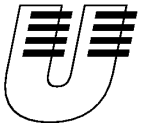

UNIVERSITÄT
KOBLENZ · LANDAU

Institut für
Wirtschaftsinformatik

Fachbereich Informatik
Universität Koblenz-Landau

JÜRGEN JUNG

KONZEPTE OBJEKTORIENTIERTER DATENBANKEN: KONKRETISIERT AM BEISPIEL GEMSTONE

Februar 2001

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 24

UNIVERSITÄT KOBLENZ · LANDAU

Institut für Wirtschaftsinformatik
Fachbereich Informatik
Universität Koblenz-Landau

JÜRGEN JUNG

KONZEPTE OBJEKTORIENTIERTER DATENBANKEN: KONKRETISIERT AM BEISPIEL GEMSTONE

Februar 2001

Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr. 24

Die Arbeitsberichte des Instituts für Wirtschaftsinformatik dienen der Darstellung vorläufiger Ergebnisse, die i.d.R. noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar.

The "Arbeitsberichte des Instituts für Wirtschaftsinformatik" comprise preliminary results which will usually be revised for subsequent publications. Critical comments would be appreciated by the authors.

Anschrift des Verfassers
Address of the author:

Dipl. Inf. Jürgen Jung
Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
D-56075 Koblenz

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgegeben von / Edited by:

Prof. Dr. Ulrich Frank
Prof. Dr. J. Felix Hampe
Prof. Dr. Gerhard Schwabe

Bezugsquelle / Source of Supply:

Institut für Wirtschaftsinformatik
Universität Koblenz-Landau
Rheinau 1
56075 Koblenz

Tel.: 0261-287-2520
Fax: 0261-287-2521
Email: iwi@uni-koblenz.de
WWW: http://www.uni-koblenz.de/~iwi

Institut für Wirtschaftsinformatik
Fachbereich Informatik
Universität Koblenz-Landau

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Thema dieses Berichtes sind die Konzepte objektorientierter Datenbanksysteme sowie deren Konkretisierung am Beispiel des objektorientierten Datenbank-Management-Systems GemStone. Hintergrund für die Erstellung fieses Berichtes ist die Entwicklung einer Literaturverwaltung im Rahmen des Projektes MORE (Migrating Objects in a Research Environment). Die persistente Datenhaltung erfolgt bei der Literaturverwaltung mit dem objektorientierten Datenbanksystem GemStone. Der Bericht arbeitet beschränkt sich aufgrund der in MORE gesammeten Erfahrungen nicht nur auf allgemeine Datenbankkonzepte, beurteilt GemStone anhand dieser Konzepte.
Dieses einleitende Kapitel beginnt zunächst in Abschnitt 1.1 mit einen Überblick über die Konzeption und die Architektur der MORE-Literaturverwaltung. Anschließend bietet Abschnitt 1.2 eine Motivation für die Beschäftigung mit der Thematik objektorientierter Datenbanken. Abschnitt 1.3 arbeitet die terminologische Grundlagen von Datenbanken und insbes. objektorientierter Datenbanken auf. Anschließend folgt in Abschnitt 1.4 eine Einführung in die Entwicklung von GemStone. Das Kapitel endet mit einer Beschreibung des Aufbaus dieses Berichtes und den Schwerpunkten der einzelnen Kapitel in Abschnitt 1.5.

1.1. Überblick über die MORE-Literaturverwaltung

Die Entwicklung der MORE-Literaturverwaltung nahm zu Beginn des Jahres 1997 als eigenständige Teilaufgabe im Rahmen des Projektes MORE ihren Anfang. Die Grundidee von MORE ist die Realisierung eines hochintegrierten objektorientierten Informationssystems zur Verwaltung universitärer Informationen. Mit Hilfe dieses Systems soll eine Basis zur Unterstützung der kooperativen Arbeit in Forschung und Lehre am Institut für Wirtschaftsinformatik geschaffen werden. Ein fundamentaler Bestandteil dieser Arbeit ist die intensive Nutzung und Begutachtung von Literatur. Einige grundlegende Anforderungen an den Entwurf der Literaturverwaltung ergeben sich aus den Bedürfnissen der Mitarbeiter des Instituts und der Konzeption von MORE:

- Die Literatur wird auf einem semantisch angemessenen Niveau erfaßt.
- Die Möglichkeit für Anmerkungen zu Literaturtiteln ist vorhanden.
- Titel können durch Projekte gruppiert verwaltet werden.
- Die Literaturverwaltung muß in MORE integriert werden.
- Die Entwicklung erfolgt streng objektorientiert.

Im wissenschaftlichen Umfeld existieren verschiedene Typen von Literatur mit teilweise unterschiedlichen Eigenschaften. Eine Zeitschrift unterscheidet sich bspw. von einer Monographie dadurch, daß sie mehrere Ausgaben umfaßt, die wiederum eine Menge von Beiträgen enthalten. Sie hat auch üblicherweise einen oder mehrere Herausgeber, wohingegen eine Monographie von Verfassern erstellt wird. Insgesamt werden in der MORE-Literaturverwaltung 19 verschiedene Typen von Literatur unterschieden, woraus sich aufgrund der Generalisierung[1] gemeinsamer Eigenschaften eine Klassenhierarchie mit 28 Klassen ergibt. Hinzu kommt noch eine große Anzahl von Klassen zur angemessenen Darstellung der an der Literatur beteiligten Objekte. Hierzu gehören neben den bereits aufgeführten Anmerkungen und Projekten auch unterschiedliche Rollen[2], die einzelne Personen oder Organisationen im Zusammenhang mit Literatur einnehmen können.
Einen Überblick über die Systemarchitektur der MORE-Literaturverwaltung[3] vermittelt Abbildung 1. Sie ist in MORE eingebettet und nutzt die Dienste weiterer MORE-Komponenten, wie die Benutzer- oder Personenverwaltung. Ihre Kernfunktionalität wird von der Literaturbasis gebildet, die der Ver-

[1] Die Begriffe Generalisierung, Klasse und Klassenhierarchie werden im Rahmen der Vorstellung objektorientierter Grundlagen in Kapitel 2 erörtert.

[2] Zur Realisierung unterschiedlicher Rollen wird das Konzept der Delegation benutzt (siehe hierzu auch [FrHa97]).

[3] Weitere Informationen über die Architektur der MORE-Literaturverwaltung findet man in [Jung98]. Haase diskutiert in [Haase98] anhand von Entwurfsmustern die Aspekte der Konstruktion von Objekten und geht dabei auch auf die Struktur der Verwaltungskomponenten von MORE ein.

waltung von Literatur und der damit direkt verbundenen Objekte dient. Die Komponente für die Suche über den Literaturbestand ist in einem eigenen Subsystem gekapselt, wodurch eine Separierung der Suchaspekte von der Literatur erreicht wird. Diese Trennung begründet sich dadurch, daß die Suche keine inhärente Eigenschaft der Literatur ist. Außerdem gewährleistet sie eine unabhängige Entwicklung mehrerer Sucher. Jeder Sucher wird durch die Suchkomponente verwaltet und kann eine individuelle Suchstrategie realisieren.

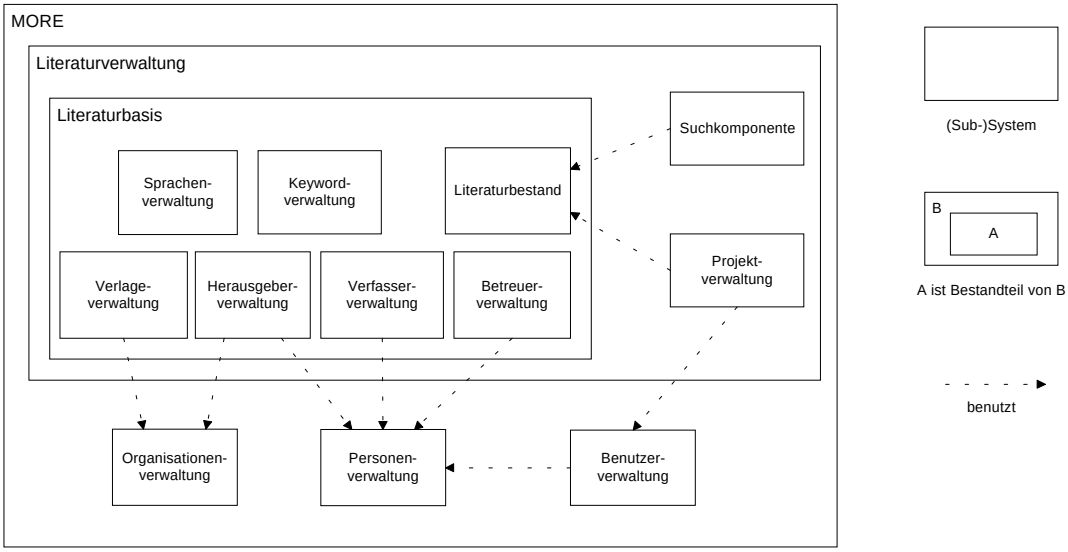

Abbildung 1: Systemarchitektur der MORE-Literaturverwaltung

Unter dem Gesichtspunkt der konsequent objektorientierten Softwareentwicklung werden in den Phasen der Modellierung und Implementierung von MORE ausschließlich objektorientierte Werkzeuge und Prinzipien benutzt resp. angewandt. Hierzu zählen die Modellierungssprachen MEMO-OML[4] und UML aber auch die OPL Smalltalk in Form der Entwicklungsumgebung VisualWorks. Die dauerhafte Speicherung der Objekte ist in zwei Arten vorhanden. Eine einfache Art wird über die Speicherungsmechanismen von Smalltalk bereitgestellt (siehe Abschnitt 2.2.1). Hierdurch kann zum einen das System schon in einem frühen Stadium der Entwicklung ohne die Anbindung einer Datenbank getestet werden. Zum anderen kann jeder Benutzer die fertige Literaturverwaltung als Einzelplatzversion ohne GemStone nutzen. Eine Datenbank, die erhöhte Anforderungen an die Leistung des Computers stellt, muß nicht installiert werden. Für den Einsatz am Institut reicht eine Einbenutzerversion jedoch nicht aus, da neben der Dauerhaftigkeit der Objekte auch der konkurrierende Zugriff mehrerer Benutzer koordiniert werden muß. Deshalb wird für die zweite Form der Realisierung von Persistenz GemStone eingesetzt. Der Vorteil der Mehrbenutzerversion ist der Aufbau einer gemeinsamen Literaturbasis zur Unterstützung der kooperativen Arbeit der Mitarbeiter am Institut.
Die Entscheidung für ein objektorientiertes Datenbanksystem und dabei insbes. GemStone wurde unter der Berücksichtigung mehrere Parameter gefällt. Durch die Einheitlichkeit des Paradigmas können die Objekte der Literaturverwaltung ohne Verlust an Strukturinformationen in der gemeinsamen Informationsbasis verwaltet werden. Relationale Datenbanken erfordern eine Umsetzung der Objekte in ihre atomaren Bestandteile, damit sie in den Zeilen einer Tabelle abgelegt werden können. Hierbei ist eine verlustarme Abbildung zwischen den Paradigmen mit einem hohen Aufwand verbunden. Auch als objektrelational postulierte Datenbanken unterstützen nicht alle Konzepte der Objektorientierung (vgl. [JoDi98], S. 25 ff.). Ein weiteres Kriterium zur Auswahl einer Datenbank für das Informationssystem MORE ist das Vorhandensein einer Schnittstelle zu Smalltalk. Die Wahl ist auf GemStone gefallen, da es nicht nur eine solche Schnittstelle bereitstellt, sondern Smalltalk auch die Sprache der Datenbank ist. Aufgrund seiner Architektur ermöglicht GemStone überdies die Ausführung von in Smalltalk geschriebenen Methoden in der Datenbank, wodurch eine Verteilung der Anwendungslast zwischen dem Anwendungsprogramm und der Datenbank realisiert werden kann. Darüber

[4] Die OML (Object Modelling Language) ist eine Sprache zur Beschreibung von Objekten in MEMO.

hinaus unterstützt GemStone spezifische Datenbankkonzepte, wie Mehrbenutzerbetrieb, Transaktionsverarbeitung und Autorisierung (vgl. [Bre89]).

1.2. Objektorientierte Datenbanken

Objektorientierte Konzepte haben in den vergangenen Jahren immer mehr an Bedeutung für die Softwareentwicklung gewonnen (vgl. [Fra97a], S. 5). Dies wird nicht nur anhand der großen Anzahl objektorientierter Modellierungsmethoden, die seit Beginn der neunziger Jahre entwickelt und publiziert wurden, deutlich. Beispiele hierfür sind MEMO (Multi Perspective Enterprise Modelling)[5], OMT (Object Modeling Technique)[6], die Booch-Methode[7] oder UML (Unified Modeling Language)[8]. Die ersten Ansätze der Objektorientierung liegen jedoch im Bereich der Programmiersprachen. Erste Vertreter objektorientierter Programmiersprachen (OPL) sind Simula und Smalltalk. Eine weite Verbreitung objektorientierter Konzepte in Programmiersprachen wurde durch die Sprachen C++ und Java gefördert (vgl. [CSS99], S. 16). C++ stellt eine Erweiterung von C dar und ist somit eine hybride Sprache, die neben objektorientierten Konstrukten auch prozedurale Elemente enthält. Java ist C++ zwar syntaktisch ähnlich, unterscheidet sich jedoch davon durch seine ausschließlich objektorientierten Konzepte.

Im Zusammenhang mit einer OPL hat der Einsatz einer objektorientierten Modellierungsmethode den Vorteil, daß in allen Phasen des Entwicklungsprozesses die gleichen Konzepte genutzt werden können. Schon zu Beginn wird der für die Entwicklung relevante Problembereich mit den Abstraktionsmöglichkeiten der Objektorientierung konzeptualisiert. Das hierbei erstellte Modell der Anwendungsdomäne wird im weiteren Verlauf schrittweise verfeinert bis letztlich das ausführbare Programm in einer OPL implementiert wird (vgl. [RBP+93], S. 6). In der logischen Konsequenz bietet sich unter diesem Gesichtspunkt der Einsatz einer objektorientierten Datenbank zur permanenten Speicherung und Verwaltung von Objekten an. Eine Datenbank ist in dem entwickelten Anwendungssystem für die dauerhafte Aufbewahrung von Objekten zuständig und regelt den quasi gleichzeitigen Zugriff mehrerer Benutzer.

Entgegen dem Trend zu objektorientierten Modellierungs- und Programmiersprachen, ist die Akzeptanz für objektorientierte Datenbanken eher gering. Die in diesem Bereich immer noch vorherrschende Technologie ist die konventioneller Datenbanken, wie bspw. relationale Systeme. Die Gründe hierfür sind mannigfaltig. So sind die Hersteller **relationaler Datenbanken** schon länger am Markt tätig und können auf eine mehrjährige Erfahrung zurückblicken. Dies manifestiert sich in der Reife und somit der Leistungsfähigkeit und der Zuverlässigkeit relationaler Datenbanken. Ihre Popularität äußert sich in einer breiten Basis von eingesetzten Systemen, deren Austausch gegen alternative Technologien einen enormen finanziellen Aufwand bedeutet. Außerdem sind ihre Konzepte durch die relationale Algebra theoretisch fundiert (vgl. [STS97], S. 17). Die Daten werden in Tabellen gespeichert, deren Zeilen jeweils ein Datenobjekt enthalten. In der relationalen Algebra spricht man hierbei auch von Relationen und Tupeln. Jedes dieser Tupel setzt sich aus mehreren atomaren Datenwerten zusammen (vgl. [ElNa94], S. 187). Ein weiterer nicht zu unterschätzender Vorteil ist das Vorhandensein eines allgemein akzeptierten Standards für relationale Datenbanken. Dieser Standard ist unter dem Namen der hierdurch normierten Datenbanksprache SQL (Standard Query Language) bekannt (vgl. [Vos91], S. 221 ff.).

Den Anforderungen der objektorientierten Softwareentwicklung begegnen die Vertreter der relationalen Technologie mit der Erweiterung ihrer Systeme um objektorientierte Konzepte. Solche Systeme sind auch unter dem Namen **objektrelationale Datenbanken** bekannt (vgl. [Luf99]). Zur Zeit herrscht aber bei den aktuellen Produkten der Datenbankhersteller keine einheitliche Interpretation dieses Begriffs. Dies äußert sich zum einen in dem unterschiedlichen Umfang adaptierter objektorien-

[5] Siehe [Fra94] und [Fra97b].
[6] Siehe [RBP+93].
[7] Siehe [Boo94].
[8] Die UML stellt in diesem Zusammenhang eine Ausnahme dar, da sie nur eine Sprache zur Modellierung von Softwaresystemen ist und keine Methode. Der zu der Notation gehörige Prozeß ist der RUP (Rational Unified Process). Eine Beschreibung der UML findet man in [RJB99] und der RUP wird in [JBR99] und [Kru99] erläutert.

tierter Konzepte und zum anderen auch in der heterogenen Realisierung benutzerdefinierter Funktionen (vgl. [JoDi98], S. 23 ff.). Es bleibt abzuwarten, ob der kürzlich verabschiedete Standard **SQL:1999** eine Vereinheitlichung objektrelationaler Systeme bewirkt (vgl. [Luf99], S. 290).

Objektorientierte Datenbanken sind seit Ende der achtziger Jahre auf dem Markt verfügbar. Sie sollen die Bedürfnisse solcher Anwendungen decken, für deren Entwicklung die einfache Struktur des relationalen Modells nicht hinreichend ist (vgl. [DiGe97], S. 275 und [Hoh98], S. 22). In diesem Zusammenhang oft genannte Beispiele sind Programme aus dem ingenieurwissenschaftlichen Bereich wie CAD/CAM-Systeme[9], aber auch Geo-Informationssysteme oder Softwareentwicklungswerkzeuge. Für die Abbildung der in diesen Systemen vorhandenen Objekte reichen die einfachen Konzepte des relationalen Datenmodells oftmals nicht aus, da hierbei Informationen verloren gehen oder durch das flache relationale Schema unnatürliche Abstraktionen entstehen können (vgl. [STS97], S. 39 ff. und [Luf99], S. 288). Demgegenüber können komplexe Objekte in einer objektorientierten Datenbank direkt gespeichert werden. Der spätere Zugriff auf diese Objekte erfolgt dabei nicht nur über mengenorientierte Anfragen, wie sie in relationalen Systemen üblich sind, sondern durch Navigation über die vernetzten Objekte (vgl. [STS97], S. 41). Des weiteren ermöglichen objektorientierte Datenbanken die bereits skizzierte Durchgängigkeit über den Prozeß der objektorientierten Softwareentwicklung. Sie zeichnen sich besonders durch das Fehlen eines semantischen Bruchs zwischen der Programmier- und Datenbanksprache aus. Saake et al. sprechen bei einem solchen semantischen Bruch auch von einem „**impedance mismatch**“ (vgl. [STS97], S. 50). Die Beseitigung eines solchen „impedance mismatch“ ermöglicht eine enge Kopplung zwischen Anwendung und Datenbank und somit einen hohen Grad an Integration. Andererseits kann dies aber auch als Nachteil angesehen werden, da eine enge Kopplung zwischen Anwendung und Datenbank oftmals auch eine technologische und wirtschaftliche Bindung an eine bestimmte Datenbank bzw. dessen Hersteller bedeutet. Beim Einsatz einer objektorientierten Datenbank hat dies die Konsequenz, daß die Austauschbarkeit eines Datenbanksystems gegen ein anderes nicht mehr gegeben ist.

Die Sprachunabhängigkeit wird oftmals auch als Vorteil relationaler Systeme gesehen, da hier durch die normierte Sprache SQL eine einheitliche Basis zum Zugriff auf eine Datenbank existiert. Diesem Umstand begegneten im Jahr 1991 einige Hersteller objektorientierter Datenbanken mit der Gründung der **Object Database Management Group** (ODMG). Das Ziel dieses Konsortiums ist die Schaffung eines gemeinsamen Industriestandards für objektorientierte Datenbanken, der erstmals unter dem Namen ODMG-93 veröffentlicht wurde (vgl. [Cat94a]). Er wurde im Laufe der Zeit weiterentwickelt und liegt mittlerweile in der Version 2.0 vor (vgl. [Cat94b] und [CaBa97])[10]. Parallel dazu wurde er auch kritisch diskutiert, was sowohl den Standard als auch die Qualität der Publikationen anbelangt (vgl. [Kim94], [Man94], S. 147 ff., [Heu97], S. 460 f. und [Pra99], S. 60 f.). Hinzu kommt noch, daß heute nur sehr wenige objektorientierte Datenbanksysteme den Standard unterstützen (vgl. [STS97], S. 522 und [JoDi98], S. 26). Trotzdem werden die Bemühungen der ODMG von vielen Leuten begrüßt (vgl. [Atw94], [MeWü97], S. 70 ff. und [STS97], S. 522). Saake et al. betonen hierzu, daß die strikte Befolgung des Standards bei der Entwicklung von Datenbankanwendungen und seine Einhaltung seitens der Hersteller die Portabilität steigern können (vgl. [STS97], S. 522). Hierdurch hätte man eine verläßliche Grundlage für den wirtschaftlichen Einsatz objektorientierter Datenbanken und könnte somit die momentane Verunsicherung der Anwender verringern.

Ein erster Versuch zur Definition einer gemeinsamen Grundlage für objektorientierte Datenbanken wurde im Jahr 1989 von einigen Wissenschaftlern, die im Bereich von Datenbanken und dabei insbesondere auch objektorientierter Systeme tätig waren, unternommen. Sie legen in ihrem „Object-Oriented Database System Manifesto“[11] Eigenschaften fest, die ein objektorientiertes Datenbanksystem (ODBS) erfüllen soll. Dabei unterteilen die Verfasser die Anforderungen in obligatorische, optionale und strittige Eigenschaften. Die obligatorischen Eigenschaften („The Golden Rules“) sind solche, die jedes objektorientierte Datenbanksystem erfüllen muß und die Kernfunktionalität darstellen. Daneben listen sie noch einige optionale Eigenschaften auf („The Goodies“), die zwar nicht den Kern eines ODBS ausmachen aber die Funktionalität eines ODBS verbessern. Eigenschaften, über die

[9] Die Abkürzung CAD und CAM stehen für „Computer Aided Design“ bzw. „Computer Aided Manufacturing“.
[10] Gegen Ende der Anfertigung dieser Arbeit ist die Version 3.0 veröffentlicht worden.
[11] Siehe [ABD+89].

kein gemeinsamer Konsens getroffen werden konnte, fassen sie als strittige Eigenschaften („Open Choices“) zusammen. Ein objektorientiertes Datenbanksystem ist gemäß Atkinson et al. ein Datenbank-Management- und gleichzeitig ein objektorientiertes System (vgl. [ABD+89], S. 41). Als Datenbank-Management-System muß es Konzepte zur Realisierung von Persistenz, zur Verwaltung des Sekundärspeichers, Transaktionsverarbeitung, Wiederherstellbarkeit im Fehlerfall und eine Ad-hoc-Anfragemöglichkeit bereitstellen. Zusätzlich muß es als objektorientiertes System Konzepte, wie komplexe Objekte, Objektidentität, Kapselung, Typen oder Klassen, Vererbung, Erweiterbarkeit und eine berechenbarkeitsvollständige Sprache unterstützen.

1.3. Grundlegende Terminologie

In Anlehnung an Saake et al. ist eine **Datenbank** (DB) eine strukturierte Sammlung von dauerhaft zu speichernden Daten, die Informationen über einen speziellen Sachverhalt des zu modellierenden Ausschnitts einer realen Anwendungsdomäne repräsentieren (vgl. [STS97], S. 2). Innerhalb einer DB werden neben diesen Daten zusätzlich auch die Metadaten verwaltet, d.h. die Fakten, die den Aufbau gültiger Informationen in der Datenbank beschreiben (vgl. [MeWü97], S. 1). Dazu gehören die sog. Schemadaten zur Beschreibung der Struktur der zu verwaltenden Daten und administrative Informationen, wie bspw. die Benutzer einer DB. Die Verwaltung dieser Informationen wird durch ein **Datenbank-Management-System** (DBMS) in Form einer Software gewährleistet, die das Hinzufügen neuer Daten, das Ändern vorhandener Daten und die Abfrage der gespeicherten Informationen realisiert (vgl. [STS97], S. 2 oder [MeWü97], S. 1). Der Begriff **Datenbanksystem** (DBS) bezeichnet in diesem Zusammenhang die Kombination eines DBMS mit einer oder mehreren DB, die von diesem DBMS verwaltet werden (vgl. [STS94], S. 2, [MeWü97], S. 1 oder [Vos94], S. 30]).
Grundlage für die Beschreibung der zu verwaltenden Daten ist das **Datenbankmodell**[12] (DM) des DBS. Nach Saake et al. ist dies ein System von Konzepten, das die Syntax und Semantik von Datenbankschemata festlegt (vgl. [STS97], S. 2). Im Falle von objektorientierten Datenbanken ist dies das objektorientierte Datenbankmodell, oder auch **Objektdatenbankmodell** (OM). Deswegen wird im folgenden ausschließlich von objektorientierten Datenbanken (ODB), einem objektorientierten Datenbank-Management-System (ODBMS) und einem objektorientierten Datenbanksystem (ODBS) gesprochen[13] (vgl. [STS97], S. 9 f.). Das OM einer ODB realisiert objektorientierte Konzepte zur Beschreibung von Schemata einer ODB. Zur Formulierung eines Schemas dient somit eine objektorientierte **Datendefinitionssprache** (DDL für den engl. Ausdruck data definition language). Mit Hilfe dieser Sprache werden die zu verwaltenden Objekte durch ihre Klassen und deren Beziehungen beschrieben. Konkrete Ausprägungen, d.h. aus diesem Schema instanzierte ODB, können durch die sog. **Datenmanipulationssprache** (DML; engl. data manipulation language) bearbeitet werden. Daneben existiert auch noch die sog. **Anfragesprache** (QL; engl. query language), die der selektiven Anfrage auf eine Menge von Daten oder Objekten dient[14].

1.4. GemStone

Das ODBMS GemStone wird seit 1987 von dem Unternehmen Servio Logic entwickelt und vertrieben. Servio Logic hat sich mittlerweile nach dem Namen seines ersten und einzigen Produktes in GemStone Inc. umbenannt. Ziel der Entwicklung von GemStone war ein DBMS mit einem objektorientierten Datenmodell und einer objektorientierten Datendefinitions-, Manipulations- und Anfragesprache. Als Grundlage für das Datenmodell und die Datenbanksprache fiel die Wahl auf die objektorientierte Programmiersprache Smalltalk-80[15], weshalb sich die Entwicklung durch den Titel einer Veröffentlichung von Copeland und Mayer charakterisieren läßt: „Making Smalltalk a Database Sy-

[12] In der deutschsprachigen Literatur findet man hier auch häufig den synonym genutzten Begriff „Datenmodell“ (vgl. [MeWü97] S. 2, [Dit98], S. 20 oder [Vos94], S. 9).

[13] Oftmals wird in der Literatur auch von OODB, OODBMS oder OODBS gesprochen, wobei die Begriffe mit Doppel-O synonym zu den entsprechenden Abkürzungen mit nur einem O verwendet werden (vgl. bspw. [MeWü97], S. 5 oder [Dit98], S. 20).

[14] Die drei Sprachtypen DDL, DML und QL werden ausführlicher in [ElNa94], S. 29 f., [STS97], S. 3 oder auch [Vos94], S. 26 ff. vorgestellt.

[15] Der Zusatz „-80“ steht für den Stand der Normierung von Smalltalk im Jahr 1980.

stem“ (vgl. [CoMa84])[16]. GemStone gehört somit zu der Menge von ODBMS, die von Heuer als Erweiterung objektorientierter Programmiersprachen um Datenbankkonzepte eingestuft wird (vgl. [Heu97] S. 557). Die Sprache Smalltalk wurde aufgrund ihrer reinen Objektorientierung, ihres intensionalen Klassenbegriffs und des Typsystems bevorzugt (vgl. [Bre89], S. 284-286).
In der aktuellen Version[17] 5 präsentiert sich GemStone dem Entwickler als umfangreiches ODBMS mit der zentralen Datenbanksprache Smalltalk[18], das außerdem die erforderlichen Konzepte für den Mehrbenutzerzugriff mehr oder weniger vollständig realisiert. Die in der Sprache Smalltalk implementierten Methoden der Objekte können direkt in der Datenbank gespeichert und ausgeführt werden. Auch sind die datenbankspezifischen Konzepte teilweise durch eigene Klassen direkt in diese Sprache eingebettet. Die Integration in die Anwendungsprogramme wird durch spezielle Schnittstellen für die Implementierungssprachen C, C++, Java und nicht zuletzt auch diverse Smalltalk-Entwicklungsumgebungen realisiert. Durch die Bereitstellung eines rein objektorientierten Datenmodells zeichnet sich speziell die objektorientierte Anwendungsentwicklung durch das Fehlen des „impedance mismatch“ aus. Dies bedeutet, daß sich die objektorientierten Konzepte, wie Objekt, Klasse, Vererbung und Kapselung, nicht nur friktionsarm durch alle Phasen des Entwicklungsprozesses der Software ziehen, sondern in dieser Form auch unmittelbar auf die persistente Datenhaltung abgebildet werden. Insbesondere können Klassen, die in Smalltalk implementiert sind, direkt in die Datenbank übersetzt werden. Hieraus ergibt sich nicht nur eine Elimination des „impedance mismatch“, sondern auch die Verwendung ein und der selben Implementierungssprache für die Entwicklung des Anwendungssystems und die Klassen der Datenbankobjekte.

1.5. Aufbau des Berichts

Das folgende Kapitel 2 widmet sich einem Überblick über vorherrschende Architekturen objektorientierter Datenbanken. Es werden prinzipielle Architekturen vorgestellt und gegeneinander verglichen. Die darauf folgenden Kapitel beschäftigen sich mit der Thematik Persistenz. Kapitel 3 erörtert verschiedene Persistenzmodelle, d.h. wie Objekte persistent gemacht werden. Kapitel 4 beschreibt den Zugriff auf persistente Objekte. Die Thematik des Mehrbenutzerzugriffs, d.h. der Synchronisation von Transaktionen auf eine objektorientierte Datenbank wird in Kapitel 5 behandelt. Durch den Zugriff mehrer Benutzer bedingt gewinnt auch das Thema der Autorisierung an Bedeutung. Diese wird in Kapitel 6 behandelt. Der Bericht schließt mit einer Zusammenfassung und einem Ausblick auf zukünftige Entwicklungen in Kapitel 7 ab. Jedes der Kapitel 3 bis 6 beginnt zunächst mit einem Überblick über eine allgemeine Darstellung der jeweiligen Konzepte. Anschließend werden die konkreten Umsetzungen dieser Konzepte in GemStone vorgestellt und kritisch diskutiert. Vor allem diesen Abschnitten liegen gesammelte Erfahrungen beim Einsatz von GemStone zugrunde.

2. Architekturen objektorientierter Datenbanken

Nach Saake et al. stellt ein DBMS, unabhängig von seinem DM, ein komplexes System dar, das aus mehreren kooperierenden Komponenten besteht (vgl. [STS97], S. 471 f.). Zur Veranschaulichung der einem solchen System inhärenten Komplexität werden seine Komponenten und ihre Beziehungen identifiziert, um sie als **Architektur** des DBMS darzustellen. Die Architektur dient der Beschreibung des Aufbaus eines informationstechnischen Systems anhand seiner Bausteine und deren Interaktionsbeziehungen zur Realisierung des gesamten Systems, in diesem Fall ein ODBMS. Die Architekturen objektorientierter Datenbanken orientieren sich am Client/Server-Prinzip (vgl. [Loo92], [Här95], [HSW97] und [Heu97], S. 415 ff.). Die grundlegende Struktur einer Client/Server-Architektur wird durch zwei Typen von Rechnern gekennzeichnet. Der Server übernimmt dabei die ~~Rolle eines Dienstanbieters. In~~ diesem Fall stellt er den Dienst der Verwaltung einer Datenbank

[16] Eine Einführung in die Thematik des Einsatzes von Smalltalk als Datenbanksprache gibt Almarode in [Alm95a].

[17] Die Versionsnummer 5 gibt nur den Stand der Hauptversion (engl. major release) an. Im Kontext dieses Berichts wird konkret das Release 5.1.4 referenziert.

[18] Mittlerweile existiert auch eine auf Java basierende Version von GemStone. Zur besseren Unterscheidung bezeichnet der Hersteller die Smalltalk-Variante mit GemStone/S und die für Java mit GemStone/J. Da sich dieser Bericht ausschließlich mit GemStone/S beschäftigt, wird vereinfachend der Name GemStone statt GemStone/S benutzt.

anbieters. In diesem Fall stellt er den Dienst der Verwaltung einer Datenbank bereit. Auf diesen kann von mehreren Clients zugegriffen werden (wobei Client und Server nicht notwendigerweise zwei getrennte Rechner sein müssen). Die Kommunikation zwischen dem Server und den auf ihn zugreifenden Clients erfolgt über ein Rechnernetz, an das der Server und die Clients angeschlossen sind (im folgenden auch Kommunikationsnetz oder Kommunikationsinfrastruktur genannt). Den hier vorgestellten Architekturen ist gemeinsam, daß sich das ODBMS auf dem Server und das Anwendungsprogramm mit der Laufzeitumgebung des ODBMS auf dem Client befindet. Die Architekturen unterscheiden sich dabei oft nur in der Verteilung der einzelnen Komponenten auf Client und Server. Hierbei stehen die Aufgaben des Servers normalerweise im Vordergrund, da er das zentrale Element des Systems darstellt. In der historischen Entwicklung haben sich drei Architekturvarianten für ODBMS herausgebildet (vgl. [Loo92], [HSW97], [Heu97] S. 416 ff. und [STS97] S. 471 ff.):

- Anfrageserver
- Objektserver
- Seitenserver

Im folgenden sollen diese drei Architekturen vorgestellt werden, um dabei einen Überblick über die Einsatzmöglichkeiten und Grenzen der drei alternativen Ansätze zu geben. Die Ausführungen konzentrieren sich auf die Kernaspekte der Architekturen. Die Betrachtung der Architekturen objektorientierter Datenbanken ist durch die Tatsache motiviert, daß die Architektur direkten Einfluß auf die Konzeption Datenbankfunktionalität und die Leistungsfähigkeit hat.

2.1. Anfrageserver

Nach Ansicht von Loomis spielt die Architektur des **Anfrageservers**[19] bei objektorientierten Datenbanken eine untergeordnete Rolle und wird hauptsächlich bei der Konzeption (objekt-) relationaler Systeme eingesetzt (vgl. [Loo92], S. 41). Trotzdem existieren heute einige ODBMS, wie bspw. ObjectStore, O_2 oder Jasmine, mit einer zusätzlichen Anfragekomponente (vgl. [Lam91], S. 55, [Ban92], S. 256 ff. und [CAI98], S. 2-2). Die komplette Datenbankfunktionalität ist bei dieser Architektur auf dem Server konzentriert. Er ist dafür zuständig Anfragen zu verarbeiten, Methoden der Objekte auszuführen, Objekte zu verwalten und die Verarbeitung paralleler Zugriffe mehrerer Benutzer (Transaktionsverarbeitung[20]) zu sichern. Er stellt somit einen potentiellen Flaschenhals im System dar. Im Normalfall sind die Ergebnisse der Anfragen, Objekt- oder Datenmengen, die ein innerhalb einer Anfrage formuliertes Kriterium erfüllen. Navigation über Objekte ist nicht vorgesehen (vgl. [STS97], S. 476 ff.).

[19] In der Literatur wird er auch mit database server (vgl. [Loo92]) oder query server (vgl. [HSW97]) bezeichnet.
[20] Die Synchronisation paralleler Zugriff wird ausführlicher in Abschnitt 5 behandelt.

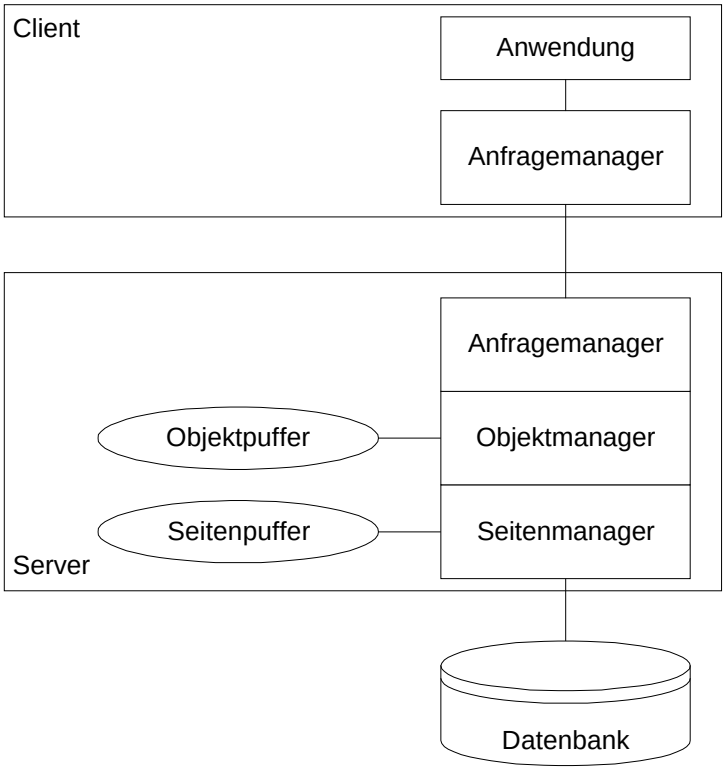

Abbildung 2: Architektur eines Anfrageservers (angelehnt an: [Heu97], S. 419)

Auf dem Client befindet sich bei dieser Architektur nur die laufende Applikation und ein **Anfragemanager**, der die Schnittstelle zwischen dem Anwendungsprogramm und dem ODBMS bildet (siehe Abbildung 2). Er nimmt die in einer evtl. deklarativen Anfragesprache formulierte Anfrage entgegen und leitet sie an den Anfragemanager auf dem Server weiter. Dieser ist dann für die Auswertung der Anfrage auf den Objekten zuständig, die vom Objektmanager verwaltet werden. Der **Objektmanager** ist die zentrale Komponente eines ODBMS und befindet sich bei der Anfrageserver-Architektur auf dem Server. Er ist dafür zuständig, Objekte auf Speicherseiten abzubilden und sie analog wieder aus diesen zu rekonstruieren. Hierzu bestimmt er anhand der OID die entsprechende Seite, fordert diese vom Seitenmanager an und konstruiert anhand der Informationen auf der Speicherseite das Objekt. Des weiteren ist er auch für die Ausführung von Methoden und die Transaktionsverarbeitung[21] auf Objektebene zuständig. Die Verwaltung der Speicherseiten wird von dem **Seitenmanager** übernommen. Seine Aufgabe besteht darin, dem Objektmanager die gewünschten Seiten zur Verfügung zu stellen und geänderte oder neue Seiten im Sekundärspeicher zu sichern. Falls der Objektmanager keine Transaktionsverarbeitung auf Objektebene vornimmt, kann dies vom Seitenmanager auf der Ebene der Speicherseiten wahrgenommen werden. Aus Gründen der Effizienz besitzen sowohl Objekt- als auch Seitenmanager jeweils einen Pufferspeicher zur temporären Ablage häufig benötigter Objekte oder Seiten.

2.2. Objektserver

Auf den ersten Blick ist der Objektserver (siehe Abbildung 3) ähnlich umfangreich aufgebaut wie der soeben beschriebene Anfrageserver. Er ist jedoch im Vergleich zu diesem um den Anfragemanager reduziert worden, so daß Client und Server direkt über die jeweils vorhandenen Objektmanager kommunizieren. Dem Server obliegt damit primär die Aufgabe die Objekte auf Speicherseiten der Daten-

[21] Im weiteren Verlauf dieses Abschnitts wird von der Synchronisation konkurrierender Zugriffe durch Sperren ausgegangen (pessimistisches Concurrency Control). Nähere Ausführungen über die Transaktionsverarbeitung in ODBMS folgen in Abschnitt 5.

bank abzubilden und umgekehrt aus den Informationen der Speicherseiten Objekte zu extrahieren, um sie den Clients zur Verfügung zu stellen.

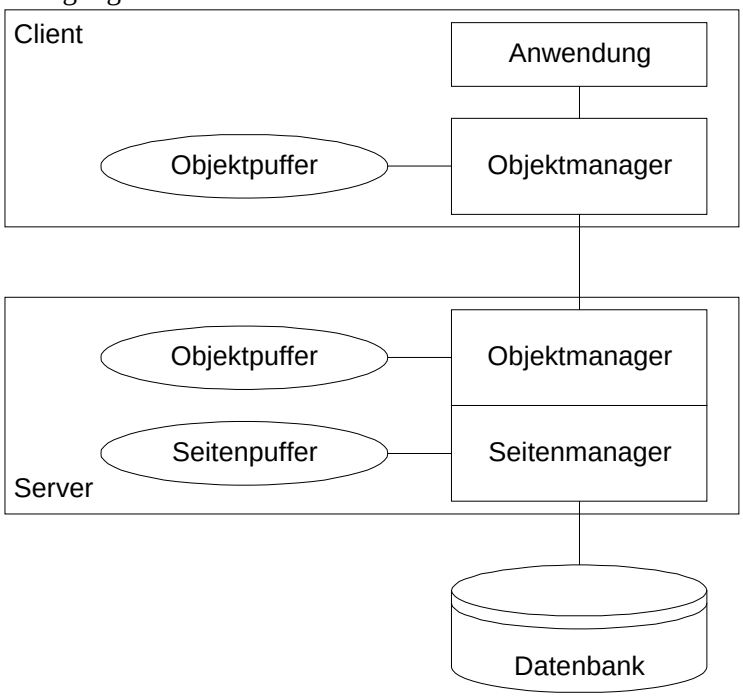

Abbildung 3: Architektur eines Objektservers (angelehnt an [Heu97], S. 418)

Vorteilhaft beim Objektserver-Ansatz ist (im Gegensatz zum Seitenserver; siehe nächster Abschnitt), daß nur die benötigten Objekte zwischen den Rechnern ausgetauscht werden, was eine geringere Belastung für die Kommunikationsinfrastruktur bedeutet. Auch die Transaktionsverarbeitung kann bei dieser Architektur auf Objektebene vorgenommen werden. Der Server ist dafür verantwortlich, die von den Clients angeforderten Sperren auf Objekten zu verwalten. Der Vorteil ist dabei, daß bei der Vergabe von Sperren auf Objektebene keine „Phantomsperren" (siehe nächster Abschnitt) auftreten. Andererseits entsteht hierbei aber ein nicht zu vernachlässigender Verwaltungsaufwand, da ein Client für jedes einzelne Objekt eine Sperre anfordern muß.
Der entscheidende Vorteil der Objektserver-Architektur ist die Möglichkeit der Ausführung von Methoden der Objekte auf beiden Rechnern. Die Objektinformationen stehen sowohl auf dem Client als auch auf dem Server zur Verfügung. Der Entwickler kann hierdurch die Methodenausführung zwischen Client und Server verteilen. So können Methoden, die nur ein Objekt und dessen Attribute betreffen, auf dem Client ausgeführt werden, wohingegen Methoden, die eine größere Menge von Objekten umfassen auf dem Server ablaufen. Ein ähnlicher Aspekt betrifft die Lokalität der Abarbeitung von Anfragen an die Datenbank. Loomis vertritt dabei den Standpunkt, daß Anfragen an die Datenbank auf dem Server abgearbeitet werden sollen (vgl. [Loo92], S. 42 f.). Sie betrachtet Anfragen dabei im Kontext einer deklarativen Anfragesprache wie SQL bei RDBMS. Heuer argumentiert hingegen:

> *„Komplexe, mengenorientierte Anfragen gehören dagegen in die Anwendungslogik und werden oft auf dem Client ausgewertet." [Heu97], S. 417*

Er läßt zwar zu, daß zur Unterstützung einer Anfragesprache zusätzlich ein Anfragemanager auf dem Server vorhanden sein kann, sieht dabei aber eine Überschreitung der Grenze zur Architektur des Anfrageserver. Es ist allerdings wenig einsichtig, wieso ein ODBMS nicht beide Architekturen unterstützen sollte. Heuer beachtet zudem nicht mögliche Konsequenzen aus der Code-Redundanz, die aus den dezentralen Client-Anwendungen resultiert. Geht man von mehreren hundert Clients aus, wird die Wartung der Software auf den Rechnern schnell sehr Umfangreich, so daß Inkonsistenzen in den einzelnen Implementierungen entstehen können.
Aber nicht nur aus diesem Grund wäre es sinnvoll, komplexe Anfragen auf dem Datenbankserver auszuwerten. Auch unter dem Gesichtspunkt, die Kommunikation zwischen den Rechnern zu mini-

mieren und somit die Belastung des Rechnernetzes möglichst gering zu halten, sollte man umfassende Anfragen auf dem Server verarbeiten. Dabei ist zu berücksichtigen, daß ansonsten alle Objekte, die von der Anfrage betroffen wären auf den Client transferiert werden müßten, dieser die Auswertung vornimmt und ein Resultat ermittelt. Das Ergebnis ist im Normalfall wesentlich kleiner als die Objektmenge, über die angefragt wurde. Ähnlich sieht es auch Loomis:

> *„Moving messages and results rather than operand objects can make enormous difference in network traffic with queries." [Loo92], S. 42*

2.3. Seitenserver

Bei der Architektur eines Seitenservers beschränkt sich die Funktionalität des Servers auf das Verwalten von Speicherseiten, die er den einzelnen Clients zur Verfügung stellt (siehe Abbildung 4). Die Anforderung der Speicherseiten obliegt dem Objektmanager auf dem Client, der anhand der Seiteninformationen die Objekte rekonstruiert. Wird dabei nur ein Objekt von dieser Seite benötigt, muß trotzdem die komplette Speicherseite übertragen werden. Andererseits kann dies aber auch ein Vorteil sein, wenn sich mehrere benötigte Objekte auf einer Seite befinden. Es muß nur für das erste Objekt die entsprechende Seite angefordert und übertragen werden. Auf alle anderen Objekte kann dann mit Hauptspeichergeschwindigkeit zugegriffen werden.

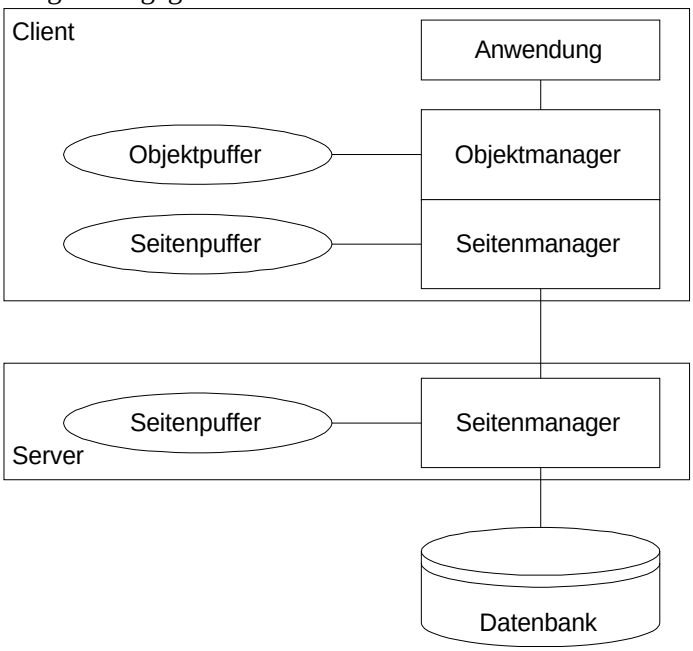

Abbildung 4: Architektur eines Seitenservers (angelehnt an [Heu97], S. 417)

Für die Transaktionsverarbeitung ergibt sich daraus, daß Sperren nicht mehr für einzelne Objekte angefordert werden können. Es müssen immer ganze Speicherseiten gesperrt werden. Dies bedeutet weniger Overhead, wenn mehrere Objekte einer Seite von der gleichen Transaktion gesperrt werden. Andererseits resultiert aus dem Sperren eines Objektes auf einer Seite automatisch das Sperren aller anderen Objekte, die sich auf dieser Seite befinden. Solche Objekte sind dann für andere Transaktionen nicht mehr zugänglich bis die Transaktion, die die Sperre hält, terminiert. Hohenstein et al. sprechen hierbei von sog. „Phantomsperren". (vgl. [HSW97], S. 7) Dieses Verhalten kann die Transaktionsverarbeitung stark einschränken, da mehr Zugriffskonflikte auftreten können als bei der Objektserver-Architektur.

Da der Server bei dieser Architektur keinerlei Kenntnis über Objekte hat, ist der Client für das Ausführen von Methoden und die Auswertung von Anfragen zuständig. Daraus folgt, daß nicht mehr der Server einen potentiellen Flaschenhals darstellt, sondern daß vielmehr leistungsfähige Arbeitsstationen für die Clients und Rechnernetze mit hohen Bandbreiten gefordert sind. Es müssen zuerst die Speicherseiten aller von der Anfrage betroffenen Objekte vom Server an den Client übermittelt werden, damit dieser die Anfrage ausführt und das Resultat ermittelt.

2.4. Abschließende Zusammenfassung

Die drei soeben vorgestellten Architekturen unterscheiden sich prinzipiell in der Aufteilung der mit der Verwaltung persistenter Objekte verbundenen Aufgaben auf Client und Server. Hierdurch wird auch festgelegt, ob zwischen den Rechnern Objekte oder Speicherseiten ausgetauscht werden. Dies hat direkten Einfluß auf die Belastung des Kommunikationsnetzes. Härder et al. betonen hier die Wichtigkeit der Entlastung des Kommunikationsnetzes zwischen Client und Server (vgl. [Här95], S. 59). Diese Forderung erfüllt der Objektserver insofern, da nur relevante Objekte zwischen den Rechnern ausgetauscht werden und nicht zusätzlich noch andere Objekte, die sich auf der gleichen Speicherseite befinden. Auch die Synchronisation des Mehrbenutzerzugriffs kann hier auf Objektebene erfolgen, wobei der Client für die Anforderung von Sperren zuständig ist, diese aber vom Server verwaltet werden. Demgegenüber steht der Nachteil des erhöhten Verwaltungsaufwandes, da jedes Objekt einzeln übertragen und jede Sperre separat angefordert werden muß. Dies ist beim Seitenserver nur eingeschränkt der Fall, da hier durch das gemeinsame Abspeichern zusammen benötigter Objekte auf einer Speicherseite diese auch in einem Durchgang übertragen und gesperrt werden können. Beim Objektserver können aber auf dem Server Methoden der Objekte ausgeführt und Anfragen ausgewertet werden. Hieraus resultiert eine erhebliche Entlastung des Kommunikationsnetzes, da nur die Anfrage und das Resultat, nicht aber die Objektmenge, auf der die Anfrage ausgewertet werden soll, übertragen werden müssen. Einen zusammenfassenden Überblick über die drei vorgestellten Architekturen liefert folgende Tabelle, in der für jede Architektur die einzelnen Kernaspekte dargestellt sind. In der ersten Zeile ist die Granularität der ausgetauschten Informationseinheiten zu sehen, welche auch Granulat der Transaktionsverarbeitung darstellt. Die beiden folgenden Zeilen führen auf, wo Methoden der Objekte resp. Anfragen ausgewertet werden. In der letzten Zeile ist dargestellt, auf welchem Rechner die Hauptlast der Transaktionsverarbeitung angesiedelt ist.

	Anfrageserver	Objektserver	Seitenserver
Granulat	Daten/Objekte	Objekte	Speicherseiten
Ausführen von Methoden	Server	Client/Server	Client
Auswerten von Anfragen	Server	Server	Client
Transaktionsverarbeitung	Server	Client	Client

Tabelle 1: Architekturen im Vergleich

2.5. Architektur von GemStone

In Abschnitt 2.5.1 werden zunächst die wichtigsten Komponenten eines GemStone-Servers einführend beschrieben. Anschließend werden in Abschnitt 2.5.2 die Zusammenhänge zwischen den Komponenten anhand zweier möglicher Konfigurationen eines GemStone-Servers erläutert.

2.5.1. Komponenten von GemStone

Ein GemStone-Server[22] besteht aus einem gemeinsamen persistenten Objektspeicher, dem sog. Object Repository, und mehreren kooperierenden Prozessen, von denen jeder für einen Teil der Funktionalität des ODBMS zuständig ist (eine vereinfachte Darstellung findet sich in Abbildung 5 auf Seite 20). Die beiden wichtigsten Prozesse bei der Verwaltung persistenter Objekte sind der Stone und der Gem, aus denen sich der Name des ODBMS zusammensetzt.

2.5.1.1. Object Repository

Das **Object Repository** (oder kurz: **Repository**) ist die logische Speichereinheit auf dem nicht flüchtigen sekundären Speichermedium, in dem GemStone persistente Objekte verwaltet (vgl. [Gem96b], S. 2-12). Die atomare Einheit beim Zugriff ist eine physikalische Speicherseite. Auf den physikalischen Aufbau eines Repository und die Möglichkeiten, es über mehrere Speichermedien zu verteilen

[22] In diesem Zusammenhang bezeichnet der Begriff „Server“ nicht den Server-Rechner, auf dem GemStone läuft, sondern das gesamte ODBMS als logische Einheit.

oder zu replizieren[23], wird innerhalb dieses Berichtes nicht weiter eingegangen. Für weiterführende Informationen sei an dieser Stelle auf [Gem96a] und [Gem96b] verwiesen. Für den Entwickler stellt das Repository den zentralen Objektspeicher dar, in dem alle persistenten Objekte abgelegt werden. Gemäß der konsequent objektorientierten Konzeption der Sprache Smalltalk werden auch die Metainformationen einer GemStone-Datenbank durch Objekte repräsentiert und somit auch im Repository verwaltet. Dies umfaßt nicht nur die Klassendefinitionen der Objekte inklusive der zugehörigen Methoden sondern auch datenbankspezifischen Konzepte und Informationen, wie z.B. Mechanismen zur Transaktionsverarbeitung, das Autorisierungsmodell und auch die Verwaltung der Benutzer der Datenbank.

2.5.1.2. Stone

Für jedes Repository existiert genau ein Stone-Prozeß (oder kurz: **Stone**), der als zentraler Koordinator für die Ressourcen der ODB fungiert (vgl. [Gem96b], S. 2-11). In der Rolle als Verwalter eines verteilten objektorientierten Systems zeichnet er sich für die Erzeugung und Verwaltung systemweit eindeutiger OID verantwortlich. Als Manager eines DBS übernimmt er zusätzlich die Verwaltung von Informationen zur Synchronisation des Mehrbenutzerzugriffs auf das Repository. Zu diesen Aufgaben gehören neben dem Führen von Logbüchern auch die Verwaltung der von den einzelnen Transaktionen gelesenen, geschriebenen und evtl. auch gesperrten Objekten, um die Transaktionen der Anwender zu synchronisieren. Da der Zugriff auf die ODB in Form von Lesen und Schreiben von Speicherseiten erfolgt, werden von ihm diese Speicherseiten verwaltet.

2.5.1.3. Gem

Für jede Applikation, die auf das Repository zugreift, erzeugt GemStone einen Gem-Prozeß (oder kurz: **Gem**), der für Client-Anwendung die Schnittstelle zu den persistenten Objekten in der Datenbank ist. Normalerweise benötigt man pro Anwendung nur eine Sitzung für den Zugriff auf eine DB, so daß genau ein Gem, der für die Koordinierung einer einzigen Sitzung zuständig ist, für diese Anwendung ausreicht. Es ist aber durchaus denkbar, daß eine Anwendung mehrere DB benutzt oder eine Anwendung über verschiedene Sitzungen auf eine DB zugreifen muß, so daß in GemStone auch die Möglichkeit der Erzeugung mehrerer Gems für eine Anwendung zugelassen ist.
Der Gem gewährleistet dem Benutzer eine konsistente Sicht auf das Object Repository und verwaltet innerhalb einer Sitzung sämtliche Zugriffe auf die Objekte der GemStone-DB (vgl. [Gem96b], S. 2-10). Beim lesenden Zugriff auf persistente Objekte transferiert er Speicherseiten in seinen lokalen Arbeitsspeicher und rekonstruiert daraus die Objekte. Methoden der Objekte können innerhalb des Gem ausgeführt werden. Der Gem unterstützt darüber hinaus die Transaktionsverarbeitung, indem er die von ihm gelesenen und/oder modifizierten Objekte verwaltet und bei einem commit mögliche Konflikte mit von anderen Gems gelesenen oder geschriebenen Objekten registriert. Dies geschieht in Kooperation mit dem Stone.

2.5.1.4. Page Server Process

Neben dem zentralen Stone für ein Repository und den Gems für jeweils eine Sitzung existieren in GemStone noch weitere Prozesse, die den Mehrbenutzerzugriff auf das Repository unterstützen. Wie bereits erwähnt ist die elementare Einheit, in der auf das Object Repository zugegriffen wird, eine Speicherseite. Analog zu den bisher vorgestellten Architekturen existiert ein Seitenmanager (in der Terminologie von GemStone: **page server process**), der für das Lesen und Schreiben der Speicherseiten zuständig ist.

[23] Mit „replizieren“ ist hier die Replikation im Sinne der Verteilung von Datenbanken gemeint. Sie bedeutet, daß eine Datenbank redundant auf mehrere Server oder Speichermedien verteilt werden kann, wodurch die Risiken im Fehlerfall oder beim Ausfall einer Einheit reduziert werden.

2.5.2. Konfiguration des GemStone-Servers

Das Anwendungspotential von GemStone wird durch die Verteilung der Prozesse auf Client- und Server-Rechner bestimmt. Zwei idealtypische Konfigurationen sind in Abbildung 5 dargestellt und werden im folgenden diskutiert.

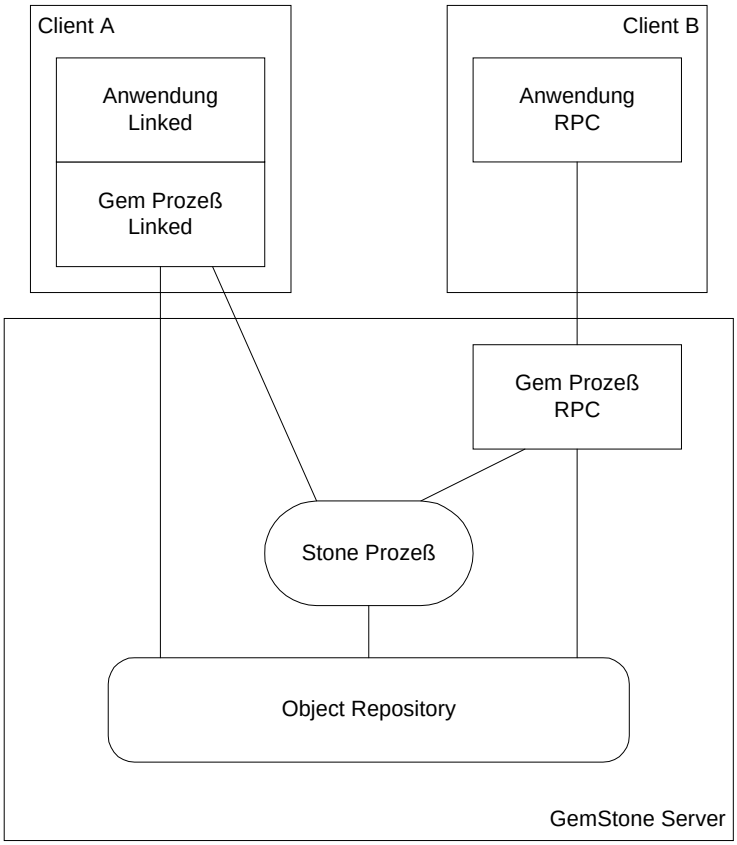

Abbildung 5: Vereinfachte Darstellung der Architektur von GemStone

2.5.2.1. Linked Session

Bei der Konfiguration, in der die Kommunikation zwischen Client-Anwendung und GemStone-Server über sog. **Linked Sessions**[24] stattfindet, ist der Gem direkt in den Prozeß der Anwendung auf dem Client-Rechner integriert. Es existiert im Gegensatz zur nachfolgend beschriebenen Konfiguration eine Einschränkung auf maximal einen Gem pro Smalltalk-System. Das Object Repository und der Stone befinden sich zusammen auf einem einzigen GemStone-Server. Diese Konfiguration ist in Abbildung 5 auf der linken Seite als Client A dargestellt, wobei aus Gründen der Übersichtlichkeit in dieser Abbildung nur die zentralen Prozesse von GemStone dargestellt sind. Die Daten werden zwischen den beiden Rechnern in Form von Speicherseiten übertragen, wofür auf beiden jeweils ein page server process (in der Abbildung nicht dargestellt) zuständig ist. Diese Konfiguration entspricht in wesentlichen Teilen der in Abschnitt 2.3 beschriebenen Architektur eines Seitenservers. Demgemäß wird ein großer Teil der Arbeit von den Gems übernommen, die die Rolle der Objektmanager einnehmen. Eine Ausnahme zu der allgemeinen Architektur des Seitenservers stellt der Stone auf dem Server dar. Er übernimmt hier die zentrale Transaktionsverarbeitung auf Objektebene und nicht auf Seitenebene.

Aus dieser Konfiguration ergeben sich somit ähnliche Vor- und Nachteile wie bei der allgemeinen Seitenserver-Architektur. Anstatt einzelner Objekte werden ganze Speicherseiten übertragen. Dies kann eine erhöhte Belastung des Kommunikationsnetzes bedingen, da mit den Speicherseiten u.U.

[24] Da eine Übersetzung dieses Begriffs künstlich erscheinen würde und ihn nur schwer ersetzen kann, wird der englische Begriff aus der GemStone-Dokumentation übernommen.

mehr Objekte übertragen als auf dem Client benötigt werden. Andererseits können sich aber auch Vorteile ergeben, wenn z.B. durch Ausnutzung von Cluster-Bildung Objekte, die mit hoher Wahrscheinlichkeit gemeinsam benötigt werden, auf einer einzigen Seite abgelegt sind. Die Extraktion der Objekte aus den Speicherseiten wird auf Client-Seite durch den Gem durchgeführt, der bei Bedarf auch Methoden der GemStone-Objekte ausführen kann. Darüber hinaus fordert er Sperren auf Objekten bei dem Stone an und führt mit ihm die Synchronisation der Transaktionen durch. Da der größte Teil der hier aufgeführten rechenintensiven Aufgaben vom Gem auf dem Client ausgeführt werden, wird der Server-Rechner durch eine Sitzung wenig belastet. Diese Konfiguration eignet sich also vor allem unter solchen Voraussetzungen, wenn möglichst viele Clients vom Server bedient werden sollen und auf einzelnen Speicherseiten möglichst viele benötigte Objekte übertragen werden (vgl. [STS97], S. 486).
An dieser Stelle sei aber nochmals hervorgehoben, daß nur die Ausführung der Methoden auf dem Client stattfindet, da die ausführende Instanz der Gem-Prozeß auf dem Client ist. Die Speicherung der Methoden hingegen erfolgt zusammen mit den Klassendefinitionen der GemStone-Objekte in dem Object Repository auf dem Server. Obwohl bei Nutzung einer Linked Session die zentrale Ausführung der Methoden auf dem Server nicht zum Tragen kommt, unterstützt dieses zentralisierte Vorhalten der Methoden jedoch eine einfachere Wartung[25] des Softwaresystems. Änderungen der Klassen und Methoden können an einer zentralen Stelle durchgeführt werden.

2.5.2.2. RPC-Session

Bei der zweiten Variante der Konfiguration eines GemStone-Servers läuft der Gem zusammen mit dem Stone auf dem Server (siehe Abbildung 5 rechts). Der Begriff **RPC-Session** steht hierbei für die Form der Kommunikation zwischen dem lokalen Anwendungsprogramm und dem Gem. Im Gegensatz zu einer Linked Session, bei der der Gem direkt in den Prozeß der Anwendung integriert ist, sind die beiden hier auf verschiedene Rechner verteilt. Zur Kommunikation über Rechnergrenzen hinaus werden sog. „remote procedure calls“ (kurz: RPC[26]) genutzt. In dieser Konfiguration sind alle Komponenten auf dem Server-Rechner angesiedelt. Dies sind das Object Repository, der Stone als zentraler Koordinator des Zugriffs auf das Repository und für jede RPC-Session ein Gem, der auf dem Server die Aufgabe des Objektmanagers einnimmt. Auf dem Client läuft nur die lokale Smalltalk-Anwendung, in die auch wieder ein Objektmanager[27] integriert ist. Diese Konfiguration spiegelt die in Abschnitt 2.2 beschriebene Architektur eines Objektservers wider.
Hierdurch gelten für die RPC-Session die gleichen Rahmenbedingungen und Einsatzmöglichkeiten, wie für die allgemeine Objektserver-Architektur. Es werden nur die benötigten Objekte zwischen GemStone-Server und Anwendungs-Client übertragen. Die Problematik von Phantomobjekten, die zusammen mit angeforderten Objekten auf den gleichen Speicherseiten liegen (vgl. [HSW97], S. 7), ist hier nicht gegeben. Demgemäß fallen Geschwindigkeitsvorteile durch Cluster-Bildung eher gering aus, da nur der Gem mit Hauptspeichergeschwindigkeit auf die bereits vom Sekundärspeicher gelesenen Seiten zugreifen kann. In diesem Fall stellt das Kommunikationsnetz zwischen Client und Server den Flaschenhals dar. Die Transaktionsverarbeitung wird vollständig von den einzelnen Gem-Prozessen und dem zentralen Stone auf dem Server durchgeführt, so daß die Client-Rechner von dieser Aufgabe und dem damit verbundenen Kommunikationsaufwand entlastet werden. Darüber hinaus

---

[25] Unter dem Begriff Wartung werden innerhalb dieser Arbeit alle Tätigkeiten verstanden, die den Betrieb und Erhalt eines eingesetzten Softwaresystems unterstützen sollen und nicht der reinen Erstellung des Systems zugeordnet werden können. Hierbei lassen sich verschiedene Aspekte der Wartung unterscheiden: korrektive Wartung (Modifikationen zur Fehlerbeseitigung), adaptive Wartung (Modifikationen aufgrund von Änderungen der Systemumgebung) und perfektive Wartung (Modifikationen zur Verbesserung der Qualität) (vgl. [GJM91], S.25 f.).

[26] RPC ist ein standardisierter Mechanismus zum Aufruf von Prozeduren oder Funktionen auf einem über ein Kommunikationsnetz erreichbaren Rechner.

[27] Obwohl bisher begrifflich nicht zwischen dem Objektmanager auf dem Server und dem auf dem Client differenziert wurde, handelt es sich in diesem Fall durchaus um unterschiedliche Komponenten. Der in die Client-Anwendung integrierte Objektmanager ist kein Gem, sondern ein eigener Objektmanager in Form des sog. GemBuilder für GemStone. Dieser wird in der Beschreibung des Zugriffs auf persistente Objekte in GemStone in Abschnitt 4.4.1 vorgestellt.

können durch das Vorhandensein des Gem als Objektmanager auf dem Server auch die Methoden auf diesem ausgeführt werden.

3. Persistenzmodelle objektorientierter Datenbanken

Nach Atkinson und Morrison ist **Persistenz** die Fähigkeit von Daten (bzw. Objekten) eine beliebig lange Lebensdauer zu haben (vgl. [AtMo95], S. 323). Saake et al. führen hierbei die Unterscheidung zwischen **persistenten** und **transienten** Objekten ein (vgl. [STS97], S. 161)[28]. Unter persistenten Objekten werden solche Objekte verstanden, die die Dauer der Ausführung eines Anwendungsprogramms überleben. Dazu werden sie von einem ODBMS verwaltet, das sie auf einem Sekundärspeicher ablegt und darüber hinaus anderen Benutzern zur Verfügung stellt. Demgegenüber werden transiente Objekte nicht vom ODBMS verwaltet, da sie nur während der Laufzeit einer Anwendung existieren. Hierzu zählen bspw. temporäre Objekte innerhalb von Methoden oder globale Objekte des Anwendungsprogramms. Die Erzeugung und das Löschen solcher transienten Objekte wird durch entsprechende Mechanismen der OPL (genauer: ihrer Laufzeitumgebung) der Anwendung gesteuert, wohingegen die Verwaltung persistenter Objekte dem ODBMS obliegt. Hierbei muß das ODBMS Konzepte und Mechanismen bereitstellen, die das persistente Ablegen von Objekten aus der Anwendung erlauben. Diese Konzepte fassen sowohl Heuer als auch Saake unter dem Begriff **Persistenzmodell** zusammen (vgl. [Heu97], S. 524 und [STS97], S. 161). In konventionellen DBMS, wie bspw. relationalen Systemen, werden Daten durch spezielle Schreiboperationen in der Datenbank persistent gemacht. Ein an diesen Ansatz angelehntes Persistenzmodell führt Heuer beim Einsatz ODBMS auf, bei dem durch Hinzufügen neuer Datenstrukturen und Bibliotheksroutinen zu einer Anwendung Persistenz von Objekten realisiert werden kann (vgl. [Heu97], S. 523). Neben diesem einfachen Persistenzmodell existieren im Bereich von ODB noch weitere, die der Komplexität des OM Rechnung tragen (vgl. [STS97], S. 161). Diese sollen im weiteren Verlauf dieses Abschnitts in Anlehnung an [Heu97], S. 521 ff. vorgestellt werden.

3.1. Persistente und persistenzfähige Klassen

Eine erste Konvergenz von Objektmodell und Datenbank ist das Konzept der **klassenabhängigen Persistenz**. Hierbei erhält der Entwickler die Möglichkeit, Klassen zu definieren, deren Extension persistent sein soll[29]. Dies bedeutet, daß automatisch alle Instanzen einer solchen Klasse in der Datenbank abgelegt werden (Heuer nennt sie in [Heu97] auf S. 523 **persistente Klassen**). Es wird somit vorausgesetzt, daß eine Klasse entweder nur persistente oder nur transiente Objekte haben kann, was aber nicht immer gegeben sein muß[30]. Demgegenüber wäre es sinnvoller, Persistenz als **objektabhängige** Eigenschaft zu betrachten. Heuer spricht in diesem Zusammenhang von **persistenzfähigen Klassen**, d.h. Klassen, die sowohl persistente als auch transiente Instanzen besitzen können (vgl. [Heu97], S. 524). Im folgenden sollen deswegen einige Techniken zur Definition persistenzfähiger Klassen vorgestellt werden[31].

Eine Form der Definition persistenzfähiger Klassen ist **Persistenzfähigkeit durch Vererbung**. Hierbei müssen alle persistenzfähigen Klassen Unterklassen einer speziellen, vom System vorgegebenen persistenzfähigen Klasse sein (vgl. [Heu97], S. 524). Dieser Ansatz kann aber in Programmiersprachen, die keine Mehrfachvererbung unterstützen, schnell zu Problemen führen, wenn zusätzlich auch noch eine andere (Bibliotheks-)Klasse durch Spezialisierung wiederverwendet werden soll. Des weiteren schränken aber auch Konflikte, die bei Mehrfachvererbung auftreten können, die Verwendung dieses Persistenzmodells ein (vgl. [Mey97], S. 535 ff.).

---

[28] Diese Unterscheidung findet man auch in [Pra99] auf S. 59, [MeWü97] auf S. 36 oder in [Heu97], S. 523.

[29] Aus diesem Grund sprechen Saake et al. hier auch von persistenten Klassenextensionen statt persistenter Klassen (vgl. [STS97] S. 165).

[30] Man denke hier z.B. an Texte einer Dokumentenverwaltung, die sowohl dauerhaft gespeichert werden sollen, aber andererseits als temporäre Textausschnitte während der Bearbeitung vorliegen können.

[31] Damit soll nicht ausgeschlossen werden, daß solche oder ähnliche Realisierungsformen auch bei persistenten Klassen möglich wären. Da objektabhängige Persistenz aber flexibler ist als klassenabhängige, wollen wir uns hier auch auf diese konzentrieren.

Diese Probleme treten bei **Persistenzfähigkeit durch explizite Kennzeichnung** nicht auf. Hierbei werden persistenzfähige Klassen bei ihrer Definition durch einen speziellen Bezeichner gekennzeichnet. Die Übersetzung in ein lauffähiges Programm kann deswegen nicht mehr nur durch den vorhandenen Compiler vorgenommen werden. Dieser muß vielmehr so modifiziert werden, daß er die entsprechenden Bezeichner interpretieren und Datenbankcode erzeugen kann (vgl. [Heu97], S. 524). Diese Lösung teilt mit den vorangegangenen aber einen großen Nachteil: Instanzen solcher Klassen, die nicht als persistenzfähig definiert wurden, können nicht persistent gemacht werden. Es stellt sich nun die Frage, wie man reagiert, wenn ein persistentes Objekt auf Instanzen einer nicht persistenzfähigen Klasse verweist. Um die Informationen nicht zu verlieren, müssen diese Objekte erst in solche einer persistenzfähigen Klasse umgewandelt werden (vgl. [AtMo95], S. 12).
Bei der **typunabhängigen Persistenzfähigkeit**[32] ist Persistenz eine Eigenschaft, die orthogonal zur Klassenzugehörigkeit eines Objektes steht. Alle Klassen können sowohl transiente als auch persistente Instanzen besitzen (vgl. [STS97], S.163 ff.). Die bisher genannten Einschränkungen existieren bei dieser Form der Persistenzfähigkeit nicht. Trotz des Overheads, den man hiermit bei Klassen hat, die nicht persistenzfähig sein müssen, stellt sie doch die universellste Art der Persistenzfähigkeit dar. Die Problematik der Typumwandlung von nicht persistenzfähigen in persistenzfähige Objekte ist nicht gegeben. Auch während der Entwicklung entfällt der Aufwand, persistente Klassen oder Objekte zu identifizieren, sofern die Klassen nicht zusätzlich auch in der ODB definiert werden müssen.

3.2. Persistentmachung von Objekten

Beim Persistenzmodell mit persistenten Klassen sind per Definition alle Instanzen dieser Klassen persistent. Die Entscheidung, ob ein Objekt persistent sein soll, kann ausschließlich anhand der Klassenzugehörigkeit getroffen werden. Bei persistenzfähigen Klassen gilt dies aber nicht mehr. Deswegen müssen bei diesem Persistenzmodell zusätzlich noch Mechanismen bereitgestellt werden, die dazu dienen, das ODBMS über im persistenten Speicher zu verwaltende Objekte zu informieren. In diesem Zusammenhang stellt Heuer mehrere Möglichkeiten vor, auf die hier jeweils kurz eingegangen wird (siehe auch [Heu97], S.525 f.).
Steht schon bei der Erzeugung eines Objektes die Persistenz fest, kann es **persistent instanziert** werden. Für diese Möglichkeit bieten einige ODBMS einen zusätzlichen oder einen überladenen Instanzierungsoperator, der ein neues Objekt zum Zeitpunkt der Erzeugung in der Datenbank anlegt. In einigen verfügbaren Systemen kann ein transientes Objekt auch nach seiner Erzeugung persistent gemacht werden. Realisiert wird dies durch **Senden einer Nachricht** oder durch **Zuweisung eines Bezeichners** in der Datenbank.
Den bisher genannten Mechanismen ist gemeinsam, daß man mit ihnen Objekte explizit als persistent identifizieren und kennzeichnen muß. Bei dem Konzept der **Persistenz durch Erreichbarkeit** werden persistente Objekte automatisch durch das ODBMS ermittelt. Der Grundgedanke ist, daß jedes Objekt, welches von einem persistenten Objekt referenziert wird, auch wiederum persistent sein muß. Aufgrund dieser Regel kann nun ein ODBMS automatisch identifizieren, ob ein Objekt im persistenten Speicher abgelegt werden muß. Explizite Schreiboperationen in die DB entfallen. Da dieser Mechanismus aber nicht für sich alleine stehen kann, wird bei Persistenz durch Erreichbarkeit mindestens ein Objekt durch Zuweisung eines Bezeichners in der Datenbank persistent gemacht. Diese Objekte dienen der Definition von Einstiegspunkten in die Datenbank (engl. database root). Die Persistenz dieser Objekte überträgt sich transitiv auf alle Objekte, die von ihnen aus im Objektgraphen erreicht werden können (vgl. auch [MeWü97], S. 61 f. und [STS97], S. 170), d.h. das ODBMS muß die transitive Hülle aller von einem Wurzelobjekt über Referenzen erreichbare Objekte berechnen. Analog dazu ist jedes Objekt (außer den Wurzelobjekten), welches nicht von einem persistenten Objekt erreicht werden kann, nicht persistent. Objekte, die nach diesen Regeln einmal persistent waren, aber mittlerweile nicht mehr von persistenten Objekten referenziert werden, müssen aus der ODB entfernt werden. Dieses Entfernen sollte ähnlich wie das Hinzufügen zur Datenbank automatisch er-

[32] Demgegenüber sprechen Saake et al. von typabhängiger Persistenz, wenn nur Instanzen einer bestimmten Klasse persistent werden können, die durch Vererbung oder explizite Kennzeichnung persistenzfähig gemacht wurde.

folgen, wofür sich eine automatische Speicherbereinigung (engl. garbage collection) eignet (vgl. [MeWü97], S. 63).

3.3. Orthogonale Persistenz

Die bisher beschriebenen Mechanismen zur Realisierung von Persistenz unterscheiden sich u.a. darin, inwieweit sie sich transparent in die Softwareentwicklung integrieren lassen. Je höher der Grad an Transparenz, desto mehr können die Entwickler von Aspekten der Persistenz abstrahieren. Atkinson und Morrison haben sich in [AtMo95] mit der Fragestellung beschäftigt, wie Persistenz unter diesen Gesichtspunkten realisiert werden sollte. Innerhalb ihres Beitrags formulieren sie drei Prinzipien, die den Persistenzmechanismen zugrunde liegen sollten. Werden alle drei Prinzipien eingehalten, sprechen die Verfasser von **orthogonaler Persistenz**.

Durch das Prinzip der Typorthogonalität (**„The Principle of Data Type Orthogonality“**) soll gewährleistet werden, daß jedes Objekt unabhängig von seinem Typ persistent werden kann. Es existiert also keine Unterscheidung zwischen Datenbanktypen und Nicht-Datenbanktypen. Dieses Prinzip wird nur durch die automatische Persistenzfähigkeit aus Abschnitt 3.1 erfüllt.

Das Prinzip der Persistenz-Identifikation (**„The Principle of Persistence Identification“**) fordert, daß die Bestimmung, ob ein Objekt persistent ist oder nicht, automatisch zu erfolgen hat. Somit scheiden die Mechanismen zur expliziten Persistentmachung aus Kapitel 3.2 aus, und es bleibt nur Persistenz durch Erreichbarkeit, die dieses Prinzip erfüllt. Die expliziten Mechanismen werden nur zur Festlegung der Datenbankwurzeln benutzt.

Das Prinzip der Persistenz-Unabhängigkeit (**„The Principle of Persistence Independence“**) besagt, daß der Programmcode unabhängig davon sein sollte, ob transiente oder persistente Objekte bearbeitet werden. Dies bedeutet, daß transiente Objekte genau so behandelt werden können wie persistente. Der Programmierer sollte von der Aufgabe entlastet werden, Objekte mittels spezieller Lese- und Schreiboperationen aus der Datenbank zu holen und wieder zurückzuschreiben. Dies sollte vom DBMS übernommen werden, welches hierzu Mechanismen zum transparenten Zugriff auf persistente Objekte bereitstellen muß (näheres hierzu in Abschnitt 4).

Atkinson und Morrison argumentieren damit, daß der Entwickler seine „intellektuellen Fähigkeiten“ mehr zur Realisierung des Systems nutzen kann und sich nicht mit Aspekten der Datenbank beschäftigen sollte. Dabei unterstützt ihn die Typorthogonalität insofern, daß Persistenzfähigkeit eine inhärente Eigenschaft aller Klassen ist und keine gesonderten persistenten oder persistenzfähigen Klassen entwickelt werden müssen. Darüber hinaus sorgen Mechanismen zur Realisierung von Persistenz durch Erreichbarkeit dafür, daß persistente Objekte automatisch identifiziert werden. Voraussetzung ist die explizite Persistentmachung mindestens eines Wurzelobjekts. Alle Objekte, die im Objektgraphen von diesem Objekt aus erreicht werden können, sind automatisch persistent. Damit einher geht auch die Forderung nach der Unabhängigkeit des Programmcodes vom zugrunde liegenden Speichermedium. Durch das Fehlen von datenbankspezifischen Anweisungen im Programmcode ist dieser unabhängig von dem benutzten Speichermedium[33] und bleibt kleiner und wartbarer.

Das Verwirklichen von Persistenz nach diesen Prinzipien erleichtert die Arbeit der Entwickler eines Anwendungssystems, indem eine transparente Anbindung der Datenbank an die zu entwickelnde Applikation ermöglicht wird. Demgegenüber steht aber die Fragestellung, inwieweit sich diese Prinzipien in kommerziellen DBS effizient realisieren lassen. Den Aspekten der Realisierung dieser Persistenzprinzipien in GemStone widmet sich der folgende Abschnitt 3.4.

3.4. Persistenzmodell von GemStone

In GemStone ist Persistenz eine Eigenschaft aller Objekte, die unabhängig von ihrer Klassenzugehörigkeit ist. Das bedeutet, daß für alle in GemStone definierten Klassen sowohl persistente als auch transiente Instanzen existieren können. Dies entspricht dem in Abschnitt 3.1 definierten Begriff der Persistenzfähigkeit einer Klasse. Die Betonung liegt aber darauf, daß die entsprechenden Klassen auch in GemStone definiert sind. Klassendefinitionen innerhalb des Anwendungsprogramms auf dem Client sind davon nur bedingt betroffen. Hierzu sind beispielsweise die Klassen zur Realisierung der

[33] Insbesondere bleibt er auch unabhängig von dem ODBMS.

Benutzungsschnittstelle zu zählen, die tendenziell kurzlebig sind und nicht in der Datenbank abgelegt werden.

3.4.1. Umsetzung des Persistenzmodells in GemStone

Aufgrund der objektabhängigen Persistenz kann diese nicht an einzelnen Klassen festgemacht werden, sondern es besteht die zusätzliche Notwendigkeit der Festlegung der letztendlich persistent abgelegten Objekte. GemStone realisiert hierzu das in Abschnitt 3.2 eingeführte Konzept der „Persistenz durch Erreichbarkeit“. Zur Definition der Datenbankwurzel (database root) benutzt GemStone einen an die Vergabe eines Datenbankbezeichners angelehnten Mechanismus. Eine Veranschaulichung dieses Mechanismus auf statischer Ebene findet sich in dem Klassendiagramm in Abbildung 6. Jedem GemStone-Benutzer ist innerhalb der Datenbank über sein Benutzerprofil (in Abbildung 6 durch die Klasse UserProfile symbolisiert) eine sog. Symbolliste zugeordnet. Diese Liste besteht aus mehreren Symbolverzeichnissen (im Sprachgebrauch von GemStone: SymbolDictionary), wobei jedes dieser Symbolverzeichnisse der Verwaltung der dem entsprechenden Benutzer zur Verfügung stehenden Objekte dient. Jedes Symbolverzeichnis ist dabei als Menge von Assoziationen organisiert, die einem innerhalb dieses Verzeichnisses eindeutigen Symbol (key) ein zugehöriges Objekt (value) zuordnen. Der Zugriff auf das Objekt erfolgt über dieses eindeutige Symbol. Über die Symbolverzeichnisse in der Symbolliste eines Benutzers werden alle Objekte festgelegt, auf die er in GemStone zugreifen darf. Analog dazu kann ein Benutzer nicht auf Objekte zugreifen, die nicht über seine Symbolliste erreichbar sind. Hierdurch hat man schon einen einfachen Mechanismus zum Verbergen bestimmter Objekte vor anderen Benutzern. Für jeden neuen Benutzer werden in GemStone automatisch drei Symbolverzeichnisse angelegt. Eines davon dient dem Ablegen von privaten Objekten und ein weiteres der Verwaltung von Objekten, die mit anderen Benutzern geteilt werden. Das dritte Symbolverzeichnis umfaßt alle Klassen[34] der in GemStone vorgegebenen Klassenhierarchie und bildet den Einstiegspunkt für die Instanzierung und Verwendung der angebotenen Basisobjekte. Die Symbolliste kann jederzeit durch weitere Symbolverzeichnisse erweitert werden. In solchen Symbolverzeichnissen können auch die Klassen und Objekte einer neu erstellten Anwendung abgelegt werden.

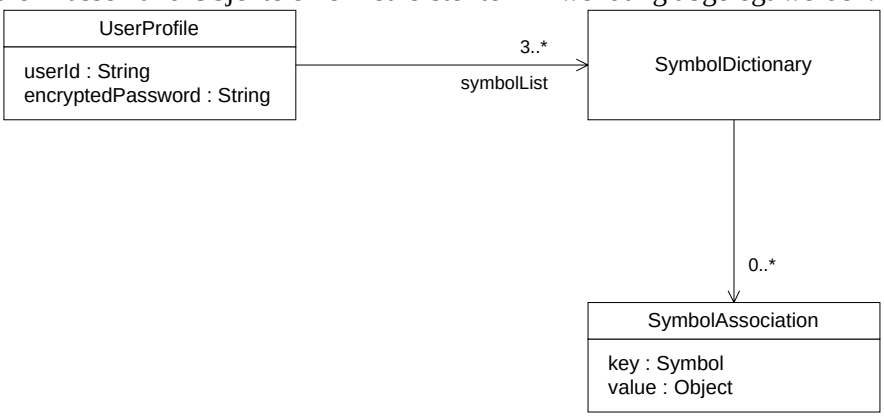

Abbildung 6: Symbolverzeichnisse in GemStone

(UML-Notation)

Wie bereits eingangs ausgeführt, können in GemStone nur solche Objekte persistent werden, deren Klassendefinitionen in GemStone existieren. Aufgrund der Persistenz durch Erreichbarkeit kann es aber vorkommen, daß innerhalb der Anwendung auf dem Client ein Objekt referenziert wird, dessen Klasse nicht in GemStone definiert worden ist. Um in einem solchen Fall dem Verlust von Objekten vorzubeugen und somit die referentielle Integrität des Objektnetzes innerhalb der Datenbank sicherzustellen, bietet GemStone die Möglichkeit, Klassen zur Laufzeit automatisch zu erzeugen. Hierdurch

[34] Klassen sind in GemStone – wie in Smalltalk – auch wieder Objekte, so daß Klassen auch wie solche behandelt werden können. Insbesondere resultiert daraus, daß überall dort, wo ein Objekt eingesetzt werden kann auch die Verwendung einer Klasse gültig ist. Deswegen können in den Symbolverzeichnissen neben Objekten auch Klassendefinitionen verwaltet werden.

werden Änderungen der Smalltalk-Anwendung auf dem Client in das Schema der DB auf dem Server übernommen. Von dieser automatischen Definition sind auch noch nicht definierte Oberklassen betroffen. An dieser Stelle ist aber auf eine wichtige Einschränkung hinzuweisen: GemStone legt in einem solchen Fall nur die Struktur der Objekte in Form der Attributdefinitionen an. Das Verhalten der Objekte wird dabei nicht berücksichtigt. Sie können deswegen innerhalb von GemStone keine Methoden ausführen außer solchen, die in evtl. schon vorhandenen Superklassen definiert sind. Daraus resultiert, daß die Methoden der Objekte zu einem späteren Zeitpunkt manuell erstellt werden müssen. Aufgrund der Erweiterbarkeit der Sprache Smalltalk kann hierfür aber auch eine (halb-) automatische Lösung selbst erstellt werden. Diese muß in den Erzeugungsmechanismus integriert werden und nach dem Anlegen der Klassen zusätzlich die Methoden übersetzen.

3.4.2. Zusammenfassende Beurteilung des Persistenzmodells

Das Persistenzmodell von GemStone verspricht eine transparente Realisierung von Persistenz, so daß der Entwickler in weiten Teilen davon abstrahieren kann. Nach der expliziten Festlegung der persistenten Wurzel erfolgt das Persistentmachen weiterer Objekte implizit über die Erreichbarkeit von diesem Wurzelobjekt. Dadurch kann während der Entwicklung von der Persistenz einzelner Objekte abstrahiert werden, da außer für die persistente Wurzel keine expliziten Speicheroperationen notwendig sind. Das Programm bleibt somit frei von datenbankspezifischen Operationen, wodurch auch die Anforderung der Unabhängigkeit des Programmcodes von dem eingesetzten Persistenzmechanismus umgesetzt werden kann. Des weiteren finden sich in diesem Persistenzmodell die von Atkinson und Morrison in [AtMo95] formulierten Anforderungen an Persistenz[35] fast vollständig realisiert. Dies liegt in der Identifikation persistenter Objekte über die „Persistenz durch Erreichbarkeit“ und der Unabhängigkeit des Programmcodes von der Persistenz begründet. Die Typorthogonalität trifft nur mit Restriktionen zu, da sie nur für solche Objekte gilt, deren Klassen in GemStone definiert sind.
Obwohl dieses Persistenzmodell eine transparente Realisierung von Persistenz verspricht, zeigen sich jedoch beim praktischen Einsatz einige Einschränkungen. Diese resultieren hauptsächlich aus der notwendigen Voraussetzung, daß die Klassen sämtlicher in GemStone abgelegter Objekte auch in GemStone definiert sein müssen. Die automatische Generierung von Klassen zur Laufzeit kann hier nur bedingt eingesetzt werden, da dabei nur die Struktur der Objekte angelegt wird, nicht aber ihr Verhalten. Solange in der Datenbank nur die Zustände der Objekte gespeichert werden sollen, ist das nicht von Bedeutung. Sobald aber auch Methoden innerhalb der DB ausgeführt werden sollen, müssen sie manuell oder durch eine eigene Lösung (halb-) automatisch hinzugefügt werden. Das ist nicht nur dann notwendig, wenn Methoden eines solchen Objektes direkt in der Datenbank aufgerufen werden, sondern auch dann, wenn der Aufruf indirekt über eine Methode eines anderen in GemStone persistent abgelegten Objektes erfolgt. Falls dieser Sachverhalt nicht oder zu spät berücksichtigt wird, kann dies zur Laufzeit zu Fehlermeldungen führen. Weitere Gefahren bei der automatischen Generierung von Klassen ergeben sich im Zusammenhang mit dem Autorisierungsmodell[36] von GemStone, da standardmäßig keine gesonderten Zugriffsrechte für die Klasse berücksichtigt werden. Für eine so generierte Klasse gelten automatisch die gleichen Zugriffsrechte wie für die mit ihr in der gleichen Transaktion erzeugten Objekte. Falls für Klassendefinitionen andere Zugriffsrechte gelten sollen, müssen diese wie auch die fehlenden Methoden nachträglich festgelegt werden. Für alle diese Probleme können aber die in Smalltalk implementierten Mechanismen von GemStone den Anforderungen entsprechend angepaßt werden.
Neben den von Atkinson und Morrison skizzierten Vorteilen der transparenten Realisierung von Persistenz[37], die sich hauptsächlich in der Abstraktion von persistenzspezifischen Aspekten bei der Softwareentwicklung widerspiegelt, existiert eine Fragestellung, die in den bisherigen Ausführungen kaum berücksichtigt wurde. Innerhalb einer Anwendung kann nicht ausgeschlossen werden, daß ein persistentes Objekt kurzfristig, aber über mehrere Transaktionen hinweg, transiente Objekte referen-

[35] Eine Beschreibung des Konzepts der „orthogonalen Persistenz“ findet sich in Abschnitt 3.3.3.
[36] Die Autorisierung in GemStone wird ausführlicher in Abschnitt 6.4 behandelt, so daß an dieser Stelle nur so weit wie nötig darauf eingegangen werden soll.
[37] Siehe hierzu Abschnitt 3.3.3 und [AtMo95].

ziert. Als Beispiel dient an dieser Stelle das in Smalltalk implementierte MVC-Konzept[38]. Bei diesem Konzept wird zwischen den Klassen der Anwendungsdomäne (Model) und den Klassen zur Realisierung der Benutzungsoberfläche, repräsentiert durch die Views zur Visualisierung von Objekten und den Controllern zur Initiierung von Aktionen, unterschieden. Das Ziel ist eine Separierung der Anwendungsdomäne, die von den Domänenobjekten realisiert wird, von der Repräsentation gegenüber dem Benutzer des Systems. Dadurch kann ein Objekt dem Benutzer auf verschiedene Arten präsentiert werden, ohne daß die grundlegende Anwendungslogik innerhalb der Domänenobjekte davon betroffen ist. Um dabei dem Benutzer eine konsistente Sicht auf die Objekte zu ermöglichen, müssen die Objekte der Benutzungsoberfläche (Views und Controller) bei jeder Änderung des Domänenobjekts informiert werden, damit sie ihre Sicht auf das Objekt entsprechend aktualisieren können. Um dies zu gewährleisten, halten die Domänenobjekte über einen Abhängigkeitsmechanismus Referenzen auf alle ihre View-Objekte und informieren sie bei jeder Änderung des eigenen Zustands. Die Realisierung dieses Abhängigkeitsmechanismus erfolgt gemäß dem von Gamma et al. dokumentierten Beobachtermuster[39] (engl. observer pattern). Die View-Objekte können zwar mehrere Transaktionen überdauern, werden aber spätestens beim Beenden der Anwendung geschlossen, so daß ihr Zustand nicht persistent abgelegt werden muß. Da aber das ODBMS das Bestreben hat, bei einem commit sämtliche referenzierten Objekte persistent zu speichern, wozu zur Laufzeit der Anwendung auch die Objekte der Benutzungsoberfläche gehören, sind auch die zu diesem Zeitpunkt noch geöffneten und von den Domänenobjekten referenzierten Views und Controller betroffen. Um deren Speicherung vorzubeugen, ist in GemStone die Möglichkeit zum Ausblenden einzelner Attribute von der Persistenz vorhanden[40].

4. Zugriff auf persistente Objekte

Im Verlauf dieses Kapitels werden Mechanismen zum Zugriff auf in der Datenbank persistent abgelegte Objekte beschrieben.

4.1. Datenbankanfragen

Die herkömmliche Vorgehensweise in relationalen Systemen zum Zugriff auf persistente (Daten-) Objekte ist die Formulierung einer Anfrage in einer deklarativen Anfragesprache. Hierbei hat sich vor allem bei relationalen Systemen die Sprache SQL durchgesetzt. Auch einige heute verfügbare ODBMS beinhalten hierzu an SQL angelehnte aber proprietäre Anfragesprachen. Beispiele hierfür sind die ODBMS ObjectStore, O_2 oder Jasmine (vgl. [Lam91], S. 55, [Ban92], S. 256 ff. und [CAI98], S. 2-2). Bei diesen Systemen formuliert ein Benutzer eine Anfrage, die eine bestimmte Objektmenge aufgrund ihrer Attributwerte beschreibt. Diese Anfrage wird an den Datenbankserver geschickt, der sie auswertet und als Ergebnis die Menge der Objekte zurückgibt, auf die das Anfragekriterium zutrifft. Der Vorteil bei dieser Vorgehensweise ist, daß in der Anfrage nur beschrieben wird, wie das Ergebnis aussehen soll und nicht wie es berechnet wird. Der Datenbankserver kann selbst die optimale Ausführung der Anfrage bestimmen. Dieser Vorteil ist aber hinfällig, wenn innerhalb der Anfrage Methoden der Objekte ausgeführt werden sollen, da hier eine Optimierung nur schwer möglich ist (vgl. [MeWü97], S. 81 und [Heu97], S. 534). Des weiteren ist bei dieser Form des Zugriffs der Programmcode nicht mehr unabhängig von der Datenbank, da spezielle Datenbankoperationen eingefügt werden. Aus diesem Grund empfehlen auch Blaha und Premerlani nicht die Benutzung von Anfragen. Sie schlagen vor, sie nur zur Ermittlung eines Einstiegspunktes in den Objektgraphen in der Datenbank zu benutzen und ab dort navigierend auf die Objekte zuzugreifen (vgl. [BlPr98], S. 385).

---

[38] Die Abkürzung MVC steht für Model-View-Controller. Ein Überblick über dieses Konzept findet sich beispielsweise in [KrPo88] und [Parc94].

[39] Siehe hierzu [GHJV98] oder [Rie97], S. 82 ff. und S. 121 ff. .

[40] Dieser Mechanismus ist eigentlich innerhalb der Funktionalität der Schnittstelle zwischen Client-Anwendung und GemStone-Server, dem sog. GemBuilder realisiert, wird aber an dieser Stelle schon eingeführt, da hierbei die Notwendigkeit des Ausblendens einzelner Attribute verdeutlicht wird, was auch schon bei der Berücksichtigung des Persistenzmodells währen der Modellierung von Bedeutung ist.

4.2. Navigation über Objekte

Die meisten kommerziellen ODBMS realisieren die Navigation über persistente Objekte durch Kopieren der Objekte in den lokalen Arbeitsspeicher der Anwendung. Dementsprechend wird sich in der Literatur intensiv mit den technischen Realisierungsmöglichkeiten für das Kopieren der Objekte in den Anwendungsspeicher beschäftigt (z.B. [KeMo94], S. 525 ff., [Man94], S. 202 ff. und [Heu97], S. 533 ff.). Der folgende Teil des Berichtes beschäftigt sich weniger mit den physikalischen Aspekten solcher Realisierungsformen, sondern es wird vorwiegend auf die konzeptuellen Eigenschaften dieser Form des Zugriffs auf DB-Objekte eingegangen.

Gemäß den von Atkinson und Morrison formulierten und in Abschnitt 3.3 vorgestellten Prinzipien sollte auch der Zugriff auf persistente Objekte für den Entwickler möglichst transparent erfolgen. ODBMS, bei denen die Objekte mittels spezieller Leseoperationen aus der Datenbank geholt und nach eventuellen Änderungen wieder zurückgeschrieben werden müssen, bürden den Programmierern mehr Aufwand auf, als eigentlich notwendig wäre. Außerdem erzwingen sie, daß der Code nicht mehr unabhängig vom Speichermedium ist. Deswegen bieten die clientseitigen Laufzeitsysteme einiger ODBMS Mechanismen an, die beim Zugriff auf ein Attribut eines lokalen Objektes erkennen, ob sich das dort referenzierte Objekt bereits im Hauptspeicher befindet oder noch in der DB liegt. Falls letzteres der Fall sein sollte, sorgen diese Mechanismen dann für das Lesen des Objekts in den Hauptspeicher.

Hosking et al. stellen in einem technischen Bericht zwei Realisierungsmöglichkeiten eines transparenten Zugriffs[41] für Smalltalk vor (siehe [HMB90]). Beide Ansätze beruhen auf einem Eingriff in die Speicherverwaltung durch die Modifikation der virtuellen Maschine des Smalltalk-Systems. Beim ersten Ansatz werden die Objektzeiger so verändert, daß in ihnen die OID des Objekts und ein Statusbit enthalten sind, welches anzeigt, ob sich das Objekt im lokalen Speicher befindet oder nicht. Trifft die virtuelle Maschine auf einen Zeiger, der auf ein noch nicht vorhandenes Objekt zeigt, lädt sie es aus der Datenbank. Einen ähnlichen Weg beschreiten auch einige Hersteller kommerzieller ODBMS, die eine transparente Auflösung persistenter Objekte über spezielle Zeiger ermöglichen (vgl. [Man94], S. 204). Der zweite Ansatz aus [HMB90] führt Pseudoobjekte – sog. „fault blocks“ – ein, die stellvertretend für nicht lokale Objekte stehen. Verweist ein lokales Objekt auf solch einen „fault block“, entnimmt ihm die virtuelle Maschine die Information über die OID des eigentlichen Objektes und kopiert dieses in den Arbeitsspeicher. Die Grundstruktur hierbei ist die einer „smart reference“, entsprechend dem von Gamma et al. beschriebenen Stellvertretermuster (engl. proxy pattern) (vgl. [GHJV98]).

Nachdem Objekte in den lokalen Arbeitsspeicher kopiert wurden, stehen sie der Anwendung zur Verfügung. Sie kann den Objekten jetzt Nachrichten schicken, den Zustand auslesen, aber auch deren Zustände verändern. Falls dies passiert, muß das Laufzeitsystem des ODBMS auf dem Client dafür Sorge tragen, daß diese geänderten Objekte auch in der zentralen Datenbank aktualisiert werden. Da die Modifikation gelesener Objekte außerhalb der ODB durchgeführt wird, muß es über die Veränderung informiert werden. Manola verweist hierzu auf einige ODBMS, die eine spezielle Markierungsoperation bereitstellen, die in jeder, den Zustand eines Objektes modifizierenden Methode aufgerufen werden muß (vgl. [Man94], S. 206). Beim Beenden einer Transaktion sammelt das ODBMS alle auf diese Weise markierten Objekte und aktualisiert ihre geänderten Zustände in der Datenbank, um sie persistent und anderen Benutzern verfügbar zu machen. Hierbei fällt aber auf, daß der Aufruf einer Operation nicht mehr transparent für den Entwickler ist. Er ist vielmehr dafür verantwortlich, alle Methoden, die eine Zustandsänderung herbeiführen können, zu identifizieren und mit dem Aufruf der Markierungsoperation zu versehen. Ob das Benutzen einer Markierungsoperation, anstatt des expliziten Schreibens eines Objekts, mehr Transparenz aufweist und damit weniger Aufwand für den Programmierer bedeutet, bleibt fragwürdig[42].

Ein weiteres Problem in diesem Zusammenhang, welches nicht automatisch von einem ODBMS gelöst werden kann, ist die Fragestellung, wie viele Objekte gleichzeitig in einem Vorgang kopiert wer-

[41] Die Verfasser sprechen dabei von „object faulting“.

[42] Es sei an dieser Stelle soviel vorweggenommen, daß GemStone hierzu einen transparenten Mechanismus unterstützt. Mehr hierzu folgt in Abschnitt 4.4 des anschließenden Kapitels

den sollen. Das eine Extrem wäre, bei jedem Zugriff auf ein noch nicht lokal vorhandenes Objekt nur dieses eine zu lesen. Diese Lösung würde aber einen hohen Aufwand verursachen, da für jedes persistente Objekt der Kopiermechanismus angestoßen werden müßte. Da es außerdem sehr wahrscheinlich sein kann, daß nicht nur ein bestimmtes Objekt, sondern auch die von ihm referenzierten lokal benötigt werden, wäre es sicherlich sinnvoller, bei jedem Zugriff auf die Datenbank gleich mehrere Objekte zu übertragen. Demgegenüber kann aber ein unüberlegtes Übertragen von Objekten über mehrere Referenzierungsebenen hinweg darin resultieren, daß mehr Objekte gelesen werden, als eigentlich benötigt werden. Es gilt daher, hier einen Mittelweg zu finden, bei dem die Ressourcen optimal ausgenutzt werden. Falls die Übertragungseinheit zwischen Anwendungs-Client und Datenbankserver nicht Objekte sondern Speicherseiten sind (wie es bei der Seitenserver-Architektur der Fall ist), kann man versuchen, häufig gemeinsam benötigte Objekte auf einer Speicherseite zusammenzufassen (vgl. [Man94], S. 206). In der Literatur wird dies als **Cluster**-Bildung bezeichnet. Grundidee ist dabei, Objekte, die wahrscheinlich oft gemeinsam benötigt werden, auf einer Speicherseite in der Datenbank zusammenzufassen. Der Vorteil ist, daß das Lesen und die Übertragung einer Speicherseite ausreicht und die benötigten Objekte schneller verfügbar sind. Wären diese Objekte hingegen über mehrere Seiten verteilt, würde dies zu mehr Aufwand führen. Leider hat diese Lösung einen entscheidenden Nachteil: Jedes Objekt kann maximal in einem Cluster liegen. Da ein Objekt aber von mehreren anderen referenziert werden kann, ist die optimale Cluster-Bildung schwer zu entscheiden[43]. Kemper und Moerkotte stellen einige approximative Algorithmen (Heuristiken) vor, die eine annähernde Lösung berechnen können (vgl. [KeMo94], S. 459 ff.).
Die herausragende Eigenschaft dieser Form des Zugriffs auf persistente Objekte ist, daß die Methoden der Objekte auf dem Client ausgeführt werden. Der Datenbankserver ist nur dafür zuständig, die Zustände der Objekte persistent abzulegen und den Anwendungsprogrammen konkurrierend zur Verfügung zu stellen. Er hat daher keinerlei Informationen über die Methoden der Objekte, sondern muß nur die Struktur der Objekte kennen. Eine Ausführung von Methoden auf dem Server ist nicht möglich.

4.3. Aktive Datenbankobjekte

Eine weitere Möglichkeit des Zugriffs auf persistente Objekte in der Datenbank findet in der Literatur kaum Beachtung und wird auch nur in wenigen verfügbaren Systemen umgesetzt. Sie beruht darauf, daß nicht nur die Zustände von Objekten in der Datenbank abgelegt werden können. In einigen Datenbanksystemen werden auch ihre Methoden gespeichert und auf dem Datenbankserver ausgeführt. Möchte eine Anwendung einen Dienst eines solchen Objektes in Anspruch nehmen, muß das Objekt nicht mehr in den lokalen Speicher transferiert werden. Es reicht aus, die Nachricht an das Objekt in der Datenbank zu delegieren, die Methode dort abzuarbeiten und das Resultat an den dienstanfordernden Klienten zurückzuschicken. Analog zur Navigation stellt sich auch hier die Frage, wie transparent dies funktionieren kann. Manola geht nur am Rande auf die Delegation an entfernte Objekte ein und merkt hierzu an, daß die Applikation sich dabei des Zugriffs auf entfernte Objekte bewußt ist, d.h. der Aufruf nicht mehr transparent erfolgt (vgl. [Man94], S. 203). Im vorangegangenen Abschnitt wird ein Verfahren zur transparenten Replikation persistenter Objekte beschrieben. In diesem Zusammenhang ist es wenig einsichtig, wieso nicht auf ähnliche Weise anstatt einer lokalen Kopie ein Stellvertreterobjekt erzeugt werden kann. Dieser Stellvertreter übernimmt die Aufgabe, lokale Nachrichten entgegenzunehmen und über die verfügbare Kommunikationsinfrastruktur an das entfernte Objekt weiterzuleiten. Dies entspricht dem von Gamma et al. beschriebenen Muster des **remote proxy** (vgl. [GHJV98]). McCullough beschreibt hierzu eine mögliche Realisierung in Smalltalk (siehe [McC87]).
Die Möglichkeit, aktive Objekte in der Datenbank realisieren zu können, hängt von der Architektur des ODBMS ab (siehe Abschnitt 2). Sofern sie aber vorhanden ist, kann man auch die daraus resultierenden Vorteile nutzen. Im Gegensatz zur Navigation muß man sich hierbei keine Gedanken über geänderte Zustände der Objekte oder Cluster machen. Da die Methoden in der DB abgearbeitet werden und Änderungen der Objekte auch dort stattfinden, benötigt man keine Mechanismen zum Mar-

[43] Nach Kemper und Moerkotte ist dieses Problem sogar NP-hart, d.h. nicht mit polynomiell wachsendem Aufwand berechenbar(vgl. [KeMo94], S. 459).

kieren geänderter Objekte. Das ODBMS führt Änderungen selbst durch und muß deshalb auch nicht von solchen unterrichtet werden. Auch die Fragestellung, wie viele Objekte bei einem Zugriff übertragen werden müssen, hat hierbei keine Bedeutung, da sich die zur Befriedigung der Dienstanfrage benötigten Objekte bereits im erforderlichen Namensraum befinden. Dies ist der persistente Speicher, der vom ODBMS kontrolliert wird. Es werden nur die Objekte, die das Ergebnis der Anfrage darstellen, an den Client übermittelt. Des weiteren haben aktive DB-Objekte ihre Vorteile, wenn Methoden der Objekte öfters geändert werden. Dies kann zum Beispiel bei einem iterativen Entwicklungsprozeß geschehen oder bei Anwendungen, die einer starken Abhängigkeit gegenüber Änderungen gesetzlicher o.ä. Vorschriften unterliegen. Liegen in solchen Fällen die Methoden zusammen mit den Objekten in der zentralen Datenbank, müssen bei deren Änderung nicht die Programme der Anwender aktualisiert werden, sondern nur die Methoden in der Datenbank. Aufgrund der nicht vorhandenen Code-Redundanz werden Inkonsistenzen in der Implementierung vermieden. Dies kann eine organisatorische Vereinfachung bedeuten, wenn man bedenkt, daß evtl. mehrere hundert Anwendungen von solchen Veränderungen betroffen sein können.

4.4. Zugriff auf persistente Objekte in GemStone

Der Zugriff auf persistente Objekte in GemStone wir durch Laufzeitumgebungen realisiert, die in die Client-Applikation eingebunden sind. Diese liegt unter dem Namen GemBuilder vor.

4.4.1. GemBuilder für VisualWorks Smalltalk

Die Anbindung einer in Smalltalk entwickelten Anwendung wird durch den von GemStone bereitgestellten GemBuilder realisiert. Diese Komponente stellt die Erweiterung der Smalltalk-Entwicklungsumgebung VisualWorks dar. Sie erweitert die vorhandenen Klassen und Objekte um das für die DB-Funktionalität notwendige Protokoll und stellt die Funktionalität für das transparente Lesen und Schreiben von Objekte aus und nach GemStone bereit. Abstrahiert man von der in Abschnitt 2.5.2 beschriebenen physikalischen Verteilung zwischen Client und Server stellt sich dem Entwickler die Umgebung auf konzeptueller Ebene wie in Abbildung 7 skizziert dar. Das Rechteck auf der rechten Seite der Abbildung repräsentiert eine GemStone-Datenbank mit den darin gespeicherten persistenten Objekten, die durch die Kreise symbolisiert sind. Das Wurzelobjekt zum Einstieg in das Objektnetz in der Datenbank ist durch eine dicke Umrandung gekennzeichnet. Die Pfeile zwischen den Objekten stehen für Referenzen zwischen einem Objekt auf die von ihm referenzierten Objekte. Auf der linken Seite der Abbildung ist durch einen geteilten Kasten das Smalltalk-Image mit dem integrierten GemBuilder dargestellt. Auch in diesem Teil werden Objekte durch Kreise und die Referenzen durch Pfeile symbolisiert. Der dick umrandete Kreis stellt das Gegenstück zur persistenten Wurzel in der Datenbank dar.

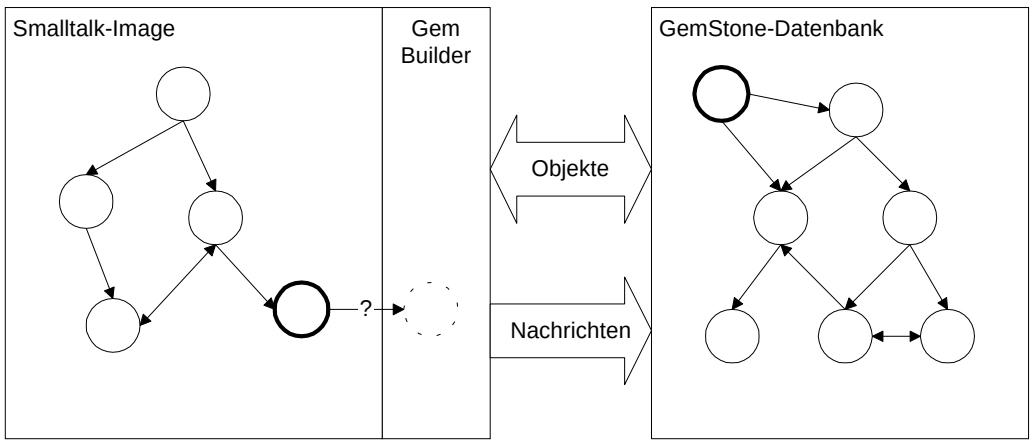

Abbildung 7: Integration der Namensräume von Smalltalk und GemStone

Der GemBuilder dient als Schnittstelle zwischen der Smalltalk-Anwendung auf dem Client und der GemStone-Datenbank auf dem Server. Dabei ist zu beachten, daß es sich dabei um zwei verschiedene

Namensräume handelt. Der Namensraum der Datenbank fällt in die Zuständigkeit von GemStone, wohingegen der auf dem Client von der Smalltalk-Umgebung resp. der dortigen virtuellen Maschine verwaltet wird.
Die Beschreibung der Integration beider Namensräume durch den GemBuilder gliedert sich in den folgenden Abschnitten in drei Teile: In Abschnitt 4.4.1.1 werden zunächst die Mechanismen beschrieben, die den Zugriff auf das persistente Wurzelobjekt des Objektnetzes in der Datenbank realisieren. In den daran anschließenden Abschnitten werden die Möglichkeiten des Zugriffs auf die von der Wurzel erreichbaren Objekte erörtert. Dabei wird zwischen zwei Alternativen unterschieden. Zum einen können die persistenten Objekte durch lokale Kopien auf dem Client verfügbar gemacht werden. Diesem Aspekt widmet sich Abschnitt 4.4.1.2. GemStone bietet auch die Möglichkeit, Dienste der persistenten Objekte auf dem Client zur Verfügung zu stellen, indem Nachrichten direkt an diese Objekte in GemStone geschickt werden (siehe auch Abbildung 7). Die zugehörigen Methoden werden dann innerhalb von GemStone ausgeführt und das Resultat an das aufrufende Objekt auf dem Client zurückgeschickt. Die dies realisierenden Mechanismen werden ausführlicher in Abschnitt 4.4.1.3 behandelt.

4.4.1.1. Zugriff auf die persistente Wurzel in GemStone

Der erste Schritt bei der Entwicklung einer Anwendung, deren Objekte in GemStone persistent verwaltet werden sollen, ist die Festlegung des Einstiegspunktes in die Datenbank, das sog. persistente Wurzelobjekt. In der GemStone-Dokumentation werden hierzu als allgemein gebräuchliche Arten von Wurzelobjekten globale Variablen im Smalltalk-Dictionary, Klassenvariablen und Klasseninstanzvariablen genannt (vgl. [Gem96c], S. 3-5). Zur Realisierung des Zugriffs auf solche Wurzelobjekte stellt GemStone resp. der GemBuilder für Smalltalk spezielle Objekte zur Verfügung, sog. **Konnektoren** (engl.: connector). Ein Konnektor legt fest, wie ein lokales Smalltalk-Objekt und sein Pendant in GemStone zum Zeitpunkt des Einloggens in die Datenbank verbunden werden sollen (vgl. [Gem96c], S, 3-5). Dabei unterscheidet der GemBuilder fünf Typen von Konnektoren, wovon drei der Festlegung des Einstiegspunktes in die Datenbank dienen (vgl. [Gem96c], S. 3-5).
Ein **Name Connector** verbindet zwei Objekte anhand von festgelegten Bezeichnern im Client-Smalltalk und in GemStone. Die Zuordnung eines Objektes zu einem bestimmten Bezeichner erfolgt dabei in Smalltalk durch Ablegen des Objektes unter diesem Bezeichner im Smalltalk-Dictionary, welches alle bzgl. des Images global verfügbaren Objekte verwaltet. In GemStone wird dieser Bezeichner innerhalb eines Symbolverzeichnisses in der Symbolliste des jeweiligen Benutzers festgelegt. Nach dem Einloggen des Benutzers stellt der Konnektor automatisch die Verbindung zwischen diesen beiden Objekten her, indem er sie im Smalltalk-Dictionary der Client-Anwendung und in der Symbolliste dieses Benutzers nachschlägt. Die beiden Bezeichner müssen dazu aber nicht identisch sein.
Ein **Class Connector** löst die Verbindung zwischen zwei Objekte auf die gleiche Weise auf wie ein Name Connector, wobei die zwei zu verbindenden Objekte aber Klassen sein müssen. Er dient nicht der Festlegung eines Wurzelobjektes sondern nur der Zuordnung von Klassendefinitionen der Smalltalk-Anwendung auf dem Client zu den jeweiligen Klassen in GemStone. Standardmäßig trifft GemStone die Zuordnung zweier Klassen automatisch anhand der Namengleichheit, wobei aber auch Ausnahmen manuell berücksichtigt werden können. Es besteht die Möglichkeit, daß ein Objekt auf dem Client und dem Server jeweils einer unterschiedlichen Klasse angehört. Diese Abbildung wird durch einen Class Connector hergestellt. Ein Beispiel hierfür ist die Klasse Timestamp, die in GemStone keine namensgleiche Realisierung besitzt. Durch einen Class Connector werden alle Instanzen der Klasse Timestamp in VisualWorks bei der Übertragung nach GemStone auf Instanzen der in GemStone definierten Klasse DateTime abgebildet und umgekehrt.
Ein **Class Variable Connector** (resp. ein **Class Instance Variable Connector**) schlägt ebenfalls zwei Klassen im Client-Smalltalk und GemStone nach, verbindet aber dann die entsprechend angegebenen Klassenvariablen (resp. Klasseninstanzvariablen) dieser Klassen. Mit diesen zwei Konnektoren können ähnlich wie mit einem Name Connector Wurzelobjekte in der Datenbank festgelegt werden.
Im Gegensatz zu den bisher beschriebenen Konnektoren verbindet ein **Fast Connector** zwei Objekte nicht anhand von festgelegten Bezeichnern sondern direkt über die eindeutigen OID. Wie der Name

schon ausdrückt ist diese Art der Verbindung schneller als die anderen, da hier das Nachschlagen der Objekte in den Verzeichnissen entfällt und sich somit vor allem bei der Etablierung einer großen Menge von Konnektoren ein Geschwindigkeitsvorteil ergibt. Andererseits ist hierbei aber sicherzustellen, daß das zu verbindende Objekt niemals ausgetauscht wird, da sich hierdurch auch die Identität ändert. Dem Geschwindigkeitsvorteil steht hier somit ein Mangel an Flexibilität und Sicherheit gegenüber, so daß vor allem während der Entwicklungsphase einer Anwendung Fast Connectoren nicht eingesetzt werden sollten (vgl. [Gem96c], S. 3-8). In GemStone dienen diese Konnektoren der Verbindung der vordefinierten Basisklassen.
Jedem Konnektor ist zusätzlich noch eine Aktion zugeordnet, die unmittelbar nach dem Verbinden der Objekte ausgeführt werden soll. Eine solche Aktion wird in GemStone **Postconnect Action** genannt. Mit der Postconnect Action wird festgelegt, ob eines der Objekte nach dem Verbinden aktualisiert werden soll. Normalerweise sollen nach dem Einloggen in GemStone die Zustände der persistenten Objekte in der Anwendung auf dem Client verfügbar sein. Dies wird durch die Postconnect Action **updateST** erreicht. Hierbei wird nach dem Einloggen dem Objekt auf dem Client der Zustand des Objektes in GemStone zugewiesen. Umgekehrt besteht aber auch die Möglichkeit nach dem Einloggen den Zustand eines GemStone-Objekts durch den des Client-Objekts zu aktualisieren (Postconnect Action **updateGS**). Bei diesen beiden Aktionen wird das entsprechende Objekt durch eine Kopie in dem jeweils anderen Namensraum zur Verfügung gestellt. Soll aber nicht der Zustand des Objekts zwischen Anwendung und GemStone transferiert werden, sondern nur die Möglichkeit geschaffen werden, Nachrichten an persistente Objekte zu schicken, muß keine lokale Kopie des Objektes erstellt werden, sondern nur ein Forwarder (siehe nächster Abschnitt), der die lokale Nachricht an das GemStone-Objekt weiterleitet. Dies geschieht bei einem Konnektor durch die Postconnect Action **forwarder**. Falls nach dem Einloggen keine Aktion durchgeführt werden soll, damit die Objekte so verbleiben wie sie sind, kann dies durch die Postconnect Action **none** festgelegt werden. Hierbei verbleiben die Objekte in dem Zustand, den sie schon vor dem Einloggen hatten und es wird auch kein Forwarder erstellt. Dies wird bei den Class Connectoren eingesetzt, da bei ihnen keine Aktualisierungen durchgeführt werden sollen. Es werden nur Beziehungen zwischen korrespondierenden Klassendefinitionen festgelegt.

4.4.1.2. Navigation über persistente Objekte in GemStone

Die Navigation über Objekte macht es erforderlich, daß sie innerhalb der Smalltalk-Anwendung auf dem Client verfügbar gemacht werden müssen. In der Client-Anwendung werden lokale Kopien der persistenten Objekte erzeugt. Im Sprachgebrauch von GemStone wird eine solche Kopie **replicate** genannt. In Anlehnung an diese Terminologie wird im folgenden der deutsche Begriff **Replikat** zur Bezeichnung einer lokalen Kopie benutzt und der Vorgang mit dem Verb **replizieren** bezeichnet. Von der Replikation sind nur die Zustände der Objekte betroffen. Das Verhalten der Replikate wird durch die Klassen der lokalen VisualWorks-Anwendung definiert. Die Methoden der Objekte werden somit nur auf dem Client ausgeführt, weswegen die Definition der Methoden in GemStone nicht nötig ist resp. dort definierte Methoden nicht berücksichtigt werden.

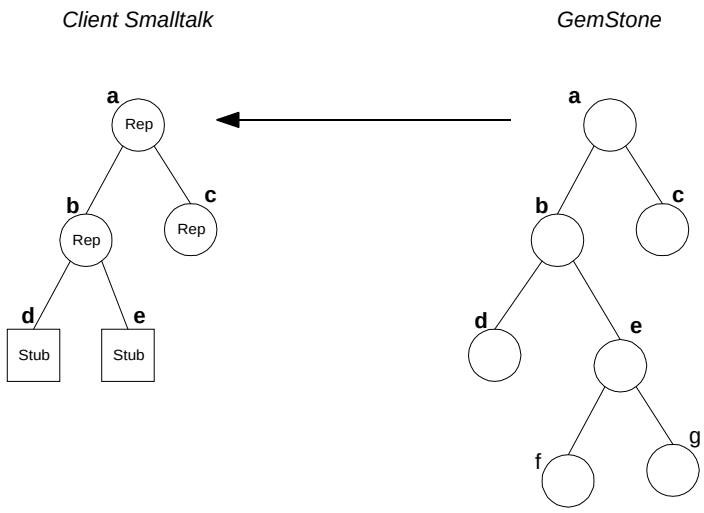

Abbildung 8: Replizieren von Objekten über zwei Ebenen (aus [Gem96c], S. 4-6)

Nach dem Einloggen in GemStone wird zunächst das Wurzelobjekt in der Datenbank in der Smalltalk-Anwendung auf dem Client repliziert (sofern im Konnektor updateST als Postconnect Action festgelegt ist). Eine Veranschaulichung dieses Vorgangs findet sich in Abbildung 8. Auf der rechten Seite sind schematisch die Objekte in der GemStone-Datenbank durch Kreise dargestellt. Auf der linken Seite der Abbildung sieht man das bereits replizierte Wurzelobjekt (gekennzeichnet durch den Buchstaben a) und die zwei direkt von ihm referenzierten Objekte b und c. Innerhalb der Darstellung wird bei der Replikation eine Tiefe von zwei Objekten[44] angenommen. Dies bedeutet, daß das benötigte Objekt selbst und die direkt von ihm referenzierten Objekte gemeinsam repliziert werden. Alle weiteren indirekt referenzierten Objekte werden nicht repliziert (in Abbildung 8 sind dies die von b referenzierten Objekte d und e). Um bei Bedarf auch auf diese zugreifen zu können, hat der GemBuilder an den entsprechenden Referenzen jeweils ein Stellvertreterobjekt, einen sog. **Stub**, erzeugt, der in der Darstellung durch ein Rechteck repräsentiert wird.

Dieses Stellvertreterobjekt hat die Aufgabe, das durch sie repräsentierte Objekt in GemStone bei Bedarf in der Client-Anwendung zu replizieren. Dabei sollte das Replizieren aber möglichst transparent stattfinden, d.h. man sollte von der Existenz etwaiger Stubs abstrahieren können. Der Vorgang der Replikation wird eingeleitet, sobald eine Nachricht an das vermeintlich an der Stelle des Stubs vorhandene Objekt geschickt wird. Dies ist beispielhaft in Abbildung 9, die auf der Ausgangssituation in Abbildung 8 aufbaut, dargestellt. Dort wird eine Nachricht an das Objekt e gesendet, welches auf dem Client aber nur in Form seines Stellvertreters vorhanden ist. Dieser Stub nimmt die Nachricht entgegen, repliziert das benötigte Objekt mitsamt der direkt von diesem referenzierten Objekte und setzt das Objekt e an seiner eigenen Stelle ein. Zum Abschluß wird die ursprüngliche Nachricht an das Objekt weitergeleitet, um die zugehörige Methode abzuarbeiten. Die Erzeugung von Stubs setzt sich immer weiter fort, solange noch referenzierte Objekte in GemStone liegen, die aber noch nicht repliziert worden sind.

[44] Die Tiefe, die angibt wie viele Objekte in einem Replikationsvorgang übertragen werden sollen, ist einstellbar. An dieser Stelle wurde aufgrund der anschaulicheren Darstellung der Standardwert aus GemStone übernommen.

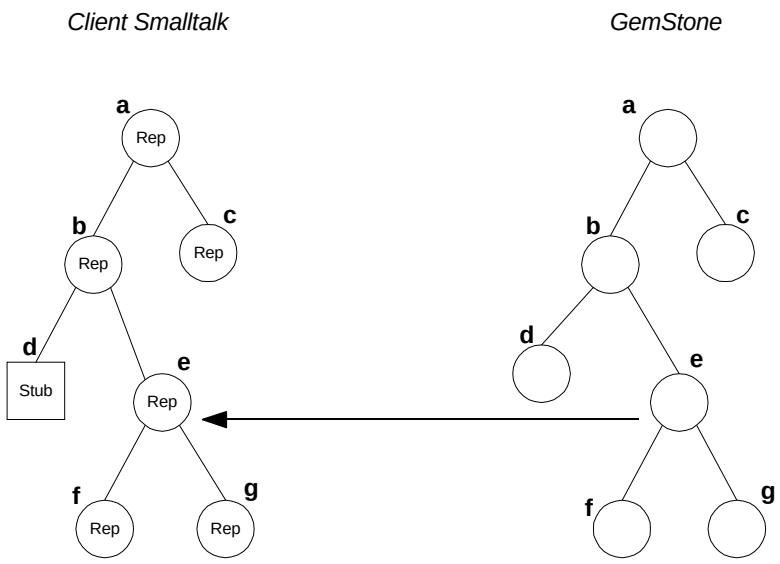

Abbildung 9: Replikation nach dem Senden einer Nachricht (aus [Gem96c], S. 4-7)

In der bisherigen Beschreibung wird davon ausgegangen, daß nur das benötigte Objekt inklusive seiner direkt referenzierten Objekte repliziert wird, d.h. die Tiefe der Replikation zwei Ebenen beträgt. Falls eine andere Replikationstiefe innerhalb einer Smalltalk-Anwendung gewünscht ist, hat man zwei Möglichkeiten sie einzustellen. Zum einen kann sie innerhalb des GemBuilders eines Smalltalk-Systems global festgelegt werden, wobei dies jedoch sehr inflexibel ist, da dieser Wert für alle Replikationen in diesem Smalltalk-Image angenommen wird. Zum anderen kann die Tiefe der Replikation für jedes einzelne Attribut der Objekte einer Klasse definiert werden. Hierzu wird in einer Klassenmethode namens replicationSpec für jedes Attribut der Instanzen dieser Klasse festgelegt, ob das entsprechende Attribut nur einen Stub enthalten, ein Forwarder sein soll oder Minimal- und Maximalwerte für die Replikationstiefe festgelegt werden sollen. Eine Beschreibung aller möglichen Einträge enthält die GemStone-Dokumentation (siehe [Gem96c], S. 4-9 f.).

Nachdem Objekte repliziert worden sind befinden sie sich im Namensraum der lokalen Smalltalk-Anwendung, so daß alle Änderungen an diesen Objekten im Einflußbereich dieser Anwendung, also außerhalb von GemStone, durchgeführt werden. Solche lokal modifizierten Objekte werden im Sprachgebrauch von GemStone als „**dirty“** bezeichnet und alle modifizierten Objekte müssen als „dirty“ markiert werden. Dies kann entweder manuell durch Senden der Nachricht markDirty an das betroffene Objekt oder automatisch geschehen. Das manuelle Markieren hat dabei den Nachteil, daß innerhalb von Methoden jede Änderung des Zustands eines Objektes um das Markieren als dirty ergänzt werden muß. Wird dies ausgelassen oder vergessen, kann der GemBuilder diese Modifikation nicht registrieren und beim Beenden der Transaktion wird das Objekt nicht in GemStone aktualisiert. Deshalb empfiehlt sich die Möglichkeit des automatischen Markierens modifizierter Objekte. Hierbei wird bei den Objekten zwischen Kollektionen und Objekten mit Instanzvariablen unterschieden. Damit soll nicht ausgeschlossen sein, daß Kollektionen auch Objekte sind oder Kollektionen Instanzvariablen besitzen können. Die Unterscheidung wird nur aufgrund des speziellen Protokolls der Kollektionen getroffen. Die Möglichkeit des automatischen Markierens als „dirty“ wird in VisualWorks mittels der durch den GemBuilder definierten Klassenmethoden markDirtyOnAtPut und markDirtyOnInstVarAssign bereitgestellt, wobei erstere für Kollektionen zuständig ist und letztere für Objekte mit Instanzvariablen. Nach dem Aufruf der Klassenmethode markDirtyOnAtPut einer Kollektionsklasse bedeutet dies für alle Instanzen dieser Klasse, daß sie nach dem Senden der Nachricht at:put: automatisch als dirty markiert werden. Dies funktioniert in Smalltalk, da sämtliche Methoden, die den Zustand einer Kollektion ändern, dies letztendlich über die Methode at:put: realisieren. Für alle anderen Klassen verwirklicht die Klassenmethode markDirtyOnInstVarAssign ein automatisches Markieren einer Instanz dieser Klasse als „dirty“, wenn einer ihrer Instanzvariablen ein neues Objekt zugewiesen wird. Für Objekte der Klassen, die als Kollektion einerseits die Nachricht at:put: verstehen aber zusätzliche Instanzvariablen definieren, existiert auch eine Kombination von markDirtyOnInstVarAssign und mark-

DirtyOnAtPut. Alle automatischen Mechanismen zum Markieren als „dirty“ gelten sowohl für die Klasse, der eine entsprechende Nachricht geschickt wird, als auch für alle ihre Unterklassen. Die von Oberklassen geerbten Attribute sind jedoch nicht eingeschlossen.
Abschließend wird an dieser Stelle noch ein weiterer Aspekt erörtert, der neben der Replikation von persistenten Objekten und deren Aktualisierung nach einer Modifikation auf dem Client einen nicht unerheblichen Einfluß auf die Integration von VisualWorks und GemStone hat. Bedingt durch die zwei getrennten Namensräume (VisualWorks und GemStone) liegen dort auch jeweils die Klassen der Objekte ab, wobei diese aber nicht notwendigerweise äquivalent sein müssen. Bedingt durch die Replikation der Objekte müssen in GemStone innerhalb der Klasse nur die Attribute der Objekte definiert werden, die Methoden aber nicht. Wie bereits in Abschnitt 4.4.1.1 erwähnt, wird die Korrelation zwischen einer VisualWorks-Klasse und ihrem Pendant in GemStone durch einen Class Connector hergestellt. Dieser Konnektor legt für alle Instanzen einer Klasse in GemStone die zugehörige VisualWorks-Klasse dieser Objekte bei der Replikation nach VisualWorks und umgekehrt fest. Gemäß des intensionalen Klassenbegriffs in Smalltalk legt eine Klasse die Struktur und das Verhalten aller ihrer Instanzen fest, wodurch auch die Betrachtung der Class Connectoren von besonderer Bedeutung ist. Üblicherweise kann davon ausgegangen werden, daß zumindest die Namen und Attributdefinitionen der zu verbindenden Klassen gleich sind, weswegen GemStone die Option bietet, Class Connectoren, die noch nicht vorhanden sind, zur Laufzeit automatisch erstellen zu lassen. Dabei wird für zwei Klassen, die in VisualWorks und GemStone jeweils den gleichen Namen haben, automatisch ein Class Connector erzeugt. Analog hierzu ist das Kriterium beim Verbinden der Attribute auch die Namengleichheit. Somit hat man mit der automatischen Generierung von Class Connectoren einen Mechanismus der transparenten Abbildung von VisualWorks-Objekten auf die korrespondierenden persistenten Objekte in GemStone unter Berücksichtigung ihrer Klassenzugehörigkeit. Wie aber schon innerhalb des die Class Connectoren betreffenden Absatzes in Abschnitt 4.4.1.1 beschrieben, muß diese Abbildung nicht immer gültig sein. Falls ein Objekt in GemStone eine andere Klassenzugehörigkeit haben soll als in VisualWorks, muß dies über einen manuell angelegten Class Connector festgelegt werden. Ebensowenig müssen auch die Namen und die Anzahl der definierten Attribute beider Klassen übereinstimmen. Dies zeigt sich u.a. in der von GemStone vorgegebenen Klassenhierarchie, die sich in einigen Teilen von der in VisualWorks unterscheidet.
In Abschnitt 3.4.2 wurde bereits die Notwendigkeit für das Ausblenden bestimmter Attribute der Client-Objekte von der Persistenz angedeutet, wofür in GemStone prinzipiell zwei Möglichkeiten existieren. Zum einen können hierzu die Klassen in GemStone so definiert werden, daß sie alle Attribute der entsprechenden VisualWorks-Klasse besitzen, bis auf die, die nicht bei der Persistenz durch Erreichbarkeit berücksichtigt werden sollen. Bei einem commit werden beim persistenten Ablegen der Instanzen dieser Klasse nur die Attribute berücksichtigt, die auch in der GemStone-Klasse definiert sind. Die dort nicht vorhandenen Attribute werden standardmäßig ausgelassen. In Ausnahmefällen können Attribute aber unterschiedlich benannt sein, was auch in GemStone berücksichtigt wird. Eine flexible Art der Zuordnung von Attributen einer VisualWorks-Klasse zu denen der zugehörigen Klasse in GemStone bietet die Implementierung der Klassenmethode instVarMap in der VisualWorks-Klasse:

```
ClientClass class>>instVarMap

        ^super instVarMap,
                #( (clientVariable1 gsVariable1)
                        (clientVariable2 nil)
                        (nil gsVariable2) )
```

Diese Methode legt fest, daß die „instVarMap“ der Oberklasse übernommen werden soll und fügt die eigenen Festlegungen innerhalb eines Array hinzu. Ein Element dieses Array besteht aus einem Paar von Attributnamen, wobei der linke Bezeichner für einen Attributnamen der VisualWorks-Klasse steht und der rechte für einen der Klasse in GemStone. Im obigen Beispiel legt das erste Element fest, daß das Attribut clientVariable1 der VisualWorks-Klasse dem Attribut gsVariable1 der Klasse in GemStone zugewiesen werden soll und dient damit der Zuordnung unterschiedlich benannter Attribute. Alle Attribute der Klasse, die innerhalb der instVarMap nicht aufgeführt sind, werden aufgrund von Namengleichheit korreliert. Des weiteren kann man innerhalb der Methode instVarMap auch die Attri-

bute angeben, die in dem jeweils anderen Namensraum keine Entsprechung haben sollen. Beispielsweise zeigt das zweite Element der oben dargestellten Methode an, daß das Attribut clientVariable2 keine Entsprechung in GemStone hat, da hier mit nil das undefinierte Objekt zugeordnet wird. Das dritte und letzte Element des Array legt genau den umgekehrten Fall fest und zwar, daß das Attribut gsVariable2 keine Entsprechung in VisualWorks hat. Auf diese Weise können auch Attribute, die zwar in der VisualWorks-Klasse und der GemStone-Klasse den gleichen Namen haben, aber bei der Replikation von Objekten nicht berücksichtigt werden sollen, von der Zuordnung ausgeschlossen werden.

4.4.1.3. Aktive Objekte in GemStone

Neben der Replikation, bei der nur die Zustände persistenter Objekte in einer VisualWorks-Anwendung verfügbar gemacht werden, besteht auch die Möglichkeit das Verhalten der Objekte innerhalb von GemStone zu nutzen. Bei dieser Form des Zugriffs auf persistente Objekte werden nicht deren Zustände auf dem Client repliziert sondern client-seitige Nachrichten an das entsprechende Objekt in GemStone weitergeleitet. Dabei müssen, im Gegensatz zur Replikation, neben den Attributen der Objekte auch deren Methoden innerhalb der Klassen in GemStone definiert sein.

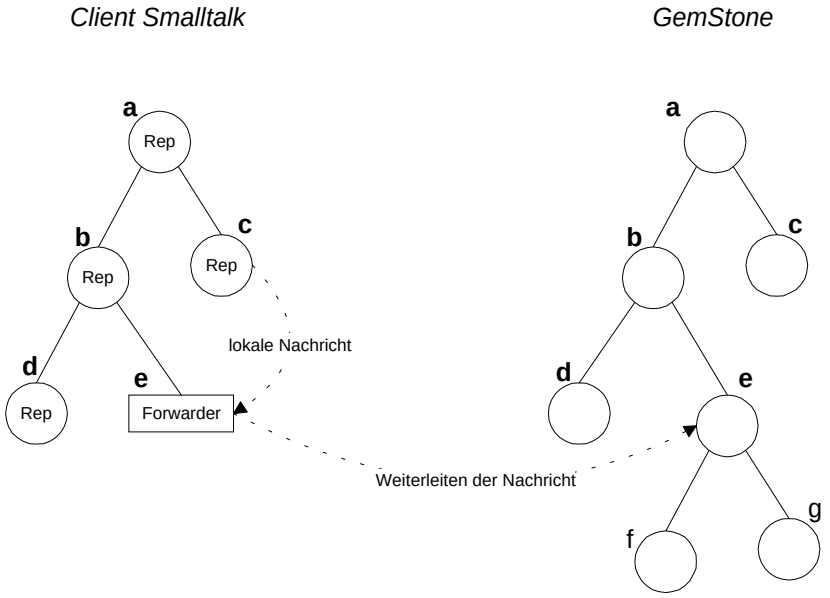

Abbildung 10: Aufruf von Methoden über Forwarder

Das Weiterleiten von Nachrichten an persistente Objekte in GemStone wird innerhalb der VisualWorks-Anwendung durch Stellvertreterobjekte, sog. **Forwarder**, realisiert. Wie in Abbildung 10 skizziert, ist in der VisualWorks-Anwendung anstelle eines Replikats von Objekt e ein Forwarder erzeugt worden. Dieser Stellvertreter nimmt lokale Nachrichten beliebiger Objekte auf dem Client entgegen und leitet sie mit den Parametern an das zugehörige Objekt in GemStone weiter. Sofern die der Nachricht entsprechende Methode für das GemStone-Objekt definiert ist, wird sie dort ausgeführt. Das Resultat wird nach deren Ausführung in GemStone über den Forwarder an das aufrufende Objekt in VisualWorks übermittelt. Der Forwarder übernimmt dabei die Aufgabe eines Stellvertreters, der für den transparenten Austausch von Nachrichten und Objekten zuständig ist, wobei Objekte als Resultat der ausgeführten Methode und als Parameter der Nachricht auftauchen können.
Bei der Übergabe der Parameter an ein persistentes Objekt in GemStone existiert jedoch eine Einschränkung, die sich besonders auf die Benutzung von Kollektions-Objekten in GemStone auswirkt. Smalltalk-Blöcke können bei der Parameterübergabe nur eingeschränkt genutzt werden, da bei ihrer Übermittlung der Kontext verlorengeht. D.h. Objekte, die nicht innerhalb eines Blockes definiert sind, werden nicht nach GemStone übertragen. Die Ursache für dieses Verhalten liegt darin, daß nicht der Block als Bytecode übermittelt wird sondern als Zeichenkette von Smalltalk-Ausdrücken. In GemStone wird diese Zeichenkette in das Binärformat der Datenbank übersetzt. Daraus folgt, daß alle in dem Block benutzten Bezeichner in GemStone oder innerhalb des Blocks definiert sein müssen. Dies kann jedoch nur für temporäre Variablen des Blocks oder globale Variablen realisiert werden (vgl.

[Gem96c], S. 4-20 f.). Blöcke sind jedoch ein essentieller Bestandteil bei der Nutzung von Kollektionen in Smalltalk, da die Methoden zur Iteration über eine Kollektion im allgemeinen einen Block als Parameter erwarten. In einem konkreten Beispiel bedeutet dies, daß folgende Methode der VisualWorks-Klasse Person nicht möglich wäre:

```
Person>>findEqualPersonIn: aGSOrderedCollection

    ^aGSOrderedCollection detect: [ :each | each name = self name]
```

Die Methode findEqualPersonIn: der Klasse Person soll in einer als Parameter übergebenen Kollektion nach einer Person mit dem gleichen Namen wie ihr eigener suchen. Die Kollektion ist in diesem Fall eine OrderedCollection in GemStone, auf die von VisualWorks über einen Forwarder zugegriffen wird (aGSOrderedCollection). Zum Auffinden dieses Personenobjektes wird die Methode detect:, die zum Standardprotokoll jeder Kollektion gehört, genutzt. Der Parameter der Methode ist ein Block, in dem das Kriterium formuliert ist, welches das gesuchte Objekt beschreibt. Dieses ist innerhalb der oben formulierten Methode die Gleichheit des Namens des innerhalb eines Iterationsschrittes betrachteten Objektes – each name – und des Namens des aktuellen Objektes, welches die Methode ausführt – self name. Sobald das gesuchte Personenobjekt gefunden ist, wird die Iteration über die Kollektion aGSOrderedCollection abgebrochen und das Resultat zurückgegeben. Das Problem bei dem Einsatz eines Forwarders liegt darin, daß der Kontext des Blocks nicht nach GemStone übertragen wird. In dem Beispiel ist es die Pseudovariable self, die zum Kontext des Blocks gehört. Allgemein sind dies nicht nur self, sondern darüber hinaus auch in der Methode definierte temporäre Variablen, die Attribute des die Methode implementierenden Objektes sowie die Pseudovariable super. Falls ein solches Objekt in einem übertragenen Block benutzt wird, reagiert das System beim Übersetzen mit einer Fehlermeldung. Da diese Einschränkung einen erheblichen Einfluß auf die Nutzung von Forwardern hat, wird in der GemStone-Dokumentation eine Lösung für dieses Problem beschrieben. Diese Lösung beruht im Prinzip auf der Übergabe der für den Block relevanten Objekte als eigenständige Parameter. Dazu müssen die benötigten Methoden in den Klassen in GemStone neu implementiert werden, so daß deren Signatur die zusätzliche Übergabe der Objekte ermöglicht. Erläuterungen und Beispiele hierzu finden sich in [Gem96c] auf Seite 4-21.

Wie bereits in Abbildung 10 angedeutet, ist innerhalb einer einzigen Anwendung der gemischte Einsatz von Replikaten und Forwardern möglich. Bedingt durch diese beiden verschiedenen Alternativen des Zugriffs auf persistente Objekte in GemStone sind Möglichkeiten der Definition der jeweils gewünschten Art des Zugriffs erforderlich. Standardmäßig ist in GemStone die Replikation der persistenten Objekte vorgesehen, wobei nur die Tiefe der Replikation vorgegeben werden kann. Die Festlegung, welche Objekte alternativ zur Replikation über einen Forwarder direkt in GemStone angesprochen werden sollen, kann in VisualWorks durch die Mechanismen des GemBuilder auf verschiedene Weisen geschehen. Zum einen kann schon bei einem Konnektor die Postconnect Action forwarder festgelegt werden, wodurch das durch den Konnektor angebundene GemStone-Objekt über einen Forwarder angesprochen wird (siehe auch Abschnitt 4.4.1.1). Außerdem kann auch die bereits in Abschnitt 4.4.1.2 vorgestellte Klassenmethode replicationSpec zur Festlegung von Attributen, auf die über einen Forwarder zugegriffen werden soll, verwendet werden. Letztendlich kann aber auch jedes bereits replizierte Objekt durch das Senden der Nachricht asForwarder durch einen Forwarder ersetzt werden.

4.4.2. Beurteilung der Konzepte zum Zugriff auf persistente Objekte

Eine herausragende Eigenschaft von GemStone ist die Möglichkeit der Ausführung der Methoden von Objekten innerhalb der Datenbank. Mittels der berechenbarkeitsvollständigen Sprache Smalltalk können in der Datenbank beliebige Methoden der Objekte implementiert und auch ausgeführt werden. Hierdurch ist der Zugriff auf persistente Objekte in GemStone nicht nur auf die Navigation über Objekte beschränkt. Es können vielmehr beliebige Anfragen über die Dienste der in der Datenbank implementierten Objekte realisiert werden, wobei auch Methodenaufrufe innerhalb dieser Anfragen möglich sind. Auch ein gemischter Einsatz von Replikaten und Forwardern innerhalb einer Client-Anwendung ist durch den Einsatz des GemBuilder erlaubt. Hierdurch ist eine Verteilung der Methodenausführung zwischen Anwendungs-Client und GemStone Server möglich, wobei eine Reduzierung

der über das Kommunikationsnetz übertragenen Objekte angestrebt werden kann, um dadurch die Performanz des Anwendungssystems zu erhöhen.
Bedingt durch die äquivalente Realisierung der objektorientierten Konzepte ergibt sich durch den GemBuilder ein transparenter Zugriff auf die persistenten Objekte in GemStone. Sowohl die Mechanismen zur Replikation referenzierter Objekte als auch das Erzeugen und die Nutzung von Forwardern sind direkt in die Programmiersprache Smalltalk integriert, so daß Entwickler weitestgehend von ihnen abstrahieren können. Dies zeigt sich insbesondere darin, daß innerhalb des Programmcodes im günstigsten Fall keine speziellen Operationen zum Lesen von Objekten aus der Datenbank, zum Schreiben neuer oder modifizierter Objekte in die Datenbank oder zum Weiterleiten von Nachrichten an persistente Objekte in der Datenbank nötig sind. Die Replikation wird automatisch durch das Senden einer Nachricht an einen vorher angelegten Stub eingeleitet. Dabei erfolgt auch das Erzeugen von Stubs anstelle von Objekten, die an der Grenze der Replikationstiefe liegen und somit noch nicht von der Replikation betroffen sind, transparent. Des weiteren kann auch die Aktualisierung modifizierter Replikate durch die Mechanismen des automatischen Markierens als „dirty“ ohne spezielle Berücksichtigung im Programmcode durchgeführt werden. Dabei werden, unterstützt durch das Konzept der „Persistenz durch Erreichbarkeit“, auch alle neu instanzierten Objekte in die Datenbank geschrieben. Auch die nahtlose Integration der Forwarder in die VisualWorks-Applikation verspricht das transparente Weiterleiten von Nachrichten an Objekte in GemStone, sofern vorher durch die zentralen Konfigurationsmöglichkeiten wie replicationSpec oder Konnektoren das Erzeugen von Forwardern anstatt von Replikaten festgelegt wird.
Neben den Vorteilen, die sich aus dem Einsatz der Mechanismen des GemBuilder zum Zugriff auf persistente Objekte ergeben, existieren aber auch einige Einschränkungen. Diese zeigen sich jedoch meistens nur in Ausnahmesituationen. Die automatische Generierung von Class Connectoren funktioniert nur bei Namengleichheit zwischen der Klasse in VisualWorks und GemStone. Wenn unterschiedlich benannte Klassen aufeinander abgebildet werden sollen, können die entsprechenden Konnektoren aber manuell angelegt werden. Die automatische Erzeugung von Klassen in GemStone sollte um die Möglichkeit des Übersetzens von Methoden erweitert werden. Bei der Nutzung von Forwardern zum Ausführen der Methoden persistenter Objekte, müssen auch alle direkt und indirekt davon betroffenen Methoden in GemStone implementiert sein. Ansonsten kann es zu Fehlern zur Laufzeit kommen. Des weiteren stellt auch die bedingte Übertragung von Smalltalk-Blöcken eine wichtige Einschränkung bei der Anbindung einer VisualWorks-Anwendung an GemStone dar. Hierbei müssen alle Methodenaufrufe an GemStone-Objekte sowie die Schnittstellen dieser Objekte so überarbeitet werden, daß eine zusätzliche Übergabe des Kontexts des Blockes möglich ist. Neben diesem Aspekt ist auch eine Überarbeitung der Methoden nötig, da die Schnittstellen der Objekte in GemStone teilweise nicht mit denen in VisualWorks übereinstimmen.
Zusätzlich zu den bisher beschriebenen Einschränkungen beim Einsatz des GemBuilder zeigen sich bei der Nutzung von Replikaten und Forwardern zwei konzeptionelle Besonderheiten. Die erste betrifft das automatische Markieren modifizierter Objekte im Zusammenhang mit der Klassenmethode instVarMap beim Ausblenden ausgezeichneter Attribute von der Persistenz. Nach dem Senden der Nachricht markDirtyOnInstVarAssign sollen alle Instanzen dieser Klasse bei einer Veränderung des Zustands als „dirty“ markiert werden. Diese werden aber auch dann markiert, wenn einer von der Persistenz ausgenommenen Instanzvariablen ein neues Objekt zugewiesen wird. Dies ist eigentlich nicht nötig, da der in VisualWorks geänderte Zustand des Replikats keinen Einfluß auf den Zustand des persistenten Objekts hat, weil genau die von der Persistenz ausgeschlossenen Instanzvariablen in GemStone nicht gespeichert werden. Dies resultiert nicht nur in überflüssigen markDirty-Nachrichten, sondern wirkt sich auch negativ auf die Transaktionsverarbeitung in GemStone aus[45]. Hierbei teilt der GemBuilder GemStone mit, daß ein Objekt verändert wurde und in der Datenbank aktualisiert werden muß, obwohl diese Veränderung keinen Einfluß auf den Zustand des Objektes in GemStone hat. Tritt dies bei mehreren parallel laufenden Transaktionen gleichzeitig auf, wird ein Schreib-Konflikt festgestellt und nur die zuerst beendende Transaktion kann ihre Änderungen in der Datenbank aktualisieren. Alle anderen müssen aufgrund des vermeintlichen Schreib-Konflikts abgebrochen werden. Zur Lösung dieses Problems gibt es folgende Möglichkeit. Man kann die transienten Attribute in einer eigen-

[45] Die Transaktionsverarbeitung in GemStone wird in Abschnitt 5.4 ausführlicher betrachtet.

ständigen Klasse auszulagern und entfernt für sie das automatische Markieren. Da dies hierbei aber auch für die Unterklassen gilt, muß man diesen anschließend wieder die Nachricht markDirtyOnInstVarAssign senden. Die Attribute der isolierten Klasse sind dadurch von dem automatischen Markieren ausgeschlossen.
Die zweite (nicht dokumentierte) Besonderheit des GemBuilders zeigt sich beim gleichzeitigen Einsatz von Replikaten und Forwardern. Falls ein Objekt aus GemStone bereits repliziert worden ist, kann es nicht mehr durch die Nutzung der Klassenmethode replicationSpec über einen Forwarder angesprochen werden. Dieser Fall tritt insbesondere dann auf, wenn ein Objekt von verschiedenen anderen referenziert wird, wobei es zu einem früheren Zeitpunkt bereits über ein anderes repliziert wurde. Soll es aber beim Zugriff von einem anderen Objekt über einen Forwarder angesprochen werden, löst der GemBuilder dies anhand der replicationSpec nicht ordnungsgemäß auf, da es ja schon repliziert wurde. Die Ursache für dieses Verhalten liegt darin, daß die Methode replicationSpec nur dann ausgewertet wird, wenn ein Stub die Replikation auslöst. Wenn ein Objekt bereits repliziert ist, kann Smalltalk es nicht mehr von anderen lokalen Objekten unterscheiden. Für den Einsatz von Forwardern muß man also sichern, daß die aktiven Datenbankobjekte nicht durch Referenzen über andere Objekte repliziert werden.
Aufgrund der beschriebenen Stärken und Einschränkungen kann das Resümee gezogen werden, daß der Zugriff auf persistente Objekte in GemStone fast vollständig transparent erfolgt und auch die Möglichkeit des gemischten Einsatzes von Replikaten und Forwardern eine ausgeglichene Lastverteilung zwischen Client und Server ermöglicht. Jedoch darf man nicht vernachlässigen, daß der Einsatz der Konzepte von GemStone während der Entwicklung einer Anwendung zusätzlich berücksichtigt werden müssen. Trotz der Automatismen zum Markieren modifizierter Objekte, zum Erzeugen von Class Connectoren und zum Generieren fehlender Klassen in GemStone unterliegen diese auch einigen Einschränkungen, die sich auf die Implementierung auswirken. Für einen problemlosen Einsatz von GemStone ist die sorgfältige Ermittlung aller potentiell persistenten Objekte und deren Klassen erforderlich. Beim Einsatz von Forwardern sind auch die Methoden der persistenten Objekte in GemStone bei der Definition der GemStone-Klassen zu integrieren. Davon sind alle Methoden betroffen, die direkt oder indirekt über einen Forwarder aufgerufen werden, wobei der Aufwand der Ermittlung aller potentiell aufgerufener Methoden sehr umfangreich sein kann. Ein Lösungsansatz wäre die Übersetzung aller Klassen persistenter Objekte inklusive der Methoden nach GemStone. Dies hat den Vorteil, daß das gesamte Verhalten aller Objekte auch in GemStone vorhanden ist. Im Zuge von Wartungsarbeiten können veränderte Klassen und Methoden durch einen erneuten Übersetzungsvorgang in GemStone aktualisiert werden. Die Ausführung von Methoden auf dem Server bietet darüber hinaus den Vorteil der Entlastung des Anwendungs-Client von der Auswertung umfangreicher Anfragen und eine damit verbundene Entlastung der Kommunikationsinfrastruktur der verteilten Client/Server-Umgebung.

5. Transaktionsverarbeitung

Im Gegensatz zu einem Anwendungsprogramm, welches exklusiv nur von einem Benutzer verwendet wird, stehen persistente Objekten in einer ODB mehreren Anwendern gleichzeitig zur Verfügung (vgl. [MeWü97], S. 65). Der parallele Zugriff dieser Benutzer erfolgt dabei innerhalb von Datenbanktransaktionen (kurz: Transaktionen), die im folgenden Abschnitt vorgestellt werden sollen. Anschließend wird anhand typischer Probleme bei diesem gleichzeitigen Zugriff die Problematik der Synchronisation von Transaktionen in Abschnitt 5.2 erläutert. In dem folgenden Abschnitt 5.3 werden erweiterte Transaktionsformen vorgestellt. Abschließend betrachtet Abschnitt 5.4 die Transaktionsverarbeitung in GemStone.

5.1. Transaktionen

Saake et al. definieren eine **Transaktion** als „...eine Folge von logisch zusammenhängenden Operationen auf einer Datenbank...“ (vgl. [STS97], S. 275). Dabei ist eine Transaktion eine atomare Ausführungseinheit, die den ACID-Eigenschaften genügen muß, wobei diese Abkürzung für folgende Eigenschaften steht (vgl. [STS97], S. 275 f.):

Atomicity (Atomarität): Eine Transaktion faßt logisch zusammenhängende Datenbank-Operationen in einer atomaren Ausführungseinheit zusammen. Es werden entweder alle Operationen ausgeführt oder keine.

Consistency (Konsistenz): Eine Transaktion überführt eine DB von einem konsistenten Zustand in einen konsistenten Zustand. Die durch das objektorientierte DB-Schema festgelegten Integritätsbedingungen werden durch eine Transaktion nicht verletzt.

Isolation: Jede Transaktion wird abgearbeitet, als ob ihr die DB exklusiv zur Verfügung stehen würde. Eine Transaktion wird nicht von den Operationen zeitgleich ablaufender Transaktionen beeinflußt. Erst am Ende einer Transaktion werden ihre Änderungen für andere sichtbar.

Durability (Dauerhaftigkeit): Sobald eine Transaktion beendet ist, sind alle ihre Änderungen dauerhaft in der DB gespeichert (persistent). Darüber hinaus müssen sie auch evtl. später auftretende Fehler überleben.

Jede gestartete Transaktion kann erfolgreich beendet (**commit**) oder vorzeitig abgebrochen (**abort**) werden. Im ersten Fall wird von dem ODBMS gesichert, daß ihre vollständigen Änderungen (wegen der Atomarität) gespeichert werden (Dauerhaftigkeit). Im zweiten Fall muß das ODBMS dafür Sorge tragen, daß die Transaktion abgebrochen wird und keine der Änderungen für andere Transaktionen sichtbar werden (Atomarität und Isolation).

5.2. Synchronisation konkurrierender Transaktionen

Um möglichst vielen Benutzern den parallelen Zugriff auf die DB zu ermöglichen, werden Transaktionen üblicherweise nicht konsekutiv abgearbeitet, sondern verzahnt. Obwohl hierbei die Transaktionen immer nur teilweise ausgeführt werden, sind aber trotzdem die ACID-Eigenschaften einzuhalten. Hierbei sind Zugriffskonflikte zwischen Transaktionen von besonderer Bedeutung. Wenn zwei Operationen auf ein Objekt O zugreifen und mindestens eine davon schreibt, spricht man von einem Konflikt dieser Operationen in O (vgl. [Vos94], S. 459). Damit solche Konflikte beim Zugriff auf eine DB nicht auftreten, ist eine Synchronisation parallel ablaufender Transaktionen erforderlich, damit die DB jederzeit in einem konsistenten Zustand ist. Saake et al. beschreiben hierzu ein Verfahren zur Synchronisation konkurrierender Zugriffe durch sog. Sperren[46]. Daneben identifizieren Meier und Wüst ein weiteres Verfahren, welches ohne Sperren auskommt[47]. Dieses soll zunächst im folgenden Abschnitt beschrieben werden, bevor im daran anschließenden Abschnitt die Synchronisation mit Sperren betrachtet wird. In Anlehnung an die angloamerikanische Terminologie werden Verfahren zur Synchronisation von Transaktionen im folgenden mit **Concurrency Control** bezeichnet.

5.2.1. Optimistisches Concurrency Control

Beim **optimistischen Concurrency Control** (OCC) wird während der Abarbeitung einer Transaktion davon ausgegangen, daß keine Konflikte auftreten. Die Ausführung einer Transaktion gliedert sich in drei Phasen (vgl. [MeWü97], S. 135):

In der ersten Phase wird die Transaktion ohne Überprüfung auf mögliche Konflikte vollständig abgearbeitet. Die persistenten Objekte werden innerhalb dieser Phase zunächst nicht direkt in der ODB verändert. Es werden vorerst nur Kopien von diesen Objekten in einem temporären Zwischenspeicher angelegt, wobei Modifikationen nur in diesem Zwischenspeicher stattfinden.

Nach Beendigung der Transaktion wird in der zweiten Phase geprüft, ob Konflikte mit anderen zeitlich gleich ablaufenden Transaktionen aufgetreten sind. Ist dies der Fall, werden die Änderungen im Zwischenspeicher verworfen und die Transaktion neu gestartet.

Falls jedoch keine Konflikte festgestellt werden konnten, werden in der dritten Phase die Änderungen an den Objekten im Zwischenspeicher in der ODB aktualisiert und die Transaktion erfolgreich beendet.

[46] Siehe hierzu [STS97], S. 279 ff. oder aber auch [MeWü97], S. 133 ff. .
[47] Siehe hierzu [MeWü97], S. 133 ff. .

Dieses Verfahren wird als „optimistisch“ bezeichnet, da davon ausgegangen wird, daß nur wenige Konflikte auftreten werden und somit keine Notwendigkeit für die Prüfung auf Konflikte während der Abarbeitung besteht (vgl. [ElNa94], S. 570). Aus diesem Grund eignet sich das OCC vor allem für solche Anwendungen, in denen hauptsächlich Objekte gelesen, aber nur wenige geschrieben werden. Hierbei spart man bei jedem lesenden oder schreibenden Zugriff auf die persistenten Objekte den Aufwand der Prüfung auf Konflikte. Darüber hinaus werden Transaktionen nicht verzögert, wie dies bspw. beim Einsatz von Sperren zur Synchronisation konkurrierender Transaktionen der Fall ist (siehe nächster Abschnitt). Andererseits kann aber ein hohes Aufkommen von Konflikten dazu führen, daß viele Transaktionen in der zweiten Phase abgebrochen und neu gestartet werden müssen. Somit eignet sich das OCC nur bedingt für Anwendungen, in denen viele Objekte geschrieben werden. Hierbei kann ein Verfahren, welches schon während der Ausführung einer Transaktion auf Konflikte prüft - bspw. das im folgenden Abschnitt beschriebene Verfahren mit Sperren - angemessener sein.

5.2.2. Synchronisation durch Sperren

Bei diesem Verfahren werden die innerhalb einer Transaktion gelesenen oder veränderten Objekte durch Sperren (engl. locks) vor dem Zugriff durch andere Transaktionen geschützt. Dabei kann man zwischen zwei verschiedenen Arten von Sperren unterscheiden (vgl. [STS97], S. 280 f. und [MeWü97], S. 134):

Schreibsperre (engl. write lock oder exclusive lock): Wenn eine Transaktion Tr ein Objekt O mit einer Schreibsperre versieht, bedeutet dies, daß keine andere Transaktion O lesen oder schreiben darf, solange diese Sperre gehalten wird. Dadurch soll verhindert werden, daß andere Transaktionen ein veraltetes Objekt lesen oder es modifizieren, während es von Tr verändert wird.

Lesesperre (engl. read lock oder shared lock): Wenn eine Transaktion Tr ein Objekt O mit einer Lesesperre versieht, bedeutet dies, daß keine andere Transaktion O schreiben darf, solange diese Sperre gehalten wird. Lesender Zugriff ist dabei für andere Transaktionen erlaubt. Dadurch soll verhindert werden, daß andere Transaktionen ein Objekt modifizieren, solange Tr mit dem alten Zustand arbeitet.

Eine Transaktion Tr, die auf ein Objekt O lesend oder schreibend zugreifen will, muß dieses vorher mit einer entsprechenden Sperroperation ankündigen. Sofern O noch nicht gesperrt ist, wird die entsprechende Sperre auf das Objekt gesetzt und Tr kann fortgesetzt werden. Wenn aber eine andere Transaktion schon eine Schreibsperre auf O hält, kann weder eine Lese- noch eine Schreibsperre gewährt werden und Tr muß solange warten, bis die andere Transaktion O wieder freigibt. Falls eine andere Transaktion eine Lesesperre auf O hält, kann für Tr auch nur eine Lesesperre gewährt werden. Das Setzen einer Schreibsperre ist für Tr in einem solchen Fall nicht möglich; Tr muß solange warten, bis die andere Transaktion O freigibt.
Dieses Verfahren wird im Gegensatz zum OCC in der Literatur oft als **pessimistisch** bezeichnet, da bei jedem Zugriff auf ein Objekt erst geprüft wird, ob schon eine andere Transaktion auf dieses Objekt zugreift. Dabei muß eine Transaktion bei einem gesperrten Objekt so lange warten, bis die andere Transaktion, die eine Sperre auf das Objekt hält, dieses wieder freigibt. Neben dem Nachteil des zusätzlichen Aufwandes für die Sperroperationen existiert bei diesem Ansatz noch ein weiteres Problem. Wenn mehrere Transaktionen auf die Freigabe von Sperren warten, kann es zu zyklischen Wartebeziehungen, sog. Deadlocks, kommen (siehe [Vos94], S. 480). Eine Transaktion Tr1 wartet auf die Freigabe einer Sperre, die von einer anderen Transaktion gehalten wird, welche aber wiederum auf die Freigabe einer Sperre wartet, die von Tr1 gehalten wird. Zur Lösung solcher Deadlocks muß das ODBMS zusätzliche Mechanismen zur deren Erkennung und Elimination implementieren.
Allein der Einsatz von Sperren reicht aber nicht aus um alle Konflikte zwischen Transaktionen zu vermeiden. Saake et al. sowie Meier und Wüst führen hierzu das sog. Zwei-Phasen-Sperr-Protokoll (engl. two phase locking protocol (2PL)) an, welches Transaktionen in zwei Phasen einteilt (siehe [STS97], S. 281 ff. und [MeWü97], S. 133 f.). In der ersten Phase darf eine Transaktion nur Sperren anfordern, aber keine freigeben. Die zweite Phase wird mit der ersten Freigabe einer Sperre eingeleitet. Innerhalb dieser Phasen dürfen keine neuen Sperren mehr angefordert, sondern nur vorhandene Sperren freigegeben werden. Durch Einsatz dieses Protokolls wird gewährleistet, daß alle Konflikte zwischen Transaktionen erkannt werden (vgl. [STS97], S. 281 und [MeWü97], S. 133). Falls darüber

hinaus auch noch das konservative 2PL (engl. conservative two phase locking protocol (C2PL)) eingehalten wird, können außerdem auch Deadlocks vermieden werden (vgl. [STS97], S. 282 f.). Beim C2PL müssen alle Sperren direkt zu Beginn einer Transaktion angefordert werden, wobei die Regel gilt, daß entweder alle oder keine Sperren gewährt werden. Hierdurch wird gesichert, daß eine Transaktion während der Abarbeitung auf keine andere Transaktion warten muß, wodurch auch keine zyklischen Wartebeziehungen entstehen können.

5.3. Erweiterte Transaktionsformen

Atkinson et al. sehen innerhalb des Manifests die Möglichkeit spezieller Transaktionsformen („long transactions“ oder „nested transactions“) als optionale Eigenschaft eines ODBMS an. Sie weisen aber darauf hin, daß herkömmliche Transaktionsformen für die Anforderungen moderner Applikationen nicht vollkommen ausreichen (vgl. [ABD+89], S. 13). Man denke hier etwa an Applikationen für CAD/CAM (Entwurfsanwendungen), Softwareentwicklung oder Dokumentenbearbeitung. Im Gegensatz zu traditionellen Applikationen können in o. g. Anwendungen Transaktionen auftreten, die mehrere Minuten, Stunden oder sogar Tage andauern. Trotz dieser langen Zeiten muß aber immer gewährleistet sein, daß Transaktionen einerseits die Konsistenz der Datenbank erhalten, aber andererseits den Mehrbenutzerzugriff nicht unnötig lange behindern.
In der Literatur findet man mehrere Ansätze für alternative Transaktionsformen (vgl. [Loo91], [KeMo94], S. 396 ff., [Heu97], S. 540 ff. und [MeWü97], S. 128 ff.). Die darin vorgestellten Transaktionsformen sind aber keineswegs neu, sondern gehen zurück auf Arbeiten im Kontext relationaler DBMS (vgl. [STS97], S. 324). Da das objektorientierte Datenbankmodell aber angetreten ist, die Anforderungen oben genannter Applikationen zu erfüllen, gewinnen auch die Ansätze für lange Transaktionen im Zusammenhang mit ODBMS an Bedeutung. Im Folgenden werden deshalb zwei Formen langer Transaktionen vorgestellt, die in der Literatur recht populär sind. Diese Ansätze haben gemeinsam, daß sie pessimistisches Concurrency Control auf Basis von Sperren voraussetzen.

5.3.1. Lange Transaktionen mit Check-Out und Check-In

Da lange Transaktionen in ODBS unter Umständen mehrere Tage andauern können, ist eine Verwaltung der Sperren auf Datenelemente, wie sie in konventionellen DBMS durchgeführt wird, nicht mehr ausreichend. Es muß dafür gesorgt werden, daß Sperren auch nach dem Beenden einer Applikation oder nach einem Fehlerfall weiterhin erhalten bleiben. Diesem Anspruch tragen lange Transaktionen mit Check-Out und Check-In Rechnung. Sperren werden nicht mehr im Hauptspeicher verwaltet, sondern persistent in der Datenbank (siehe hierzu auch [Loo91] und [MeWü97], S. 128 f.). Um Objekte aus der zentralen Datenbank bearbeiten zu können, werden sie zuerst durch einen speziellen Check-Out-Mechanismus dauerhaft gegen lesenden oder schreibenden Zugriff gesperrt. Gleichzeitig kopiert das Check-Out die Objekte aus der zentralen Datenbank in einen privaten Arbeitsbereich (engl. private workspace) oder eine lokale Datenbank[48]. Hier stehen sie dann für Modifikationen zur Verfügung, bis der endgültige Bearbeitungszustand erreicht ist. Erst jetzt werden die modifizierten Objekte durch ein Check-In in der zentralen Datenbank aktualisiert und somit für andere Benutzer sichtbar gemacht. Das Check-In hat dabei die Aufgabe, die Änderungen in die Datenbank zu schreiben, Sperren aufzuheben und die Objekte aus dem lokalen Arbeitsbereich zu entfernen. Sowohl das Check-Out als auch das Check-In laufen in jeweils eigenen (kleinen) Transaktionen ab, welche die ACID-Eigenschaften erfüllen.
Das Zurücksetzen einer Transaktion, die wegen eines Fehlers abgebrochen wurde, gestaltet sich ähnlich einfach wie bei OCC. Da noch keine Änderungen in der zentralen DB vorgenommen wurden, muß dort auch nichts zurückgesetzt werden. Es müssen nur die lokalen Änderungen durch Leeren des privaten Arbeitsbereichs verworfen und die Sperren in der Datenbank zurückgenommen werden.
Der Vorteil der langen Transaktionen mit Check-Out und Check-In ist, daß Sperren persistent gespeichert werden und somit auch nach einem Ausfall noch zur Verfügung stehen. Des weiteren lassen sie sich einfach durch vorhandene Transaktionsmechanismen realisieren. Das Check-Out und das Check-

[48] Diese lokale Datenbank steht nur dieser einen langen Transaktion zur Verfügung und kann somit als Einbenutzerdatenbank konzipiert werden, in der keine Kontrolle konkurrierender Zugriffe benötigt wird.

In werden jeweils in einer kurzen Transaktion zusammengefaßt. Die Entscheidung, wann Check-Out und Check-In durchgeführt werden, obliegt aber dem Benutzer oder dem Entwickler der laufenden Anwendung[49]. Das Problem lang gesperrter Objekte wird durch das bisher beschriebene Verfahren aber noch nicht gelöst. Es sei hier nur erwähnt, daß von Kemper und Moerkotte zu dessen Lösung eine weitere Art von Sperren (sog. „browsing locks“) vorgeschlagen werden, die zumindest das eingeschränkte Lesen der aktuellen Zustände gesperrter Objekte ermöglicht (vgl. [KeMo94], S. 401 f.).

5.3.2. Geschachtelte Transaktionen

Die Grundidee bei geschachtelten Transaktionen ist, daß jede Transaktion nicht nur aus atomaren Datenbankoperationen, sondern aus mehreren Subtransaktionen bestehen kann (vgl. [VoGr93], S. 251 ff. und [MeWü97], S. 131 ff.). Jede (Sub-) Transaktion in dem so gebildeten Baum geschachtelter Transaktionen ist erst nach dem erfolgreichen Abschluß aller in ihr enthaltenen Subtransaktionen beendet. Andererseits wirkt sich der Abbruch einer Transaktion nur insofern aus, daß zwar ihre Subtransaktionen zurückgesetzt werden, nicht aber die ihr übergeordnete Transaktion. Tritt innerhalb einer Subtransaktion ein Fehler auf, werden nur diese eine Transaktion und die evtl. in ihr geschachtelte Subtransaktionen zurückgesetzt. Die ihr übergeordnete Transaktion ist davon nicht betroffen, die Ergebnisse bereits erfolgreich beendeter Subtransaktionen bleiben erhalten. Um den Vorgang fortzusetzen, kann die abgebrochene Transaktion wieder neu gestartet werden.
Geschachtelte Transaktionen verbessern die Transaktionsverarbeitung, indem sie die Einheit, die im Fehlerfall zurückgesetzt und wiederholt werden muß, von langen Transaktionen auf kleinere Subtransaktionen reduzieren. Ein weiterer Vorteil bei geschachtelten Transaktionen ist die Möglichkeit, den Grad an Parallelität durch gleichzeitiges Abarbeiten mehrerer unabhängiger Subtransaktionen innerhalb einer Wurzeltransaktion zu erhöhen (vgl. [MeWü97], S. 133). Unabhängige Transaktionen sind in diesem Zusammenhang solche, zwischen denen keine Konflikte beim Zugriff auf Objekte auftreten.
Ein Problem besteht aber immer noch in der Dauer, in der Objekte gesperrt bleiben, weil bei diesem Konzept Transaktionen die Sperren ihrer Subtransaktionen übernehmen und diese somit erst nach Beenden der Wurzeltransaktion freigegeben werden. Andererseits ist dadurch aber gewährleistet, daß die Transaktion immer noch eine rücksetzbare Einheit bildet, weil modifizierte Objekte erst nach dem commit der Wurzeltransaktion sichtbar sind. Vossen und Groß-Hardt schlagen alternativ dazu vor, Sperren direkt beim Beenden einer Subtransaktion freizugeben, weil dadurch die Anzahl paralleler Zugriffe erhöht wird. Die Verfasser sprechen hierbei von „offen geschachtelten Transaktionen“ (vgl. [VoGr93], S. 254 f.). Andererseits muß man dabei aber beachten, daß beim Abbruch der Wurzeltransaktion die schon abgeschlossenen Subtransaktionen nicht mehr mit den Verfahren der Datenbank zurückgesetzt werden können.

5.4. Transaktionsverarbeitung in GemStone

In Abschnitt 5.4.1 werden die Verfahren und Konzepte zur Transaktionsverarbeitung in GemStone vorgestellt. Eine beurteilende Zusammenfassung erfolgt in Abschnitt 5.4.2.

5.4.1. Überblick über die Transaktionsverarbeitung in GemStone

Dieser Abschnitt behandelt hauptsächlich die Realisierung der Mechanismen zur Transaktionsverarbeitung in GemStone. Dabei werden neben der Einordnung der Mechanismen in die bereits vorgestellten Verfahren objektorientierter Datenbanken auch die Spezifika der Transaktionsverarbeitung in GemStone betrachtet.

5.4.1.1. Transaktionen in GemStone

Zur Synchronisation konkurrierender Zugriffe mehrerer Benutzer kann das Lesen und Schreiben persistenter Objekte in GemStone nur innerhalb einer Transaktion erfolgen. Hierbei bietet das ODBMS einen manuellen und einen automatischen Transaktionsmodus (vgl. [Gem96b], S. 6-7). Bei dem ma-

[49] Im Falle eines CAD-Systems wäre dies z. B., wenn der Ingenieur einen Teil eines Projektes bearbeiten und die Konstruktionsänderungen nach einer längeren Zeit als finalen Zustand in der Datenbank aktualisieren will.

nuellen Modus muß jede Transaktion explizit gestartet und durch ein commit oder abort beendet werden. Der Benutzer resp. der Entwickler muß dafür Sorge tragen, daß bei jedem Zugriff auf GemStone eine Transaktion gestartet worden ist. Bei der automatischen Variante befindet sich der Benutzer direkt nach dem Einloggen in die Datenbank in einer Transaktion und nach jedem Beenden einer Transaktion durch commit oder abort wird automatisch eine neue Transaktion gestartet. Hierdurch wird gewährleistet, daß sich ein Benutzer immer in einer Transaktion befindet und jederzeit auf persistente Objekte zugreifen kann. Zur dauerhaften Speicherung von Änderungen an persistenten Objekten muß durch das Senden der Nachricht commitTransaction an die aktuelle Sitzung das commit in GemStone eingeleitet werden. Hierdurch werden, sofern keine Konflikte aufgetreten, diese Änderungen in GemStone übernommen. Analog dazu wird durch die Nachricht abortTransaction die aktuelle Transaktion der Sitzung zurückgesetzt, wobei alle Änderungen an persistenten Objekten rückgängig gemacht werden. Hiervon sind aber nur die persistenten Objekte betroffen, nicht aber die transienten Objekte, die außerhalb von GemStone liegen. Die maximale Länge einer Transaktion wird in GemStone nur von der Laufzeit der Smalltalk-Anwendung auf dem Client begrenzt, d.h. eine Transaktion kann innerhalb dieser Grenze beliebig lange andauern. Die Synchronisation paralleler Transaktionen erfolgt in GemStone optimistisch oder pessimistisch (siehe Abschnitte 5.4.1.2 und 5.4.1.3). Spezielle Transaktionsformen werden in GemStone nicht bereitgestellt. Hierzu zählen die beispielsweise die in Abschnitt 5.3 vorgestellten geschachtelten Transaktionen oder das Konzept langer Transaktionen mit Check-Out und Check-In. Persistente Sperren, die die Ausführung einer Anwendung überdauern sind nicht möglich. Sperren werden manuell, beim Beenden einer Transaktion oder beim Ausloggen des Benutzers aus der Datenbank entfernt.

5.4.1.2. Optimistisches Concurrency Control

Zu Beginn der Entwicklung von GemStone wurde zunächst optimistisches Concurrency Control implementiert. Dieses wurde später um die Möglichkeit von Sperren, also pessimistischem Concurrency Control, erweitert, wobei hier die gleichzeitige Nutzung beider Verfahren innerhalb einer Transaktion möglich ist. Diese Entscheidung begründet sich auf der Einschätzung der Entwickler von GemStone, daß die Realisierung von pessimistischem CC aufbauend auf einem optimistischen einfacher zu gestalten ist (vgl. [Bre89], S. 291). Die Entwickler erachten optimistisches Concurrency Control – ohne Angabe von Gründen – als angemessener für objektorientierte Systeme (vgl. [Bre89], S. 291). Diese Einschätzung läßt sich nur durch die aufgrund der Möglichkeit des nicht seiteneffektfreien Aufrufs von Methoden bedingte Komplexität des Zugriffs auf persistente Objekte erklären. Beim Zugriff auf die Objekte einer objektorientierten Datenbank werden nicht nur Objekte gelesen oder geschrieben, sondern über die Methoden dieser Objekte können noch weitere Objekte betroffen sein. Der Zugriff reduziert sich nicht nur auf unmittelbar angesprochene Objekte, sondern auch die zusätzlich durch die Methoden gelesenen und geschriebenen Objekte müssen betrachtet werden.
Zur Realisierung des parallelen Zugriffs mehrerer konkurrierender Benutzer wird in GemStone für jede Datenbanksitzung ein eigener Arbeitsbereich angelegt (Bretl et al. sprechen in [Bre89] hierbei von einem „workspace“). Sobald ein Objekt in GemStone modifiziert oder neu instanziert wird, wird von diesem zunächst nur eine Kopie (shadow copy) in diesem workspace angelegt. Alle hier abgelegten Objekte bleiben für die Transaktionen anderer Sitzungen unsichtbar, bis die erzeugende Transaktion erfolgreich beendet wird. Falls eine Transaktion abgebrochen wird (abort) oder aufgrund eines Konflikts nicht erfolgreich abgeschlossen werden kann, werden einfach nur die Änderungen im workspace verworfen. Dadurch, daß keine Änderungen in der Datenbank vorgenommen wurde, bleibt deren konsistenter Zustand erhalten und es müssen auch keine Operationen zum Rücksetzen von Änderungen durchgeführt werden. Falls eine Transaktion beendet werden soll, d.h. bei einem commit, überprüft GemStone zunächst, ob Konflikte bezüglich aller innerhalb der Transaktion gelesenen oder geschriebenen Objekte aufgetreten sind. Können dabei keine Konflikte festgestellt werden, werden eventuelle Änderungen in der Datenbank aktualisiert und die Transaktion erfolgreich beendet. Falls ein Konflikt festgestellt wurde, können die Änderungen nicht in GemStone übernommen werden, wobei zwei unterschiedliche Konfliktarten in Frage kommen (vgl. [Bre89], S. 291):

Lese-Schreib-Konflikt: Eine Transaktion T, die ein Objekt O gelesen und ein anderes Objekt beschrieben hat, konfligiert in O, wenn eine andere Transaktion O geschrieben hat und erfolgreich beendet wurde (commit), nachdem T gestartet wurde.

Schreib-Schreib-Konflikt: Eine Transaktion T, die ein Objekt O schreiben will, konfligiert in O, wenn bereits eine andere Transaktion O geschrieben hat und erfolgreich beendet wurde (commit), nachdem T gestartet wurde.

Ebenfalls gilt, daß Transaktionen, die nur Objekte gelesen und keine geschrieben haben, in keinem Objekt mit anderen Transaktionen in Konflikt stehen können. Der Vorteil dieses CC-Verfahrens ist seine transparente Integration in das Anwendungsprogramm, da im Gegensatz zum pessimistischen CC keine expliziten Nachrichten zur Anforderung von Sperren auf Objekte nötig sind. Die Verwaltung der gelesenen und geschriebenen Objekte wird vollständig durch das ODBMS übernommen und beim commit einer Transaktion wird auch die Prüfung, ob Konflikte aufgetreten sind, von GemStone durchgeführt. Das OCC behindert keine Transaktionen, die nur lesend auf die Datenbank zugreifen. Nachteilig am OCC ist der späte Zeitpunkt am Ende einer Transaktion, zu dem auf Konflikte mit anderen Transaktionen geprüft wird. Hierbei können u. U. die Änderungen eine ganzen Transaktion aufgrund eines Konfliktes verloren gehen, dessen Ursache schon zu einem frühen Zeitpunkt innerhalb der Transaktion liegt. In einem solchen Fall müssen die Änderungen dieser Transaktion solange in einer jeweils neuen wiederholt werden, bis das commit erfolgreich ist.
Bei der Ermittlung von Konflikten hält sich GemStone streng an die oben formulierten Regeln. Insbesondere bei Lese-Schreib-Konflikten ist irrelevant, ob der Zustand eines gelesenen konfligierenden Objektes Auswirkungen auf die Zustände der in der Transaktion geschriebenen Objekte hat. Dieser Sachverhalt soll anhand eines einfachen Beispiels verdeutlicht werden. Ein Benutzer einer Literaturverwaltung läßt sich die Menge aller im System vorhandenen Verfasser anzeigen, wodurch innerhalb der aktuellen Transaktion alle Verfasserobjekte gelesen werden. Aus dieser Menge wählt er einen Verfasser V1 aus, für den er einen neuen Titel T1 anlegen möchte. Der Benutzer erfaßt dann alle Angaben für T1 und weist V1 diesen Titel T1 zu, modifiziert also den Zustand von V1. Parallel dazu kann ein anderer Benutzer die gleiche Aktion für einen anderen Verfasser V2 und einen anderen Titel T2 durchgeführt haben. In einem solchen Fall kann nur die Transaktionen erfolgreich beendet werden, die als erstes ein commit durchführt. Beim Beenden der zweiten Transaktion wird ein Lese-Schreib-Konflikt in einem Verfasserobjekt festgestellt wird. Dieser tritt auf, da beide Benutzer alle Verfasserobjekte (einschließlich V1 und V2) gelesen und jeweils unterschiedliche Verfasser modifiziert haben. Eigentlich könnten aber beide Transaktionen erfolgreich beendet werden, da die Änderung an V1 unabhängig von der an V2 ist. Allgemein kann dieses Verhalten dazu führen, daß einige Transaktionen, die viele Objekte gelesen und nur wenige geschrieben haben, nicht erfolgreich beendet werden können. Das erfolglose commit kann insbesondere dann überflüssig sein, wenn die Zustände der konfligierenden Objekte keinen direkten oder indirekten Einfluß auf die Zustände der geschriebenen Objekte haben. Um für solche Fälle die Anzahl erfolgreicher Transaktionen zu erhöhen, bietet GemStone die Möglichkeit die Prüfung auf Lese-Schreib-Konflikte zu deaktivieren. Diese Einstellung gilt aber dann für alle Transaktionen einer GemStone-Datenbank und beinhaltet ein Problem. Da nicht mehr auf Lese-Schreib-Konflikte geprüft wird, kann das sog. Phantom-Problem auftreten (siehe hierzu [Vos94], S. 449). Dies bedeutet, daß eine Transaktion Tr1 ein Objekt liest, das zu ihrer Laufzeit durch eine andere Transaktion Tr2 verändert worden ist. Berechnet Tr1 anhand des gelesenen Objekts ein neues Objekt, kann der Wert des neuen Objekts ungültig sein, da das gelesene Objekt sich geändert hat.

5.4.1.3. Pessimistisches Concurrency Control

„GemStone's pessimistic concurrency control merges seamlessly with its optimistic concurrency control."
(vgl. [Bre89], S. 291)

Neben dem OCC bietet GemStone auch die zusätzliche Möglichkeit zum Sperren von Objekten. Transaktionen können auf einige Objekte optimistisch zugreifen, wohingegen innerhalb der gleichen

Transaktion die Synchronisation paralleler Zugriffe auf andere Objekte durch Sperren erfolgen kann (vgl. [Bre89], S291). Dabei werden drei Arten von Sperren unterstützt (vgl. [Gem96b], S. 6-14 ff.):

Lesesperre (Read Lock): Wenn eine Transaktion Tr1 ein Objekt O1 durch einen read lock sperrt, bedeutet dies, daß keine andere Transaktion Tr2 O1 schreiben oder durch eine andere Sperre außer einem read lock sperren darf. Hierdurch wird gewährleistet, daß das von Tr1 gelesene O1 nicht durch Tr2 überschrieben wird, bevor Tr1 beendet wurde.

Schreibsperre (Write Lock): Wenn eine Transaktion Tr1 ein Objekt O1 durch einen write lock sperrt, bedeutet dies, daß keine andere Transaktion Tr2 O1 schreiben oder durch eine beliebige Sperre sperren darf. Lesende Zugriffe auf O1 sind aufgrund des OCC trotzdem noch erlaubt.

Exklusive Sperre (Exclusive Lock): Wenn eine Transaktion Tr1 ein Objekt O1 durch einen exclusive lock sperrt, bedeutet dies, daß keine andere Transaktion Tr2 O1 lesen, schreiben oder durch eine beliebige Sperre sperren darf. Lesende Zugriffe auf O1 durch das OCC sind nicht mehr erlaubt.

Die Integration von optimistischem und pessimistischem CC drückt sich somit nicht nur durch den parallelen Einsatz von Sperren innerhalb des OCC aus, sondern auch in der Interpretation der in GemStone realisierten Sperren. Auch wenn Objekte durch Lese- oder Schreibsperren gesperrt sind, können sie noch immer durch das optimistische CC gelesen werden. Nur durch Setzen einer exklusiven Sperre kann auch dieser lesende Zugriff verboten werden. Aufgrund der Konzeption des OCC drückt sich dies nicht in Fehlermeldungen beim Zugriff auf gesperrte Objekte aus, sondern erst zum Zeitpunkt des commit einer Transaktion durch eine Meldung, daß das commit fehlgeschlagen ist. Die Anforderung und das Entfernen von Sperren geschieht in GemStone über die Klasse System, die alle hierzu benötigten Operationen in Form von Klassenmethoden implementiert[50].
Beim Einsatz von Sperren zur Synchronisation konkurrierender Zugriff auf die Datenbank können Deadlocks[51] entstehen. Falls nicht schon zu Beginn einer Transaktion alle Sperren angefordert werden (konservatives 2PL zur Deadlock-Vermeidung[52]), muß zur Laufzeit regelmäßig auf das Vorhandensein von Deadlocks geprüft werden (Deadlock-Erkennung[53]) und Maßnahmen eingeleitet werden, um die zyklische Wartebeziehung aufzulösen (Deadlock-Elimination[54]). GemStone löst dieses Problem, indem gar nicht erst auf die Gewährung einer Sperre gewartet wird, sondern sofort nach der Anforderung einer Sperre meldet, ob sie gesetzt werden konnte oder nicht. Da hierbei keine systembedingten zyklischen Wartebeziehungen entstehen können, sind in GemStone auch keinerlei Mechanismen zur Erkennung und Elimination von Deadlocks implementiert (vgl. [Gem96b], S. 6-18). Es obliegt vielmehr den Entwicklern einer Anwendung die Verwendung von Sperren zu organisieren.

5.4.1.4. „Reduced Conflict“-Klassen

Eine weitere Möglichkeit zur Reduzierung von Konflikten beim Zugriff auf persistente Objekte in GemStone bieten die sog. **„Reduced Conflict“-Klassen** (im folgenden kurz RC-Klassen genannt). Diese werden von Almarode in [Alm95c] einführend vorgestellt und in der GemStone-Dokumentation (vgl. [Gem96b], S. 6-26 ff.) ausführlicher beschrieben. Die Grundidee der RC-Klassen soll hier anhand des Beispiels einer Multimenge (engl. bag) beschrieben werden. Eine Multimenge hat eine ähnliche Semantik wie eine Menge, unterscheidet sich aber von dieser, daß die Elemente auch mehrfach vorkommen können. Die einzelnen Elemente haben dabei, genau wie die einer Menge, keine Ordnung innerhalb der Multimenge. In Smalltalk werden Multimengen durch die Kollektionsklasse Bag realisiert, die auch u. a. die Methoden zum Hinzufügen oder Entfernen von Objekten implementiert. Falls nun zwei unterschiedliche Benutzer jeweils ein Objekt zu ein und derselben Instanz der Klasse Bag in GemStone hinzufügen, resultiert dies in einem Schreib-Schreib-Konflikt, da beide Benutzer dasselbe Bag-Objekt modifizieren. Almarode spricht in diesem Fall von einem Konflikt auf physikalischer Ebene, der durch die konkrete Realisierung einer Multimenge als Kollektion bedingt wird. Anderer-

[50] Vgl. [Gem96b], S. 6-16 ff. für weitere Informationen über die Schnittstelle der Klasse System.
[51] Siehe auch Abschnitt 3.5.2 .
[52] vgl. [ElNa94], S. 562 oder [Vos94], S. 481
[53] vgl. [Vos94], S. 480 oder [ElNa94], S.563-564
[54] vgl. [Vos94], S. 480-481 oder [ElNa94], S. 564

seits spricht gemäß Almarode aber nichts dagegen, daß die beiden Objekte nicht gleichzeitig hinzugefügt werden können (vgl. [Alm95c], S. 16). Dies begründet sich darauf, daß die Reihenfolge, in der die Objekte in die Multimenge eingefügt werden, nicht entscheidend ist. Almarode spricht in diesem Fall davon, daß auf logischer Ebene kein Konflikt vorliegt, d.h. die Semantik einer Multimenge wird nicht verletzt. Auf diesem Ansatz setzt auch die Realisierung der Klasse RcBag[55] an. Eine Instanz der Klasse RcBag ist in der Lage, konkurrierende Operationen zu verarbeiten, bei denen zwar ein Konflikt auf physikalischer Ebene, nicht aber auf logischer Ebene vorliegt. Hierdurch können mehr Transaktionen erfolgreich beendet werden, als dies bei der Benutzung einer Instanz der Klasse Bag möglich wäre. Es entstehen keine Konflikte, wenn mehre Benutzer Objekte zu einer Multimenge hinzufügen oder unterschiedliche Objekte entfernen. Es entstehen auch keine Konflikte, wenn ein Benutzer Objekte aus der Multimenge entfernt, während andere Benutzer Objekte hinzufügen. Konflikte treten nur dann auf, wenn mehrere Benutzer gleichzeitig mehr Vorkommen eines Objektes aus der Multimenge entfernen als sich darin befinden. Dem Vorteil der RC-Klassen steht der Nachteil gegenüber, daß ihre Instanzen mehr Speicherplatz beanspruchen als die von korrespondierenden Nicht-RC-Klassen. Für weitere Informationen über die in GemStone implementierten RC-Klassen sei an dieser Stelle auf die oben angeführte Literatur verwiesen.

5.4.2. Beurteilung der Transaktionsverarbeitung in GemStone

Der Fokus der Beurteilung der Konzepte und Mechanismen zur Transaktionsverarbeitung liegt auch hier wieder auf der Transparenz ihrer Integration in eine Smalltalk-Anwendung. Dies wird durch den automatischen Transaktionsmodus unterstützt, da innerhalb der Anwendung nur noch die Nachrichten zum Beenden von Transaktionen berücksichtigt werden müssen und nach dem Einloggen jederzeit auf persistente Objekte in GemStone zugegriffen werden kann. Des weiteren bedingt auch der Einsatz des optimistischen Concurrency Control eine Entlastung der Client-Anwendung von zusätzlichen Aufgaben der Transaktionsverarbeitung. Das Protokollieren gelesener und modifizierter Objekte wird von GemStone übernommen. Dabei bleibt man aber nicht auf das OCC beschränkt. Falls in einer Anwendung an einigen Stellen das OCC nicht angemessen ist, da bspw. unter bestimmten Umständen viele Objekte verändert werden, existiert auch noch die zusätzliche Möglichkeit, kritische Objekte zu sperren. Durch den Einsatz der RC-Klassen kann die Anzahl erfolgreicher Transaktionen im Rahmen des OCC erhöht werden.
Andererseits unterliegt die ausschließliche Anwendung des automatischen Transaktionsmodus und des OCC einigen Einschränkungen. Man sollte bei dem automatischen Transaktionsmodus bedenken, daß sich ein eingeloggter Benutzer jederzeit in einer Transaktion befindet. Eine solche Transaktion Tr1 kann dabei u.U. sehr lang werden, wenn ein Benutzer nach einer längeren Zeit der Inaktivität Modifikationen an den Objekten in GemStone durchführt und Tr1 durch ein commit beenden will. Bei der Überprüfung auf Konflikte werden hierbei die Objekte aller anderen Transaktionen, die seit dem Beginn von Tr1 erfolgreich beendet wurden, herangezogen. Die Anzahl dieser Objekte kann durch die lange Laufzeit von Tr1 bedingt sehr groß sein, so daß die Wahrscheinlichkeit für Konflikte steigt und auch die Wahrscheinlichkeit für ein fehlgeschlagenes commit von Tr1. Aus diesem Grund sollte man nicht nur Transaktionen allgemein so kurz wie möglich halten, sondern darüber hinaus auch Inaktivitäten der Benutzer bei der Arbeit mit der Anwendung berücksichtigen. Dies bedeutet, daß solche Transaktionen zum Ende einer Inaktivitätsphase. zurückgesetzt werden sollten. Des weiteren sollte beim Einsatz des OCC die Möglichkeit eines fehlgeschlagenen commit eingeplant werden und eventuelle Modifikationen in transienten Objekten (wie bspw. der Benutzungsoberfläche) zwischengespeichert werden. Falls das commit fehlschlägt, können die Änderungen automatisch innerhalb einer neuen (kurzen) Transaktion wiederholt werden, so daß kein manuelles Eingreifen des Benutzers erforderlich ist.
Im Gegensatz zum OCC ist dieser Fall aber bei der Nutzung von Sperren auf Objekten zur Synchronisation konkurrierender Zugriffe weniger von Bedeutung, da hier Konflikte schon zur Laufzeit einer Transaktion festgestellt werden. Dieser pessimistische Ansatz bedingt aber das Einfügen von Sperr-

[55] In der aktuellen Implementierung von GemStone heißt diese Klasse RcIdentityBag. Bei dieser Multimenge wird die Gleichheit von Objekte nicht aufgrund ihres Zustands bestimmt, sondern durch die Identität, d.h. der OID.

operationen in den Programmcode und die Berücksichtigung nicht gewährter Sperren. In GemStone wird bei der Anforderung einer Sperre nur mitgeteilt, ob diese gesetzt werden konnte oder nicht. Wartezeiten auf das Setzen von Sperren existieren in GemStone somit nicht, wodurch auch keinerlei Mechanismen zur Erkennung und Elimination von Deadlocks implementiert sind. Es liegt hier vielmehr in der Verantwortung der Entwickler, solche Fälle bei der Anwendungsentwicklung zu berücksichtigen, was hierbei einen zusätzlichen Aufwand bedeutet. Darüber hinaus erhält der Entwickler auch keine Unterstützung in Form spezieller Transaktionsformen wie bspw. geschachtelte Transaktionen oder lange Transaktionen mit Check-Out und Check-In und persistenten Sperren. Falls die Anwendung solche Transaktionsformen erfordert, müssen diese explizit vom Entwickler in der Anwendung simuliert werden.

6. Autorisierung in objektorientierten Datenbanken

In aktuellen Lehrbüchern über ODB findet man nur wenige konkrete Ausführungen zum Thema Autorisierung. Auch Atkinson et al. fordern im Manifest[56] Autorisierung weder als obligatorische noch als optionale Eigenschaft eines ODBMS. Bertino und Martino führen hierzu an, daß in der Vergangenheit zwar einige Autorisierungsmodelle für RDBMS entwickelt wurden, diese aber nicht vollständig auf das Objektmodell anwendbar sind. Darüber hinaus ist die Entwicklung eines Autorisierungsmodells, welches einerseits die objektorientierten Konzepte berücksichtigt und andererseits die Performanz des Systems nicht zu stark beeinträchtigt, eine komplexe Aufgabe (vgl. [BeMa93], S. 147). Die meisten ODBMS beschränken Autorisierung derzeit auf die Sicherheitsmechanismen des zugrundeliegenden Dateisystems (vgl. [BlPr98], S.337 oder auch [STS97], S. 419). Gleichwohl spielt Sicherheit in verteilten Mehbenutzersystemen eine große Rolle. Meier und Wüst merken hierzu an: „Der Schutz der Objekte vor unbefugtem Zugriff oder Mißbrauch muß durch geeignete Autorisierungsmaßnahmen gewährleistet sein.“ (vgl. [MeWü97], S. 117). Bertino und Martino sehen Mechanismen zur Autorisierung sogar als eine Basiskomponente eines DBMS an (vgl. [BeMa93], S. 147).

6.1. Autorisierung

Unter dem Begriff **Autorisierung** werden Verfahren zum Schutz der Daten vor unberechtigtem Zugriff zusammengefaßt. Dabei geht es darum, die in einer DB verwalteten Objekte nur den Benutzern zur Verfügung zu stellen, die das Recht zum Lesen oder Schreiben dieser Objekte besitzen. Auf der anderen Seite sollen die Objekte vor dem Zugriff nicht autorisierter Benutzer geschützt werden, um den Mißbrauch der Informationen zu verhindern. Ein verwandter Begriff ist die **Authentisierung**. Er steht für Verfahren, die dazu dienen, einen Benutzer beim Anmelden sicher als den zu identifizieren, für den er sich ausgibt. Es soll hier aber nicht weiter auf Authentisierung eingegangen werden, da sie unabhängig von der konkreten Anwendung ist – insbes. auch vom verwendeten Datenbankmodell – und gemäß Saake et al. kein datenbankspezifisches Problem ist (vgl. [STS97], S. 409).
Im weiteren Verlauf dieses Kapitels werden die Ausführungen von Kemper und Moerkotte kurz zusammengefaßt (siehe [KeMo94], S. 411 ff.). Diese basieren in erster Linie auf einem Beitrag von Rabitti et al., in welchem die Verfasser die Grundlagen des Autorisierungsmodells des ODBMS ORION beschreiben (vgl. [RBKW91]). Dieser Beitrag wird vielfach in der aktuellen Literatur zitiert, wie bspw. in [BeMa93] (S. 147 ff.), [MeWü97] (S. 119 ff.) und auch in [STS97] (S. 409 ff.). Obwohl es sich dabei um eine konkrete Realisierung eines Autorisierungsmodells handelt, können die Ideen allgemeiner angewendet werden (vgl. [BeMa93], S. 148).

6.2. Explizite Autorisierung

Nach Kemper und Moerkotte hat die Zugriffskontrolle auf eine Datenbank drei Dimensionen[57] (vgl. [KeMo94], S. 411):

1. Das **Subjekt** (ein Benutzer oder eine Gruppe von Benutzern), welches auf Informationen zugreift.

[56] Siehe [ABD+89].
[57] Ähnliche Definitionen findet man auch in [BeMa93], [Bud96], [MeWü97] und [STS97].

2. Das **Objekt**, auf welches zugegriffen wird.

3. Der **Modus** (eine Datenbankoperation), in dem das Subjekt auf das Objekt zugreift.

Somit läßt sich ein Zugriffsrecht als ein Tripel (s, o, op) mit dem Subjekt s, dem Objekt o und der zugreifenden Operation op darstellen. Das Zugriffsrecht (MoreUser, MoreList, read) würde z.B. ausdrücken, daß die Benutzer aus der Gruppe MoreUser lesend auf das Objekt mit dem Namen MoreList zugreifen dürfen. Diese Form des Zugriffsrechtes nennt man **positives** Zugriffsrecht, da es die Erlaubnis zum Zugriff erteilt. Analog dazu existieren auch **negative** Zugriffsrechte die folgendermaßen dargestellt werden: (s, o, -op). Dieser Ausdruck bedeutet, daß das Subjekt s nicht das Recht hat, gemäß der Operation op auf das Objekt o zuzugreifen und kommt somit einem Verbot gleich. Werden die Zugriffsrechte in der bisherigen Form formuliert, spricht man von **expliziter** Autorisierung.

6.3. Implizite Autorisierung

Eine vollständige Definition und Verwaltung aller expliziten Zugriffsrechte für alle möglichen Kombinationen aus Subjekt, Objekt und Operation erscheint aber zu aufwendig. Aus diesem Grund besteht zusätzlich die Möglichkeit der impliziten Ableitung von Autorisierungen aus expliziten Zugriffsrechten. In diesem Fall spricht man auch von **impliziter** Autorisierung. Die Regeln zur Beschreibung impliziter Zugriffsrechte können in graphischer Form in einer **Autorisierungs-Hierarchie** dargestellt werden. Dies verkleinert die Autorisierungsbasis expliziter Zugriffsrechte und gewährleistet somit einen schnellen Zugriff (vgl. [KeMo94], S. 412). Implizite Autorisierungen können über alle Dimensionen der Zugriffskontrolle beschrieben werden, d.h. über

- Autorisierungssubjekten
- Autorisierungsobjekten
- und Autorisierungsmodi,

wobei diese drei Dimensionen orthogonal zueinander stehen.

6.3.1. Implizite Zugriffsrechte für Autorisierungssubjekte

Innerhalb einer Anwendung kann man oftmals mehrere Benutzergruppen identifizieren, d.h. Benutzer können in unterschiedlichen Rollen auftreten. Betrachtet man Benutzergruppen als Mengen, bestehen zwischen ihnen Teilmengenbeziehungen. Folglich gelten die expliziten Zugriffsrechte einer Benutzergruppe auch für ihre echten Teilmengen. Sieht man die Zugehörigkeit eines Benutzers zu einer Benutzergruppe als Rolle, die er im System hat, spricht man auch davon, daß eine Benutzergruppe die Rechte der anderen erbt.
In Abbildung 11 ist beispielhaft ein Autorisierungs-Hierarchie von vier Gruppen (resp. Rollen) dargestellt. Die Pfeile zwischen den einzelnen Rollen stellen eine Vererbungsbeziehung bzgl. der Zugriffsrechte dar. Der Pfeil zwischen MoreLiteraturUser und MoreAdministrator bedeutet, daß ein MoreAdministrator (mindestens) die gleichen Rechte hat wie ein MoreLiteraturUser, d.h. die Rechte des MoreLiteraturUser erbt. Man kann mit dieser Autorisierungs-Hierarchie bei den Subjekten implizite Zugriffsrechte darstellen und es gilt: Falls es eine Abhängigkeitsbeziehung zwischen s' und s (s' erbt von s) in der Autorisierungs-Hierarchie gibt, gilt: (s, o, op) impliziert (s', o, op). Umgekehrt gilt bei negativer Autorisierung: (s', o, -op) impliziert (s, o, -op), d.h. z.B. wenn nicht einmal ein MoreAdministrator ein Objekt lesen darf, darf es auch kein MoreLiteraturUser.

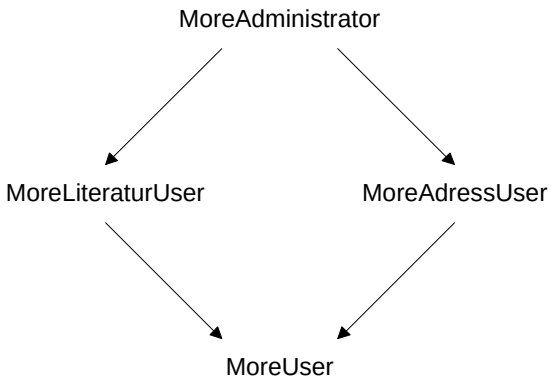

Abbildung 11: Beispiel einer Autorisierungs-Hierarchie über Subjekte

6.3.2. Implizite Zugriffsrechte für Autorisierungsobjekte

Ebenso wie bei den Subjekten kann auch über den Objekten einer Datenbank eine Autorisierungs-Hierarchie aufgestellt werden. Da in einer Datenbank üblicherweise deutlich mehr Objekte als Subjekte und Operationen vorhanden sind, hat implizite Autorisierung über dieser Dimension ein größeres Potential für die Einsparung expliziter Regeln als die beiden anderen Dimensionen (vgl. [KeMo94], S. 416). Aber aufgrund der großen Anzahl von Objekten wird die Darstellung einer Autorisierungs-Hierarchie auf Objektebene zu aufwendig (vgl. [KeMo94], S. 417). Aus diesem Grund werden die Implikationsregeln für Autorisierungsobjekte schematisch anhand eines **Autorisierungs-Schemas** dargestellt.

Ein Beispiel für ein solches Autorisierungs-Schema findet sich in Abbildung 12. Es legt implizite Zugriffsregeln auf Basis der objektorientierten Konzepte Klasse und Objekt fest. Gemäß der in der Abbildung dargestellte Implikationsregeln vererben sich die Zugriffsrechte für eine Klasse (in der Abbildung: Class) auch auf die innerhalb der Klasse definierten Attribute (Attribute) und Methoden (Methods). Alle in einer Klasse definierten Attribute und Methoden werden jeweils in einer eigenen Menge zusammengefaßt (Setof-Attributes und Setof-Methods). Aufgrund dieser Regeln wird gewährleistet, daß ein Subjekt, welches in einem festgelegten Modus auf eine Klasse zugreifen darf, auch das gleiche Zugriffsrecht auf die in dieser Klasse definierten Eigenschaften hat. Neben der Definition der Zugriffsrechte auf die Intension einer Klasse wird in der Abbildung auch das gleiche Zugriffsrecht für die Extension (Setof-Instances) der Klasse und jede einzelne Instanz (Instance) festgelegt. Jede Instanz der Extension einer Klasse wird bei Rabitti et al. als Menge von Attributwerten angesehen, so daß den Subjekten für den Zugriff auf die Attributwerte implizit das gleiche Zugriffsrecht gewährt wird wie für das Objekt.

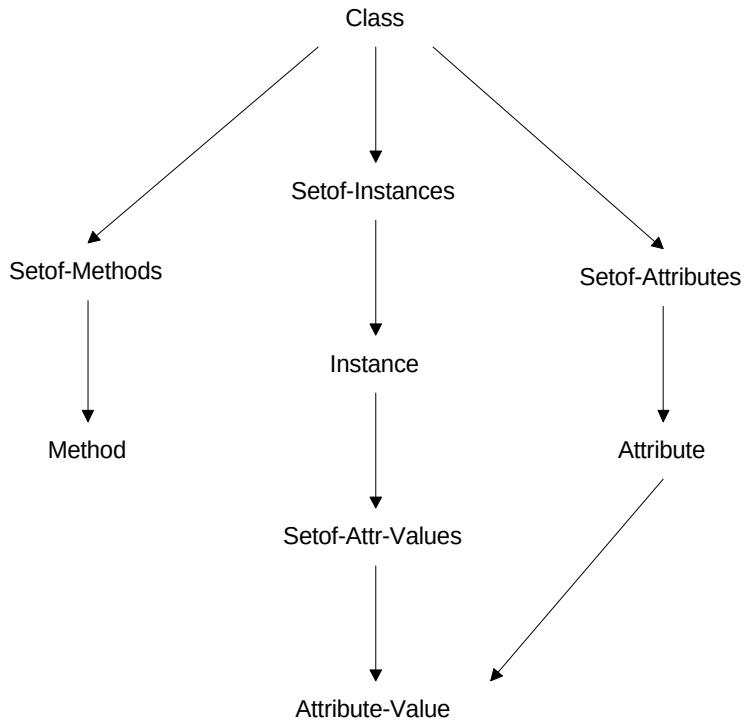

Abbildung 12: Autorisierungs-Schema für Objekte (vereinfacht aus [BeMa93], S. 168)

Das in Abbildung 12 dargestellte Autorisierungs-Schema skizziert nur einige der Möglichkeiten der impliziten Autorisierung anhand der Intension und der Extension einer Klasse. Die Darstellung erhebt dabei keinen Anspruch auf Vollständigkeit, da bspw. Implikationsregeln zwischen Instanzen und Methoden nicht erfaßt sind. Daneben bieten auch noch weitere Konzepte der Objektorientierung die Möglichkeit der Vergabe impliziter Zugriffsrechte. Hierbei ist zum einen an Implikationsregeln in der Vererbungshierarchie zu denken und zum anderen an implizite Autorisierung bei zusammengesetzten Objekten und deren Klassen. Einen kurzen Überblick hierüber liefern Saake et al. in [STS97] auf den Seiten 415 ff. Eine ausführlichere Beschreibung findet man in der Originalliteratur[58] von Rabitti et al. oder in [BeMa94] ab Seite 147.

6.3.3. Implizite Zugriffsrechte für Autorisierungsmodi

Die dritte Dimension zur Definition impliziter Zugriffsrechte, ist die der Modi des Zugriffs auf ein Objekt. In Abbildung 13 ist exemplarisch die Autorisierungs-Hierarchie der Operationen read (lesender Zugriff) und write (schreibender Zugriff) dargestellt. Mit der dort definierten Implikationsregel ist festgelegt, daß eine Autorisierung zum modifizierenden Zugriff eines Subjektes s auf ein Objekt o (s, o, write) immer auch die Erlaubnis des Subjekts zum lesenden Zugriff auf das gleiche Objekt (s, o, read) impliziert. Umgekehrt gilt bei expliziter negativer Autorisierung: (s, o, -read) impliziert (s, o, -write).

[58] Siehe [RBKW91].

Abbildung 13: Autorisierungs-Hierarchie von Operationen (aus [KeMo94], S. 416)

6.4. Autorisierung in GemStone

Im folgenden wird das Autorisierungsmodell von GemStone vorgestellt und zusammenfassend diskutiert. Für jeden berechtigten Benutzer existiert in GemStone ein Benutzerprofil (eine Instanz der Klasse UserProfile), in der neben seiner Kennung auch sein Paßwort verschlüsselt abgelegt wird. Zusätzlich werden in diesem Benutzerprofil auch alle Gruppen festgehalten, denen der Benutzer angehört. Mit dem Paar aus Kennung und Paßwort authentifiziert sich ein Benutzer beim Einloggen.

6.4.1. Autorisierungsmodell von GemStone

6.4.1.1. Explizite Autorisierung in GemStone

In GemStone dienen **Segmente** als logische Einheit zur Festlegung der Autorisierung bzgl. eines Objektes (vgl. [Bre89], S. 292). Dabei ist mit einem Segment nicht der physikalische Aufbau oder der Speicherort eines Objektes gemeint, sondern nur die Festlegung seiner Autorisierung (vgl. [Alm95e], S. 15). Jedem Objekt in GemStone ist genau ein Segment zugeordnet, d.h. es gibt kein Objekt, das in keinem oder in mehreren Segmenten liegt. Ein Segment hat keinerlei Kenntnis über die in ihm abgelegten Objekte. Das Segment eines Objektes legt bzgl. der Autorisierungsobjekte jeweils für

- den Eigentümer des Objekts (owner),
- Benutzergruppen in GemStone (groups),
- und für alle Benutzer in GemStone (world)

explizite Zugriffsrechte fest. Der Eigentümer ist der Benutzer, der das Segment angelegt hat und für die Vergabe der Zugriffsrechte für dieses Segment zuständig ist. Dabei kann er für verschiedene Gruppen von Benutzern Zugriffsrechte festlegen. Für alle anderen Benutzer kann die Autorisierung pauschal über das Synonym world zugewiesen werden. Die Festlegung der Autorisierung für einzelne Benutzer ist nicht möglich, sondern immer nur über deren Gruppen oder über „world“.
Für jedes Autorisierungssubjekt wird über das Segment eines Objekts explizit die Operation (Autorisierungsmodus) festgelegt werden. GemStone unterscheidet hier zwischen

- lesendem (read)
- schreibendem (write)
- oder keinem (none)

Zugriff. Lesender Zugriff erlaubt dem Subjekt hierbei das Objekt zu lesen, aber nicht es zu verändern (schreiben), wohingegen beim schreibenden Zugriff das Lesen und Verändern des Objektes zugelassen wird. Darf ein Subjekt überhaupt nicht auf ein Objekt zugreifen, wird dies durch den Autorisierungsmodus none festgeschrieben. Hierbei hat das entsprechende Subjekt zwar Zugriff auf die OID eines in diesem Segment abgelegten Objektes, darf aber über die Methoden weder lesend noch schreibend auf dessen Attribute zugreifen. Das Lesen der OID ist erforderlich, falls das gesperrte Objekt O2 von einem anderen O1 referenziert wird. In einem solchen Fall soll das Subjekt auf O1 ungehindert zugreifen können und somit auch die OID von O2 lesen dürfen, wohingegen beim Senden einer Nachricht an O2 der Zugriff verweigert wird (vgl. [Bre89], S. 293).
Die Zuordnung von Objekten zu Segmenten stellt den wichtigsten Aspekt dar. Hierzu bietet GemStone zwei verschiedene Möglichkeiten. Die erste beruht auf der Auszeichnung eines Segments als aktu-

elles Segment (im Sprachgebrauch von GemStone: **current segment**). Alle Objekte, die ein Benutzer erzeugt, werden automatisch in dem zum Zeitpunkt ihrer Instanzierung aktuellen Segment abgelegt. Das aktuelle Segment kann jederzeit innerhalb einer Sitzung geändert werden, falls Objekte in einem vom Standardsegment verschiedenen Segment abgelegt werden sollen. Diese Vorgehensweise erlaubt zwar zur Laufzeit eine automatische Zuordnung von Objekten zu Segmenten, ist aber auf der anderen Seite inflexibel, wenn Objekte unterschiedlichen Segmenten zugeordnet werden sollen.
An dieser Stelle setzt die zweite Möglichkeit der Zuordnung von Objekten zu Segmenten an. Jedes persistente Objekt in GemStone versteht die Nachrichten assignToSegment: und changeToSegment:. Durch das Senden der Nachricht assignToSegment: wird das Objekt in das als Parameter übergebene Segment verschoben. Bei diesem Vorgang wird nur das Objekt ohne die von ihm referenzierten Objekte verschoben. Sollen auch seine Komponentenobjekte in dem neuen Segment abgelegt werden, kann dies durch den Aufruf der Methode changeToSegment: realisiert werden. Im Gegensatz zu assignToSegment: muß diese Methode normalerweise neu implementiert werden, um festzulegen, welche Komponenten des Objekts in das neue Segment verschoben werden sollen.

6.4.1.2. Implizite Autorisierung in GemStone

Neben der Festlegung expliziter Zugriffsrechte bietet GemStone nur wenige Möglichkeiten für die Definition impliziter Zugriffsrechte, wovon aber einige vom System vorgegeben sind und andere mit den Mechanismen zur Autorisierung individuell dargestellt werden können. Bei den Operationen ist die implizite Autorisierung, daß aus dem Recht zum Schreiben eines Objektes auch das Recht zum Lesen dieses Objektes resultiert, im System festgelegt (siehe Abbildung 14 a)). Bei Autorisierungsobjekten existiert nur die implizite Autorisierung, daß für ein Objekt O' die gleiche Autorisierung gilt, wie für ein Objekt O, wenn O und O' im gleichen Segment liegen. Diese Regel ist in Abbildung 14 b) dargestellt, wobei hier die Funktion segment(O) ein Objekt O auf das Segment abbildet, in dem es liegt.
Es existieren weder implizite Autorisierungen bzgl. der Autorisierungssubjekte, noch implizite Autorisierungen bzgl. der Objekte, die die Konzepte objektorientierter Systeme ausnutzen. Dies bedeutet, daß zwischen den einzelnen Gruppen keine Teilmengenbeziehungen angegeben werden können, wodurch implizite Autorisierungen nur durch die Zuordnung von Benutzern zu mehreren Gruppen oder der Zuordnung mehrerer Gruppen zu Segmenten simuliert werden können. Bei den Autorisierungsobjekten können zumindest implizite Autorisierungen für Komponentenobjekte durch die Implementierung der Methode changeToSegment: dargestellt werden.

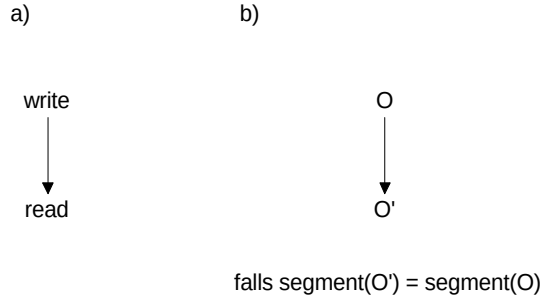

Abbildung 14: Implizite Autorisierung in GemStone

6.4.2. Zusammenfassende Beurteilung der Autorisierung in GemStone

GemStone bietet einen einfachen aber flexiblen Mechanismus zur Autorisierung, da für jedes Objekt beliebige Zugriffsrechte festgelegt werden können. Die einzige Einschränkung für die Gewährung eines Zugriffsrechtes auf ein Autorisierungsobjekt ist die Festlegung des Zugriffsrecht durch ein entsprechendes Segment. Die Grenze wird hier während des Entwurfs einzig durch die Anzahl der Segmente und damit die Übersichtlichkeit bei der Zuordnung von Objekten zu Segmenten gesetzt. Der Vorteil dieses Ansatzes ist, daß jedem Objekt dabei beliebige Autorisierungen über die Segmente zugeordnet werden können. Demgegenüber erfordert der Ansatz aber auch einen nicht unerheblichen Aufwand bei der Festlegung der Zugriffsrechte. Neben der expliziten Zuweisung der Zugriffsrechte

sind nur wenige Möglichkeiten für die Definition impliziter Zugriffsrechte vorhanden. Dies drückt sich bei den Subjekten darin aus, daß jede Gruppe, die auf die Objekte eines Segments zugreifen darf, explizit innerhalb des Segments berücksichtigt werden muß.
Implizite Zugriffsrechte anhand der Konzepte der Objektorientierung sind nicht global definierbar, was sich insbesondere darin ausdrückt, daß implizite Autorisierung nicht direkt über die Klassenzugehörigkeit, Vererbungsbeziehungen oder auch Komponentenobjekte möglich ist. Zur Darstellung impliziter Zugriffsrechte für Objekte können aber die Methoden zum Verschieben von Objekten zwischen Segmenten benutzt werden. Für referenzierte Objekte aller Instanzen einer Klasse können die Regeln in der Methode changeToSegment: beschrieben werden. Aufgrund der Möglichkeit der Redefinition von Methoden in Unterklassen und die Benutzung der Pseudovariablen super können auch Zugriffsrechte geerbt werden. Hierdurch müssen in der Methode changeToSegment: nur die in den entsprechenden Klassen definierten Variablen berücksichtigt werden. Die Zugriffsrechte für die geerbten Variablen können von den Methoden der Oberklassen durch einen super-Aufruf übernommen werden. Der Nachteil ist dabei aber, daß für jede Klasse festgelegt werden muß, ob die Zugriffsrechte der Oberklassen geerbt oder neue definiert werden müssen. Eine homogene Festlegung für alle Klassen in GemStone ist nicht vorgesehen. Auf der anderen Seite stellt dieses Verhalten aber einen Vorteil dar. Bertino und Martino diskutieren in [BeMa93] ab Seite 169, ob Autorisierungen geerbt werden sollen oder nicht, wobei sie die Extensionen von Klassen betrachten. Erben dabei die Unterklassen die Zugriffsrechte der Oberklassen, gilt für die Extension der Unterklassen, daß die darin enthaltenen Objekte bzgl. der Autorisierung genauso behandelt werden können wie die Objekte der Oberklasse. Auf der anderen Seite wird hierbei aber die Möglichkeit der Definition privater Objekte in Unterklassen ausgeschlossen. Würden Zugriffsrechte nicht geerbt, bestünde diese Einschränkung nicht. In GemStone muß diese Entscheidung nicht global getroffen werden. Hier kann für jede Klasse einzeln entschieden werden, ob Zugriffsrechte geerbt werden sollen oder nicht.
Darüber hinaus ist noch eine weitere Alternative möglich. Bertino und Martino betrachten nur gesamte Objekte, d.h. die Menge aller ihrer Attribute. Für die Autorisierung bedeutet das eine implizite Weitergabe der Zugriffsrechte an grundsätzlich alle Komponentenobjekte eines Objekts. In GemStone können die Attribute eines Objekts aber in unterschiedlichen Segmenten abgelegt werden, wodurch eine differenziertere Autorisierung möglich ist. Die Zugriffsrechte geerbter Attribute können komplett verändert und an neue Anforderungen der Unterklassen angepaßt werden. Dies kann aber die Konformität einer Klasse zu ihren Oberklassen einschränken. Ein Benutzer kann evtl. nicht mehr auf Instanzen von Klassen mit modifizierter Autorisierung zugreifen, obwohl er es bei den Oberklassen darf. Ein solcher Fall sollte nach Meinung von Buddrus nicht auftreten. Wenn ein Benutzer Zugriff auf die in einer Klasse definierten Attribute hat, dann sollte er auch in den Unterklassen ein vergleichbares Zugriffsrecht auf diese Attribute besitzen (vgl. [Bud96], S. 117). Dies kann in GemStone dadurch gesichert werden, wenn:

a) Jede Klasse nur die Zugriffsrechte für die durch sie neu definierten Attribute festlegt.

b) Zugriffsrechte grundsätzlich geerbt und nicht überschrieben werden.

Hierdurch wird gewährleistet, daß jeder Benutzer die Instanzen der Unterklassen einer für ihn freigegebenen Klasse zugreifen darf. In den Unterklassen neu definierte Attribute können jedoch durch stärkere Autorisierungen geschützt werden.

7. Zusammenfassung und Ausblick

Die Thematik objektorientierter Datenbanken ist durch einen großen Umfang an Konzepten, Mechanismen und behandelnder Literatur geprägt. Dies ist u.a. durch die Komplexität des Objektmodells bedingt, das gegenüber dem vergleichsweise einfachen relationalen Modell viele Ansatzpunkte für semantisch anspruchsvolle Umgestaltungen traditioneller Datenbankkonzepte bietet. Die Kernaspekte dieses Berichts werden hier zum Abschluß nochmals zusammengefaßt und die einzelnen Ergebnisse festgehalten. Das Kapitel wird durch einen abschließenden Ausblick auf Potentiale für weitere Betrachtungen und zukünftige Entwicklungen abgerundet.
Nach der Einleitung in die Materie in Kapitel 1 werden in Kapitel 2 Architekturen objektorientierter Datenbanken vorgestellt und diskutiert. Anhand dieser Diskussion wird speziell die Architektur von

GemStone beurteilend besprochen. Anschließend gibt Kapitel 3 einen Überblick über Persistenzmodelle objektorientierter Datenbanken im allgemeinen und GemStone im speziellen. Dem analogen Aspekt des Zugriff auf persistente Objekte widmet sich Kapitel 4. Aus dem Anspruch des parallelen Zugriffs mehrerer Benutzer auf eine objektorientierte Datenbank werden in den Kapiteln 5 und 6 Aspekte der Transaktionsverarbeitung resp. Autorisierung behandelt.
Zum Schluß bleibt noch die Frage nach der möglichen Zukunft objektorientierter Datenbanken. Hier stehen auf der positiven Seite die bereits in der Einleitung zu diesem Bericht aufgeführten Vorteile von ODBMS. Sie erlauben im Gegensatz zu traditionellen Systemen eine angemessenere Abbildung der Objekte des Problembereichs eines Softwaresystems. Innerhalb eines objektorientierten Entwicklungsprozesses können sie ihr Potential anhand eines friktionsarmen Übergangs der Objekte aus dem Analysemodell bis zur Implementierung und der damit verbundenen Datenhaltung ausspielen. Auf der negativen Seite stehen aber die Ausgereiftheit und Marktdominanz relationaler Systeme. Sicherlich wird auch die Anfang 2000 veröffentlichte Version 3.0 des ODMG-Standards eine wichtige Rolle spielen. Man sollte weiterhin beobachten, inwiefern dieser Standard auch auf breite Akzeptanz bei den Herstellern von ODBMS stößt und in verfügbare Produkte umgesetzt wird. Daneben kann auch eine evtl. Konvergenz der Standards ODMG 3.0 und SQL:1999 zu einer Vereinheitlichung von Datenbanken beitragen. Dittrich und Geppert sehen jedenfalls die Zukunft für ihre ideale Vorstellung nicht sehr positiv:

> *Unfortunately, the academically clean solution will not be among the winners: a clear-cut, comprehensive, object-oriented data model and system that would include the "relational world" as a special case. Whereas OODBMS-vendors are too small to engage in and promote such an approach, RDBMS-vendors do not have any interest in it since this would incur huge investments.*
>
> K. Dittrich und A. Geppert in [DiGe97], S. 281

Literatur- und Quellenverzeichnis

[ABD+89] Atkinson, M.; Bancilhon, F.; DeWitt, D.; Dittrich, K.; Maier, D.; Zdonik, S. (1989): „The Object-Oriented Database System Manifesto.“ In: Kim, W.; et al. (Hrsg.): „Procedings of the 1st International Conference on Deductive and Object-Oriented Databases.“ Amsterdam: Elsevier Science Publishers, S. 40-57

[Alh98] Alhir, S. (1998): „Unified Modeling Language: Extension Mechanisms.“ In: Distributed Computing (http://www.DistributedComputing.com), Dezember 1998, S. 29-32

[Alm95a] Almarode, J. (1995): „Multi-User Smalltalk.“ In: The Smalltalk Report, Band 4, Heft 4 (Januar 1995), S. 27-30

[Alm95b] Almarode, J. (1995): „Transactions in Smalltalk.“ In: The Smalltalk Report, Band 4, Heft 5 (Februar1995), S. 4-8

[Alm95c] Almarode, J. (1995): „Managing Concurrency Conflicts in Multi-User Smalltalk.“ In: The Smalltalk Report, Band 4, Heft 7 (Mai 1995), S. 15-17

[Alm95d] Almarode, J. (1995): „Queries in Smalltalk.“ In: The Smalltalk Report, Band 4, Heft 8 (Juni 1995), S. 17-19

[Alm95e] Almarode, J. (1995): „Object Security.“ In: The Smalltalk Report, Band 5, Heft 3 (November/Dezember 1995), S. 15-17

[Atk91] Atkinson, M. (1991): „A Vision of Persistent Systems.“ In: Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases, Lecture Notes in Computer Science, Band 566, S. 453-459

[AtMo95] Atkinson, M.; Morrison, R. (1995): „Orthogonally Persistent Object Systems.“ In: VLDB Journal Band 4, Heft 3, S. 319-401

[Atw94] Atwood, T. (1994): „Der Objekt-DBMS-Standard.“ In: OBJEKTspektrum, Heft 1/94 (März/April 1994), S. 32-40

[Ban92] Bancilhon, F.; Delobel, C.; Kanellakis, P. (1992): „Building an Object-Oriented Database System: The Story of O_2.“ San Mateo: Morgan Kaufmann Publishers

[Beck97] Beck, K. (1997): „Smalltalk: Praxisnahe Gebrauchsmuster.“ München et al.: Prentice Hall

[BeMa93] Bertino, E.; Martino, L. (1993): „Object-Oriented Database Systems: Concepts and Architectures.“ Wokingham et al.: Addison Wesley

[BlPr98] Blaha, M.; Premerlani, W. (1998): „Object-Oriented Modeling and Design for Database Applications.“ Upper Saddle River (New Jersey): Prentice Hall

[Boo94] Booch, G. (1995): „Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen.“ 1. Korrigierter Nachdruck. Bonn et al.: Addison Wesley

[BPSG95] Baraani-Dastjerdi, A.; Pieprzyk, J.; Safavi-Naini, R.; Getta, J. (1995): „A Model of Content-based Authorization in Objekt-Oriented Databases based on Object Views.“ Wollongong (Australien): Department of Computer Science, University of Wolongong

[Bre89] Bretl, R; et al. (1989): „The GemStone Data Management System.“ In: Kim, W.; Lochovsky, F. H. (Hrsg.) (1989): „Object-Oriented Concepts, Databases, and Applications.“ Reading (Massachusetts) et al.: Addison Wesley, S. 283-308

[Bud96] Buddrus, F. (1996): „Enacting Authorization Models for Object-Oriented Databases.“ In: Wagner, R.; Thoma, C. (Hrsg.): „Procedings of the 7th International Conference on Database and Expert Systems Applications.“ Zürich: IEEE Computer Society Press , S. 116-121

[Bur97] Burkhardt, R. (1997): „UML - Unified Modeling Language: Objektorientierte Modellierung für die Praxis.“ Bonn et al.: Addison Wesley

[CaBa97] Cattell, R.G.G.; Barry, D. (Hrsg.) (1997): „The Object Database Standard: ODMG 2.0.“ San Francisco: Morgan Kaufmann Publishers

[CAI98] Computer Associates (1998): „Jasmine Reference Version 1.21“

[Cat94a] Cattell, R.G.G. (Hrsg.) (1994): „The Object Database Standard: ODMG-93.“ San Mateo: Morgan Kaufmann Publishers

[Cat94b] Cattell, R.G.G. (Hrsg.) (1994): „The Object Database Standard: ODMG-93 – Release 1.1.“ San Mateo: Morgan Kaufmann Publishers

[ChLo98] Chaudhri, A. B.; Loomis, M. (1998): „Object Databases in Practice.“ Upper Saddle River (New Jersey): Prentice Hall

[CoMa84] Copeland, G.; Maier, D. (1984): „Making Smalltalk a Database System.“ In: Proceedings of the ACM/SIGMOD International Conference on the Management of Data, S. 316-325

[CSS99] Calic, M.; Sieling, B.; Simon, P. (1999): „Grundbegriffe der Objektorientierung.“ In: HMD: Praxis der Wirtschaftsinformatik, Heft 210, 36. Jahrgang (Dezember 1999), S. 7-22

[DaDa95] Darwen, H.; Date, C. (1995): „The Third Manifesto.“ In: ACM SIGMOD Record, Band 24, Heft 1 (März 1995), S. 39-49

[DiGe97] Dittrich, K.; Geppert, A. (1997): „Object-Oriented DBMS and Beyond.“ In: Lecture Notes in Computer Science, Band 1338, Heidelberg: Springer Verlag, S. 275-294

[Dit98] Dittrich, K. (1998): „Objekte und Datenbanken: Wohngemeinschaft oder Liebesheirat?“ In: OBJEKTspektrum, Heft 3/98 (Mai/Juni 1998), S. 20-21

[DRKU98] Dietrich, S.; Reghabi, S.; Krahling, D.; Urban, S. (1998): „On Acquiring OODBMS Technology: A Industry Perspective and a Case Study Comparison of Objectivity/DB and VERSANT“ In: [ChLo98], S. 193-209

[Ebe96] Ebert, J. (1996): „Software Engeneering II“ (Mitschrift der gleichnamigen Vorlesung im WS95/96, Universität Koblenz-Landau)

[EiMe98] Eisenberg, A.; Melton, J. (1998): „Standards in Practice.“ In: SIGMOD Record, Band 27, Heft 3 (September), S. 53-58

[EiMe99] Eisenberg, A.; Melton, J. (1999): „SQL:1999, formerly known as SQL3.“ In: SIGMOD Record, Band 28, Heft 1 (März), S. 131-138

[ElNa94] Elmasri, R.; Navathe, S. (1994): „Fundamentals of Database Systems.“ 2. Auflage. Redwood City (California) et al.: The Benjamin/Cummings Publishing Company.

[FlBe96] Floyd, R.; Beigel, R. (1996): „Die Sprache der Maschinen.“ Bonn et al.: International Thomson Publishing

[Fra94] Frank, U. (1994): „Multiperspektivische Unternehmensmodellierung: Theoretischer Hintergrund und Entwurf einer objektorientierten Entwicklungsumgebung.“ München: Oldenbourg

[Fra97a] Frank, U. (1997): „Towards a Standardization of Object-Oriented Modelling Languages.“ (Arbeitsbericht des Instituts für Wirtschaftsinformatik Nr. 3) Koblenz: Universität Koblenz-Landau

[Fra97b] Frank, U. (1997): „Enriching Object-Oriented Methods with Domain Specific Knowledge: Outline of a Method für Enterprise Modelling.“ (Arbeitsbericht des Instituts für Wirtschaftsinformatik Nr. 4) Koblenz: Universität Koblenz-Landau

[FrHa97] Frank, U.; Halter, S. (1997): „Enhancing Object-Oriented Software Development with Delegation.“ (Arbeitsbericht des Instituts für Wirtschaftsinformatik Nr. 2) Koblenz: Universität Koblenz-Landau

[Fri99] Friedrich, H. (1999): „Der Weg zum Projekterfolg: Management objektorientierter Entwicklungsprojekte.“ In: HMD: Praxis der Wirtschaftsinformatik, Heft 210, 36. Jahrgang (Dezember 1999), S. 23-36

[FrPr97] Frank, U.; Prasse, M. (1997): „Ein Bezugsrahmen zur Beurteilung objektorientierter Modellierungssprachen - veranschaulicht am Beispiel von OML und UML.“ (Arbeitsbericht des Instituts für Wirtschaftsinformatik Nr. 6) Koblenz: Universität Koblenz-Landau

[Gem96a] GemStone Systems, Inc. (1996): „GemStone System Administration Guide: Version 5.0 for Windows NT.“ Stand: Juli 1996

[Gem96b] GemStone Systems, Inc. (1996): „GemStone Programming Guide: Version 5.0.“ Stand: Juli 1996

[Gem96c] GemStone Systems, Inc. (1996): „GemBuilder for VisualWorks: Version 5.0.“ Stand: Juli 1996

[GHJV98] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1998): „Design Patterns: Elements of Reusable Object-Oriented Software.“ (CD-ROM-Version des gleichnamigen Buches) Reading (Massachusetts): Addison Wesley

[GJM91] Ghezzi, C.; Jazayeri, M.; Mandrioli, D. (1991): „Fundamentals of Software Engineering.“ Englewood Cliffs (New Jersey): Prentice Hall

[Gün88] Gündel, A. (1988): „Access Protection in a Distributed Smalltalk-System.“ (Forschungsbericht Nr. 271) Universität Dortmund

[Haase98] Haase, T. (1998): „Objektorientierte Softwareentwicklung mit Hilfe von Entwurfsmustern: Objektkonstruktion am Beispiel einer Literaturverwaltung im Kontext des Projektes MORE.“ In: Tagungsband MoBIS’ 98, S. 158-161

[Här95] Härder, T.; Mitschang, B. Nink, U.; Ritter, N. (1995): „Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen.“ In: Informatik Forschung und Entwicklung, Heft 10/1995, S. 55-72

[Heu97] Heuer, A. (1997): „Objektorientierte Datenbanken: Konzepte, Modelle, Standards und Systeme.“ 2. Auflage. Bonn et al.: Addison Wesley

[HMB90] Hosking, A. L.; Moss, J. E. B.; Bliss, C. (1990): „Design of an Object Faulting Persistent Smalltalk.“ (COINS Technical Report 90-45) Amhurst: University of Massachusetts

[Hoh98] Hohenstein, ,U. (1998): „Objektorientierte und objektrelationale Datenbanksysteme im Vergleich.“ In: OBJEKTspektrum, Heft 3/98 (Mai/Juni 1998), S. 22-27

[HSW97] Hohenstein, U.; Schmatz, K.; Weikert, P. (1997): „Architekturen von OO-Datenbanken: Tuning erwünscht.“ In: Objekt Fokus 1/97, S. 6-12

[JBR99] Jacobson, I.; Booch, G.; Rumbaugh, J. (1999): „The Unified Software Development Process.“ Reading (Massachusetts) et al.: Addison Wesley

[Jep97] Jepson, B. (1997): „The Object Database Management Group: What’s inside the ODMG-93 Standard.“ In: DBMS Magazine (http://www.dbmsmag.com/-9707d131.html) Juli 1997

[JoDi98] Jonscher, D.; Dittrich, K. (1998): „Welches ist das beste DBMS? Die Qual der Wahl zwischen relationalen, objektrelationalen und objektorientierten DBMS.“ In: Datenbank Fokus 12/98, S. 20-28

[Jun99] Jung, W. (1999): „A Survey of Lokcing-Based Concurrency Controls in Obejct-Oriented Databasese.“ In: Journal of Object-Oriented Programming, Mai 1999, S. 38-44

[Jung98] Jung, J. (1998): „Architekturentscheidungen bei der objektorientierten Modellierung und Entwicklung einer Literaturverwaltung im Kontext des Projektes MORE.“ In: Tagungsband MoBIS’ 98, S. 154-157

[KeMo94] Kemper, A.; Moerkotte, G. (1994): „Object-Oriented Database Management: Applications in Engineering and Computer Science.“ Englewood Cliffs (New Jersey): Prentice Hall

[KiLo89] Kim, W.; Lochovsky, F. H. (Hrsg.) (1989): „Object-Oriented Concepts, Databases, and Applications.“ Reading (Massachusetts) et al.: Addison Wesley

[Kim94] Kim, W. (1994): „Observations on the ODMG-93 Proposal for an Object-Oriented Database Language.“ In: SIGMOD Record, Band 23, Heft 1 (März), S. 4-9

[KrPo88] Krasner, G.; Pope, S. (1988): „A Cookbook for Using the Model-View-Controler User Interface Paradigm in Smalltalk-80.“ In: Journal of Object-Oriented Programming, August/September 1988, S. 26-49

[Kru99] Kruchten, P. (1999): „Der ‘Rational Unified Process’.“ In: OBJEKTspektrum, Heft 1/99 (Januar/Februar 1999), S. 38-42

[Lam91] Lamb, C.; Landis, G.; Orenstein, J.; Weinreb, D. (1991): „The ObjectStore Database System.“ In: Communications of the ACM, Band 34, Heft 10 (Oktober 1991), S. 50-63

[Lau98] Laux, F. (1998): „Datenbankmodellierung mit der UML.“ In: Tagungsband „Smalltalk und Java in Industrie und Ausbildung“ Erfurt, Oktober 1998

[Lin99] Linssen, O. (1999(: Szenarien und Use Cases in der objektorientierten Modellierung.“ In: HMD: Praxis der Wirtschaftsinformatik, Heft 210, 36. Jahrgang (Dezember 1999), S. 69-82

[Liu96] Liu, C. (1996): „Smalltalk, Objects and Design.“ Greenwich: Manning

[Loo91] Loomis, M. E. S. (1991): „More on Transactions.“ In: Journal of Object-Oriented Programming, Band 3, Heft 5, S. 63-67

[Loo92] Loomis, M. E. S. (1992): „Client-server architecture.“ In: Journal of Object-Oriented Programming, Band 4, Heft 9, S. 40-44

[Luf99] Lufter, J. (1999): „Objektrelationale Datenbanksysteme.“ In: Informatik Spektrum Band 22, Heft 4 (August 1999), S. 288-290

[MaMi94] Manola, F.; Mitchell, G. (1994): „A Comparison of Candidate Object Models for Object Query Services.“ (Discussion Paper No. X3H7-94-32v1) Waltham: GTE Laboratories

[Man94] Manola, F. (1994): „An Evaluation of Object-Oriented DBMS Developments.“ (Technical Report TR-0263-08-94-165) Waltham: GTE Laboratories

[McC87] McCullough, P. (1987): „Transparent Forwarding: First Steps.“ In: „Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings“ Orlando, Oktober 1987, S. 331-341

[MeWü97] Meier, A.; Wüst, T. (1997): „Objektorientierte Datenbanken: Ein Kompaß für die Praxis.“ Heidelberg: dpunkt-Verlag

[Mey97] Meyer, B. (1997): „Object-Oriented Software Construction.“ 2. Auflage. Englewood Cliffs (New Jersey): Prentice Hall

[Oes98] Oestereich, B. (1998): „Objektorientierte Softwareentwicklung: Analyse und Design mit der UML.“ 4. aktualisierte Auflage. München, Wien: Oldenbourg

[Oes99] Oestereich, b. (1999): „Wie setzt man Use-Cases wirklich sinnvoll zur Anforderungsanalyse ein?“ In: OBJEKTspektrum, Heft 1/99 (Januar/Februar 1999), S. 44-46

[Parc94] ParcPlace Systems (1994): „VisualWorks User’s Guide“, Revision 1.0

[Pra98] Prasse, M. (1998): „Die Objektklassifikatoren Typ, Klasse, Rolle und Schnittstelle innerhalb der objektorientierten Modellierungssprache MEMO-OML.“ In: Tagungsband MoBIS’ 98, S. 23-30

[Pra99] Prasse, M. (1999): „Objektorientierte Datenbanken: Architekturkonzepte, Standards und Nutzungsvoraussetzungen.“ In: HMD: Praxis der Wirtschaftsinformatik, Heft 210, 36. Jahrgang (Dezember 1999), S. 55-67

[RBKW91] Rabitti, F.; Bertino, E.; Kim, W.; Woelk, D. (1991): „A model of authorization for next-generation database systems.“ In: ACM Transactions on Database Systems, Band 16, Heft 1, S. 88-131

[RBP+93] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. (1993): „Objektorientiertes Modellieren und Entwerfen.“ München, Wien: Hanser Verlag; London: Prentice-Hall

[Rie97] Riehle, D. (1997): „Entwurfsmuster für Softwarewerkzeuge: Gestaltung und Entwurf von Anwendungen mit grafischer Benutzungsoberfläche.“ Bonn et al.: Addison Wesley

[RJB99] Rumbaugh, J.; Jacobson, I.; Booch, G. (1999): „The Unified Modeling Language Reference Manual.“ Reading (Massachusetts) et al.: Addison Wesley

[Sch92a] Schöning, U. (1992): „Logik für Informatiker.“ 3. überarbeitete Auflage. Mannheim, Leipzig, Wien, Zürich: BI-Wissenschaftsverlag

[Sch92b] Schöning, U. (1992): „Theoretische Informatik kurz gefaßt.“ Mannheim, Leipzig, Wien, Zürich: BI-Wissenschaftsverlag

[Sto90] Stonebraker, M. et al. (1990): „Third-Generation Database System Manifesto.“ In: ACM SIGMOD Record, Band 19, Heft 3 (September 1990), S. 31-44

[STS97] Saake, G.; Türker, C.; Schmitt, I. (1997): „Objektdatenbanken: Konzepte, Sprachen und Architekturen.“ Bonn et al.: International Thomson Publishing

[VoGr93] Vossen, G.; Groß-Hardt, M. (1993): „Grundlagen der Transaktionsverarbeitung.“ Bonn et al.: Addison Wesley

[Vos91] Vossen, G. (1991): „Objekt-orientierte Datenbank-Systeme.“ Manuskript zur Vorlesung an der Universität Koblenz-Landau im Wintersemester 1990/91

[Vos94] Vossen, G. (1994): „Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme.“ 2. Auflage. Bonn et al.: Addison Wesley

[Wei94] Weinberg, G. (1994): „Systemdenken und Softwarequalität.“ München, Wien: Hanser Verlag

Bisherige Arbeitsberichte

Hampe, J. F.; Lehmann, S.: Konzeption eines erweiterten, integrativen Telekommunikationsdienstes. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 1**, Koblenz 1996

Frank, U.; Halter, S.: Enhancing Object-Oriented Software Development with Delegation. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 2**, Koblenz 1997

Frank, U.: Towards a Standardization of Object-Oriented Modelling Languages? Arbeitsbericht des Instituts für Wirtschaftsinformatik, **Nr. 3**, Koblenz 1997

Frank, U.: Enriching Object-Oriented Methods with Domain Specific Knowledge: Outline of a Method for Enterprise Modelling. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 4**, Koblenz 1997

Prasse, M.; Rittgen,P.: Bemerkungen zu Peter Wegners Ausführungen über Interaktion und Berechenbarkeit, Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 5**, Koblenz 1997

Frank, U.; Prasse, M.: Ein Bezugsrahmen zur Beurteilung objektorientierter Modellierungssprachen - veranschaulicht am Beispiel vom OML und UML. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 6**, Koblenz 1997

Klein, S.; Zickhardt, J.: Auktionen auf dem World Wide Web: Bezugsrahmen, Fallbeispiele und annotierte Linksammlung. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 7**, Koblenz 1997

Prasse, M.; Rittgen, P.: Why Church's Thesis still holds - Some Notes on Peter Wegner's Tracts on Interaction and Computability. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 8**, Koblenz 1997

Frank, U.: The MEMO Meta-Metamodel, Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 9**, Koblenz 1998

Frank, U.: The Memo Object Modelling Language (MEMO-OML), Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr.** 10, Koblenz 1998

Frank, U.: Applying the MEMO-OML: Guidelines and Examples. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr.** 11, Koblenz 1998

Glabbeek, R.J. van; Rittgen, P.: Scheduling Algebra. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr.** 12, Koblenz 1998

Klein, S.; Güler, S.; Tempelhoff, S.: Verteilte Entscheidungen im Rahmen eines Unternehmensplanspiels mit Videokonferenzunterstützung, Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr.** 13, Koblenz 1997

Frank, U.: Reflections on the Core of the Information Systems Discipline. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr.** 14, Koblenz 1998

Frank, U.: Evaluating Modelling Languages: Relevant Issues, Epistemological Challenges and a Preliminary Research Framework. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr.** 15, Koblenz 1998

Frank, U.: An Object-Oriented Architecture for Knowledge Management Systems. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr.** 16, Koblenz

Rittgen, P.: Vom Prozessmodell zum elektronischen Geschäftsprozess. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 17**, Koblenz 1999

Frank, U.: Memo: Visual Languages for Enterprise Modelling. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 18**, Koblenz 1999

Rittgen, P.: Modified EPCs and their Formal Semantics. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 19**, Koblenz 1999

Prasse, M., Rittgen, P.: Success Factors and Future Challenges for the Development of Object Orientation. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 20**, Koblenz 2000

Schönert, S.: Virtuelle Projektteams - Ein Ansatz zur Unterstützung der Kommunikationsprozesse im Rahmen standortverteilter Projektarbeit. Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 21**, Koblenz 2000

Frank, U.: Vergleichende Betrachtung von Standardisierungsvorhaben zur Realisierung von Infrastrukturen für das E-Business. . Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 22**, Koblenz 2000

Jung, J.; Hampe, J.F.: Konzeption einer Architektur für ein Flottenmanagementsystem. . Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 23**, Koblenz 2001

Jung, J.: Konzepte objektorientierter Datenbanken – Konkretisiert am Beispiel GemStone. . Arbeitsberichte des Instituts für Wirtschaftsinformatik, **Nr. 24**, Koblenz 2001