SOH können aktiv-persistente Instanzen inhaltsorientiert lokalisiert werden. Realisiert ist in der Originalversion die Möglichkeit, alle Objekte einer aktiv-persistenten Klasse mit einem bestimmten *Slot*-Zustand zu suchen.

6.4.7 Datenbank-Konzepte in SOH

Durch die Verwendung von POSTGRES als persistenten Speicher würden sich prinzipiell umfangreiche Möglichkeiten zur Datenmanipulation ergeben; die Originalversion nutzt weder die angebotenen Möglichkeiten noch werden sie den BenutzerInnen von SOH zur Verfügung gestellt.

6.5 Zusammenfassung und Bewertung

SOH realisiert Persistenz für Common LISP Objekte unter Verwendung der erweitert-relationalen Datenbank POSTGRES. Die in [Rowe 87] vorgestellte Konzeption und die vorliegende Originalversion unterscheiden sich zum Teil beträchtlich; insbesondere wurde die in der Konzeption vorgeschlagene Nutzung der Möglichkeiten von POSTGRES nicht realisiert. Die in der Konzeption nur andeutungsweise erläuterte Verarbeitung von passiv-persistenten Instanzen wird durch das nur zum Teil in SOH integrierte Subsystem `store` realisiert.

Meine abschließende Feststellung ist, daß SOH in der vorliegenden Realisierung für den beabsichtigten Einsatz nicht geeignet ist:

- Durch die Repräsentation von *Slots* durch statisch typisierte Attribute in der Datenbank müssen in aktiv-persistenten Objekten die *Slots* ebenfalls statisch typisiert werden.
- Passiv-persistente Objekte werden als unevaluierte S-Ausdrücke gespeichert; die Generierung der S-Ausdrücke und ihre Übertragung in die Datenbank ist sehr aufwendig. Von mir durchgeführte Messungen ergaben eine Übertragungsrate von ca. 50 Byte/Sekunde. Damit dauert das Speichern einer durch 84 Zeichen repräsentierten *cons*-Zelle (Tabelle 6.3, Zeile **loid** CONS4711, S. 73) bereits 1,68 Sekunden.
- Die Referenzen eines Objektes sind bei passiv-persistenten Objekten implizit in den das Objekt repräsentierenden unevaluierten S-Ausdruck eingebettet; die Auflösung einer Referenz bedingt die Materialisierung des Objektes im transienten Speicher. Durch die damit beanspruchten Ressourcen kann das Traversieren großer Objektgraphen (z.B. zur Speicherrückgewinnung von passiv-persistenten Objekten über Erreichbarkeit durch aktiv-persistente Objekte) unmöglich werden.
- Die gewählte Repräsentationsform für Objekte über die Evaluierung von S-Ausdrücken macht eine sinnvolle Nutzung der von der Datenbank angebotenen Manipulationskriterien

unmöglich; da in einer Objektrepräsentation in der relationalen Datenbank nicht der Objektzustand selbst, sondern seine ‚Umschreibung' enthalten ist, kann die Datenbank nicht nach dem Objektzustand selbst suchen. Die in [Rowe 87, S. 2] geführte Argumentation der Nutzung der Datenbankfunktionalitäten ist damit hinfällig.

Die vorliegende Originalversion wurde mit dem Objektsystem PCL erstellt, aus dem später CLOS hervorging. Es wird oft auf PCL Interna zurückgegriffen; eine vollständige Anpassung an den CLOS Standard wäre nur durch eine erneute Realisierung der in [Rowe 87] geschilderten Konzepte möglich gewesen. Da ich die Probleme der Integration einer relationalen Datenbank in ein persistentes System für Common LISP Objekte für größer halte als die sich ergebenden Vorteile, verfolgte ich diesen Weg nicht weiter.

Kapitel 7

Persistente LISP Objekte

Die Zielsetzung für die Entwicklung von PLOB! (*P*ersistente *L*ISP *Ob*jekte) besteht in einem Einsatz des Systems im Rahmen des Projekts OSCAR zur langlebigen Speicherung von LISP Objekten, die unter Umständen sehr große Daten referenzieren, wie z.B. Satellitenbilder hoher Auflösung; ebenso kann die Anzahl der zu speichernden Objekte sehr groß werden. Aus diesem Grund soll das persistente System auch eine Speicherrückgewinnung haben. Da ein Teil der persistenten Instanzen geografische Objekte repräsentiert, sollen persistente Objekte assoziativ möglichst mit mehrdimensionalen Schlüsseln gesucht werden können.

Die Anforderungen an ein persistentes System im Rahmen dieser Arbeit werden von den von mir betrachteten vorhandenen Systemen *Shared Object Hierarchy* (SOH, Kapitel 6, S. 67) und *Persistent CLOS-Objects* (PCLOS [Paepke 91a]) nicht erfüllt.

7.1 Überblick

Das System PLOB! bietet orthogonale Persistenz für den Zustand fast aller in einem Common LISP-System vorkommenden Daten mit Ausnahme von Funktionscode. Durch Orthogonalität wird eine hohe Transparenz erreicht. Das System nimmt zunächst an, daß beim Speichern eines Objektes grundsätzlich alle davon referenzierten Instanzen ebenfalls persistent gehalten werden sollen. Es werden hierarchische, auf Modularisierungsebene geordnete Deklarationen benutzt, um festzustellen, welche referenzierten Instanzen tatsächlich persistent gehalten werden sollen; damit kann die Anzahl der persistent gehaltenen Objekte begrenzt werden. Die angebotenen Datenbankfunktionalitäten können ebenfalls deklarativ eingebunden werden.

Für Instanzen der Basistypen wird Persistenz über Funktionen zur Verfügung gestellt; die Namen der Funktionen wurden aus dem in [CLtLII] spezifizierten Common LISP Standard für transiente Objekte abgeleitet.

Die Struktur eines persistenten Objektes wird ebenfalls als persistentes Objekt repräsentiert; PLOB! sorgt dafür, daß sich beim Abspeichern eines transienten Objektes seine aus der Klasse des Objektes abgeleitete Strukturrepräsentation ebenfalls im persistenten Speicher befindet.

Außer Persistenz wurden noch zusätzliche als nützlich angesehene Funktionalitäten in PLOB! integriert:

Sitzungen Auf persistente Objekte kann gleichzeitig durch mehrere Sitzungen zugegriffen werden.[1]

Transaktionen Alle Zugriffe auf persistente Objekte laufen innerhalb einer an ihre Sitzung gebundenen Transaktion ab.

Objektsperren (*locking*) Ein persistentes Objekt wird transaktionsgebunden durch die lesende bzw. modifizierende Sitzung gesperrt; Konflikte durch mehrere auf ein gesperrtes Objekt zugreifende Sitzungen werden erkannt und aufgelöst.

B-Bäume Zum assoziativen Zugriff auf persistente Objekte wurden persistente B-Bäume realisiert; diese können ähnlich wie *Hash*-Tabellen verwendet werden.

Indizes über *Slots*; assoziativer Zugriff Für einen *Slot* von aktiv-persistenten Instanzen kann deklarativ ein Index definiert werden, mit dem assoziativ auf aktiv-persistente Instanzen zugegriffen werden kann.

Modifikationen an Klassen-Metaobjekten (*schema evolution*) Änderungen an einer transienten Klassendefinition werden in den persistenten Speicher propagiert; die Struktur der betroffenen persistenten Objekte wird nach den üblichen CLOS Regeln auf den Stand der neuen Klassendefinition gebracht.

Der folgende Text erläutert die oben aufgeführten Punkte im Einzelnen.

7.2 Entwurfsprinzipien

Beim Entwurf des Gesamtsystems wurden die folgenden Prinzipien berücksichtigt.

7.2.1 Common LISP-gemäße Semantik

Um eine möglichst einfache Schnittstelle zu Common LISP zu erhalten, wurde dessen Datenmodellierung übernommen. An einigen Stellen ergaben sich durch die Erweiterung der Lebensdauer von Objekten über den transienten LISP Prozeß hinaus Probleme mit der Semantik; dies betrifft in erster Linie Klassen, deren Instanzen von LISP als eindeutig angesehen werden, wie Symbole und Pakete. Für diese Klassen wurde eine entsprechende Semantik definiert.

7.2.2 Schichtenmodell

Das Gesamtsystem ist in aufeinander aufbauende Schichten gegliedert; die verschiedenen Abstraktionsebenen sind damit klar getrennt. Ebenso lassen sich untere Schichten austauschen, ohne die darüber angesiedelten Schichten ändern zu müssen.

7.2.3 Portabilität

Aus interner Sicht war Portabilität sowohl der System-Quelltexte als auch der im persistenten Speicher abgelegten Objekte ein wichtiges Grundprinzip.

[1] Für Mehr-BenutzerInnen-Zugriff siehe Abschnitt 7.9, S. 145.

Portabler Code

Der C Code wurde nach ANSI Standard erstellt [Kernighan et al. 88]. Der Common LISP Code hält sich soweit möglich an die in [CLtLII] und [AMOP] gegebenen Spezifikationen; systemspezifischer LISP Code wurde, soweit sinnvoll, in einem einzigen Modul zusammengefaßt. Da bisher keine systemübergreifende Standardisierung einer Schnittstelle zwischen Common LISP und C erfolgte, wurde darauf geachtet, zwischen Common LISP und C nur einfache Daten, d.h. Instanzen einiger wenigen Basistypen, auszutauschen. Ausnahmen erfolgten lediglich dann, wenn eine Transformation in eine für den Austausch geeignete Instanz einen unverhältnismäßig hohen Effizienzverlust bedeutet hätte.

Portable persistente Objekte

Um auf die im persistenten Speicher enthaltenen persistenten Objekte mit verschieden realisierten Applikationsschichten zugreifen zu können, wurde darauf Wert gelegt, bereits in den unteren Schichten eine vollständige Typisierung der persistenten Objekte einzuführen; jedes persistente Objekt ist entweder Instanz eines Basistyps oder enthält eine Referenz auf eine (persistente) Beschreibung seiner Klasse. Klassen-Beschreibungsobjekte selbst sind auch persistente Objekte. Die im persistenten Speicher enthaltenen Objekte sind damit selbstbeschreibend (Abschnitt 5.2.2, S. 51).

7.2.4 Objekt-orientierte Programmierung

Die Vorteile objekt-orientierter Programmierung wurden bereits in anderen Arbeiten ausführlich geschildert [Goldberg et al. 89]; aus diesen Gründen wurde in der in C realisierten 2. Schicht (siehe nachfolgender Abschnitt 7.3) ein einfaches, nur in dieser Schicht benutztes Objektsystem realisiert, das direkt auf den Repräsentationen der persistenten Objekte arbeitet. Dieses Objektsystem ergänzt die C-typische statische Typisierung von Datenobjekten zurÜbersetzungszeit um eine dynamische Klassenbindung der persistenten Objekte zur Laufzeit.

Durch dieses Objektsystem können einige erstellte Komponenten mehrfach verwendet werden; so werden beispielsweise die in der 2. Schicht realisierten persistenten B-Bäume sowohl für verschiedene interne Tabellen genutzt als auch den BenutzerInnen extern zur Verfügung gestellt.

Ein anderer pragmatischer Aspekt war, daß durch die Common LISP-C-Kopplung kein Programm zur Fehlersuche (*debugger*) in der 2. Schicht verwendet werden konnte; durch die dynamische Klassenbindung können beispielsweise Typprüfungen von Objekten vorgenommen werden, um bei einem unerwartetem Typ kontrolliert einen Fehler signalisieren zu können.

7.2.5 Objektspeicher als persistenter Speicher

Die Verwendung eines Objektspeichers als persistenter Speicher resultiert aus den bisher in dieser Arbeit dargelegten Einschränkungen bei Verwendung einer relationalen Datenbank; sie ergeben sich aus der unterschiedlichen Datenmodellierung von Common LISP und relationalen Datenbanken:

- In Common LISP wird die Identität von Objekten repräsentiert; bei einer relationalen Datenbank muß eine Repräsentation der Objektidentität durch das persistente System auf höheren Schichten realisiert werden. Die Datenbank selbst hat keine Kenntnis von dieser Identitätsrepräsentation und kann damit keine Referenzen auflösen.

- Die Typinformation eines Objektes ist in Common LISP an das Objekt gebunden. In relationalen Datenbanken ist das Konzept eines Objektes im Sinne der objekt-orientierten Programmierung unbekannt; bei der in persistenten Systemen mit einer relationalen Datenbank benutzten Abbildung zwischen den Objektsystem- und Datenbank-Strukturen müssen im allgemeinen die die *Slot*-Zustände repräsentierenden Attribute statisch typisiert werden.

- Innerhalb einer relationalen Datenbank werden Tupel als Werte und nicht als‚echte' Objekte aufgefaßt. Die Identität eines Tupels ergibt sich damit aus seinem Zustand. In Common LISP hingegen ist die Identität eines Objektes unabhängig von seinem Zustand; es kann sowohl Werte oder Objekte referenzieren als auch selber referenziert werden. SOH und PCLOS umgehen dieses Problem durch Konvertierung von Referenzen zwischen Common LISP Daten in Datenbankwerte, die zur Auflösung der Referenz wiederum evaluiert oder konvertiert werden müssen; damit sind die Referenzen zwischen Objekten in der relationalen Datenbank aber nur noch implizit erkennbar.

  Referenzen zwischen transienten Common LISP Objekten werden über ihre transiente Identitätsrepräsentation (Adresse) hergestellt; daraus folgte für meinen Entwurf, daß Referenzen zwischen Objekten im persistenten Speicher auch über ihre dortige Identitätsrepräsentation (*objid*) hergestellt werden sollen.

- Die von SOH und PCLOS vorgenommenen Konvertierungen zwischen der transienten und der persistenten Repräsentationsform führen meist dazu, daß die persistente Repräsentationsform wesentlich ineffizienter als die transiente Form ist (siehe z.B. Tabelle 6.3, Zeile **loid** CONS4711, S. 73) und die Konvertierung relativ viel Ressourcen verbraucht.

  Aus diesem Grund paßte ich die Repräsentationsform von persistenten Objekten im Objektspeicher soweit möglich an die im transienten LISP vorgefundenen transienten Repräsentationsformen an; damit wird der Aufwand für Konvertierungen wesentlich geringer.

Um den Zeitaufwand für die Entwicklung des Systems zu begrenzen, wurde festgelegt, für den Objektspeicher ein fertiges Subsystem zu verwenden. Eine ausführliche Analyse mehrerer in Frage kommender Subsysteme findet sich in [Müller 91, S. 15–83]. In dieser Arbeit wurde auf einen erneuten Vergleich verzichtet; als einzige Möglichkeit der in [Müller 91, S. 85–87] getroffenen Auswahl des Objektspeichers blieb auch für diese Arbeit das System POSTORE (*p*ersistent *o*bject *store*) übrig.

7.3 Architektur

Abbildung 7.1 zeigt ein Schichtenmodell des PLOB! Systems. Die 1. Schicht wird aus dem übernommenen System POSTORE gebildet. Bei der Aufteilung der übrigen Schichten wurden alle Basisfunktionalitäten, wie Transaktionen sowie Allozieren, Deallozieren oder Sperren von persistenten Objekten, in der 2. Schicht realisiert. Die Konzeption erfolgte so, daß die Operationen über persistente Objekte oberhalb der 2. Schicht die in [Gray et al. 93, S. 6, 166] definierten ACID[2] genannten Eigenschaften für Transaktionen erfüllen:

> **Unteilbarkeit (*atomicity*):** Die von einer Transaktion vorgenommenen Zustandsänderungen sind unteilbar. [...]

[2] Von der deutschen Übersetzung UKSB wird hier Abstand genommen.

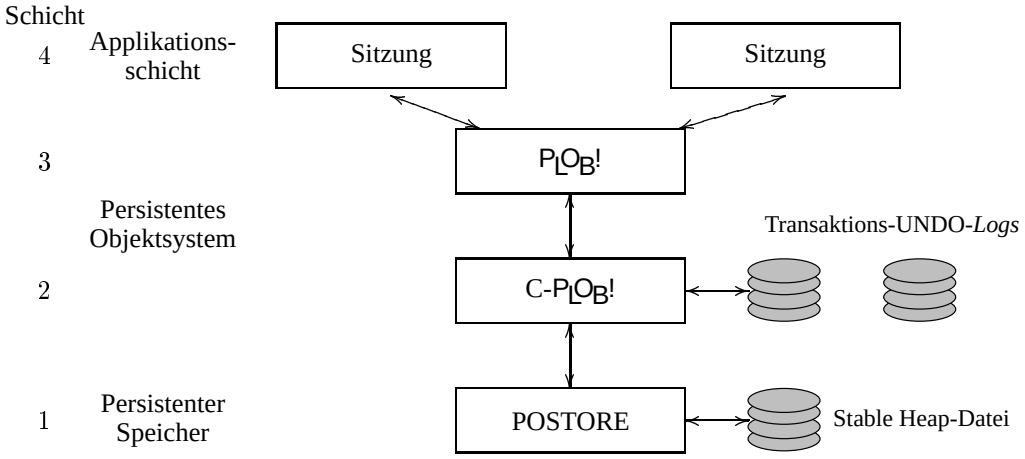


Abbildung 7.1: PLOB! Schichtenmodell

Konsistenz (*consistency*): Eine Transaktion ist eine korrekte Zustandsänderung. Die zu einer Gruppe zusammengefaßten Aktionen verletzen keinerlei Integritätsbeschränkungen des Zustands. [. . .]

Serialisierbarkeit (*isolation*): Auch bei gleichzeitig laufenden Transaktionen sieht es für jede Transaktion *T* so aus, als ob andere Transaktionen entweder vor *T* oder nach *T* ausgeführt würden.

Beständigkeit (*durability*): Nach Ende einer Transaktion (*commit*) überstehen die Zustandsänderungen Systemausfälle.

Die 3. Schicht stellt unter Benutzung der Basisfunktionalitäten der 2. Schicht die Verbindung zu den transienten Objekten in den Sitzungen der 4. Schicht her; so wird in der 3. Schicht z.B. beim Abspeichern eines transienten Objektes aus der 4. Schicht die transitiv referenzierte Hülle des Objektes traversiert und mit abgespeichert.

In den nachfolgenden Abschnitten werden die extern sichtbaren Konzepte der einzelnen Schichten erläutert.

7.4 Schicht 1: Die POSTORE Schicht

Das System POSTORE [Brown 92] [Müller 91, S. 63–83] realisiert einen persistenten *Heap* (Abschnitt 4.2.2, S. 40). Die Implementation erfolgte in C nach Kernighan & Ritchie- (d.h. nicht-ANSI-) Standard auf SUN Arbeitsplatzrechnern unter SunOS.

In [Müller 91, S. 63–83] wird das System POSTORE sehr ausführlich beschrieben und analysiert; die rein funktionale Schnittstelle wird vom Autor des Systems selbst in [Brown 92] nur sehr knapp beschrieben. An dieser Stelle werden lediglich die zum weiteren Verständnis wichtigen Konzepte geschildert.

Als persistenter *Heap* bietet POSTORE die Möglichkeit, einen persistenten Speicherbereich zu allozieren und zu modifizieren; ein solcher Speicherbereich wird im folgenden POSTORE-Vektor genannt. Ein allozierter POSTORE-Vektor ist genau dann persistent, wenn er erreichbar ist, d.h. wenn er transitiv vom Wurzelobjekt des POSTORE referenziert wird. Abbildung 7.2 zeigt die Struktur eines POSTORE-Vektors. POSTORE benutzt 2 Worte zu je 32 Bit Kopfinformationen. Danach folgen die frei zu belegenden Worte, die sich in einen Bereich mit *m* Referenzen auf

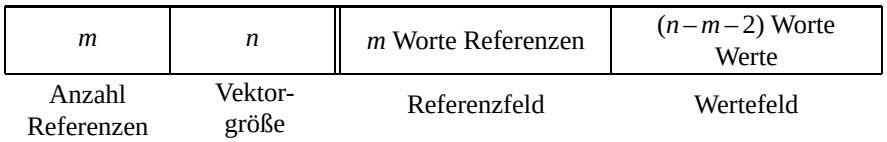


Abbildung 7.2: POSTORE-Vektor

andere POSTORE-Vektoren und einen Bereich mit $(n-m-2)$ Worten uninterpretierten Werten aufteilen. Referenziert wird ein POSTORE-Vektor über seine 32 Bit *objid*, die bei der Allozierung des Vektors vergeben wird; die *objid* selbst enthält keinerlei Informationen über die repräsentierten Instanzen und ist damit nach der in Abschnitt 3.2 (S. 15) gegebenenÜbersicht ein Surrogat. Die in dieser Schicht vergebene *objid* wird zur Unterscheidung der in höheren Schichten verwendeten *short objid* hier als *long objid* bezeichnet.

Es werden verschiedene Funktionen zum Zugriff auf einen POSTORE-Vektor angeboten. Die in [Müller 91, S. 63] erwähnte generische Architektur von POSTORE macht sich in dieser Schicht auch dadurch bemerkbar, daß die Zugriffsfunktionen keinerlei Gültigkeitsprüfungen ihrer Argumente durchführen. Konkurrenter Zugriff kann durch Setzen einer Schreibsperre synchronisiert werden. Die Auswertung, ob ein POSTORE-Vektor schreibgesperrt ist oder nicht, muß in einer höheren Schicht erfolgen; POSTORE führt bei schreibenden Zugriffen keinerlei Prüfungen in dieser Hinsicht durch.

Als Hilfsmittel für Persistenz benutzt POSTORE die Stable Heap-Datei, in der die Zustände der persistenten Objekte abgelegt werden.[3] Modifikationen an den POSTORE-Vektoren wirken sich nicht sofort auf die Stable Heap-Datei aus; alleÄnderungen der Vektoren sind temporär, bis sie durch den Aufruf einer Stabilisierungsfunktion in die Stable Heap-Datei übertragen werden.

Im POSTORE ist eine Speicherrückgewinnung realisiert, die nicht vom Wurzelobjekt referenzierte POSTORE-Vektoren freigibt; die Rückgewinnung wird nicht automatisch ausgelöst, sondern muß durch einen expliziten Aufruf der entsprechenden Funktion veranlaßt werden [Müller 91, S. 65].

7.4.1 Zusammenfassung der 1. Schicht

Da das System POSTORE als generisches System konzipiert wurde, bietet es keine Transaktionsverarbeitung für einzelne POSTORE-Vektoren; die Transaktionsverarbeitung wird statt dessen für den persistenten Speicher als gesamte Einheit durchgeführt, d.h. eine Transaktionsverarbeitung betrifft alle POSTORE-Vektoren. Ebenso ist die Transaktionsverarbeitung nicht transparent, da die entsprechenden Funktionen explizit aufgerufen werden müssen. Damit können höhere Schichten eine eigene Transaktionsverarbeitung realisieren.

Unteilbarkeit Die angebotenen Schreibsperren sind als Semaphoren [Jessen et al. 87, S. 143] realisiert; in Verbindung mit auf höheren Schichten emulierten Lesesperren [Müller 91, S. 73] [Brown 92, S. 2] kann Unteilbarkeit erreicht werden. Die Sperrung eines POSTORE-Vektors durch einen Semaphor muß von den BenutzerInnen beachtet werden, d.h. das POSTORE-benutzende Programm muß selbst prüfen, ob der Semaphor gesetzt ist und gegebenenfalls darauf warten, daß er wieder freigegeben wird.

[3] Die Stable Heap-Datei wird über den Aufruf der Betriebssystemfunktion `mmap` als virtueller Speicher in den Adreßraum des UNIX Prozesses eingeblendet [Müller 91, S. 81] [Gray et al. 93, S. 666] [man mmap]; die POSTORE-Vektoren werden dann direkt in diesem virtuellen Adreßraum alloziert und modifiziert.

Konsistenz Ein konsistenter Zustand des Gesamtsystems wird POSTORE durch Aufruf der Stabilisierungsfunktion signalisiert; das System POSTORE stellt dann sicher, daß der temporäre Zustand aller modifizierter POSTORE-Vektoren in die Stable Heap-Datei übertragen wird.

Bei Auftreten eines Fehlers wird der persistente Speicher beim nächsten Öffnen in den Zustand nach der letzten erfolgreichen Stabilisierung zurückgesetzt; diese Rücksetzung kann simuliert, aber nicht explizit veranlaßt werden.

Serialisierbarkeit Serialisierbarkeit kann erreicht werden, sofern der Zugriff auf die POSTORE-Vektoren von den BenutzerInnen entsprechend organisiert wird, indem z.B. alle Zugriffe innerhalb einer Zwei-Phasen-Transaktion stattfinden. POSTORE bietet alle Hilfsmittel (Schreibsperren) zur Realisierung dieses Konzeptes; eine konkrete Realisierung muß aber auf einer höheren Schicht erfolgen.

Nach der in [Gray et al. 93, S. 401] angegebenen Übersicht realisiert POSTORE eine Serialisierbarkeit von 0°.

Beständigkeit Die POSTORE-Vektoren werden in der Stable Heap-Datei gesichert. Damit hängt die Beständigkeit des Gesamtsystems von der Beständigkeit dieser Datei ab; eine Erhöhung derselben ließe sich auf Betriebssystemebene erreichen (z.B. durch Sichern oder Spiegeln der Stable Heap-Datei).

Der folgende Abschnitt faßt noch einmal kurz die Funktionalitäten der POSTORE Schicht aus der Sicht der nächsthöheren 2. Schicht zusammen.

Allozieren eines POSTORE-Vektors Bei der Allozierung wird die Gesamtgröße *m* des Vektors angegeben; zurückgegeben wird die zur Referenzierung des Vektors nötige *long objid*.

Zugriff auf den Zustand von POSTORE-Vektoren Es werden Funktionen zum ungeprüften Zugriff auf einzelne oder mehrere Worte innerhalb eines POSTORE-Vektors angeboten.

Wurzelobjekt Die Struktur und der Zustand des POSTORE-Wurzelobjektes werden innerhalb der 1. Schicht definiert und verwaltet; es kann nicht explizit gelesen oder überschrieben werden außer der frei zugreifbaren Komponente, die die *long objid* des Wurzelobjektes der nächsthöheren Schicht enthält (Abbildung 7.10, S. 99).

Speicherrückgewinnung Bei der Speicherrückgewinnung wird der Speicher der nicht transitiv vom Wurzelobjekt referenzierten persistenten Objekte wieder freigegeben. Die Speicherrückgewinnung muß explizit veranlaßt werden.

7.5 Schicht 2: Die C-PLOB! Schicht

Die Implementation der 2. Schicht erfolgte in ANSI-C unter zusätzlicher Benutzung eines ebenfalls von mir erstellten minimalen Objektsystems. Diese Schicht realisiert transaktionsgesicherten, durch das Sperrprotokoll unteilbaren Zugriff auf dynamisch typisierte, selbstbeschreibende Objekte und stellt die für eine Indexverwaltung notwendigen Hilfsmittel zur Verfügung.

7.5.1 Überlegungen zur Performanz und Datenabstraktion

Eine hohe Performanz kann erreicht werden, wenn wie in WOOD [St. Clair 93] die transienten LISP Daten auf ihrer niedrigsten Abstraktionsebene verarbeitet werden, d.h. wenn dieÜbertragung eines Objektes zwischen dem transienten und dem persistenten Speicher innerhalb der 2. Schicht erfolgen würde und dazu die transiente Repräsentation des Objektes ausgenutzt werden würde. Dies wäre mit folgenden Schwierigkeiten verbunden:

- Die dazu notwendigen Informationen über die Struktur der transienten Objekte auf dieser niedrigen Ebene sind nur schwierig zu erhalten, da sie von den AutorInnen des verwendeten LISP-Systems festgelegt wurden und im allgemeinen von diesen nicht öffentlich zugänglich gemacht werden (siehe z.B. [Snape 93]); weiterhin wäre das persistente System bei einer derartigen Festlegung auf ein transientes System nicht portabel, d.h. es ist nicht ohne weiteres möglich, die höheren Schichten des Systems gegen ein anderes Objektsystem auszutauschen. In WOOD stellten sich diese Probleme nicht, da zum einen sein Autor an der Entwicklung des zur Realisierung von WOOD verwendeten Macintosh Common LISP (MCL) direkt beteiligt ist und zum anderen das Format des persistenten Speichers nach seinen Worten mit der folgenden Absicht erstellt wurde [St. Clair 93]:

  „It is not intended to be compatible with anything.“

- Ein transientes Datenobjekt enthält außer dem Objektzustand noch unbekannte systeminterne Informationen zu seiner Verwaltung. Für die Repräsentation eines persistenten Objektes im transienten Speicher muß gegebenenfalls eine transiente Repräsentation alloziert und mit den systeminternen Informationen initialisiert werden; ferner kann die Allozierung auch zu berücksichtigende Seiteneffekte nach sich ziehen.

- Common LISP Daten sind oft sehr redundant (beispielsweise enthält ein Klassen-Metaobjekt eine Liste mit den Klassen-Metaobjekten der Subklassen). Auf der LISP Ebene der 3. Schicht ist die Bedeutung eines Datums oft bekannt; damit können die Redundanzen eines transienten Objektes bei seiner Übertragung in den persistenten Speicher weggelassen und bei der Rückübertragung wieder hinzugefügt werden. Dieses Wissen um die Bedeutung fehlt in der 2. Schicht.

- Bei der Übertragung eines Objektzustands auf einer niedrigen Abstraktionsebene kann die Bedeutung des Objektes für die höheren Ebenen verloren gehen. Für die Rekonstruktion eines solchen Objektes reicht eine bloße Erzeugung der transienten Repräsentation und die Übertragung des Zustands nicht aus, sondern es muß zusätzlich die Bedeutung des Objektes mit berücksichtigt werden. In LISPWORKS werden beispielsweise Pakete (*packages*) durch Instanzen einer Strukturklasse repräsentiert; eine Rekonstruktion muß das wiederhergestellte Paket dem LISP-System auch als Paket bekannt machen.

Ähnliche Feststellungen wurden auch bei der Realisierung des persistenten Objektsystems ZEITGEIST [Ford et al. 88, S. 35] gemacht. Aus den genannten Gründen wird dieÜbertragung von Objektzuständen von der 3. Schicht aus durchgeführt.

7.5.2 Typisierung der *long objids* für *Immediates*

Das System POSTORE garantiert, daß die vergebenen *long objids* immer ganzzahlige Vielfache von 8 sind, d.h. die untersten 3 Bit sind immer 0. Zusätzlich werden die im Referenzfeld eines

POSTORE-Vektors abgelegten *long objids* von POSTORE nur als Referenz angesehen, wenn sie ganzzahlige Vielfache von 8 sind; andere Werte werden niemals dereferenziert. Dies wurde benutzt, um *long objids* zu typisieren: die untersten 3 Bit der *long objids* werden als Typkennung (*type tag*) benutzt. Die Belegung ist in Tabelle 7.3 wiedergegeben. Eine *long objid* ist damit

Bit-Position				Kennung	Zustand in	Beschreibung
31...3	2	1	0			
...	0	0	0	0	Bit 31...3	*long objid* als Referenz auf Objekt
...	.	0	1	1, 5	Bit 31...2	Instanz der Klasse **fixnum**
...	0	1	0	2	Bit 31...3	Instanz der Klasse **short-float**
...	0	1	1	3	Bit 31...3	Instanz der Klasse **character**
...	1	0	0	4	Bit 31...3	[Instanz der Klasse] Markierung
				6, 7		Zur Zeit unbenutzte Typkennungen

Tabelle 7.3: Typkennungen der typisierten *long objids*

entweder eine Referenz auf einen POSTORE-Vektor (Kennung 0 in Tabelle 7.3) oder enthält den Zustand einer Instanz der in den Zeilen 2–5 in Tabelle 7.3 aufgeführten Basisklassen. Für *Immediates* enthält eine *long objid* damit den Zustand des repräsentierten Objektes; für alle anderen Objekte ist eine *long objid* ein ‚echtes' Surrogat ohne Informationen über das repräsentierte Objekt. Die hier realisierten typisierten *long objids* sind Surrogate mit nicht vollständig kanonischer Form (Abschnitt 3.2.5, S. 24).

Markierungen

In Common LISP werden üblicherweise ausgezeichnete Objekte als Markierungen für bestimmte singuläre Eigenschaften oder Bedingungen verwendet, wie z.B. in der CLOSETTE Implementation das in der Variablen `secret-unbound-value` enthaltene Objekt als Repräsentation der Eigenschaft ‚ungebundener *Slot*' verwendet wird [AMOP, S. 28]. In PLOB! gibt es für Markierungen eine eigene ‚Klasse' (Kennung 4 in Tabelle 7.3); sie sind damit disjunkt zu allen anderen Daten. Markierungen werden durch Instanzen der‚Klasse' Markierung repräsentiert.

Repräsentation von *objids*

Aus Effizienzgründen sollen die in den nachfolgenden Schichten verwendeten *objids* für ihre Repräsentation möglichst wenig Speicherplatz benötigen; deswegen werden in den in Common LISP realisierten höheren Schichten zu ihrer Repräsentation Instanzen des Typs **fixnum** verwendet. Diese Repräsentation beötigt damit wesentlich weniger Speicherplatz als beispielsweise die in SOH realisierte Repräsentation von *objids* als typstrukturierte Bezeichner (Abschnitt 6.4.3, S. 72).

Da drei der 32 Bit einer *long objid* für die Typisierung verwendet werden, kann eine *long objid* $2^{(32-3)}$ = 512 *M* Objekte adressieren. Instanzen des Typs **fixnum** werden im verwendeten LISPWORKS Common LISP durch 30 Bit repräsentiert. Eine *long objid* in einer der höheren Schichten hätte dann $2^{(30-3)}$ = 128 *M* Objekte adressieren können. Um die Anzahl der adressierbaren Objekte möglichst hoch zu halten, werden daher die *long objids* beim Austausch mit der 3. Schicht durch Weglassen der Typkennung in ein kürzeres 29 Bit Format, die *short objids*, konvertiert. Die Typkennung wird, falls nötig, von der 3. in die 2. Schicht mit einem zusätzlichen Funktionsparameter übergeben und von der 2. in die 3. Schicht als zusätzlicher Rückgabewert eines Funktionsaufrufs zurückgegeben.

Allozierung von persistenten Objekten

Jeder Basistyp wird im Code durch einen Datensatz beschrieben, der die Größe und Struktur seiner Instanzen festlegt; alle anderen Klassen werden explizit durch Klassen-Beschreibungsobjekte beschrieben. Für die Allozierung eines persistenten Objektes wird nicht wie in der 1. Schicht seine Speichergröße, sondern seine Typkennung angegeben.

7.5.3 Repräsentation von persistenten Objekten

In Gegensatz zu relationalen Datenbanken definiert der Objektspeicher POSTORE keine feste Menge von Basistypen und Möglichkeiten zur Typbildung aus diesen Basistypen, sondern bietet ein generisches Objektformat zur Festlegung benutzerInnen-definierter Datentypen an. Die von mir gewählte Repräsentationsform für persistente Objekte orientiert sich daher an der in LISPWORKS vorgefundenen Repräsentationsform für transiente Objekte, erweitert um datenbankspezifische Komponenten.

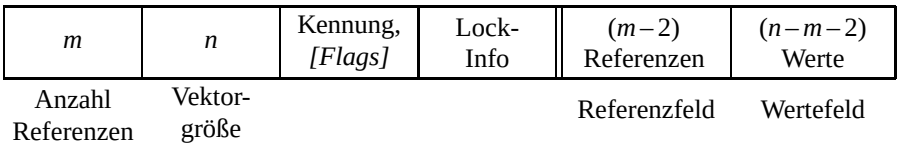


Abbildung 7.4: Repräsentation eines persistenten Objektes

In der 2. Schicht werden direkt hinter die von POSTORE benutzten Kopfinformationen weitere 2 Worte mit Kopfinformationen angefügt (Abbildung 7.4). Die einzelnen Objektkomponenten werden folgendermaßen genutzt:

Kennung Das Feld Kennung enthält eine zu den bereits in Tabelle 7.3 (S. 89) vergebenen Typkennungen disjunkte numerische Typkennung in Form einer auf die Klasse **fixnum** typisierten *long objid*, die die Basisklasse des persistenten Objektes angibt. Für jeden Common LISP Basistyp wurde ein entsprechender Basistyp mit ähnlicher Struktur in PROB! definiert. Wie auch in transienten LISP-Systemen werden die Basistypen als‚eingebaut' angesehen; sie unterliegen damit den gleichen Einschränkungen.

Flags Die angedeuteten Statusvariablen (*flags*) sind zwar realisiert, werden aber im Moment nicht genutzt.

Lock-Info Das Feld Lock-Info enthält eine Referenz auf die Objekte, die zur Repräsentation des aktuellen Sperrzustands benutzt werden.

Referenzfeld Das Referenzfeld umfaßt $(m-2)$ typisierte *long objids*, die, wie im vorigen Abschnitt erläutert, entweder Verweise auf persistente Objekte oder den Zustand von Instanzen der in Tabelle 7.3 aufgeführten Basistypen enthalten.

Wertefeld Das Wertefeld mit $(n-m-2)$ Worten wird für persistente Werte benutzt, wie z.B. Zeichenketten oder Bit-Vektoren.

Repräsentation eines persistenten Feldes

Abbildung 7.5 zeigt beispielhaft die Repräsentation eines persistenten Feldes (*array*) mit den

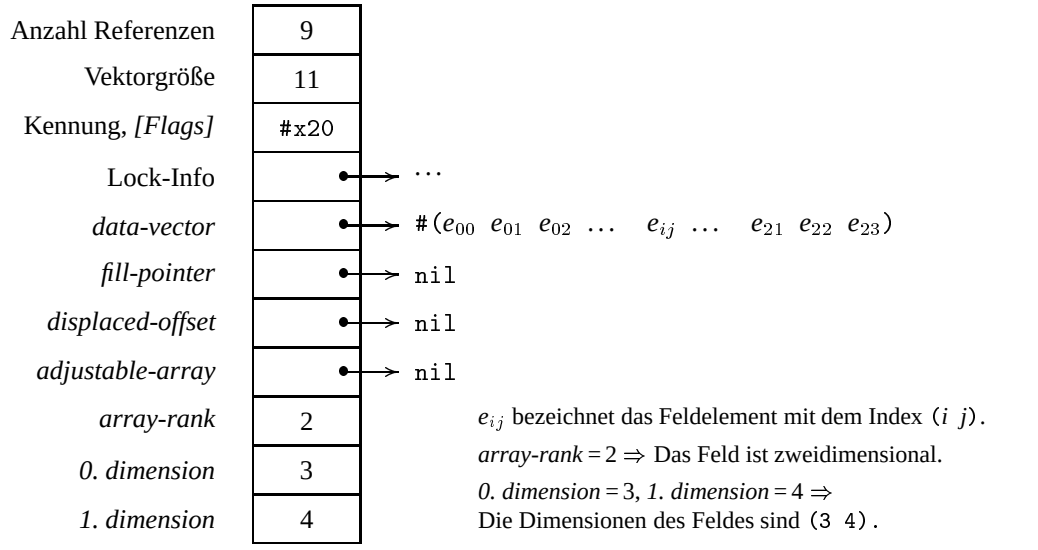


Abbildung 7.5: Beispiel für die Repräsentation eines persistenten Feldes

Dimensionen `(3 4)` (im Vergleich zur Abbildung 7.4 ist die Darstellung der Objektkomponenten um 270° gedreht).

Das Referenzfeld teilt sich auf in einem Bereich mit konstanter Größe (Komponenten *data-vector* bis *array-rank*) und einen Bereich mit variabler Größe (Komponenten *0. dimension* bis *1. dimension*). Die Größe des variablen Bereichs (in diesem Beispiel 2) ist die Anzahl der Dimensionen des persistenten Feldes und wird bei der Allozierung angegeben. Das Wertefeld ist für persistente Felder immer leer.

Die Bedeutungen der Komponenten *fill-pointer* [CLtLII, S. 455], *adjustable-array* [CLtLII, S. 452] und *array-rank* [CLtLII, S. 448] entsprechen dem in [CLtLII] definierten Standard für LISP Felder. Die Komponenten *0. dimension* und *1. dimension* enthalten die Dimensionen des Feldes (*array dimensions*) [CLtLII, S. 449].

Die Komponente *data-vector* enthält einen persistenten Vektor mit den Feldelementen. Die Anzahl der Elemente des Vektors ist gleich der Anzahl der Feldelemente. Der Typ des Vektors *data-vector* ergibt sich aus dem Elementtyp, auf den das Feld spezialisiert ist:

Elementtyp `t` Der persistente Vektor *data-vector* ist eine Instanz des‚normalen' persistenten Vektortyps, d.h. des Vektortyps, dessen Instanzen Elemente des Typs `t` enthalten.

Elementtyp `(unsigned-byte 1)` Der Vektor *data-vector* ist ein persistenter Bitvektor.

Elementtyp `float`, `(unsigned-byte` n`)`, $n > \mathbf{1}$ In diesem Fall ist der Vektor *data-vector* ein persistenter *ivector* (*i*mmediate *vector*), der nur Werte genau eines Basistyps enthalten kann. Der *ivector* wird auf den jeweiligen Elementtyp spezialisiert.

Der Elementtyp, auf den das persistente Feld spezialisiert ist, läßt sich damit aus dem Typ der Komponente *data-vector* bestimmen.

Die Komponente *displaced-offset* enthält für ein Feld, das nicht Elemente eines anderen Feldes referenziert (*non-displaced array*) [CLtLII, S. 444], das persistente Symbol `nil`. Sofern das Feld Elemente eines anderen Feldes referenziert (*displaced array*), enthält die Komponente *displaced-offset* den Index, ab dem das andere Feld referenziert wird. In diesem Fall verweist

die Komponente *data-vector* für ein referenzierendes Feld variabler Größe (*adjustable array*) nicht auf einen persistenten Vektor, sondern auf das referenzierte persistente Feld; bei einem referenzierendem Feld mit konstanter Größe (*non-adjustable array*) enthält die Komponente *data-vector* des referenzierenden Feldes direkt die Komponente *data-vector* des referenzierten Feldes.

Persistente Felder entstehen im allgemeinen aus der Speicherung eines transienten Feldes; aus dem transienten Feld werden mit den Feld-Informationsfunktionen (*array information functions*) die entsprechenden Parameter gelesen und in das persistente Feld eingetragen. Der Elementtyp des persistenten Feldes wird beim Speichern eines transienten Feldes aus dem bei der Erzeugung des transienten Feldes mit dem Schlüsselwort-Parameter `:element-type` [CLtLII, S. 443] übergebenen Elementtyp bestimmt, für den das transiente Feld spezialisiert wurde. Für transiente Felder, deren Elementtyp bei der Erzeugung nicht spezialisiert wurde, wird der Elementtyp `t` verwendet.

Repräsentation von Struktur- und CLOS-Objekten

Für persistente Struktur- und CLOS-Instanzen gibt es jeweils noch eine gesonderte Typkennung; die Kopfinformation wird um ein weiteres Wort erweitert, das eine Referenz auf eine persistente Struktur- bzw. Klassen-Beschreibung enthält (Abbildungen 7.6 und 7.7). Das Wertefeld ist für persistente Struktur- und CLOS-Instanzen immer leer.

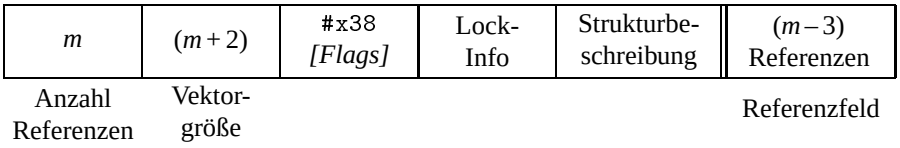


Abbildung 7.6: Repräsentation eines persistenten Strukturobjektes

Persistente Strukturobjekte Alle Strukturobjekte haben die einheitliche Typkennung `#x38`; die Klasse des Strukturobjektes geht aus der referenzierten Strukturbeschreibung hervor. Die Strukturbeschreibung selbst ist ebenfalls ein Strukturobjekt. Die Größe des Referenzfeldes entspricht der Anzahl der Struktur-*Slots* (Abbildung 7.6).

Persistente CLOS-Instanzen Da die Klasse einer CLOS-Instanz bei Erhalt ihrer Identität geändert werden kann, werden CLOS-Instanzen anders als Strukturobjekte repräsentiert. Die *Slot*-Zustände werden nicht direkt in einer persistenten CLOS-Instanz repräsentiert, sondern durch einen referenzierten persistenten Zustandsvektor; die Anzahl der Elemente des Zustandsvektors ergibt sich aus der Anzahl der *Slots*. Der Zustandsvektor ist eine Instanz des Basistyps persistenter Vektor mit der in Abbildung 7.4 (S. 90) gezeigten Repräsentation. Persistente CLOS-Instanzen haben immer die Typkennung `#x48`; die Klasse der CLOS-Instanz geht aus der referenzierten Klassenbeschreibung hervor. Die Klassenbeschreibung selbst ist auch eine persistente CLOS-Instanz (Abbildung 7.7).

7.5.4 Bedeutung von Objekten und Portabilität

In diesem Abschnitt wird die Bedeutung von Objekten in Bezug auf Portabilität untersucht. Portabilität meint den Austausch der 3. Schicht gegen ein anderes, nicht auf CLOS basierendes

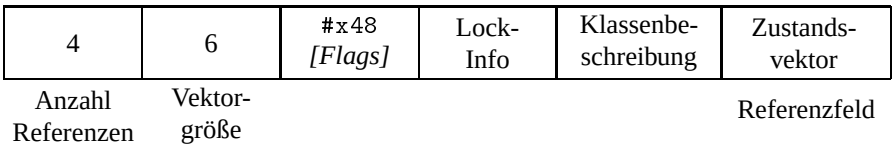


Abbildung 7.7: Repräsentation einer persistenten CLOS-Instanz

Objektsystem.

Bedeutung der Objekte von Basisklassen

Basisklassen sind grundlegende Klassen eines Programmsystems mit einer von den AutorInnen in das System fest ‚eingebauten' (*built in*) Struktur und einem fest eingebautem Verhalten. Da die Bedeutung ihrer Instanzen allgemein bekannt ist, bereitet eine Nutzung keine Schwierigkeiten. Ein Beispiel ist die Bedeutung von *cons*-Zellen, für die festgelegt wurde, das sie ausschließlich Paare repräsentieren. Aus dieser festgelegten Bedeutung erschließt sich auch das Verhalten einer *cons*-Zelle; so können die beiden Elemente manipuliert werden.

Da unter anderem die Struktur der Instanzen von Basisklassen festgelegt ist und damit ihre Repräsentationsform konstant bleibt, kann intern der Zugriff auf den Zustand einer Basisklasse auf einer niedrigen Abstraktionsebene realisiert werden, bei der die Bedeutung des *Slots*, auf den zugegriffen wird, klar definiert ist.

Bedeutung von CLOS-Instanzen und Metaobjekten

Dieser Abschnitt betrachtet (transiente) CLOS-Instanzen mit Struktur und Verhalten. Für eine CLOS-Instanz ergibt sich ihre Bedeutung aus ihrer Klasse, die alle Informationen über Struktur und Verhalten enthält. Anders als bei Basisklassen ist eine CLOS Klasse nicht als‚eingebaut' festgelegt, sondern kann dynamisch erzeugt und verändert werden. Nur für die AutorInnen einer Klasse erschließt sich die vollständige Bedeutung der Objekte selbst, d.h. welche realen Dinge oder Sachverhalten zu Objekten abstrahiert wurden.

Klassen werden wiederum durch Objekte, genauer Klassen-Metaobjekte, repräsentiert. Für CLOS selbst muß die Bedeutung der Klassen-Metaobjekte und der von ihnen referenzierten weiteren Metaobjekte bekannt sein, um sie nutzen zu können. Zu diesem Zweck wird die Bedeutung der Metaobjekte in [AMOP] ausschließlich über ihr Verhalten festgelegt; eine Festlegung für die Strukturen der Metaobjekte ist im Rahmen der Normierung unnötig und läßt den AutorInnen eines CLOS-Systems Freiheit in der Auswahl der Strukturen.

Da die Struktur von CLOS-Objekten nicht von vornherein festgelegt ist und sich ihre Repräsentationsform aus den im Klassen-Metaobjekt abgelegten Informationen ergibt, erfolgt der Zugriff auf den Zustand eines CLOS-Objektes intern auf einer relativ hohen Abstraktionsebene über das Verhalten des Klassen-Metaobjektes.

Bedeutung von persistenten Metaobjekten

Da die Bedeutung von Metaobjekten wie im letzten Abschnitt erläutert ausschließlich über ihr Verhalten definiert wird und das persistente System nur Struktur, aber kein Verhalten speichern kann, ist es in der 2. Schicht nicht möglich, die Bedeutung von persistenten Metaobjekten zu kennen.

Folgerungen für die Bedeutung von Objekten und Portabilität

Wir können auch anders!

— Detlev Buck

Bei einem Austausch der 3. Schicht gegen ein nicht auf CLOS basierendes Objektsystem wäre es vorteilhaft, über die im persistenten Speicher befindlichen Objekte Aussagen auf Klassenebene treffen zu können, z.B. wie der Name der Klasse ist oder welche *Slots* eine persistente CLOS-Instanz hat. Für persistente Instanzen von Basistypen stellt dies kein Problem dar, da ihre Bedeutung ähnlich wie in einem Common LISP-System in das persistente System‚eingebaut' wurde. Bei persistenten CLOS-Instanzen muß für Aussagen auf Klassenebene die Bedeutung der Klassenbeschreibung (Abbildung 7.7, S. 93) bekannt sein; wie im letzten Abschnitt gezeigt, können persistente Metaobjekte diese Bedingung nicht erfüllen.

Verzicht auf Portabilität Sofern Portabilität nicht benötigt wird, besteht eine Lösung darin, die Klassenbeschreibungen ausschließlich den höheren Schichten zu überlassen; deren Bedeutung ist dann auch nur dort bekannt. Vorteilhaft ist, daß völlige Freiheit in der Wahl der Klassenbeschreibung besteht und damit keine bestimmte Datenmodellierung für persistente Objekte von der 2. Schicht vorgeschrieben wird. Der Nachteil ist, daß Portabilität nicht gewährleistet werden kann, da die Bedeutung einer Klassenbeschreibung auf eine konkrete höhere Schicht spezialisiert ist.

Integration von CLOS Eine radikale Lösung bestände in der Integration von CLOS in die 2. Schicht, damit die Bedeutung der Klassenbeschreibung wieder über ihr Verhalten definiert werden könnte; dies würde allerdings den in Abschnitt 4.1.2 (S. 36) erläuterten Architekturansatz realisieren, der aus den dort genannten Gründen im Rahmen dieser Arbeit ausscheidet.

Festlegung der Klassenbeschreibungen in der 2. Schicht Ein moderater Ansatz besteht darin, Klassenbeschreibungen nicht durch Klassen-Metaobjekte und die von ihnen referenzierten Metaobjekte zu repräsentieren, deren Bedeutung über ihr Verhalten definiert wird, sondern stattdessen Klassen-Beschreibungsobjekte und von ihnen referenzierte Beschreibungsobjekte zu benutzen, deren Bedeutung ausschließlich über ihre Struktur festgelegt wird. Die Struktur der Beschreibungsobjekte wird durch das in den höheren Schichten benutzte Objektsystem (in diesem Fall CLOS) bestimmt und anschließend in der 2. Schicht festgelegt; für die höheren Schichten wird die Konvention eingeführt, beim Speichern einer persistenten CLOS-Instanz sicherzustellen, daß seine Klassenbeschreibung gültig ist. Die Klassen der Beschreibungsobjekte können sinnvollerweise aus den Metaobjekt-Klassen von CLOS abgeleitet werden (siehe Abschnitt 7.6.2, S. 103); dabei können durch Auslassen der für ein Beschreibungsobjekt nicht benötigten *Slots* der Metaobjekte die in Abschnitt 7.5.1 (S. 88) erwähnten Redundanzen vermieden werden, beispielsweise indem in Klassen-Beschreibungsobjekten die Liste der Subklassen oder der direkten Methoden nicht repräsentiert werden.

Nachteilig ist, daß durch die Festlegung der Beschreibungsobjekt-Klassen über ihre Struktur Abstraktion verloren geht; die Struktur jeder Beschreibungsobjekt-Klasse muß in der 2. Schicht verbindlich festgelegt werden. Damit werden Beschreibungsobjekt-Klassen zu Basisklassen des persistenten Systems. Ebenso müssen bei Einhaltung der oben genannten Konvention die Beschreibungsobjekte von der 3. Schicht aus aufgebaut werden, da die vorhandenen Metaobjekte selbst nicht verwendet werden dürfen. Implizit wird durch diese Vorgehensweise für persistente

Objekte ihre Datenmodellierung auf die von CLOS festgelegt; für höhere Schichten folgt daraus, für persistente Objekte ebenfalls die Datenmodellierung von CLOS übernehmen zu müssen. Dies kann insbesondere für höhere Schichten mit Objektsystemen ohne ein in die Sprache integriertes Metaobjekt-Protokoll (wie z.B. C++) schwierig werden, da in diesem Fall ein Metaobjekt-Protokoll (wenn auch nicht so umfangreich wie das MOP von CLOS) zur höheren Schicht hinzugefügt werden muß.[4]

Realisierte Lösung Nach Abwägung der Vor- und Nachteile entschied ich mich für den letztgenannten Ansatz, Beschreibungsobjekt-Klassen zu realisieren und in der 3. Schicht die oben genannte Konvention zu befolgen, um Portabilität sicherzustellen; eine genaue Erläuterung der Beschreibungsobjekt-Klassen folgt in Abschnitt 7.6.2 (S. 103). Ein weiterer Vorteil dieses Vorgehens ist, daß bereits die 2. Schicht mit Kenntnis der Bedeutung umfangreiche Prüfungen zur Fehlersuche vornimmt; bei auftretenden Laufzeitfehlern in der 2. Schicht werden die Fehlermeldungen aussagekräftiger gestaltet. Die hier getroffenen Aussagen und Schlußfolgerungen für CLOS-Instanzen gelten sinngemäß auch für Strukturobjekte.

7.5.5 Sitzungen

Eine Sitzung (*session*) besteht aus einer Folge von zeitlich disjunkten Transaktionen sowie einem Prozeß, der innerhalb einer Transaktion auf persistente Objekte zugreifen kann. Ein an eine Sitzung gebundener Prozeß ist kein UNIX Prozeß, sondern ein LISPWORKS *lightweight*-Prozeß.

Vor Beginn einer Transaktion wird der Gesamtzustand aller persistenten Objekte als konsistent angenommen; innerhalb einer Transaktion dürfen Objekte modifiziert werden, bis die Transaktion entweder beendet oder abgebrochen wird. Bei einem Ende gilt der aus denÄnderungen hervorgegangene Zustand als neuer konsistenter Gesamtzustand; bei einem Abbruch (*abort*) wird angenommen, daß der Gesamtzustand durch die erfolgtenÄnderungen inkonsistent ist und der letzte konsistente Gesamtzustand (bei Beginn der Transaktion) wiederhergestellt werden soll (*rollback*) (Abbildung 7.8). Eine Transaktion ist damit die Realisierung eines Konzepts

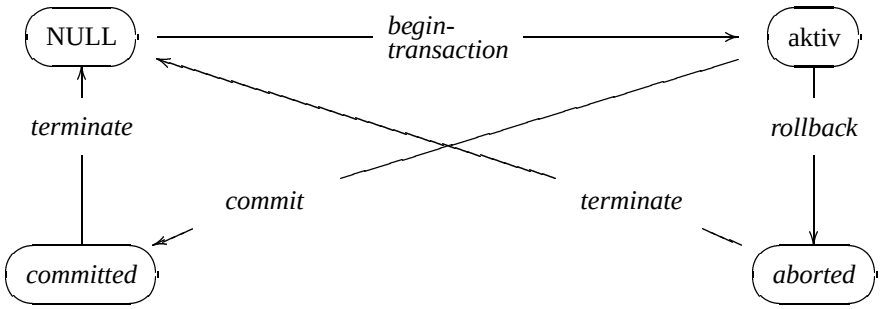


Abbildung 7.8: S/T-Diagramm einer Transaktion aus Applikationssicht (nach [Gray et al. 93, S. 181])

für den Übergang des Gesamtsystems von einem konsistentem Zustand in den nächsten konsistenten Zustand. Die Aufgabe des persistenten Systems dabei ist, diesenÜbergang als unteilbar erscheinen zu lassen.

[4]Die in Abschnitt 5.2.2 (S. 51) erwähnte angenehme Eigenschaft der Selbstbeschreibung eines Systems bedingt zusätzlich ein Metaobjekt-Protokoll [Wegner 90, S. 12].

Die Zustandsänderungen werden von dem an die Sitzung gebundenen Prozeß durchgeführt. Bei mehreren beteiligten Sitzungen und damit Prozessen muß jede Sitzung für schreibenden Zugriff die persistenten Objekte alleine im Zugriff haben; bei lesendem Zugriff sollten schreibende Zugriffe durch andere Sitzungen gesperrt werden. Das persistente System muß also dafür sorgen, daß Zugriffe auf die Objekte ausschließlich durch Sperren derselben ermöglicht werden.

Sitzungen werden durch Instanzen des Basistyps **persistent-heap** repräsentiert; ihre Verwendung ist schicht-abhängig:

Oberhalb der 2. Schicht In den oberen Schichten wird eine Sitzung für den transaktionsgesteuerten Zugriff auf alle im persistenten Speicher enthaltenen persistenten Objekte benutzt. Ihre Aufgabe besteht darin, die für das Sperrprotokoll notwendigen Transaktionen zur Verfügung zu stellen.

Sitzungen werden damit ähnlich benutzt wie die Instanzen der in PCLOS definierten Klasse **protector** [Paepke 91a, S. 9].

Innerhalb der 2. Schicht Hier werden Sitzungen als Repräsentation des (zur Zeit noch) implizit an die Sitzung gebundenen Prozesses angesehen (zur expliziten Repräsentation von Prozessen siehe Abschnitt 8.3.2, S. 150). Beispielsweise werden sie in diesem Sinne beim Sperrprotokoll als sperrendes Objekt benutzt; Sperrkonflikte werden der Sitzung signalisiert, um den daran gebundenen Prozeß zum passiven Warten zu veranlassen.

Das hier realisierte Sitzungskonzept lehnt sich an das in [Kim et al. 89, S. 269] [Itasca 91, S. 20] [Ahmed et al. 91, S. 29] für die Datenbank ORION (jetzt ITASCA) vorgestellte Konzept an. Transaktionen wurden in der 2. Schicht als Zwei-Phasen-Transaktionen (*two-phased transactions*) realisiert. Für nicht-wohlgeformte Transaktionen (*non-well-formed transactions*) [Gray et al. 93, S. 385] wird ein Fehler signalisiert. Wie bei der Datenbank ORION wurde kein vollständiges DO-UNDO-REDO-Protokoll [Gray et al. 93, S. 538] realisiert; es wird lediglich eine UNDO-*Log*-Datei pro Sitzung geführt, in der die Zustände der modifizierten Objekte vor der Modifikation abgelegt werden [Kim et al. 89, S. 272] [Itasca 91, S. 21]. Bei Ende einer Transaktion wird der Gesamtzustand des persistenten Speichers mittels eines Aufrufs der Stabilisierungsfunktion aus der 1. Schicht gesichert; damit ist eine Mitführung der neuen Objektzustände in einer REDO-*Log*-Datei unnötig. Bei einem Abbruch werden die Objektzustände aus der UNDO-*Log*-Datei restauriert.

Durch den Fortfall der REDO-*Log*-Datei halbiert sich ferner die Datenmenge, die sonst pro Schreibzugriff auf ein Objekt in den *Log*-Dateien vermerkt werden müßte.

In ORION kann eine Sitzung von mehreren Prozessen (dort *windows* genannt) genutzt werden [Kim et al. 89, S. 269] [Itasca 91, S. 20]. Dies ist in PLOB! nicht möglich; eine Sitzung entspricht genau einem Prozeß. Die in [Kim et al. 89, S. 270] [Ahmed et al. 91, S. 29] erläuterten hypothetischen Transaktionen, d.h. Transaktionen, die bei Ende immer abgebrochen werden, wurden von mir nicht realisiert.[5]

7.5.6 Objektsperren

Die in der 1. Schicht angebotenen Schreibsperren werden in der 2. Schicht lediglich benutzt, um den konkurrenten Zugriff auf das Feld mit der Sperrinformation zu synchronisieren. In diesem Feld befindet sich, sofern das persistente Objekt gesperrt ist, eine Referenz auf ein persistentes Objekt mit dem aktuellen Sperrzustand.

[5] Offensichtlich wurden hypothetische Transaktionen im ORION-Nachfolger ITASCA ebenfalls nicht mehr realisisert, da sie in [Itasca 91] nicht mehr erwähnt werden.

Hierarchisches Sperrprotokoll

Realisiert wurde ein hierarchisches pessimistisches Sperrprotokoll (*tree locking*) nach [Gray et al. 93, S. 406] für die hierarchisch geordneten Ebenen *Speicher* (gesamter persistenter Speicher mit allen persistenten Objekten), *Vektor* (ein persistentes Objekt) und *Element* (ein Element, d.h. ein Wort innerhalb eines persistenten Objektes) mit den extern verfügbaren, nicht-exklusiven Modi (*shared modes*) Nur-Lesen (*read-only*), Lesen (*read*) und dem exklusiven Modus (*exclusive mode*) Schreiben (*write*) sowie den intern benutzten Modi Nur-Lese-Absicht (*read-only-intent*), Leseabsicht (*read-intent*) und Schreibabsicht (*write-intent*). Beim hierarchischen Sperrprotokoll impliziert das Sperren eines Objektes auf einer höheren Ebene mit einem Modus den gleichen Sperrmodus für alle unteren Ebenen. Die intern benutzten Absichtssperren (*intent locks*) repräsentieren die Absicht eines sperrenden Objektes, auf einer tieferen Ebene eine entsprechende Sperre vornehmen zu wollen.

An einem Sperrvorgang sind immer zwei persistente Objekte beteiligt: eine sperrende und eine zu sperrende Instanz. Für die Modi Leseabsicht, Lesen, Schreiben und Schreibabsicht repräsentiert das sperrende Objekt als Instanz der Klasse **persistent-heap** die benutzende Sitzung. An das sperrende Objekt ist eine Transaktion gebunden, in der die Sperre vermerkt wird. Bei Transaktionsende sorgt eine auf die Klasse **persistent-heap** spezialisierte Methode dafür, daß alle innerhalb der Transaktion vorgenommenen Sperren aufgehoben werden. Bei den Modi Nur-Lesen und Nur-Lese-Absicht sind das sperrende und das zu sperrende Objekt identisch; sie können daher nur explizit aufgehoben werden. Tabelle 7.9 zeigt die Kompatibilitätsmatrix

Gesetzt ist→ / Angefordert wird↓	**RO-Intent**	**Read-Only**	**Read-Intent**	**Read**	**Write-Intent**	**Write**
Read-Only-Intent	✔	✔	✔	✔	✔	✔
Read-Only	✔	✔	✔	✔	✔	✔
Read-Intent	✔	✔	✔	✔	✔	✘
Read	✔	✔	✔	✔	✘	✘
Write-Intent	✔	✔	✔	✘	✔	✘
Write	✔	✘	✘	✘	✘	✘

✔ ≡ Kompatible Sperrmodi ✘ ≡ Inkompatible Sperrmodi

Tabelle 7.9: Kompatibilitätsmatrix für Sperrmodi

der realisierten Sperrmodi. Eine Sperre wird zugelassen, wenn der angeforderte Sperrmodus kompatibel zu allen bisher auf das Objekt gesetzten Sperrmodi ist. Zusätzlich zu den in Tabelle 7.9 gezeigten Fällen ist eine Sperre auch dann kompatibel, wenn das sperrende Objekt bereits eine Sperre mit einem beliebigem anderen Modus erhielt und kein inkompatibler Sperrmodus zur Sperre eines anderen sperrenden Objektes besteht; alle Sperren eines einzigen sperrenden Objektes sind unabhängig vom Modus kompatibel zueinander.

Sperrkonflikte

Kann eine Sperre nicht erteilt werden, muß diejenige Instanz (d.h. Sitzung), die die Sperre zuletzt anforderte, passiv warten, bis die bestehenden Sperren, deren Modi inkompatibel zum Modus der angeforderten Sperre sind, von den anderen sperrenden Objekten (d.h. Sitzungen) aufgehoben werden. Für eine möglichst schnelle Aktualisierung der Daten werden wartende Instanzen mit Anforderungen für Schreibsperren vor wartenden Instanzen mit Anforderungen für Lesesperren aktiviert [Jessen et al. 87, S. 189].

Verklemmungen

Da Verklemmungen (*deadlocks*) relativ selten auftreten, wird auf eine explizite Detektion verzichtet; statt dessen wird die in [Gray et al. 93, S. 424] vorgeschlagene Technik benutzt, nach Ablauf einer bestimmten Wartezeit (*time out*) für ein sperrendes Objekt anzunehmen, daß seine Sperranforderung eine Verklemmung verursachte. Die Anforderung wird in diesem Fall mit einer entsprechenden Fehlermeldung abgebrochen.

Zugriff auf den Zustand von persistenten Objekten

Sowohl lesender als auch schreibender Zugriff aus den höheren Schichten auf ein persistentes Objekt bedingen eine Sperre. Bei jedem Aufruf einer Zugriffsfunktion für ein persistentes Objekt wird von der höheren Schicht die Sitzung mit übergeben; sofern eine Sperre noch nicht erfolgte, wird sie vor dem eigentlichen Zugriff von der 2. Schicht für die angegebene Sitzung angefordert.

7.5.7 Persistente B-Bäume

Persistente B-Bäume [Gray et al. 93, S. 851–858] werden benutzt, um Einträge bestehend aus einem persistentem Datenobjekt assoziiert zu einem persistentem Schlüsselobjekt zu speichern. Sie werden ähnlich wie *Hash*-Tabellen [CLtLII, S. 435–441] verwendet. Von Vorteil ist zum einen, daß die Verwaltung sehr großer Datenmengen effizienter als bei *Hash*-Tabellen ist und zum anderen die Schlüssel sortiert abgelegt werden; damit können die zu einem Schlüsselbereich assoziierten Daten effizient gesucht werden.

Repräsentiert werden sie durch Instanzen des Basistyps **persistent-btree**. Die realisierten Ordnungskriterien für Schlüsselobjekte sind:

Ordnungskriterium `eq` **oder** `eql` Als Ordnungskriterium werden die Identitäten, d.h. die numerischen Werte der *long objids*, der Schlüsselobjekte benutzt.

Ordnungskriterium `equal` Bei diesem Ordnungskriterium werden die Einträge nach dem Zustand ihrer Schlüsselobjekte aufsteigend sortiert. Die Festlegung eines B-Baumes auf einen bestimmten Schlüsseltyp erfolgt implizit mit dem Einfügen des ersten Schlüssels. Als nachfolgende Objekte können nur Schlüssel eingefügt werden, für deren Typ eine Vergleichsmethode mit den Typen aller bereits im B-Baum befindlichen Schlüssel exisitiert; andernfalls wird ein Fehler signalisiert.

Der in [Wirth 83, S. 283–287] angegebene Code wurde übernommen und auf das in der 2. Schicht verwendete Objektsystem angepaßt; ferner werden alle Operationen in das Sperrprotokoll eingebettet.

7.5.8 Das C-PLOB! Wurzelobjekt

Die 1., 2. und 3. Schicht haben jeweils eigene Wurzelobjekte; ein Wurzelobjekt einer höheren Schicht wird transitiv von allen Wurzelobjekten der niedrigeren Schichten referenziert. Zur leichteren Unterscheidung werden die Wurzelobjekte im folgenden Text mit der Nummer ihrer Schicht-Ebene versehen, z.B. bezeichnet ‚2. Wurzelobjekt' das Wurzelobjekt der 2. Schicht (Abbildung 7.10).

Das 2. Wurzelobjekt wird als einziges Objekt vom 1. Wurzelobjekt referenziert. Es enthält folgende Komponenten:

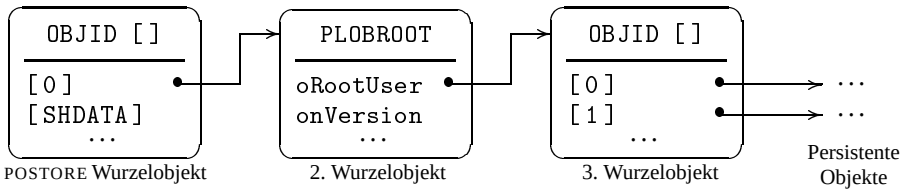


Abbildung 7.10: Wurzelobjekte der Schichten 1–3

Drittes Wurzelobjekt Innerhalb des 2. Wurzelobjektes gibt es eine Referenz auf das Wurzelobjekt der 3. Schicht.

Versionsnummer Sie gibt an, mit welcher Version das 2. Wurzelobjekt in den persistenten Speicher geschrieben wurde; über die Versionsnumer können Inkompatibilitäten zwischen der Struktur des im persistenten Speicher befindlichen 2. Wurzelobjektes und ihrer Strukturdefinition im Code detektiert und aufgelöst werden.

Transaktionszähler Jede Transaktion erhält zur Identifikation eine eindeutige Nummer; innerhalb des 2. Wurzelobjektes wird diese Nummer durch Inkrementieren des Transaktionszählers generiert.

Liste von freien Objekten Persistente Instanzen bestimmter Klassen werden bei expliziter Freigabe in eine Freiliste eingeordnet; solange die Freiliste nicht leer ist, werden dann Objekte nicht neu alloziert, sondern aus der Freiliste entfernt und wiederverwendet. Damit entfällt für diese Objekte der Aufruf der Allozierungsfunktion aus der 1. Schicht.

Warteschlange mit passiv wartenden Sitzungen In der Warteschlange befinden sich nach Prioritäten geordnete Sitzungen, die eine inkompatible Sperranforderung stellten und daher passiv warten müssen, bis entweder ihre Wartezeit abgelaufen ist oder ihre Sperranforderung erfüllt werden kann.

Sperrzustand des gesamten persistenten Speichers Der Sperrzustand für die Ebene *Speicher* wird im 2. Wurzelobjekt repräsentiert.

Zähler für die Anzahl der Aufrufe der Speicherrückgewinnung Nach einer Speicherrückgewinnung haben sich durch die in der letzten Phase durchgeführte Kompaktierung des persistenten Speichers die *long objids* der persistenten Objekte unter Umständen verschoben. Der Zähler für die Anzahl der Aufrufe der Speicherrückgewinnung wird benutzt, um durch Vergleich mit einem Referenzwert einen zwischenzeitlichen Aufruf der Speicherrückgewinnung und damit eine mögliche Verschiebung der *long objids* zu detektieren. Diese Detektion wird für persistente B-Bäume mit dem Ordnungskriterium `eq` genutzt, um nach einer Speicherrückgewinnung die B-Bäume neu aufzubauen.

Innerhalb der 2. Schicht wird auch bereits die Repräsentationsform des 3. Wurzelobjektes festgelegt, um verschieden realisierten 3. Schichten einen konsistenten Zugriff auf die persistenten Objekte über die 2. Schicht zu gewährleisten. Das 3. Wurzelobjekt hat unter anderem folgende Komponenten:

Liste der Sitzungen Alle Sitzungen befinden sich in einer Liste, damit sie gegebenenfalls von der 2. Schicht aus benachrichtigt werden können.

Zeitpunkt der Formatierung Bei der Formatierung werden die Klassen-Beschreibungsobjekte aller Beschreibungsobjekt-Klassen in den persistenten Speicher geschrieben; diese Komponente enthält den Zeitpunkt der letzten Formatierung in Common LISP *Universal Time*.

Pakettabelle Die Pakettabelle (*package table*) wird durch einen persistenten B-Baum repräsentiert, der ein persistentes Paket zu einen Paketnamen (*package name*) in Form einer Zeichenkette assoziiert. Innerhalb des persistenten Pakets befindet sich eine Tabelle seiner internen persistenten Symbole, die durch einen persistenten B-Baum repräsentiert wird, der ein persistentes Symbol zu seinem Namen in Form einer Zeichenkette assoziiert.

Klassentabelle Die persistente Klassentabelle wird durch einen persistenten B-Baum repräsentiert, der ein persistentes Klassen-Beschreibungsobjekt zu einem Klassenamen in Form eines persistenten Symbols assoziiert.

Beschreibungsobjekte Schließlich enthält das 3. Wurzelobjekt die *objids* der Klassen-Beschreibungsobjekte aller Beschreibungsobjekt-Klassen (Abschnitt 7.6.2, S. 103).

7.5.9 Fehlerbehandlung

Wie bereits in Abschnitt 7.2.4 (S. 83) erwähnt, konnte für die Fehlersuche in der 2. Schicht kein *Debugger* verwendet werden; schlimmer noch war der Umstand, daß die Laufzeitfehler im C Code nicht vom LISP-System abgefangen werden konnten und ohne weitere Warnung zur Terminierung des LISP Prozesses führten. Der C Code wurde daher so erstellt, daß möglichst viele der Bedingungen, die zu einem Laufzeitfehler führen, kontrolliert erkannt werden; bei einem derartig erkannten Fehler wird mit einer entsprechenden Meldung eine Funktion der 3. Schicht aufgerufen, die den Fehler wiederum dem LISP *Debugger* signalisiert. Die Fehlerbehandlung ist die einzige Funktion der 2. Schicht, die direkt in die 3. Schicht verzweigt.

7.5.10 Zusammenfassung der 2. Schicht

Die 2. Schicht erfüllt die ACID Eigenschaften (S. 84) folgendermaßen:

Unteilbarkeit Unteilbarkeit wird durch das realisierte hierarchische Sperrprotokoll erreicht. Sperren sind an Transaktionen gebunden; bei Ende einer Transaktion werden alle an sie gebundenen Sperren aufgehoben.

Konsistenz Zuständsänderungen, die einen temporär inkonsistenten Gesamtzustand zur Folge haben, müssen der 2. Schicht durch Starten einer Transaktion mitgeteilt werden. Bei Ende der Transaktion wird der neue Gesamtzustand als konsistent angenommen; bei einem Abbruch wird der letzte konsistente Gesamtzustand wiederhergestellt.

Serialisierbarkeit Die in der 2. Schicht realisierten Zwei-Phasen-Transaktionen stellen Serialisierbarkeit sicher [Jessen et al. 87, S. 163].

Nach der in [Gray et al. 93, S. 401] angegebenen Übersicht realisiert die 2. Schicht eine Serialisierbarkeit von 3°.

Beständigkeit Für die Beständigkeit gibt es keinen Unterschied zur 1. Schicht.

Der folgende Abschnitt faßt noch einmal kurz die Funktionalitäten der 2. Schicht aus der Sicht der nächsthöheren 3. Schicht zusammen.

Typisierung der persistenten Objekte Alle persistenten Objekte sind dynamisch typisiert.

Allozieren eines persistenten Objektes Das Allozieren eines persistenten Objektes erfolgt anhand seiner Typkennung; zurückgegeben wird die zur Referenzierung des persistenten Objektes nötige *short objid*. Das erzeugte Objekt ist bereits typisiert.

Zugriff auf den Zustand von persistenten Objekten Zugriffe auf persistente Objekte werden immer in an Transaktionen gebundene Sperren eingebettet (*Lock-Commit*-Modell [Cardelli et al. 88, S. 38]); ferner wird beim Zugriff auf einzelne Worte geprüft, ob sie innerhalb des Referenz- bzw. Wertefeldes liegen.

Wurzelobjekt Struktur und Zustand des 2. Wurzelobjektes werden innerhalb der 2. Schicht definiert und verwaltet; es kann nicht explizit gelesen oder überschrieben werden außer der frei zugreifbaren Komponente, die die *short objid* des Wurzelobjektes der 3. Schicht enthält.

Speicherrückgewinnung Wie in der 1. Schicht muß die Speicherrückgewinnung explizit veranlaßt werden.

7.6 Schicht 3: Die PLOB! Schicht

Für die Realisierung der dritten Schicht wurde LISPWORKS Common LISP benutzt; eine Portierung nach ALLEGRO Common LISP ist geplant.

Abbildung 7.11 zeigt einen Gesamtüberblick, wie eine Sitzung Zugriff auf persistente Ob-

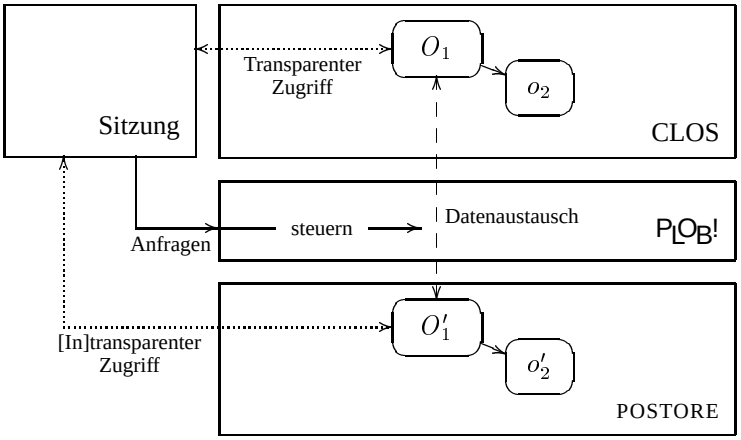


Abbildung 7.11: Zugriff auf persistente Objekte

jekte erhält. Die Aufgabe der 3. Schicht besteht darin, die von der Sitzung kommenden Anfragen (Funktionsaufrufe) zu bearbeiten und unter Benutzung der 2. Schicht den Zugriff auf die persistenten Objekte zu ermöglichen. Ein persistentes Objekt erhält zusätzlich zu seiner persistenten eine transiente CLOS representation, auf die die Sitzung transparent zugreifen kann. Zusätzlich kann eine Sitzung auch direkt die persistente Repräsentation eines persistenten Objektes manipulieren.

7.6.1 Der *Cache*

Der *Cache* ist als assoziativer transienter Zwei-Wege-Speicher in der 3. Schicht realisiert; er bildet sowohl die innerhalb des persistenten Speichers zur Identitatsrepräsentation verwendeten *objids* auf die Identitätsrepräsentation der transienten Repräsentation eines persistenten Objektes ab als auch umgekehrt. Da der Zustand eines *Immediates* bereits in seiner Identitätsrepräsentation enthalten ist, wird für *Immediates* kein Speicher zu ihrer Repräsentation alloziert; damit entfällt auch die Notwendigkeit, in den *Cache* Referenzen auf *Immediates* einzutragen.

Effizienzsteigerung

Eine Referenz auf die transiente Repräsentation eines persistenten Objektes werden assoziiert zur *short objid* in den *Cache* eingetragen; für nachfolgende Zugriffe auf die transiente Repräsentation desselben persistenten Objektes werden so erneute Aufrufe der unteren Schichten unnötig.

Identitätserhaltung

Soweit möglich, sollte die transiente Repräsentation eines persistenten Objektes die Identitätsrepräsentation (d.h. die *short objid*) der persistenten Repräsentation enthalten; damit kann die für ein Objekt vergebene *short objid* aus der transienten Repräsentationsform des Objektes selbst ermittelt werden. Die Repräsentationsform der Instanzen von Basistypen kann nicht zur zur Aufnahme der Identitätsrepräsentation erweitert werden; mit der Umkehrabbildung des *Cache* wird eine Abbildung der Identität der transienten Repräsentation (d.h. der Adresse) auf die Identität der persistenten Repräsentation (d.h. die *short objid*) eines persistenten Objektes einer Basisklasse realisiert.

Für *Immediates* ergibt sich ihre Identitätsrepräsentation unabhängig von ihrer Umgebung aus ihrem Zustand; damit wird ihre Identität durch den Zustand selbst erhalten.

Speicherrückgewinnung des transienten Systems

Da die Referenzen auf die transienten Repräsentationen aller dereferenzierten persistenten Objekte in den *Cache* eingetragen werden, sind sie für das LISP-System erreichbar; damit werden sie bei einer Speicherrückgewinnung des LISP-Systems nicht freigegeben, auch wenn sie ausschließlich durch den *Cache* referenziert werden.

Die Lösung besteht darin, für die Referenzen auf die transienten Repräsentationen im *Cache* schwache Referenzen zu verwenden. Leider stehen diese in dem zur Realisierung verwendetem LISPWORKS Common LISP nicht zur Verfügung; statt dessen muß der *Cache* falls nötig von den BenutzerInnen durch Aufruf einer entsprechenden Funktion gelöscht werden. Bei der Wahl des Zeitpunktes muß berücksichtigt werden, daß durch das Löschen der *Cache* nicht länger identitätserhaltend wirkt.

Konsistenz

Im *Cache* befinden sich die transienten Repräsentationen von persistenten Objekten ohne eine feste Kopplung an die persistente Repräsentation. Sofern mehrere Sitzungen gleichzeitig modifizierend auf die persistenten Objekte zugreifen, kann es passieren, daß die Zustände der transienten Repräsentation im *Cache* und der persistenten Repräsentation nicht mehr übereinstimmen.

Bei der Modifikation eines persistenten Objektes sollten daher alle Sitzungen, die eine transiente Repräsentation dieses Objektes im *Cache* stehen haben, benachrichtigt werden, daß sich der Zustand geändert hat [Rowe 87, S. 17–21]. Realisiert werden kann dies durch einen Sperrmodus, der lediglich anzeigt, daß das gesperrte Objekt im *Cache* einer Sitzung eingetragen ist; bei Zustandsänderungen werden dann alle Sitzungen, die diese Sperre gesetzt haben, benachrichtigt und können ihren *Cache* aktualisieren (Benachrichtigungssperre (*notify lock*) [Fernandez et al. 89, S. 342]). Die Realisierung dieses Sperrmodus erfolgte bisher nicht; sie ist für eine spätere Programmversion vorgesehen.

7.6.2 Datentypen

Dieser Abschnitt beschreibt die realisierten Strukturen verschiedener Datentypen, die benutzt werden, um transiente LISP Objekte persistent zu halten.

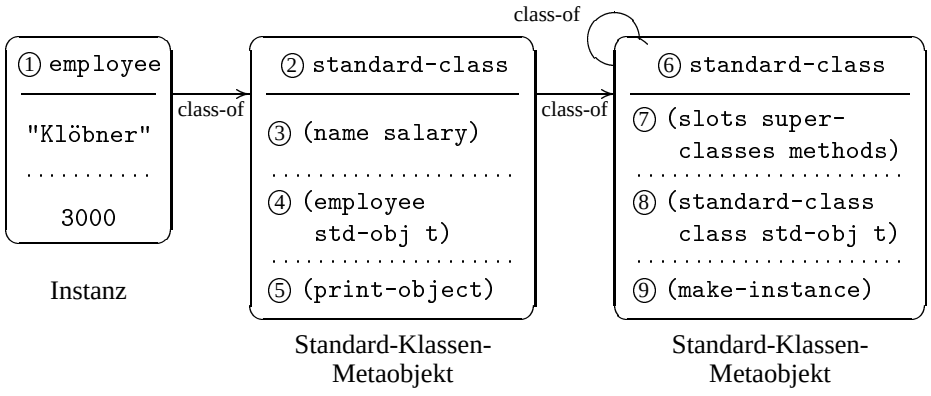
Instanzen von Basistypen

Der Zustand von Instanzen von LISP Basistypen wird in eine entsprechende Instanz eines in der 2. Schicht definierten Basistyps abgebildet.

Klassenbeschreibungen von CLOS Klassen

Bei der Verwendung von relationalen Datenbanken als persistenter Speicher werden Klassen oft nicht explizit gespeichert, sondern ihre Strukturdefinition geht lediglich implizit in die Festlegung der zur Instanzrepräsentation verwendeten Relationen ein (Abschnitt 5.3.1, S. 53). Ein Objektspeicher hingegen bietet die Möglichkeit, auch Klassen ähnlich wie Klassen-Metaobjekte explizit zu repräsentieren.

In CLOS werden Klassen-Metaobjekte zur Repräsentation von Klassen verwendet; sie enthalten alle Informationen über die Struktur und das Verhalten der Instanzen der repräsentierten Klasse (Abbildung 7.12). CLOS-Objekte sind damit selbstbeschreibend. Die im persistenten



Struktur und Verhalten einer Instanz ① wird durch ihr Klassen-Metaobjekt ② festgelegt; es enthält unter anderem eine Liste der *Slots* ③, eine Klassen-Präzedenzliste ④ und eine Liste der Methoden ⑤. Die Struktur der Klasse ② selbst wird durch ihr Klassen-Metaobjekt ⑥ definiert. Sie enthält unter anderem ebenfalls eine Liste der *Slots* ⑦, eine Klassen-Präzendenzliste ⑧ und eine Liste der Methoden ⑨.

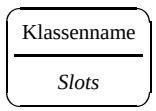


Abbildung 7.12: CLOS-Instanz und Klassen-Metaobjekt

System enthaltenen Objekte sind durch ihre Repräsentationsform (Abbildungen 7.6, S. 92 und 7.7, S. 93) ebenfalls selbstbeschreibend. In der 3. Schicht wurden wie in Abschnitt 7.5.4 (S. 95) erklärt Klassen-Beschreibungsobjekt-Klassen für Klassenbeschreibungen von persistenten Instanzen definiert, die Informationen über die Struktur der beschriebenen Instanzen enthalten (Abschnitt 5.2.2, S. 51). Wegen der in Abschnitt 5.2.3 (S. 51) genannten Restriktionen wird das Verhalten von persistenten Instanzen nicht in einem Klassen-Beschreibungsobjekt mit abgelegt. Der Begriff ‚Beschreibung' wurde somit gewählt, um zu verdeutlichen, daß ein Klassen-Beschreibungsobjekt im Gegensatz zu einem Klassen-Metaobjekt nicht alle Informationen über die repräsentierte Klasse enthält.

Aufgrund der Ähnlichkeit zwischen Klassen-Metaobjekten und der für Klassen-Beschreibungsobjekte geforderten Eigenschaften in Bezug auf die Repräsentation von Informationen über die Struktur von Instanzen wurden die Klassen der von einem Klassen-Beschreibungsobjekt referenzierten Beschreibungsobjekte aus den in [AMOP] gegebenen Spezifikationen für Standard-Metaobjekt-Klassen abgeleitet. Die Spezifikation in [AMOP] vermeidet allerdings eine Festlegung der Struktur der Standard-Metaobjekt-Klassen, sondern legt ausschließlich ihre Stellung innerhalb der Metaobjekt-Klassenhierarchie und ihr Verhalten fest; die spezifizierten generischen Standard-Lese-Funktionen (*standard reader generic functions*) gaben aber indirekt Hinweise auf die Festlegung ihrer Struktur, d.h. für jede generische Standard-Lese-Funktion wurde ein entsprechender *Slot* innerhalb der Beschreibungsobjekt-Klasse definiert. Es ergibt sich ein ähnlicher Zusammenhang wie für Standard-Metaobjekt-Klassen (Abbildung 7.13).

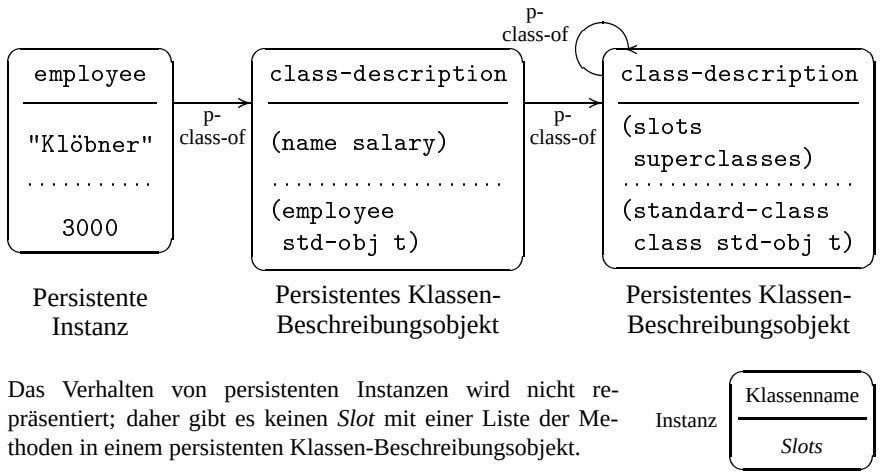


Abbildung 7.13: Persistente Instanz und Klassen-Beschreibungsobjekt

Um Portabilität der persistenten Objekte zu ermöglichen, wurden nach der Bestimmung der Beschreibungsobjekt-Klassen ihre Strukturen in der 2. Schicht festgelegt (Abschnitt 7.5.4, S. 95). Nachträgliche Änderungen an Beschreibungsobjekt-Klassen in der 3. Schicht müssen dementsprechend in ihre Strukturdefinitionen in der 2. Schicht propagiert werden.

Klassenbeschreibungen von Strukturklassen

Die Repräsentation von Strukturklassen ist im Gegensatz zu CLOS Klassen nur teilweise definiert; in [CLtLII, S. 800] wird zwar definiert, daß eine Strukturklasse durch ein Klassen-Metaobjekt repräsentiert wird, es wird aber nicht weiter festgelegt, welche Informationen über die Strukturklasse von ihrem Klassen-Metaobjekt erfragt werden können. Wegen derÄhnlichkeit

zwischen Struktur- und CLOS-Klassen orientiert sich die letztendlich gewählte Repräsentation der Beschreibung von Strukturklassen ebenfalls an dem in [AMOP] definiertem Standard.

Objekte und Klassen

Die Struktur eines persistenten Objektes wird im persistenten Speicher durch sein Klassen-Beschreibungsobjekt und die von ihm referenzierten Beschreibungsobjekte, speziell *Slot*-Beschreibungsobjekte, festgelegt; die im persistenten Klassen-Beschreibungsobjekt und den *Slot*-Beschreibungsobjekten enthaltenen Informationen stammen aus dem Klassen-Metaobjekt des transienten Objektes sowie aus den davon referenzierten *Slot*-Metaobjekten. Es wird nur die Struktur eines persistenten Objektes beschrieben, da der Funktionscode von Methoden aus den in Abschnitt 5.2.3 (S. 51) genannten Gründen nicht gespeichert werden kann. Die Hierarchie der definierten Beschreibungsobjekt-Klassen zeigt Abbildung 7.14. Wie bereits in Abschnitt 7.6.2

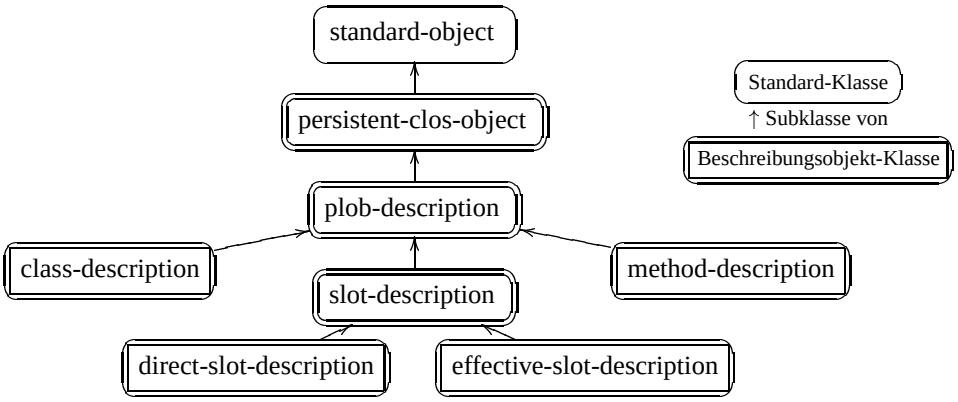


Abbildung 7.14: Beschreibungsobjekt-Klassen-Hierarchie für CLOS Klassen

(S. 103) erläutert, wurden die Strukturen der Beschreibungsobjekt-Klassen aus den Strukturen der Standard-Metaobjekt-Klassen abgeleitet (Tabelle 7.15). Für das Lesen der *Slot*-Zustände

	Standard-Metaobjekt-Klasse	Beschreibungsobjekt-Klasse
❑	**metaobject**	**plob-description**
	standard-class	**class-description**
❑	**slot-definition**	**slot-description**
	direct-slot-definition	**direct-slot-description**
	effective-slot-definition	**effective-slot-description**
	standard-method	**method-description**[a]

❑ ≡ Abstrakte Superklasse

[a] Die Beschreibungsobjekt-Klasse **method-description** wird zur Zeit nicht weiter benutzt.

Tabelle 7.15: Gegenüberstellung von Metaobjekt-Klassen und Beschreibungsobjekt-Klassen

der Beschreibungsobjekte wurden Methoden der generischen Standard-Lese-Funktionen auf die entsprechenden Beschreibungsobjekt-Klassen spezialisiert, so daß sich Beschreibungsobjekte ähnlich wie Metaobjekte verhalten.

Die Struktur der Klassen-Beschreibungsobjekt-Klasse class-description Eine Instanz der Klassen-Beschreibungsobjekt-Klasse **class-description** beschreibt eine Klasse; sie enthält unter

anderem folgende aus dem transienten Klassen-Metaobjekt ermittelten Informationen:

- Den Namen der Klasse.
- Die Superklassen und die Klassen-Präzedenzliste als Liste von Instanzen der Klassen-Beschreibungsobjekt-Klasse **class-description**.
- Die Liste der direkten bzw. effektiven *Slots* als Liste von Instanzen der *Slot*-Beschreibungsobjekt-Klassen **direct-slot-description** bzw. **effective-slot-description**.

Zusätzlich werden unter anderem noch folgende Informationen abgelegt:

- Das Klassen-Beschreibungsobjekt der Klasse, deren Namen in der Klassendefinition im Makro `defclass` der beschriebenen Klasse mit der Klassenoption `:metaclass` angegebenen wurde. Fehlt diese Klassenoption, wird das Klassen-Beschreibungsobjekt der Klasse **standard-class** abgelegt.
- Die Art der Schemaentwicklung (*schema evolution*), die für die Klasse benutzt werden soll.
- Der Name einer Konstruktor-Funktion. Sofern der Name nicht-`nil` ist, wird die an den Namen gebundene Funktion zur Erzeugung der transienten Repräsentation eines persistenten Objektes der beschriebenen Klasse aufgerufen, ansonsten wird die Funktion **make-instance** aufgerufen.

Die Struktur der *Slot*-Beschreibungsobjekt-Klasse slot-description Eine Instanz der *Slot*-Beschreibungsobjekt-Klasse **slot-description** beschreibt einen *Slot* einer Klasse; sie enthält unter anderem folgende aus dem transienten *Slot*-Metaobjekt ermittelten Informationen:

- Den Namen des *Slots*.
- Die Liste der Initialisierungs-Argumentnamen (*initargs*), den Initialisierungsausdruck (*initform*) und die Initialisierungsfunktion (*initfunction*).
- Die Allozierungsart (*slot allocation*) des *Slots*.

Zusätzlich werden unter anderem noch folgende Informationen abgelegt:

- Die Lebensdauer (*extent*) des *Slot*-Zustands.
- In einem direkten *Slot*-Beschreibungsobjekt befindet sich ein S-Ausdruck, der den für den *Slot* zu haltenden Index definiert; ein effektives *Slot*-Beschreibungsobjekt enthält eine daraus generierte Indextabelle, die assoziativ jeden Zustand des beschriebenen *Slots* auf eine dazugehörige persistente CLOS-Instanz abbildet. Damit können persistente CLOS-Instanzen assoziativ gesucht werden; der Suchschlüssel ist ein bestimmter *Slot*-Zustand, das assoziierte Datum ist die dazugehörige persistente CLOS-Instanz. Als Indextabellen werden persisistente B-Bäume benutzt, die eine Abfrage über einen Schlüsselbereich ermöglichen.

Die Struktur der *Slot*-Beschreibungsobjekt-Klasse direct-slot-description Eine Instanz der Klassen-Beschreibungsobjekt-Klasse **direct-slot-description** beschreibt einen direkten *Slot* einer Klasse; sie enthält zusätzlich zu den ererbten Informationen eine aus dem transienten direkten *Slot*-Metaobjekt ermittelte Liste mit den Namen der generischen Zugriffs-Funktionen.

Die Struktur der *Slot*-Beschreibungsobjekt-Klasse effective-slot-description Eine Instanz der *Slot*-Beschreibungsobjekt-Klasse **effective-slot-description** beschreibt einen effektiven *Slot* einer Klasse; sie enthält zusätzlich zu den ererbten Informationen eine Positions-Information (*slot location*). Für *Slots* mit der Allozierungsart `:instance` ist dies die Position des Elementes mit dem *Slot*-Zustand innerhalb des Zustandsvektors (Abbildung 7.7, S. 93). Bei *Slots* mit der Allozierungsart `:class` wird in der Positions-Information der Zustand des *Slots* abgelegt.

Strukturobjekte und Strukturklassen

Abbildung 7.16 zeigt die definierten Beschreibungsobjekt-Klassen für Strukturklassen. Im Gegensatz zu CLOS Klassen gibt es für Strukturklassen keine Standardisierung im Sinne des MOP; die in einem Struktur-Klassen-Beschreibungsobjekt abgelegten Informationen müssen daher anderweitig besorgt werden. Abschnitt 7.6.7 (S. 129) erläutert die gewählte Lösung.

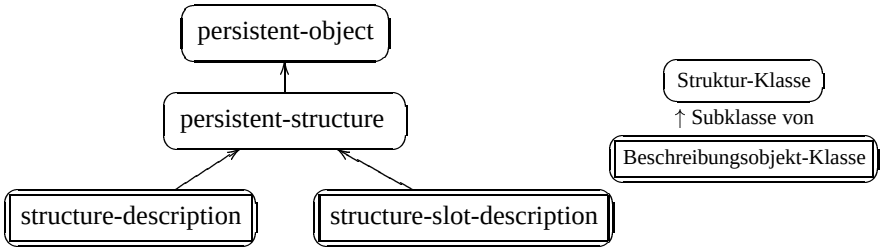


Abbildung 7.16: Beschreibungsobjekt-Klassen-Hierarchie für Strukturklassen

Die Struktur der Klassen-Beschreibungsobjekt-Klasse structure-description Eine Instanz der Klassen-Beschreibungsobjekt-Klasse **structure-description** repräsentiert eine Beschreibung einer Strukturklasse; sie enthält unter anderem folgende Informationen:

- Den Namen der Klasse.
- Die Liste der effektiven *Slots* als Liste von Instanzen der *Slot*-Beschreibungsobjekt-Klasse **structure-slot-description**.
- Die Art der Schemaentwicklung.
- Den Namen der Konstruktor-Funktion.

Die Struktur der *Slot*-Beschreibungsobjekt-Klasse structure-slot-description Eine Instanz der *Slot*-Beschreibungsobjekt-Klasse **structure-slot-description** beschreibt einen effektiven *Slot* einer Strukturklasse; sie enthält unter anderem folgende Informationen:

- Den Namen des *Slots*.
- Das Initialisierungs-Schlüsselwort-Symbol und den Initialisierungsausdruck.
- Die Lebensdauer des *Slot*-Zustands.
- Die Position des Elementes mit dem *Slot*-Zustand innerhalb des Referenzfeldes (Abbildung 7.6, S. 92).

Da für Strukturklassen keine multiple Vererbung möglich ist, gibt es hier keine Unterscheidung zwischen direkten und effektiven *Slots*; sie entsprechen in etwa den effektiven *Slots* einer CLOS Klasse.

Funktionen

Der hier gewählte Ansatz sieht Funktionen als extern-persistente Objekte an und ermöglicht damit eine teilweise Speicherung. Von einer Funktion wird nur der Name gespeichert; der Name einer persistenten Funktion wird durch ein persistentes Symbol repräsentiert. Kann der Name einer transienten Funktion nicht ermittelt werden, wird statt der *short objid* einer gespeicherten Funktion die *short objid* des Markierungsobjektes‚nicht speicherbares Objekt' als Ergebnis der Speicherung zurückgegeben.

Beim Laden wird eine (transiente) Funktion im LISP *image* gesucht, die an das transiente Symbol mit dem durch den Namen der persistenten Funktion vorgegebenen persistenten Symbolnamen und persistenten Paketnamen gebunden ist. Damit wird implizit angenommen, daß eine unter dem gleichen Namen im transienten System‚wiedergefundene' Funktion identisch zu der ursprünglich gespeicherten Funktion ist. Wenn keine Funktion gefunden wird, wird stattdessen eine Funktion zurückgegeben, die bei einem Aufruf einen Fehler signalisiert.

7.6.3 Spezialisierte Metaobjekt-Klassen

Anders als in SOH [Rowe 87, S. 9] und PCLOS [Paepke 91a, S. 11] ist in PLOB! die Möglichkeit der Persistenz grundsätzlich für alle Objekte unabhängig von ihrer Klasse gegeben; die realisierten spezialisierten Metaobjekt-Klassen werden lediglich als Hilfsmittel benutzt, um transiente Instanzen der aus der spezialisierten Klassen-Metaobjekt-Klasse **persistent-metaclass** hervorgegangenen Klassen mit zusätzlichen Eigenschaften bezüglich Persistenz zu versehen. Die durch ein Klassen-Metaobjekt der spezialisierten Klassen-Metaobjekt-Klasse **persistent-metaclass** repräsentierten Klassen sind aktiv-persistente Klassen; alle anderen Klassen sind passiv-persistent.

Aktiv-persistente Klassen haben gegenüber passiv-persistenten folgende zusätzliche Eigenschaften:

- In der Klassendefinition kann innerhalb einer *Slot*-Definition mit einer *Slot*-Option ein Index für den *Slot* deklariert werden; die Verwaltung des Indexes wird vollständig in der 3. Schicht durchgeführt.

- Die Lebensdauer jedes *Slot*-Zustands kann einfacher deklariert werden; die Deklaration kann direkt als *Slot*-Option in die *Slot*-Definition innerhalb der Klassendefinition eingefügt werden. Für einen *Slot* einer passiv-persistenten Instanz muß die Deklaration über die Lebensdauer seines Zustands über einen Funktionsaufruf an das persistente System übergeben werden.

- Für den *Slot*-Zustand stehen mehr Repräsentationsarten zur Verfügung als bei passiv-persistenten Klassen (Abbildung 7.27, S. 120).

- Die Art der Schemaentwicklung kann einfacher deklariert werden; sie kann direkt als Klassen-Option in die Klassendefinition eingefügt werden. Für eine passiv-persistente Klasse muß die Deklaration für die Schemaentwicklung über einen Funktionsaufruf an das persistente System übergeben werden.

Dieser Abschnitt beschreibt zunächst die zur Realisierung dieser zusätzlichen Eigenschaften festgelegten Strukturen der spezialisierten Metaobjekt-Klassen; ihr Verhalten wird in Abschnitt 7.6.5 (S. 111) erläutert. Abbildung 7.17 zeigt die Hierarchie der spezialisierten Klassen- und *Slot*-Metaobjekt-Klassen.

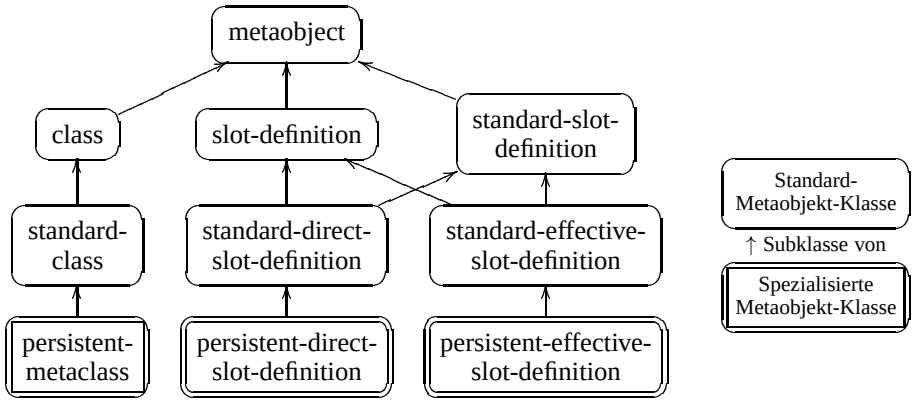


Abbildung 7.17: Spezialisierte Klassen- und *Slot*-Metaobjekt-Klassen

Die Struktur der spezialisierten Klassen-Metaobjekt-Klasse persistent-metaclass

Eine Instanz der Klassen-Metaobjekt-Klasse **persistent-metaclass** repräsentiert eine aktiv-persistente Klasse. Sie enthält unter anderem folgende aus den zusätzlichen Klassenoptionen der `defclass`-Anweisung ermittelten Informationen, die zum Aufbau des Klassen-Beschreibungsobjektes benutzt werden:

Klassenoption `:extent` Die auf Klassenebene deklarierte Lebensdauer der *Slot*-Zustände; sie wird standardmäßig für alle *Slots* benutzt, in deren *Slot*-Definition keine andere Lebensdauer deklariert wird.

Klassenoption `:schema-evolution` Die Art der Schemaentwicklung, die für die Klasse deklariert wurde.

Klassenoption `:constructor` Der Name der Konstruktor-Funktion.

Zusätzlich enthält sie noch folgende Informationen:

- Eine Statusvariable (*flag*), die angibt, ob die Klassendefinition seit der Speicherung der letzten Version des Klassen-Beschreibungsobjektes geändert wurde.
- Eine Referenz auf die transiente Repräsentation des zur Klasse gehörigen Klassen-Beschreibungsobjektes.
- Eine Statusvariable mit der Information, ob die Initialisierung des Klassen-Beschreibungsobjektes endgültig abgeschlossen ist (*finalized*).

Die Strukturen der spezialisierten *Slot*-Metaobjekt-Klassen persistent-direct-slot-definition und persistent-effective-slot-definition

Instanzen der *Slot*-Beschreibungsobjekt-Klassen **persistent-direct-slot-definition** bzw. **persistent-effective-slot-definition** repräsentieren einen direkten bzw. effektiven *Slot* einer aktivpersistenten Klasse.

Die in einer Instanz der *Slot*-Beschreibungsobjekt-Klasse **persistent-direct-slot-definition** enthaltenen Informationen stammen aus den zusätzlichen *Slot*-Optionen einer *Slot*-Definition; diese Informationen werden zum Aufbau eines direkten *Slot*-Beschreibungsobjektes benutzt.

***Slot*-Option** `:extent` Die Lebensdauer des *Slot*-Zustands.

***Slot*-Option** `:index` Ein S-Ausdruck mit einer Beschreibung des Indexes, der für den *Slot* angelegt und verwaltet werden soll.

Gleichnamige direkte *Slots* werden im Verlauf der Klasseninitialisierung zu einem einzigen effektiven *Slot* zusammengefaßt (Protokoll 7.20 **(finalize-inheritance)**, S. 113). Eine Instanz der *Slot*-Beschreibungsobjekt-Klasse **persistent-effective-slot-definition** enthält außerdem noch die ursprünglich für den *Slot* spezifizierte Allozierungsart.

Die spezialisierten Methoden-Metaobjekt-Klassen persistent-reader-method und persistent-writer-method

Für den *Slot*-Zugriff auf persistente CLOS-Instanzen wurden die beiden spezialisierten Methoden-Metaobjekt-Klassen **persistent-reader-method** und **persistent-writer-method** definiert (Abbildung 7.18); Instanzen dieser spezialisierten Methoden-Metaobjekt-Klassen ersetzen die Standard-Zugriffs-Methoden.

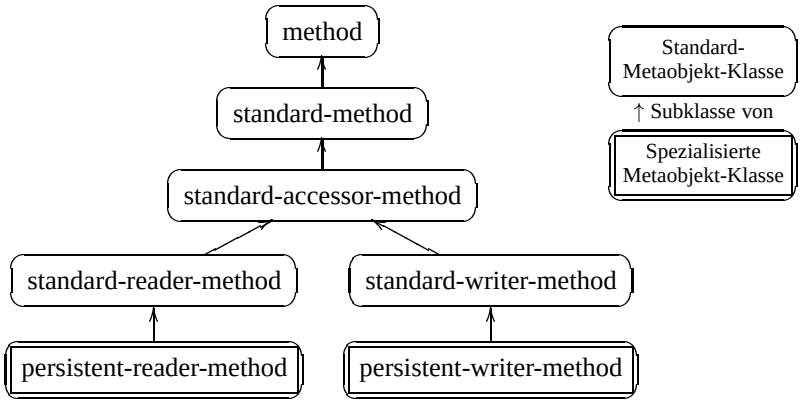


Abbildung 7.18: Spezialisierte Methoden-Metaobjekt-Klassen

7.6.4 Schnitt zwischen Objekten

Beim Transfer eines Objektes zwischen transientem und persistentem Speicher werden normalerweise alle transitiv referenzierten Objekte mit transferiert; wie sich bei der Evaluierung des Systems SOH zeigte, kann dies unter Umständen dazu führen, daß das persistente System versucht, den gesamten Zustand des LISP-Systems zu übertragen. Um die Anzahl der transferierten

Objekte zu begrenzen, können die BenutzerInnen daher Hinweise geben, ab wann die transitiv referenzierten Objekte nicht mit übertragen werden sollen.

Bei Instanzen von Basisklassen kann beim Speichern bzw. Laden ein Parameter angegeben werden, bis zu dem übertragen werden soll; beispielsweise werden bei einer bestimmten Angabe Objekte, die bereits zu einem früheren Zeitpunkt gespeichert oder geladen wurden, nicht weiter betrachtet. Andere Angaben schränken die Anzahl der Objektbestandteile, die gespeichert werden sollen, ein; so kann z.B. für ein Symbol angegeben werden, daß nur der Name des Symbols in die persistente Repräsentation übertragen wird.

Für Struktur- und CLOS-Objekte kann deklarativ die Lebensdauer eines *Slot*-Zustands als transient festgelegt werden; transiente *Slots* werden beim Abspeichern nicht weiter traversiert. Deklarationen beziehen sich auf Klassen und gelten dann für alle ihre Instanzen.

7.6.5 Protokolle

In den folgenden Beschreibungen werden die verschiedenen im Verlauf eines Protokolls durchgeführten Aktivitäten in der Spezifikationssprache INFOL (*Info*rmal *L*anguage) beschrieben. Eingerückter Text spezifiziert eine Verfeinerung der in seiner Überschrift genannten Aktivität; da die Sprache informal ist, existieren keine weiteren Festlegungen bezüglich ihrer Syntax oder Semantik. Die hier benutzte Beschreibungsform richtet sich nach [Paepcke 91b, S. 18, 31–33].

Definition von aktiv-persistenten Klassen

Im Verlauf der Definition von aktiv-persistenten Klassen werden folgende Aufgaben erledigt:

1. Die Deklarationen der aktiv-persistenten Klassen werden ausgewertet; sie befinden sich in der Klassendefinitions (*defclass*)-Anweisung:

   - Deklarationen auf Klassenebene werden als zusätzliche Klassenoptionen spezifiziert und ausgewertet; sie werden in das die Klasse repräsentierende Klassen-Metaobjekt eingetragen.
   - Deklarationen auf *Slot*-Ebene werden als zusätzliche *Slot*-Optionen spezifiziert und ausgewertet; sie werden in die die *Slots* der Klasse repräsentierenden *Slot*-Metaobjekte eingetragen.

2. Für den Zugriff auf den *Slot*-Zustand der Instanzen von aktiv-persistenten Klassen über die in der Klassendefinitions-Anweisung spezifizierten generischen Lese (*:reader*)- und Schreib (*:writer*)-Funktionen werden spezialisierte Methoden-Metaobjekt-Klassen eingebunden, die die verschiedenen möglichen *Slot*-Repräsentationen (Abbildung 7.27, S. 120) korrekt verarbeiten.

In SOH wurde bei jeder Evaluierung einer Klassendefinition die Relation zur Repräsentation der Instanzen in der Datenbank angelegt; dies führte dazu, daß diese Evaluierung relativ viel Zeit in Anspruch nahm. In PLOB! wird daher das die Klasse beschreibende Klassen-Beschreibungsobjekt nicht im Verlauf der Definitionsevaluierung, sondern zum spätestmöglichen Zeitpunkt (d.h. beim nächsten Dereferenzieren des Zustands einer Instanz der Klasse) im Speicher abgelegt.

Protokoll 7.19: Protokoll für die Definition einer aktiv-persistenten Klasse (nach [Paepcke 91b, S. 31]):
(defclass)

1 Syntaxprüfung.
2 Das Klassen-Metaobjekt wird ermittelt:
(ensure-class), (ensure-class-using-class)
2.1 Suche das Klassen-Metaobjekt; wird es nicht gefunden, erzeuge ein Klassen-Metaobjekt als Instanz der Klassen-Metaobjekt-Klasse, dessen Name mit der Klassenoption `:metaclass` angegeben wurde:[6]
(make-instance (find-class <*Wert der Klassenoption* `:metaclass`>**))**
2.2 Die Klassenoptionen werden in eine kanonische Form gebracht:[7] ✘
(clos::canonicalize-class-options)
2.3 (Re)initialisiere das Klassen-Metaobjekt:
((re)initialize-instance)
↑ Bei der Initialisierung werden die Klassenoptionen als Initialisierungs-Schlüsselwort-Parameter an die Methode **(re)initialize-instance** übergeben und im Klassen-Metaobjekt gespeichert.
Die Liste der Superklassen wird um die Klasse **persistent-clos-object** erweitert; damit wird an die definierte Klasse u.a. ein *Slot* vererbt, der für jede Instanz der definierten Klasse die *short objid* der persistenten Repräsentation enthält.
2.3.1 Nicht übergebene Initialisierungs-Schlüsselwort-Parameter werden mit Standardwerten versehen; die Initialisierungs-Schlüsselwort-Parameter werden auf Konsistenz geprüft.
2.3.2 Es wird geprüft, ob die Superklassen kompatibel sind:
(validate-superclasses)
↑ Die Klassen **persistent-metaclass** und **standard-class** sind kompatibel; die Methode gibt daher immer `t` zurück.
2.3.3 Die Superklassen werden zum Klassen-Metaobjekt assoziiert.
2.3.4 Bestimme die *Slot*-Metaobjekt-Klasse der direkten *Slots*:
(direct-slot-definition-class)
↑ Die direkten *Slots* einer aktiv-persistenten Klasse werden durch die spezialisierte *Slot*-Metaobjekt-Klasse **persistent-direct-slot-definition** repräsentiert; sie enthält im Vergleich zur Standard-*Slot*-Metaobjekt-Klasse **standard-direct-slot-definition** zusätzliche *Slots* zur Speicherung der *Slot*-Optionen.
2.3.5 Die *Slot*-Optionen werden in eine kanonische Form gebracht: ✘
(clos::canonicalize-defclass-slot)
2.3.6 Die direkten *Slot*-Metaobjekte werden erzeugt und initialisiert:
(make-instance), (initialize-instance)
↑ Bei der Initialisierung werden die *Slot*-Optionen als Initialisierungs-Schlüsselwort-Parameter an die Methode **initialize-instance** übergeben und im direkten *Slot*-Metaobjekt gespeichert.
2.3.7 Die direkten *Slot*-Metaobjekte werden zum Klassen-Metaobjekt assoziiert.
2.3.8 Die Standard-Initialisierungs-Argumente (*default initargs*) werden zum Klassen-Metaobjekt assoziiert.
2.3.9 Die Klasse wird in die Subklassenliste aller Superklassen eingetragen:
(add-direct-subclass), (remove-direct-subclass)

[6] Für aktiv-persistente Klassen ist der Wert der Klassenoption `:metaclass` immer **persistent-metaclass** oder der Name einer Subklasse der Klasse **persistent-metaclass**.

[7] Dieser Schritt wird nur in LISPWORKS an dieser Stelle ausgeführt; nach [Paepcke 91b, S. 31] befindet er sich normalerweise hinter Schritt 1.

2.3.10 Falls möglich, wird die Vererbungs-Initialisierung wie in Protokoll 7.20 **(finalize-inheritance)** (S. 113) angegeben endgültig abgeschlossen:
(finalize-inheritance)

2.3.11 Bestimme die Methoden-Metaobjekt-Klasse der Zugriffs-Methoden: ✚
(reader-method-class), (writer-method-class)

↑ Die auf **persistent-metaclass** spezialisierte Methode der generischen Funktion **reader-method-class** gibt das Methoden-Metaobjekt der spezialisierten Methoden-Metaobjekt-Klasse **persistent-reader-method**, die auf **persistent-metaclass** spezialisierte Methode der generischen Funktion **writer-method-class** gibt das Methoden-Metaobjekt der spezialisierten Methoden-Metaobjekt-Klasse **persistent-writer-method** zurück; damit werden diese Methoden-Metaobjekt-Klassen für die Generierung der Zugriffs-Methoden benutzt.

2.3.12 Die Methoden-Metaobjekte der Zugriffs-Methoden werden erzeugt und initialisiert: ✚
(make-instance), (initialize-instance) [, (make-method-lambda)]

↑ Da die λ-Liste der in LISPWORKS realisierten generischen Funktion **make-method-lambda** nicht mit dem in [AMOP, S. 207] definierten Standard übereinstimmt, wurde die generische Funktion **make-accessor-lambda** definiert. Ihre Methoden erzeugen die λ-Ausdrücke der Zugriffs-Methoden-Metaobjekte; sie werden übersetzt und im Verlauf der Aufrufe der generischen Zugriffs-Funktionen ausgeführt. Die erzeugten λ-Ausdrücke sind im Moment so aufgebaut, daß sie immer die generische Funktion **slot-value-using-class** aufrufen; in einer späteren Programmversion können optimierte λ-Ausdrücke für den *Slot*-Zugriff auf persistente Instanzen erzeugt werden.

2.3.13 Die Methoden-Metaobjekte der Zugriffs-Methoden werden zum Klassen-Metaobjekt assoziiert: ✚
(add-direct-method)

↑: Beschreibung der realisierten Methode
System-(d.h. LISPWORKS-) spezifischer Schritt: ✘
Hinzugefügter oder geänderter Schritt im Vergleich zu [Paepcke 91b, S. 31]: ✚

Das in [Paepcke 91b, S. 31] angegebene Protokoll ist wegen einiger fehlender Schritte nicht vollständig MOP-konform. Der hinzugefügte Schritt 2.3.11 läßt sich aus den Definitionen der generischen Funktionen **reader-method-class** [AMOP, S. 224] und **writer-method-class** [AMOP, S. 242] herleiten. Für die neuen Schritte 2.3.12 und 2.3.13 findet sich keine direkte Begründung in [AMOP]. Ich habe sie aus der Tatsache hergeleitet, daß in [AMOP] nicht zwischen‚normalen' benutzerInnen-definierten Methoden und aus der Klassendefinition generierten Zugriffs-Methoden unterschieden wird; demzufolge sollten auch sie wie normale Methoden behandelt werden. Leider wird diese Ansicht (noch) nicht von den Autoren des LISPWORKS Common LISP geteilt, da das dort realisierte MOP die Schritte 2.3.11–2.3.13 nicht befolgt; statt dessen werden die Methoden-Metaobjekte der Zugriffs-Methoden im Verlauf der Initialisierung des Klassen-Metaobjektes erzeugt und eingetragen, ohne daß weitere generische Funktionen dazu aufgerufen werden. Um trotzdem die Zugriffs-Methoden durch Instanzen der Methoden-Metaobjekt-Klassen **persistent-reader-method** bzw. **persistent-writer-method** repräsentieren zu können, werden die vom System erzeugten Methoden-Metaobjekte nach ihrer Generierung ersetzt; sinngemäß wird wie in Protokoll 7.19 **(defclass)**, Schritt 2.3.11–2.3.13, spezifiziert vorgegangen.

Protokoll 7.20: Protokoll für den endgültigen Abschluß der Vererbungs-Initialisierung (nach [Paepcke 91b, S. 32]):
(finalize-inheritance)

1 Die Klassen-Präzedenzliste wird berechnet:
(compute-class-precedence-list)
2 Konflikte zwischen gleichnamigen ererbten *Slots* werden aufgelöst:
2.1 Bestimme die *Slot*-Metaobjekt-Klasse der effektiven *Slots*:
(effective-slot-definition-class)
↑ Die effektiven *Slots* einer aktiv-persistenten Klasse werden durch die spezialisierte *Slot*-Metaobjekt-Klasse **persistent-effective-slot-definition** repräsentiert; sie enthält im Vergleich zur Standard-*Slot*-Metaobjekt-Klasse **standard-effective-slot-definition** zusätzliche *Slots* zur Speicherung der aus den direkten *Slots* abgeleiteten Informationen.
2.2 Die effektiven *Slot*-Metaobjekte werden erzeugt:
(make-instance)
2.3 Die effektiven *Slot*-Metaobjekte werden initialisiert:
(initialize-instance), (compute-effective-slot-definition)
↑ Gleichnamige direkte *Slots* werden in der Methode **compute-effective-slot-definition** zu einem effektiven *Slot* zusammengefaßt.
2.4 Die effektiven *Slot*-Metaobjekte werden zum Klassen-Metaobjekt assoziiert.

In der Methode **compute-effective-slot-definition** wird die Lebensdauer des *Slot*-Zustands bestimmt; dazu werden hierarchische Deklarationen ausgewertet. Eine Lebensdauer kann auf einer der Ebenen ‚*Slot*' – ‚Klasse' – ‚Paket des Klassen-Namens' deklariert werden; die Lebensdauer des *Slot*-Zustands ergibt sich aus der ersten Deklaration, die in der angebenen Reihenfolge für eine der Ebenen gefunden wird. Wurde keine Deklaration gefunden, wird für die Lebensdauer des *Slot*-Zustands der in der Konstanten **+default-clos-slot-extent+** enthaltene Standardwert `:cached` benutzt.

Passiv-persistente CLOS- und Struktur-Klassen

Die Möglichkeiten des MOP, Zustand und Verhalten eines Klassen-Metaobjektes im Verlauf seiner Definition wie in Protokoll 7.19 **(defclass)** (S. 111) angegeben über spezialisierte Methoden der vom MOP aufgerufenen generischen Funktionen zu modifizieren, ist zum einen an die Form (Spezifikation der Klassenoption `:metaclass`) und zum anderen an die Evaluierung der Klassendefinition gebunden. Für Klassen, deren Definition bereits evaluiert wurde, besteht daher nicht die Möglichkeit, ihr Klassen-Metaobjekt wie in Protokoll 7.19 **(defclass)** angegeben zu beeinflussen. Um unvorhersehbare Seiteneffekte auszuschließen, verzichtete ich darauf, diese Klassen bei Bedarf neu zu definieren, um eine Modifikation zu erreichen. Statt dessen sind die Möglichkeiten dieser passiv-persistenten Klassen eingeschränkt:

- Da die Zugriffs-Methoden der Klasse nicht verändert werden, kann auf den Objektzustand nur über *Swizzling* zugegriffen werden.
- Aus dem gleichen Grund kann für *Slots* von passiv-persistenten Klassen kein Index definiert werden.
- Da keine zusätzlichen Klassen- oder *Slot*-Optionen spezifiziert werden können, müssen Deklarationen bezüglich Persistenz bei passiv-persistenten Klassen über Aufrufe entsprechender Funktionen erfolgen, die die Deklarationen in Tabellen speichern.

Die eingeschränkten Möglichkeiten gelten sowohl für passiv-persistente CLOS Klassen als auch für Strukturklassen.

Abgleich zwischen einer Klassendefinition und seiner Klassenbeschreibung

Der Abgleich sorgt dafür, daß die aktuelle Klassendefinition im transienten System durch eine äquivalente Klassenbeschreibung im persistenten Speicher repräsentiert wird. Der Abgleich erfolgt immer vor der Übertragung des Zustands eines Struktur- oder CLOS-Objektes zwischen dem transienten und persistenten Speicher für die Klasse der zu übertragenden Instanz. Für Instanzen von Basistypen ist dieser Abgleich nicht notwendig, da ihre Strukturen als unveränderlich festgelegt sind.

Protokoll 7.21: Protokoll für den Abgleich einer transienten CLOS Klassendefinition mit seiner persistentenKlassenbeschreibung:
(ensure-class-description)

1 Unter dem Namen der Klasse wird das dazugehörige Klassen-Metaobjekt gesucht:
(find-class)
2 In der persistenten Klassentabelle wird unter dem Namen der Klasse nach einem bereits existierendem Klassen-Beschreibungsobjekt gesucht:
(p-find-class)
3 Falls in Schritt 2 ein Klassen-Beschreibungsobjekt gefunden wurde:
3.1 Falls in Schritt 1 ein Klassen-Metaobjekt gefunden wurde:
3.1.1 Es wird geprüft, ob sich das transiente Klassen-Metaobjekt aus Schritt 1 seit dem letzten Aufruf von **(ensure-class-description)** geändert haben könnte:
(mismatch-p)
Der Aufruf von **(mismatch-p)** gibt den Wert einer an das transiente Klassen-Metaobjekt assoziierten Statusvariablen zurück, das in den auf die Klasse **class** spezialisierten Methoden der generischen Funktionen **initialize-instance** und **reinitialize-instance** auf `t` gesetzt wird; die (Re-)Initialisierung eines Klassen-Metaobjektes wird damit als Indikator einer eventuell stattgefundenen (Re-)Definition benutzt.
A. Aus dem transienten Klassen-Metaobjekt aus Schritt 1 wird ein transientes Klassen-Beschreibungsobjekt erzeugt:
(fill-description)
Das erzeugte transiente Klassen-Beschreibungsobjekt enthält alle Informationen zur Beschreibung der durch das transiente Klassen-Metaobjekt repräsentierten Klasse, insbesondere auch die *Slot*-Beschreibungsobjekte der direkten und effektiven *Slots*.
B. Es wird geprüft, ob das Klassen-Beschreibungsobjekt aus Schritt 2 äquivalent zu dem Klassen-Beschreibungsobjekt aus Schritt 3.1.1.A ist:
(class-description-equal-p)
C. Sind die Klassen-Beschreibungsobjekte nicht äquivalent, wird das neu erzeugte Klassen-Beschreibungsobjekt aus Schritt 3.1.1.A nicht-destruktiv im persistenten Speicher abgelegt, d.h. das vorhandene alte Klassen-Beschreibungsobjekt bleibt unverändert erhalten. In der persistenten Klassentabelle wird die unter dem Namen der Klasse eingetragene Referenz auf das neue Klassen-Beschreibungsobjekt gesetzt:
(update-class)
Dabei wird auch die in Schritt 3.1.1 abgefragte Statusvariable einer eventuellen Klassenänderung auf `nil` gesetzt. Jetzt ist das Klassen-Beschreibungsobjekt aus Schritt 3.1.1.A das aktuelle Klassen-Beschreibungsobjekt.

3.2 sonst exisitiert ein persistentes Klassen-Beschreibungsobjekt, aber kein transientes Klassen-Metaobjekt. Dies kann beim Laden eines persistenten Objektes passieren, dessen Klassendefinition nicht anderweitig evaluiert oder geladen wurde:

3.2.1 Erzeuge die Klassendefinition aus dem Klassen-Beschreibungsobjekt und evaluiere sie:
(compile-description)

4 sonst existiert noch kein Klassen-Beschreibungsobjekt der Klasse:

4.1 Aus dem transienten Klassen-Metaobjekt von Schritt 1 wird ein transientes Klassen-Beschreibungsobjekt erzeugt:
(fill-description)
Das erzeugte transiente Klassen-Beschreibungsobjekt enthält alle Informationen zur Beschreibung der durch das transiente Klassen-Metaobjekt repräsentierten Klasse, insbesondere auch die *Slot*-Beschreibungsobjekte der direkten und effektiven *Slots*.

4.2 Speichere das erzeugte Klassen-Beschreibungsobjekt:
(update-class)

5 Jetzt ist sichergestellt, daß die Klassendefinition im transienten System und das dazugehörige aktuelle Klassen-Beschreibungsobjekt zueinander passen. Als Ergebnis werden zwei Werte zurückgegeben:

- Das aktuelle transiente Klassen-Beschreibungsobjekt.
- Eine Statusinformation, die angibt, ob in Schritt 3.1.1.C oder 4.2 ein (neues) Klassen-Beschreibungsobjekt gespeichert wurde.

Für Strukturklassen gibt es ein ähnliches Protokoll **(ensure-structure-description)**, das sich vom Protokoll **(ensure-class-description)** bezüglich der hier gewählten Abstraktionsebene nur wenig unterscheidet.

Die Struktur- bzw. Klassen-Beschreibung einer neu erzeugten persistenten Repräsentation verweist immer auf das im Moment der Erzeugung aktuelle Klassen-Beschreibungsobjekt. In Schritt 3.1.1.C wird bei Änderung der Klassendefinition eine neue Version einer Klassenbeschreibung (*type version*) [Skarra et al. 87, S. 401] mit neuer Identität erzeugt; beim nächsten Zugriff auf die persistente Repräsentation wird durch Vergleich des durch sie referenzierten Klassen-Beschreibungsobjektes mit dem aktuellen Klassen-Beschreibungsobjekt gegebenenfalls eine Anpassung der Repräsentation und eine Änderung der Referenz auf das Klassen-Beschreibungsobjekt veranlaßt.

Die persistente Instanz ③ in Abbildung 7.22 wurde erzeugt, als das Klassen-Beschreibungsobjekt mit der Version 1.00 aktuell war (Referenz ①); bei der nächsten Dereferenzierung der persistenten Instanz ③ wird die Referenz ① auf das aktuelle Klassen-Beschreibungsobjekt ② geändert.

Speichern eines transienten Objektes

Das Speichern eines transienten Objektes beinhaltet die Übertragung des Objektzustands in eine persistente Repräsentation. In das Protokoll der Übertragung flossen folgende Überlegungen ein:

1. Die persistente Repräsentation soll den transienten Zustand angemessen widerspiegeln. Daraus folgt, daß sowohl die im transienten Objekt enthaltenen Referenzen als auch die Zustände der referenzierten Objekte ebenfalls mit abgelegt werden müssen.

2. Die Übertragung soll effizient sein; da ein Objekt transitiv sehr viele Instanzen referenzieren kann, sollte die Möglichkeit eines Schnitts bestehen.

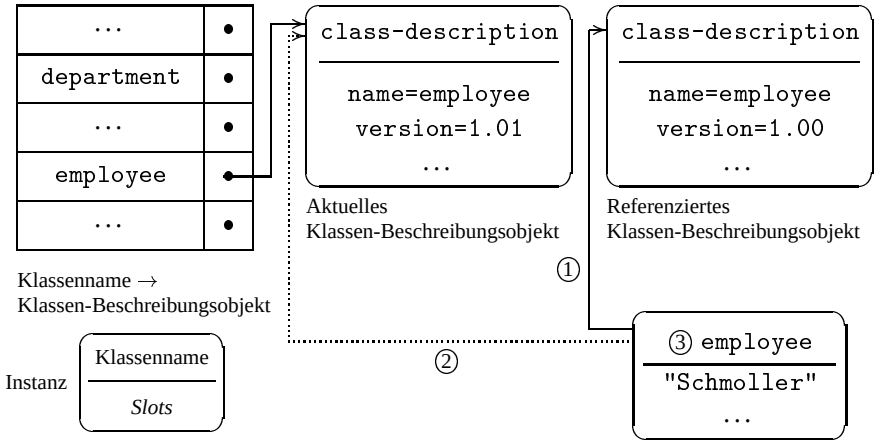


Abbildung 7.22: Schemaentwicklung für eine persistente Instanz

3. Die Identität der Objekte soll berücksichtigt werden; sofern für den Zustand eines transienten Objektes eine entsprechende persistente Repräsentation existiert, soll eine erneute Übertragung des Zustands immer in dieselbe persistente Repräsentation stattfinden.

Die Eigenschaften von Punkt 1 und 2 widersprechen sich; je nach Verwendungszweck des persistenten Systems kann die mehr gewünschte Eigenschaft durch Deklarationen oder Parameter hervorgehoben werden.

Protokoll 7.23: Protokoll für das Speichern eines transienten Objektes:
(store-object)

1 Es wird geprüft, ob die Stable Heap-Datei geöffnet ist:
(assert-sh-open-p)

2 Mit dem Aufruf der generischen Funktion
(t-object-to-p-objid)
wird das transiente Objekt gespeichert:

2.1 Ist das zu speichernde Objekt ein *Immediate*, werden als Ergebnis von Schritt 2 zwei Werte zurückgegeben:

- Der erste Rückgabewert ist die *short objid* des gespeicherten Objektes; für *Immediates* ist dies der Zustand des Objektes.
- Der zweite Rückgabewert ist die Typkennung des *Immediates* nach Tabelle 7.3 (S. 89).

2.2 Für die Instanz einer Struktur- bzw. CLOS-Klasse wird das aktuelle transiente Klassen-Beschreibungsobjekt wie im Protokoll 7.21 **(ensure-class-description)** (S. 115) angegeben ermittelt:
(ensure-structure-description) bzw. **(ensure-class-description)**

2.3 Die *short objid* des Objektes wird ermittelt:

2.3.1 Das transiente Objekt wird im *Cache* gesucht:
(is-registered-object)
Bei erfolgreicher Suche wird als Ergebnis von Schritt 2.3 die zum Objekt gehörige *short objid* der persistenten Repräsentation zurückgegeben.
Der *Cache* stellt somit sicher, daß der Zustand eines transienten Objektes immer in dieselbe persistente Repräsentation übertragen wird.

2.3.2 Wurde das Objekt nicht im *Cache* gefunden, wird ein persistentes Objekt mit einer der transienten Klasse entsprechenden Klasse im persistenten Speicher alloziert:
(p-allocate-<*Klassenname*>**)**
Die erhaltene *short objid* und das transiente Objekt werden in den *Cache* eingetragen:
(register-to-cache)
Die *short objid* wird als Ergebnis von Schritt 2.3 zurückgegeben.

2.4 Sofern noch keine Transaktion aktiv ist, wird eine Transaktion gestartet:
(with-transaction)
Eine hier gestartete Transaktion wird von rekursiven Aufrufen des Protokolls 7.23 **(store-object)** mitbenutzt.

2.5 Für das durch die *short objid* identifizierte persistente Objekt wird eine Schreibsperre angefordert:
(with-write-lock)
Dabei wird festgestellt, ob auf das durch die *short objid* referenzierte persistente Objekt bereits eine Schreibsperre innerhalb der aktiven Transaktion gesetzt wurde. Wenn ja, wurde der Zustand des transienten Objektes in der aktiven Transaktion bereits in die persistente Repräsentation übertragen; in diesem Fall verzweigt das Protokoll zu Schritt 2.7.

2.6 Der Zustand des transienten Objektes wird in die persistente Repräsentation übertragen:

2.6.1 Für Instanzen von Basisklassen siehe das Subprotokoll 7.24 für das Speichern einer transienten Instanz einer Basisklasse (S. 118).

2.6.2 Für Strukturobjekte siehe das Subprotokoll 7.25 **(setf p-structure)** (S. 119).

2.6.3 Für CLOS-Instanzen siehe das Subprotokoll 7.26 **(setf p-instance)** (S. 119).

2.7 Wenn in Schritt 2.4 eine Transaktion gestartet wurde, wird sie in diesem Schritt beendet. Damit werden alle innerhalb der Transaktion gesetzten Sperren (insbesondere die Schreibsperre aus Schritt 2.5) aufgehoben.

2.8 Als Ergebnis von Schritt 2 werden zwei Werte zurückgegeben:

- Der erste Rückgabewert ist die in Schritt 2.3 ermittelte *short objid* des gespeicherten Objektes.
- Der zweite Rückgabewert ist die Typkennung, die den ersten Rückgabewert als *short objid* eines persistentes Objektes typisiert (Tabelle 7.3, S. 89, Kennung 0).

3 Wenn aus dem zweiten Rückgabewert von Schritt 2 hervorgeht, daß ein *Immediate* gespeichert wurde (Tabelle 7.3, S. 89, Kennung $\neq 0$), wird als Ergebnis des Protokolls der *Immediate* selbst zurückgegeben; andernfalls wird die *short objid* des persistenten Objektes innerhalb einer transienten Instanz der Strukturklasse **persistent-object** zurückgegeben, damit sich die Klasse der zurückgegebenen *short objid* vom intern zur Repräsentation von *short objids* verwendeten Typ **fixnum** unterscheidet.

Protokoll 7.24: Subprotokoll für das Speichern einer transienten Instanz einer Basisklasse:

1 Für Instanzen von Basisklassen, die nicht Werte sind, wird für jeden *Slot* des Objektes die generische Funktion
(t-object-to-p-objid)
aufgerufen, d.h. das Protokoll 7.23 **(store-object)** (S. 117) wird rekursiv mit Schritt 2 fortgeführt. Die aus dem rekursiven Aufruf erhaltene *short objid* und die Typkennung des *Slot-*

Zustands wird in die zum *Slot* gehörige Komponente des Referenzfeldes (Protokoll 7.4, S. 90) des persistenten Objektes eingetragen.

2 Für transiente Werte wird ihr Zustand mit der Funktion
(setf p-values)
in ihrer binären Repräsentation in das Wertefeld des persistenten Objektes (Protokoll 7.4, S. 90) kopiert.

Für Symbole kann durch einen dem Protokoll 7.23 mitgegebenen Parameter die Anzahl der *Slots*, deren Zustand vom transienten in das persistente Symbol übertragen werden soll, eingeschränkt werden; so wird für die Angabe `:name` lediglich der Name des Symbols gespeichert. Beim Speichern des Wertes eines Symbols wird immer der dynamisch gebundene Wert benutzt, da zum einen der lexikalisch gebundene Wert nicht einfach ermittelt werden kann [CLtLII, S. 119] und zum anderen in diesem Fall die (unter Umständen sehr umfangreiche) lexikalische Umgebung des Symbols mit gespeichert werden müßte.

Protokoll 7.25: Subprotokoll für das Speichern eines transienten Strukturobjektes:
(setf p-structure)

1 Aus dem in Schritt 2.2 von Protokoll 7.23 **(store-object)** (S. 117) erhaltenem aktuellem transientem Klassen-Beschreibungsobjekt wird die Liste der *Slot*-Beschreibungsobjekte gelesen:
(class-effective-slots)

2 Für jeden *Slot*, dessen Zustands-Lebensdauer nicht als `:transient` deklariert wurde:

2.1 Rufe für den *Slot*-Zustand die generische Funktion
(t-object-to-p-objid)
auf, d.h. das Protokoll 7.23 **(store-object)** (S. 117) wird ab Schritt 2 aufgerufen. Die daraus erhaltene *short objid* und die Typkennung des *Slot*-Zustands wird in die zum *Slot* gehörige Komponente des Referenzfeldes (Protokoll 7.6, S. 92) des persistenten Strukturobjektes eingetragen.

Die Iteration über alle *Slots* sorgt dafür, daß alle Referenzen und referenzierten Objekte mit im persistenten Speicher abgelegt werden; die Deklaration eines transienten *Slot*-Zustands ermöglicht einen Schnitt, ab dem Objekte nicht weiter gespeichert werden.

Ein transientes Paket wird als Instanz der Basis-Strukturklasse **persistent-package** abgespeichert. Die im transienten Paket enthaltene Liste der ihm zugeordneten transienten Symbole wird beim Speichern eines Paketes nicht mit abgelegt; statt dessen werden persistente Symbole im Moment des Speicherns ihrem persistenten Paket zugeordnet.

Protokoll 7.26: Subprotokoll für das Speichern einer transienten CLOS-Instanz:
(setf p-instance)

1 Aus dem in Schritt 2.2 von Protokoll 7.23 **(store-object)** (S. 117) erhaltenem aktuellem transientem Klassen-Beschreibungsobjekt wird die Liste der *Slot*-Beschreibungsobjekte gelesen:
(class-effective-slots)

2 Für jeden *Slot*, dessen Zustands-Lebensdauer als `:cached` deklariert wurde:

2.1 Es wird geprüft, ob der *Slot* an einen Wert gebunden ist:
(slot-boundp)

2.2 Wenn der *Slot* ungebunden ist, wird in die zum *Slot* gehörige Komponente des durch das Referenzfeld referenzierten Zustandsvektors (Protokoll 7.7, S. 93) der persistenten

CLOS-Instanz die *short objid* der Markierung ‚ungebundener *Slot*' und die Typkennung Markierung (Tabelle 7.3, S. 89, Kennung 4) eingetragen:
(setf p-index)

2.3 Wenn der *Slot* an einen Wert gebunden ist, rufe für den *Slot*-Zustand die generische Funktion
(t-object-to-p-objid)
auf, d.h. das Protokoll 7.23 **(store-object)** (S. 117) wird ab Schritt 2 aufgerufen. Die daraus erhaltene *short objid* und Typkennung des *Slot*-Zustands wird in die zum *Slot* gehörige Komponente des durch das Referenzfeld referenzierten Zustandsvektors (Protokoll 7.7, S. 93) der persistenten CLOS-Instanz eingetragen:
(setf p-index)

Um Metaobjekte mit einer fest definierten, systemunabhängigen Struktur zu speichern, werden alle Metaobjekte als Beschreibungsobjekte gespeichert; die Abbildung einer Metaobjekt-Klasse auf seine Beschreibungsobjekt-Klasse ergibt sich nach Tabelle 7.15 (S. 105).

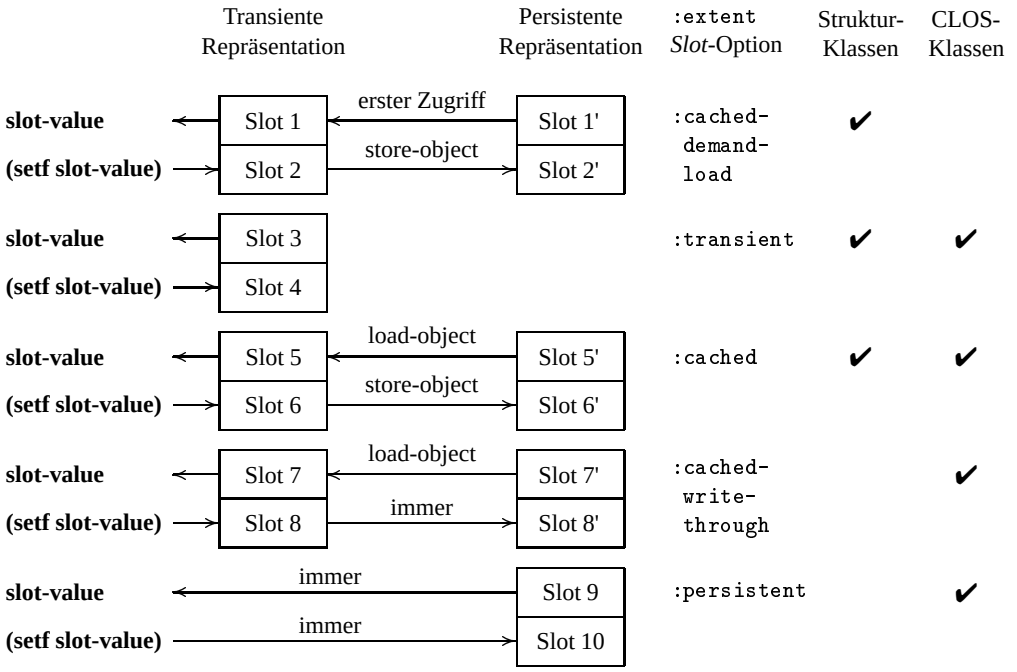


Abbildung 7.27: Lebensdauer von *Slot*-Zuständen

Abbildung 7.27 zeigt den Zusammenhang zwischen der Deklaration einer Lebensdauer für einen *Slot*-Zustand einer CLOS-Instanz (mittlere Spalte in Abbildung 7.27) und der für die jeweilige Lebensdauer allozierten Repräsentation des *Slot*-Zustands im transienten (2. Spalte) und persistenten (3. Spalte) Speicher, die Zeitpunkte, zu denen die Zustände zwischen der transienten und persistenten Repräsentation ausgetauscht werden, sowie die Verfügbarkeit der Deklaration für *Slots* von Struktur- (2. Spalte von rechts) und CLOS- (rechte Spalte) Klassen. Die Zustände der als `:cached-write-through` und `:persistent` deklarierten *Slots* werden durch spezialisierte Methoden der generischen Funktion **(setf slot-value-using-class)** (Protokoll 7.32, S. 125) gespeichert und stehen damit nur aktiv-persistenten Instanzen zur Verfügung.

Laden eines persistenten Objektes

Die Eigenschaften der Übertragung eines transienten Objektzustands in eine persistente Repräsentation gelten sinngemäß auch für das Laden eines persistenten Objektes.

Protokoll 7.28: Protokoll für das Laden eines transienten Objektes:
(load-object)

1 Es wird geprüft, ob die Stable Heap-Datei geöffnet ist:
(assert-sh-open-p)
2 Die Typkennung des durch die *short objid* referenzierten persistenten Objektes wird gelesen:
(p-type-tag-of)
3 Mit dem Aufruf der generischen Funktion
(p-objid-to-t-object)
wird die transiente Repräsentation eines persistenten Objektes geladen; beim Aufruf werden die *short objid* und die Typkennung des persistenten Objektes, die in Schritt 2 ermittelt wurde, mitgegeben:
3.1 Wenn aus der Typkennung hervorgeht, daß ein *Immediate* geladen wird (Tabelle 7.3, S. 89, Kennung $\neq 0$), wird als Ergebnis von Schritt 3 ein transienter *Immediate* mit dem in der *short objid* abgelegten Zustand zurückgegeben.
3.2 Für die Instanz einer Struktur- bzw. CLOS-Klasse wird das aktuelle transiente Klassen-Beschreibungsobjekt wie im Protokoll 7.21 **(ensure-class-description)** (S. 115) angegeben ermittelt:
(ensure-structure-description) bzw. **(ensure-class-description)**
3.3 Die transiente Repräsentation des persistenten Objektes wird ermittelt:
3.3.1 Das transiente Objekt wird im *Cache* gesucht:
(is-registered-objid)
Bei erfolgreicher Suche wird als Ergebnis von Schritt 3.3 die zum Objekt gehörige transiente Repräsentation zurückgegeben.
Der *Cache* stellt somit sicher, daß der Zustand eines persistenten Objektes immer in dieselbe transiente Repräsentation übertragen wird.
3.3.2 Wurde das Objekt nicht im *Cache* gefunden, wird klassenabhängig ein transientes Objekt erzeugt:
(make-<*Klassenname*>**)**
Für Instanzen von Struktur- und CLOS Klassen wird gegebenenfalls die für die Klasse deklarierte Konstruktor-Funktion aufgerufen. Das erzeugte transiente Objekt und seine *short objid* werden in den *Cache* eingetragen:
(register-to-cache)
Die transiente Repräsentation wird als Ergebnis von Schritt 3.3 zurückgegeben.
3.4 Sofern in Schritt 3.3 eine transiente Repräsentation im *Cache* gefunden wurde, bleibt der Zustand der transienten Repräsentation unverändert; das Protokoll verzweigt zu Schritt 3.9.
Dieses Kriterium sorgt damit für den Schnitt beim Laden eines Objektes.
3.5 Sofern noch keine Transaktion aktiv ist, wird eine Transaktion gestartet:
(with-transaction)
Eine hier gestartete Transaktion wird von rekursiven Aufrufen des Protokolls 7.28 **(load-object)** mitbenutzt.
3.6 Für das durch die *short objid* identifizierte persistente Objekt wird eine Lesesperre angefordert:

(with-read-lock)
Dabei wird festgestellt, ob auf das durch die *short objid* referenzierte persistente Objekt bereits eine Lesesperre innerhalb der aktiven Transaktion gesetzt wurde. Wenn ja, wurde der Zustand des persistenten Objektes in der aktiven Transaktion bereits in seine transiente Repräsentation übertragen; in diesem Fall verzweigt das Protokoll zu Schritt 3.8.

3.7 Der Zustand des persistenten Objektes wird in die transiente Repräsentation übertragen:
- 3.7.1 Für Instanzen von Basisklassen siehe das Subprotokoll 7.29 für das Laden einer transienten Instanz einer Basisklasse (S. 122).
- 3.7.2 Für Strukturobjekte siehe das Subprotokoll 7.30 **(p-structure)** (S. 122).
- 3.7.3 Für CLOS-Instanzen siehe das Subprotokoll 7.31 **(p-instance)** (S. 123).

3.8 Wenn in Schritt 3.5 eine Transaktion gestartet wurde, wird sie in diesem Schritt beendet. Damit werden alle innerhalb der Transaktion gesetzten Sperren (insbesondere die Lesesperre aus Schritt 3.6) aufgehoben.

3.9 Als Ergebnis wird die in Schritt 3.3 ermittelte transiente Repräsentation des persistenten Objektes zurückgegeben.

Durch die Verzweigung in Schritt 3.4 existiert im Moment keine Möglichkeit, den Zustand eines Objektes erneut zu laden; dies kann in einer zukünftigen Programmversion realisiert werden. Nötig wird das erneute Laden des Zustands von persistenten Objekten nach Änderung ihrer persistenten Repräsentation durch parallelen Zugriff; eine Änderung kann über die in Abschnitt 7.6.1 (S. 102) erläuterten Benachrichtigungssperren erkannt werden.

Protokoll 7.29: Subprotokoll für das Laden einer transienten Instanz einer Basisklasse:

1 Für Instanzen von Basisklassen, die nicht Werte sind, wird für jeden *Slot* des Objektes die *short objid* und die Typkennung des im entsprechenden *Slot* in der persistenten Repräsentation enthaltenen persistenten Objektes gelesen; mit diesen Parametern wird die generische Funktion
(p-objid-to-t-object)
aufgerufen, d.h. das Protokoll 7.28 **(load-object)** (S. 121) wird rekursiv mit Schritt 3 fortgeführt. Die aus dem rekursiven Aufruf erhaltene transiente Repräsentation des *Slot*-Zustands wird in den *Slot* der transienten Repräsentation eingetragen.

2 Für Werte wird ihr Zustand mit der Funktion
(p-values)
in ihrer binären Repräsentation aus dem Wertefeld des persistenten Objektes (Protokoll 7.4, S. 90) in die transiente Repräsentation kopiert.

Beim Laden von Symbolen wird der Zustand nur dann vom persistenten in das transiente Symbol kopiert, wenn das transiente Symbol durch den Ladevorgang selbst erzeugt wurde; damit wird eine eventuell von den BenutzerInnen unerwartete Änderung des Zustands des transienten Symbols verhindert.

Ein transientes Paket wird genau dann erzeugt, wenn ein persistentes Symbol geladen wird, das sich in einem persistenten Paket befindet, dessen Name kein transientes Paket identifiziert. Vorhandene transiente Pakete werden ‚mitbenutzt'; insbesondere wird für sie niemals die Funktion **make-package** [CLtLII, S. 262] aufgerufen, da die Semantik für ein bereits exisitierendes transientes Paket nicht spezifiziert ist.

Protokoll 7.30: Subprotokoll für das Laden eines transienten Strukturobjektes:
(p-structure)

1 Das referenzierte Klassen-Beschreibungsobjekt wird aus dem Feld mit der Strukturbeschreibung (Protokoll 7.6, S. 92) des Strukturobjektes wie im Protokoll 7.28 **(load-object)** (S. 121) angegeben geladen.
2 Wenn das in Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) erhaltene aktuelle transiente Klassen-Beschreibungsobjekt und das in Schritt 1 erhaltene referenzierte transiente Klassen-Beschreibungsobjekt nicht identisch sind, wird die persistente Repräsentation des Strukturobjektes in die durch das aktuelle transiente Klassen-Beschreibungsobjekt von Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) vorgegebene Repräsentation transformiert, sofern sich die Anzahl der *Slots* nicht erhöht hat:
   2.1 Die *Slot*-Zustände werden in der persistenten Repräsentation von ihrer bisherigen Position an die Position des gleichnamigen *Slots* der aktuellen transienten Strukturbeschreibung verschoben.
   2.2 Die Strukturbeschreibung des persistenten Strukturobjektes wird auf das in Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) ermittelte aktuelle transiente Klassen-Beschreibungsobjekt geändert, d.h. in das Feld mit der Strukturbeschreibung wird die *objid* der zum aktuellen transienten Klassen-Beschreibungsobjekt gehörigen persistenten Repräsentation eingetragen.
3 Wenn in Schritt 2.2 die Strukturbeschreibung des Strukturobjektes geändert wurde, wird die Liste der *Slot*-Beschreibungsobjekte aus dem in Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) erhaltenem aktuellem transientem Klassen-Beschreibungsobjekt gelesen; ansonsten wird die Liste der *Slot*-Beschreibungsobjekte aus dem in Schritt 1 erhaltenem referenziertem transientem Klassen-Beschreibungsobjekt gelesen:
   (class-effective-slots)
4 Für jeden *Slot*, dessen Zustands-Lebensdauer nicht als `:transient` deklariert wurde:
   4.1 Rufe für die im *Slot* enthaltene *short objid* die generische Funktion
   (p-objid-to-t-object)
   auf, d.h. das Protokoll 7.28 **(load-object)** (S. 121) wird mit Schritt 3 aufgerufen. Die daraus erhaltene transiente Repräsentation des *Slot*-Zustands wird in den *Slot* des transienten Strukturobjektes eingetragen.

Protokoll 7.31: Subprotokoll für das Laden einer transienten CLOS-Instanz:
(p-instance)

1 Das referenzierte Klassen-Beschreibungsobjekt wird aus dem Feld mit der Klassenbeschreibung (Protokoll 7.7, S. 93) der CLOS-Instanz wie im Protokoll 7.28 **(load-object)** (S. 121) angegeben geladen.
2 Wenn das in Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) erhaltene aktuelle transiente Klassen-Beschreibungsobjekt und das in Schritt 1 erhaltene referenzierte transiente Klassen-Beschreibungsobjekt nicht identisch sind:
   2.1 Der Zustandsvektor der persistenten CLOS-Instanz wird in die durch das aktuelle transiente Klassen-Beschreibungsobjekt von Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) vorgegebene Repräsentation transformiert. Für die Transformation werden die Zustände gleichnamiger *Slots* aus dem alten in einen neu allozierten persistenten Zustandsvektor kopiert.
   2.2 Der neu aufgebaute persistente Zustandsvektor wird in die persistente CLOS-Instanz eingetragen.
   2.3 Die Klassenbeschreibung der persistenten CLOS-Instanz wird auf das in Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) ermittelte aktuelle transiente Klassen-Be-

schreibungsobjekt geändert, d.h. in das Feld mit der Klassenbeschreibung wird die *objid* der zum aktuellen transienten Klassen-Beschreibungsobjekt gehörigen persistenten Repräsentation eingetragen.

Jetzt ist sichergestellt, daß das Klassen-Beschreibungsobjekt der CLOS-Instanz und die Klassendefinition im transienten System zueinander passen.

3 Aus dem in Schritt 3.2 von Protokoll 7.28 **(load-object)** (S. 121) erhaltenem aktuellem transientem Klassen-Beschreibungsobjekt wird die Liste der *Slot*-Beschreibungsobjekte gelesen:
(class-effective-slots)

4 Für jeden *Slot*, dessen Zustands-Lebensdauer als `:cached` deklariert wurde:

4.1 Aus der zum *Slot* gehörigen Komponente des durch das Referenzfeld referenzierten Zustandsvektors (Protokoll 7.7, S. 93) der persistenten CLOS-Instanz werden die *short objid* und die Typkennung des vom *Slot* referenzierten persistenten Objektes gelesen:
(p-index)

4.2 Referenziert die in Schritt 4.1 erhaltene *short objid* in Verbindung mit der Typkennung das Markierungsobjekt ‚ungebundener *Slot*', ist der persistente *Slot* ungebunden; dementsprechend wird die Bindung des *Slots* in der transienten Repräsentation aufgehoben:
(slot-makunbound)

4.3 Ist der persistente *Slot* gebunden, rufe mit der der in Schritt 4.1 enthaltenen *short objid* und Typkennung die generische Funktion
(p-objid-to-t-object)
auf, d.h. das Protokoll 7.28 **(load-object)** (S. 121) wird mit Schritt 3 aufgerufen. Die daraus erhaltene transiente Repräsentation des *Slot*-Zustands wird in den *Slot* der transienten CLOS-Instanz eingetragen.

Der Zustand eines nur im transienten Speicher repräsentierten *Slots* einer Instanz einer CLOS Klasse (Abbildung 7.27, S. 120, *Slot*-Lebensdauer `:transient`) wird, sofern die transiente CLOS Repräsentation im Verlauf des Ladens erzeugt wurde, auf den Zustand gesetzt, der sich aus seinem Initialisierungsausdruck ergibt. Für *Slots* von Strukturobjekten wird analog dazu der für den *Slot* in der Definition angegebene Initialisierungswert benutzt.

Schemaentwicklung

Schritt 2 von Subprotokoll 7.30 bzw. 7.31 beschreibt die momentan realisierte Schemaentwicklung für Struktur- bzw. CLOS-Objekte. Sie besteht darin, die persistente Repräsentation eines persistenten Objektes beim nächsten Zugriff nach einer Neuablage einer Klassenbeschreibung destruktiv auf die durch die aktuelle Klassenbeschreibung gegebene Repräsentationsform anzupassen (inkrementelle Konvertierung (*incremental conversion*) [Björnerstedt et al. 89, S. 454]) und die Referenz auf das Klassen-Beschreibungsobjekt entsprechend zu ändern (Abbildung 7.22, S. 117). Für persistente CLOS-Instanzen bleibt ihre Identität auch nach der Änderung immer erhalten; für persistente Strukturobjekte ist eine Schemaentwicklung zur Zeit nur dann möglich, wenn sich die Anzahl der *Slots* nicht erhöht hat.

Es ist geplant, in einer späteren Programmversion mehrere Arten für die Schemaentwicklung zur Verfügung zu stellen; welche Art benutzt werden soll, geht dann aus der für die Klasse deklarierten Art der Schemaentwicklung hervor. Um eine Schemaentwicklung auch für verschieden realisierte 3. Schichten zu erhalten, soll ihre Verarbeitung aus der 3. in die 2. Schicht verlegt werden.

Zugriff auf den Objektzustand

Passiv-persistente Objekte Auf passiv-persistente Instanzen kann entweder mit *Swizzling* oder über die direkte Manipulation der persistenten Repräsentation zugegriffen werden.

Swizzling Die transiente Repräsentation eines passiv-persistenten Objektes wird entweder explizit durch einen Aufruf einer entsprechenden Funktion (Protokoll 7.28 **(load-object)**, S. 121) geladen oder implizit erzeugt, wenn das passiv-persistente Objekt von einer zu ladenden Instanz referenziert wird. Auf die transiente Repräsentation kann mit den üblichen Common LISP Funktionen zugegriffen werden. Der Abgleich zwischen der transienten und persistenten Repräsentation eines passiv-persistenten Objektes erfolgt entweder explizit durch Aufruf einer entsprechenden Funktion (Protokoll 7.23 **(store-object)**, S. 117) oder implizit, sofern es von einer zu sichernden Instanz referenziert wird.

Direkte Manipulation der persistenten Repräsentation Zur direkten Manipulation der persistenten Repräsentationen gibt es entsprechende Funktionen; analog zu WOOD [St. Clair 93] wurde als Name einer direkt auf einer persistenten Repräsentation arbeitenden Funktion eine Konkatenation des Präfixes ‚p-' mit dem in [CLtLII] definierten Namen der auf einer transienten Repräsentation gleichen Typs arbeitenden Funktion gebildet; beispielsweise gibt die Funktion **p-car** das erste Element einer persistenten *cons*-Zelle zurück.

Auf die Zustände der *Slots* von persistenten Struktur- und CLOS-Objekten kann direkt mit den Funktionen **slot-...**[8] zugegriffen werden. Dazu wird an diese Funktionen anstatt einer transienten Repräsentation eines persistenten Struktur- oder CLOS-Objektes eine *short objid* übergeben; die spezialisierten Methoden greifen in diesem Fall immer direkt auf die persistente Repräsentation zu.

Aktiv-persistente Instanzen Für Instanzen von aktiv-persistenten Klassen erfolgt der Zugriff auf den Objektzustand über Methoden der generischen Funktionen **slot-...-using-class**, die auf die Klassen-Metaobjekt-Klasse **persistent-metaclass** spezialisiert sind.

Die spezialisierten Methoden der generischen Funktionen **(setf slot-value-using-class)** und **slot-makunbound-using-class** pflegen den für einen *Slot* deklarierten Index; in der Methode zu **(setf slot-value-using-class)** werden persistente CLOS-Instanzen zu ihrem *Slot*-Zustand assoziiert, in der Methode zu **slot-makunbound-using-class** wird eine bestehende Assoziation aufgehoben. Für eine spätere Programmversion ist geplant, diese Indexverwaltung in die 2. Schicht zu verschieben.

Protokoll 7.32: Protokoll für das Schreiben eines *Slot*-Zustands einer aktiv-persistenten CLOS-Instanz:
(setf slot-value-using-class)

1 Sofern noch keine Transaktion aktiv ist, wird eine Transaktion gestartet:
(with-transaction)
2 Ermittle die *short objid* der persistenten CLOS-Instanz:
(persistent-object-objid)
Damit wird der von der Superklasse **persistent-clos-object** geerbte *Slot* mit der *short objid* (Protokoll 7.19 **(defclass)**, Schritt 2.3, S. 112) der persistenten Repräsentation der CLOS-Instanz gelesen. Ist die *short objid* `nil`, wird die persistente Repräsentation des Objektes

[8] Im einzelnen sind dies die Funktionen **slot-value**, **(setf slot-value)**, **slot-boundp** und **slot-makunbound**.

alloziert:
(p-allocate-instance)

3 Das den *Slot* repräsentierende *Slot*-Beschreibungsobjekt wird gesucht:
(find-effective-slot-description)
Im Verlauf der Auswertung von **(find-effective-slot-description)** wird das Protokoll 7.21 **(ensure-class-description)** (S. 115) aufgerufen.

4 Wenn die Lebensdauer des *Slot*-Zustands als `:persistent` oder `:cached-write-through` (Protokoll 7.27, S. 120) deklariert wurde, speichere den neuen *Slot*-Zustand in der persistenten Repräsentation:

4.1 Wenn für den *Slot* ein Index deklariert wurde, wird zunächst der alte *Slot*-Zustand wie in Protokoll 7.34 **(slot-value-using-class)** (S. 127) angegeben gelesen:
(slot-value-using-class)

4.2 Aus der persistenten Repräsentation des Objektes wird die *short objid* des Zustandsvektors (Protokoll 7.7, S. 93) gelesen:
(get-instance-vector-and-location)

4.3 Der neue *Slot*-Zustand wird mit einen Aufruf der generische Funktion
(t-object-to-p-objid)
im persistenten Speicher abgelegt (Protokoll 7.23 **(store-object)**, Schritt 2, S. 117). Die erhaltene *short objid* und Typkennung wird in den Zustandsvektor an der durch die Positions-Information des *Slot*-Beschreibungsobjektes gegebenen Position eingetragen.

4.4 Wenn für den *Slot* ein Index deklariert wurde:

4.4.1 Prüfe, ob der unter Schritt 4.3 gespeicherte neue *Slot*-Zustand in der Indextabelle bereits zu einer persistenten CLOS-Instanz assoziiert ist:
(getindex-by-tag)
Wenn eine persistente Instanz gefunden wird und sie nicht identisch zu der an dieses Protokoll übergebenen Instanz ist, ist der neue *Slot*-Zustand bereits als Schlüssel auf eine andere persistente CLOS-Instanz vergeben; in diesem Fall wird ein Fehler signalisiert.

4.4.2 Entferne aus der Indextabelle die Assoziation vom alten in Schritt 4.1 ermittelten *Slot*-Zustand auf die persistente CLOS-Instanz:
(remindex-by-tag)

4.4.3 Assoziiere in der Indextabelle den neuen in Schritt 4.3 gespeicherten *Slot*-Zustand zur persistenten CLOS-Instanz:
(setf getindex-by-tag)

4.5 Wenn die Lebensdauer des *Slot*-Zustands als `:cached-write-through` deklariert wurde, speichere den neuen *Slot*-Zustand auch in der transienten Repräsentation:
(call-next-method)

5 sonst wird der neue *Slot*-Zustand mithilfe der Standard-Methode lediglich in die transiente Repräsentation eingetragen:
(call-next-method)

6 Wenn in Schritt 1 eine Transaktion gestartet wurde, wird sie in diesem Schritt beendet. Damit werden alle innerhalb der Transaktion gesetzten Sperren aufgehoben.

In der momentanen Programmversion sind keine doppelten Schlüssel (*duplicate keys*) für einen Index zugelassen; die Prüfung auf einen doppelten Schlüssel erfolgt in Schritt 4.4.1 und signalisiert gegebenenfalls einen Fehler. Für eine spätere Programmversion ist geplant, beim Auftreten von doppelten Schlüsseln die zum *Slot*-Zustand assoziierten persistenten CLOS-Instanzen in einer Liste abzulegen.

Protokoll 7.33: Protokoll für das Aufheben der Bindung eines *Slots* einer aktiv-persistenten CLOS-Instanz an einen Wert:
(slot-makunbound-using-class)

1 Sofern noch keine Transaktion aktiv ist, wird eine Transaktion gestartet:
(with-transaction)
2 Ermittle die *short objid* der persistenten CLOS-Instanz:
(persistent-object-objid)
3 Das den *Slot* repräsentierende *Slot*-Beschreibungsobjekt wird gesucht:
(find-effective-slot-description)
Im Verlauf der Auswertung von **(find-effective-slot-description)** wird das Protokoll 7.21 **(ensure-class-description)** (S. 115) aufgerufen.
4 Wenn die Lebensdauer des *Slot*-Zustands als `:persistent` oder `:cached-write-through` (Protokoll 7.27, S. 120) deklariert wurde, hebe die Bindung des *Slots* in der persistenten Repräsentation auf:
4.1 Wenn für den *Slot* ein Index deklariert wurde, wird zunächst der alte *Slot*-Zustand wie in Protokoll 7.34 **(slot-value-using-class)** (S. 127) angegeben gelesen:
(slot-value-using-class)
4.2 Aus der persistenten Repräsentation des Objektes wird die *short objid* des Zustandsvektors (Protokoll 7.7, S. 93) gelesen:
(get-instance-vector-and-location)
4.3 In den Zustandsvektor wird an der durch die Positions-Information des *Slot*-Beschreibungsobjektes gegebenen Position das Markierungsobjekt‚ungebundener *Slot*‘ eingetragen.
4.4 Wenn für den *Slot* ein Index deklariert wurde, wird aus der Indextabelle die Assoziation vom in Schritt 4.1 ermittelten *Slot*-Zustand auf die persistente CLOS-Instanz entfernt:
(remindex-by-tag)
4.5 Wenn die Lebensdauer des *Slot*-Zustands als `:cached-write-through` deklariert wurde, wird auch die Bindung des *Slots* in der transienten Repräsentation aufgehoben:
(call-next-method)
5 sonst wird lediglich die Bindung des *Slots* mithilfe der Standard-Methode in der transienten Repräsentation aufgehoben:
(call-next-method)
6 Wenn in Schritt 1 eine Transaktion gestartet wurde, wird sie in diesem Schritt beendet. Damit werden alle innerhalb der Transaktion gesetzten Sperren aufgehoben.

Protokoll 7.34: Protokoll für das Lesen eines *Slot*-Zustands einer aktiv-persistenten CLOS-Instanz:
(slot-value-using-class)

1 Sofern noch keine Transaktion aktiv ist, wird eine Transaktion gestartet:
(with-transaction)
2 Ermittle die *short objid* der persistenten CLOS-Instanz:
(persistent-object-objid)
3 Das den *Slot* repräsentierende *Slot*-Beschreibungsobjekt wird gesucht:
(find-effective-slot-description)
Im Verlauf der Auswertung von **(find-effective-slot-description)** wird das Protokoll 7.21 **(ensure-class-description)** (S. 115) aufgerufen.

4 Wenn die Lebensdauer des *Slot*-Zustands als `:persistent` deklariert wurde, lade den *Slot*-Zustand aus der persistenten Repräsentation:
   4.1 Aus der persistenten Repräsentation des Objektes wird die *short objid* des Zustandsvektors gelesen:
   (get-instance-vector-and-location)
   4.2 Der *Slot*-Zustand befindet sich im Zustandsvektor an der durch die Positions-Information des *Slot*-Beschreibungsobjektes gegebenen Position; er wird mit einem Aufruf der generische Funktion
   (p-objid-to-t-object)
   mit der *short objid* und der Typkennung des vom *Slot* referenzierten persistenten Objektes geladen (Protokoll 7.28 **(load-object)**, Schritt 3, S. 121).
5 sonst wird der *Slot*-Zustand mithilfe der Standard-Methode lediglich aus der transienten Repräsentation geladen:
   (call-next-method)
6 Wenn in Schritt 1 eine Transaktion gestartet wurde, wird sie in diesem Schritt beendet. Damit werden alle innerhalb der Transaktion gesetzten Sperren aufgehoben.
7 Als Ergebnis des Protokolls 7.34 **(slot-value-using-class)** wird das Ergebnis von Schritt 4.2 bzw. 5 zurückgegeben.

Protokoll 7.35: Protokoll für die Prüfung, ob ein *Slot* einer aktiv-persistenten CLOS-Instanz an einen Wert gebunden ist:
(slot-boundp-using-class)

1 Sofern noch keine Transaktion aktiv ist, wird eine Transaktion gestartet:
   (with-transaction)
2 Ermittle die *short objid* der persistenten CLOS-Instanz:
   (persistent-object-objid)
3 Das den *Slot* repräsentierende *Slot*-Beschreibungsobjekt wird gesucht:
   (find-effective-slot-description)
   Im Verlauf der Auswertung von **(find-effective-slot-description)** wird das Protokoll 7.21 **(ensure-class-description)** (S. 115) aufgerufen.
4 Wenn die Lebensdauer des *Slot*-Zustands als `:persistent` deklariert wurde, prüfe die Bindung des *Slots* in der persistenten Repräsentation:
   4.1 Aus der persistenten Repräsentation des Objektes wird die *short objid* des Zustandsvektors gelesen:
   (get-instance-vector-and-location)
   4.2 Im Zustandsvektor wird an der durch die Positions-Information des *Slot*-Beschreibungsobjektes gegebenen Position geprüft, ob sich dort nicht das Markierungsobjekt ‚ungebundener *Slot*' befindet.
5 sonst wird lediglich die Bindung des *Slots* mithilfe der Standard-Methode in der transienten Repräsentation geprüft:
   (call-next-method)
6 Wenn in Schritt 1 eine Transaktion gestartet wurde, wird sie in diesem Schritt beendet. Damit werden alle innerhalb der Transaktion gesetzten Sperren aufgehoben.
7 Als Ergebnis des Protokolls 7.35 **(slot-boundp-using-class)** wird das Ergebnis von Schritt 4.2 bzw. 5 zurückgegeben.

7.6.6 Lokalisierung

Binden von Objekten an persistente Symbole

Ein Objekt ist genau dann persistent, wenn es erreichbar ist, d.h. wenn es transitiv vom Wurzelobjekt referenziert wird. In WOOD wird beispielsweise deswegen die Möglichkeit angeboten, das Wurzelobjekt explizit zu setzen [St. Clair 93]; seine Struktur muß von den BenutzerInnen so angelegt und verwaltet werden, daß alle persistenten Objekte über dieses Wurzelobjekt erreichbar sind.

In PLOB! wurde ein anderer, LISP-ähnlicher Weg gewählt. In LISP können Symbole über ihren Namen referenziert werden [CLtLII, S. 13, 27]; an ein Symbol kann ein Objekt gebunden werden. Analog dazu können hier persistente Symbole benutzt werden, an die ein persistentes Objekt gebunden werden kann. Persistente Symbole können ebenfalls über ihren Namen lokalisiert werden; für die Suche kann als Name entweder eine transiente oder persistente Zeichenkette oder ein transientes Symbol angegeben werden. Bei Angabe eines transienten Symbols wird ein persistentes Symbol mit gleichem Namen und Paket gesucht. Für jedes persistente Symbol ist garantiert, daß es erreichbar ist; damit ist auch ein an ein persistentes Symbol gebundenes Objekt erreichbar.

Indizes über die Zustände eines *Slots*

Für den *Slot* einer aktiv-persistenten Klasse mit deklariertem Index kann nach den persistenten CLOS-Instanzen gesucht werden, die in diesem *Slot* einen bestimmten Zustand enthalten; Abfragen über einen Zustandsbereich sind ebenfalls möglich. Für die Suche werden folgende Parameter angegeben:

- Der Name der aktiv-persistenten Klasse.
- Der Name des *Slots*, über den gesucht werden soll. Für diesen *Slot* muß ein Index deklariert worden sein.
- Der Zustand oder Zustandsbereich, nach dem gesucht werden soll.

Als Ergebnis wird eine Liste der passenden Instanzen der Klasse oder Superklassen zurückgegeben.

Durch Erstellen entsprechender Methoden können relativ einfach zusätzliche Indexarten eingebunden werden; genaueres dazu findet sich im Referenzhandbuch [Kirschke 94] im Abschnitt ‚index . . . ‘.

7.6.7 Standardisierung

In PLOB! wird eine orthogonale Sichtweise für Persistenz realisiert; dies bedeutet, daß jedes transiente Objekt persistent werden kann. Ein transientes Objekt wird persistent, indem sein Zustand bestehend aus Referenzen und Werten vollständig in ein persistentes Objekt übertragen wird. Daraus ergibt sich die Notwendigkeit, über das Verhalten jedes transienten Objektes auf seinen Gesamtzustand zugreifen können zu müssen. Das Verhalten der Instanzen von Common LISP Typen wird in [CLtLII] weitestgehend normiert.

Für fehlende Normierungen gibt es prinzipiell die zwei Lösungen, sie entweder hinzunehmen oder eigene Normierungen festzulegen. Die erste Lösung schränkt die Transparenz oder Orthogonalität von Persistenz ein; deswegen wurde hier die zweite Lösung vorgezogen. Im folgen-

den Text werden die Arten der fehlenden Normierungen sowie die sich bietenden Möglichkeiten, doch noch zu einer Normierung zu kommen, geschildert.

‚Vergessen' einer Norm

In diese Kategorie fallen Normierungen, die offensichtlich wären, aber trotzdem nicht erfolgten, obwohl der Normierungsprozeß abgeschlossen ist.

Ein Beispiel dafür sind die fehlenden Feld-Informationsfunktionen (*array information functions*) [CLtLII, S. 448–452] im Zusammenhang mit Feldern, die Elemente anderer Felder referenzieren (*displaced array*s) [CLtLII, S. 444]; für diese Felder gibt es keine Informationsfunktionen, die darüber Auskunft geben, zu welchem Feld die referenzierten Elemente gehören und ab welchem Index (*displaced index offset*) [CLtLII, S. 445] der gemeinsam referenzierte Teil beginnt.

Fehlende Norm

Fehlende Normen resultieren aus einem noch nicht begonnenem Normierungsprozeß.

Ein Beispiel ist die fehlende Normierung des Verhaltens von Strukturklassen. In [CLtLII, S. 800] wird nur festgelegt, daß Strukturklassen repräsentiert werden; eine Festlegung ihres Verhaltens erfolgte nicht. In [AMOP] werden Strukturklassen überhaupt nicht erwähnt.

Eine Normierung für LISP-interne Prozesse erfolgte ebenfalls nicht; viele LISP-Systeme, darunter auch LISPWORKS Common LISP und ALLEGRO Common LISP, halten sich aber an den durch die LISP Implementation auf den Maschinen der Firma SYMBOLICS gegebenen Standard.

Lösung Das Problem der ‚vergessenen' und fehlenden Normen stellte sich bereits den AutorInnen der verschiedenen LISP-Systeme, die dafür meist untereinander äquivalente vernünftige Festlegungen trafen, die als Norm im Rahmen dieser Arbeit verwendet werden konnten. Es wurde Wert darauf gelegt, die entsprechenden, meist systemabhängigen Konstrukte nicht direkt zu nutzen, sondern eine dokumentierte Schnittstelle zu schaffen.

Unvollständige oder unklare Norm

Unvollständige oder unklare Normen resultieren aus einem noch nicht abgeschlossenem Normierungsprozeß.

In [CLtLII] wurde beispielsweise keine Normierung des MOP vogenommen; statt dessen wurde der Hinweis gegeben, daß die dem ANSI Kommitee im Juni 1988 vorgeschlagene Standardisierung des MOP nicht angenommen wurde [CLtLII, S. 770]. Die AutorInnen des zur Realisierung von PLOB! verwendeten LISPWORKS Common LISP hatten also zu diesem Zeitpunkt keine verbindliche Norm zur Implementation des MOP; letztendlich erfolgte sie doch nach der nicht angenommenen Standardisierung vom Juni 1988 [Snape 94]. Im Jahr 1991 erschien dann [AMOP], das inzwischen als de-fakto Normierung des MOP angesehen wird; es unterscheidet sich aber von dem 1988 nicht angenommenen Standard in einigen Punkten, die für die Realisierung von transparenter Persistenz für aktiv-persistente CLOS-Objekte nicht unerhebliche Schwierigkeiten bereiteten.

Eine Unklarheit findet sich in [AMOP] bezüglich des Protokolls zum Erzeugen des effektiven Funktionscodes der Methoden-Metaobjekte der generischen Zugriffs-Funktionen, deren Namen in einer Klassendefinition in den Argumenten der *Slot*-Optionen `:reader`, `:writer`

oder `:accessor` angegeben wurden. Der effektive Funktionscode aller Methoden-Metaobjekte wird nach [AMOP, S. 207–209] über einen Aufruf der generischen Funktion **make-method-lambda** erzeugt, die u.a. den (von den BenutzerInnen erstellten) direkten Code der Methode als Argument erhält. Das Problem im Zusammenhang mit generischen Zugriffs-Funktionen besteht darin, daß kein expliziter direkter Code für ihre Methoden spezifiziert wird, sondern vom LISP-System selbst generiert wird. Diese Tatsache wird von LISPWORKS so interpretiert, daß die generische Funktion **make-method-lambda** nur für Methoden-Metaobjekte aufgerufen wird, die aus explizit erstellten Methoden erzeugt wurden und nicht für die implizit erzeugten Methoden-Metaobjekte der Methoden der generischen Zugriffs-Funktionen (Protokoll 7.19 **(defclass)**, Schritte 2.3.11–2.3.13, S. 113).

Lösung Die unklaren oder unvollständigen Normen resultieren in einem unerwartetem Verhalten des LISP-Systems; eine gründliche Lösung dieses Problems besteht in der Beseitigung der Unklarheiten bzw. in der Ergänzung der Normierung und einer Korrektur des LISP Systems. Dies war im Rahmen dieser Arbeit nicht möglich.

Ich vertrete die Meinung, daß, sofern der Produzent eines Systems angibt, einem gewissem Standard zu genügen, der Produzent auch dafür zuständig ist, daß das System den Standard auch tatsächlich einhält. Deswegen wurde hier darauf verzichtet, für den Ausgleich der Inkompatibilitäten der LISP-Systeme zum gegebenen Standard zusätzlich einen‚Meta‘-Standard zu definieren; statt dessen wurden die Inkompatibilitäten durch dokumentierte ad-hoc-Lösungen ausgeglichen.

Keine Festlegung von Strukturen

Die Standardisierung in [CLtLII] legt keine Strukturen für Instanzen der dort definierten Typen und Klassen fest; damit kann diese Festlegung von den AutorInnen eines LISP-Systems vorgenommen werden und ermöglicht so beispielsweise Optimierungen. Statt dessen wurde ihr Verhalten so normiert, daß darüber auf den Gesamtzustand einer Instanz als Summe ihrer Teilzustände zugegriffen werden kann. Bei der Realisation zeigte sich, daß diese‚Aufsummierung‘ mit erheblichen Effizienzverlusten verbunden sein kann und also unter Umständen vermieden werden sollte; es ist dann günstiger, auf den Gesamtzustand zugreifen zu können. Dies bedingt aber die Verfügbarkeit von Informationen über die Struktur dieser Instanzen.

Beispiele dafür sind Zeichenketten und Bit-Vektoren, auf die lediglich elementweise zugegriffen werden kann; die Übertragung eines Zustands wäre in diesem Fall sehr ineffizient.

Lösung Die in den verschiedenen LISP-Systemen verwendeten Strukturen sind oft äquivalent; daher wurden ihre internen Zugriffsfunktionen als Norm im Rahmen dieser Arbeit verwendet. Es wurde Wert darauf gelegt, die entsprechenden systemabhängigen Zugriffsfunktionen nicht direkt aufzurufen, sondern eine dokumentierte Schnittstelle zu schaffen.

System-interne Konstrukte

Der Common LISP Standard verbietet es einem LISP-System selbstverständlich nicht, zusätzliche Konstrukte zu benutzen oder anzubieten, solange sie nicht inkompatibel zum Standard sind. Dies können beispielsweise systemintern definierte Typen sein, deren Instanzen zur Repräsentation von Instanzen von Standard-Typen benutzt werden.

Lösung Die systeminternen Konstrukte wurden entweder als Basiskonstrukte zu PLOB! hinzugefügt oder durch andere Basiskonstrukte nachgebildet.

7.6.8 Zusammenfassung der 3. Schicht

Für die Einhaltung der ACID Kriterien (S. 84) sorgt die 2. Schicht; die 3. Schicht benutzt lediglich die dort realisierten Funktionalitäten und bietet höheren Schichten einfache Einbindungsmöglichkeiten für sie an. Der folgende Abschnitt faßt noch einmal kurz die Funktionalitäten der 3. Schicht aus der Sicht der nächsthöheren 4. Schicht zusammen.

Abbildung zwischen transienten und persistenten Objekten LISP Objekte werden auf persistente Objekte mit einer der transienten Klasse angemessenen Beschreibung abgebildet.

Orthoghonalität; aktiv-persistente Klassen Jedes transiente LISP Objekt kann persistent werden; Instanzen von aktiv-persistenten CLOS Klassen haben bezüglich Persistenz mehr Möglichkeiten als passiv-persistente Instanzen.

Zugriff auf den Zustand von persistenten Objekten Für den Zugriff auf persistente Objekte gibt es die Möglichkeit, die persistente Repräsentation eines persistenten Objektes auf eine transiente Repräsentation abzubilden und auf diese zuzugreifen (*Swizzling*). Für aktiv-persistente Objekte kann auch transparent direkt auf die persistente Repräsentation zugegriffen werden, für alle anderen Objekte gibt es die Möglichkeit des intransparenten direkten Zugriffs.

Wurzelobjekt; persistente Symbole Das Wurzelobjekt wird lediglich intern in der 3. Schicht verwendet; damit ein persistentes Objekt vom Wurzelobjekt aus erreichbar wird, genügt es, wenn es als Wert an ein persistentes Symbol gebunden wird. Persistente Symbole und damit auch der jeweils an sie gebundende Wert können über Namen lokalisiert werden.

Assoziativer Zugriff Für *Slots* von aktiv-persistenten Instanzen kann ein Index deklariert werden, der persistente CLOS-Instanzen zu *Slot*-Zuständen assoziiert. Die Verwaltung der Indextabellen erfolgt transparent in spezialisierten Methoden der generischen Funktionen **slot-. . . -using-class**.

Die 3. Schicht wird ausführlich im Referenz-Handbuch [Kirschke 94] dokumentiert.

7.7 Schicht 4: Die Sitzungs-Schicht

Dieser Abschnitt erläutert die zur Verfügung gestellten Konzepte und zeigt Beispiele für ihre Benutzung.

7.7.1 Konzepte

Persistente Pakete

Persistente Pakete dienen zur Gruppierung von persistenten Symbolen. Alle persistenten Pakete sind erreichbar und damit immer persistent. Zur Manipulation von persistenten Paketen gibt es Funktionen mit gleicher Funktionalität wie für transiente Pakete; die entsprechenden Funktionsnamen stammen aus [CLtLII] und wurden zur Unterscheidung zu den auf transienten Paketen

arbeitenden Funktionen mit dem Präfix ‚p-' versehen; im einzelnen sind dies die Funktionen **p-find-package**, **p-delete-package**[9], **p-package-name** und **p-package-internals**.

Persistente Symbole

Persistente Symbole sind mit Namen versehene persistente Objekte. Sie gehören zu genau einem persistenten Paket und sind von diesem erreichbar; damit sind sie persistent. Ein an ein persistentes Symbol gebundenes Objekt ist, da es von einem persistenten Symbol referenziert wird, ebenfalls persistent und erreichbar. Aus der Sicht der BenutzerInnen gibt es kein explizites Wurzelobjekt; diese Aufgabe haben in PLOB! persistente Pakete und speziell die von ihnen erreichbaren persistenten Symbole übernommen. Dies entspricht vom Konzept her der Erreichbarkeit von transienten Symbolen in Common LISP über ihr transientes Paket.

Zur Manipulation von persistenten Symbolen gibt es Funktionen mit gleicher Funktionalität wie für transiente Symbole. Die entsprechenden Funktionsnamen stammen aus [CLtLII] und wurden zur Unterscheidung zu den auf transienten Symbolen arbeitenden Funktionen mit dem Präfix ‚p-' versehen; im einzelnen sind dies die Funktionen **p-boundp**, **p-fboundp**, **p-symbol-function**, **(setf p-symbol-function)**, **p-symbol-package**, **p-symbol-plist**, **(setf p-symbol-plist)**, **p-symbol-name**, **p-symbol-value**, **(setf p-symbol-value)**, **p-fmakunbound** und **p-makunbound**. Bei den Manipulationsfunktionen für persistente Symbole kann zur Referenzierung des persistenten Symbols das transiente Symbol angegeben werden; die Manipulationsfunktionen suchen dann das persistente Symbol mit gleichem Namen und Paket. Der Wert des persistenten Symbols `'foo` kann beispielsweise mit dem Ausdruck `(setf (p-symbol-value 'foo) 5)` auf den Wert 5 gesetzt werden; das transiente Symbol `'foo` dient zur Lokalisierung des persistenten Symbols `'foo`. Das Setzen des Wertes eines persistenten Symbols auf ein transientes Objektes impliziert die Speicherung des transienten Objektes.

Eine der häufigsten Operationen mit Symbolen ist das Setzen bzw. Abfragen des an das Symbol gebundenen Wertes; dazu gibt es das *macro-reader*-Zeichen

```
#l<symbol-name>
```

das zu

```
(p-symbol-value <symbol>)
```

evaluiert; der Buchstabe `l` ist eine Abkürzung für‚load'.

Zur Verwaltung von persistenten Symbolen gibt es die Funktionen **p-find-symbol**, **p-intern** und **p-unintern** mit gleicher Funktionalität wie in [CLtLII].

Deklarationen

Über Deklarationen kann die Lebensdauer und Repräsentationsform von *Slot*-Zuständen, die Art der Schemaentwicklung für Klassen und die für die Erzeugung der transienten Repräsentationen von persistenten Objekten einer Klasse zu benutzende Konstruktor-Funktion festgelegt werden. Die Möglichkeiten der Beeinflussung von Persistenz über Deklarationen besteht für alle Klassen, unabhängig davon, ob sie aktiv- oder passiv-persistent sind; lediglich die Art und Weise, in der die Deklarationen angegeben werden können, ist unterschiedlich.

[9] Da persistente Pakete implizit erreichbar sind, gibt es für sie einen expliziten Destruktor.

Für aktiv-persistente Klassen werden die Deklarationen als zusätzliche Klassen- und *Slot*-Optionen in die Klassendefinition eingefügt. Ihre Auswertung erfolgt im Laufe der Initialisierung des die Klasse repräsentierenden Klassen-Metaobjektes. Aus dem Klassen-Metaobjekt und den davon referenzierten Metaobjekten werden die Informationen in das persistente Klassen-Beschreibungsobjekt und die davon referenzierten Beschreibungsobjekte der Klasse übernommen.

Bei passiv-persistenten Klassen werden die Deklarationen über Aufrufe von Funktionen an PLOB! weitergegeben; Deklarationen auf Klassenebene werden an das Klassen-Metaobjekt der Klasse und Deklarationen auf *Slot*-Ebene werden an das Klassen-Metaobjekt und den Namen des *Slots* assoziiert.

Deklaration der Lebensdauer von *Slot*-Zuständen Die Deklaration der Lebensdauer von *Slot*-Zuständen beeinflußt sowohl die Lebensdauer als auch die Repräsentationsform von *Slot*-Zuständen (Abbildung 7.27, S. 120). Die einzelnen Deklarationswerte bedeuten:

Lebensdauer `:cached-demand-load` Diese Lebensdauer gibt es nur für *Slots* von Strukturobjekten. Der transiente *Slot*-Zustand wird bei der ersten lesenden Dereferenzierung des *Slots* geladen, es sei denn, der persistente *Slot*-Zustand referenziert ein *Immediate* oder ein bereits geladenes Objekt; in diesem Fall wird der transiente *Slot*-Zustand bereits beim Laden des Strukturobjektes gesetzt.

Da Strukturobjekte passiv-persistente Instanzen sind, erfolgt das Speichern des transienten *Slot*-Zustands beim Speichern des transienten Strukturobjektes.

Lebensdauer `:transient` Mit der Deklaration einer transienten Lebensdauer wird der *Slot* nicht im persistenten Speicher repräsentiert; demzufolge wird der Zustand eines als transient deklarierten *Slots* beim Speichern des transienten Objektes auch nicht weiter traversiert.

Lebensdauer `:cached` Der transiente *Slot*-Zustand wird nur beim Laden bzw. Speichern des transienten Objektes übertragen.

Lebensdauer `:cached-write-through` Diese Lebensdauer gibt es nur für *Slots* von aktiv-persistenten Instanzen. Der Zustand des transienten *Slots* wird beim Laden auf den Zustand des persistenten *Slots* gesetzt; Schreibzugriffe werden in den *Slot* sowohl der transienten als auch der persistenten Repräsentation propagiert.

Lebensdauer `:persistent` Diese Lebensdauer gibt es nur für *Slots* von aktiv-persistenten Instanzen. Der *Slot* wird nur im persistenten Speicher repräsentiert; alle Zugriffe finden direkt auf die persistente Repräsentation statt.

Eine Lebensdauer kann auf einer der Ebenen ‚*Slot*' – ‚Klasse' – ‚Paket des Klassen-Namens' deklariert werden; die Lebensdauer des *Slot*-Zustands ergibt sich aus der ersten Deklaration, die in der angebenen Reihenfolge für eine der Ebenen gefunden wird.

Für aktiv-persistente Klassen erfolgt die Einbindung der oben genannten Deklarationswerte auf *Slot*- und Klassen-Ebene mittels der zusätzlichen *Slot*- bzw. Klassen-Option

`:extent` <*Lebensdauer*>

Bei passiv-persistenten Klassen erfolgt die Einbindung der Deklarationswerte auf *Slot*-Ebene über die Evaluierung des Ausdrucks

```
(setf (slot-extent <Slot-Name> (find-class <Klassenname>))
      <Lebensdauer>)
```

und auf Klassenebene über die Evaluierung des Ausdrucks

```
(setf (class-extent (find-class <Klassenname>)) <Lebensdauer>)
```

Auf Paketebene erfolgt die Einbindung der Deklaration einer Lebensdauer über die Evaluierung des Ausdrucks

```
(setf (package-extent (find-package <Paketname>)) <Lebensdauer>)
```

Damit können alle Klassen, die innerhalb eines Paketes definiert werden, mit einer Lebensdauer versehen werden; im allgemeinen wird die Lebensdauer auf Paketebene für Pakete mit den Namen von ‚uninteressanten Klassen' auf transient gesetzt.

Sitzungen und Transaktionen

Eine Sitzung wird implizit von PLOB! bei Bedarf für den laufenden LISPWORKS *lightweight*-Prozeß geöffnet; durch Aufruf der Funktion

```
(open-session)
```

kann sie auch explizit geöffnet werden. Die Funktion

```
(show-sessions)
```

zeigt alle momentan offenen Sitzungen an.

Das Makro **with-transaction** bettet die <*Anweisungen*> in eine an die Sitzung gebundene Transaktion ein:

```
(with-transaction { <Anweisungen> }*)
```

Werden die <*Anweisungen*> vollständig abgearbeitet, wird die Transaktion beendet (*committed*); andernfalls wird sie abgebrochen (*aborted*). Eine nicht-vollständige Abarbeitung ist durch jeden nicht-lokalen Sprung aus den <*Anweisungen*> heraus in den umfassenden Block des Makros gegeben; dies tritt insbesondere bei folgenden Ereignissen auf:

Laufzeitfehler Bei Fehlern, die nicht innerhalb der <*Anweisungen*> behandelt werden, ist eine nicht-vollständige Abarbeitung gegeben.

`block`/`return`-**Konstrukte** Bei `block`/`return`-Konstrukten [CLtLII, S. 161] ist zu beachten, daß bei Einbettung des Makros **with-transaction** in einen Block die Ausführung der Anweisung `return` bzw. `return-from` eine nicht vollständige Abarbeitung der <*Anweisungen*> zur Folge hat.

`catch`/`throw`-**Konstrukte** Die Konstrukte `catch` und `throw` [CLtLII, S. 187] mit der Anweisung `catch` im umfassenden Block des Makros **with-transaction** und der Anweisung `throw` innerhalb der <*Anweisungen*> führen bei der Ausführung der Anweisung `throw` zu einer nicht vollständigen Abarbeitung der <*Anweisungen*>.

Beim Auftreten eines der oben genannten Ereignisse wird eine Transaktion demzufolge abgebrochen.

Assoziativer Zugriff

Für den *Slot* einer aktiv-persistenten Instanz kann ein Index definiert werden; die Art des Indexes wird über die zusätzliche *Slot*-Option `:index` deklariert:

`:index` <*index-definierender S-Ausdruck*>

Der <*index-definierende S-Ausdruck*> definiert den anzulegenden Index; momentan stehen persistente B-Bäume (Abschnitt 7.5.7, S. 98) als Repräsentation für Indextabellen zur Verfügung. Die *Slot*-Option für Verwendung eines B-Baumes als Indextabelle lautet:

`:index (btree :test` <*Ordnungskriterium*> `:cached` <*cached*>`)`

Die möglichen <*Ordnungskriterien*> werden in Abschnitt 7.5.7 (S. 98) erläutert. Wenn für die Angabe <*cached*> nicht-`nil` angegeben wird, finden alle Einzelanfragen (d.h. alle Anfragen, die nicht über einen Schlüsselbereich gehen) auf den B-Baum über einen an ihn gebundenen transienten *Cache* statt; damit wird der lesende Zugriff für einzelne Schlüsselwerte beschleunigt, aber gleichzeitig der Speicherbedarf erhöht.

Die Deklaration eines Indexes hat zwei Seiteneffekte. Der erste Seiteneffekt besteht darin, daß durch die Deklaration eines Indexes mit dem Ordnungskriterium `equal` der *Slot* auf den Typ des ersten *Slot*-Zustands festgelegt wird, der in den B-Baum eingefügt wird. Der Grund liegt darin, daß die Schlüsselobjekte in einem B-Baum mit dem Ordnungskriterium `equal` alle den gleichen Typ haben müssen, damit sie sortiert abgelegt werden können.

Der zweite Seiteneffekt ist, daß alle Objekte der Klassen, bei denen für einen *Slot* ein Index definiert wurde, erreichbar sind und daher bei einer Speicherrückgewinnung nicht gelöscht werden (Abbildung 5.4, S. 64). Damit ein Objekt bei einer Speicherrückgewinnung gelöscht werden kann, darf keine Referenz aus einer Indextabelle auf das Objekt existieren. Erreicht wird dies durch Entfernen des Objektes aus den Indextabellen, indem die Bindung jedes *Slots* des Objektes, für den ein Index deklariert wurde, aufgehoben wird.

Objekte können mit der Funktion

`(p-select` <*Klassenname*> `:where` <*Slot-Name*> <*Auswahlkriterien*>`)`

nach einem *Slot*-Zustand ausgewählt werden. Für den durch <*Slot-Namen*> identifizierten *Slot* in der durch <*Klassenname*> benannten Klasse muß ein Index deklariert worden sein. Für Einzelheiten der <*Auswahlkriterien*> verweise ich auf das Referenzhandbuch [Kirschke 94], Abschnitt ‚p-select'.

Löschen von Objekten und Speicherrückgewinnung

Der Speicherplatz eines Objektes kann dann und nur dann freigegeben werden, wenn keine Referenzen mehr auf das Objekt existieren; ein Objekt wird durch nicht-referenzieren als zu löschendes Objekt für die nächste Speicherrückgewinnung ‚markiert'. Eine Speicherrückgewinnung wird durch den Aufruf der Funktion

`(plob-gc)`

veranlaßt; vor einer Speicherrückgewinnung sollte der POSTORE durch einen Aufruf der Funktion

`(plob-close)`

geschlossen werden, da sich durch die in der letzten Phase der Speicherrückgewinnung durchgeführten Kompaktierung des POSTORE die *objids* der persistenten Objekte ändern können und damit bei geöffnetem POSTORE Inkonsistenzen mit den transienten Repräsentationen der persistenten Objekte auftreten können.

Der Zustand eines Objektes kann explizit mit der Funktion

`(p-destroy` <*short objid*>`)`

gelöscht werden; nach dem Aufruf besteht der Zustand nur noch aus *Immediates*, die keine weiteren persistenten Objekte referenzieren. Bei diesem Aufruf wird auch die Klasse des Objektes auf die Klasse **Zombie** geändert; eine nächste Dereferenzierung des persistenten Objektes führt im allgemeinen zu einer Fehlermeldung.

Beim Abbruch einer Transaktion erhalten alle im Verlauf der Transaktion erzeugten Zombies ihre vorherige Klasse und ihren vorherigen Zustand zurück.

7.7.2 Beispiele

Dieser Abschnitt zeigt anhand von Beispielen die realisierten Konzepte von PLOB!.

Anzeigen von Klassen-Beschreibungsobjekten

In Abbildung 7.36 werden durch Aufruf der Funktion **p-apropos-classes** alle Klassen-Beschreibungsobjekte angezeigt, deren Namen die an diese Funktion übergebene Zeichenkette `"description"` enthalten.

```
CL-USER 1 > (p-apropos-classes "description")
① Error: The stable heap is not open.
     1 (continue) Open it.
     2 (abort) Return to level 0.
Type :c followed by a number to proceed
CL-USER 2 : 1 > :c
; Bootstrap, phase 1: Loaded structure description
;                     plob::structure-description, 10/12 slots.
; Bootstrap, phase 1: Loaded structure description
;                     plob::structure-slot-description, 8/10 slots.
usw. bis
; LISP root formatted at 18.07.1994 14:15
plob::method-description, data: #<instance class-description
②    method-description 1.00 slots=5/6 short-objid=8388579>
plob::effective-slot-description, data: #<instance class-description
      effective-slot-description 1.00 slots=9/10 short-objid=8388581>
usw. bis
plob::structure-description, data: #<structure structure-description
      structure-description 1.00 slots=10/12 short-objid=8388602>
```

Abbildung 7.36: Öffnen des Stable Heap und Anzeigen der Klassen-Beschreibungsobjekte

Da der Stable Heap vor dem Aufruf geschlossen war, wird er nach Rückfrage bei den BenutzerInnen geöffnet ①. Bei der Form der Ausgabe der Funktion **p-apropos-classes** (ab ②) habe ich mich nach der in LISPWORKS realisierten Funktion **apropos** [CLtLII, S. 701] gerichtet. Vor der Angabe `data:` befindet sich der Name der Klasse, dahinter die in der 2. Schicht

generierte Druckrepräsentation der Klassen-Beschreibungsobjekte; sie enthält eine Klasseninformation, den Klassennamen, die Versionsnummer, die Anzahl der im persistenten Speicher repräsentierten sowie der gesamten *Slots* und die *short objid* des persistenten Klassen-Beschreibungsobjektes.

Definition einer aktiv-persistenten Klasse

Die Definition der aktiv-persistenten Klasse **person** zeigt Abbildung 7.37; aktiv-persistent wird die Klasse durch die Klassenoption `(:metaclass persistent-metaclass)` ②.

```
CL-USER 3 > (defclass person ()
              ((name :initarg :name :accessor person-name
①                    :index (btree :test equal))
               (age :initarg :age :accessor person-age))
②             (:metaclass persistent-metaclass))
#<persistent-metaclass person 100A952C>
CL-USER 4 > (setf #l*p-1* (make-instance 'person :name "Klöbner"))
; Stored #<instance class-description person 1.00 slots=2/3
;                  short-objid=8384713>.
#<instance person short-objid=8384679>
CL-USER 5 > (setf #l*p-2* (make-instance 'person :name "Schmoller"))
#<instance person short-objid=8384672>
CL-USER 6 > (mapcar #'person-name
                    (p-select 'person :where 'name :below "Sch"))
("Klöbner")
```

Abbildung 7.37: Definition einer aktiv-persistenten Klasse

Für den *Slot* **name** wird über die *Slot*-Option `:index` ein Index definiert ①. Der Index wird durch einen persistenten B-Baum repräsentiert (Argument `btree` der *Slot*-Option), der für den Schlüsselvergleich die Zustände des *Slots* **name** benutzt (Argumente `:test equal` der *Slot*-Option).

Beim ersten Erzeugen einer Instanz einer aktiv-persistenten Klasse wird das Klassen-Beschreibungsobjekt der Klasse erzeugt und gespeichert (CL-USER 4). Der *Slot* **name** wird durch das Initialisierungs-Schlüsselwort-Argument `:name` auf den übergebenen Namen gesetzt; da für den *Slot* ein Index definiert ist, wird das erzeugte persistente Objekt assoziiert zum Namen in die Indextabelle des *Slots* eingetragen. In diesem Zusammenhang ist zu beachten, daß die Schlüssel einer Indextabelle (d.h. die Zustände des *Slots*, für den ein Index deklariert wurde, über alle Instanzen einer Klasse) eindeutig sein müssen und damit der *Slot* keinen Initialisierungsausdruck haben darf, der für jeden Aufruf den gleichen Wert zurückgibt. Damit würde im Verlauf der Erzeugung einer persistenten Instanz versucht werden, diesen immer gleichen Wert assoziiert zur jeweils erzeugten persistenten Instanz in die Indextabelle einzutragen; da keine doppelten Schlüssel zulässig sind, würde ab der zweiten erzeugten Instanz immer ein Fehler signalisiert werden. Deswegen sollte der Initialisierungsausdruck im allgemeinen weggelassen werden; damit wird der *Slot* nicht implizit während der Objekterzeugung an einen Wert gebunden und sein Zustand wird nicht implizit in die Indextabelle eingetragen.

Der Ausdruck `#l*p-1*` (CL-USER 4) referenziert den Wert des persistenten Symbols mit dem Namen `*p-1*`. Zwischen dem transienten und dem persistenten Symbol `*p-1*` gibt es bei dieser Art der Zuweisung keine Kopplung der Symbolwerte; insbesondere bleibt der Wert des transienten Symbols `*p-1*` unverändert.

Mit der Funktion **p-select** kann assoziativ nach Instanzen gesucht werden (CL-USER 6); in diesem Fall wird nach allen Instanzen der Klasse **person** (oder einer ihrer Superklassen) gesucht, deren im *Slot* **name** enthaltener Zustand literal kleiner als die Zeichenkette `"Sch"` ist.

Abbildung 7.38 zeigt das effektive *Slot*-Beschreibungsobjekt des *Slots* **name** der Klasse

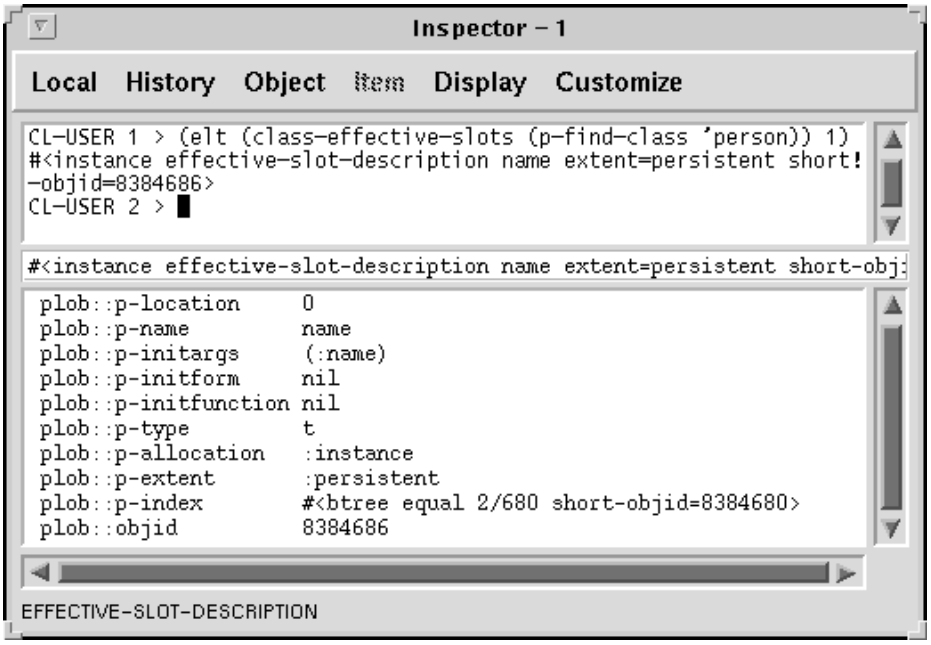


Abbildung 7.38: Beispiel für ein *Slot*-Beschreibungsobjekt

person so, wie es vom LISPWORKS Inspektor visualisiert wird. Der *Slot* **plob::p-location** gibt die Position des Elementes innerhalb des Zustandsvektors von persistenten Instanzen der Klasse **person** an, in dem der *Slot*-Zustand gespeichert wird (Abbildung 7.7, S. 93). Die im *Slot* **plob::p-extent** enthaltene Angabe `:persistent` bedeutet, daß der *Slot* **name** lediglich in der persistenten Repräsentation alloziert wird; es existiert keine Repräsentation für den *Slot*-Zustand in der transienten Repräsentation einer Instanz der Klasse **person** (Abbildung 7.27, S. 120). Der *Slot* **plob::p-index** enthält die Indextabelle des *Slots*, in diesem Fall ein B-Baum mit den zwei bisher erzeugten Elementen (aus Abbildung 7.37, S. 138, CL-USER 4–5). Der B-Baum enthält im Moment eine Seite mit maximal 680 Elementen vor der nächsten Vergrößerung.

Deklarationen für passiv-persistente Klassen

Instanzen von systeminternen Klassen können lediglich passiv-persistent gehalten werden (S. 114). Das Problem für diese Klassen besteht darin, daß ihre Struktur oft redundant ist und sehr viele Objekte referenziert. Über die angebotenen Deklarationen kann veranlaßt werden, bestimmte *Slots* der Klasse nicht persistent zu halten. Abbildung 7.39 zeigt die in LISPWORKS für die Strukturklasse **logical-pathname** notwendigen Deklarationen:

```
① (setf (class-constructor (find-class 'logical-pathname))
        'make-logical-pathname-by-plob)
② (defun make-logical-pathname-by-plob
         (&key host directory name type version)
    (system::make-logical-pathname-from-components
```

```
      host directory name type version))
③ (setf (class-extent (find-class 'logical-pathname)) :cached)
④ (setf (slot-extent 'system::device (find-class 'logical-pathname))
        :transient)
```

Abbildung 7.39: Deklarationen für die systeminterne Strukturklasse **logical-pathname**

Deklaration ①, Funktion ② Die Deklaration ① veranlaßt, daß die in ② definierte Funktion für die Erzeugung der transienten Repräsentation einer persistenten Instanz der Klasse **logical-pathname** aufgerufen wird.

Deklaration ③ Damit wird die Lebensdauer der Zustände aller *Slots* der Klasse **logical-pathname** als `:cached` (Abbildung 7.27, S. 120) deklariert.

Deklaration ④ Durch die Deklaration der Lebensdauer `:transient` für die Lebensdauer des *Slots* **system::device** wird erreicht, daß der *Slot* nicht in der persistenten Repräsentation des Objektes gehalten wird.

Die angegebenen Deklarationen für eine Klasse müssen vor dem Speichern ihres Klassen-Beschreibungsobjektes evaluiert werden.

7.8 Performanz

Don't let the sun go down on me.

— Elton John

Dieser Abschnitt enthält verschiedene Messungen der Performanz von PLOB!. Gemessen wurden die elementaren Funktionen, wie Allozieren und Zugriff auf den Zustand von persistenten Objekten, sowie Einfügen und Lesen von aktiv-persistenten Instanzen, bei denen ein Index für einen *Slot* deklariert wurde. Die Tabellen geben in der Spalte‚LISP' die Zeit an, die LISPWORKS für die Ausführung der Aktion benötigte; in der Spalte‚BS' findet sich die Zeit, die das Betriebssystem mit der Ausführung der Aktion beschäftigt war. Die insgesamt von PLOB! benötigte Zeit ergibt sich aus der Summe der in der Spalten‚LISP' und ‚BS' aufgeführten Zeiten. Die Einheiten sind in eckigen Klammern in der Spaltenüberschrift angegeben; [s] steht für Sekunden und [ms] für Millisekunden. Bei Ladevorgängen wurde für die Messungen darauf geachtet, daß die Objekte tatsächlich aus dem persistenten Speicher und nicht aus dem *Cache* geladen wurden; bei der Benutzung von PLOB! wird man im allgemeinen daher eine wesentlich bessere Performanz als hier angegeben erhalten. Die für die Messung benutzte Rechnerkonfiguration ist in [Kirschke 94] im Abschnitt ‚Installation' beschrieben.

7.8.1 Transaktionen

Die ermittelte Zeit ist der Verwaltungsaufwand für das Starten und Beenden einer Transaktion:

Erklärung	**LISP [ms]**	**BS [ms]**
Transaktion mit leerer Anweisung: `(with-transaction nil)`	0,250	0,0

7.8.2 Sperren

Bei den nachfolgenden Messungen wurde ein Vektor mit 1000 *Immediates* benutzt:

```
(setf *v* (make-array 1000 :initial-element <Immediate>))
```

Auf die transiente Repräsentation wurde mit *Swizzling* (Zugriffsart 1) sowie der direkten Manipulation der persistenten Repräsentation (Zugriffsarten 2–4) schreibend (1. Tabelle) und lesend (2. Tabelle) zugegriffen. Die Zugriffsarten unterscheiden sich in der Anzahl der Transaktionen und der Anzahl der gesetzten Sperren, die für den Zugriff ausgeführt wurden; die Spalten‚Tr.‘ bzw. ‚Sp.‘ geben die Anzahl der durchgeführten Transaktionen bzw. Sperren an:

Nr	Erklärung	Tr.	Sp.	LISP [s]	BS [s]
1	Speichern des transienten Vektors `*v*`: `(setf *objid* (store-object *v*))`	1	1	0,290	0,0
2	Schreiben eines *Immediates* in jede Komponente von `*objid*` mit einer Transaktion für alle Schreibzugriffe und einer Sperre auf Vektorebene für alle Schreibzugriffe	1	1	0,320	0,010
3	Schreiben eines *Immediates* in jede Komponente von `*objid*` mit einer Transaktion für alle Schreibzugriffe und einer Sperre auf Elementebene pro Schreibzugriff	1	1000	4,020	0,020
4	Schreiben eines *Immediates* in jede Komponente von `*objid*` mit einer Transaktion pro Schreibzugriff und einer Sperre auf Elementebene pro Schreibzugriff	1000	1000	15,470	0,050

Die Meßwerte zeigen, daß die Kosten des Zugriffs auf den Zustand eines persistenten Objektes überwiegend bei der Transaktionsverarbeitung und beim Sperrprotokoll liegen. Die Zugriffsart 2 entspricht vom Zeitaufwand her der Zugriffsart 1; Zugriffsart 3 lohnt sich für den schreibenden Zugriff bei bis zu $\frac{0{,}290}{4{,}020/1000} \approx 72$ Elementen, Zugriffsart 4 bei bis zu $\frac{0{,}290}{15{,}470/1000} \approx 19$ Elementen, d.h. ab der jeweils genannten Zahl wird der Aufwand für das Sperrprotokoll so groß, daß es sich lohnt, auf den Vektor nur mit *Swizzling* zuzugreifen.

Nr	Erklärung	Tr.	Sp.	LISP [s]	BS [s]
1	Laden des persistenten Vektors `*objid*`: `(setf *v* (load-object *objid*))`	1	1	0,280	0,0
2	Lesen jeder Komponente von `*objid*` mit einer Transaktion für alle Lesezugriffe und einer Sperre auf Vektorebene für alle Lesezugriffe	1	1	0,280	0,030
3	Lesen jeder Komponente von `*objid*` mit einer Transaktion für alle Lesezugriffe und einer Sperre auf Elementebene pro Lesezugriff	1	1000	4,230	0,010
4	Lesen jeder Komponente von `*objid*` mit einer Transaktion pro Lesezugriff und einer Sperre auf Elementebene pro Lesezugriff	1000	1000	15,580	0,250

Auch hier wird der Gesamtzeitbedarf in erster Linie durch das Sperrprotokoll bestimmt; die eigentliche Übertragung des Zustands fällt kaum ins Gewicht. Die Zugriffsarten 2 und 1 sind vom Zeitaufwand her gleich; Zugriffsart 3 lohnt sich für den lesenden Zugriff bei bis zu $\frac{0{,}280}{4{,}230/1000} \approx 66$ Elementen, Zugriffsart 4 bei bis zu $\frac{0{,}280}{15{,}580/1000} \approx 18$ Elementen.

7.8.3 Listen

Erklärung	LISP [ms]	BS [ms]
Speichern einer einelementigen Liste `'(`<*Immediate*>`)`	14,890	0,020
Laden einer einelementigen Liste `'(`<*Immediate*>`)`	15,200	0,020

Auch beim Speichern und Laden von Listen wird die meiste Zeit für das Sperren der persistenten *cons*-Zellen benötigt.

7.8.4 Felder

Bei den nachfolgenden Messungen wurde ein Feld benutzt, dessen Dimensionen in etwa denen eines RGB-Farbbildes entsprechen:

```
(setf *array* (make-array '(256 256 3)
                          :element-type 'single-float
                          :initial-element 1.0e0))
```

Erklärung	LISP [ms]	BS [ms]
Erzeugen des transienten Feldes `*array*`: `(setf *array* (make-array` <*siehe oben*>`))`	35,0	6,0
Speichern des transienten Feldes `*array*`: `(setf *objid* (store-object *array*))`	222,0	366,0
Erzeugen der transienten Repräsentation und Laden des persistenten Feldes `*objid*`: `(setf *array* (load-object *objid*))`	97,0	59,0

Das Feld besteht aus *Immediates* des Typs **single-float**; daher wird der Zustand der Feldelemente sowohl in der transienten als auch in der persistenten Repräsentation in einem *ivector* abgelegt

(Abbildung 7.5, S. 91). Bei der Übertragung des Feldes zwischen transientem und persistentem Speicher wird der Zustand des *ivectors* unkonvertiert zwischen den Umgebungen kopiert; damit erklärt sich die relativ hohe Betriebssystemzeit bei der Übertragung. Für das Kopieren des Feldzustands wird die C Funktion **memcpy** benutzt, die wahrscheinlich byteweise kopiert; in diesem Fall würde ein Ersetzen der Standard-C-Funktion **memcpy** durch eine selbst realisierte Kopierfunktion, die wortweise arbeitet, einen erheblichen Effizienzgewinn bringen.

7.8.5 Aktiv-persistente Instanzen

Die *Slot*-Namen und -Lebensdauern in der Definition der zur Messung verwendeten aktiv-persistenten Klasse **p-example-class** wurden so gewählt, daß sie zu Abbildung 7.27 (S. 120) passen. Transiente Instanzen der Klasse **t-example-class** werden zu Vergleichsmessungen benutzt.

```
(defclass p-example-class ()
  ((slot-4  :initform nil :extent :transient)
   (slot-6  :initform nil :extent :cached)
   (slot-8  :initform nil :extent :cached-write-through)
   (slot-10 :initform nil :extent :persistent))
  (:metaclass persistent-metaclass))

(defclass t-example-class ()
  ((slot-3 :initform nil) (slot-5 :initform nil)
   (slot-7 :initform nil) (slot-9 :initform nil)))
```

Erzeugung von aktiv-persistenten Instanzen

Erklärung	LISP [ms]	BS [ms]
Erzeugen einer (rein transienten) Instanz der Klasse **t-example-class**	0,090	0,0
Erzeugen einer persistenten Instanz der Klasse **p-example-class**	45,810	0,770
Laden einer persistenten Instanz der Klasse **p-example-class**	38,610	0,240

Die Werte in der ersten Zeile zeigen zum Vergleich die Werte der Erzeugung einer Instanz für eine rein transiente Klasse.

Zugriff auf den Zustand von aktiv-persistenten Instanzen

Bei der folgenden Messung wurde jeder *Slot*-Zustand einmal auf ein *Immediate* gesetzt.

```
(setf *objid* (make-instance 'p-example-class))
(setf (slot-value *objid* <Slot-Name>) <Immediate>)
```

Slot-Name	*Slot*-Lebensdauer	LISP [ms]	BS [ms]
`slot-3`	*Slot* aus **t-example-class**	0,007	0,0
`slot-4`	`:transient`	0,430	0,0
`slot-6`	`:cached`	0,440	0,0
`slot-8`	`:cached-write-through`	1,340	0,0
`slot-10`	`:persistent`	1,280	0,0

Wie sich zeigt, ist die Performanz schlechter als bei rein transienten Klassen, auch bei der *Slot*-Lebensdauer `:transient` des *Slots* **slot-4** der aktiv-persistenten Klasse **p-example-class**. Dies ist darin begründet, daß die entsprechenden *Slot*-Lebensdauern aus dem im Klassen-Beschreibungsobjekt enthaltenen *Slot*-Beschreibungsobjekt gelesen werden; offensichtlich ist die Ermittlung des *Slot*-Beschreibungsobjektes relativ zeitintensiv.

Für die *Slot*-Lebensdauer `:cached-write-through` wird der *Slot*-Zustand auch in die transiente Repräsentation propagiert; damit ist der schreibende Zugriff langsamer als bei der *Slot*-Lebensdauer `:persistent`.

Bei der folgenden Messung wurde jeder *Slot*-Zustand einmal gelesen.

`(slot-value *objid* <`*Slot-Name*`>)`

***Slot*-Name**	***Slot*-Lebensdauer**	**LISP [ms]**	**BS [ms]**
`slot-3`	*Slot* aus **t-example-class**	0,006	0,0
`slot-4`	`:transient`	0,390	0,0
`slot-6`	`:cached`	0,400	0,0
`slot-8`	`:cached-write-through`	0,400	0,0
`slot-10`	`:persistent`	1,170	0,0

Erwartungsgemäß unterscheiden sich die Zeiten für das *Slot*-Lesen und -Schreiben signifikant bei der *Slot*-Lebensdauer `:cached-write-through` des *Slots* **slot-8** der aktiv-persistenten Klasse **p-example-class**; Schreibzugriffe auf den *Slot* werden sowohl in die transiente als auch in die persistente Repräsentation des persistenten Objektes propagiert, während Lesezugriffe immer die transiente Repräsentation referenzieren.

Indexdeklarationen für aktiv-persistente Klassen

Für die letzte Messung wurde eine aktiv-persistente Klasse mit einem Index benutzt. Der *Slot* **slot-i**, für den der Index `(btree :test equal)` deklariert wurde, wurde bei der Objekterzeugung initialisiert; damit wurde die Assoziation vom *Slot*-Zustand auf das Objekt im Verlauf der Erzeugung der aktiv-persistenten Instanz aufgebaut:

```
(defclass p-example-index-class ()
  ((slot-i :initarg :slot-i
           :index (btree :test equal)
           :extent :persistent))
  (:metaclass persistent-metaclass))
```

Erklärung	**LISP [ms]**	**BS [ms]**

Erzeugung einer Instanz der Klasse **p-example-index-class**:
`(make-instance`
`'p-example-index-class`
`:slot-i "Slot-I with string <`*Nummer*`>")`
	114,800	0,970

Laden einer Instanz der Klasse **p-example-index-class**:
`(p-select 'p-example-index-class`
`:where 'slot-i)`
	64,490	0,250

Im Verlauf der Objektinitialisierung wird die Zeichenkette abgespeichert und die Assoziation vom *Slot*-Zustand auf das Objekt in die Indextabelle eingetragen; damit ergibt sich ein höherer Zeitbedarf.

7.8.6 Bewertung der Performanz

Wie insbesondere die erste Messung zeigt, ist der Zeitbedarf für das Sperrprotokoll in Relation zur eigentlichen Zustandsübertragung recht hoch. In der 2. Schicht könnten noch Optimierungen durchgeführt werden, da dort im Moment das Sichern des Zustands der POSTORE-Vektoren in die UNDO-*Log*-Datei mit den relativ ineffizienten *Stream*-Funktionen der Standard-C-Bibliothek erfolgt.

In der 3. Schicht ließe sich eine Verbesserung der Performanz beim Zugriff auf *Slots* von aktiv-persistenten Instanzen durch Optimierung der spezialisierten Methoden der generischen Funktion **make-accessor-lambda** (Protkoll 7.19 **(defclass)**, Schritt 2.3.12, S. 113) erreichen. Sofern Wert auf Performanz gelegt wird, sollten in der 4. Schicht Transaktionen möglichst große Blöcke umfassen.

7.9 Zusammenfassung und Bewertung

Realisiert wurde ein persistentes System unter Verwendung eines Objektspeichers, dessen Datenmodellierung sich an LISP orientiert. Die Möglichkeit der Persistenz ist für den Zustand aller LISP Objekte unabhängig vom Typ gegeben; insbesondere ist für Persistenz von CLOS-Objekten nicht die Einbindung einer spezialisierten Klassen-Metaobjekt-Klasse in ihre Klassendefinition notwendig. Die Schnittstelle für die Benutzung des persistenten Systems orientiert sich soweit möglich an den in [CLtLII] dargestellten Konzepten; damit wird die Benutzung des Systems sehr einfach und die Einbindung in vorhandene Programmodule erleichtert.

Durch die realisierte Möglichkeit, Persistenz deklarativ auch für beliebige, im System vorgefundene Klassen beeinflussen zu können, wird eine hohe Effizienz des Systems erreicht.

Die Realisierung von Sitzungen, Zwei-Phasen-Transaktionen, des hierarchischen Sperrprotokolls und einer Indexverwaltung für eindimensionale Schlüssel bietet die grundlegenden Funktionalitäten einer Datenbank.

Insgesamt bietet das hier realisierte System Persistenz mit LISP-gemäßer Datenmodellierung mit guter Performanz an.

Noch nicht realisiert ist assoziativer Zugriff mit mehrdimensionalen Schlüsseln. Ebenso sind die Anfragemöglichkeiten noch sehr einfach gehalten. Echte Mehr-BenutzerInnen-Zugriffe sind im Moment nicht möglich, da der in der 1. Schicht verwendete Objektspeicher POSTORE nicht *reentrant*-fähig ist.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Ziel dieser Arbeit war die Realisierung eines Systems, das Common LISP Datenobjekte unabhängig vom daten-erzeugenden transienten Prozeß persistent hält. Wegen des beabsichtigten Einsatzgebietes kann die Anzahl der persistent zu haltenden Objekte sehr groß werden; ebenso können die Objekte selbst sehr groß sein, wie z.B. Satellitenbilder hoher Auflösung.

Der gewählte Weg bestand zunächst darin, die für persistente Systeme wichtigen Konzepte zu evaluieren, um sowohl vorhandene Systeme bewerten als auch ein eigenes System entwerfen zu können:

- Common LISP Datenobjekte haben eine Identität, die explizit repräsentiert wird; dazu war es zunächst nötig, den Begriff der Identität zu klären und die verschiedenen Repräsentationsformen für Identität im Hinblick auf Persistenz zu analysieren und zu bewerten, um eine geeignete Wahl für die Realisierung treffen zu können.

- Aus dem beabsichtigten Einsatzzweck des zu realisierenden Systems ergab sich die Konzeption für die Gesamtarchitektur des Systems und die Auswahl der Subkomponenten. Als Speicher für die Ablage von Objektzuständen ergaben sich im Rahmen dieser Arbeit die beiden Möglichkeiten der Verwendung einer relationalen Datenbank und eines Objektspeichers. Eine relationale Datenbank hat den Vorteil mächtiger Manipulationsmöglichkeiten; nachteilig ist die für Objekte mit Referenzen ungeeignete Datenmodellierung. Ein Objektspeicher hingegen basiert nicht auf dem relationalen Modell und hat lediglich einfache Zugriffsmöglichkeiten, bietet aber dafür die Möglichkeit, explizite Referenzen zwischen Objekten zu etablieren.

Nach der Konzeption folgte die Realisierungsphase:

- Zunächst wurde geprüft, ob das mit einer relationalen Datenbank als Speicher für die Zustände von persistenten Objekten realisierte System Shared Object Hierarchy (SOH) für den beabsichtigten Zweck geeignet sei. Dazu wurde der Code an das zur Verfügung stehende LISP-System und an eine neuere Version der Datenbank angepaßt. Im Verlauf

der Portierung stellte sich heraus, daß SOH die Probleme, die sich bei Verwendung einer Datenbank zur Speicherung von Objekten ergeben, nur ungenügend löst.

- Die entwickelten Konzepte und die Erfahrungen mit dem System SOH mündeten in den Entwurf und die Realisierung des Systems PLOB!. Wegen der prinzipiellen Schwierigkeiten der Repräsentation von Objektzuständen und Referenzen zwischen Instanzen in relationalen Datenbanken entschied ich mich dafür, einen Objektspeicher für die Ablage der Objektzustände zu verwenden.

  Auf diesen Objektspeicher wurde eine Schicht mit Datenbankfunktionalitäten aufgesetzt, die Sitzungen, Zwei-Phasen-Transaktionen, ein hierarchisches Sperrprotokoll und die für eine Indexverwaltung notwendigen Hilfsmittel über persistente Objekte realisiert. Diese Ebene realisiert transaktionsgesicherten und durch das Sperrprotokoll unteilbaren Zugriff auf dynamisch typisierte, selbstbeschreibende Objekte.

  Die oberste Ebene realisiert unter Benutzung der Funktionalität der nächst-niedrigen Schicht transparente und orthogonale Persistenz für Common LISP Daten sowie die Möglichkeit des assoziativen Zugriffs auf persistente Objekte.

Bei der Realisierung zeigten sich die Vorteile von objekt-orientierten prozedural-reflektiven Sprachen wie CLOS bei der Erweiterung der Sprache um Persistenz; für PASCAL/R [Schmidt 77, S. 259] und *E* [Richardson et al. 89] mußten jeweils Übersetzer modifiziert bzw. erstellt werden, während für die Einbindung von Persistenz in CLOS eine Modifikation des Systems mit Mitteln des Systems selbst möglich ist.

8.2 Anwendungen für persistente Systeme

8.2.1 Softwareentwicklung in einem transienten und persistenten Common LISP-System

Das nachfolgende Beispiel zeigt, daß persistente Objektsysteme für Software-Entwicklungsumgebungen verwendet werden können; die persistenten Objekte ersetzen dabei das übliche Hilfsmittel für Persistenz in transienten Systemen, in diesem Beispiel also Dateien mit Quell- und Objekt-Codes. Das hier angegebene Beispiel stellt einige prinzipielle Möglichkeiten eines persistenten Systems dar; nicht alle der folgenden Schritte lassen sich mit dem aktuellen Entwicklungsstand von PLOB! realisieren.

	Transientes Common LISP	**Persistentes Common LISP**
1	Starten des LISP-Systems	Starten des LISP-Systems
2	Laden der Programm-Definitionsdatei (sie definiert die zum Programm gehörigen Module)	Programm-Definitionsdatei befindet sich im persistenten Speicher
3	Laden der Programm-Module	Programm-Module befinden sich im persistenten Speicher
4	Sichern des Quellcodes vor Beginn der Modifikationen	Start einer langandauernden Transaktion
5	Laden des zu ändernden Quellcodes in den Editor	Quellcode befindet sich im persistenten Speicher

	Transientes Common LISP	Persistentes Common LISP
6	Lokalisierung und Änderung einer Klassendefinition im Quellcode	Klassendefinition wird über einen *Browser* erst selektiert und dann editiert
7	Übersetzen und Testen der geänderten Klasse	Übersetzen und Testen der geänderten Klasse
8	Die Benutzerin stellt fest, daß ihre Änderungen falsch waren; sie muß den in Schritt 4 gesicherten Quellcode erneut laden	Die Benutzerin stellt fest, daß ihre Änderungen falsch waren; sie bricht die in Schritt 4 begonnene langandauernde Transaktion ab. Damit werden alle betroffenen Module auf den alten Stand zurück gebracht (*rollback*)
9	Erneutes Lokalisieren, Ändern (Schritt 6) und Testen (Schritt 7)	Erneutes Selektieren, Ändern (Schritt 6) und Testen (Schritt 7)
10	Die Änderungen wurden erfolgreich abgeschlossen; die Quellcodes werden gesichert	Die langandauernde Transaktion, die in Schritt 4 gestartet wurde, wird erfolgreich beendet (*committed*)

Für das persistente System wird in diesem Beispiel angenommen, daß Persistenz ohne Einschränkung für alle im System auftretenden Datentypen zur Verfügung steht. Ebenso wird angenommen, daß das persistente System bereits einige Datenbankfunktionalität beinhaltet, wie z.B. Transaktionen (Schritt 4 und 10) sowie Introspektions- und Selektionsmöglichkeiten (Schritt 6).

Wie das Beispiel zeigt, sind keine Zugriffe auf Dateien nötig; damit wird die Abgeschlossenheit des Systems erhöht. Ebenso lassen sichÄnderungen leicht rückgängig machen (Schritt 4 und 8). Sofern das System zusätzlich noch eine Versionskontrolle bietet, ließe sich ebenfalls ein älterer Zustand des Gesamtsystems wiederherstellen.

8.3 Ausblick

Dieser Abschnitt enthält Vorschläge, wie das in Kapitel 7 beschriebene System erweitert werden könnte.

8.3.1 Verlagerung von Funktionen aus der 3. in die 2. Schicht

Einige Funktionalitäten der 3. Schicht sollten in die 2. Schicht verlagert werden; dies erleichtert eine Nutzung des persistenten Systems durch nicht-LISP-Systeme. Ebenso wäre damit eine erhöhte Effizienz des persistenten Systems verbunden.

Sitzungsverwaltung

Die Sitzungsverwaltung (Öffnen und Schließen einer Sitzung) sollte in die 2. Schicht verlagert werden.

Indexverwaltung

Die Verwaltung der *Slot*-Indextabellen wird im Moment in der 3. Schicht in den spezialisierten Methoden der generischen Funktionen **slot-...-using-class** durchgeführt. Alle Informationen zur Indexverwaltung sind in der 2. Schicht verfügbar; die Zugriffs-Funktionen der 2. Schicht könnten dementsprechend die Pflege des für einen *Slot* deklarierten Index bei Zugriff auf den *Slot*-Zustand übernehmen.

8.3.2 Erweiterungen der 2. Schicht

Dieser Abschnitt schlägt diverse Erweiterungen für die 2. Schicht vor.

Doppelte Schlüssel für persistente B-Bäume

Die momentan realisierten persistenten B-Bäume können keine doppelten Schlüssel verarbeiten; sie sollten dementsprechend erweitert werden. In diesem Zusammenhang würde sich für den einfachen Zugriff auf B-Bäume mit doppelten Schlüsseln die Realisierung von *Portals* [Wensel 93, S. 114] anbieten. In etwa gleichbedeutend ist das *Cursor*-Konzept von SQL [O'Neil 94, S. 220]; *Portals* sind allgemeiner als *Cursor*, da in ihnen Tupel verschiedener Relationen enthalten sein können.

Transaktionsarten

Momentan wurde nur das einfachste Transaktionsmodell der‚flachen' Transaktionen realisiert; in [Gray et al. 93, S. 187–210] werden verschiedene andere Transaktionsmodelle vorgestellt. Meiner Meinung nach wäre insbesondere die Realisierung von verschachtelten Transaktionen (*nested transactions*) [Gray et al. 93, S. 195] lohnend, da sie die Möglichkeit bieten, die Transaktionsverarbeitung an die Modulstruktur des persistenten Systems anzupassen.

Zusätzliche Sperre

Um im Betrieb mit mehreren BenutzerInnen den *Cache* konsistent zu halten, sollte eine zusätzliche, zu allen bisher definierten Sperren kompatible Sperre‚*cached*' realisiert werden, die im Verlauf der Übertragung eines Objektzustands zwischen persistentem und transientem Speicher auf das übertragene Objekt für den übertragenden Prozeß gesetzt wird. Die 2. Schicht kann dann bei Modifikation des im persistenten Speicher abgelegten Objektzustands durch einen Prozeß an alle Prozesse, die eine ‚*cached*'-Sperre auf das Objekt gesetzt haben, benachrichtigen, daß der Zustand der vom *Cache* referenzierten transienten Repräsentation des persistenten Objektes ungültig ist und nachgeladen werden sollte.

Mehrdimensionale Indizes

Persistente B-Bäume bieten lediglich eindimensionale Schlüssel; für den angestrebten assoziativen Zugriff auf geografische Daten wäre die Realisierung von mehrdimensionalen Indizes nötig.

Trennung der Prozesse in einen *Frontend*- und *Backend*-Prozeß

Ähnlich wie in POSTGRES [Wensel 93, S. 30] wäre eine Trennung in einen *Backend*-Prozeß zur Verwaltung des persistenten Speichers und einen benutzenden *Frontend*-Prozeß vorteilhaft. Ein-

facher zu realisieren wäre zunächst die Bedienung aller *Frontend*-Prozesse durch einen einzigen globalen *Backend*-Prozeß. Nötig wäre dazu die Realisierung einer Kommunikationsschicht, die zwischen den Schichten 1 und 2 anzusiedeln wäre.

Die Bedienung der *Frontend*-Prozesse durch jeweils einen lokalen, dazugehörigen *Backend*-Prozeß wie in der objekt-orientierten relationalen Datenbank POSTGRES ist im Moment nicht realisierbar, da der in der 1. Schicht verwendete Objektspeicher POSTORE nicht mit mehreren Prozessen gleichzeitig auf einen persistenten Speicher zugreifen kann [Brown 92, S. 1].

Explizite Repräsentation von Prozessen

Benutzende Prozesse werden im Moment durch Instanzen der Klasse **persistent-heap** implizit repräsentiert; für Prozesse sollte daher in der 2. Schicht eine Basisklasse definiert werden. Die Klasse **persistent-heap** sollte in diesem Zusammenhang um einen *Slot* erweitert werden, der auf die Repräsentation des benutzenden Prozesses verweist; beimÖffnen einer Sitzung sollte die Referenz auf die Prozeßrepräsentation in diesen *Slot* eingetragen werden.

Glossar

Abbbruch einer Transaktion (*abort*) (S. 95) Damit gilt der aus den während der Transaktion erfolgten Modifikationen hervorgegangene Gesamtzustand der Datenbank als inkonsistent; der Gesamtzustand wird daher auf den letzten konsistenten Stand bei Beginn der Transaktion zurückgebracht (→*rollback*).

abort (S. 95) →Abbbruch einer Transaktion

Absichtssperre (*intent lock*) (S. 97) Eine Absichtssperre repräsentiert beim →hierarchischem Sperrprotokoll die Absicht eines sperrenden Objektes, auf einer tieferen Ebene eine entsprechende Sperre vornehmen zu wollen. Wenn beispielsweise die Sperrebenen‚Speicher‘ – ‚Vektor‘ – ‚Komponente eines Vektors‘ gegeben sind, wird mit einer Leseabsichtssperre auf der Ebene ‚Speicher‘ die Absicht angezeigt, auf einer der Ebenen ‚Vektor‘ oder ‚Komponente eines Vektors‘ eine Lesesperre setzen zu wollen.

ACID (S. 84) Abkürzung für die vier Eigenschaften einer Datenbank-Transaktion: →Unteilbarkeit (*atomicity*), →Konsistenz (*consistency*), →Serialisierbarkeit (*isolation*) und →Beständigkeit (*durability*).

Aktiv-persistente Instanz (S. 56) Eine CLOS-Instanz, deren Klassen-Metaobjekt-Klasse auf Persistenz spezialisiert ist.

Aktiv-persistente Klasse (S. 56) Eine Klasse, die durch ein Klassen-Metaobjekt einer auf Persistenz spezialisierten Klassen-Metaobjekt-Klasse repräsentiert wird.

Aktive Instanz (S. 54) Bei einer aktiven Instanz kann eine Änderung oder Erweiterung ihrer Semantik (Erweiterung um Persistenz in diesem Rahmen) grundsätzlich mittels der *vorhandenen*, in [CLtLII] und [AMOP] definierten generischen Standard-Funktionen eingebunden werden; in Common LISP sind alle CLOS-Instanzen aktive Instanzen.

Allozierungsart eines *Slots* (*slot allocation*) (S. 106) Die Allozierungsart eines *Slots* bestimmt, wie der *Slot*-Zustand von CLOS repräsentiert werden soll. Für die Art `:instance` wird der *Slot*-Zustand für jede Instanz repräsentiert, für die Art `:class` wird der *Slot*-Zustand genau einmal repräsentiert, für alle anderen Allozierungsarten wird der *Slot*-Zustand nicht von CLOS repräsentiert.

array information function (S. 130) →Feld-Informationsfunktion

atomicity (S. 84) →Unteilbarkeit

Backend-Prozeß (S. 70) Datenbank-bearbeitender UNIX Prozeß; er verarbeitet die vom →*Frontend*-Prozeß kommenden Datenbank-Anfragen.

Basisklasse (*built-in class*) (S. 5) Eine Basisklasse ist eine in das LISP-System‚eingebaute' (*built-in*) Klasse, zu der im Gegensatz zu allgemeinen Klassen keine Subklassen spezialisiert werden können. Ebenso wird auf den Zustand der Instanzen von Basisklassen nicht mit Methoden, sondern mit Funktionen zugegriffen; damit entfällt ebenfalls die Möglichkeit, die Standard-Zugriffs-Methoden weiter spezialisieren zu können.

Beständigkeit (*durability*) (S. 85) Die 4. Eigenschaft der ACID-Eigenschaften für Datenbank-Transaktionen: Nach Ende einer Transaktion (→*commit*) überstehen die Zustandsänderungen Systemausfälle.

built-in class (S. 5) →Basisklasse

clustering (S. 44) Nahe Gruppierung von Objekten bei ihren referenzierten Objekten.

commit (S. 85) →Ende einer Transaktion

compiler (S. 45) Übersetzer.

consistency (S. 85) →Konsistenz

dangling reference (S. 41) →Ungültige Referenz

deadlock (S. 98) Verklemmung.

debugger (S. 83) Programm zur Fehlersuche; kontrolliert den Ablauf des zu untersuchenden Programms und fängt dessen Laufzeitfehler ab.

displaced array (S. 130) Ein *displaced array* ist ein Feld, das Elemente eines anderen Feldes referenziert. Jeder Zugriff auf ein Feldelement liest bzw. schreibt ein Element des referenzierten Feldes.

displaced index offset (S. 130) Der erste Index für ein Feld, daß ein anderes Feld referenziert (→*displaced array*) des gemeinsam referenzierten Teils.

durability (S. 85) →Beständigkeit

Ende einer Transaktion (*commit*) (S. 85) Damit gilt der aus den während der Transaktion erfolgten Modifikationen hervorgegangene Gesamtzustand der Datenbank als konsistent.

Endlos-Schleife →Schleife, Endlos-∼

exclusive mode (S. 97) →Exklusiver Sperrmodus

Exklusiver Sperrmodus (*exclusive mode*) (S. 97) Ein Modus, der nur einem sperrenden Objekt Zugriff auf eine gesperrte Instanz erlaubt.

extent (S. 106) Lebensdauer.

Feld-Informationsfunktion (*array information function*) (S. 130) Eine Feld-Informationsfunktion liefert Informationen über ein LISP Datenobjekt der Klasse Feld (*array*), wie z.B. die Anzahl Feldelemente, eine Liste der Felddimensionen, den Elementtyp, für den das Feld spezialisiert wurde usw.

finalized (S. 109) Endgültig abgeschlossene Initialisierung eines Klassen-Metaobjektes oder Klassen-Beschreibungsobjektes.

flag (S. 109) Statusvariable.

Freizeiten (*idle time*) (S. 42) Zeitabschnitte, in denen kein Prozeß aktiv ist.

Frontend-Prozeß (S. 70) Datenbank-benutzender UNIX Prozeß; er startet einen →*Backend*-Prozeß, sendet ihm die Datenbank-Anfragen und gibt die Antworten an das in den höheren Schichten des *Frontend*-Prozesses laufende von den BenutzerInnen erstellte Programm weiter.

garbage collection (S. 41) →Speicherrückgewinnung

Generische Standard-Lese-Funktion (*standard reader generic function*) (S. 104) Alle in [AMOP, S. 212–224] definierten generischen Standard-Lese-Funktionen.

Großes Objekt (*large object*) (S. 70) Ein großes Objekt ist ein Objekt innerhalb einer relationalen Datenbank, das die maximale Tupelgröße überschreitet; beispielsweise ist in POSTGRES die maximale Größe eines Tupels auf 8 KByte beschränkt.

Hierarchisches Sperrprotokoll (*tree locking*) (S. 97) Das hierarchische Sperrprotokoll impliziert mit dem →Sperren eines Objektes auf einer Ebene mit einem Modus denselben Sperrmodus für alle Unterebenen. Wenn beispielsweise die Sperrebenen‚Speicher‘ – ‚Vektor‘ – ‚Komponente eines Vektors‘ gegeben sind, impliziert eine Lesesperre auf der Ebene ‚Speicher‘ Lesesperren für alle weiteren unteren Ebenen‚Vektor‘ (d.h. für *alle* Vektoren) und ‚Komponente eines Vektors‘.

Hypothetische Transaktion (S. 96) Eine hypothetische Transaktion ist eine Transaktion, die bei Ende immer abgebrochen wird; sie kann z.B. für Entwicklungsarbeiten genutzt werden, deren Ergebnisse niemals persistent gehalten werden sollen.

idle time (S. 42) →Freizeiten

Immediate (S. 4) Ein *Immediate* ist ein Wert, dessen Zustand direkt in einer Identitätsrepräsentation repräsentiert werden kann.

initarg (S. 106) →Initialisierungs-Argumentname

initform (S. 106) →Initialisierungsausdruck

initfunction (S. 106) →Initialisierungsfunktion

Initialisierungs-Argumentname (*initarg*) (S. 106) Ein Initialisierungs-Argumentname wird zur Initialisierung eines *Slots* bei der Erzeugung eines Struktur- oder CLOS-Objektes als Schlüssel eines Schlüsselwort-Parameters (*&key parameter*) zusammen mit dem Initialisierungswert an die objekt-erzeugende Funktion übergeben.

Initialisierungsausdruck (*initform*) (S. 106) LISP Ausdruck, dessen Evaluierung im Verlauf der Initialisierung eines *Slots* eines CLOS-Objektes als Initialisierungswert verwendet wird. Nicht der Initialisierungsausdruck selbst wird zur Initialisierung benutzt, sondern eine aus ihm durch eine Übersetzung erzeugte Initialisierungsfunktion (→*initfunction*).

Initialisierungsfunktion (*initfunction*) (S. 106) LISP Funktion ohne Argumente, deren Rückgabewert im Verlauf der Initialisierung eines *Slots* eines CLOS-Objektes als Initialisierungswert verwendet wird. Die Initialisierungsfunktion wird während der Klasseninitialisierung durch eine Übersetzung aus dem Initialisierungsausdruck (→*initform*) gewonnen.

intent lock (S. 97) →Absichtssperre

isolation (S. 85) →Serialisierbarkeit

keyword symbol (S. 106) →Schlüsselwort-Symbol

Konsistenz (*consistency*) (S. 85) Die 2. Eigenschaft der ACID-Eigenschaften für Datenbank-Transaktionen: Eine Transaktion ist eine korrekte Zustandsänderung. Die zu einer Gruppe zusammengefaßten Aktionen verletzen keinerlei Integritätsbeschränkungen des Zustands.

large object (S. 70) →Großes Objekt

locking (S. 96) →Objektsperre

Lokalisierung von Objekten (S. 61) Auswahl von persistenten Objekten aus dem persistenten Speicher anhand von Kriterien, wie z.B. nach dem Namen des Objektes oder nach bestimmten *Slot*-Zuständen.

Nicht-exklusiver Sperrmodus (*shared mode*) (S. 97) Ein Modus, der mehreren sperrenden Objekten Zugriff auf eine gesperrte Instanz erlaubt.

Objekt-Swizzling (S. 60) Abbildung zwischen einer transienten und einer persistenten Repräsentation eines persistenten Objektes.

Objektsperre (*locking*) (S. 96) Vor der Dereferenzierung des Objektzustands wird das Objekt mit einem der →nicht-exklusiven Sperrmodi (*shared mode*) Nur-Lesen (*read-only*), Lesen (*read*) oder dem →exklusivem Sperrmodus (*exclusive mode*) Schreiben (*write*) durch ein sperrendes Objekt (im allgemeinen eine →Sitzung) für den Zugriff durch andere Instanzen gesperrt.

Passiv-persistente Instanz (S. 56) Eine nicht →aktiv-persistente Instanz.

Passiv-persistente Klasse (S. 56) Eine nicht →aktiv-persistente Klasse.

Passives Objekt (S. 56) Eine nicht →aktive Instanz. In Common LISP sind alle Objekte der Basis- und Struktur-Klassen passive Objekte.

Pointer-Swizzling (S. 19) Konvertierung einer persistenten Identitätsrepräsentation in eine Adresse.

rollback (S. 95) Zurücksetzen der Datenbank auf den Zustand bei Beginn der Transaktion nach Abbruch einer Datenbank-Transaktion.

Schleife, Endlos-∼ →Endlos-Schleife[1]

Schlüsselwort-Symbol (*keyword symbol*) (S. 106) Jedes Schlüsselwort-Symbol ist per Definition immer global sichtbar und hat sich selbst als Wert.

Schreibsperre (S. 86) Markierung eines Objektes als‚gesperrt', um anderen zugreifenden Instanzen zu signalisieren, daß der Objektzustand schreibend referenziert wird.

Schwache Referenz (*weak pointer*) (S. 50) Schwache Referenzen werden zur Laufzeit wie normale Referenzen angesehen. Während der →Speicherrückgewinnung werden schwache Referenzen nicht berücksichtigt, d.h. Objekte, die nur über schwache Referenzen erreichbar sind, gelten als nicht-erreichbar.

Selbstbeschreibendes System (*self-descriptive*) (S. 51) Ein Objektsystem ist selbst-beschreibend, wenn Struktur und Verhalten aller Instanzen entweder als‚eingebaut' (*built-in*) definiert ist oder anderweitig vom Objektsystem erfragt werden kann.

self-descriptive (S. 51) →Selbstbeschreibendes System

Serialisierbarkeit (*isolation*) (S. 85) Die 3. Eigenschaft der ACID-Eigenschaften für Datenbank-Transaktionen: Auch bei gleichzeitig laufenden Transaktionen sieht es für jede Transaktion *T* so aus, als ob andere Transaktionen entweder vor *T* oder nach *T* ausgeführt würden.

session (S. 95) →Sitzung

shared mode (S. 97) →Nicht-exklusiver Sperrmodus

Sitzung (*session*) (S. 95) Eine Folge von zeitlich disjunkten Datenbank-Transaktionen sowie einem Prozeß, der innerhalb einer Transaktion auf persistente Objekte zugreifen kann.

slot allocation (S. 106) →Allozierungsart eines *Slots*

Speicherrückgewinnung (*garbage collection*) (S. 41) Freigabe des Speicherplatzes von nicht-erreichbaren Objekten.

standard reader generic function (S. 104) →Generische Standard-Lese-Funktion

time out (S. 98) Ablauf einer bestimmten Wartezeit.

tree locking (S. 97) →Hierarchisches Sperrprotokoll

two-phased transaction (S. 64) →Zwei-Phasen-Transaktion

type tag (S. 89) →Typkennung

Typkennung (*type tag*) (S. 89) Numerischer Wert, der den Typ eines Objektes eineindeutig identifiziert.

Ungültige Referenz (*dangling reference*) (S. 41) Eine ungültige Referenz ist eine Identitätsrepräsentation, die kein gültiges Objekt referenziert.

[1] – Random Shack Data Processing Dictionary

Unteilbarkeit (*atomicity*) (S. 84) Die 1. Eigenschaft der ACID-Eigenschaften für Datenbank-Transaktionen: Die von einer Transaktion vorgenommenen Zustandsänderungen sind unteilbar.

weak pointer (S. 50) →Schwache Referenz

well-formed transaction (S. 96) →Wohlgeformte Transaktion

Wohlgeformte Transaktion (*well-formed transaction*) (S. 96) In einer wohlgeformten Transaktion sind die Objekte für den Zugriff auf ihren Zustand durch die dereferenzierende Instanz (im allgemeinen eine →Sitzung) gesperrt. Damit in einer Transaktion →Serialisierbarkeit gewährleistet wird, muß die Transaktion wohlgeformt sein.

Zwei-Phasen-Transaktion (*two-phased transaction*) (S. 64) In einer Zwei-Phasen-Transaktion werden in der ersten Phase die Objekte, auf die zugegriffen werden soll, durch die zugreifende Transaktion für alle konkurrenten Transaktionen gesperrt; die zweite Phase, in der für die zugreifende Transaktion keine neuen Sperren gesetzt werden dürfen, beginnt mit der ersten Aufhebung einer ihrer Sperren. Damit wird →Serialisierbarkeit gewährleistet.

Literaturverzeichnis

„I saw Hamlet last night. [...] It's full of quotations."

— Samuel Goldwyn

After all, all he did was string together a lot of old, well-known quotations.

— H. L. Mencken über Shakespeare

[Ahmed et al. 91] Shamim Ahmed, Albert Wong, Duvvuru Sriram, Robert Logcher: A Comparision of Object-Oriented Database Management Systems for Engineering Applications. Bericht R91-12, MIT, 1991

[AMOP] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow: The Art of the Metaobject Protocol. MIT Press, Cambridge, Mass., 1991

[Atkinson et al. 89] Malcolm Atkinson, Ronald Morrison: Persistent System Architectures. John Rosenberg, David Koch (Hrsg.): *Persistent Object Systems, Newcastle, Australia 1989*, S. 73–97, Springer-Verlag, Berlin, 1989

[Beech 87] David Beech: Groundwork for an Object Database Model. Bruce Shriver, Peter Wegner (Hrsg.): *Research Directions in Object-Oriented Programming*, S. 317–354, MIT Press, Cambridge, Mass., 1987

[Björnerstedt et al. 89] Anders Björnerstedt, Christer Hultén: Version Control in an Object-Oriented Architecture. Won Kim, Frederick H. Lochovsky (Hrsg.): *Object-Oriented Concepts, Databases, and Applications*, S. 451–485, ACM Press, New York, 1989

[Brössler et al. 89] P. Brössler, B. Freisleben: Transactions on Persistent Objects. John Rosenberg, David Koch (Hrsg.): *Persistent Object Systems, Newcastle, Australia 1989*, S. 303–318, Springer-Verlag, Berlin, 1989

[Brown 92] A. L. Brown: Stable Heap manual pages, Datei `postore/man/stable_heap.3p`, 25 Mai 1992

[Cardelli et al. 88] Luca Cardelli, David MacQueen: Persistence and Type Abstraction. Malcolm P. Atkinson, Peter Buneman, Ronald Morrison (Hrsg.): *Data Types and Persistence*, S. 31–41, Springer-Verlag, Berlin, 1988

[Clamen 91] Stewart M. Clamen: Data Persistency in Programming Languages, A Survey. Bericht CMU-CS-91-155, Carnegie Mellon University, Pittsburgh, 1991

[CLtLII] Guy L. Steele Jr.: Common LISP the Language, Second Edition. Digital Press, Bedford, Mass., 1990

[Codd 79] E. F. Codd: Extending The Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, Dezember 1979

[Fernandez et al. 89] Mary F. Fernandez, Stanley B. Zdonik: Transaction Groups: A Model for Controlling Cooperative Transactions. John Rosenberg, David Koch (Hrsg.): *Persistent Object Systems, Newcastle, Australia 1989*, S. 341–350, Springer-Verlag, Berlin, 1989

[Ford et al. 88] Steve Ford, John Joseph, David E. Langworthy, David F. Lively, Girish Pathak, Edward R. Perez, Robert W. Peterson, Diana M. Sparacin, Satish M. Thatte, David L. Wells, Sanjive Agarwala: ZEITGEIST: Database Support for Object-Oriented Programming. K. R. Dittrich (Hrsg.): *Advances in Object-Oriented Database Systems, Bad Minster am Stein-Ebernburg, Deutschland 1988*, S. 23–42, Springer-Verlag, Berlin, 1988

[Goldberg et al. 89] Adele Goldberg, David Robson: Smalltalk-80: The Language. Addison-Wesley, Reading, Mass., 1989

[Gray et al. 93] Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Mateo, Ca., 1993

[Heiler et al. 89] Sandra Heiler, Barbara Blaustein: Generating and Manipulating Identifiers for Heterogeneous, Distributed Objects. John Rosenberg, David Koch (Hrsg.): *Persistent Object Systems, Newcastle, Australia 1989*, S. 235–247, Springer-Verlag, Berlin, 1989

[Itasca 91] Itasca Systems, Inc.: ITASCA Distributed Object Database Management System, Technical Summary. Itasca Systems, Inc., Minneapolis, 1991

[Jessen et al. 87] Eike Jessen, Rüdiger Valk: Rechensysteme. Springer-Verlag, Berlin, 1987

[Kaehler et al. 90] Ted Kaehler, Glenn Krasner: LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems. Stanley B. Zdonik, David Maier (Hrsg.): *Readings in Object-Oriented Database Systems*, S. 298–307, Morgan Kaufmann Publishers, San Mateo, Ca., 1990

[Keene 89] Sonya E. Keene: Object-oriented Programming in Common LISP: A Programmer's Guide to CLOS. Addison-Wesley, Reading, Mass., 1989

[Kernighan et al. 88] Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language, Second Edition. Prentice Hall, Englewood Cliffs, N.J., 1988

[Khoshafian et al. 90] Setrag N. Khoshafian, George P. Copeland: Object Identity. Stanley B. Zdonik, David Maier (Hrsg.): *Readings in Object-Oriented Database Systems*, S. 37–46, Morgan Kaufmann Publishers, San Mateo, Ca., 1990

[Kim et al. 89] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Darrell Woelk: Features of the ORION Object-Oriented Database System. Won Kim, Frederick H. Lochovsky (Hrsg.): *Object-Oriented Concepts, Databases, and Applications*, S. 251–282, ACM Press, New York, 1989

[Kirschke 94] Heiko Kirschke: Persistent LISP Objects: User's Guide, Reference Manual. Fachbereich Informatik, Universität Hamburg, 1994

[Klaus et al. 76] Georg Klaus, Manfred Buhr (Hrsg.): Philosophisches Wörterbuch, 12. Auflage. Verlag das europäische Buch, Berlin, 1976

[Maier et al. 87] David Maier, Jacob Stein: Development and Implementation of an Object-Oriented DBMS. Bruce Shriver, Peter Wegner (Hrsg.): *Research Directions in Object-Oriented Programming*, S. 355–392, MIT Press, Cambridge, Mass., 1987

[man mmap] UNIX Manualseiten zu `mmap`

[man socket] UNIX Manualseiten zu `socket`

[Matthes 92a] F. Matthes: The Database Programming Language DBPL. Rationale and Report. Bericht FBI-HH-B-158/92, Fachbereich Informatik, Universität Hamburg, 1992

[Matthes et al. 92b] F. Matthes, A. Rudloff, J. Schmidt, K. Subieta: The Database Programming Language DBPL. User and System Manual. Bericht FBI-HH-B-159/92, Fachbereich Informatik, Universität Hamburg, 1992

[Matthes et al. 92c] F. Matthes, J. Schmidt: Definition of the Tycoon Laguage TL – A Preliminary Report Bericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, 1992

[Matthes et al. 93] F. Matthes, J. Schmidt: System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. P. P. Spies (Hrsg.): *Proceedings Euro-ARCH '93*, S. 301–317, Informatik Aktuell, Springer-Verlag, Berlin, 1993

[Morrison et al. 89a] R. Morrison, A.L. Brown, R. Carrick, R. Connor, A. Dearle: The Napier Type System. John Rosenberg, David Koch (Hrsg.): *Persistent Object Systems, Newcastle, Australia 1989*, S. 3–18, Springer-Verlag, Berlin, 1989

[Morrison et al. 89b] R. Morrison, A.L. Brown, R. Connor, A. Dearle: The Napier88 Reference Manual. Bericht PPRR 77-89, Universities of Glasgow and St. Andrews, 1989

[Müller 91] Rainer Müller: Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1991

[Nikhil 88] Rishiyur S. Nikhil: Functional Databases, Functional Languages. Malcolm P. Atkinson, Peter Buneman, Ronald Morrison (Hrsg.): *Data Types and Persistence*, S. 51–67, Springer-Verlag, Berlin, 1988

[Norvig 92] Peter Norvig: Paradigms of Artificial Intelligence Programming: Case Study in Common LISP. Morgan Kaufmann Publishers, San Mateo, Ca., 1992

[O'Neil 94] Patrick O'Neil: Database Principles, Programming, Performance. Morgan Kaufmann Publishers, San Francisco, Ca., 1994

[Paepke 91a] Andreas Paepke: PCLOS Reference Manual. HP Laboratories, Palo Alto, Ca., 1991

[Paepcke 91b] Andreas Paepke: User-Level Language Crafting: Introducing the CLOS Metaobject Protocol. HP Laboratories Technical Report HPL-91-169, 1991

[Paton et al. 88] Norman W. Paton, Peter M. D. Gray: Identification of Database Objects by Key. K. R. Dittrich (Hrsg.): *Advances in Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, Deutschland 1988*, S. 280–285, Springer-Verlag, Berlin, 1988

[Rhein et al. 93] John Rhein, Greg Kemnitz (Hrsg.): The POSTGRES User Manual, Version 4.1. Bericht, University of California, Berkeley, 1993

[Richardson et al. 89] Joel E. Richardson, Michael J. Carey: Implementing Persistence in E. John Rosenberg, David Koch (Hrsg.): *Persistent Object Systems, Newcastle, Australia 1989*, S. 175–199, Springer-Verlag, Berlin, 1989

[Rowe 87] Lawrence A. Rowe: A Shared Object Hierarchy. Bericht, University of California, Berkeley, 1987

[Schank et al. 90] Patricia Schank, Joe Konstan, Chung Liu, Lawrence A. Rowe, Steve Seitz, Brian Smith: PICASSO Reference Manual, Version 1.0. Bericht, University of California, Berkeley, 1990

[Schewe et al. 92] Klaus-Dieter Schewe, Bernhard Thalheim, Ingrid Wetzel: Foundations of Object Oriented Database Concepts. Bericht FBI-HH-B-157/92, Fachbereich Informatik, Universität Hamburg, 1992

[Schlageter et al. 83] G. Schlageter, W. Stucky: Datenbanksysteme: Konzepte und Modelle. 2. Auflage, Teubner, Stuttgart, 1983

[Schmidt 77] Joachim W. Schmidt: Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977

[Schmidt et al. 80] Joachim W. Schmidt, Manuel Mall: PASCAL/R Report. Bericht IFI-HH-B-66/80, Fachbereich Informatik, Universität Hamburg, 1980

[Schmidt 82] Heinrich Schmidt: Philosophisches Wörterbuch, 21. Auflage. Stuttgart, 1982

[Singhal et al. 93] Vivek Singhal, Sheetal V. Kakkad, Paul R. Wilson: Texas: Good, Fast, Cheap Persistence for C++. *OOPS Messenger*, 4(2):145–147, April 1993

[Skarra et al. 87] Andrea H. Skarra, Stanley B. Zdonik: Type Evolution in an Object-Oriented Database, Bruce Shriver, Peter Wegner (Hrsg.): *Research Directions in Object-Oriented Programming*, S. 393–415, MIT Press, Cambridge, Mass., 1987

[Snape 93] Persönliche Kommunikation über E-Mail mit Guy Snape, Harlequin Limited: Re: [kirschke: memory representation in LispWorks]. Text in Datei `plob/mail/binary-repr`, 9. November 1993

[Snape 94] Persönliche Kommunikation über E-Mail mit Guy Snape, Harlequin Limited: Re: Metaobject-Protocol support for LispWorks CLOS. Text in Datei `plob/mail/mop-support`, 22. April 1994

[St. Clair 93] Bill St. Clair: WOOD (William's Object Oriented Database), Datei `cambridge.apple.com:/pub/MCL2/contrib/wood.doc`, 1993

[Stonebraker et al. 86a] Michael Stonebraker, Lawrence A. Rowe: The Design of POSTGRES. Bericht, University of California, Berkeley, 1986

[Stonebraker 86b] Michael Stonebraker: Inclusion of New Types in Relational Data Base Systems. Bericht, University of California, Berkeley, 1986

[Stonebraker et al. 89] Michael Stonebraker, Lawrence A. Rowe, Michael Hirohama: The Implementation of POSTGRES. Bericht, University of California, Berkeley, 1989

[Thatte 90] Satish Thatte: Persistent Memory: Merging AI-Knowledge and Databases. Stanley B. Zdonik, David Maier (Hrsg.): *Readings in Object-Oriented Database Systems*, S. 242–250, Morgan Kaufmann Publishers, San Mateo, Ca., 1990

[Uhl et al. 93] Jürgen Uhl, Dietmar Theobald, Bernhard Schiefer, Michael Ranft, Walter Zimmer, Jochen Alt: The Object Management System of STONE. Bericht aus `ftp.fzi.de:/pub/OBST/OBST3-3.3`, Forschungszentrum Informatik, Karlsruhe, 1993

[Valduriez et al. 89] Patrick Valduriez, Georges Gardarin: Analysis and Comparison of Relational Database Systems. Addison-Wesley, Reading, Mass., 1989

[Wahrig 68] Gerhard Wahrig: Deutsches Wörterbuch. Bertelsmann Lexikon-Verlag, Gütersloh, 1968

[Wand 89] Yair Wand: A Proposal for a Formal Model of Objects. Won Kim, Frederick H. Lochovsky (Hrsg.): *Object-Oriented Concepts, Databases, and Applications*, S. 537–559, ACM Press, New York, 1989

[Wegner 90] Peter Wegner: Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, August 1990

[Wensel 93] S. Wensel (Hrsg.): The POSTGRES Reference Manual, Version 4.1. Bericht M88/20, University of California, Berkeley, 1993

[Winston et al. 89] Patrick Henry Winston, Berthold Klaus Paul Horn: LISP, Third Edition. Addison-Wesley, Reading, Mass., 1989

[Wirth 83] Niklaus Wirth: Algorithmen und Datenstrukturen, 3. Auflage. B. G. Teubner, Stuttgart, 1983

[Zdonik et al. 90] Stanley B. Zdonik, David Maier: Fundamentals of Object-Oriented Databases. Stanley B. Zdonik, David Maier (Hrsg.): *Readings in Object-Oriented Database Systems*, S. 1–32, Morgan Kaufmann Publishers, San Mateo, Ca., 1990

There are *(sic!)* a finite number of jokes in the universe.

— David Byrne: Stop Making Sense