Abu-Alwan, Ihssan; Schlagheck, Bernhard; Unland, R.

Working Paper

Evaluierung des objektorientierten Datenbankmanagementsystems ObjectStore

Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 43

Provided in Cooperation with:
University of Münster, Department of Information Systems

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgeber: Prof. Dr. J. Becker, Prof. Dr. H. L. Grob,
Prof. Dr. U. Müller-Funk, Prof. Dr. R. Unland, Prof. Dr. G. Vossen

Arbeitsbericht Nr. 43

Evaluierung des objektorientierten Datenbankmanagementsystems ObjectStore

Ihssan Abu–Alwan
Bernhard Schlagheck
Prof. Dr. R. Unland

Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster,
Grevener Str. 91, 48159 Münster, Tel. (0251) 83-9750, Fax (0251) 83-9754
Dezember 1995

Inhaltsverzeichnis

1 Einleitung

Traditionelle Datenbanksysteme bewältigen die heutigen hohen Anforderungen in CAD/CAM, CASE, datenintensiven KI-Anwendungen oder Bild- und Sprachverarbeitung oft nur in unzureichendem Maße. Zu den wesentlichen Nachteilen dieser Systeme zählen die unnatürliche Modellierung komplexer Sachverhalte, das Fehlen der Möglichkeit zur Definition von Datentypen und zur Verhaltensdefinition sowie die Kluft zwischen Programmiersprache und Datenbankanfragesprache (*impedance mismatch*). Ein vielversprechender Ansatz zur Lösung dieser Probleme ist das objektorientierte (OO) Paradigma. Realweltobjekte können mit diesem Ansatz natürlicher und i.d.R. auch effizienter modelliert werden. Durch die Konzentration auf Objekte als eigenständige und abgeschlossene Einheiten wird zudem die Wiederverwendbarkeit von Software unterstützt. Objektorientierte Datenbankmanagementsysteme (OODBMS) haben in diesem Zusammenhang die Aufgabe der persistenten Verwaltung von Objekten.

Die Entwicklung von OODBMS wird grundsätzlich aus zwei Richtungen bestimmt. Zum einen aus dem Datenbankumfeld und zum anderen aus dem Bereich der Programmiersprachen[1]. Aus dem Bereich der Datenbanken resultiert der *Bottom-up*-Ansatz, der Datenbanktechnologie um relevante objektorientierte Konzepte erweitert. Der *Top-down*-Ansatz wird entsprechend durch die Programmiersprachenwelt verfolgt und erweitert objektorientierte Sprachen um die notwendigen Datenbanktechnologien und -konzepte.

Der vorliegende Bericht untersucht die Leistungsfähigkeit des kommerziellen OODBMS ObjectStore in der Version 3.0 und zwar auf Basis des DML Interfaces. Ausgangspunkt dafür sind zwei Diplomarbeiten, die am Lehrstuhl für Praktische Informatik am Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster im Herbst 1994 durchgeführt wurden (s. [AbuA94, Schl94]).

Zunächst wird in Kapitel 2 die Architektur von ObjectStore beschrieben, danach in Kapitel 3 das Datenmodell untersucht. Kapitel 4 widmet sich dem Datenbank- und Laufzeitsystem von ObjectStore und abschließend werden Bewertungstabellen angeführt.

Für das grundlegende Verständnis der objektorientierten Konzepte und Modelle sei stellvertretend auf [Heue92] und [Unla95] verwiesen.

[1] Die gegenwärtig mit Abstand verbreitetesten objektorientierten Programmiersprachen sind *C++* und *Smalltalk*.

2 Die Architektur von ObjectStore

2.1 Die Client-Server-Architektur

Tabelle 1 (entnommen aus [ObTe94, S. 18]) verdeutlicht die Architektur von ObjectStore.

Server
Directory Manager
Datenspeicherung
Deadlock-Kontrolle
Datensicherung (Backup/Restore)
Wiederanlauf
Transaktions-Kontrolle
Client-Kommunikation

Client
API
Anfragen; Collections und Beziehungen
Versionen und Konfigurationen
Speicherverwaltung
Pufferverwaltung (Cache)
Schema-Service
Transaktionen
Server-Kommunikation

Tabelle 1: Client-Server-Architektur von ObjectStore

Abbildung 1 veranschaulicht das Zusammenspiel der ObjectStore-Komponenten *Server*, *Directory Manager* und *Cache Manager*. Clients fordern vom Server Seiten an, die dieser mit Hilfe des Directory Managers zur Verfügung stellt und verwaltet. Der auf dem Client laufende Cache Manager-Prozeß reduziert die Kommunikation zwischen Client und Server, indem er aktuelle Seiten puffert (vgl. [LLOW91, S. 56 ff.]).

2.2 Verteilung

ObjectStore bietet eine Client Server Architektur, die die Verteilung von Daten in einem Netzwerk durch eine parallele Nutzung von Datenbanken, die sogar auf Rechnern unterschiedlicher Architektur installiert sein können (vgl. [ObTe94, S. 21]), unterstützt. Diese Datenbanken durften in der Vergangenheit allerdings nur ObjectStore-Datenbanken sein und benötigten eine spezielle Anpassung (vgl. [ObPG93, S. Admin-98 ff.]). Seit Version 4.0 kann über die sogenannten *„Connectivity Services“* allerdings direkt auf Datenbanken anderer Hersteller zugegriffen werden. Weiterhin ist es erlaubt, daß Clients unter unterschiedlichen Betriebssystemen laufen können und trotzdem von einem Server bedient werden. Zur Zeit gibt es jedoch die Restriktion, daß Anwendungen, die gemeinsam mit einer Datenbank arbeiten wollen, durch den gleichen Compiler übersetzt sein müssen. Anders als in traditionellen Datenbankmanagementsystemen liegt in ObjectStore ein Teil des Datenbankmanagements, wie beispielsweise die Anfragebearbeitung, in der Verantwortung des Client. Dies ist transparent für den Anwender. Auf diese Weise wird neben einer möglichen Datenverteilung auch eine „Verarbeitungsverteilung“ in ObjectStore realisiert. In einer ObjectStore-Systemumgebung können mehrere Server installiert sein. Diese haben vor allem die Aufgabe, die Datenbanken mit Hilfe des Directory Managers zu verwalten. Der Directory Manager ist eine ObjectStore-Applikation, mit dessen Hilfe eine Datenverteilung vorgenommen werden kann. Die kleinste Einheit der Verteilung ist eine

einzelne Datenbank. Diese scheinbare Restriktion relativiert sich jedoch, wenn berücksichtigt wird, daß ein Schema logisch aus mehreren Datenbanken zusammengesetzt sein kann.

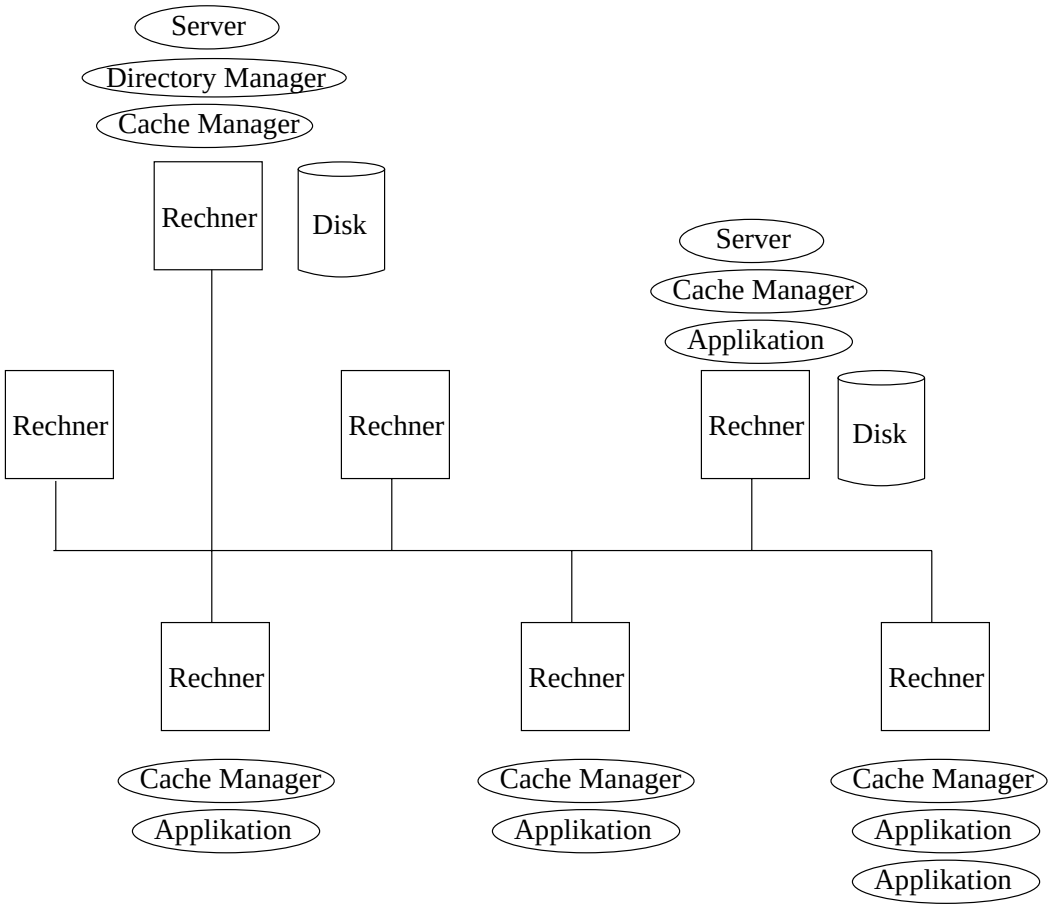

Abbildung 1: ObjectStore-Prozesse (entnommen aus [ObDM93, S. 14])

ObjectStore behandelt verschiedene Verteilungsaspekte wie folgt:

- Die Verteilungstransparenz wird durch den Directory Manager realisiert.
- Die Aufteilung der Daten und Programme muß durch den Benutzer erfolgen.
- Die globale Sicht wird durch das Schema gewährleistet. Die Anfragemöglichkeit ändert sich durch die Verteilung nicht.
- Die Anfragebearbeitung erfolgt durch den Client.
- Globale Transaktionen werden durch ein Zwei-Phasen-Freigabe-Protokoll synchronisiert (s. Unterabschnitt 4.8).
- Die globale Recovery ist abhängig von den globalen Transaktionen sowie den Datensicherungsmaßnahmen.

Die Verteilungsmöglichkeiten in ObjectStore sind zwar noch in der Ausbauphase, aber gut konzipiert und können weitgehend transparent verwendet werden.

3 Das Datenmodell von ObjectStore

3.1 Typen, Klassen und Metaklassen

Jeder Typ und jedes Objekt, das in Smalltalk, C oder C++ definiert werden kann, kann auch in ObjectStore abgelegt werden (vgl. [ObTe94, S. 11]).

ObjectStore übernimmt das Typen- und Klassenparadigma von C++. Es existieren zwei Arten von Typen: atomare und abgeleitete (vgl. [Stro92, S. 534]). Zu den atomaren Typen gehören die üblichen Basistypen wie `integer`, `float`, `char` zu verstehen (vgl. [Stro92, S. 57 f.]).

Bei den abgeleiteten Typen interessieren vor allem die Typkonstruktoren und die benutzerdefinierten Typen. Die Typkonstruktoren werden später im Rahmen der komplexen Objekte intensiver behandelt.

Benutzerdefinierte Typen werden, wie in objektorientierten Programmiersprachen üblich, über die Definition von *Klassen* realisiert. Eine Klasse besteht aus einer Struktur- und einer Verhaltensdefinition. Die Struktur der Instanzen einer Klasse wird durch die Aggregation von notwendigen Standarddatentypen, eingebetteten Klassen (auch *data members* genannt) und Zeigern auf (woanders abgelegte) Komponentenobjekte definiert. Das Verhalten der Instanzen der Klasse wird über Funktionen (*member functions*) realisiert. Spezielle Funktionen fungieren dabei als Objektfabrik, d.h. mit ihrer Hilfe können neue Instanzen einer Klasse erzeugt werden. Sie werden *Konstruktoren* (engl. *constructors*) genannt. Die Konstruktoren besitzen denselben Namen wie die Klasse, wobei eine Klasse auch mehrere Konstruktoren besitzen kann. Zusätzlich gibt es in C++ auch die komplementäre Funktion, den *Destruktor* (engl. *destructors*). Destruktoren werden durch den Klassennamen mit einem Komplementzeichen vor dem Namen deklariert. Die Hauptaufgabe von Destruktoren (auch *cleanup functions* genannt) ist es, Objekte durch Freigabe des von ihnen belegten Haupt– und Sekundärspeicherbereichs ordnungsgemäß zu löschen, (vgl. [JeRe92, S. 54]; [Stro92, S. 166]). Konstruktoren und Destruktoren müssen in ObjectStore explizit implementiert werden. Funktionen einer Klasse können einen Rückgabewert haben. Der Aufruf von Funktionen besitzt die Syntax `<Klassenname>::<Funktionsname>`. Klassen werden in C++ mit dem Schlüsselwort `class` definiert.

C++ bietet die Möglichkeit, *abstrakte* Klassen[2] zu definieren. Durch sie werden Modellierungen verständlicher und Redundanzen vermieden werden. Es ist nicht erlaubt, von abstrakten Klassen Instanzen zu bilden. Methoden abstrakter Klassen benötigen keine Implementierung, sie werden als *pure virtual functions* bezeichnet und mit 0 initialisiert. In den von den abstrakten Klassen abgeleiteten Klassen müssen solche Methoden dann explizit definiert werden (vgl. [JeRe92, S. 14]; [Stro92, S. 206 f.]).

In C++ wird die Definition von *generischen* Klassen[3] unterstützt. Sie dienen zur Erzeugung von konkreten Klassen[4]. Die Definition einer generischen Klasse enthält Parameter. Bei einer Instanziierung werden für diese Parameter konkrete Typen eingesetzt.

---

[2] *Abstrakte* Klassen werden auch *virtuelle Klassen* genannt.

[3] Generische Klassen heißen in C++ *Templates*.

[4] Die Kollektionen in ObjectStore sind generische Klassen. Nähere Erläuterungen dazu in Abschnitt 3.3.

C++ erlaubt die Definition von *statischen* Elementen (Attribute und Methoden) und somit die Modellierung von gemeinsamen Klassenelementen, auch *Metainformationen* genannt. Sie beschreiben die Klasse als solche und nicht einzelne Instanzen der Klasse. Die Definition solcher Elemente erfolgt über das Schlüsselwort `static`. Hinsichtlich der Speicherung von statischen Attributen behandelt ObjectStore diese genau so wie die Methoden. Sie werden nicht zusammen mit den anderen Attributen in der Datenbank, sondern in der Objektdatei von C++ (.o-File) gespeichert.

Die Unterstützungskonzepte von ObjectStore für Typen und Klassen sind umfangreich und ausgereift. Bei den statischen Elementen, die häufig für die Verwendung von Metaklassen verwendet werden, muß negativ angemerkt werden, daß sie nicht mit in der Datenbank abgelegt werden.

3.2 Kapselung

Über die Kapselung wird sichergestellt, daß Anwender nicht direkt auf die Daten einer Instanz zugreifen können, sonder nur über festgelegte Methoden. Damit wird sichergestellt, daß nur die freigegebenen Daten und diese nur in der definierten Form gesehen und manipuliert werden können (*information hiding*), Elemente einer Klasse können verschiedenartig gekapselt werden (vgl. [Lipp91, S. 242]):

(a) *public* (dt. öffentlich). Die öffentlichen Elemente sind für alle Klassen und Programmteile sichtbar. Auf sie kann daher uneingeschränkt zugegriffen werden.

(b) *private* (dt. privat). Private Elemente sind ausschließlich innerhalb der Klasse sichtbar. Nur die Funktionen der zugrundeliegenden Klasse haben freien Zugriff auf sie. Zugriff von außen ist nicht möglich. Eine Ausnahme bilden allerdings die *friend* Klassen, auf die weiter unten eingegangen wird.

(c) *protected* (dt. geschützt). Geschützte Elemente sind nur in der Klasse selbst und in ihren abgeleiteten Klassen sichtbar. Ein geschütztes Element verhält sich gegenüber einer abgeleiteten Klasse wie ein öffentliches Element.

In manchen Fällen ist es sinnvoll und wünschenswert, die Kapselung verborgener Elemente zu lockern. Zu diesem Zweck bietet C++ das Konzept der *friend* Klassen. Diese Klassen haben die Möglichkeit, auf die mit *private* geschützten Elemente zuzugreifen und damit die Kapselung zu verletzen. Außer durch *friend* Klassen kann die Kapselung durch das *Metaobjekt Protokoll* verletzt werden[5].

3.3 Komplexe Objekte

Objektkonstruktoren (Ausnahme der Tupelkonstruktor) werden in ObjectStore von parametrisierten Klassen[6] realisiert. Sie werden unter dem Begriff *Kollektionen* (engl. *collections*) eingeordnet. Kollektionen sind abstrakte Strukturen, die Objekte gruppieren. Sie

[5] Nähere Erläuterungen zum *Metaobjekt Protokoll* befinden sich in Abschnitt 4.11

[6] Objektkonstruktoren in ObjectStore können auch durch nicht–parametrisierte Klassen unterstützt werden. Dies ist vom Compiler abhängig. Der Unterschied zwischen parametrisierten und nicht–

sind vergleichbar den `arrays` in Programmiersprachen oder den *Tabellen* in relationalen Datenbanken (vgl. [LLOW91, S. 54]). Wie `arrays` können auch sie viele Instanzen des selben Typs speichern. Kollektionen in ObjectStore erlauben die Erzeugung von geordneten oder ungeordneten Objektmengen. Die Erzeugung von Elementduplikaten innerhalb der Menge kann für bestimmte Kollektionen erlaubt werden. Zu den Kollektionen in ObjectStore zählen folgende Typkonstruktoren:

(a) `os_Set<T>` (Mengenkonstruktor),

(b) `os_Bag<T>` (Multimengenonstruktor, d.h. Duplikate sind erlaubt),

(c) `os_List<T>` (Listenkonstruktor) und

(d) `os_Array<T>` (Arraykonstruktor)

Durch diese vier Konstruktoren wird die in [ABD+89, S. 5] gestellte Minimalmenge an Objektkonsturktoren erfüllt. Die Elemente einer Menge sind im Gegensatz zu denen einer Liste ungeordnet. Im Gegensatz zu Bags dürfen Mengen keine Duplikate enthalten. Bags sind wie Mengen ungeordnet. Listen und Arrays sind geordnete Kollektionen, die Duplikate enthalten können oder nicht. In ObjectStore ist Persistenz orthogonal zu diesen Typkonstruktoren[7]. Demnach kann eine Menge eines beliebigen Types transient oder persistent allokiert werden (vgl.[OHMS92, S. 404]).

Die Objektkonstruktoren Set, Bag und List können auch durch die direkte Instanziierung in der Klasse `os_Collection` erzeugt werden. Eine Instanz dieser Klasse kann zu einem Zeitpunkt entweder die Eigenschaften von Set, Bag oder List annehmen, doch besteht die Möglichkeit mit Hilfe der Methode `os_Collection::change_behavior()` das Verhalten der Instanz während ihrer Lebensdauer zu ändern (vgl. [ObDM93, S. 137]). Auf diese Objektkonstruktoren können verschiedene Operationen wie *insert*, *remove*, *retrieve* usw. und verschiedene Vergleichs- und Kopieroperatoren[8] ausgeführt werden.

Kollektionen werden oft als Klassencontainer (`class extents`) benutzt. Der Container einer Klasse beinhaltet Zeiger zu allen Klasseninstanzen (vgl. [ObDM93, S. 112]). Der Klassencontainer kann als Einstiegspunkt in die Datenbank fungieren.

Komplexe Objekte können Komponentenobjekte in unterschiedlicher Form beinhalten. Man unterscheidet einerseits zwischen *abhängigen* und *unabhängigen*, andererseits zwischen *gemeinsamen* und *exklusiven* Objekten (siehe z.B. [Unla95]). Aus diese Eigenschaften lassen sich folgende Kombinationen ableiten:

abhängig, gemeinsam

Abhängige gemeinsame Objekte werden in ObjectStore *nicht* automatisch unterstützt.

---

parametrisierten Klassen liegt darin, daß bei parametrisierten Klassen der Compiler den Elemententyp einer Kollektion kennt und somit die Typsicherheit garantiert. Das ist bei den nicht–parametrisierten Klassen nicht der Fall (vgl. [ObDM93, S. 101]

[7] Die Kollektionen sind nicht vollständig typorthogonal, da sie durch „Zeiger“ auf Typen (Klassen) realisiert werden. In zukünftigen Releases werden Kollektionen Klasseninstanzen besitzen. Somit wird den Kollektionen eine vollständige Typorthogonalität garantiert (vgl. [ObDM93, S. 105]

[8] Eine vollständige Auflistung aller zur Verfügung stehenden Operationen und Operatoren ist im *Reference Manual* ([ObRM93]) zu finden.

abhängig, exklusiv
: ObjectStore erlaubt die Modellierung von Objekten (Unterobjekten), die nur mit genau einem Vaterobjekt existieren können.

unabhängig, gemeinsam
: Objekte mit solchen Eigenschaften können in ObjectStore modelliert werden. Somit kann ein Objekt eigenständig, aber auch als Unterobjekt von einem oder mehreren Objekten existieren. ObjectStore gewährleistet zusätzlich die referentielle Integrität zwischen den Objekten.

unabhängig, exklusiv
: Ein unabhängiges exklusives Objekt bzw. Unterobjekt läßt sich in Objectstore *nicht* direkt modellieren.

Die *Klasse-Unterklasse-Beziehung* kann in ObjectStore durch Vererbung abgebildet werden. Eine Basisklasse ist eine Generalisierung der abgeleiteten Klasse, die andererseits Spezialisierung (*IS-A*) der Basisklasse ist.

Die Möglichkeiten zur Modellierung von komplexen Objekten in ObjectStore sind umfangreich. Die wichtigsten Objektkonstruktoren werden unterstützt.

3.4 Objektidentität

Objekte in ObjectStore besitzen eine Identität, die von ihren jeweiligen Attributwerten und Strukturen unabhängig ist. Die Identität wird durch systemweite, eindeutige Identifikatoren gewährleistet, die ObjectStore beim Anlegen der Objekte vergibt. Die Objektidentität besteht aus dem Datenbanknamen, einer Segmentnummer und einer Offsetnummer. Es handelt sich also um eine physikalische Objektidentität, da der Speicherplatz die Identität festlegt.

Ein wesentliches Konzept von ObjectStore ist, daß die Objektidentität in den meisten Fällen nicht verwendet muß, sondern die normalen hauptspeicherbezogenen C++ Zeiger ausreichend sind. Dies wird durch die von ObjectStore patentierte „Virtual Memory Mapping Architecture“-Technik realisiert.

Wird ein noch nicht geladenes Objekt (über C++) angesprochen, so lädt ObjectStore die entsprechende Seite nach und nimmt, falls der für sie ursprünglich vorgesehene Hauptspeicherplatz anderweitig belegt ist, für die auf dieser Seite befindlichen Objektverweise eine sogennante „Relocation“ (auch pointer swizzling) vor (vgl. [LLOW91, S. 58]). Die nötigen Informationen, wie Relocation durchgeführt werden muß, befinden sich in „Relocation“-Tabellen.

Die Objektverweise der neuen Seite sind also bereits aktualisiert, wenn das Programm nach dem Ladefehler die Kontrolle zurückerhält. Auf diese Weise können Objekte ohne direkte Verwendung der Objektidentität angesprochen werden. Das bietet den Vorteil, daß Objekte, die einmal geladen sind (und damit über C++ Zeiger angesprochen werden können), keinen weiteren Laufzeitoverhead verursachen.

3.5 Vererbung und Polymorphismus

Vererbung

Die Vererbung ist ein Grundkonzept des Datenmodells von C++ und damit auch von ObjectStore. Mit der Vererbung werden Eigenschaften der *Basisklassen* an *abgeleitete Klassen* (engl. *derived classes*) weitergegeben[9]. So werden die bereits existierenden Klassen bzw. abstrakten Datentypen um zusätzliche Fähigkeiten erweitert ([Stro92, S. 193]; [Lipp91, S. 409]). Die Vererbungsmechanismen in C++ sind sehr umfangreich. Sie unterstützen neben der Einfachvererbung auch die Mehrfachvererbung.

Die Vererbung kann, wie bei den Attributen einer Klasse auch, als *private* oder *public* deklariert werden. Im Gegensatz zu den Elementen einer Klasse können Klassen bei der Vererbung nicht als `protected` deklariert werden. Bei der Vererbung mit dem Schlüsselwort `private` werden alle Elemente der Basisklasse in der abgeleiteten Klasse als private Elemente betrachtet unabhängig davon, welchen Zustand sie in der Basisklasse haben[10]. Bei der Vererbung mit dem Schlüsselwort `public` behalten die Elemente ihre ursprüngliche, in der Basisklasse definierte Kapselung bei. Außerdem können in C++-Methoden der Basisklasse in der abgeleiteten Klasse redefiniert werden[11]. Bei der Vererbung bilden die Klassen eine Klassenhierarchie, d.h. die abgeleiteten Klassen können wiederum als Basisklassen dienen (vgl. [Stro92, S. 201]). Die Klassenhierarchie hat bei der Einfachvererbung die Gestalt eines Baumes und bei der Mehrfachvererbung die eines azyklischen, gerichteten Graphen.

C++ erkennt die Problematik der wiederholten Vererbung und des Namenskonflikts und bietet Möglichkeiten, diese zu beseitigen. Mit Hilfe der Mechanismen des sog. *information sharing* können Namenskonflikte, die sich auf eine Basisklasse (hier besitzt der Graph genau eine Wurzelklasse) beziehen, verhindert werden. Diese Vorgehensweise ist nur dann möglich, wenn die Vorgänger keine Änderungen an Attributen und Methoden der Wurzelklasse vornehmen. Mit der expliziten Deklaration der Wurzelkasse mit dem Schlüsselwort `virtual`[12] wird der Mechanismus des information sharing realisiert (vgl. [Stro92, S. 221 f.]). Erbt eine Klasse namentlich identische Methoden von zwei verschiedenen Basisklassen (hier kann der Graph mehr als eine Wurzelklasse besitzen), wird der Namenskonflikt durch eine explizite Klasssenangabe gelöst.

Nach der Untersuchung kann zusammenfassend festgestellt werden, daß Vererbungsmechanismen in ObjectStore gut unterstützt werden.

Polymorphismus

Die Möglichkeiten des Polymorphismus werden auf Basis der Vererbung, der Redefinition und des Überladens von Methoden in ObjectStore unterstützt. Polymorphismus und dynamisches Binden werden über *virtuelle Funktionen* (engl. *virtual member functions*) verwirklicht. Eine virtuelle Funktion kann in der abgeleiteten Klasse redefiniert oder erweitert

[9] In C++ wird eine Oberklasse *Basisklasse* und die Unterklasse *abgeleitete Klasse* genannt.
[10] Diese Art von Vererbung wird oft auch als *gesteuert* bezeichnet.
[11] Diese Eigenschaft wird im Rahmen des Polymorphismus (Abschnitt 3.5) diskutiert werden.
[12] Hier spricht man auch von virtuellen Basisklassen.

werden. Deshalb kann sie unterschiedliche Implementierungen in verschiedenen abgeleiteten Klassen besitzen. Die virtuellen Funktionen werden in C++ mit dem Schlüsselwort `virtual` gekennzeichnet. Bereits existierende Funktionen können nicht nachträglich ohne weiteres als virtuelle Funktionen deklariert werden (vgl. [JeRe92, 148 f.]). Das Laufzeitsystem findet beim Aufruf der virtuellen Methode mit Hilfe des dynamischen Bindens die entsprechende Version der Implementierung heraus.

ObjectStore unterstützt den *Ad-hoc-Polymorphismus* und den *universellen Polymorphismus*.

Zum Ad-hoc-Polymorphismus zählen das Overriding, das Overloading und die implizite Typkonversion (Coercien).

Overloading
Overloading ist in C++ verwirklicht, da eine Methode verschiedene Implementierungen besitzen kann. Der Compiler entscheidet hier anhand des Vergleichs der Typen der aktuellen Argumente mit den Typen der formalen Argumente, welche Implementierung einer Methode aufgerufen werden muß[13] (vgl. [Stro92, S. 641]). C++ unterstützt auch das (implizite) Überladen von Operatoren und erlaubt somit eine neue Definition (vgl. [JeRe92, S. 96]).

Typkonversion
C++ unterstützt die implizite Typkonversion. Deshalb können elementare Typen in Ausdrücken und Zuweisungen, wo es sinnvoll ist, frei komponiert werden (vgl. [Stro92, S. 59, S. 537ff.]).

Zum universellen Polymorphismus zählt der parametrische und der Teilmengen-Polymorphismus. Parametrischer Polymorphismus wird in C++ durch die *Templates* verwirklicht, Teilmengen-Polymorphismus durch die Vererbung.

3.6 Erweiterbarkeit

Eine Erweiterbarkeit auf interner Ebene ist in ObjectStore nicht möglich, auf konzeptueller Ebene, wie in objektorientierten Sprachen üblich, allerdings wohl: ObjectStore kann um neue Typen und Operationen erweitert werden. Solche Typen werden über Klassen definiert. Die Klassen und ihre Operationen können übersetzt und deren Objektdateien in Bibliotheken zusammengefaßt werden. Bei der Verwendung dieser Klassen muß mit Hilfe der Präprozessoranweisung `#include` die Headerdatei eingeladen werden. Die benutzerdefinierten Typen verhalten sich zwar wie die integrierten Systemtypen, können jedoch nicht als systemeigen betrachtet werden. Bibliotheken mit anwendungsspezifischen Klassen/Typen werden von verschiedenen Organisationen bereitgestellt[14].

[13] Eine Verwendung von Overloading findet bei der Benutzung von Konstruktoren statt. So besitzen sie den gleichen Namen wie ihre Klassen.

[14] Zu den Standardbibliotheken zählen folgende:
1. Tool.h++ von Rogue Wave Software Inc., 260 SW Madison, Corvallis, Oregon 97333 USA.
2. NIHCL (**N**ational **I**nstitute of **H**ealth **C**lass **L**ibrary): wird von ObjectStore als „Third–party–tool“ verwendet.
3. COOL (C++ object–oriented Library).

4 Datenbanksystem, Laufzeitsystem und Administration

4.1 Dauerhaftigkeit

Die Persistenz in ObjectStore ist typorthogonal und implizit, d.h. es werden keine expliziten Kopierbefehle benötigt, um Objekte persistent zu machen (s. [ObRM93, S. 3]). Es existieren allerdings mitgelieferte Klassen, deren Instanzen nicht persistent gemacht werden dürfen (z.B. `os_bound_query`). Objekte anderer ObjectStore-Klassen dürfen nicht transient allokiert werden. Dies kann zu Problemen führen, wie sich am Beispiel `os_configuration` zeigt. Erbt eine benutzerdefinierte Klasse von der ObjectStore Klasse `os_configuration` (s. [ObDM93, S. 203]), so können keine transienten Instanzen dieser Klasse angelegt werden. In den meisten Fällen sprechen semantische Gründe für diese Restriktionen. Der hierdurch entstehende Verstoß gegen die Typorthogonalität der Persistenz ist als gering einzustufen.

Die Persistenz wird durch eine Redefinition des Operators `new` realisiert. Als Argument wird die Datenbank angegeben, in der das Objekt persistent abgelegt werden soll. Ein einmal als persistent gespeichertes Objekt bleibt persistent, bis es explizit gelöscht wird. Transportbefehle zur Realisierung der Persistenz werden nicht benötigt. Als unbefriedigend hat sich erwiesen, daß Persistenz nicht durch Erreichbarkeit realisiert wird. Bei persistenten Objekten, die unabhängige Komponentenobjekte beinhalten, ist der Programmierer bzw. der Anwender dafür verantwortlich, daß die referenzierten Komponentenobjekte ebenfalls persistent gespeichert werden.[15] Praktisch gesehen, muß der Klassenmodellierer immer eine Fallunterscheidung zwischen transienten und persistenten Instanzen vornehmen.

Die in [Loom94] geforderte Typorthogonalität bezüglich Transaktionen, d.h. daß die Auswirkungen einer Transaktion zurückgesetzt werden unabhängig davon, ob sie durch persistente oder transiente Daten ausgedrückt werden, ist in ObjectStore nur für lokale Variablen erfüllt (s. [ObDM93, S. 98]). Variablen, die transient und nicht lokal sind, werden bei einer Transaktionszurücksetzung nicht berücksichtigt.[16]

Der nächste zu untersuchende Punkt ist die Art und Weise, wie Objekte persistent gespeichert werden. In ObjectStore werden nur Zustände von Instanzen in der Datenbank abgelegt, d.h. nur die Daten und nicht die Methoden werden gespeichert. Dies hat natürlich seine Gründe und auch weitreichende Konsequenzen.

Applikationen in C++ können modular aufgebaut werden, einzelne Module können separat entwickelt, übersetzt und anschließend zusammengebunden werden. Die Vorteile dieser Vorgehensweise bezüglich der Wiederverwendbarkeit sind offensichtlich. Die Schnittstellen zwischen den Modulen werden in sogenannten Headerfiles (`.h` - Files) festgelegt, während die Implementierungen nach der Übersetzung in ein bindefähiges Format in den Objektdateien (`.o` - Files) gespeichert werden. Diese können zu Bibliotheken zusammengefaßt werden. Der Binder des Betriebssystems wählt bei der Zusammenstellung einer

[15] Die in der Version 3.0 von ObjectStore eingeführten „access hooks“ bieten eine minimale Unterstützung, persistente und transiente Objekte zu mischen, vgl. [ObPG93, S. RN-7].

[16] Mit den in Version 3.0 eingeführten „transaction hooks“ läßt sich dies prinzipiell realisieren, benötigt jedoch eigene Programmierung, vgl. [ObPG93, S. RN-8].

Applikation aus diesen Bibliotheken automatisch die benötigten Implementierungen aus. Der Nachteil dieser Konzeption ist die Trennung des Verhaltens von der Struktur. Das Verhalten wird extern, d.h. in Dateien und daher durch das Betriebssystem verwaltet, während die Struktur eines Objektes in der Datenbank abgelegt wird. Bei gezielter Manipulation der Methoden können in zwei unterschiedlichen Anwendungen dieselben Objekte ein völlig unterschiedliches Verhalten aufweisen - ein Zustand, der höchst unerwünscht sein kann.

4.2 Anfragesprache

ObjectStore besitzt keine eigene Anfragesprache, es können jedoch Selektionsanfragen gestellt werden. Dies erfolgt in der entsprechenden sogenannten Gast-Sprache, die der Anwender zuvor auswählen muß. Die Gast-Sprache stellte die Schnittstelle zum Datenbanksystem dar. Die Sprache determiniert neben der Syntax für Objektdefinition auch die Syntax der Selektionsanfrage. In der „high-level"-Datenmanipulationssprache für C++ (DML[17]-Interface) können Selektionsanfragen mit einer besonders einfachen Syntax gestellt werden.

Eine SQL-Anweisung der Form

`select * from` *Menge* `where` *Bedingung*

wird durch die DML-Syntax

Menge`[:` *Bedingung*`:];`

simuliert. Zusätzlich zu dieser Syntax können Bibliotheksfunktionen zur Anfrage verwendet werden. Diese Funktionen sind die sogenannten *query-Funktionen* oder *Anfragefunktionen.*

Anwendungsunabhängigkeit

Anfragen in ObjectStore können an *collections* gestellt werden. Diese Möglichkeit ist unabhängig vom Inhalt und Typ der *collection*, also anwendungsunabhängig.

Deskriptive Sprache

Anfragen werden nur in der Form einer Selektion von Objekten aus einer Objektmenge unterstützt.

Der horizontale Zugriff auf die Objektmenge wird dadurch realisiert. Die DML-Syntax und die Anfragefunktionen bilden jedoch keine eigene Sprache.

Generische Operationen und Kapselung

Die DML-Syntax verhindert die Verletzung der Kapselung. Anfragefunktionen hingegen berücksichtigen die Kapselung nicht. Es ist möglich, explizite und implizite Attribute abzufragen. Vergleichsoperatoren wie z.B. `>` oder `<` können generisch verwendet werden, auch wenn sie überladen worden sind. Funktionen, die implizite Attribute darstellen, sollten keine Seiteneffekte produzieren, d.h. sie sollten keine Datenänderungen vornehmen. ObjectStore bietet gegen Seiteneffekte keine Maßnahmen an.

[17] DML darf hier nicht mit Abfragesprache verwechselt werden, sondern umfaßt den kompletten C++ Sprachumfang, ergänzt um die Syntax der Selektionsanfragen.

Orthogonalität

Bei den hier vorgestellten Konstrukten der Anfrage ist Orthogonalität gewährleistet. Insbesondere lassen sich Anfragen über Anfrageergebnisse stellen.

Optimierbarkeit und Effizienz

Optimierungen, d.h. effiziente Nutzung von Indizies, können in ObjectStore häufig erst zur Laufzeit erfolgen. Gründe hierfür liegen vor allem darin, daß einer Anfrage mehrere unterschiedlich aufgebaute Mengen zugrundeliegen können (vgl. [OHMS92, S. 408]). Die Anfragebearbeitung läuft in folgender Reihenfolge ab: Anfrageanalyse, Erstellen der Anfragestrategie und Ausführen der Anfragestrategie. Beim Aufstellen der Anfragestrategie wird untersucht, ob und welche Indizes existieren. Sind welche vorhanden, werden sie auf jeden Fall genutzt. Heuristiken finden leider keinen Einsatz, weshalb nicht wirklich von Optimierung gesprochen werden kann.

Vollständigkeit, Abgeschlossenheit und Angemessenheit

Anfragen können nur an Mengen gestellt werden. Dies impliziert, daß alle Objekte in Mengen erfaßt sind. Eine automatische Unterstützung des Klassencontainers wäre für diesen Zweck sinnvoll, wird jedoch von ObjectStore nicht unterstützt. Die objektorientierten Konzepte, wie beispielsweise komplexe Objekte und Vererbung, werden unterstützt, und auch die Abgeschlossenheit ist gewährleistet. Die Anforderungen bezüglich der Angemessenheit sind hingegen nicht erfüllt. So wird ausschließlich die Selektion, jedoch nicht die Projektion und der Verbund unterstützt. Bei Anfragen in ObjectStore wird die objekterhaltende Semantik verwendet. Es werden lediglich Referenzen auf Objekte in Mengen verwaltet. Aus diesem Grund ist es auch nicht notwendig, Objekte in die bestehende Klassenhierarchie einzuordnen. Mengen, die aus Referenzen auf Objekte bestehen, stehen quasi separat neben der Klassenhierarchie.

Die Fähigkeit zur Definition von Sichten wird, wie in allen anderen objektorientierten Datenbankmanagementsystemen, in ObjectStore nicht verwirklicht. Dies wäre insbesondere für die Realisation einer Zugriffskontrolle im Rahmen des Datenschutzes interessant. Anfragen in ObjectStore lassen viele Wünsche offen, insbesondere wird eine eigene Sprache benötigt. Ad-hoc-Anfragen können mittels eines ObjectStore-Zusatzprogrammes gestellt werden – dem `osbrowser`. Die Fähigkeiten des `osbrowser` sind jedoch äußerst eingeschränkt.

4.3 Transaktionsmanagement

Mit Transaktion wird in ObjectStore die traditionelle „kurze“ Transaktion bezeichnet. Lang andauernde Transaktionen, wie sie für Nicht-Standard-Anwendungen gefordert werden, werden nicht direkt unterstützt, können aber in bestimmten Fällen über Versionen und das Versionsmanagement simuliert werden (vgl. [ObDM93, ObTe94, S. 89, S.15]).[18]

Transaktionen können in ObjectStore durch zwei Varianten verwirklicht werden. Zum einen durch die Transaktionsanweisung (engl. *transaction statement*) und zum anderen durch dynamische Transaktionen. Für die Nutzung der Transaktionsanweisung müssen

[18] Für Versionsmanagement siehe Abschnitt 4.6.

die einzelnen Programmschritte in fester Sequenz vorliegen. Ist dies nicht gegeben, so kann eine dynamische Transaktion verwendet werden (vgl. [ObDM93, S. 89]). ObjectStore unterstützt die geschlossen geschachtelten Transaktionen in dem Sinne, daß innerhalb einer Transaktion (rekursiv) eine Subtransaktion gestartet werden kann. Die Folge aller Operationen einer geschachtelten Transaktion bleibt allerdings eine Sequenz, d.h. eine geschachtelte Transaktion stellt nur feinere Einheiten für die Recovery zur Verfügung (Subtransaktion). Paralleles Abarbeiten von Subtransaktionen oder kooperatives Arbeiten werden dadurch nicht ermöglicht. Innere Transaktionen können auch die sie umgebende Transaktionen abbrechen. Persistente Daten werden automatisch zurückgesetzt (vgl. [ObDM93, S. 97])[19], für transiente ist der Benutzer / die Anwendung selbst verantwortlich.

Transaktionen können in zwei unterschiedlichen Modi ablaufen, als `read_only`- und als `update`-Transaktionen, je nachdem, ob Daten nur gelesen oder auch verändert werden sollen. Die Synchronisation benötigt diese Angaben, um die entsprechenden Sperren zu setzen. Die Sperrstrategie von ObjectStore ist *pessimistisch*. Das Sperren wird automatisch und transparent für den Benutzer durchgeführt (s. [ObTe94, S. 9]). Es wird die aus konventionellen Datenbanken bekannte Technik des Zwei-Phasen Sperrens eingesetzt (s. [LLOW91, AWSL92, S. 57, S.39]). Die Granularität einer Sperre ist entweder eine Seite oder ein ganzes Segment (vgl. [ObRM93, S.365]). Objektabhängiges Sperren wird aus Performanzgründen nicht unterstützt (vgl. [ObTe94, S. 6]). Sperren werden erst nach dem Ende der äußersten Transaktion wieder freigegeben (s. [ObDM93, Solo92, S. 93, S. 93]). Anschließend werden alle Seiten aus dem Adreßbereich des Client entfernt und gegebenenfalls zurückgeschrieben. Nach einer erfolgreichen Rückmeldung des Servers verbleiben die Seiten aus Effizienzüberlegungen jedoch im Cachespeicher des Rechners. Es wird aber angenommen, daß die Wahrscheinlichkeit, daß nachfolgende Transaktionen mit (Teilen der) „alten“ Objekte arbeiten wollen, durchaus gegeben ist. Treten aus diesem Grund Sperrkonflikte mit anderen Rechnern auf, so verlangt der Server als zentrales Organ der Synchronisation die Seite vom Client zurück (*„callback message“*, vgl. [LLOW91, S. 57 f.]).

Eine Transaktion kann vom Anwender oder vom System abgebrochen werden. Systembedingte Transaktionsabbrüche können z.B. durch Netzwerkfehler oder Verklemmungen (engl. *„deadlocks“*) hervorgerufen werden (vgl. [ObDM93, S. 97 f.]). In diesen Fällen werden bei Verwendung der Transaktionsanweisung automatisch Wiederholungsversuche gestartet. Welche Transaktion bei einer Verklemmung zurückgesetzt wird, kann beim Start des Servers spezifiziert werden (s. [ObPG93, S. Admin-9]). Transaktionen, bei denen mehrere Server involviert sind, werden durch ein Zwei-Phasen-Freigabe-Protokoll abgearbeitet (vgl. [ObRM93, S. 447]).

4.4 Integritätsmechanismen

Für die Beurteilung der Integritätsmechanismen von ObjectStore ist zunächst die Unterstützung der Integritätsbedingungen zu untersuchen.

Schlüssel

Eine explizite Systemunterstützung für Schlüssel als Objektidentifikatoren ist in

[19] Mit Hilfe der Klasse `os_transaction_hooks` können benutzerdefinierte Funktionen aufgerufen werden, die es ermöglichen, auch nicht lokale Variablen zurückzusetzten.

ObjectStore nicht realisiert.

Kardinalitäten

ObjectStore unterstützt binäre Beziehungen zwischen Klassen. Im DML-Interface werden sie durch das Schlüsselwort `inverse_member` angezeigt. Die Art der Variablen, die durch `inverse_member` in Beziehung gesetzt werden, legt die Art der Beziehungsform ($1:1$, $1:n$ oder $n:m$) fest.

Die Beispieldeklaration in der Klasse LEHRSTUHL:

`os_Set<ASSISTENT*>` Assistenten `inverse_member` pLehrstuhl;

und die entsprechende Deklaration in der Klasse ASSISTENT:

LEHRSTUHL *pLehrstuhl `inverse_member` Assistenten;

legt eine 1:n Beziehung zwischen LEHRSTUHL und ASSISTENT an. Die Handhabung ist einfach und für die Einhaltung dieser Integritätsbedingungen sehr gut geeignet. Eine Unterstützung der Kardinalitäten von Mengen bietet ObjectStore nicht.

Inhärente Integritätsbedingungen

Die modellinhärenten Integritätsbedingungen werden hier nicht vollständig beschrieben. Eine im ObjectStore-Modell enthaltene Integritätsbedingung soll jedoch erwähnt werden. Ungültige Zeiger, also z.B. Zeiger von persistenten zu transienten Objekten, können auf `null` gesetzt oder ein Fehler kann signalisiert werden. Für eine ausreichende Behandlung des Problems der Referenzen auf transiente Objekte, wie es im Abschnitt 4.1 angesprochen wurde, reicht diese Fähigkeit nicht aus.

Allgemeine Integritätsbedingungen

ObjectStore unterstützt keine expliziten Integritätsbedingungen, insbesondere wird kein Triggermechanismus angeboten. Sämtliche anwendungsunabhängigen Integritätsbedingungen müssen über Methoden implementiert werden. Eine Systemunterstützung in diesem Bereich würde die Sicherheit erhöhen und den Modellierungsaufwand herabsetzen. Ein Ereignis, wie z.B. das Löschen eines Objektes, könnte dann beispielsweise in allen umgebenden Objekten („Oberobjekten“) abgefangen und eine entsprechende Aktion ausgeführt werden. Eine Programmierung dieser Funktionalität ist mit weit mehr Aufwand verbunden.

Bei der Untersuchung der Integritätsbedingungen fällt zusammenfassend auf, daß die Kardinalitätsbedingungen gut unterstützt, die Forderung nach Schlüsseln und einem Triggermechanismus jedoch nicht erfüllt werden.

4.5 Schemaevolution

Die Schemaevolution in ObjectStore wird notwendig, wenn die Schemavalidation eine Änderung in der Klassendefinition feststellt. Dies ist ausschließlich bei internen Objekt-

strukturänderungen der Fall. Das Hinzufügen einer „normalen“ Funktion wird beispielsweise nicht berücksichtigt.

Der Schemaevolutionsprozeß in ObjectStore wird in zwei Phasen eingeteilt, zum einen in die Modifikation der Schemainformation und zum anderen in die Instanzmigration. Die Instanzmigration selbst erfolgt in zwei Teilphasen, die Instanzinitialisierung und die Instanztransformation (s. [ObDM93, S. 306]). Bei der Instanzinitialisierung werden Speicherkomponenten so initialisiert, daß sie konform mit der neuen Klassendefinition sind. Dies ist beispielsweise beim Hinzufügen eines Attributes erforderlich. Für den Fall, daß anwendungsabhängige Instanzänderungen vorgenommen werden sollen, kann die Instanztransformation verwendet werden. Bei dieser kann der Anwender genau spezifizieren, wie jede einzelne Instanz modifiziert werden soll. In einigen Fällen muß dazu auf Werte zugegriffen werden, die in der neuen Klassendefinition nicht mehr vorhanden sind, wohl aber im alten Klassenschema. Dies kann mit Hilfe des Metaobjekt Protokoll „MOP“ (siehe Abschnitt 4.11) erreicht werden. Problematisch ist bei den Transformationsfunktionen die Kapselung. Soll beispielsweise ein privates Attribut von `int` nach `char` umgewandelt werden, so können die Zugriffsfunktionen nicht auf das alte Objekt angewendet werden. Abhilfe schafft auch hier das MOP. Eine einfachere Handhabung der Instanztransformation wäre sehr wünschenswert.

An dieser Stelle werden die in [BSG+88, S. 449] angeführten Änderungsmöglichkeiten für die Schemaevolution von ObjectStore betrachtet.

(a) Klassendefinitionsänderungen

  (i) Strukturänderungen
  Das Löschen von Attributen ist problemlos möglich. Das Hinzufügen von Attributen bedarf der Verwendung des MOP, falls die Initialisierung dieser Attribute auf Werte von anderen Attributen beruht. Das Ändern des Wertebereichs ist problemlos, sofern die Wertebereiche für das C++-Laufzeitsystem zuweisungskompatibel sind. Semantische Probleme können dadurch entstehen, daß keine Evolution bei Änderungen von `signed-` nach `unsigned`-Datentypen erforderlich ist. Namenskonflikte durch Vererbung von gleichnamigen Attributen werden bereits während des Übersetzungsvorganges entdeckt. Änderungen des Vorgabewertes von Attributen erfordern keine Schemaevolution, ebenso die üblicherweise als `static`-Variablen realisierten Klassenattribute. Static-Variablen werden extern verwaltet.[20]

  (ii) Verhaltensänderungen
  Verhaltensänderungen erfordern in ObjectStore nur dann eine Evolution, wenn sie die interne Strukturrepräsentation modifizieren (s. o.).

(b) Änderungen innerhalb der Klassenhierarchie
Folgende Klassenhierarchieänderungen sind in ObjectStore realisierbar (s. [ObDM93, S. 357 ff.]):

  (i) eine Klasse O zur Oberklasse der Klasse K machen

  (ii) Löschen der Klasse O aus der Liste der Oberklassen von Klasse K

[20] Static-Variablen werden wie die Funktionselemente in den `.o`-Dateien gespeichert.

(iii) Ändern der Reihenfolge der Oberklasse einer Klasse K

(c) Änderungen einer Klasse
Das Hinzufügen und Löschen einer Klasse ist problemlos zu verwirklichen. Eine Namensänderung von Klassen wird nicht unterstützt.

Die Schemaänderungen können in der Regel nicht interaktiv durchgeführt werden, eine Anwendung muß nach der Änderung neu übersetzt werden. Die Instanzen werden im Evaluationsprozeß automatisch oder durch benutzerdefinierte Transformationsfunktionen geändert.

4.6 Versionsmanagement

Versionen in ObjectStore können auf beliebige Typen angewendet werden. Zwei wichtige Konzepte im Versionsmanagement von ObjectStore sind die Konfigurationen (engl. *configurations*) und die Arbeitsbereiche (engl. *workspaces*).

Konfigurationen
Konfigurationen dienen dazu, Objekte zu gruppieren, die für den Zweck der Versionierung als Einheit behandelt werden müssen. Sie sind als ein Werkzeug zur Gestaltung von Komponenten (engl. *design component*) anzusehen. Eine Konfiguration kann aus einem einzelnen Objekt, verschiedenen Objekten oder aus einem vollständigen Design bestehen.

Eine neue Version eines bestimmten Objekts kann durch dessen Konfiguration erzeugt werden. Dies geschieht durch das *checkout* und *checkin* von Objekten von oder zu privaten und Mehrplatz-Arbeitsbereichen (*workspaces*). So können sowohl „lineare“ als auch „hierarchische“ Versionsverläufe (*version history graph*) kreiert werden (vgl. [AWSL91, S. 38]). Konfigurationen können in ObjectStore verschachtelt werden, d.h. sie können in andere Konfigurationen eingebettet werden. Dabei kann ein Sperren (*checkout*) der ganzen Konfiguration einschließlich seiner Unterkonfigurationen oder nur der Unterkonfigurationen durchgeführt werden. Die Konfiguration eines bestimmten Objektes kann in ObjectStore entweder durch die Deklaration einer persistenten Variable vom Typ `os_configuration` oder durch direkte Vererbung dieses Typs geschehen (vgl. [ObDM93, S. 189]). Von einer Konfiguration kann zu einem Zeitpunkt nicht nur eine Version sondern auch alternative von anderen Benutzern angelegte Versionen existieren. Das ermöglicht die Bearbeitung einer bestimmten Version, auch wenn diese parallel schon von anderen benutzt wird. ObjectStore verhindert dabei das Auftreten von Konflikten und die Verklemmung von Transaktionen (*deadlocks*). Die alternative Version wird mit der Methode `checkout_branch()` der Klasse `os_configuration` erzeugt. ObjectStore unterstützt das Einfrieren von Konfigurationen, falls weitere Versionen nicht gewünscht sind. Die eingefrorenen Konfigurationen können aber zu einem späteren Zeitpunkt weiterverarbeitet werden. ObjectStore unterstützt ferner die Möglichkeit, verschiedene Versionen zu verschmelzen. Das Vergleichen und Verschmelzen von Versionen muß vom Benutzer implementiert werden.

Arbeitsbereiche
ObjectStore unterstützt zwei Arten von Arbeitsbereichen, nämlich die *privaten* und *gemeinsamen/globalen* Arbeitsbereiche. Objekte einer Konfiguration können erst in einem gültigen Arbeitsbereich manipuliert werden. Die Arbeit an einer gemeinsamen Konfiguration innerhalb eines öffentlichen Arbeitsbereichs erfordert die Ableitung eines neuen

privaten Arbeitsbereichs von einem öffentlichen. Dieser wird dann als ein gültiger Arbeitsbereich zwecks privater Weiterbearbeitung benutzt. Dadurch wird eine Arbeitsbereichshierarchie erzeugt. Persistente Arbeitsbereiche werden durch die Instanziierung der Klasse `os_workspace` erreicht.

ObjectStore stellt zahlreiche Methoden für die Unterstützung von Versionen zur Verfügung. Mit deren Hilfe können Konfigurationen und Arbeitsbereiche u.a. erzeugt, geändert und gelöscht werden. Zeitversionen werden in ObjectStore nicht automatisch unterstützt. Man kann aber den Zeitpunkt der Realisierung, Speicherung und die Gültigkeit als Versionsidentität durch die Methode `os_configuration::name_version()` angeben.

Bei der Untersuchung des Versionsmanagements von ObjectStore kann zusammenfassend gesagt werden, daß ObjectStore „mächtige" Versionsmechanismen unterstützt, wenn von den Schwächen im Bereich der Zeitversionen abgesehen wird.

4.7 Sekundärspeicherverwaltung und Zugriffsmethoden

Der Einsatz des Sekundärspeichers in ObjectStore erweitert den virtuellen Speicher in ähnlicher Weise, wie der virtuelle Speicher den physischen Speicher eines Rechners erweitert (vgl. [ObTe94, Abb. 2.1 S. 24, und Abb. 3.1 S. 31]). Dies wird durch eine spezielle Speicherabbildungstechnik, der sogenannten VMMA (***V**irtual **M**emory **M**apping **A**rchitecture*) realisiert. ObjectStore nutzt das Verwaltungssystem der virtuellen Hauptspeicherverwaltung, indem es Einfluß auf den durch das Betriebssystems verwalteten Status von Seiten (*no access, read only, read/write*) nimmt. Wird ein Objekt referenziert, das sich noch nicht im virtuellen Speicher des Rechners befindet (d.h. es wird z.B. eine Seite angesprochen, die den Status *„no access"* besitzt), so meldet das Betriebssystem an ObjectStore einen Seitenfehler. ObjectStore lädt die entsprechende Seite von der Platte, setzt den Status der Seite auf *read only* und setzt gleichzeitig eine Lesesperre auf die Seite. Soll später ändernd auf die Seite zugegriffen werden, erfolgt wegen des *read only*-Schutzes wieder ein Interrupt. Es wird jetzt eine Schreibsperre vergeben (falls kein Konflikt vorliegt) und der Status der Seite auf *read/write* gesetzt. Spätere Zugriffsversuche laufen mit der Geschwindigkeit des virtuellen Speichers ab. Objekte sind im persistenten Speicher genauso abgelegt wie im transienten Speicher, sodaß in den meisten Fällen ein Nachladen einer Seite vom Sekundärmedium ausreichend ist (kein *pointer swizzeling*). Kann eine Seite nicht in die zugehörige Seite des virtuellen Speichers geladen werden, so ist eine Adreßtransformation notwendig (*relocation* oder *pointer swizzling* genannt) (vgl. [LLOW91, ObTe94, S. 56 ff., S. 23 ff.]).

Die Implementierung der Objektidentität erfolgt über eine physikalische Speicheradresse, die aus dem Tripel (Datenbankname, Segmentnummer, Offsetnummer) besteht. Durch diesen Sachverhalt steht in ObjectStore ein Adressraum von 2^{89} Byte zur Verfügung. Üblicherweise arbeitet der Anwender nicht mit diesen Identifikatoren, sondern er nutzt die üblichen C++-Zeiger. ObjectStore erlaubt diese transparente Nutzung der Zeiger durch die oben beschriebene Speicherabbildungstechnik.

In [HuPC93, S. 57 f.] wird zwischen einfachen und höheren Clusterangaben unterschieden. Die einfachen Angaben beziehen sich auf einzelne Objekte, die höheren auf Objektbeziehungen. Letztere werden in ObjectStore nicht unterstützt. Stattdessen werden

Benutzerangaben darüber erwartet, welches Objekt in der Nähe welches anderen Objektes gespeichert werden soll. Ein dynamisches Re-Cluster ist nicht möglich. Die Clusterungsmöglichkeiten in ObjectStore sind als rudimentär und nicht transparent anzusehen.

Objekte werden seiten- oder segmentweise vom Server zum Client transportiert und dort gepuffert. Zeitintensive Netzwerkkommunikation wird dadurch herabgesetzt (vgl. [ObTe94, S. 24]).

ObjectStore nutzt Anwenderangaben zur Indexverwaltung. Attribute, die einen Index erhalten sollen, müssen mit dem reservierten Wort `indexable` gekennzeichnet werden. Ein Index wird im Zusammenhang mit Collections verwendet. Der Zugriff auf Elemente einer Collection wird dadurch optimiert. Indizes können dynamisch vom Benutzer erstellt und gelöscht werden. Die Wartung dieser Indizes erfolgt automatisch durch das System (s. [ObDM93, S. 149 f.]). Ein Index kann auch für Attribute erstellt werden, die Komponentenobjekte beinhalten. Hierzu werden sogenannte *rank*- und *hash*- Funktionen benötigt (s. [ObDM93, S. 163 f.]).

Die Sekundärspeicherverwaltung und die Zugriffsmethoden in ObjectStore zeichnen sich durch weitgehend transparente und effiziente Verwendung aus. Eine Ausnahme bildet die Clusterungstechnik, die vom Benutzer zu detaillierte Angaben verlangt.

4.8 Wiederanlauf

Die Wiederanlaufmaßnahmen dienen der Datensicherheit. Die Komponenten, die dies gewährleisten, sind in ObjectStore die *Log-Datenbank* (vgl. [ObPG93, S. Admin-6 f.]) und Programme zur Erstellung und Wiederherstellung von Sicherungskopien (vgl. [ObTe94, S. 51 ff.]). Änderungen, die im Verlauf von Transaktionen vorgenommen werden, werden mittels eines *„write-ahead-log“* Protokolls (vgl. [Reut87, S. 449]) in die Log-Datenbank geschrieben. Transaktionen, bei denen mehrere Server beteiligt sind, werden durch das bereits erwähnte Zwei-Phasen-Freigabe-Protokoll koordiniert (vgl. [LLOW91, S. 57]). Beim erfolgreichen Abschluß einer Transaktion werden die Informationen permanent gespeichert. Die ObjectStore-Versionen nach 3.0 werden zusätzlich mit einer Kopie der Log-Datenbank arbeiten (s. [ObTe94, S. 52]).

Sicherungskopien können ohne Unterbrechung des Systems vorgenommen werden. Diese können vollständig oder inkrementell erstellt und fortgeführt werden. Die Konsistenz ist aus Transaktionssicht gewährleistet. Die notwendigen Programme sind `osbackup` und `osrestore` (s. [ObPG93, S. Admin-32 f. und S. Admin-43 ff.]).

Die Unterstützungskonzepte von ObjectStore für das Recovery sind umfangreich und ausgereift.

4.9 Effizienz

Als Grundlage für die Effizienzbewertung soll der in [CaDN94c] beschriebene OO7-Benchmark herangezogen werden. Dieser Test bietet eine umfassende Untersuchung der Leistungsmerkmale eines objektorientierten Datenbankmanagementsystems.

Probleme ergeben sich jedoch mit der Zuverlässigkeit der dokumentierten Resultate. Der Grund hierfür ist darin zu sehen, daß ObjectStore nicht am offiziellen Test teilgenommen hat, sondern in einer Vergleichsumgebung getestet worden ist.[21] Dennoch werden die Resultate des ObjectStore-Tests mit denen des offiziellen OO7-Benchmark verglichen. Dies erfolgt über die Methode der gewichteten Rangfolge, wie sie in [CaDN94b] vorgestellt wird. Eine intensivere Untersuchung der Meßergebnisse kann [CaDN94c] und [ObOO93a] entnommen werden. Die hier zu Grunde gelegten Daten sind [CaDN94a] und [ObOO93a] entnommen.

In den Tabellen 4.1 bis 4.6 werden die Vergleichsergebnisse zusammengefaßt. Sie sind wie folgt zu interpretieren: für alle zugehörigen Testfälle wird die Anzahl der erreichten Plätze des einzelnen Systems ausgezählt. Eine Gewichtung der Plätze (1. Platz ein Punkt, 2. Platz zwei, usw.) kann anschließend zur Rangfolgenermittlung herangezogen werden. Die untersuchten Systeme sind:

- Exodus Version 2.2 (Ex),
- Ontos Version 2.2 (On),
- Objectivity/DB Version 2.1 (Ob),
- Versant Version 3.0 Beta (Vr) und
- ObjectStore Version 2.0.1 (OS).

Die Tabelle 4.1 bietet eine Gesamtübersicht über 102 Testfälle. ObjectStore erreicht mit dem hier benutzten Verfahren den zweiten Rang. In den Tabellen 4.3 und 4.4 sind die Ergebnisse der Tests mit der kleinen bzw. mittleren Datenbank ersichtlich. ObjectStore schneidet bei der mittelgroßen Datenbank besser ab. Tabelle 4.5 und 4.6 stellen die Resultate bei navigierendem bzw. assoziativem Zugriff dar. Das ObjectStore-Ergebnis für die erste Zugriffsart ist sehr gut, das für den beschreibenden Zugriff hingegen mäßig. Die Tabelle 4.2 mit lediglich 12 Testfällen behandelt die Resultate des Hinzufügens und Löschens von Objekten. Die ObjectStore-Werte sind in diesem Bereich schlecht.

	Ex	On	Ob	Vr	OS
# 1.	46	10	-	22	24
# 2.	25	27	8	24	18
# 3.	17	29	24	6	26
# 4.	11	18	41	9	23
# 5.	3	18	29	41	11
gew.	206	313	397	329	285
Rang	**1**	**3**	**5**	**4**	**2**

Tabelle 4.1: Gesamte Testfälle

	Ex	On	Ob	Vr	OS
# 1.	6	-	-	6	-
# 2.	6	1	5	-	-
# 3.	-	5	4	1	2
# 4.	-	2	1	-	9
# 5.	-	4	2	5	1
gew.	18	45	36	34	47
Rang	**1**	**4**	**3**	**2**	**5**

Tabelle 4.2: Testfälle zum Einfügen und Löschen

[21] Zur Begründung der Nichtteilnahme am OO7-Benchmark siehe [CaDN93] und [ObOO93b].

	Ex	On	Ob	Vr	OS
# 1.	19	10	-	18	14
# 2.	17	21	2	10	11
# 3.	12	25	6	2	16
# 4.	11	05	30	1	14
# 5.	2	-	23	30	6
gew.	143	147	257	198	170
Rang	**1**	**2**	**5**	**4**	**3**

Tabelle 4.3: Testfälle zur kleinen Datenbank

	Ex	On	Ob	Vr	OS
# 1.	27	-	-	4	10
# 2.	8	6	6	14	7
# 3.	5	4	18	4	10
# 4.	-	13	11	8	9
# 5.	1	18	6	11	5
gew.	63	166	140	131	115
Rang	**1**	**5**	**4**	**3**	**2**

Tabelle 4.4: Testfälle zur mittleren Datenbank

	Ex	On	Ob	Vr	OS
# 1.	17	3	-	7	24
# 2.	12	13	3	6	17
# 3.	15	16	15	1	4
# 4.	6	12	25	7	1
# 5.	1	7	8	30	5
gew.	115	160	191	200	99
Rang	**2**	**3**	**4**	**5**	**1**

Tabelle 4.5: Testfälle mit navigierendem Zugriff

	Ex	On	Ob	Vr	OS
# 1.	23	7	-	9	-
# 2.	7	13	-	18	1
# 3.	2	8	5	4	20
# 4.	5	4	15	2	13
# 5.	2	7	19	6	5
gew.	73	108	170	95	139
Rang	**1**	**3**	**5**	**2**	**4**

Tabelle 4.6: Testfälle mit assoziativem Zugriff

4.10 Sicherheit

ObjectStore unterstützt Datenschutz über die Autorisierungsmechanismen des Betriebssystems. Vergleichbar mit dem Unix Verzeichnis- und Dateischutz werden ObjectStore Verzeichnisse und Datenbanken geschützt. Mit Hilfe des *Directory Managers* können diese erstellt und mit unterschiedlichen Zugriffsrechten (*user, group, other*) ausgestattet werden. Diese Zugriffsrechte können durch den Ersteller der Dateien verändert werden. Die kleinste schützbare Einheit ist demnach eine einzelne Datenbank. Diese sehr grobe Granularität ist in einer kooperierenden Arbeitsumgebung unangemessen. Vielmehr sollten Autorisierungsmechanismen auf Objekte, zumindest aber auf Bereiche von Datenbanken vorhanden sein.

Ein weiteres Sicherheitsproblem ergibt sich aus der bereits angesprochenen Trennung von Strukturdaten und Verhalten von Objekten. Die Schemavalidation, also die Überprüfung der Übereinstimmung des Applikationsschemas mit dem Datenbankschema ist der zentrale Sicherungsmechanismus in ObjectStore. Die Intention der Schemavalidation ist jedoch die Struktursicherung der Objekte und nicht die Verhaltenssicherung. Entsprechend liegt für die Schemavalidation ausschließlich bei internen Strukturänderungen eine Schemaänderung vor (s. [ObDM93, S. 51]).

Die Datenschutzmaßnahmen von ObjectStore sind keineswegs als ausreichend einzuschätzen.

4.11 Sonstiges

Mit der Untersuchung weiterer Funktionalitäten wird die Bewertung des Laufzeitsystems abgerundet. Zu diesen Funktionalitäten gehören das Metaobjekt-Protokoll, der „Browser“ und die Datenreorganisation. Als weiteres Konzept wird der Ansatz zu einem offenen Produkt beschrieben.

Das Metaobjekt-Protokoll

Das Metaobjekt-Protokoll (kurz MOP) von ObjectStore dient dazu, einen Zugriff auf die Schemainformationen zur Laufzeit zu erhalten, und zwar sowohl lesend als auch schreibend. Dadurch kann eine Applikation, wie z.B. ein Browser, Datenbankinhalte anzeigen. Mit Hilfe des MOP ist es möglich, dynamisch Klassen zu erstellen oder zu verändern, also eine Schemaevolution zur Laufzeit durchzuführen. Da sich diese Fähigkeit jedoch auf Strukturänderungen beschränkt, ist ihre intensive Verwendung nicht zu empfehlen. Wird beispielsweise der Wertebereich eines als `private` definierten Attributes geändert, gibt es erhebliche Probleme mit den Zugriffsfunktionen für dieses Attribut, da diese nicht zur Laufzeit geändert werden können. Die Datenkapselung wird durch das MOP vollständig ausgeschaltet. Der schreibende Zugriff erlaubt auch eine Manipulation des Inhalts von Datenelementen, ohne die ihnen zugedachten Zugriffsfunktionen zu verwenden.[22] Das MOP wurde entwickelt, um den Herstellern von Zusatzprodukten für ObjectStore Schemainformationen über Datenbanken zur Verfügung zu stellen (vgl. [ObTe94, S. 42]). Sämtliche Metadaten werden in die entsprechenden Datenbanken als Instanzen von Metaklassen (vgl. [ObDM93, S. 241 ff.]) gespeichert. Die Nutzung des MOP ist für den ungeübten Anwender umständlich und wird schnell komplex.

Die Bereitstellung des MOP ist bezüglich der Intention der Informationsbereitstellung positiv zu bewerten, im Hinblick auf die geschilderten Probleme jedoch negativ einzuschätzen.

Der Browser

Mit dem `osbrowser` von ObjectStore können Datenbankinhalte visualisiert werden. Komplexe Objekte werden durch den navigierenden Zugriff auf Komponentenobjekte dargestellt. Der Aufbau des Browsers zeichnet sich durch seine unkomfortable Bedienungsoberfläche aus. Das Hauptfenster dient dem Öffnen der Datenbanken. Einzelne Datenbanken können anschließend in den sogenannten *stack windows* angezeigt werden. Hier bietet sich die Möglichkeit, das Datenbankschema oder den Inhalt der Datenbank anzuschauen. Das Datenbankschema besteht aus einer Liste der in der Datenbank vorhandenen Klassen. Es ist eine Kopie des automatisch durch den Schemagenerator erstellten Applikationsschemas. Auf den Datenbankinhalt kann nur über die Einstiegspunkte zugegriffen werden. Die Darstellungsweise ist ausschließlich auf Textbasis. Der Browser kann bestenfalls als „Datenbankdebugger“ bezeichnet werden, denn die angebotene Anfragemöglichkeit kann nur auf Attribute, die aus Standarddatentypen bestehen, angewendet werden und ist somit für Ad-hoc-Anfragen unbrauchbar. Der `osbrowser` ist unzulänglich und fehlerhaft.

[22] Für die Zugriffsfunktionen des MOP siehe [ObDM93, S. 286].

Datenreorganisation

ObjectStore bietet die Möglichkeit zur Datenreorganisation, d.h. Daten werden in den Datenbanksegmenten so umorganisiert, daß keine interne Fragmentierung vorliegt (s. [ObDM93, S. 365]). Eine *„garbage collection“*, also das Entfernen von Daten, auf die nicht mehr zugegriffen werden kann, wird dadurch nicht realisiert. In zukünftigen Versionen wird es möglich sein zu kontrollieren, wohin einzelne Objekte verlagert werden sollen.

Die Handhabung der Datenreorganisation ist einfach und kann sowohl von Applikationen als auch durch ein Administrationswerkzeug (`oscompact`) verwendet werden.

Ansatz zu einem offenen Produkt

Die Systemarchitektur von ObjectStore ist so konzipiert, daß andere Hersteller die Möglichkeit erhalten, eigene Produkte als Erweiterung von ObjectStore anzubieten. Die wesentlichen Bereiche sind grafische Benutzerschnittstellen, Analyse- und Designwerkzeuge sowie Klassenbibliotheken und Werkzeuge zur Fehlerbeseitigung. Eine wichtige Schnittstelle bildet das MOP (vgl. [ObTe94, S. 41 f.]). Die in zukünftigen Versionen von ObjectStore angebotenen Zusatzprodukte, wie z.B. „ObjectStore/DBconnect“, fallen ebenfalls in die Rubrik der Unterstützungskonzepte für ein offenes Produkt. Negativ ist bei diesem Konzept anzumerken, daß ein Anwender, der hofft, mit ObjectStore ein vollständiges Datenbankmanagementsystem erworben zu haben, nur mit dem Notwendigsten versorgt wird. Ein Analyse- und Designwerkzeug sowie eine Ad-hoc-Anfragemöglichkeit, die über die Darstellungsmöglichkeit des `osbrowser` hinausgeht, wäre an dieser Stelle sehr wünschenswert. Desweiteren ist eine Unterstützung von allgemein gebräuchlichen Typen, wie z.B. `date` und `string`, Grundvoraussetzung für einen effizienten Einsatz von ObjectStore.

5 Bewertungstabellen

Datenmodell		
Typen, Klassen, Metaklassen		
	atomare Typen	+
	abgeleitete Typen	++
	abstrakte Klassen	++
	generische Klassen	++
	Metaklassen Klassen	+
Kapselung		
	Methodenspeicherung	--
	abstrakte Datentypen	++
	Modularität	++
	Granularität der Kapselung	++
Objektkonstruktoren		
	Collections	++
	Tupelkonstruktor	++
	Konstruktor-operationen	++
Beziehungen		
	Klasse-Unterklasse-Beziehung	++
	1:1–Beziehung	++
	1:n–Beziehung	++
	n:m–Beziehung	++
	referenzielle Integrität	++
	Beziehungsoperationen	-
Objektidentität		
	Art der Identität	physikalisch
	automatische Generierung	+
	Lebensdauer unverändert	+
	Identität	++
	Gleichheit	--
	flaches Kopieren	++
	tiefes Kopieren	--
Vererbung		
	Einfachvererbung	++
	Mehrfachvererbung	++
Polymorphismus		
	Methodenredefinition	++
	Methodenverfeinerung	+
	dynamisches Binden	++
	statisches Binden	++
	Uberladen von Funktionen	++
	Überladen von Operatoren	++

	Erweiterbarkeit		
		benutzerdefi. Typen un Operationen	++
		Intergration von Typen	o
		generische Anfragen	-
		generische Update–Operationen	+
Datenbanksystem			
	Dauerhaftigkeit		
		orthogonal	+
		implizit	++
		objektabhängige Persistenz	++
		Persistenz durch Erreichbarkeit	-
		abspeichern gesamter Objekte	--
	Anfragesprache		
		anwendungsunabhängig	+
		deskriptive Sprache	--
		generische Operationen	+
		orthogonal	+
		Optimierbarkeit & Effizienz	--
		Vollständigkeit & Abgeschlossenheit	o
		Angemessenheit	--
		Ad-hoc-Anfragen	--
	Transaktionsmanagement		
		kurze Transaktionen	+
		zurücksetzen von Daten	o
		Transparenz von Sperren	++
		Granularität von Sperren	+
		globale Transaktionen	++
	Integritätsmechanismen		
		Schlüssel	--
		Kardinalitäten von Beziehungen	++
		Kardinalitäten von Mengen	--
		Trigger	--
	Schemaevolution		
		Strukturänderungen	+
		Verhaltensänderungen	--
		Klassenhierarchieänderungen	++
		Klassenänderungen	+
		interaktive Änderungsmöglichkeit	-
		automatische Anderungen	-
		Änderungen an Instanzen	+
	Versionsmanagement		
		Konfigurierung	++
		Verwaltung	++
		Verschmelzung	++
		Einfrierung	++
		Identität	++
		weitere Funktionen	++
		Speicherungszeitpunkt	--
		Realisierungszeitpunkt	o
		Gültigkeitszeitraum	--

		transiente Version	++
		Arbeitsversion	++
		alternative Versionen	++
		private Arbeitsbereiche	++
		gemeinsame Arbeitsbereiche	++
		Arbeitbereich–Hierarchie	++
	Datenschutz		
		Granularität	-
		Zugriffsschutz	-
Laufzeitsystem und Administration			
	Sekundärspeicherverwaltung & Zugriffsmethoden		
		Objektidentität	+
		Speicherungstechnik	++
		Datenpufferung	++
		Indizierungstechnik	+
		Clusterung	-
	Wiederanlauf & Datensicherheit		
		Änderungsprotokoll	++
		Online Sicherungskopien	++
		Transparenz	++
		globales Recovery	++
	Effizienz		
		navigierender Zugriff	++
		assoziativer Zugriff	-
		Datenbankgröße	+
	Verteilung		
		Autonomie	+
		Transparenz	++
		Aufteilung	-
		Heterogene Verteilung	+
	Zusätzliche Bewertungen		
		MOP Idee	+
		MOP Anwendung	-
		Browser	--
		Datenreoganisation	+
		offenes Produkt	o

Tabelle 8: Übersicht über ObjectStore Fähigkeiten

Die in der Tabelle 8 verwendeten Symbole sind folgendermaßen zu interpretieren:

sehr gut	++
gut	+
befriedigend	o
ausreichend	-
mangelhaft	--

Literatur

[ABD+89] Atkinson, M.; Bancilhon, F.; DeWitt D.; Dittrich, K.; Maier, D.; Zdonik, S: *The Object-Oriented Database System Manifesto*; 1989; in: 'Building an Object-Oriented Database System. The Story of O_2'; Francois Bancilhon et al. (Hrsg.); Morgan Kaufmann Publishers, Inc.; San Mateo, CA; 1992

[AbuA94] Abu Alwan, I.; *Evaluierung des Datenmodells des objektorientierten Datenbanksystems ObjectStore*; Diplomarbeit, unveröffentlichtes Manuskript; Münster; 1994

[Atwo90] Atwood, T.: *Two Approaches to Adding Persistence to C++*; in: 'Implementing persistent Object Bases. Principles and Practice. The Fourth International Workshop on Persistent Object Stystems. September 23-27, 1990'; Dearle, A. et al (Hrsg.); Morgan Kaufmann Publishers, Inc.; San Mateo, CA; 1991

[AWSL91] Ahmed, S.; Wong, A.; Sriram D.; Logcher, R.: *A Comparison of Object-Oriented Database Management Systems for Engineering Applications*; Research Report R91-12, Massachusetts Institute of Technologie May 1991

[AWSL92] Ahmed, S.; Wong, A.; Sriram D.; Logcher, R.: *Object-oriented database management systems for engineering: A comparison*; in: Journal of Object-Oriented Programming, 5(6) S. 27-44; 1992

[Bowm94] Bowman, C.F.: *Why We Need Object-Oriented Systems*; in: Database Programming & Design S. 27-30; Feb 1994

[BSG+88] Banerjee, J.; Schou, H.; Garza, F.; Kim, W.; Woelk, D.; Ballou, N.; Kim H.: *Datamodel Issuses for Object Oriented Applications*; in: 'Reading in Databasesystems'; Stonebraker, M. (Hrsg.); Morgan Kaufmann Publishers, Inc.; San Mateo, CA; 1988

[CaDN93] Carey, M.; DeWitt, D.J.; Naughton, J.F.: *README*; verfügbar über FTP: ftp.cs.wisc.edu, Verzeichnis: /oo7, Datei README; Apr 1993

[CaDN94a] Carey, M.; DeWitt, D.J.; Naughton, J.F.: *README.jan10.1994*; verfügbar über FTP: ftp.cs.wisc.edu, Verzeichnis: /oo7, Datei README.jan10.1994; Jan 1994

[CaDN94b] Carey, M.; DeWitt, D.J.; Naughton, J.F.: *interpreting.oo7*; verfügbar über FTP: ftp.cs.wisc.edu, Verzeichnis: /oo7, Datei interpreting.oo7; Jan 1994

[CaDN94c] Carey, M.; DeWitt, D.J.; Naughton, J.F.: *The OO7 Benchmark*; Technical Report, University of Wisconsin-Madison; verfügbar über FTP: ftp.cs.wisc.edu, Verzeichnis /oo7, Datei: techreport.ps; Jan 1994

[Catt94] Cattell, R.G.G. (Hrsg.): *The Object Database Standard: ODMG-93*; Morgan Kaufmann Publishers, San Mateo, CA 94403; 1994

[Clau93] Claussen, U.: *Objektorientiertes Programmieren. Mit Beispielen und Übungen in C++*; Springer-Verlag, Berlin et al.; 1993

[Copl92] Coplien, J. O.: *Advanced C++ Programming Styles and Idioms*; Addison-Wesley Pub. Comp.; 1992

[Fisc93] Fischbach, R.: *Jenseits von Hollerith*; in: iX, Multiuser Multitasking Magazine S. 144-154; June 1993

[HaSV94] Hauten, K.; Schmitz-Lenders, J.; Vaillant, S.: *Stauraum*; in: iX, Multiuser Multitasking Magazine S. 102-109; May 1994

[Heue92] Heuer, A.: *Objektorientierte Datenbanken Konzepte, Modelle, Systeme*; Addison-Wesley; 1992

[Hugh91] Hughes, J. G.: *Object-Oriented Databases*; Prentice Hall International (UK) Ltd.; Hertfordshire; 1991

[HuPC93] Hurson, A.R.; Pakzad, S.H.; Cheng, J.: *Object-Oriented Database Management Systems: Evolution and Performance Issues*; in: Computer IEEE S. 48-60; Feb 1993

[JeRe92] Jell, T.; von Reeken, A.: *Objektorientiertes Programmieren mit C++. Eine Einführung mit vielen Beispielen, Übungsaufgaben und Musterlösungen.* 2. Auflage. Carl Hanser, 1992

[KhAb90] Khoshafian, S.; Abnous, R.: *Object orientation: concepts, languages, databases, user, interfaces*; John Wiley & Sons, Inc.; 1990

[Kim90] Kim, W.: *Introduction to Object-Oriented Databases*; The MIT Press Cambridge; 1990

[Kim93] Kim, W.: *Object-Oriented Database Systems: Promises, Reality, and Future*; in: 'Proceedings of the 19th VLDB Conference'; Agrawal, R.; Baker, S.; Bell, D. (Hrsg.); Dublin, Ireland; S. 676-687; 1993

[Kim94] Kim, W.: *Observations on the ODMG-93 Proposal for an Object-Oriented Database Language*; in: SIGMOD RECORD, 23(1) S. 4-9; March 1994

[Kotu92] Kotulla, A.; *Das Vieweg Buch zu Borland C++ 3.0*; Vieweg, 1992

[KRRV94] Kappel, G.; Rausch-Schott, S.; Retschitzegger, W.; Vieweg S.: *TriGS: Making a passive object-oriented database system active*; in: Journal of Object-Oriented Programming, 7(4) S. 44-51+63; 1994

[Lipp91] Lippman, S.B.: *C++. Einführung und Leitfaden*; 2. erweiterte Auflage. Addison-Wesley, 1991

[LLOW91] Lamb, C.; Landis, G.; Orenstein, J.; Weinreb, D.: *The ObjectStore Database System*; in: Communications of the ACM, 34(10) S. 50-63; 1991

[Loom94] Loomis, M.E.S.: *ODBMS myths and ralities*; in: Journal of Object-Oriented Programming, 7(4) S. 77-80; July 1994

[ObDM93] Object Design: *User Guide: DML, Release 3.0 for UNIX Systems*; Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; Dec. 1993

[ObLI93] Object Design: *User Guide: LI, Release 3.0 for UNIX Systems*; Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; Dec. 1993

[ObOO93a] Object Design: *Understanding the OO7 Research Project*; verfügbar über FTP: ftp.odi.com, Verzeichnis: /pub/benchmark/oo7, Datei: result.ps; Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; Apr. 1993

[ObOO93b] Object Design: *README*; verfügbar über FTP: ftp.odi.com, Verzeichnis: /pub/benchmark/oo7, Datei: README; Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; Apr. 1993

[ObPG93] Object Design: *Platform Guide, Release 3.0 for UNIX Systems*; Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; Dec. 1993

[ObRM93] Object Design: *Reference Manual, Release 3.0 for UNIX Systems*; Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; Dec. 1993

[ObTe94] Object Design: *ObjectStore Technical Overview* Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; March 1994

[ObTU93] ObjectStore: *Tutorial, Release 3.0 for UNIX Systems*; Object Design, Inc.; 25 Burlington Mall Road, Burlington, MA 01803; Dec. 1993

[OHMS92] Orenstein, J.; Haradhvala, S.; Margulies, B.; Sakahara, D.: *Query Processing in the ObjectStore Database System*; in: ACM SIGMOD S. 403-412; June 1992

[Rao94] Rao, B. R.: *Object-Oriented Databases: technology, applications, and products*; McGraw-Hill, Inc.; New York et al.; 1994

[Reut87] Reuter, A.: *Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen*; in: 'Datenbank-Handbuch'; Lockemann, P.C.; Schmidt, J.W. (Hrsg.); Springer-Verlag, Berlin; S. 337-479; 1991

[Solo92] Soloviev, V.: *An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, ObjectStore, and* O_2; in: SIGMOD RECORD, 21(1) S. 93-104; 1992

[SRL+90] Stonebraker, M.; Rowe, L. A.; Lindsay, B.; Gray, J.; Carey, M.; Brodie, M.; Bernstein, P.; Beech, B.: *Third-Generation Database System Manifesto*; in: Sigmod Record, 19(3) S. 31-44; Sep. 1990

[Stro92] Stroustrup, B.: *Die C++ Programmiersprache*; 2. Auflage; Addison-Wesley; 1992

[Thur89] M.B. Thuraisingham: *Mandatory Security in Object-Oriented Database Systems*; in: ACM SIGPLAN 24(11) S. 203-210; Nov. 1989

[Schl94] Schlagheck, B.; *Evaluierung des Laufzeitsystems des objektorientierten Datenbanksystems ObjectStore*; Diplomarbeit, unveröffentliches Manuskript; Münster; 1994

[Schn91] Schneider, H.-J. (Hrsg.): *Lexikon der Informatik und Datenverarbeitung*; 3. Auflage; Oldenbourg; 1991

[Unla92] Unland, R.: *Vorlesung: Verteilte Systeme*; (unveröffentlichtes Manuskript); Institut für Wirtschaftsinformatik, Lehrstuhl für Praktische Informatik, Münster; WS 1992/93

[Unla95] Unland, R.: *Objektorientierte Datenbanken: Konzepte und Modelle*; Thompson Pub. 1995

[UnSc92] Unland, R.; Schlageter, G.: *Object-Oriented Database Systems: State of the Art and Research Problems*; in: 'Expert Database Systems'; Jeffrey, K. (Hrsg.); Academic Press Ltd.; S. 117-222; 1992

[Varm93] Varma, S.: *Objects and Databases: Where Are We Now?* in: Database Programming & Design S. 60-64; May 1993