Universität Kaiserslautern
Fachbereich Elektrotechnik
Lehrstuhl für Digitale Systeme
Prof. Dr.–Ing. S. Wendt

Diplomarbeit

Bereitstellung einer Laborumgebung und Untersuchung objektorientierter Datenbanktechnologien

Bernhard Gröne
Juli 1996

Betreuer: Prof. Dr.–Ing. S. Wendt

Bearbeiter: Bernhard Gröne
Friedrichstraße 25
67069 Ludwigshafen–Oppau

Erklärung

Hiermit erkläre ich, die vorliegende Diplomarbeit unter Verwendung der angegebenen Quellen und ohne fremde Hilfe angefertigt zu haben.

Walldorf, den 29.Juli 1996

Bernhard Gröne

Vorwort

Objektorientierte Technologien werden in zunehmenden Maß bei der Entwicklung informationsverarbeitender Systeme eingesetzt. Beim Einsatz von Datenbanken zur permanenten Speicherung von Daten stellt man schnell fest, daß die heute weit verbreiteten relationalen Datenbanksysteme für objektorientierte Anwendungen nur bedingt geeignet sind. Objektdatenbanksysteme bieten diesen Anwendungen dagegen die Möglichkeit, Objekte permanent zu speichern und anderen Anwendungen zugänglich zu machen.

In dieser Arbeit werden Merkmale und Besonderheiten von Objektdatenbanksystemen allgemein und anhand kommerzieller Produkte vorgestellt. Da Client–Server–Systeme hohe Anforderungen an Datenbanksysteme stellen, erfolgt die Untersuchung der Produkte unter Berücksichtigung dieser Anforderungen.

Kapitel 1 beschreibt das technische Umfeld der Untersuchungen, das Software–Technologie–Labor. Eine kurze Vorstellung der Begriffswelt und der Gründe, überhaupt eine Objektdatenbank einzusetzen, ist das Ziel von Kapitel 2. Darin wird außerdem kurz beschrieben, wie relationale Datenbanken für objektorientierte Anwendungen genutzt werden können. In Kapitel 3 werden die Merkmale und Besonderheiten von Objektdatenbanksystemen vorgestellt, um eine Einordnung der Produkte zu ermöglichen. Die Anforderungen, die der Einsatz in Client–Server–Systemen mit sich bringt, werden in Kapitel 4 gesammelt und erläutert. Die Untersuchung der einzelnen Produkte erfolgt aufgrund der gesammelten Anforderungen im Kapitel 5, wobei die Informationen zum Großteil den Handbüchern zu den Produkten entnommen wurden. In Kapitel 6 werden die Produkte schließlich verglichen und bewertet.

Danksagung

An dieser Stelle möchte ich den Firmen danken, die uns ihre Produkte zur Verfügung gestellt und bei Fragen schnell und kompetent geantwortet haben. Das waren im einzelnen:

ObjectStore:	Object Design Software GmbH, Wiesbaden
Versant:	Versant Object Technology GmbH Europe, München
Poet:	Poet Software GmbH, Hamburg
GemStone,	
Visualworks Smalltalk:	Georg Heeg Objektorientierte Systeme, Dortmund

Herrn Professor Dr.–Ing. Siegfried Wendt danke ich sehr für die interessanten Gespräche, aus denen ich viele Anregungen mitgenommen habe. Meinen Kollegen Johannes Otto und Eberhard Iglhaut danke ich für zahlreiche Diskussionen, die zur Klärung vieler Sachverhalte und zum gemeinsamen Entwurf eines Beispielsystems führten.

Schließlich danke ich der SAP AG, die die Einrichtung des Labors mit allen Mitteln unterstützt hat und uns ein angenehmes Arbeitsumfeld bietet.

Bernhard Gröne

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Das Software–Technologie–Labor

In der Industrie ist es nicht ungewöhnlich, neue Technologien und Produkte anderer Hersteller unter Laborbedingungen zu untersuchen und sie zu bewerten (Evaluation). Das Ziel der Laborarbeit ist nicht die Entwicklung marktfähiger Produkte, sondern das Sammeln von Erfahrungen anhand von Beispielen und Prototypen. Die Ergebnisse der Laborarbeit dienen der Weiterbildung und der Entscheidungsfindung.

In der Softwarebranche ist eine ausgesprochene Laborarbeit selten zu finden. Testen Entwickler Produkte in ihrem Bereich, bleiben die Erkenntnisse oft auf diesen Bereich beschränkt. Die Dokumentation der Ergebnisse ist selten einfach und zentral zu finden.

Professor Dr. S. Wendt beschloß 1995 die Einrichtung eines Software–Technologie–Labors innerhalb der Firma SAP, Walldorf. Drei Diplomanden, Bernhard Gröne, Johannes Otto und Eberhard Iglhaut, wurden mit der Einrichtung des Labors und ersten Untersuchungen betraut.

1.1 Zweck des Labors

Das Software–Technologie-Labor ist eine Experimentierumgebung mit Laborcharakter für Software–Produkte. An Technologien werden dort insbesondere objektorientierte Entwicklungswerkzeuge und Objektdatenbanksysteme untersucht. Ferner wird ein Software–Bus–System eingesetzt.

Die Infrastruktur des Labors ermöglicht Experimentiersysteme in einer 3–Ebenen–Client–Server–Architektur. Das Beispielsystem, das innerhalb der drei Diplomarbeiten modelliert und teilweise realisiert wurde, ist ein einfaches Videothekenverwaltungssystem.

1.2 Ausstattung

Es folgt ein kurzer Überblick über die Ausstattung des Labors. Details sind dem Anhang zu entnehmen.

Ausstattung an Geräten

- 4 PCs, die als Entwicklungsplattform, Applikationsserver und Frontend eingesetzt werden können
- 1 HP 9000 Workstation, die als Datenbank-Server eingesetzt wird
- Laserdrucker, DAT–Laufwerk

Software

- Entwicklungswerkzeuge
  - ISE Eiffel
  - VISUALWORKS Smalltalk von Parcplace–Digitalk
  - VISUAL C++ von Microsoft
  - JAVA DEVELOPMENT KIT von Sun
- Datenbanken
  - OBJECTSTORE
  - VERSANT
  - POET
  - GEMSTONE
  - ORACLE (relationale Datenbank)
- Software Bus
  - RENDEZVOUS SOFTWARE BUS von TIBCO (vormals Teknekron)

Kapitel 2

Objektorientierung und Datenbanken

2.1 Objektorientierung

Das Konzept der Objektorientierung resultiert aus der Erkenntnis, daß es bei der Verarbeitung von Daten eine starke Kopplung zwischen Datentypen und darauf arbeitenden Routinen gibt. Die Verbindung eines Datums mit den dazugehörigen Routinen ergibt das Objekt, das als aktive Komponente im System auftritt. Objektorientierung wird in zahlreichen Publikationen vorgestellt, hier sei vor allem auf [Meyer 90] verwiesen. Es folgt eine Auswahl von Begriffen:

Attribute: Datenteil eines Objekts

Methoden: Routinen eines Objekts

Die Attribute und Methoden eines Objekts werden in seiner **Klassenbeschreibung** definiert.

Schnittstelle eines Objekts: Nach außen, das heißt für andere Objekte sichtbare Routinen zum Datenzugriff und zur Datenverarbeitung.

Objekt–ID: Jedem Objekt ist eineindeutig ein Identifikator, die Objekt–ID, zugeordnet, welcher zusammen mit dem Objekt erzeugt wird und sich über die Lebensdauer des Objekts nicht ändert. Ein Objekt kann dadurch unabhängig von der Belegung seiner Attribute identifiziert werden. Der direkte Verweis auf ein Objekt entspricht der Kenntnis von dessen Objekt-ID.

Kunde: Eine Klasse ist Kunde einer anderen, wenn in ihren Methoden von Diensten der anderen Gebrauch gemacht wird. Die Dienste sind in der Schnittstelle definiert. [Meyer 90, S.65].

2.2 Datenbanken

2.2.1 Das Datenbanksystem

Ein Datenbanksystem hat im wesentlichen drei Aufgaben zu erfüllen:

1. Permanente Speicherung von Daten
2. Zugriff auf die gespeicherten Daten unabhängig von der internen Organisation des Systems
3. Wahrung der Konsistenz des Datenbestands und Fehlerbehandlung

Zur Erfüllung der ersten Aufgabe benötigt man Massenspeicher, die Daten auf unbegrenzte Zeit bewahren können. Die zweite erfordert eine logische Sicht auf die gespeicherten Daten, die unabhängig von den tatsächlich eingesetzten Speichermedien ist. Die Sicht und die Art des Zugriffs bestimmen den Typ der Datenbank. Der dritte Punkt betrifft den Schutz der Daten im Zusammenhang mit konkurrierenden Zugriffen und bei auftretenden Fehlern. Es soll jederzeit möglich sein, die Datenbank nach Auftreten eines Fehlers in einen konsistenten Zustand zu bringen.

In Abbildung 2.1 ist ein allgemeines Modell eines Datenbanksystems dargestellt. Eine Anwendung hat einerseits Daten im lokalen Speicher und kann andererseits über den Datenbankverwalter auf Daten der Datenbank zugreifen. Der Speicher, der die Daten hält, heißt Datenbank. Auf die Daten kann nur über eine logische Sicht zugegriffen werden. Die Umsetzung in konkrete Ortsangaben, wie etwa Blöcke auf einer Festplatte, erfolgt durch den Datenbankverwalter und bleibt der Anwendung verborgen. Die Datenbank kann auf mehrere Massenspeicher verteilt sein. Ist sie auf mehrere Rechner verteilt, spricht man von einer verteilten Datenbank. Ein Datenbankverwalter kann mehrere Datenbanken verwalten.

Weiterhin soll mehreren Anwendungen Zugriff auf die gleichen Daten gewährt werden. Die Anwendungen kommunizieren über eine vom Datenbanksystem festgelegte Schnittstelle mit dem Datenbankverwalter. Dieser muß konkurrierende Zugriffe erkennen und behandeln können.

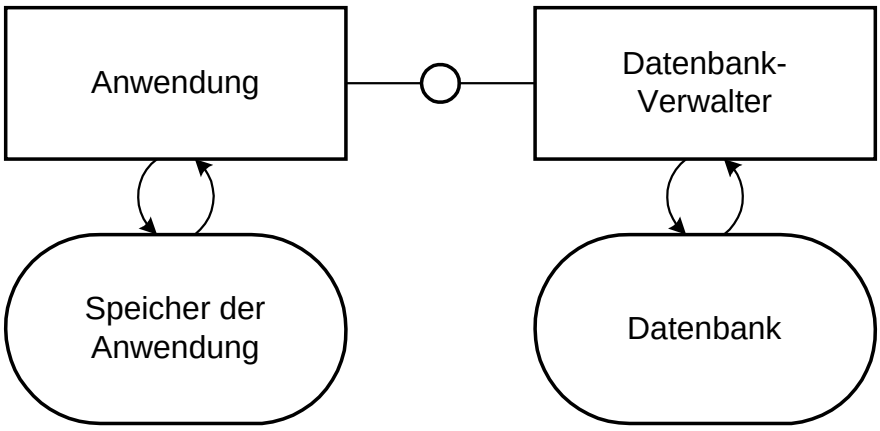

Abbildung 2.1: Allgemeines Modell eines Datenbanksystems

2.2.2 Relationale Datenbanken

Relationale Datenbanken bieten eine logische Sicht auf die Daten, der das relationale Modell zugrundeliegt. Beziehungen zwischen Entitäten werden durch Tabellen ausgedrückt. Datensätze werden repräsentiert durch Zeilen in Tabellen. Suchanfragen (Queries) werden in einer deklarativen Sprache, SQL, formuliert.

Zu relationalen Datenbanken existiert eine große Anzahl an Publikationen. Aus diesem Grund folgt nun lediglich eine Auswahl an Begriffen:

Tabellen: Eine Tabelle enthält Datensätze eines Entitätstyps. Eine Zeile einer Tabelle repräsentiert einen Datensatz. Die Spalten entsprechen den Feldern eines Datensatzes. Eine Spalte kann nur Daten eines bestimmten Basistyps aufnehmen und hat einen Namen. Die Namen einer Tabelle und all ihrer Spalten müssen dem Datenbanksystem bekannt sein, um auf die Daten zugreifen zu können. Neue Datentypen für Felder können normalerweise nicht hinzugefügt werden. Die Typen der Spalten und deren Namen werden bei Definition der Tabelle festgelegt und sind Teil des Datenbank–Schemas. Die Spalten einer Tabelle können zur Laufzeit nicht verändert werden.

Primär- und Fremdschlüssel: Die Lage einer Zeile in einer Tabelle ist ohne Bedeutung. Schlüssel dienen zur Identifikation von Datensätzen. Sie sind Bestandteil des Datensatzes und können aus einem oder mehreren Feldern bestehen. Der Primärschlüssel ist ein eindeutiger Identifikator eines Datensatzes in einer Tabelle und darf in dieser daher nur einmal vorkommen. Um von einem Datensatz einer Tabelle auf einen anderen in einer anderen Tabelle zu verweisen, muß man dessen Primärschlüssel als Identifikator, welcher Fremdschlüssel genannt wird, in einem oder mehreren Feldern ablegen.

Datenmanipulation: Über INSERT, DELETE und UPDATE werden Datensätze in Tabellen eingefügt, gelöscht und geändert. Ein SELECT–Ausdruck beschreibt eine Menge von Datensätzen, deren Feldbelegungen die angegebenen Bedingungen erfüllen. Es kann sich dabei um umfangreiche Abfragen handeln, die mehrere Tabellen betreffen.

Embedded SQL: SQL ist eine interpretierte deklarative Sprache. Um Abfragen in Programmen zu ermöglichen, die in anderen Programmiersprachen realisiert sind, werden SQL–Ausdrücke als Strings eingebettet oder durch einen speziellen Präprozessor vorverarbeitet.

Das Ergebnis einer SELECT–Abfrage ist eine Tabelle; mit den meisten Programmiersprachen ist dieser Datentyp nicht direkt verarbeitbar, so daß die Tabelle zeilenweise ausgelesen werden muß, wobei man sich Cursors bedient.

2.2.3 Einschränkungen relationaler Datenbanken

Die Stärken relationaler Datenbanken liegen im Durchsuchen großer Datenmengen. Die Datenbanksysteme bieten ausgefeilte Suchmechanismen und Optimierungsfilter für die SQL–Ausdrücke.

Die Anforderungen objektorientierter Systeme können sie nicht oder nur zum Teil erfüllen. Die Einschränkungen sind im Einzelnen:

Keine benutzerdefinierten Datentypen: Im Objektmodell wird das Typsystem durch benutzerdefinierte Klassen erweitert. Im relationalen Modell ist diese Möglichkeit nicht vorgesehen. Relationale Datenbanksysteme erlauben nur Basistypen und fest vorgegebene komplexe Typen wie Datum und Währung als Feldtyp.

Beschränkte Modellierung der Beziehungen zwischen Daten: Direkte Verweise sind im relationalen Modell nicht vorgesehen. Ein Verweis eines Datensatzes auf einen anderen erfolgt durch Angabe des Primärschlüssels in dessen Tabelle. Bei Löschen eines Datensatzes kann es zur Verletzung der referentiellen Integrität der Daten kommen, wenn nämlich noch ein Datensatz einen Verweis auf den gelöschten trägt.
1:m- und n:m–Beziehungen bereiten Schwierigkeiten: Da Tabellen keine Freilängenattribute zulassen (das würde eine variable Spaltenzahl oder Spaltenbreite bedeuten), muß für die Zuordnung eine eigene Tabelle angelegt werden. Noch schwieriger sind „ist ein“–Beziehungen der Vererbung, da hierarchische Strukturen nicht von relationalen Systemen unterstützt werden.

Keine Kapselung: Im Objektmodell wird auf private Daten eines Objekts nicht direkt zugegriffen, sondern indirekt über die Zugriffsmethoden, die über seine Schnittstelle öffentlich zugänglich sind. Dadurch wird auch verborgen, daß manche Daten unter Umständen erst berechnet oder von anderen Stellen geholt werden müssen. Diese Möglichkeit bietet das relationale Modell nicht.

Schlechte Einbettung von SQL in die Programmiersprache der Anwendung: SQL erlaubt durch seinen deklarativen Charakter eine sehr elegante Möglichkeit, nach Daten zu suchen, hat aber nur beschränkte Mittel zur Datenmanipulation. Aus diesem Grund muß SQL in Sprachen wie C eingebettet werden, in denen die eigentliche Verarbeitung der Daten stattfindet.
Die zur Verarbeitung der Daten eingesetzten Programmiersprachen haben kein Konzept für Tabellen, während SQL nichts mit den Datenstrukturen der Sprachen anfangen kann (Impedance Mismatch). Der Programmierer ist also verantwortlich für die Verbindung der beiden Modelle und die Umsetzung der Daten.

2.3 Objektorientierte Sprachen und relationale Datenbanken

Oft tritt der Fall auf, daß eine objektorientierte Anwendung auf eine relationale Datenbank zugreifen soll. Weiterhin soll sie auch eigene Daten auf der Datenbank ablegen können. Für diese Aufgabe gibt es zwei Lösungen:

1. Kapselung des Datenbanksystems als Objekt
2. Benutzung der relationalen Datenbank zur Speicherung von Objekten

2.3.1 Kapselung des Datenbanksystems als Objekt

Das komplette relationale Datenbanksystem existiert im objektorientierten System als ein einziges Objekt. Zugriffe auf dieses Objekt erfordern das Übertragen von SQL–Ausdrücken und die Interpretation der Ergebnistabellen. Das Wissen, wie der Zugriff auf eine relationale Datenbank erfolgt, muß daher in jeder Klasse vorhanden sein, deren Exemplare auf die Datenbank zugreifen müssen. Hier findet man die oben genannten Probleme der Einbettung (Impedance Mismatch).

2.3.2 Benutzung der relationalen Datenbank zur Speicherung von Objekten

Hier werden Objekte in der Datenbank abgelegt. Ein Zugriffsakteur kümmert sich um die Abbildung zwischen Objektmodell und Tabellen. Der Nachteil dieser Methode gegenüber der Kapselung in 2.3.1 ist, daß das Datenbank–Schema speziell für den Zugriffsakteur generiert werden muß, so daß andere nicht objektorientierte Anwendungen nicht sinnvoll darauf zugreifen können. Weiterhin kostet die Abbildung Rechenzeit.

Abbildung 2.2 zeigt ein solches System. Auf der linken Seite befindet sich die Anwendung mit ihrem Speicher. Hier ist angedeutet, daß sich im Speicher der Datenteil der Objekte befindet, während die Methoden fest im Anwendungsakteur enthalten sind. Auf der rechten Seite befindet sich das relationale Datenbanksystem. Daten sind in Tabellen abgelegt. Der Zugriff auf die Daten erfolgt über SQL–Ausdrücke.

Der Zugriffsakteur in der Mitte hat folgende Aufgaben (siehe auch [Hahn 95] und [Shekar 95]):

- Die Zuordnung von Tabellen zu Klassen
- Die Abbildung von Objekten auf Zeilen in Tabellen
- Die Erzeugung eines eindeutigen Primärschlüssels zur Abbildung der Objekt–ID, des sogenannten Surrogats
- Die Umsetzung von Zugriffen in SQL-Ausdrücke

Es gibt verschiedene Techniken für die Anbindung des Zugriffsakteurs an die Anwendung, die genauso bei Objektdatenbanken angewendet werden. Diese werden in Kapitel 3 betrachtet.

Gründe für den Einsatz eines solchen Systems sind:

- Ein bereits vorhandenes relationales Datenbanksystem soll auch für die Objektspeicherung verwendet werden, unter anderem, um den Aufwand zur Administration klein zu halten.
- Das relationale Datenbanksystem bietet beispielsweise höhere Skalierbarkeit bezüglich verteilter Datenbanken als vergleichbare Objektdatenbanksysteme.

Ein Produkt, welches einen Zugriffsakteur für die Anbindung an eine relationale Datenbank zur Verfügung stellt, ist PERSISTENCE von Persistence Software.

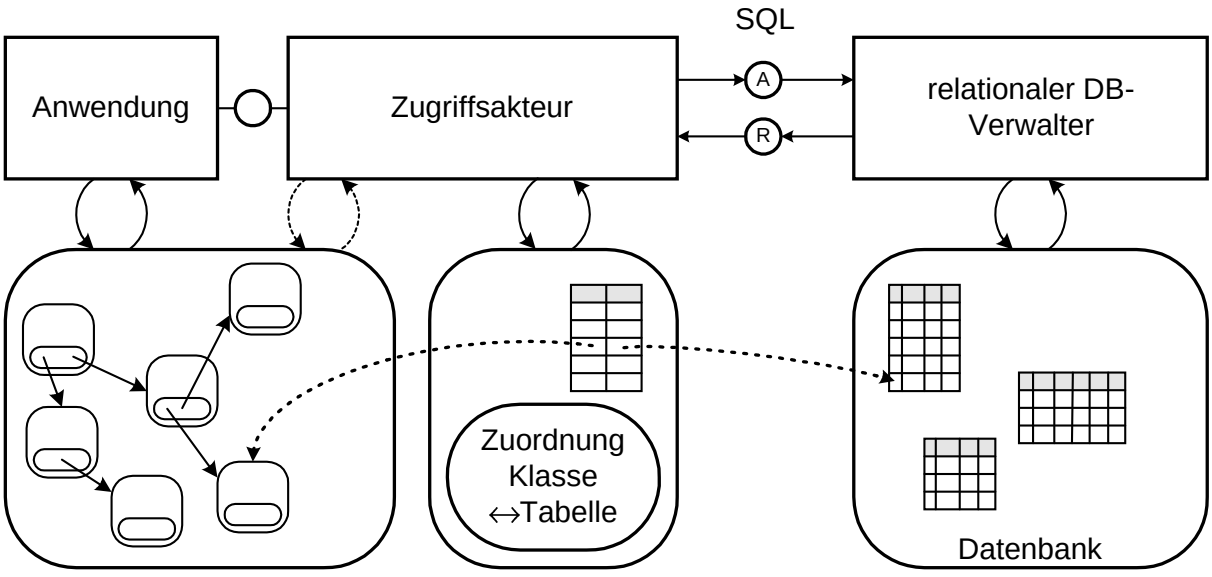

Abbildung 2.2: Objektorientierte Anwendung und relationale Datenbank

Kapitel 3

Objektdatenbanksysteme

Eine Objektdatenbank ist eine Datenbank zur permanenten Speicherung von Objekten. Der Anwendung wird die logische Sicht auf die Daten gemäß dem Objektmodell präsentiert.

Daten werden auf der Objektdatenbank in Form persistenter Objekte abgelegt. Die Eigenschaften persistenter Objekte und die Methoden zu ihrer Erzeugung werden in Abschnitt 3.1 vorgestellt. Bei Zugriff auf persistente Objekte muß unterschieden werden, wie sie erreicht werden und auf welche Art sie modifiziert werden können — siehe Abschnitt 3.2.

Objektdatenbanksysteme können aus Sicht einer Anwendung auf zwei Arten auftreten, die bei relationalen Datenbanken nicht möglich sind: als aktive Datenbank, auf der Methoden von Objekten ausgeführt werden können, und als persistenter Objektspeicher, bei dem der Zugriff auf persistente Objekte transparent möglich ist. Diese Sichtweisen werden in Abschnitt 3.3 vorgestellt.

Bei Entwicklung einer Anwendung, die auf eine Objektdatenbank zugreift, müssen einige Besonderheiten beachtet werden. In Abschnitt 3.4 wird gezeigt, wie im Fall der Sprache C++ die Übersetzung des Quelltextes um die Erzeugung des Datenbank–Schemas erweitert wird. Bei Änderungen am Datenbank–Schema müssen die bereits gespeicherten Daten an das neue Schema angepaßt werden. Dieses Thema wird in Abschnitt 3.5 behandelt.

3.1 Persistente Objekte

3.1.1 Wunscheigenschaften persistenter Objekte

Ein persistentes Objekt sollte drei Eigenschaften haben:

- Die Persistenz ist unabhängig von der Klasse: Von einer Klasse sollen sowohl transiente als auch persistente Exemplare erzeugt werden können; im Idealfall können Exemplare einer beliebigen Klasse persistent sein.
- Das persistente Objekt unterscheidet sich aus Sicht eines Kundenobjekts nicht von seiner transienten Version.

- Das Speichern und Laden muß nicht explizit verlangt werden.

3.1.2 Die Erzeugung persistenter Objekte

Objekte können auf verschiedene Arten persistent werden:

- Zugehörigkeit zu einer speziellen Klasse:
  Ausnahmslos alle Exemplare dieser Klasse sind persistent. Diese Art ist unerwünscht, da hier die Persistenz nicht unabhängig von der Klasse ist.
- Entscheidung über Persistenz bei Erzeugung:
  Bei Erzeugung eines Objekts wird festgelegt, ob es transient oder persistent sein soll. Diese Eigenschaft behält das Objekt während seiner gesamten Lebensdauer.
- Entscheidung über Persistenz durch Kunden:
  Die Eigenschaft der Persistenz kann einem Objekt nachträglich gegeben und auch wieder genommen werden.
- Bekanntschaft mit einem anderen persistenten Objekt:
  Verweist ein persistentes Objekt auf ein transientes, „kennt“ es also das transiente, wird der Verweis ungültig, sobald das transiente nicht mehr existiert. Aus diesem Grund gibt es die Möglichkeit, allen transienten Objekten, die ein persistentes Objekt kennt, die Eigenschaft der Persistenz nachträglich zu geben. Das geschieht automatisch beim Speichern des persistenten Objekts (Persistence by Reachability).

3.2 Zugriff auf persistente Objekte

Im vorigen Abschnitt wurde gezeigt, daß Objekte die Eigenschaft der Persistenz irgendwann bekommen müssen. Jetzt wird die Frage untersucht, wie sich diese Eigenschaft auf die Implementierung ihrer Klassen und auf die Schnittstellen auswirkt.

Laden eines persistenten Objekts: Um ein persistentes Objekt zu laden, muß es zunächst identifiziert werden. Die Formen der Identifikation und die daraus resultierenden Zugriffsmöglichkeiten auf persistente Objekte werden in 3.2.1 beschrieben. Im Normalfall wird im Speicher der Anwendung eine Kopie des Objekts angelegt, mit der wie mit einem transienten Objekt gearbeitet werden kann.

Speichern eines persistenten Objekts: Kommt es beim Zugriff auf ein persistentes Objekt zu Änderungen an dessen Attributbelegungen, muß für die Übernahme der Änderungen in die Datenbank gesorgt werden. Zu untersuchen ist, wer für die Speicherung sorgt und wie sie angestoßen wird. Es können 3 Methoden unterschieden werden:

- Explizite Speicherung (Abschnitt 3.2.2)
- Geändert–Markierung (Abschnitt 3.2.3)
- Implizite Speicherung (Abschnitt 3.2.4)

3.2.1 Identifikation und Laden persistenter Objekte

Es gibt drei Formen der Identifikation: Zeigen, Benennen und Umschreiben [Wendt 91, S.16]. Das *Zeigen* erfordert die direkte Verfügbarkeit des Objekts in der Umgebung des Zeigenden, welcher mit Hilfe eines Zeigers auf das Objekt verweist. Der Zeiger entspricht einem Verweis auf den Ort des Objekts. Beim *Benennen* wird das Objekt über ein Symbol, z.B. einen Namen, identifiziert. Das *Umschreiben* betrifft seine Eigenschaften und Beziehungen zu anderen Objekten und kann mehrdeutig sein.

Der Zugriff auf Objekte der Objektdatenbank kann direkt oder indirekt erfolgen. Beim direkten Zugriff tritt der Datenbankverwalter nicht mehr auf; die Anwendung greift auf Objekte der Datenbank in gleicher Weise zu wie auf Objekte in ihrem Speicher. Beim indirekten Zugriff wird der Datenbankverwalter beauftragt, die identifizierten Objekte entweder für den direkten Zugriff zur Verfügung zu stellen, oder Methodenaufrufe und Argumente an sie durchzureichen.

Der Zugriff auf persistente Objekte kann auf vier verschiedene Arten erfolgen:

Direkt über die Objekt–ID: Mit Hilfe der Objekt–ID des Zielobjekts, die gleichzeitig ein Verweis auf dessen Ort ist, wird auf das Objekt zugegriffen. Das Auflösen der Objekt–ID führt wie bei transienten Objekten direkt zum Ziel. (Beispiel: Man besitzt die Telefonnummer von Gerhard Müller und kann ihn jederzeit anrufen)

Indirekt durch Benennung: Das Objekt wird durch ein Symbol, beispielsweise durch einen Namen, identifiziert, und vom Datenbankverwalter für den Zugriff zur Verfügung gestellt. Auch die Objekt–ID kann als Symbol benutzt werden, wenn der direkte Zugriff nicht möglich ist. (Beispiel: Die Telefonzentrale wird beauftragt, eine Verbindung zu Gerhard Müller aufzubauen und das Gespräch durchzustellen oder die Telefonnummer mitzuteilen, damit man in Zukunft selbst anrufen kann)

Indirekt durch Navigation über Verweise: Der Datenbankverwalter wird beauftragt, ausgehend von einem Einstiegspunkt (*Database Root, Root Object*) Verweisen bis zum Zielobjekt zu folgen. Mit dem Begriff der Navigation ist verbunden, daß der Zugriff auch durch Folgen einer Kette von Verweisen möglich ist. Es handelt sich um eine Umschreibung des Objekts. (Beispiel: Die Telefonzentrale wird beauftragt, bei der Firma Kugellager AG anzurufen, sich mit dem Spezialisten für Schmierstoffe (hoffentlich noch Gerhard Müller) verbinden zu lassen und das Gespräch durchzustellen)

Indirekt über die Suche in Objektmengen: Der Datenbankverwalter wird beauftragt, die Menge aller Objekte, deren Attributbelegungen bestimmte Bedingungen erfüllen, zur Verfügung zu stellen. Die Suche wird in einer Objektmenge vorgenommen, das Ergebnis ist wiederum eine Menge. Die Suche in Objektmengen entspricht der Ausführung von Queries in relationalen Datenbanksystemen. (Beispiel: Die Telefonzentrale wird beauftragt, in ihrem Telefonbuch alle Gerhard Müller herauszusuchen, die in Berlin in der Friedrichstraße wohnen, und die Telefonnummern mitzuteilen.)

3.2.2 Explizite Speicherung

Abbildung 3.1 zeigt auf der linken Seite die Kopie eines persistenten Objekts im Speicher der Anwendung und dessen Verbindung zum Objektdatenbanksystem auf der rechten Seite. Das Objekt auf der linken Seite ist mit Attributen und Methoden dargestellt. Der Kanal nach oben stellt die Verbindung zu anderen Objekten dar. Die rechte Seite zeigt die Objektdatenbank, auf der der Datenteil des Objekts, die Attribute, gespeichert ist. Um nach Änderungen an den Attributen die Objektkopie in der Objektdatenbank zu speichern, müssen spezielle Methoden verwendet werden, um die die Schnittstelle des Objekts erweitert wird. Im Bild werden sie ODB–Zugriffsmethoden genannt. Die Kommunikation mit dem Datenbankverwalter erfolgt über diese Methoden.

Ein Kundenobjekt, welches bei einem persistenten Objekt Methoden aufruft, die dessen Attribute ändern, muß für dessen Speicherung sorgen, indem es nach den Änderungen ODB–Methoden wie `Store` aufruft. In einer Anwendung müssen persistente Objekte an diesen Stellen anders behandelt werden als rein transiente. Das Wissen um die Speicherung persistenter Objekte muß unter Umständen in jeder Klasse vorhanden sein, deren Exemplare auf persistente Objekte zugreifen.

Die explizite Speicherung ist einfach zu realisieren. Da jede Speicherung einen Zugriff auf die Datenbank nach sich zieht, sind implizite Transaktionen für diese Zugriffe nötig, falls sie nicht innerhalb einer explizit deklarierten Transaktion stattfinden — siehe auch Abschnitt 5.3.3. Die Eigenschaft der Persistenz kann jederzeit einem Objekt gegeben und wieder genommen werden.

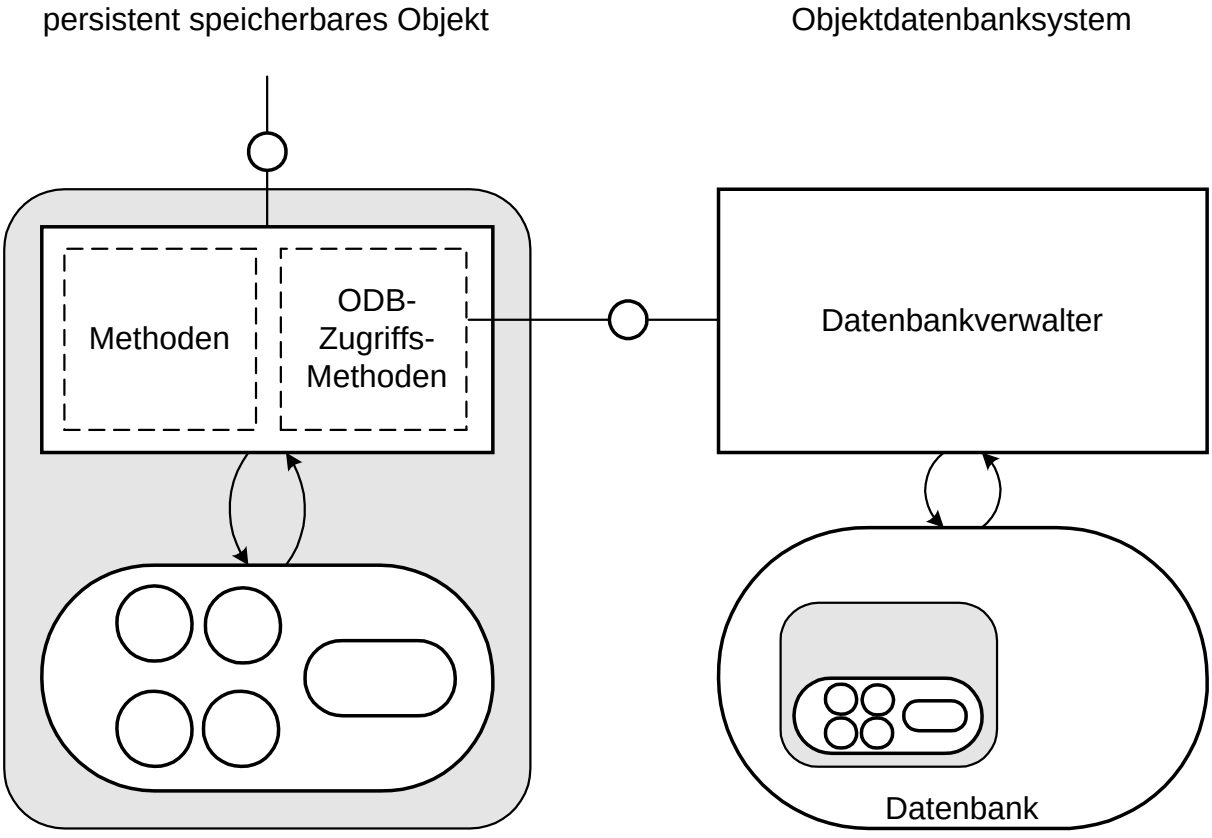

Abbildung 3.1: Explizite Speicherung

Von Nachteil ist der Aufwand, da entweder alle Routinen, die den Zustand eines persistenten Objekts ändern, sich um die Speicherung kümmern müssen, oder aber die Methoden generell so erweitert werden, daß jeder ändernde Zugriff das Speichern auslöst, wobei ein Objekt unter Umständen unnötig oft abgespeichert wird.

3.2.3 Geändert–Markierung

In Abbildung 3.2 ist die gleiche Beziehung zwischen persistentem Objekt und Objektdatenbank dargestellt wie in Abbildung 3.1. Der Unterschied ist in der Schnittstelle zu finden: Eine Zugriffsmethode, die ein Attribut ändert, wird um eine Routine erweitert, die das Objekt als geändert markiert (Mark Dirty, Mark Modified). Die Markierung erfolgt entweder auf einem speziellen Attribut des Objekts selbst oder durch eine Mitteilung an den Teil des Datenbankverwalters auf Seite des Clients.

Voraussetzung für das Verfahren ist das Transaktionskonzept. Der Datenbankverwalter führt eine Liste aller Objekte, auf die seit Beginn der Transaktion zugegriffen wurde. Wird das Commit ausgeführt, speichert er alle Objekte ab, die als geändert markiert wurden. Der Datenbankverwalter muß diese Liste ohnehin führen, da er verhindern muß, daß vom selben Objekt mehrere Kopien im Speicher der Anwendung angelegt werden.

Aus Sicht eines Kundenobjekts unterscheidet sich die Schnittstelle eines persistenten Objekts nicht von der eines transienten. Gegenüber einer Klasse mit rein transienten Exemplaren müs-

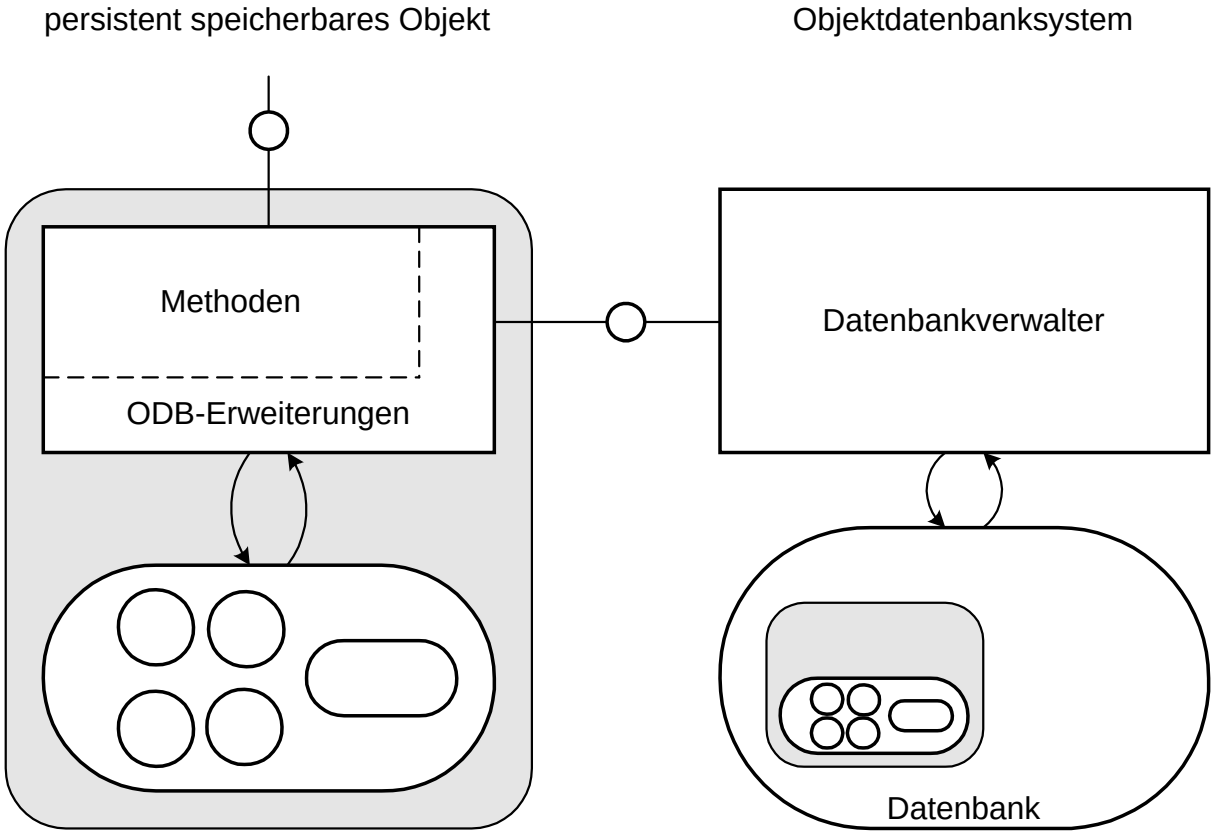

Abbildung 3.2: Geändert–Markierung innerhalb erweiterter Zugriffsmethoden

sen nur die Zugriffsmethoden, die Attribute ändern, um das Setzen des Merkers erweitert werden. Das Speichern der Objekte muß nun nicht mehr für jedes einzelne Objekt verlangt werden, sondern erfolgt zu einem Zeitpunkt, nämlich zum Commit.

Da sich die Schnittstelle des Objekts nicht ändert, bleiben Änderungen aufgrund der potentiellen Persistenz des Objekts nur auf seine Klasse beschränkt.

3.2.4 Implizite Speicherung

Bei der impliziten Speicherung sorgen andere Mechanismen dafür, daß Änderungen an den Attributen zur Speicherung bei Ausführung der Transaktion führen. Weder die Schnittstelle noch die Implementierung der Zugriffsmethoden ändert sich. Die implizite Speicherung bietet die größte Transparenz.

Die Mechanismen dazu werden zusammen mit dem Objektdatenbanksystem ObjectStore in Abschnitt 5.1 vorgestellt.

3.3 Zwei Sichtweisen auf ein Objektdatenbanksystem

Eine Objektdatenbank speichert Daten in Form von persistenten Objekten. In den letzten beiden Abschnitten wurde die Erzeugung von und der Zugriff auf persistente Objekte vorgestellt. In Abschnitt 3.2, Zugriff auf persistente Objekte, wurde bereits angedeutet, daß durch die Objektorientierung Mechanismen möglich sind, die den Datenbankverwalter beim Zugriff transparent machen. Auf der anderen Seite müssen Objekte nicht unbedingt zur Anwendung transportiert werden, wenn diese über den Datenbankverwalter darauf zugreift. Daraus resultieren zwei Sichtweisen auf ein Objektdatenbanksystem, die im folgenden vorgestellt werden. Die später vorgestellten Produkte bieten beide Sichtweisen in unterschiedlicher Ausprägung an.

3.3.1 Das Objektorientierte Datenbanksystem

Ein Objektorientiertes Datenbanksystem benutzt das Objektmodell zur Speicherung der Daten und bietet der Anwendung diese logische Sicht an. Die Anwendung greift über den Datenbankverwalter auf die Datenbank zu. Im Gegensatz zu relationalen Datenbanken kann die Identifikation der Objekte über deren Objekt–ID erfolgen. Die Anwendung kann objektorientiert oder rein prozedural programmiert sein, muß also innerhalb des eigenen Speichers den Objektbegriff nicht berücksichtigen.

Es handelt sich um eine *aktive Datenbank*, wenn Objekte in der Datenbank als Akteure auftreten können, der Datenbank–Server also Methoden der gespeicherten Objekte ausführen kann.

Abbildung 3.3 zeigt ein objektorientiertes Datenbanksystem. Auf der linken Seite befindet sich die Anwendung. Um anzudeuten, daß diese nicht objektorientiert realisiert zu sein braucht, sind im Speicher Zellen abgebildet, in denen die Daten abgelegt sind. Die Datenbank auf der rechten Seite beinhaltet ein Netzwerk von Objekten. Im Fall einer aktiven Datenbank sind auch die Methoden abgespeichert und ausführbar, so daß das System in der Datenbank aktiviert

werden kann. Die abgespeicherten Methoden sorgen mindestens für die Kapselung der Daten der Objekte. Es ist möglich, Teile der Anwendung oder die komplette Datenverarbeitung der Anwendung auf Seite des Datenbanksystems auszuführen.

Die Anwendung kommuniziert über eine Auftrags-/Rückmeldeschnittstelle mit dem Datenbankverwalter, um auf Objekte zuzugreifen, Suchaufträge zu vergeben und Methoden aufzurufen. Ist auch die Anwendung objektorientiert, können Verbindungen zwischen Objekten der Datenbank und Objekten der Anwendung hergestellt werden. Eine Verbindung ist beispielsweise ein Forwarder–Objekt, welches die Schnittstelle eines Objekts in der Objektdatenbank der Anwendung zur Verfügung stellt und alle Aufrufe, Argumente und Ergebnisse durchreicht (Beispiel: Gerhard Müller ist nicht direkt zu erreichen, man kann aber mit seinem Ansprechpartner telefonieren, der mit ihm in Verbindung steht). Die Kopplung von aktiver Objektdatenbank mit objektorientierter Anwendung wird bei der Vorstellung von GemStone in Abschnitt 5.4 behandelt.

GemStone ist ein Vertreter der aktiven objektorientierten Datenbanksysteme. In der ersten Version von GemStone existierte nur eine C-Sprachschnittstelle, über die die Smalltalk–Objekte auf der Datenbank angesprochen werden konnten.

3.3.2 Der Persistente Objektspeicher

Abbildung 3.4 zeigt eine Objektdatenbank als Persistenten Objektspeicher. Der Speicher der Anwendung wird um einen persistenten Teil, hier grau unterlegt, erweitert. Der Datenbankverwalter sorgt dafür, daß die Anwendung auf ein persistentes Objekt direkt zugreifen kann, indem er eine Kopie in dessen Speicher erzeugt und die geänderte Objektkopie wieder in die Datenbank zurückspeichert. Die Objekte im grau unterlegten Teil des Speichers auf der linken Seite sind Kopien der Objekte im grau unterlegten Teil der Datenbank auf der rechten Seite.

Die Anwendung kann auf persistente Objekte in der gleichen Weise zugreifen wie auf tran-

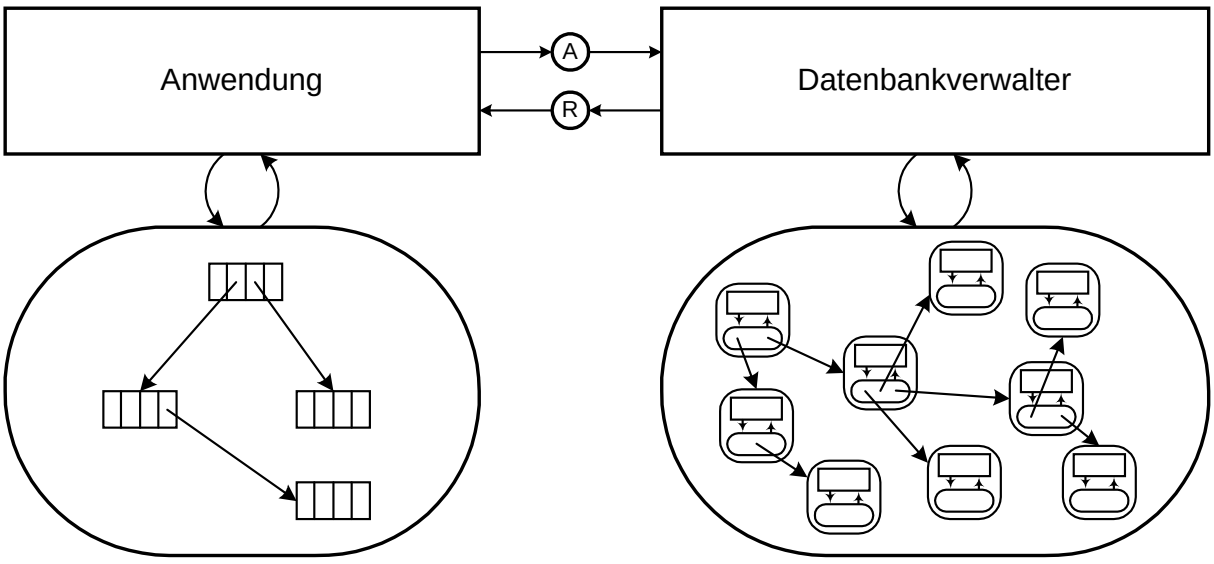

Abbildung 3.3: Objektorientiertes Datenbanksystem

siente. Der Zugriff erfolgt direkt, der Datenbankverwalter ist beim Zugriff transparent. Die Transparenz wird erreicht, indem entweder die von der Programmiersprache bereitgestellten Zugriffsmethoden auf Objekte um das Laden und Speichern persistenter Objekte erweitert werden oder die Speicherverwaltung des Betriebssystems modifiziert wird. Der Datenbankverwalter tritt nur auf zur Eröffnung einer Sitzung, zum Anmelden und Anstoßen von Transaktionen, zur Anforderung von Sperren und zur Erzeugung persistenter Objekte. Methoden werden nicht auf der Datenbank abgelegt, sondern sind fest mit dem Anwendungscode verbunden.

Greifen mehrere Anwendungen auf die Objektdatenbank zu, ist der Vergleich zu Shared Memory angemessen. Alle Anwendungen können auf Objekte im gemeinsamen persistenten Objektbereich zugreifen. Ein Objekt kann aber immer nur von einer Anwendung modifiziert werden.

Ein Datenbanksystem, das besonders große Transparenz bietet, ist ObjectStore (siehe Abschnitt 5.1).

3.4 Das Datenbank–Schema

Der Objektdatenbankverwalter benötigt zur Abspeicherung von Objekten Informationen über deren Struktur. Diese Information wird *Datenbank–Schema* genannt. Die Struktur eines Objekts wird in seiner Klasse beschrieben.

In Smalltalk trägt jedes Objekt die Information seiner Klassenzugehörigkeit; die Klassenbeschreibungen sind als Objekte vorhanden. Das Datenbank–Schema wird hier dynamisch erzeugt und mitgeführt. Die Sprache C++ verlangt einen höheren Aufwand, was im folgenden beschrieben wird.

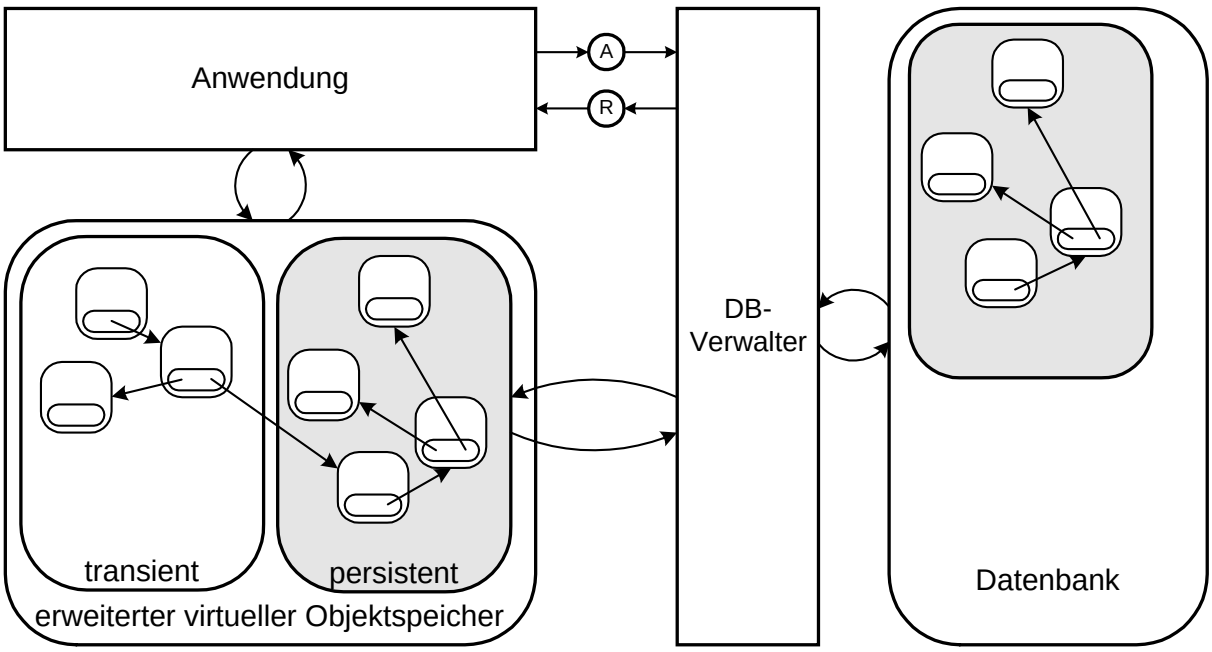

Abbildung 3.4: Persistenter Objektspeicher

Erweiterung von C++

Die Sprache C++ bietet zwei Informationen nicht zur Laufzeit, die für die Kommunikation mit Objektdatenbanken benötigt werden:

- Die Information, zu welcher Klasse ein Objekt gehört[1]
- Die Klassenbeschreibung selbst

Bei Einsatz einer Objektdatenbank wird die Anwendung um eine Laufzeit–Klasseninformation erweitert. Ein Objekt kann damit Auskunft geben, zu welcher Klasse es gehört. Die Informationen über die Struktur von Objekten kann man aus den Klassenbeschreibungen, die in Form von Quelltexten vorhanden sind, und dem zur Übersetzung der Quelltexte verwendeten Compiler gewinnen. Beim Übersetzen der Quelltexte der Anwendung wird das Datenbank–Schema erzeugt und zum Datenbankverwalter übertragen.

Die Erzeugung des Datenbank–Schemas

Abbildung 3.5 zeigt den Aufbau des Systems, das sich bei der Compilierung der Anwendung ergibt:

Auf der rechten Seite befindet sich als Eingabegröße der Quelltext, der im Normalfall vom Compiler verarbeitet wird, wobei ausführbarer Code entsteht. Bei Verwendung einer Objektdatenbank beschreibt ein Teil des Quelltextes Klassen, die persistente Exemplare haben können. Diese Klassen müssen als solche gekennzeichnet werden, wodurch eine weitere Eingabegröße hinzukommt: Die Persistenz–Information (oben links). Sie trägt die Informationen, welche Klassen persistente Exemplare haben können. Diese Information kann Teil des Quelltextes sein, z.B. eine Markierung einer Klassenbeschreibung, oder eine eigene Datei, die eine Liste von persistenten Klassen enthält.

Aus Quelltext und Persistenz–Information erzeugt der Schema–Compiler zum einen die Datenbank–Schema–Information, welche zum Datenbankverwalter übertragen wird, der sie bei der Datenbank ablegt, zum anderen wird Quelltext generiert, der die Anwendung um die Laufzeit–Klasseninformation erweitert.

3.5 Schema–Evolution und Objektmigration

Es kommt vor, daß eine Anwendung modifiziert und neu übersetzt werden muß, wenn bereits Daten in die Datenbank gespeichert wurden. Änderungen an Klassen ziehen die Änderung des Datenbank–Schemas nach sich, wenn Klassen persistenter Objekten betroffen sind. Die Änderung oder Erweiterung des Schemas wird oft *Schema–Evolution* genannt.

[1] Das späte Binden (Late Binding) von Methoden zu einem Objekt im Zusammenhang mit Polymorphismus benutzt eine Laufzeitinformation der Zugehörigkeit zu einer Klasse, die aber nicht abfragbar ist.

Eine Änderung am Datenteil einer Klasse hat zur Folge, daß bereits gespeicherte Exemplare nicht mehr zur Klasse passen. Die *Objektmigration* bringt persistente Objekte auf den neuesten Stand ihrer Klasse. Da unter Umständen Attribute neu hinzukommen und vorbelegt werden müssen, besteht manchmal die Notwendigkeit, die Migrationsprozedur um eigene Routinen zu erweitern.

Die Objektmigration kann auf zwei Arten erfolgen: Bei der expliziten Migration werden alle Exemplare einer Klasse auf eine Version, normalerweise die neueste, gebracht. Für die Betrachtung der Verfügbarkeit des Datenbanksystems ist es interessant, ob die Migration aller Exemplare einer oder aller Klassen online erfolgen kann. Daneben gibt es die Möglichkeit, Exemplare erst bei Bedarf in der Datenbank zu migrieren, nämlich bei Zugriff durch eine Anwendung.

Eine Objektdatenbank kann *Versionen von Klassen* zulassen. Eine Änderung an einer Klasse bewirkt nicht das Überschreiben der alten Version, sondern das Anlegen einer neuen. Damit ist es möglich, Exemplare einer Klasse in verschiedenen Versionen in der Datenbank zu halten. Konvertierungs–Automatismen und spezielle Routinen können dafür sorgen, daß ein Exemplar bei Zugriff durch eine Anwendung auf die Version der Klasse gebracht wird, die die Anwendung benötigt.

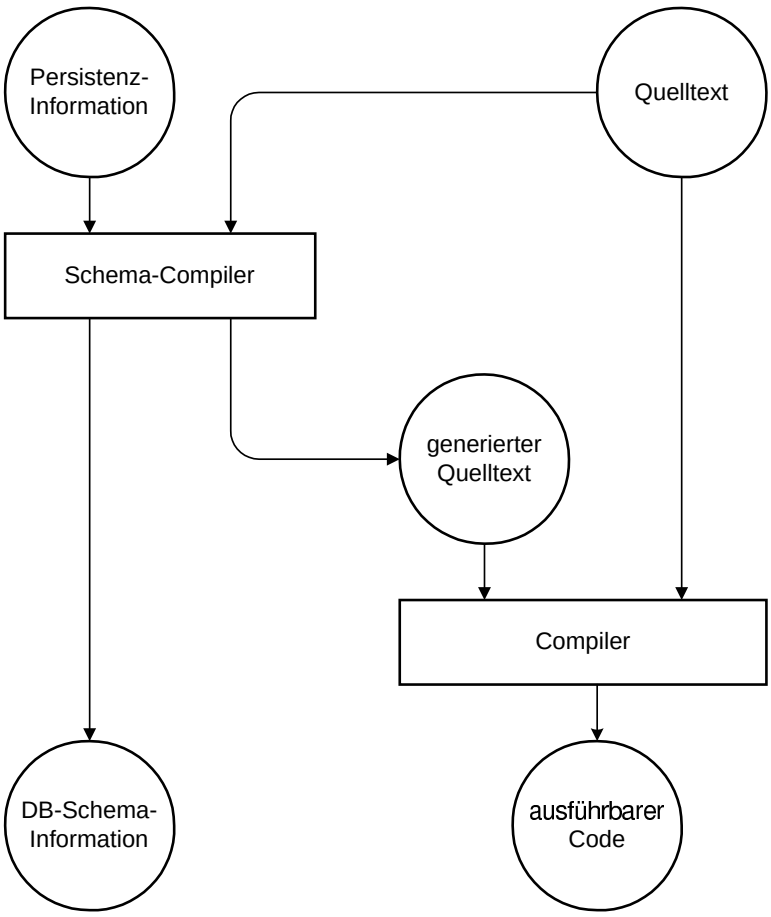

Abbildung 3.5: Gewinnung des Datenbank-Schemas bei C++

Kapitel 4

Anforderungen an Datenbanken in Client-Server-Systemen

Der Einsatz von Datenbanksystemen in einer heterogenen Client–Server–Umgebung stellt hohe Anforderungen an die Produkte. Diese Anforderungen lassen sich in 4 Kategorien unterteilen:

- Fähigkeit zum Mehrbenutzerbetrieb mit konkurrierenden Zugriffen
- Betrieb in heterogenen Systemen
- Skalierbarkeit
- Unterstützung der Datenbank–Administration und Entwicklung

4.1 Mehrbenutzerbetrieb und konkurrierende Zugriffe

4.1.1 Sperren (Locks)

Eine Anwendung kann den schreibenden oder lesenden Zugriff auf Daten durch andere verhindern, wenn sie beim Datenbankverwalter eine Sperre darauf beantragt. Sperren können automatisch gesetzt (Implizite Sperren) oder explizit beantragt werden. Sperren können bis zum Ende einer Transaktion, bis zum Ende einer Sitzung oder dauerhaft gültig sein. Werden Sperren falsch verwendet, ist die Gefahr einer Blockierung groß. Das feinste Sperr–Granulat sollte das einzelne Objekt sein.

4.1.2 Transaktionen

Operationen der Anwendung betreffen meist mehrere Objekte der Datenbank. Da immer die Gefahr besteht, daß das Eintreten eines Fehlers zu einem Zeitpunkt erfolgen kann, in dem noch nicht alle nötigen Änderungen gemacht wurden, werden die Operationen zu einer Transaktion zusammengefaßt.

Das Wesen der Transaktion ist, daß entweder alle Operationen ausgeführt werden oder keine (Atomarität). Auf diese Weise kann gesichert werden, daß der zu bearbeitende Datenbestand konsistent bleibt. Mit Start der Transaktion werden alle geänderten Objekte gepuffert, bei Commit werden sie auf die Datenbank übertragen. Bei Fehlschlagen der Transaktion wird der Puffer verworfen.
In Abbildung 4.1 sind zwei Transaktionsstrategien dargestellt. In beiden Fällen wird der Beginn der Transaktion von der Anwendung beim Datenbankverwalter angemeldet. Die Anwendung führt ihre Operationen auf den Daten aus, bis sie schließlich durch das Commit dem Datenbankverwalter den Auftrag gibt, alle geänderten Objekte zu speichern, oder aber die Transaktion abbricht (Abort). Der Datenbankverwalter versucht im Fall des Commit, die geänderten Objekte in der Datenbank zu speichern. Ist er erfolgreich, ist die Transaktion beendet.

Pessimistische Transaktionsstrategie: Alle Objekte, auf die innerhalb einer Transaktion zugegriffen wird, werden gesperrt. Konnten alle Sperren gewährt werden, wird die Transaktion erfolgreich sein. Der Nachteil daran ist, daß es unter Umständen zu Einbußen beim Durchsatz (Performance) oder sogar zu Verklemmungen (Deadlocks) kommt, da normalerweise mehr und länger gesperrt wird, als tatsächlich nötig ist.

Optimistische Transaktionsstrategie: Die Sperren werden erst bei Commit in der Hoffnung beantragt, daß sie noch verfügbar sind und daß seit Beginn der Transaktion kein Zugriff auf die Objekte von anderer Seite erfolgte, der mit den Sperren kollidiert. Sind beide Voraussetzungen gegeben, gibt es keinen Unterschied zu der Situation, in der von vornherein gesperrt wird. Sperren werden damit nur zum Zeitpunkt des Commit und nur für die Objekte angefordert, auf die tatsächlich zugegriffen wurde. Der Nachteil an dieser Strategie ist, daß ganze Transaktionen unter Umständen mehrfach wiederholt werden müssen, bis sie erfolgreich ausgeführt werden können.

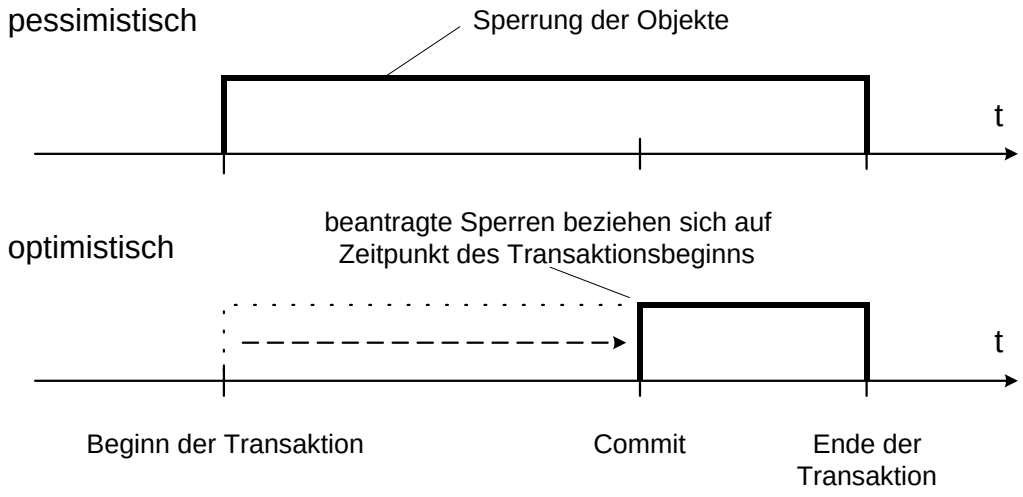

Abbildung 4.1: Transaktionsstrategien

4.1.3 Zugriffsbeschränkungen und Autorisierung

In einem Mehrbenutzersystem kommt es vor, daß bestimmte Daten nicht jedem Benutzer zum Schreiben oder Lesen zugänglich sein sollen. Zu untersuchen ist, in welcher Granularität Zugriffsbeschränkungen vergeben werden können. Möglich sind: Objekte, Mengen von Objekten, alle Exemplare einer Klasse oder ganze Datenbanken.

Die Autorisierung eines Benutzers gegenüber der Datenbank kann durch das Betriebssystem des Datenbank–Servers oder durch den Datenbankverwalter selbst erfolgen.

4.1.4 Lange Transaktionen (Check-In/Out) und Objektversionen

Lange Transaktionen sind Transaktionen, die über eine Datenbanksitzung hinausgehen können. Üblicherweise erfolgt eine lange Transaktion in 3 Schritten:

1. Sperren der Objekte und Übertragen in einen privaten Bereich (Workspace): Check-Out. Der Workspace befindet sich im Normalfall auf Seite des Clients und stellt eine eigene Objektdatenbank dar.
2. Bearbeiten der Objekte im privaten Bereich. Hierbei ist es nicht notwendig, mit der Datenbank verbunden zu sein.
3. Zurückschreiben der Objekte in die Datenbank, Aufheben der Sperren: Check–In

Für lange Transaktionen gibt es mehrere Voraussetzungen:

- Sperren müssen auch über eine Sitzung hinaus erhalten bleiben (persistente Sperren).
- Die Objekt–ID darf sich bei allen Übertragungsvorgängen nicht ändern.

Versionierte Objekte: Eine andere Spielart der langen Transaktion benutzt Versionen von Objekten. Der Begriff Version bezieht sich hier auf die Daten selbst, nicht auf das Format, in dem sie abgespeichert sind.
Beim Check–Out werden keine Sperren auf die Objekte gesetzt. Erfolgt das Check–In, wird eine neue Version auf der Datenbank angelegt. Der Versionsgraph kann sich verzweigen, wenn nach dem Check–Out des ersten Benutzers ein anderer Benutzer ebenfalls ein Check–Out durchführt. Versionen können wieder zusammengeführt werden.

4.1.5 Ereignismeldungen des Servers

Es gibt Situationen, in denen eine Anwendung benachrichtigt werden muß, wenn auf der Objektdatenbank Änderungen am Zustand bestimmter Objekte vorgenommen werden. Beispielsweise zeigen mehrere Anwendungen Daten von Objekten aus der Datenbank an. Ändert eine Anwendung ein Objekt, werden alle anderen Anwendungen vom Server benachrichtigt, die

Meldungen bezüglich dessen Änderung abonniert haben. Eine Reaktion auf diese Benachrichtigung ist im Beispiel das Auffrischen der Darstellung auf dem Bildschirm.

Watch/Notify: Eine Anwendung abonniert beim Server Meldungen bezüglich Änderungen oder Sperranträgen auf bestimmte Objekte oder Klassen, setzt einen *Watch*[1]. Der Server benachrichtigt die Anwendungen (*Notify*), indem er die für diesen Zweck angemeldeten Routinen der Anwendungen, die *Callbacks*, aufruft. Eine andere Bezeichnung ist Publish/Subscribe: Der Server veröffentlicht Änderungsmeldungen, auf die sich die Anwendungen abonnieren.

Neben Änderungen und Sperranträgen gibt es weitere Ereignisse, die abonniert werden können. Ein Beispiel sind die sogenannten *Progress Callbacks* von POET, die in Abschnitt 5.3.3 näher behandelt werden.

4.2 Heterogene Systeme

4.2.1 Anwendungen in unterschiedlichen Programmiersprachen

Um eine Objektdatenbank für eine Anwendung benutzen zu können, muß der Datenbankhersteller eine Sprachschnittstelle für die Programmiersprache der Anwendung bereitstellen. Hierbei ist zu unterscheiden, ob die Anwendungen ganze Objekte übertragen kann (Objektzugriff) oder ob sie nur einzelne Attribute von Objekten abfragen kann (Datenzugriff).

Daneben ist zu untersuchen, ob Objekte, die von einer Anwendung in der ersten Sprache auf der Datenbank gespeichert wurden, von einer anderen in der zweiten lesbar sind. Smalltalk und C++ haben beispielsweise große Unterschiede in der Datenstruktur von Objekten, so daß eine Konvertierung nötig ist.

Weitere Schnittstellen betreffen Standards für Datenbankabfragen wie SQL, OQL und ODBC und Datenübertragung zwischen Anwendungen wie OLE[2]

4.2.2 Clients auf verschiedenartigen Rechnern

Werden verschiedene Compiler, Betriebssysteme oder gar Rechnertypen verwendet, kann die Speicherrepräsentation der Objekte verschieden sein. Um Objekte der Datenbank gleichermaßen verwenden zu können, muß eine Konvertierung in das jeweilige Format erfolgen.

[1] Der Begriff Watch kommt von einem Werkzeug, mit dem Programme getestet werden können, dem Debugger. Einen Watch setzen bedeutet hier, ständig die Belegung einer Variable anzuzeigen und unter Umständen jede Änderung an deren Belegung zu melden.

[2] Abkürzungen:
ODBC (Open Database Connectivity): Datenbank–Schnittstelle für Desktop–Anwendungen von Microsoft
SQL (Structured Query Language): Deklarative Abfragesprache für relationale Datenbanken
OQL (Object Query Language): Deklarative Abfragesprache für Objektdatenbanken, siehe [ODMG 93]
OLE (Object Linking and Embedding): Verfahren zum Austausch von Daten zwischen Windows–Programmen, von Microsoft spezifiziert.

4.2.3 Kommunikationsprotokolle

Die Server und Clients können in verschiedenartigen Netzwerken liegen. Die Netzwerk–Protokoll–Schicht muß gekapselt werden, um verschiedene Protokolle fahren zu können (z.B. TCP/IP, IPX, NetBios)

4.3 Skalierbarkeit

4.3.1 Verteilung der Datenbank, Multi–Server–Betrieb

Ein Datenbankserver kann mehrere Datenbanken verwalten. Die einfachste Leistungserhöhung besteht darin, einen weiteren Rechner mit weiteren Datenbanken zur Verfügung zu stellen. Zu diesem Zweck ist es erforderlich, von einem Schema mehrere Datenbanken erzeugen zu können. Eine Anwendung wiederum muß die Möglichkeit haben, Sitzungen für verschiedene Datenbanken bei verschiedenen Servern anzumelden und Daten dieser Datenbanken zu verknüpfen. Bei *verteilten Transaktionen* sind in einer Transaktion Daten von verschiedenen Datenbanken betroffen, so daß erst für jede einzelne und dann für alle Datenbanken zusammen festgestellt werden muß, ob die Transaktion erfolgreich war; dieses Transaktionsprotokoll wird Two–phase Commit genannt. Objekte sollen auf Objekte in anderen Datenbanken verweisen können (Cross–Database Reference), unabhängig davon, von welchem Server sie verwaltet werden.

Weiterhin besteht die Möglichkeit, den Datenbankverwalter auf mehrere Rechner zu verteilen und über mehrere Rechner verteilte Datenbanken als ganzes anzusprechen.

4.3.2 Minimierung von Netzauslastung, verteilte Ausführung der Anwendungen

Das Netzwerk stellt oft einen Flaschenhals beim Durchsatz dar. Durch Pufferung (Caching) können mehrfache Zugriffe auf die selben Objekte bei weniger Netzlast erfolgen.

Insbesondere Abfragen auf großen Datenmengen (Queries) verursachen eine große Netzlast. Es ist daher sinnvoll, datenintensive Aufgaben dort abzuwickeln, wo sich die zu verarbeitenden Daten befinden, nämlich auf dem Server.

4.3.3 Verfügbarkeit und Sicherheit

Datenbanken sollen rund um die Uhr verfügbar sein. Aus diesem Grund darf ein Backup den Betrieb genausowenig stören wie Änderungen am Datenbankschema.

Im Fehlerfall muß die Datenbank auf einen konsistenten Stand gebracht werden können. Log–Dateien werden vom Datenbankverwalter geführt, um mit ihrer Hilfe den letzten konsistenten

Stand vor Eintreten des Fehlers wieder herzustellen. Eine Datenbank-Spiegelung sorgt für eine redundante Speicherung der Daten, die im Fehlerfall für die ausgefallene Datenbank einspringt.

4.4 Datenbank–Administration, Entwicklung

Bei größeren Systemen kann die Administration des Datenbanksystems mit viel Aufwand verbunden sein. Die Hersteller liefern verschiedene Werkzeuge zur Administration mit. Einige grundsätzliche Fragen sind folgende:

- Welche Privilegien benötigt der Datenbank–Administrator auf dem Server–Rechner? Ist er für die Benutzerverwaltung zuständig?
- Können Administrations-Vorgänge auch von anderen Rechnern aus angestoßen werden?
- Wie wird die Administration verteilter Datenbanken und Multi–Server–Systeme unterstützt?

Bei Verwendung einer Objektdatenbank muß bei Übersetzen des Quelltexts ein zur Anwendung passendes Datenbank–Schema generiert werden, wie in Abschnitt 3.4 gezeigt. Der Schema–Compiler und andere Werkzeuge sollen zu diesem Zweck möglichst gut in die bestehende Entwicklungsumgebung integriert werden können.

Unter Umständen müssen bereits vorhandene Klassen geändert werden, wenn deren Exemplare nun auf persistente Objekte zugreifen sollen. Das ist insbesondere der Fall, wenn persistente Objekte explizit gespeichert werden müssen, wie in Abschnitt 3.2.2 gezeigt wurde.

Änderungen an persistenten Klassen ziehen die Migration der Objekte auf der Datenbank nach sich (siehe Abschnitt 3.5). Zu untersuchen ist, mit welchen Werkzeugen die Migration unterstützt wird, wie sie den jeweiligen Bedürfnissen angepaßt werden kann und ob sie online möglich ist.

Erlaubt das Datenbanksystem Versionen von Klassen, können Objekte der gleichen Klasse, aber verschiedener Version, gleichzeitig in der Datenbank existieren (siehe Abschnitt 3.5). Auch hier sind die zur Verfügung gestellten Werkzeuge von Interesse.

Kapitel 5

Vorstellung der Produkte

In diesem Kapitel werden vier Objektdatenbanken untersucht. Die Auswahl der Produkte war willkürlich, wobei damit durchaus die auf dem Markt wichtigsten Objektdatenbanksysteme vertreten sind, die sich in der verwendeten Technik zum Teil erheblich unterscheiden. Es sind im einzelnen:

- OBJECTSTORE Release 4 von Object Design Inc.
- VERSANT Release 4.0 von Versant Object Technology
- POET 3.0 von Poet Software GmbH
- GEMSTONE Version 4.1 von GemStone Systems Inc., vormals Servio

Die Produkte wurden im Labor installiert und mit einfachen Beispielen getestet. Der Großteil der Informationen stammt aus dem umfangreichen Dokumentationsmaterial, welches mit den Produkten geliefert wurde. Einen Überblick über die verwendeten Handbücher bietet das Literaturverzeichnis am Ende dieser Schrift. Die wichtigsten Informationen bieten die Handbücher für die Programmentwicklung mit den Produkten und die Handbücher zur Systemverwaltung. Bei Unklarheiten oder Fragen, die nicht mit Hilfe der Dokumentation zu beantworten waren, wurde der Support der jeweiligen Firma bemüht, der schnell und präzise antwortete.

Die Vorstellung der Produkte erfolgt anhand der in Kapitel 3 vorgestellten Konzepte, wobei insbesondere die Eignung für Client-Server-Umgebungen untersucht wird. Die Kriterien wurden in Kapitel 4 vorgestellt. Mit Ausnahme von GemStone wird nur die C++-Sprachschnittstelle betrachtet.

Gliederung

Die Vorstellung eines Produkts ist wie folgt gegliedert:

1. Einordnung des Produkts
2. Technik

3. Mehrbenutzerbetrieb
4. Heterogene Systeme
5. Skalierbarkeit
6. Administration
7. Entwicklung
8. Einschätzung

Erläuterung zu den Punkten:

Punkte 1 und 2 befassen sich vor allem mit der Einordnung des Produkts in die in Kapitel 3 vorgestellten Konzepte.

Die Punkte 3 bis 7 untersuchen die Erfüllung der Kriterien für die Eignung für Client–Server–Systeme, die in Kapitel 4 vorgestellt wurden.

In Punkt 8 werden Stärken und Schwächen des Produkts zusammengefaßt und eine Einschätzung zum Einsatzbereich des Produkts gegeben.

ODMG

Die Hersteller sind Mitglieder der ODMG, der Object Database Management Group, die Standards für Objektdatenbanken setzt. Die Standards betreffen die Sprachschnittstellen für C++ und Smalltalk, Container-Konzepte und Queries sowie die Abfragesprache OQL [ODMG 93].

Da der Standard neu ist, wird er noch bei keinem Hersteller vollständig umgesetzt. Für die neuen Versionen ihrer Produkte haben die Hersteller die Umsetzung des Standards angekündigt.

Lieferumfang

Zu einem Objektdatenbanksystem gehören folgende Komponenten:

- Datenbank–Server
- Datenbank–Clients: Laufzeitbibliotheken und Treiber
- Sprachschnittstellen mit:
  - Klassenbibliotheken
  - Schema–Compiler und andere Entwicklungswerkzeuge
- Administrationswerkzeuge
- Monitorprogramme und Browser, mit denen der Datenbestand interaktiv untersucht und abgefragt werden kann.

5.1 ObjectStore

Die Firma Object Design wurde 1988 gegründet. Mit dem Produkt ObjectStore wurden zu Anfang vor allem objektorientierte CAD-Anwendungen auf Unix-Plattformen realisiert.

5.1.1 Einordnung

ObjectStore ist ein typischer Vertreter der Ausprägung Persistenter Objektspeicher — siehe auch Abschnitt 3.3.2. Unter Ausnutzung von Betriebssystem–Mechanismen zur Verwaltung des virtuellen Speichers werden Speicherseiten auf einem Server abgespeichert.

Der Objektzugriff ist transparent, die Speicherung erfolgt implizit.

5.1.2 Technik

Abbildung 5.1 zeigt die Wirkungsweise des Seiten–Servers: Auf der linken Seite ist ein Ausschnitt des Speichers der Anwendung zu sehen. Der Speicher wird in kleine Bereiche, die Seiten, aufgeteilt. Die Größe einer Seite ist von der Plattform abhängig, unter Windows NT beträgt sie 4 KB. Seiten können zu logischen Einheiten, den Segmenten, zusammengefaßt werden. Die grau unterlegten Seiten sind Speicherbereiche, die für die Anwendung nicht zur

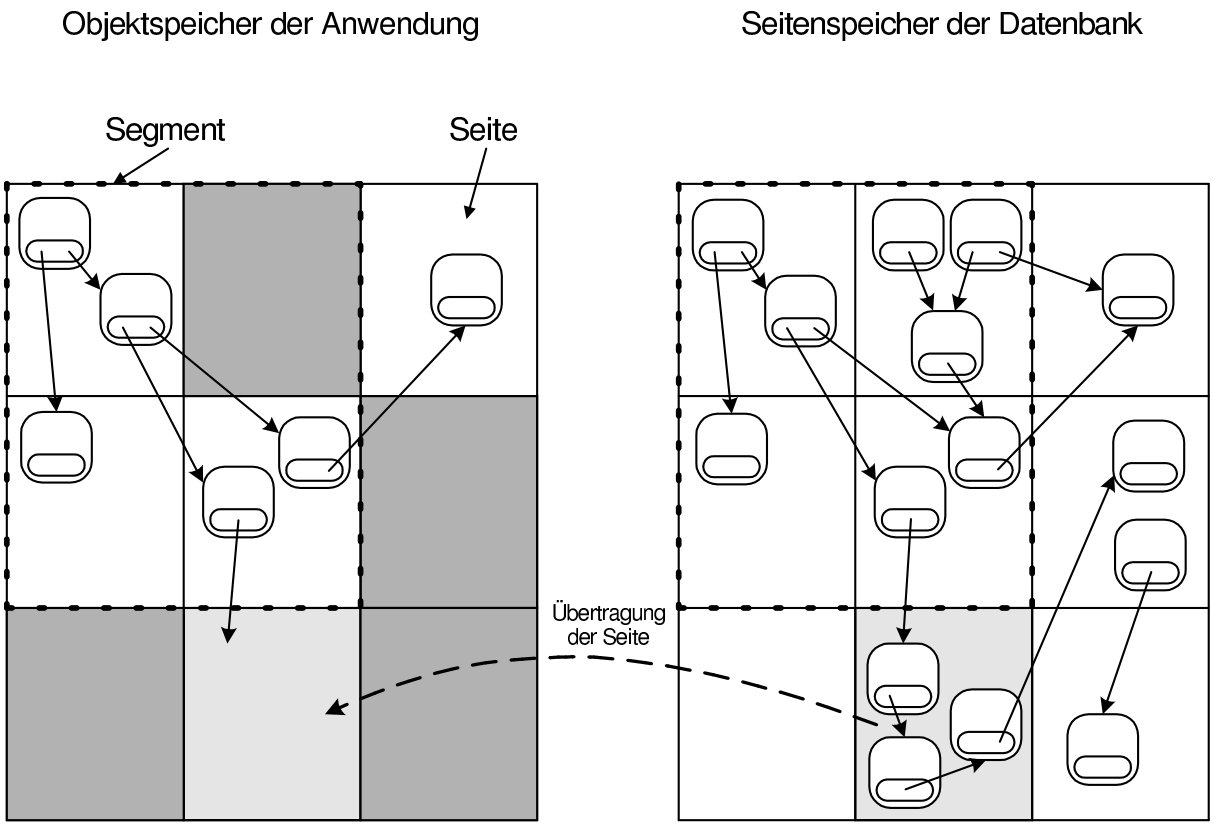

Abbildung 5.1: ObjectStore: Objekte und Seiten

Verfügung stehen. Auf der rechten Seite ist ein Ausschnitt aus dem Seitenspeicher der Datenbank dargestellt. Die vier hellen Seiten links im Speicher der Anwendung entsprechen denen in der Datenbank rechts. Sie sind bereits in den Objektspeicher der Anwendung übertragen worden.

Objekte können auf einer oder mehreren Seiten liegen, eine Seite kann mehrere Objekte enthalten. Da die Objekte im Speicher in der jeweiligen Binär–Repräsentation abgelegt sind, werden sie genauso auf der Datenbank abgelegt. Ein Objekt im Speicher enthält nur die Attribute; die Methoden finden sich im Programmcode (Executable oder Library). Methoden werden nicht in der Datenbank gespeichert.

Es wird nun das untere Objekt in der mittleren Seite betrachtet. Folgt die Anwendung dem Zeiger des Objekts auf die hellgraue Seite, die noch nicht im Speicher steht, wird dadurch eine Übertragung der Seite angestoßen. Zur Feststellung des Zugriffs auf diesen Speicherbereich wird die Speicherverwaltung des Betriebssystems erweitert:

Abbildung 5.2 zeigt den Aufbau eines ObjectStore–Systems. Wie in Abbildung 3.4 (Persistenter Objektspeicher) sieht die Anwendung einen erweiterten Objektspeicher. Der Datenbankverwalter wird in drei Komponenten unterteilt: Der Seiten–Server auf dem Server–Rechner, die Auftragsannahme und der Seitenverwalter auf dem Client–Rechner.

Lesender Zugriff: Es wird zunächst betrachtet, was passiert, wenn ein Zeiger auf eine Speicherseite verfolgt wird, die sich noch nicht im Speicher der Anwendung befindet (In Abbildung 5.1 die Seite unten in der Mitte). Die Anwendung greift lesend auf einen

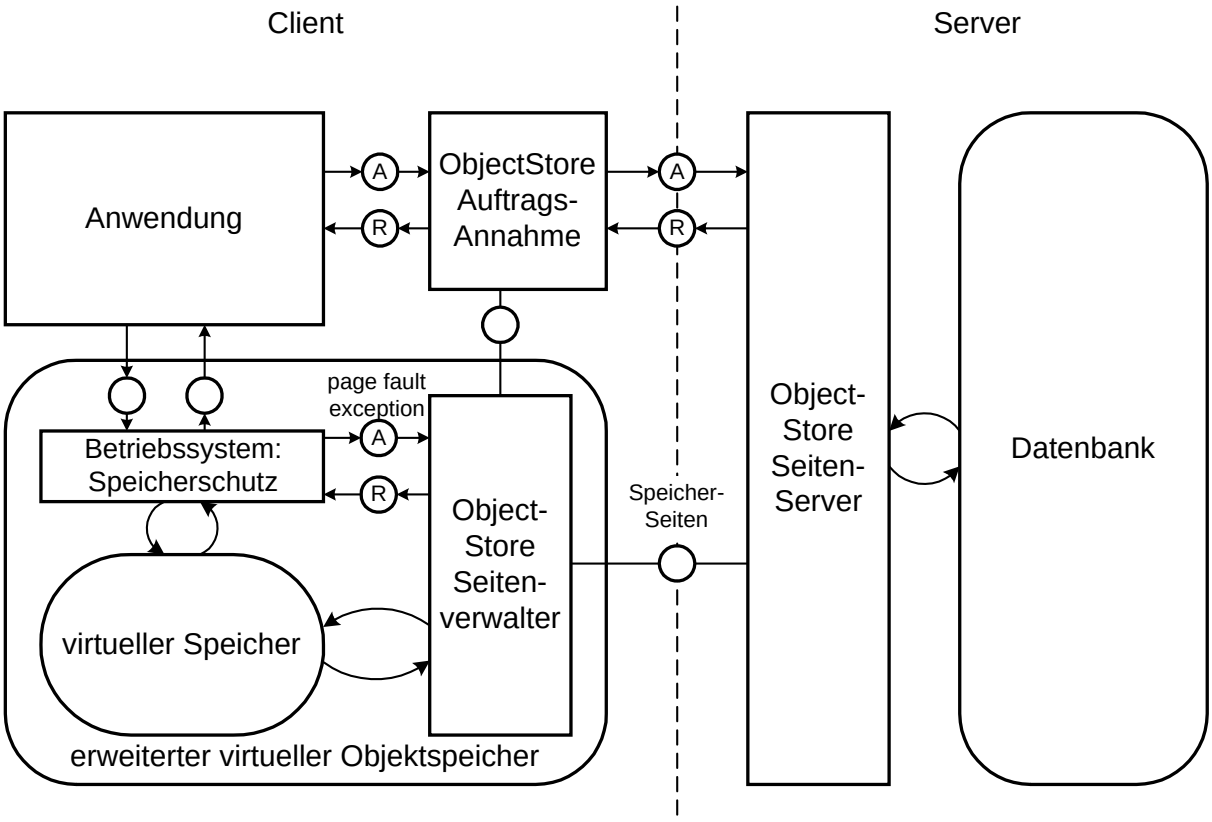

Abbildung 5.2: ObjectStore Architektur

Speicherbereich zu, der ihr noch nicht zugewiesen wurde (z.B. Zuweisung per `malloc`). Der Speicherschutz des Betriebssystems löst eine Page Fault Exception aufgrund eines Adressierungsfehlers aus. Diese wird vom Seitenverwalter abgefangen. Nach einer Prüfung, ob es sich um eine Seite aus der Datenbank handelt, fordert er die Seite vom Server an und kopiert sie in den Speicherbereich (Memory Mapping). Er gibt an den Speicherschutz–Akteur zurück mit der Aufforderung, den letzten Befehl, der die Exception ausgelöst hat, zu wiederholen. Da die Seite geladen ist, kann der Befehl erfolgreich ausgeführt werden. Der Anwendung ist die Übertragung der Seite transparent.

Schreibender Zugriff: Attribute eines persistenten Objekts sollen verändert werden. Beim ersten Lesen werden die Speicherseiten geladen und als schreibgeschützt beim Betriebssystem gemeldet. Versucht die Anwendung, auf eine Seite zu schreiben, wird wiederum vom Speicherschutz eine Ausnahme ausgelöst, die vom Seitenverwalter abgefangen wird. Dieser markiert die Seite als geändert, hebt den Schreibschutz auf und gibt wieder zurück. Beim Commit der Transaktion wird die Seite zurückgeschrieben.

Die oben beschriebene Methode dient zur Erweiterung des virtuellen Speichers und zur Einrichtung eines Shared Memory Bereiches, der rechnerübergreifend genutzt werden kann. Das Wissen um Objekte liegt in der Auftragsannahme. Der Seiten–Server kennt nur Speicherseiten, nicht aber deren Inhalt. Er muß daneben zu Beginn einer Sitzung die Schema–Information zur Verfügung stellen, damit die Auftragsannahme entscheiden kann, ob die Anwendung Objekte auf der Datenbank ablegen darf und ob sie die Struktur der gespeicherten Objekte kennt.

Ist der Speicherbereich für eine Seite im Speicher der Anwendung bereits durch andere Daten belegt, wird die Seite in einen freien Bereich abgebildet, wobei alle Zeiger auf dieser Seite angepaßt werden müssen. Diese Anpassung erfolgt durch den Seitenverwalter [OSTech 3, S. 23]. Der Seitenverwalter ist daneben auch für Anpassungen an die Plattform zuständig. Liefert der Server zum Beispiel Seiten, auf denen die Objekte Integer im Big Endian Format enthalten, konvertiert der Seitenverwalter diese auf das Little Endian Format, wenn die Client–Plattform das verlangt. Zu jeder Seite werden Informationen über die Lage der Objekte und Zeiger darauf, sowie plattformabhängige Daten mitgeliefert, die vom Seitenverwalter ausgewertet werden.

Der Einstieg in die Objektdatenbank erfolgt entweder durch benannte Einstiegspunkte oder durch Identifikation durch einen speziellen Verweis, `os_Reference`. Benannte Einstiegspunkte (Database Roots) sind Verweise, die durch einen Namen in Form eines Strings identifizierbar sind und einen C++-Zeiger auf ein Datenbankobjekt enthalten. Die Direkte Identifikation über C++-Zeiger ist nur innerhalb einer Transaktion möglich. Ein von ObjectStore eingeführter Verweistyp, `os_Reference`, erlaubt die Identifikation von Objekten über Transaktionsgrenzen hinweg und auf andere Datenbanken. Die Objekt–ID von `os_Reference` enthält Informationen über die Datenbank, in der das Objekt gespeichert ist.

Persistente Objekte

Die Technik von ObjectStore erlaubt durch die implizite Speicherung einen nahezu transparenten Umgang mit persistenten Objekten. Ein Objekt ist persistent, wenn es mit einem speziellen (überladenen) `new`–Operator erzeugt wurde und behält die Eigenschaft, solange es existiert.

Bei Erzeugung muß die Datenbank angegeben werden, auf die das Objekt zu speichern ist. Aus Optimierungsgründen kann ein Segment angegeben werden, zu dem es gehören soll.

Bei Ausführen einer Transaktion werden Objekte nicht daraufhin untersucht, ob sie auf transiente Objekte zeigen. Es gibt allerdings die Möglichkeit, den Datenbankverwalter aufzufordern, ungültige gewordene Verweise auf Null zu setzen.

Queries

Suchanfragen an den Datenbankverwalter werden auf Seite des Clients ausgeführt. Grund dafür ist die Tatsache, daß der Server nur für die Verwaltung von Seiten zuständig ist und nicht auf die Objekte der Seiten zugreift. Optimierte indexbasierte Suchalgorithmen und Pufferung der Speicherseiten durch den Client versuchen, die Netzbelastung dennoch gering zu halten, da im günstigsten Fall nur der Index, im ungünstigsten Fall ein großer Teil der Daten zum Client transportiert werden muß.

Queries werden unter Verwendung von C++-Syntax formuliert und als Nachricht an ObjectStore–Container–Objekte gerichtet. Die Ausdrücke können vorcompiliert werden oder zur Laufzeit dynamisch in Form eines Strings erstellt werden. Query-Ausdrücke dürfen geschachtelt sein. Sie dürfen keine Variablen enthalten — variable Werte werden bei dynamischer Erzeugung des Query–Ausdrucks eingetragen — sowie keine Funktionsaufrufe.

5.1.3 Mehrbenutzerbetrieb

Sperren

Das feinste sperrbare Granulat ist die Speicherseite. Das liegt daran, daß der Server die Sperren verwaltet. Um Blockierungen zu vermeiden, falls Objekte auf der gleichen Seite liegen, die von unterschiedlichen Anwendungen benötigt werden, ist es ratsam, solche Objekte unterschiedlichen Segmenten zuzuordnen.

Implizites Sperren: Eine Seite, die von einer Anwendung gelesen wird, ist für andere zum Schreiben gesperrt. Eine Seite, auf die geschrieben wurde, steht anderen Anwendungen bis zum Ende der Transaktion auch zum Lesen nicht mehr zur Verfügung.

Sperren können auch explizit angefordert werden.

Transaktionen

Der Zugriff auf persistente Objekte ist nur innerhalb von Transaktionen möglich. Bei Beginn einer Transaktion wird ein Speicherbereich für die Abbildung der Datenbankspeicherseiten (Memory Mapping) reserviert. Über die Einstiegspunkte muß zu den gewünschten Objekten navigiert werden. Bei Ausführung der Transaktion werden die als geändert markierten Seiten auf die Datenbank übertragen und der Speicher freigegeben. Zeiger auf persistente Objekte sind nach Ausführung der Transaktion nicht mehr gültig, da sie, wie in C++ üblich, nur eine Spei-

cheradresse tragen. Ein spezieller Verweistyp, `os_Reference`, behält die Gültigkeit auch in der nächsten Transaktion und kann auf Objekte anderer Datenbanken verweisen.

Transaktionen können geschachtelt werden. Bei Beginn einer Transaktion kann angegeben werden, ob nur gelesen oder auch geändert wird.

Zugriffsbeschränkungen und Autorisierung

Datenbanken werden als Dateien im Dateisystem des Servers realisiert, wobei zu einem Datenbank–Schema mehrere Datenbanken existieren können. Neben dem Dateisystem des Servers kann ObjectStore auch ein eigenes Dateisystem, das auf einer separaten Partition angelegt wird, verwalten, das sogenannte *Raw Filesystem (raw fs)*.

Die **Zugriffsbeschränkung** ist nur im Rahmen des unterliegenden Dateisystems möglich. Im ersten Fall kann der Zugriff nur für die ganze Datenbank beschränkt werden, im zweiten Fall, bei Verwendung des Raw Filesystem, ist das Segment das feinste Granulat. Hier ist eine Zugriffsbeschränkung auf Segmentebene möglich, allerdings gehört die Datenbank (und damit das Recht zur Vergabe von Zugriffsrechten) wiederum nur einem Benutzer.

Die **Autorisierung** erfolgt über das Betriebssystem des Servers. Ist der Client bereits über das Network File System (NFS) mit dem Server verbunden und hat die Erlaubnis, Dateien auf dem Server abzulegen, oder gewährt der Server dem Client den Zugang ohne Autorisation (Verwendung von `.rhosts`–Einträgen), befinden sich die beiden Rechner in einem „Vertrauensverhältnis“, das bereits vorher aufgebaut wurde. Ist das nicht der Fall, muß die Autorisation über Login und Paßwort bei jeder ObjectStore–Sitzung entweder interaktiv oder über eine Callback–Routine erfolgen. In letztem Fall müssen Login und Paßwort für den Server–Rechner als Strings von einer Prozedur bereitgestellt werden.

Schema Keys: Zusätzlich zum Schutzmechanismus des Dateisystems können Datenbanken und ihre Schema–Informationen mit einem Code zugangsgeschützt werden, dem sogenannten Schema Key.

lange Transaktionen, Check-In/Out, Objektversionen

Es ist möglich, Objekte in einen *Workspace* auszulagern, dort zu bearbeiten und später wieder auf die Datenbank zu übertragen [OSUser 4, S.335ff]. Das Auslagern (Check-Out) erzeugt eine neue Version der Objekte. Das Zurückschreiben (Check-In) ersetzt die alte Version auf der Datenbank durch die neue. Erfolgen Check-Outs der gleichen Objekte von verschiedenen Anwendungen aus, verzweigt sich der Versionsgraph und es werden eigene Versionszweige angelegt. Versionen aus verschiedenen Zweigen können wieder zusammengeführt werden.

Ereignismeldungen

ObjectStore benutzt den C++ Exception-Mechanismus, um auf Ausnahme- und Fehlersituationen zu reagieren. Da einige Compiler–Hersteller Exceptions nicht implementiert haben, bietet ObjectStore auch eine eigene Exception–Behandlung an.

Ereignismeldungen des Servers an die Anwendung werden nicht unterstützt; Anwendungen können aber untereinander Ereignismeldungen nach dem Publish/Subscribe–Prinzip veröffentlichen und abonnieren, die über den Server verteilt werden.

5.1.4 Heterogene Systeme

Sprachschnittstellen

ObjectStore bietet folgende Sprachschnittstellen an:

Objektzugriff: C++, Smalltalk, Java

Datenzugriff: C, ODBC

Smalltalk: In der bisherigen Version wurde die virtuelle Smalltalk–Maschine durch eine modifizierte Version ersetzt. Die Objekte waren mit denen einer C++-Anwendung nicht verträglich. In der neuen Version wird auf C++-Objekte zugegriffen, die von Smalltalk aus über die C-Schnittstelle erreicht werden.

Die **Java**–Schnittstelle befindet sich momentan noch im Beta-Stadium; die Entwicklung führt zu einem generischen Objektmodell hin.

Die **C**–Schnittstelle bietet die Möglichkeit, auf C++-Objekte der Objektdatenbank zuzugreifen: Auf Client–Seite wird ein ObjectStore–Zugriffs–Akteur angesprochen, der die entsprechenden Seiten lädt und die darauf befindlichen Objekte für den Datenzugriff zur Verfügung stellt.

Im Labor wurde nur die C++-Schnittstelle untersucht.

Unterschiedliche Compiler, Betriebssysteme oder Plattformen

Der ObjectStore Server speichert Seiten ab, auf denen die Objekte so abgelegt sind, wie sie im Speicher der Anwendung erscheinen. Aus diesem Grund ist die Nutzung der gleichen Objektdaten durch Anwendungen auf verschiedenen Client–Plattformen ein Problem für ObjectStore. Selbst wenn der gleiche Rechnertyp und das gleiche Betriebssystem verwendet wird, können Compiler verschiedener Hersteller die Objektstruktur verschieden realisieren.

Der Schema–Compiler kennt die Neutralize–Funktion. Hierbei wird aus vorbestimmten Mengen verträglicher Plattformen eine passende Menge bestimmt und der Quelltext daraufhin untersucht, ob die Struktur der Objekte verträglich ist. Ist das nicht der Fall, gibt der „Neutralisator“ Hinweise zur Änderung des Quelltextes. Wird einem bestehenden System ein Client auf einer Plattform hinzugefügt, die nicht in der verwendeten Verträglichkeitsmenge des Schema–Compilers auftaucht, kann er ohne Schema–Evolution der Datenbank nicht auf die Daten zugreifen.

Bei übereinstimmender Objektstruktur erfolgt eventuell eine Konvertierung von Integer- und Fließkommazahlen durch den Seitenverwalter auf dem Client.

Ein weiteres Problem ergibt sich mit der Autorisation: Für gleichartige Rechner sind „Vertrauens–Netzwerke“ (NFS, `.rhosts`, siehe oben) kein Problem; bei verschiedenen Plattformen (z.B. Windows NT und Unix) hat man hier mit Schwierigkeiten zu rechnen.

ObjectStore-Systeme sind für eine ganze Reihe an Plattformen erhältlich. Neben einigen Unix-Plattformen werden auch verschiedene PC-Betriebssysteme unterstützt. An Netzwerkprotokollen können neben TCP/IP auch PC–Netzwerkprotokolle eingesetzt werden.

Pro Betriebssystem wird normalerweise nur ein bestimmter Compiler, meist derjenige des Betriebssystemherstellers, unterstützt. Der Einsatz beliebiger Compiler ist schwierig, da ObjectStore die Objektstruktur im Speicher kennen muß, die von Compiler zu Compiler verschieden ist.

5.1.5 Skalierbarkeit

Verweise zwischen Objekten verschiedener Datenbanken, Multi-Server-Betrieb

Eine ObjectStore Anwendung kann innerhalb einer Transaktion mehrere Datenbanken auf verschiedenen Servern ansprechen. Beim Commit wird ein Two–Phase–Commit–Protokoll ausgeführt.

Innerhalb einer Transaktion können nur soviele Objekte bearbeitet werden, wie in das Speicherfenster im virtuellen Adreßraum passen — das Fenster ist standardmäßig 1,5 GB groß. Es ist möglich, ein größeres Fenster anzufordern, dessen maximale Größe durch den virtuellen Adreßraum von 4 GB bei 32–Bit–Maschinen beschränkt ist.

Verweise auf Objekte anderer ObjectStore Datenbanken auf dem gleichen oder auf anderen Servern sind möglich. Auf jedem Rechner mit ObjectStore–Datenbank–Dateien läuft normalerweise ein ObjectStore Server.

Netzwerkbelastung, Server-Queries

Seiten werden auf Seite des Clients gepuffert. Das Lösen von Sperren auf Seiten durch den Client erfolgt nicht unmittelbar, sondern erst auf Anfrage des Servers (Lazy Release). Der Server kennt die Verteilung der Seiten auf die Puffer der Clients. Wird eine Seite angefordert, die im sich Puffer einer anderen Anwendung befindet, wird der dortige *Cache Manager* gefragt, ob die Seite noch in Gebrauch ist. Wenn nicht, wird sie aus dem Puffer gelöscht und steht damit wieder zur Verfügung. Der Vorteil daran ist, daß Zugriffe auf die gleichen Seiten aus verschiedenen Transaktionen heraus nicht jedesmal das Übertragen der Seiten erfordern. Der *Cache Manager* ist ein Prozeß auf dem Client–Rechner, der nur für die Rückfragen des Servers zuständig ist und auf den Seitenpuffer der Anwendungen zugreifen kann [OSTech 3, 18].

Queries werden vom Datenbankverwalter auf Seite des Clients ausgeführt, siehe Seite 30. Die Netzbelastung kann bei großen Containern zum Problem werden, wenn Indizes nicht optimal eingesetzt werden können. Zwar kann ein Teil der Anwendung, der für Queries zuständig ist, als Client–Anwendung auf dem Server–Rechner abgewickelt werden, jedoch erfordert diese Lösung ein eigenes Protokoll zwischen den Teilen der Anwendung (z.B. Corba).

Es gibt Systeme, in denen es von Vorteil ist, daß der Server von den Anwendungen wenig belastet wird, da die Client-Rechner genug Rechenleistung bieten und das Netzwerk mit dem Datenaufkommen zurechtkommt.

Verfügbarkeit und Sicherheit

Verklemmungen (Deadlock–Situationen) durch Sperren können vom Server erkannt und aufgelöst werden.

Es gibt in C++ und bei ObjectStore keine Garbage Collection. „Verlorene“ Objekte, also solche, die nicht mehr durch Verweise erreichbar sind, können nicht im Betrieb aus der Datenbank entfernt werden. Diese und andere Aktionen wie Schema–Evolution müssen offline ausgeführt werden. Die Datensicherung durch Backups kann dagegen online durchgeführt werden.

5.1.6 Administration

Die Administration erfolgt durch Kommandozeilenwerkzeuge, die zum Teil nur auf dem Server ausgeführt werden können. Administration erfordert teilweise Systemrechte auf dem Server–Rechner, zumindest die Benutzerverwaltung.

Benutzerverwaltung: Es wird keine eigene angeboten, sondern die des Server–Betriebssystems benutzt. Das hat zur Folge, daß das Einrichten neuer Benutzer für die Datenbank automatisch deren Einrichtung für das Server–System beinhaltet, wodurch der DB–Administrator entsprechende Rechte für das Betriebssystem benötigt.

5.1.7 Entwicklung mit ObjectStore

Integration in Entwicklungsumgebungen

Mit ObjectStore werden neben einem Browser für den Datenbestand mit grafischer Benutzerschnittstelle nur Kommandozeilenwerkzeuge mitgeliefert. Unter erhöhtem Aufwand ist eine Integration dieser Werkzeuge in Entwicklungsumgebungen möglich, die zum Teil von Fremdherstellern übernommen wird.

Versionen von Klassen

Es kann von einer Klasse nur eine Version auf der Datenbank geben. Die Versionsverwaltung betrifft Objekte und Objekt–Verbunde, den sogenannten Configurations, im Zusammenhang mit langen Transaktionen, siehe Seite 31.

Schema–Evolution und Objektmigration

Werden Änderungen an einer Klasse vorgenommen, die die Datenstruktur der Exemplare verändern, kann auf die bereits gespeicherten Objekte nicht mehr zugegriffen werden. Es ist nötig, den Objektbestand auf die neueste Version zu bringen. Bei der Objektmigration (bei ObjectStore Schema–Evolution genannt) wird die Datenbank offline nach Objekten der alten Version durchforstet. Einige Änderungen können automatisch vorgenommen werden, andere, wie zum Beispiel das Vorbelegen eines neu hinzugekommenen Attributs, müssen von Routinen erledigt werden, die der Entwickler schreiben muß.

5.1.8 Einschätzung

ObjectStore ist der schnellste persistente Objektspeicher auf dem Markt. Seine Marktpräsenz verdankte das Produkt anfangs vor allem CAD–Anwendungen, die genau diese Eigenschaft benötigen. Mittlerweile ist der wichtigste Einsatzbereich für Objektdatenbanken die Telekommunikation, gefolgt von Finanzmarkt und Internet–Diensten. Falls es sich um homogene Rechner–Netzwerke handelt, am besten zu Clustern verbunden, sind Client–Server–Systeme mit ObjectStore gut zu realisieren.

Probleme tauchen bei heterogenen Systemen auf. Das betrifft einerseits die Autorisierung der Clients beim Server, andererseits die Austauschbarkeit der Daten, falls sich die Plattformen unterscheiden. Object Design plant für neue Versionen und Produkte wie die Java–Anbindung die Einführung eines generischen Objektformats.

Falls große Datenmengen durchsucht werden müssen und der Zugriff nicht navigierend ist, sondern mit dem Durchsuchen großer Container verbunden ist, kann ObjectStore im Nachteil sein, da diese Operationen im Client ausgeführt werden und unter Umständen eine größere Netzbelastung verursachen.

5.2 Versant

Versant wurde 1988 von Entwicklern relationaler Datenbanken mit der Zielrichtung gegründet, verteilte Objektdatenbanksysteme mit hoher Verfügbarkeit zu entwickeln.

5.2.1 Einordnung

Versant tritt sowohl als persistenter Objektspeicher (siehe 3.3.2) als auch als objektorientierte Datenbank auf (siehe 3.3.1), bietet aber keine aktive Datenbank an. Die Sprachschnittstelle benutzt den Mechanismus der Geändert–Markierung (siehe Abschnitt 3.2.3).

Besondere Merkmale sind:

- Eindeutiger Identifikator eines Objekts (Objekt–ID) unabhängig von Datenbank und Server; eine Objekt–ID wird nur einmal verwendet
- Generisches Objektmodell: In der Datenbank werden Objekte in einer Repräsentation gespeichert, die unabhängig von Plattform und Programmiersprache ist.

5.2.2 Technik

Architektur

Abbildung 5.3 zeigt die Versant–Architektur, die vom Hersteller „VSDA — Versant Scalable Distributed Architecture“ genannt wird.

Auf der linken Seite ist die Anwendung mit ihrem Objektspeicher zu sehen. Der Versant Manager auf Seite des Clients stellt der Anwendung die persistenten Objekte zur Verfügung und puffert sie in einem eigenen Speicherbereich, dem Objektpuffer.

Die rechte Seite zeigt die Versant–Server–Prozesse, die gemeinsam auf Massenspeicher und Seitenpuffer zugreifen. Für jede Datenbanksitzung wird ein Versant–Server–Prozeß gestartet. Die Koordination der Server–Prozesse erfolgt über Shared Memory. Im Seitenpuffer werden Seiten des Massenspeichers zur Durchsatzerhöhung gepuffert.

Versant Manager: Der *Versant Manager* ist Partner der Anwendung auf Client–Seite [VConcept 40, 8-9]. Seine Aufgaben sind folgende:

- Bereitstellung persistenter Objekte für den Zugriff durch die Anwendung
- Konvertierung des generischen Objektformats der Datenbank in das auf dem Client unter der Sprache übliche Format
- Aufbau der Verbindung zu den Datenbank-Servern, Verwaltung von Datenbank-Sitzungen

- Koordinierung verteilter Transaktionen nach dem „Two–phase Commit Protocol“
- Weiterleiten von Suchanfragen (Queries) an die betroffenen Datenbank–Server
- Verwaltung langer Transaktionen und versionierter Objekte
- Pufferung persistenter Objekte im Objektpuffer (Object Cache)
- Verwaltung von DB-Schemas: Erzeugung und Manipulation von Versant–Klassenobjekten

Der Versant Manager wird mit den von der Sprachschnittstelle zur Verfügung gestellten Mitteln von der Anwendung angesprochen. Technisch gesehen wird der Versant Manager nach dem Übersetzen des Programms beim Binden der Anwendung hinzugefügt.

Objektpuffer: Im Speicherbereich der Anwendung werden ein Puffer für die Objekte, auf die in der aktuellen Transaktion zugegriffen wird, sowie die zugehörigen Verwaltungdaten wie Prozeßinformation, Datenbankverbindungen und Transaktionsdaten verwaltet [VConcept 40, 8-2]. Der Objektpuffer ist auch der Speicherbereich, in dem persistente Objekte angelegt werden. Nach einer Transaktion wird der Puffer geleert, wobei alle als geändert markierten Objekte in die Datenbank gespeichert werden. Der Verbleib von Objekten im Puffer kann aber erzwungen werden (Pinning).

Im Puffer befindet sich des weiteren die Zuordnungstabelle zwischen Zeigern im transienten Bereich und den logischen Objektidentifikatoren (loid), über die auch die zugehörige Datenbank zu ermitteln ist. In der Tabelle sind in sogenannten *CODs* — Cached Object Descriptors

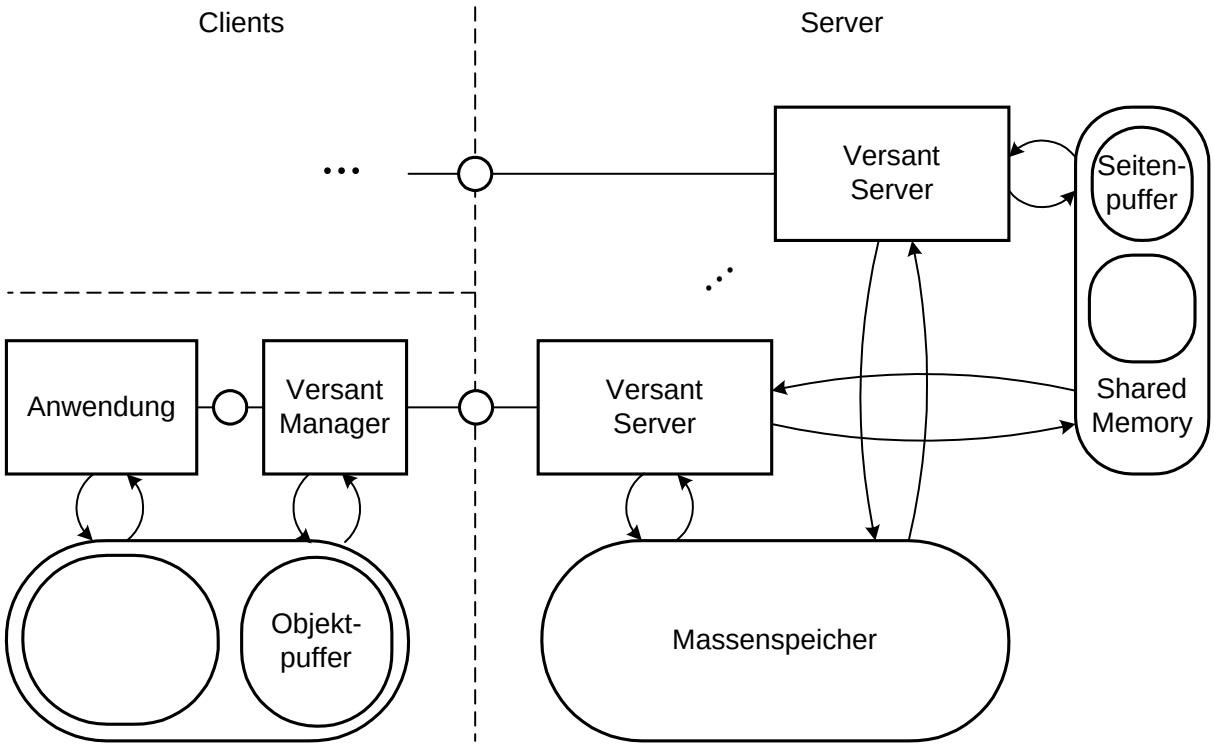

Abbildung 5.3: Versant Architektur

— Informationen zum Status eines Objekts abgelegt. Es handelt sich dabei um folgende Zustandsinformationen:

- Persistenz:
  - transient
  - persistent, aber nicht im Speicher der Anwendung
  - persistent, Kopie im Speicher der Anwendung
  - persistent, aber noch nicht in der Datenbank — es wurde in der Transaktion erzeugt
  - gelöscht, aber noch auf der Datenbank — es wurde in der Transaktion gelöscht
- Geändert–Markierung
- Information über Sperrung, Art der Sperre (Lock Type)
- Pinning (Verbleib im Puffer erzwungen)

Die Anwendung hat transparenten Zugriff auf die Objektdatenbank, muß sich daher normalerweise nicht um den Objektpuffer kümmern.

Versant Server: Für jede Sitzung wird auf dem Server, auf dem sich die Datenbank befindet, ein *Versant–Server*–Prozeß gestartet [VConcept 40, 8-10]. Es kann pro Datenbank mehrere Versant–Server–Prozesse geben. Deren Aufgaben sind:

- Bereitstellung persistenter Objekte für den Versant Manager durch Zugriff auf die Massenspeicher und den Seitenpuffer.
- Ausführung von Anfragen (Queries) der Clients.
- Verwaltung von Transaktionen, Sperren und Indizes von Containern
- Kapselung des unterliegenden Betriebssystems des Server–Rechners für den Versant Manager: Dateiverwaltung und Autorisation
- Verwaltung von Massenspeicher und Shared Memory, insbesondere Seitenpuffer
- Verwaltung von Dateien für Log- und Wiederherstellungsinformationen (Recovery)

Shared Memory: Falls mehrere Versant Server auf eine Datenbank zugreifen, erfolgt die Kommunikation zwischen ihnen über Shared Memory. Beim Start des Datenbanksystems wird automatisch ein Seitenpuffer erzeugt. Hier werden die Seiten vom Massenspeicher (Disk Pages) zwischengespeichert. Der Puffer befindet sich im Shared Memory, damit alle Server–Prozesse auf den Puffer zugreifen können.

In Tabellen wird gespeichert, welche Objekte auf welchen Seiten sind und welche Anwendungen sie gesperrt haben. Auch Log Files werden gepuffert. Bei Änderung werden gepufferte Seiten markiert, so daß nur so viel wie tatsächlich nötig auf den Massenspeicher zurückgeschrieben werden muß. Der Puffer wird bei Sitzungsende oder Halt der Datenbank geleert; die Leerung kann aber auch explizit angestoßen werden.

Globale Datenbanken: Globale Versant–Datenbanken bestehen aus einem Netzwerk von Datenbanken, die auf verschiedenen Server–Rechnern verteilt sind und wie eine einzige Datenbank angesprochen werden können. Auf jedem Server–Rechner können mehrere Datenbanken gleichzeitig aktiv sein. Von einem Datenbank–Schema können mehrere Datenbanken erzeugt werden.

Der Zugriff auf globale Datenbanken ist für Anwendungen transparent. Verweise zwischen Objekten können über Datenbank- und Servergrenzen hinausgehen. Clients können auch private Datenbanken, zu denen nur sie selbst Zugriff haben, an beliebigen Stellen im Server–Netz anlegen.

Die Kommunikation zwischen Client und Server erfolgt über eine eigene Kommunikationsschicht, dem *Versant Network Layer*, welche das unterliegende Netzwerkprotokoll kapselt [VConcept 40, 8-10].

Persistente Objekte

Objektmodell: Versant verwendet intern ein generisches Objektmodell, auf welches die verschiedenen Sprachschnittstellen abgebildet werden. Es handelt sich um ein passives Modell, das heißt, es sind keine Methoden speicherbar, nur die Attribute des Objekts.

C++-Sprachschnittstelle: Die C++-Sprachschnittstelle bietet drei wichtige Klassen an, über deren Exemplare das Datenbanksystem angesprochen wird. Die Klassen und ihre Funktionen sind:

`PDOM` (Persistent Distributed Object Manager):
Klasse des Datenbanksitzungsverwalters auf Seite des Clients. Kapselung der grundlegenden Datenbankoperationen wie Verbindungsaufbau zu einer Datenbank, Start einer Sitzung, Transaktionskommandos.
Es gibt in einer Versant–Anwendung genau ein Exemplar von `PDOM`.

`PClass` (Persistent Class):
Klasse der Laufzeit–Typinformationsobjekte persistenter Klassen auf Seite des Clients. Pro Klasse existiert ein Exemplar von `PClass`. Seine Aufgaben sind:

- Bereitstellung der Typinformation
- Bearbeitung von Anfragen, die die Menge aller Exemplare der Klasse (Class Extent) betreffen
- Check-Out von Exemplaren der Klasse
- Erzeugung von Indizes für den Class Extent.

`PObject` (Persistent Object):
Alle persistenten Objekte müssen von dieser Klasse abstammen. Sie kapselt das Wissen über persistente Objekte:
Erzeugen und Löschen, Kopieren und Verschieben, Statusabfrage und -änderung, Kenntnis des Laufzeit–Typinformationsobjekts der Klasse `PClass`.

Erzeugung und Speicherung: Ein Objekt wird persistent durch einen speziellen `new`–Operator erzeugt, wenn seine Klasse von der Versant–Persistenzklasse `PObject` geerbt hat. Ein Objekt kann nicht nachträglich persistent werden.

Damit geänderte Objekte zurückgeschrieben werden, müssen sie markiert werden (Mark Dirty). Die Markierungsoperationen müssen in den Zugriffsmethoden der Objekte eingefügt werden, siehe Abschnitt 3.2.3.

Objekt–ID: Die Identifikation persistenter Objekte erfolgt über den logischen Objektidentifikator (loid), der sich in der gesamten Lebensdauer des Objekts nicht ändert, auch nicht bei Verschieben in eine andere Datenbank. Diese Technik ist nützlich für persistente Verweise und transparente Verschiebung von Objekten über Datenbank- und Servergrenzen hinweg. [VRef 40, 3-3].

Verweise: Ein Objekt kann Verbindungen zu anderen haben. Versant bietet eigene C++-Objekt–Referenzierungsklassen an, *Link* (*loid* eines Objekts) und *LinkVstr*, Klassen von Containern von Links. Das Verfolgen eines Links bewirkt das Laden des Objekts, falls es sich nicht bereits im Speicher befindet. Links können getypt oder typfrei sein. Auch zweiseitige Verbindungen (*Bidirectional Links)* sind möglich. Versant hebt hervor, daß die Vorteile von Links größer sind als die Nachteile durch den Verzicht auf C++-Zeigern ([VRef 40, 3-2] und [Manola 94, S.73]).

Klassenschnittstelle: Bei der Entwicklung einer Anwendung unter Verwendung einer Versant–Sprachschnittstelle werden persistente Klassen definiert, die bei Übersetzen des Quelltextes im Datenbankschema gespeichert werden (siehe Schema–Compilierung in 3.4). Wird das Schema zum Datenbankverwalter übertragen, erzeugt dieser Klassenbeschreibungsobjekte für jede Klasse und speichert mit ihnen die Klassendefinitionen in der Datenbank [VRef 40, 15-1]. Bei Verwendung von C++ werden bei Start der Anwendung auf Seite des Clients zusätzlich Laufzeit–Typ–Identifikatoren der Klasse `PClass` erzeugt.

Jedes persistente Objekt und sein Klassenbeschreibungsobjekt sind immer einer Datenbank zugeordnet, können aber zwischen Datenbanken wechseln. Das Wechseln der Datenbank zieht das Kopieren des zugehörigen Klassenbeschreibungsobjekts nach sich, so daß dieses auf allen Datenbanken zu finden ist, auf denen ein Exemplar der Klasse gespeichert wurde. Dadurch ist es möglich, auch ohne Zugriff auf die Ausgangsdatenbank auf ein Objekt zuzugreifen, das die Datenbank gewechselt hat. Falls auf zwei Datenbanken verschiedene Klassen gleichen Namens existieren, wird der Wechsel verhindert.

Klassenbeschreibungsobjekte können unabhängig von der Schema–Compilierung zur Laufzeit in der Datenbank erzeugt werden. Auf diese Weise können nicht objektorientierte Anwendungen Klassen in der Datenbank anlegen und manipulieren[VRef 40, 16-1].

Queries

Eine Suchanfrage (Query) wird an den Datenbankverwalter gerichtet und liefert als Ergebnis einen Container mit Verweisen auf die Objekte zurück, die das Suchprädikat erfüllen. Diese werden erst bei Referenzierung einzeln geladen [VConcept 40, 5-1].

Als Suchausdrücke sind neben Anfragen an Container Pfadausdrücke zur Navigation über Verweise möglich. *Query Starting Point Objects* sind Objekte, die als Einstiegspunkt dienen und von denen aus die Suche erfolgt. Query–Ausdrücke können zur Laufzeit als String zusammengebaut werden.

Eine Suchanfrage wird immer an einen Versant Server gerichtet und bezieht sich auf eine Datenbank. Betrifft die Suche mehrere Datenbanken, muß für jede eine Suchanfrage an ihren Server gerichtet werden. Die Ergebnismengen können zu einer Menge vereinigt werden.

5.2.3 Mehrbenutzerbetrieb

Sperren

Versant Locks haben folgende Eigenschaften [VConcept 40, 4-1]:

- Das Sperren erfolgt auf Objektebene
- Dauerhafte Sperren (Persistent Locks) für die Realisierung langer Transaktionen sind möglich
- Alle Arten von Objekten, auch versionierte, können gesperrt werden
- Das Sperren eines Klassenobjekts veranlaßt das Sperren aller Klassenexemplare
- Im Normalfall wird implizit gesperrt: Der Zugriff auf ein Objekt veranlaßt das Setzen einer Sperre. Der lesende Zugriff impliziert eine Schreibsperre, der modifizierende Zugriff eine Lese- und Schreibsperre.

Es ist möglich, neue Typen von Sperren, die sich auf Versant Sperren zurückführen lassen, zu definieren [VConcept 40, 15-14]. Auf diese Weise kann die Behandlung konkurrierender Zugriffe an spezielle Situationen angepaßt werden.

Transaktionen

Versant unterstützt die optimistische und die pessimistische Transaktionsstrategie. Voreinstellung ist die pessimistische. Deadlock–Situationen können erkannt und aufgelöst werden. Transaktionen dürfen geschachtelt werden [VConcept 40, 15-1].

Verteilte Transaktionen, also solche, die auf verschiedene Datenbanken verteilte Objekte betreffen, werden automatisch durch ein Two–phase Commit durchgeführt; so wird sichergestellt, daß die Transaktion auf allen beteiligten Datenbanken erfolgreich ausgeführt wurde.

Zugriffsbeschränkung und Autorisierung

Die Benutzerrechte beziehen sich auf ganze Datenbankdateien. Für eine Datenbank kann der Besitzer Zugriffsrechte an andere Benutzer vergeben, die in sogenannten Database Roles festgelegt werden. Die Liste mit den Rollen der Benutzer wird in der Datenbank selbst abgelegt. Der Mechanismus zum Zugriffsschutz wird damit vom Dateisystem entkoppelt, wobei nur die Datenbank als ganzes geschützt werden kann. Der Schutz einzelner Objekte wird nicht unterstützt, sondern liegt in der Verantwortung des Programmierers.

Benutzerverwaltung: Die Benutzer werden durch das Betriebssystem des Server–Rechners verwaltet. Auch die Autorisierung der Benutzer erfolgt über dessen Mechanismen. Ab Versant 4.2 ist eine benutzerdefinierte Autorisation, beispielsweise die Verwendung des Kerberos–Protokolls, möglich.

lange Transaktionen, Check–In/Out, Objektversionen

Objekte können von einer globalen in eine private Datenbank kopiert werden (Check–Out). Während des Check–Out–Prozesses können verschiedenartige persistente Sperren gesetzt werden [VConcept 40, 3-7].

Wird ein Objekt als *versioned* spezifiziert, kann der Check–Out–Prozeß ohne Sperren erfolgen [VConcept 40, 7-1]. Beim Check–In wird es zum „Kind“ (Child Object) der Originalversion, eine neue Version des Objekts. Bei Check–Ins von mehreren Anwendungen bezüglich eines Objekts werden mehrere „Kinder“ erzeugt; es dürfen also parallel verschiedene Versionen existieren. Durch Angabe der Versionskennung sind alle Versionen eines Objekts erreichbar.

Ereignismeldungen

Watch/Notify: Ein Server kann Clients benachrichtigen, und zwar bei Erzeugen, Löschen oder Ändern von Objekten [VConcept 40, 11-1]. Zu diesem Zweck wird der *Signal*–Mechanismus des Betriebssystems verwendet.

5.2.4 Heterogene Systeme

Sprachschnittstellen

Durch das generische Objektmodell besitzt Versant die Fähigkeit, Anwendungen verschiedener Programmiersprache den Zugriff auf die gleichen persistenten Objekte zu ermöglichen.

Objektzugriff: C++, Smalltalk, Java

Datenzugriff: C, SQL, ODBC

Die C–Schnittstelle bietet mit Ausnahme der Übertragung von Objekten die volle Funktionalität. Es können auf dem Server Klassen definiert und geändert, Objekte erzeugt und gelöscht, Attribute gelesen und modifiziert werden.

Unterschiedliche Compiler, Betriebssysteme oder Plattformen

Der Versant–Server verschickt Objekte im generischen Objektformat. Auf Client–Seite werden diese an die jeweilige Plattform und die Programmiersprache angepaßt (Little/Big Endian, etc.). Auf diese Weise kommt Versant gut mit heterogenen Systemen zurecht und ist auch nicht von bestimmten Compilern abhängig.

Heterogene Netzwerke werden durch den *Versant Network Layer* gekapselt [VConcept 40, 8-10].

5.2.5 Skalierbarkeit

Verweise zwischen Objekten verschiedener Datenbanken, Multi-Server-Betrieb

Versant unterstützt den Multi-Server-Betrieb. Objekte haben über alle Server und Datenbanken hinweg eine eindeutige ID (loid), die sie beibehalten, auch wenn sie von einer Datenbank in eine andere wechseln.

In der Versant–Objekt–ID *loid* ist die für das Objekt zuständige Datenbank eincodiert. Wird ein Objekt von einer Datenbank in eine andere verschoben (*Migration*, siehe [VConcept 40, 6-2]), bleibt ein *Forwarder* zurück, der alle Anfragen bezüglich des Objekts an seine neue Adresse weiterreicht. Bildlich gesehen verläßt das Objekt die Datenbank und stellt einen Nachsendeauftrag für die neue Adresse (die Zieldatenbank). Das Klassenbeschreibungsobjekt, welches Informationen über die Struktur der Exemplare der Klasse enthält, wird ebenfalls auf die Zieldatenbank übertragen. Auf diese Weise wird ein Objekt unabhängig vom aktuellen Ort erreicht. Beispiel: Die Telefonnummer von Gerhard Müller besteht aus Firmennummer und Durchwahl. Gerhard Müller hat die Firma gewechselt, er ist nun nicht mehr in der Kugellager AG, sondern in der Achsen GmbH. Er kann seine alte Telefonnummer behalten, da die Telefonzentrale der Kugellager AG alle Anrufe freundlicherweise an seine neue Adresse weiterleitet.

Netzwerkbelastung, Server–Queries

Die Pufferung auf Server- und Client–Seite vermindert die Netzwerkbelastung, genauso die Bearbeitung von Queries auf dem Server.

Verfügbarkeit und Sicherheit

Schema–Evolution und Backup sind online möglich. Datenbanken können gespiegelt werden, um die Verfügbarkeit bei Ausfall des Originals zu gewährleisten.
Hochgradig verfügbare Systeme sind möglich durch Einsatz des *Versant Fault Tolerant Server*.

5.2.6 Administration

Die Administration erfolgt über ein Programm mit grafischer Oberfläche auf dem Datenbank–Server, welches allerdings nicht für alle Plattformen erhältlich ist [VSystem 40, 7-1]. Für die Administration von einem anderen Rechner aus können nur die Kommandozeilenwerkzeuge benutzt werden.

Die Benutzerverwaltung muß vom Systemadministrator der Datenbank–Server–Rechner erledigt werden. Die Zugriffsrechte auf eine Datenbank können von ihrem Besitzer gesetzt werden.

5.2.7 Entwicklung mit Versant

Integration in Entwicklungsumgebungen:

Mit Versant werden nur die Kommandozeilenwerkzeuge mitgeliefert. Unter erhöhtem Aufwand ist eine Integration dieser Werkzeuge in Entwicklungsumgebungen möglich.

Die Schema–Compilierung ist von der Übertragung des Schemas zur Datenbank entkoppelt. Die Datenbank muß daher nicht online verfügbar sein, um die Anwendung compilieren zu können. Der Schema–Compiler durchsucht alle der Anwendung bekannten Klassen (rekursives Scannen der Header-Dateien). Aus diesem Grund ist eine Schema–Compilation recht umfangreich und erfordert einen entsprechend leistungsfähigen Entwicklungsrechner.

Versionen von Klassen

Es gibt bei Versant immer nur eine Version einer Klasse auf der Datenbank. Durch die dynamische Objektmigration (siehe unten) können zwar Exemplare von Klassen älterer Version existieren, die aber beim Zugriff auf den neuesten Stand gebracht werden.

Schema–Evolution und Objektmigration

Wird eine Klasse geändert, wirkt sich das auf das Datenbank–Schema aus. Die Änderung des Schemas kann bei laufender Datenbank erfolgen. Bei Zugriff auf Exemplare der geänderten Klasse werden diese auf den neuen Stand gebracht [VRef 40, 15-5]. Bei Änderungen an der Klasse, die eine sinnvolle automatische Konvertierung auf den neuesten Stand verhindern, können die Exemplare durch eigene Routinen auf den neuesten Stand gebracht werden, ohne ihre ID zu ändern, da Versant die Möglichkeit bietet, zur Laufzeit sowohl Objekte als auch Klassen auf der Datenbank anzulegen und zu ändern [VRef 40, 16-1].

5.2.8 Einschätzung

Versants Stärken liegen in heterogenen Client-Server-Systemen mit verteilten Datenbanken. Die ortsunabhängige Objekt–ID *loid* und das generische Objektmodell sind wichtige Konzepte

für diesen Bereich. Die Verwendung der Geändert–Markierung schafft ausreichende Transparenz beim Zugriff. Bemerkenswert ist auch die Betonung von Sicherheit und Verfügbarkeit.

Die Zugriffsbeschränkung auf Benutzerebene ist ein Schwachpunkt von Versant, da Zugriffsbeschränkungen nur für die ganze Datenbank gelten und die Einrichtung von Benutzern durch die Einrichtung von Accounts auf dem Server–Rechner erfolgt.

5.3 Poet

Die Firma Poet ist ein Hamburger Softwareunternehmen, das 1993 aus der Firma BKS ausgegründet wurde. Die Objektdatenbank Poet wurde für kleine Plattformen wie PCs unter DOS und Windows entwickelt und erfährt mit neuen Versionen Erweiterungen für große Systeme.

5.3.1 Einordnung

Poet ist eine Objektdatenbank, die das explizite Speichern (siehe Abschnitt 3.2.2) verwendet.

Besondere Merkmale sind:

- Unterstützung heterogener Systeme durch ein plattformunabhängiges C++-Objektmodell
- Gute Integration in den Desktop–Bereich durch Schnittstellen wie OLE.

5.3.2 Technik

Architektur

Durch den expliziten Aufruf von Poet–Methoden zum Laden und Speichern von Objekten werden größere Teile des Poet–Datenbankverwalters Bestandteil des Anwendungscodes. Beim Einsatz in einem Client–Server–System wird der Datenbankverwalter aufgeteilt in den Poet Server und den Poet Client, die folgende Aufgaben haben:

Poet Server:

- Gewährung des Zugriffs auf Poet–Datenbanken durch Poet–Anwendungen von anderen Rechnern
- Koordinierung konkurrierender Zugriffe auf eine Datenbank
- Ausführung von Suchanfragen bezüglich in der Datenbank gespeicherte Objekte
- Autorisation und Verwaltung der Zugriffsbeschränkungen
- Mitteilung von Ereignissen an Clients

Poet Client:

- Konvertierung von Objekten zwischen dem Poet–Format und dem Format der Anwendung
- Aufbau von Verbindungen zu Datenbanken, Sitzungsverwaltung

- Zugriff auf lokale Datenbanken wie beispielsweise einen *Workspace*; der Poet Client kann ebenso wie der Poet Server Objektdatenbanken verwalten, allerdings nur zum exklusiven Zugriff der Anwendung.
- Ausführen von Suchanfragen nach transienten, weiterleiten von Anfragen nach persistenten Objekten
- Registrierung (Anmeldung) und Aufruf von Callback–Routinen für die Ereignisbehandlung

Persistente Objekte

Erzeugung: Persistente Objekte müssen Exemplar einer Klasse sein, die mit dem Schlüsselwort `persistent` markiert wurde. Alle Objekte werden durch den normalen `new`–Operator erzeugt. Objekte werden „nachträglich“ persistent, wenn sie durch `Store` explizit gepeichert werden, oder durch die Bekanntschaft mit einem anderen persistenten Objekt (welches also Verweise auf sie trägt); Voraussetzung dafür ist allerdings die Zugehörigkeit zu einer `persistent` Klasse. Die Eigenschaft der Persistenz kann ihnen „zu Lebzeiten“ wieder genommen werden, wenn sie in der Datenbank gelöscht werden, im Speicher der Anwendung aber noch existieren — siehe Abschnitt 3.1.2.

Objektmodell: Poet benutzt das C++-Objektmodell, wobei Methoden nicht gespeichert werden. Poet erweitert C++ um das Schlüsselwort `persistent`, um die Klassen zu kennzeichnen, deren Exemplare die Fähigkeit haben, persistent zu werden.

Objekt–ID und Verweise: Poet benutzt normale C++-Zeiger und spezielle Poet–Verweise, die *Ondemands*. Während Zeiger als Ortsangabe die Adresse im Speicher tragen, enthält ein Ondemand Verweis die Objekt–ID des persistenten Objekts in der Datenbank. Das Folgen eines C++-Zeigers zu einem persistenten Objekt ist nur erfolgreich, wenn sich dieses Objekt bereits im Speicher befindet. Aus diesem Grund werden beim Laden eines persistenten Objekts alle weiteren Objekte, auf die es mit Zeigern verweist, ebenfalls geladen. Die Information über die Zeiger und die damit verbundenen Klassen wird durch den Poet–Precompiler PTXX aus dem Quelltext geholt. *Ondemands* sind spezielle Verweise, die Objekte auf Poet–Objektdatenbanken identifizieren. Erst das Folgen eines Ondemand–Verweises bewirkt das Laden des Objekts in den Speicher.

Poet wandelt C++-Zeiger in eine Form um, die in der Datenbank gespeichert werden kann [PProg 30, 30]. Die Objekte, auf die verwiesen wird, werden ebenfalls gespeichert, so daß die Beziehungen korrekt in der Datenbank wiedergegeben werden können. Ebenso werden beim Laden eines Objekts alle Objekte mitgeladen, die Ziel von Zeigern sind. C++-Zeiger können daher auch bei persistenten Objekten direkt ausgewertet werden. Voraussetzung für das Abspeichern bisher transienter Objekte ist, daß diese Exemplare von Klassen sind, die mit `persistent` markiert wurden.

Wird ein Objekt von der Datenbank geladen und kann ein Objekt, auf das verwiesen wurde, nicht mehr in der Datenbank gefunden werden, weil es bereits gelöscht wurde, wird der

C++-Zeiger automatisch auf Null gesetzt und der Verweis beim Speichern in die Datenbank korrigiert. Eine andere Möglichkeit ist eine spezielle Art des Verweises, einer Abhängigkeitserklärung (*depend*). Wird ein Objekt gelöscht, werden alle anderen mitgelöscht, die davon abhängig sind.

Sprachschnittstelle: Persistente Klassen werden durch das Schlüsselwort `persistent` als solche gekennzeichnet. Der Präprozessor PTXX erweitert diese Klasse zum Teil durch Vererbung um die Methoden zur persistenten Speicherung und zur Annahme von Queries. Die Speicherung eines Objekts erfolgt explizit durch `Store`. Das Problem dabei ist, daß ein Kunde des Objekts nach Änderung für dessen Speicherung zuständig ist. Wird der `Store`–Aufruf dagegen in die Zugriffsmethoden des Objekts implementiert, wird bei jeder Änderung das Objekt zurückgeschrieben.

Vorteil am expliziten Speichern mit `Store` ist, daß die geänderten Objekte dem Server bekannt sind, so daß er in der gleichen Transaktion Queries darauf ausführen kann. Werden Objekte mit `Store` innerhalb einer Transaktion gespeichert, werden sie auf Serverseite gepuffert und bei Commit in die Datenbank gespeichert. Ab Poet 4.0 können Objekte alternativ auch auf Seite des Clients gepuffert werden (ähnlich Mark Dirty), wobei Queries Änderungen erst nach Ende der Transaktion berücksichtigen.

Die Speichertiefe ist wählbar; normalerweise werden zusammen mit einem Objekt alle Objekte gespeichert, auf die es mit Zeigern verweist, nicht aber die, die über Ondemands erreicht werden können (Dieser mit einem Objekt fest verbundene Teil des Objektnetzes wird bei GemStone transitive Hülle genannt). Der Automatismus kann für einzelne Verweise unterbunden werden, wenn zum Beispiel auf transiente Objekt verwiesen wird, die nicht gespeichert werden sollen. Genauso kann erzwungen werden, daß auch alle Objekte, auf die Ondemands verweisen, gespeichert werden.

Der Präprozessor *PTXX* führt die Schema–Compilierung durch. Seine Aufgaben sind [PProg 30, S.91]:

- Erzeugen oder Ändern des Datenbank–Schemas in einer eigenen Datenbank (*Class Dictionary*). Ist eine Klasse bereits vorhanden, wird die alte Version der Klasse beibehalten und eine neue Version angelegt.
- Erzeugen der Datenbank, falls noch nicht geschehen
- Generieren von Quelltexten, die in die Quelltexte der Anwendung eingebunden werden. Die Definition einer mit `persistent` markierten Klasse erfolgt in einer Datei, deren Namen die Endung `.hcd` trägt. Diese Datei wird vom Präprozessor verarbeitet, wobei mehrere neue Dateien entstehen. Die Implementierung der Klasse erfolgt in einer anderen Datei, die für den Präprozessor ohne Interesse ist. Wichtig ist aber, daß von PTXX generierte Quelltexte in die Implementierung eingebunden werden. Von PTXX generiert werden:

Class Factory (`*.cxx`):	Implementierung der Erzeugung und Verwaltung persistenter Exemplare im Speicher
Class Header (`*.hxx`):	Deklaration der persistenten Klasse in reinem C++

Class Factory Header (`*.ptx`): Deklaration der typsicheren klassenspezifischen Container und Ondemands sowie Query Routinen

Object Class Definition (`*.ocd`): Vor–compilierte Klassendeklaration. Sie wird benötigt, wenn die Schema–Compilation offline, also ohne Verbindung zur Datenbank geschehen soll.

Queries

Über Suchanfragen können Objekte mit bestimmten Attributbelegungen innerhalb einer Menge von Objekten derselben Klasse gefunden werden. Das Ergebnis ist ein Container von Objekten, dessen Inhalt unter Umständen sortiert ist. Suchanfragen werden an Container gerichtet, die mit Indizes versehen werden können, um die Suche zu beschleunigen.

Eine Suchanfrage durch ein Objekt repräsentiert, welches zur Laufzeit zusammengebaut wird. Dieses Query–Objekt wird an das Container–Objekt geschickt, auf dessen Elemente der Suchausdruck angewendet werden soll. Das Container–Objekt reicht den Suchausdruck an den Server durch, der die Suche ausführt. Dieser bietet je nach Wunsch die Ergebnismenge am Stück oder Element für Element an [PProg 30, S.189]. Ab Version 4.0 kann der Such-Ausdruck auch als OQL–Ausdruck in einem String formuliert werden.

5.3.3 Mehrbenutzerbetrieb

Sperren

Implizite Sperren (Transaktionssperren): Während einer Transaktion werden die Objekte gesperrt, die modifiziert wurden, bei denen also die `Store` oder `Delete` Methode aufgerufen wurde. Damit soll sichergestellt werden, daß korrekt zurückgeschrieben werden kann. Ein Objekt wird nicht automatisch gesperrt, wenn es während einer Transaktion von der Datenbank gelesen wurde. Transaktionssperren werden mit Ende der Transaktion aufgehoben.

Explizite Sperren: Sperren können für Objekte und Mengen von Objekten vergeben werden. Alle Objekte, die über Verweise in Beziehung mit einem Objekt stehen, können ebenfalls gesperrt werden [PProg 30, S.241]. Explizite Sperren sind unabhängig von Transaktionen, werden also nicht bei Ausführung oder Abbruch einer Transaktion gelöst. Bei den verschiedenen Arten von Sperren wird unterschieden, welche Garantien die Sperre für den Anforderer gibt und wie sie die übrigen Benutzer im Zugriff einschränkt [PProg 30, S.249].

Transaktionen

Implizite (System–Level) Transaktionen: Jede einzelne Schreiboperation ist eine vollständige Transaktion, um im Fehlerfall eine Wiederherstellung der Daten zu ermöglichen. Die System–Level–Transaktion ist notwendig, da die Speicherung eines persistenten Objekts

unter Umständen die Speicherung mehrerer Objekte nach sich zieht, von denen jede erfolgreich verlaufen muß.

Explizite Transaktionen: Transaktionen können explizit begonnen, committed oder abgebrochen werden. Alle Datenbankoperationen werden auf dem Server gepuffert, bis das Commit kommt. Verschachtelte Transaktionen (Nested Transactions) sind möglich. Über eine Funktion kann die aktuelle Transaktionstiefe abgefragt werden. Das Ändern und Löschen von Objekten innerhalb einer Transaktion führt zum Setzen von Transaktionssperren auf die betroffenen Objekte (siehe oben). Das Lesen eines Objekts innerhalb einer Transaktion bewirkt dagegen nicht das Setzen einer Sperre.

Die optimistische Transaktionsstrategie wird nicht unterstützt. Die Pessimistische kann nur mit Aufwand nachgebildet werden, da die Sperren nicht nur explizit gesetzt, sondern auch wieder explizit aufgehoben werden müssen.

Zugriffsbeschränkung und Autorisierung

Zugriffsbeschränkung: Eine benutzerbezogene Zugriffsbeschränkung ist möglich für alle Exemplare einer Klasse und deren Unterklassen sowie für Teilmengen, die durch Attributbelegung bestimmt werden. So ist es beispielsweise möglich, einem Benutzer nur die Personalakten–Objekte zugänglich zu machen, bei denen der Nachname mit 'A' bis 'G' anfängt. Rechte können für einzelne Benutzer und für Gruppen vergeben werden. Die Rechte sind: Lesen, Schreiben und Löschen.

Autorisierung: Die Autorisierung erfolgt durch den Poet–Server. Es ist nicht erforderlich, einen Account auf dem Server–Rechner zu haben. Die Benutzer- und Gruppenprofile und die Rechte werden in der jeweiligen Datenbank gespeichert.

lange Transaktionen, Check–In/Out, Objekt–Versionen

Um für längere Zeit Objekte in privatem Zugriff zu haben, kann ein privater Datenbankbereich auf dem Client erzeugt werden, der *Workspace*. Die Objekte existieren im Workspace als exakte Kopie, besitzen also die gleiche ID wie die Objekte der Datenbank. Jede Datenbank hat eine Liste der Workspaces, die zu ihr gehören. Nach einem Check–Out sind die Objekte in der Datenbank gesperrt.

Die Schema–Information zu den Klassen befindet sich im *Class Dictionary*. Es muß ebenfalls vom Server kopiert werden, um auf die Objekte des Workspace zugreifen zu können, falls der Server zwischenzeitlich nicht erreichbar ist (z.B. weil das Check–Out auf ein Notebook erfolgte). Werden neue Objekte in einem Workspace erzeugt, erhalten sie eine temporäre Objekt–ID; erst bei zurückspielen in die Datenbank erhalten sie eine feste ID.

Das Check–In verschiebt die Objekte aus dem Workspace in die Datenbank; `updateSource` bringt die Objekte der Datenbank auf den Stand des Workspace. Es gibt dabei normalerweise keine Konflikte, da die Objekte auf der Datenbank durch ein Check–Out gesperrt wurden. Alle Objekte des Workspace werden übertragen (Ab Version 4.0 ist ein partieller Check–In möglich).

Objekt–Versionen sind bei Poet nicht vorgesehen.

Ereignismeldung

Watch/Notify: Der Poet Server kann den Aufruf von Callback–Routinen der Anwendung bei Auftreten verschiedener Ereignisse bewirken. Die Anwendung muß sich vorher für bestimmte Ereignisse beim Poet Client angemeldet und die dazu passenden Callback–Routinen bekannt gemacht haben. Folgende Ereignisse können vom Server mitgeteilt werden [PProg 30, S.280]:

- Schreibzugriff auf Objekte,
- Änderung,
- Löschung,
- Sperrung und
- Erzeugung von Objekten

Die Ereignisse können sich beziehen auf

- Ein Objekt
- Ein Objekt und alle davon abhängigen (*depend*)
- Ein Objekt und alle davon referenzierten mit Ausnahme von *Ondemands*
- Ein Objekt und alle davon referenzierten

Im Poet–Programmierhandbuch wird eine interessante Anwendung des Watch/Notify–Mechanismus vorgestellt: Die Realisierung eines verteilten Objekts, eines Objekts, das von mehreren Anwendungen gleichzeitig benutzt und als ein immer konsistentes Objekt betrachtet werden kann, so wie es die Anschauung als Shared Memory Bereich zwischen den Rechnern nahelegt [PProg 30, S.292].

Progress Callbacks: Um bei langen Datenbankoperationen auf dem Server, wie zum Beispiel die Bearbeitung umfangreicher Suchanfragen, „auf dem laufenden“ gehalten zu werden, gibt es diese speziellen Ereignismitteilungen [PProg 30, S.276]. Drei verschiedene Callbacks werden aufgerufen:

1. Start der Operation
2. Ausführung der Operation. Dieser Callback wird periodisch aufgerufen, wobei ein Integer–Wert angibt, zu wieviel Prozent die Ausführung fortgeschritten ist. Die Einstellung erfolgt über eine Konfigurationsdatei zur jeweiligen Datenbank [PRef 30, S.42].
3. Ende der Operation

5.3.4 Heterogene Systeme

Sprachschnittstellen

Objektzugriff: C++, OLE 2.0 (Ab Poet 4.0 auch OLE Automation und Java)

Datenzugriff: ODBC, OQL, ab Poet 4.0 auch Visual Basic[1]

Standards: ODMG 93, ab Poet 4.0: STL (Standard Template Library)[2]

Unterschiedliche Compiler, Betriebssysteme oder Plattformen

Poet speichert C++-Objekte in einem plattformunabhängigen Format ab. Die `Store`- und `Get`–Methoden der persistenten Objekte führen die Wandlung durch. Da keine Betriebssystem–Interna benötigt werden, sind Poet–Clients und Server für viele Unix- und PC–Plattformen und Compiler erhältlich und unterstützen gängige Netzwerk–Protokolle.

Von Poet–Seite sind Objekt- und Quelltext–Austauschbarkeit zwischen verschiedenen Plattformen möglich.

5.3.5 Skalierbarkeit

Multi-Server-Betrieb, Verweise zwischen Objekten verschiedener Datenbanken

Der Betrieb mit mehreren Servern wird unterstützt. Poet deckt hinsichtlich Skalierbarkeit besonders nach unten einen weiten Bereich ab. Die Mindestanforderungen an einen Rechner zum Betrieb einer Poet-Datenbank sind klein; nicht einmal Multitasking-Fähigkeit ist zwingend nötig. Ab Version 4.0 ist der Poet-Server multithreaded und SMP–fähig (Symmetric Multi–Processor — Mehrprozessormaschinen) ausgestattet.

Verweise zwischen Objekten verschiedener Datenbanken (Cross–DB–References): *Ondemand References* können auf Objekte in anderen Datenbanken verweisen, selbst wenn diese von einem anderen Server verwaltet werden. In Version 3.0 sind Cross–DB–References nur erlaubt, wenn die Datenbanken das gleiche *Class Dictionary* benutzen [PProg 30, S.317]. In jeder Datenbank befindet sich eine Tabelle, in der Informationen zu den anderen Datenbanken abgelegt sind, auf deren Objekte über *Ondemands* verwiesen wird [PProg 30, S.322]. Eine Zeile der Tabelle enthält:

1. eine ID der Fremd-Datenbank, die als Bestandteil der Objekt–ID von Ondemands verwendet wird
2. den Namen des Rechners, auf dem der Server läuft

[1] Visual Basic von Microsoft ist eine Programmiersprache zur einfachen Entwicklung von Anwendungen unter MS Windows, die vorgefertigte Komponenten von Windows–Produkten verwenden kann.

[2] Die Standard Template Library ist ein Standard für C++-Container und die notwendigen Zugriffsmethoden (Im Sprachstandard von C++ werden keine Vorgaben für Container gemacht). Auch im ODMG–Standard für Sprachschnittstellen für Objektdatenbanken werden Konzepte für Containerklassen vorgestellt.

3. den Namen der Datenbank. Das kann ein logischer Name sein oder ein Pfadname. Logische Namen werden vom Server über eine Zuordnungsdatei aufgelöst.

Ein Ondemand–Verweis enthält die ID der Datenbank, die über die Tabelle entschlüsselt wird. Wird einem Ondemand–Verweis auf eine Fremd–Datenbank gefolgt, stellt der Client die Verbindung zur anderen Datenbank her. Wird eine Datenbank von einem auf einen anderen Server verschoben, werden die Einträge nicht automatisch mitgeführt. Aus diesem Grund ist die Benutzung logischer Namen für Datenbanken angebracht, da die Änderungen dann nur in einer Konfigurationsdatei nachgeführt werden müssen. Um mit den resultierenden Problemen zurechtzukommen, gibt es Link–Repair–Callbacks, die aufgerufen werden, wenn eine Fremd–Datenbank zur Laufzeit nicht gefunden werden kann [PProg 30, S.325].

Das Verschieben von Objekten zwischen Datenbanken wird unterstützt. Um Verweise korrekt wiederzugeben, wird an der alten Stelle ein Proxy–Objekt installiert (sozusagen ein Nachsendeauftrag).

Transaktionen und Queries beziehen sich auf eine Datenbank, selbst wenn über Ondemands Objekte anderer Datenbanken angesprochen werden.

Netzwerkbelastung, Server–Queries

Durch die explizite Speicherung kommt es zu einer erhöhten Netzbelastung. Bei jedem Aufruf von `Store` wird das Objekt und seine transitive Hülle (alle Objekte, auf die es mit Zeigern verweist) konvertiert und an den Server übertragen, selbst innerhalb einer Transaktion, da sich der Transaktionspuffer auf dem Server befindet[3].

Queries werden auf dem Server ausgeführt, wenn sie sich auf persistente Objekte beziehen. Die Ergebnismenge einer Suchanfrage wird auf dem Server erzeugt und als Container von Ondemand–Verweisen übergeben [PProg 30, S.189]. *Filter* geben das Ergebnis Objekt für Objekt zurück. Der Vorteil an Filtern ist, daß der Server bereits die ersten Ergebnis–Objekte liefert, bevor die Ergebnismenge komplett ist. Auf diese Weise kann die Anwendung mit der Verarbeitung des Ergebnisses beginnen, während der Server noch mit der Ausführung der Query beschäftigt ist [PProg 30, S.228].

Verfügbarkeit und Sicherheit

Es gibt administrative Operationen, die nur offline ausgeführt werden können, wie die Migration aller Objekte auf die neueste Version ihrer Klassen. Backups können dagegen online durchgeführt werden [PAdm 30, S.30].

[3] Ab Poet 4.0 ist es möglich, die Objekte bis zur Ausführung oder Abbruch einer Transaktion auf Seite des Clients zu puffern.

5.3.6 Administration

Die Administration erfolgt interaktiv über ein grafisches Interface oder über eine Programmierschnittstelle (API). Es ist problemlos möglich, das Administrationsprogramm auf einem anderen Rechner zu starten und das Datenbanksystem von einem Client aus zu verwalten. Für den Großteil der Verwaltungsaufgaben wird noch nicht einmal ein Account auf dem Server benötigt.

Benutzerverwaltung: Die Benutzerverwaltung ist unabhängig vom Betriebssystem. Nach Einschalten der Autorisierung kann der Datenbankadministrator (über Paßwort in der Datenbank selbst festgelegt) die Zugriffsrechte für Klassen und Mengen von Objekten setzen. Benutzer werden in der Datenbank verwaltet. Zugriffsrechte können einzelnen Benutzern oder Gruppen gewährt werden.

5.3.7 Entwicklung mit Poet

Integration in Entwicklungsumgebungen:

Poet bietet neben Kommandozeilenwerkzeugen eine Entwicklungsumgebung für Poet–Projekte an, die *Poet Developer's Workbench*. Die Parameter von Poet–Projekten werden in einer Projektdatenbank gespeichert, womit die Entwicklung in Arbeitsgruppen unterstützt werden soll. Der Präprozessor wird innerhalb der Poet–Entwicklungsumgebung gestartet. Die Schema–Information wird nach erfolgreichem Durchlauf auf dem Datenbankserver gespeichert. Die erzeugten Quelltextdateien können durch die Projektverwaltung der C++-Entwicklungsumgebung eingebunden und ohne Probleme übersetzt werden. Die Schema–Compilierung ist nach jeder Änderung der Struktur einer persistenten Klasse nötig.

Versionen von Klassen, Objektmigration

Bei Weiterentwicklung einer Anwendung ist die Wahrscheinlichkeit hoch, daß sich Klassen ändern. Es kommt also vor, daß ein Objekt als Exemplar einer Klasse abgespeichert wurde und als Exemplar einer neuen Version der Klasse angefordert wird. Die Konvertierung durch das Datenbanksystem, die Objektmigration, wird bei Poet *Versionierung* genannt [PRef 30, S.29].

In der Schema–Datenbank *Class Dictionary* werden die Informationen über die Klassen der Objektdatenbank abgelegt. Wird eine neue Version einer Klasse angelegt, wird die alte nicht überschrieben, da sie für die Zugriffe älterer Versionen noch benötigt wird. Um einer Anwendung die richtige Version zuordnen zu können, werden Zeitstempel im *Class Dictionary* der Objektdatenbank und in der *Class Factory* der Anwendung benutzt. Die Objektmigration kann im laufenden Betrieb erfolgen (On–the–fly Versioning). Alternativ kann mit dem interaktiven *ClassExplorer* in der Entwicklungsoberfläche oder über API–Aufrufe die Migration der Objekte zur neuesten Version erfolgen. Die Objektmigration erfolgt nach festen Regeln, kann aber durch den Benutzer erweitert werden, z.B. um die Initialisierung neuer hinzugekommener Attribute. Die automatische Objektmigration hat eine Reihe von Einschränkungen [PRef 30, S.34].

Synchronisation der Klassenversionen auf verschiedenen Datenbanken: Da die Klasseninformationen getrennt von den Objektdatenbank im *Class Dictionary* aufbewahrt werden, können durch Übertragen desselben andere Datenbanken auf den neuesten Stand gebracht werden.

5.3.8 Einschätzung

Die Stärken von Poet liegen im Desktop–Bereich und im Bereich heterogener Systeme. Ein Poet–System ist überschaubar, einfach zu verwalten und daher ein geeigneter Einstieg in Objektdatenbanken. Poet hat die geringsten Anforderungen an Hardware und Betriebssystem und ist daher für den Desktop–Bereich prädestiniert, wovon auch die Schnittstellen für OLE und Visual Basic zeugen.

Für den Einsatz im Client–Server–Bereich hat Poet noch Schwachpunkte. Hier sind das Transaktionskonzept und die Sprachschnittstelle mit der expliziten Speicherung zu nennen. In anderen Bereichen bietet Poet bemerkenswerte Fähigkeiten, z.B. bei den Zugriffsrechten oder Versionen von Klassen.

5.4 GemStone

GemStone kam bereits 1987 auf den Markt und war damit das erste kommerzielle Objektdatenbanksystem. GemStone tritt vor allem im Zusammenhang mit Smalltalk auf.

5.4.1 Einordnung

GemStone ist eine aktives objektorientiertes Datenbanksystem und kann auch als persistenter Objektspeicher benutzt werden (siehe Abschnitte 3.3.1 und 3.3.2). Zu speichernde Objekte werden als geändert markiert (siehe Abschnitt 3.2.3).

Als aktive Datenbank verwendet GemStone Smalltalk als Programmiersprache auf dem Server. Ein GemStone–Server bietet damit ein Multi–User–Smalltalk mit persistenten Objekten an.

Während beim persistenten Objektspeicher (siehe Abschnitt 3.3.2) die Vorstellung des Shared Memory Bereichs mit persistenten Objekten herrscht, ist es bei GemStone angebracht, von einem zweiten Smalltalk–System zu sprechen, welches zusätzlich zu den Fähigkeiten des Client–Smalltalk–Systems folgende Eigenschaften bietet:

- Unbeschränkter Speicher für große Objekte und Container
- Mehrbenutzer–Betrieb, öffentliche Objektbereiche
- Transaktionsverwaltung, Sperren und Zugriffsbeschränkung

Erst das GemStone Smalltalk Interface (GSI) ermöglicht die nahtlose Integration in das Smalltalk–System des Clients und die Funktionalität als persistenter Objektspeicher.

5.4.2 Technik

Architektur

Abbildung 5.4 zeigt den Aufbau eines GemStone–Systems. Die Client–Anwendungen sind im oberen Teil dargestellt, das GemStone Objektdatenbanksystem im unteren.

Gem Server: Für jede Sitzung wird ein Gem Server-Prozeß gestartet. Dieser kann auf die Objektdatenbank, das *Object Repository*, zugreifen und ist in der Lage, Smalltalk–Code auszuführen. Objekte werden zur Bearbeitung in einen eigenen Bereich geladen, den *Workspace*. Im *Status* wird vermerkt, welche Objekte gelesen oder geändert wurden.

Stone Manager: Der Stone Manager ist der Transaktionsverwalter für das Object Repository, er verwaltet Ressourcen wie Objektidentifikatoren, Sperren und Puffer. Seine Aufgaben sind die Verwaltung konkurrierender Zugriffe und Sicherung der Konsistenz durch Verarbeitung der

Transaktionsaufforderungen und das Schreiben der Logs. Pro Object Repository existiert ein Stone Manager, welcher beliebig viele Gems bedient.

Beantragt die Anwendung durch „Commit“ die Ausführung einer Transaktion, bittet der zuständige Gem Server den Transaktionsverwalter um Erlaubnis. Geht aus dessen Transaktionsverwaltungsdaten hervor, daß die Voraussetzungen für die Transaktion noch gelten (bei optimistischer Strategie), wird die Erlaubnis erteilt und der Gem Server schreibt die Daten auf die Objektdatenbank.

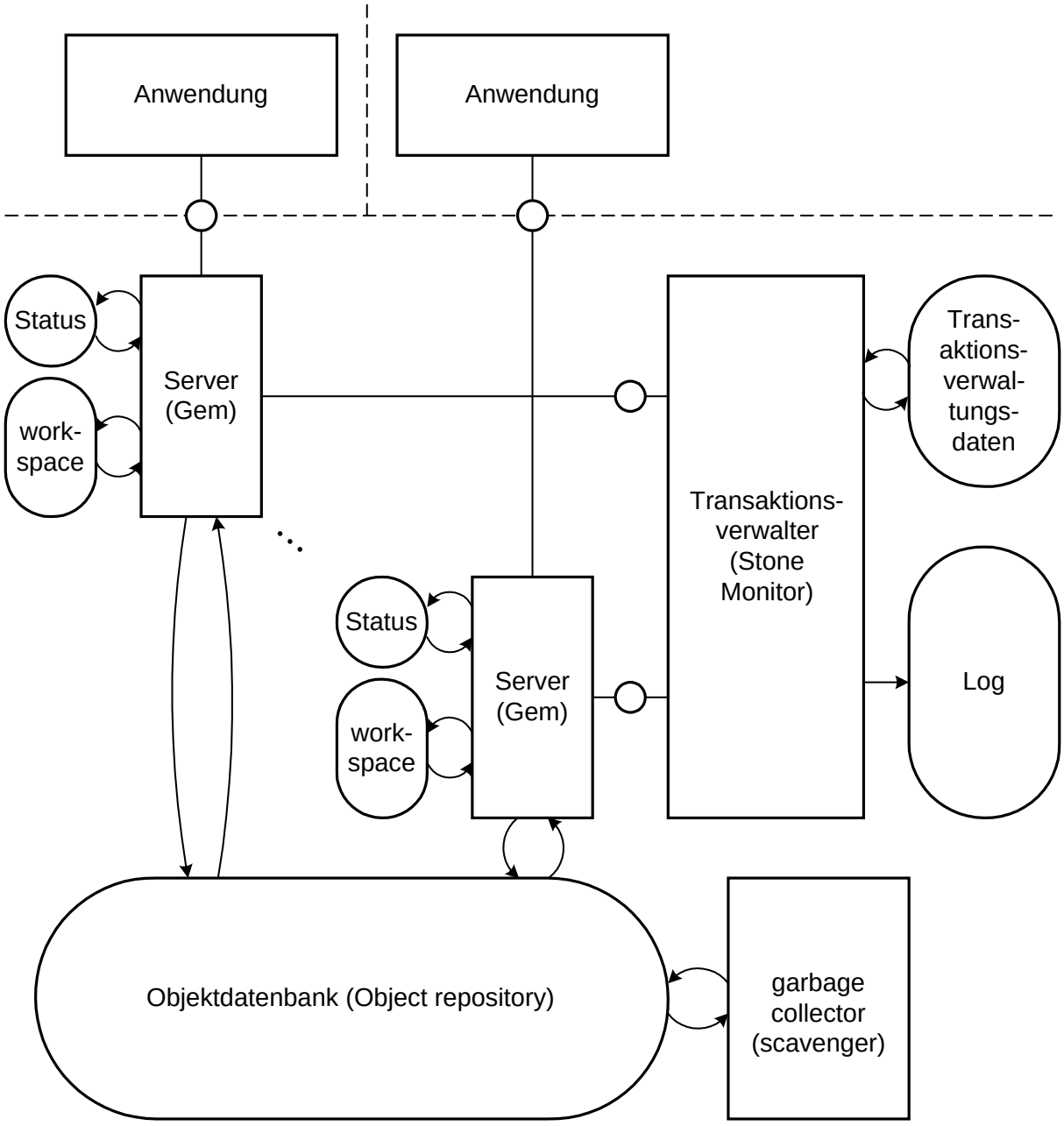

Abbildung 5.4: GemStone Architektur

Scavenger: Der Garbage Collector *Scavenger* säubert das Object Repository von Objekten, auf die kein Verweis mehr zeigt. Neben dem automatischen Betrieb im Hintergrund gibt es auch das gründliche Durchforsten des Repository, welches explizit angestoßen werden muß.

Object Repository: Das Object Repository stellt die eigentliche Objektdatenbank dar. Alle Zugriffe der Gem Server müssen vom Transaktionsverwalter Stone abgesegnet werden. Die Datenbank kann auf verschiedene Rechner verteilt werden. Intern ist das Object Repository als Seitenserver realisiert (siehe Abschnitt 5.1.2).

Abbildung 5.5 zeigt, was sich hinter dem Object Repository verbirgt: Das Object Repository besteht aus Dateien und Festplatten–Partitionen, die als *Extents* bezeichnet werden. Der Zugriff auf die Daten erfolgt seitenweise durch Gem Server. Extents können auf verschiedene Rechner verteilt werden (rechte Seite). Ist das der Fall, wird auf dem fremden Rechner ein *Page Server* Prozeß gestartet, über den der *Cache Manager* auf den Extent zugreifen kann. Der *Cache Manager* ist für die Beschaffung der Seiten und die Verwaltung des Seitenpuffers, des *Shared Object Cache*, zuständig. Er ist dem Gem Server transparent, der Gem Server sieht also nur

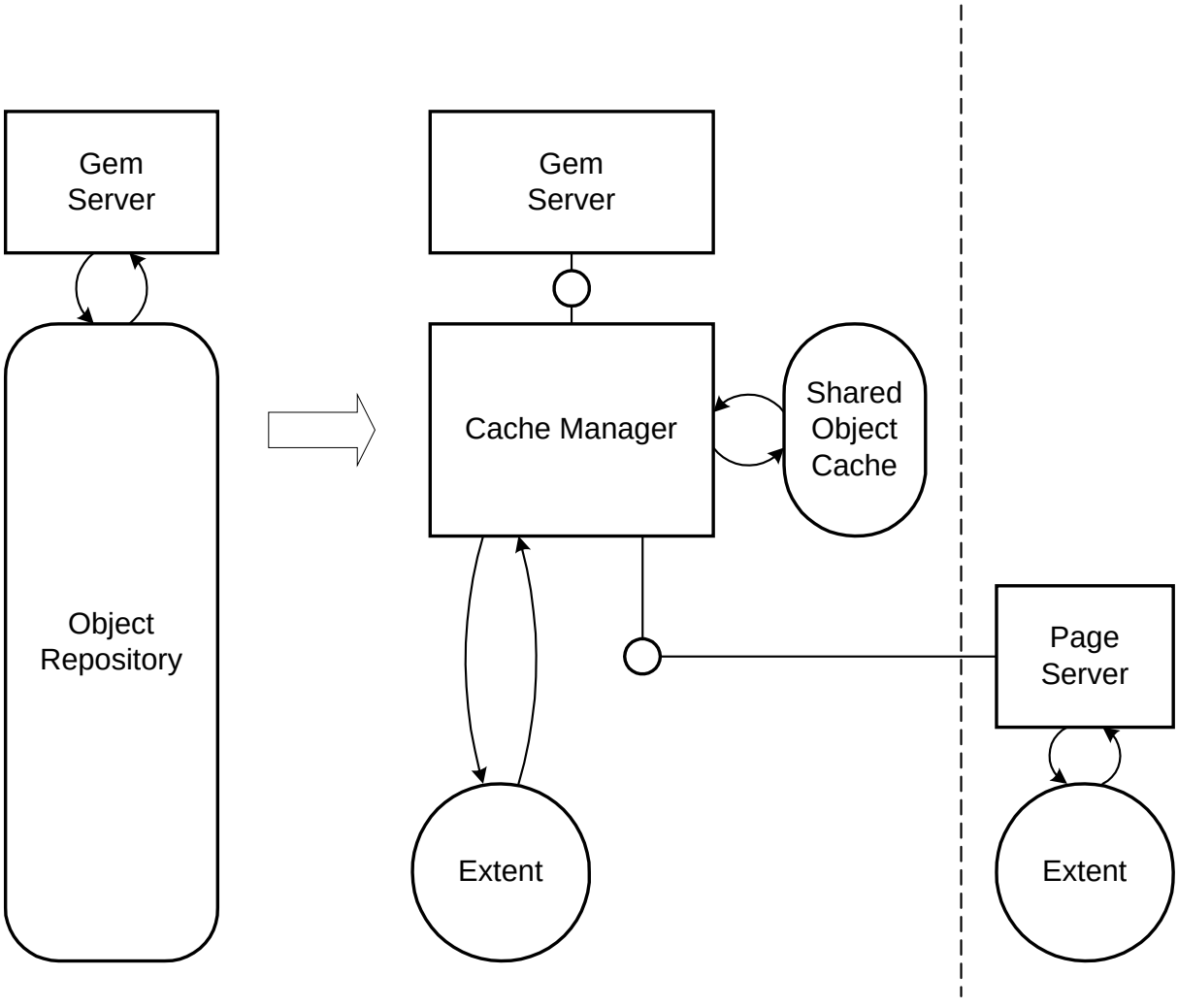

Abbildung 5.5: GemStone Object Repository

eine seitenorientierte Objektdatenbank.

Das Object Repository „gehört“ dem Transaktionsverwalter Stone. Aus diesem Grund gibt es im Object Repository keine Probleme mit konkurrierendem Zugriff und Sperren, da diese vom Stone Manager verwaltet werden. Die Entscheidung für einen seitenbasierten Server für die interne Objektdatenbank läßt sich mit dem hohen Durchsatz, die mit dieser Technik erreicht werden kann, begründen.

Persistente Objekte

Es gibt unter GemStone zwei Arten von persistenten Objekten:

1. Objekte, die auf Anwendungsseite erzeugt und deren Datenteil durch GemStone gespeichert werden (GemStone als persistenter Objektspeicher)

2. Objekte, die auf GemStone–Seite erzeugt und deren Methoden dort ausgeführt werden (GemStone als aktive Datenbank)

Die erste Art ist die, die man von den C++-Objektdatenbanken kennt. Da Smalltalk eine interpretierte Sprache ist und die Klassenbeschreibungen als Objekte verfügbar sind, kann der Datenteil jedes Smalltalk–Objekts persistent gemacht werden. Ist die Klasse auf GemStone–Seite nicht bekannt, wird ein passendes Klassenobjekt ohne Methodenteil erzeugt.

Bei der zweiten Art entspricht GemStone einem separaten Objektraum mit einer eigenen Smalltalk–Maschine. Objekte werden in diesem Raum erzeugt, Methoden können dort ausgeführt werden.

Es ist möglich, zwischen beiden Arten zu wechseln. Ein Objekt auf Seite des Clients kann auf GemStone–Seite mitsamt seinen Methoden repliziert werden und umgekehrt. Voraussetzung dafür ist, daß alle Methoden von GemStone interpretiert werden können — das ist nicht der Fall bei Ein/Ausgabe- und Bildschirmroutinen. Ein Objekt aus der GemStone–Welt kann auf Seite der Anwendung repliziert und bearbeitet werden. Alternativ ist es möglich, eine Hülse, einen sogenannten *Forwarder*, auf Seite des Clients aufzubauen, welche alle Methodenaufrufe an das Objekt auf GemStone–Seite durchreicht.

Bei Benutzung von GemStone sind alle Objekte auf Client–Seite potentiell persistent, da die Basisklasse *Object*, von der alle Klassen erben, um diese Eigenschaft erweitert wird. Für jede Klasse kann ein sogenannter Transparenzmodus eingestellt werden, bei dem *jeder* modifizierende Attributzugriff eine Geändert–Markierung (Mark Dirty) auslöst [GSI 41, 3-29 und 9-2]. Ein Objekt auf Client–Seite wird persistent, wenn es zur transitiven Hülle eines persistenten Objekts gehört, wenn also das persistente Objekt direkt oder indirekt darauf verweist.

Verweise: Smalltalk–Verweise gelten nur innerhalb eines Objektraums, auf dem Client innerhalb des sogenannten *Image*, auf GemStone–Seite im gesamten Object Repository. Ein *Connector* stellt die Verbindung zwischen beiden Objekträumen her, wobei er die Objekt–IDs auf beiden Seiten kennen muß. Um ohne Objekt–ID zu einem Objekt zu kommen, gibt es benannte

Einstiegspunkte, die sogenannten *Symbols*, von denen ein Verweispfad zum gewünschten Objekt führt (siehe auch Abschnitt 3.2.1). Benutzerbezogene Symbollisten definieren Namensräume und legen damit die Sichtbarkeit von Objekten fest.

Connector: Ein *Connector* stellt die Beziehung zwischen einem Objekt auf Client-Seite und einem Objekt auf GemStone-Seite her [GSI 41, 3-4]. Ein auf einer Seite existierendes Objekt kann auf der anderen repliziert werden (*Replicate*) oder auf Client–Seite kann ein *Forwarder*–Objekt eines GemStone–Objekts erzeugt werden. Sollte auf einer Seite die Klasse des Objekts noch nicht bekannt sein, wird das Klassenobjekt dort automatisch erzeugt [GSI 41, 3-16]. Durch den Einsatz von Connectoren kann die Ausführung einer Anwendung zwischen Client und Gem Server aufgeteilt (partitioniert) werden, ohne den Quelltext ändern zu müssen.

Stub: Die Replikation eines Objekts von GemStone- auf Client–Seite zieht normalerweise die Replikation sämtlicher Objekte nach sich, auf die das Objekt verweist; da für diese Objekte wiederum das gleiche gilt, kann die Replikation eines Objekts unter Umständen die Übertragung einer großen Datenmenge auslösen. In GemStone kann bei Replikation eines Objekts gewählt werden, bis zu welcher Verweistiefe Objekte repliziert werden. Alle weiteren Verweise zeigen auf spezielle Lade–Objekte, die *Stubs*. Folgt Smalltalk einem Verweis auf ein Stub, veranlaßt dieser die Replikation des Objekts, für das er zuständig ist [GSI 41, 3-21]. Ein Stub ist damit vergleichbar mit den Ondemands von Poet — siehe Abschnitt 5.3.2.

Queries

Anfragen werden in GemStone Smalltalk formuliert, welches auf dem Server ausgeführt wird. Dadurch sind komplexe Operationen möglich. Möglichkeiten des Ergebniszugriffs:

- Forwarder: Die Ergebnismenge verbleibt auf dem Server, auf Seite des Clients wird nur ein Forwarder auf das Ergebnis erzeugt.
- Stream: Die Ergebnismenge verbleibt auf dem Server, auf Seite des Clients wird ein Stream–Objekt erzeugt, mit dessen Hilfe der Inhalt der Ergebnismenge Objekt um Objekt angefordert werden kann.
- Replikation: Die Ergebnismenge wird durch Replikation des Ergebniscontainers als ganzes oder teilweise auf den Client gebracht. Die teilweise Replikation benutzt Stubs.

5.4.3 Mehrbenutzerbetrieb

Sperren

Es gibt in GemStone drei Sorten von Sperren [GProg 41, 12-17]:

Read Locks: Es wird garantiert, daß das Objekt bis zur Ausführung der Transaktion seinen Wert nicht ändert. Ein Read Lock verbietet anderen damit das Schreiben und das Setzen von Sperren, die zum Schreiben berechtigen.

Write Locks: Es wird garantiert, daß das Objekt geändert und bei Ausführung der Transaktion geschrieben werden kann. Das Objekt kann nun nicht mehr von anderen gesperrt oder geschrieben werden.

Exclusive Locks: Zusätzlich zum Write Lock betrifft die Sperre auch das Lesen des gesperrten Objekts durch andere. Eine Transaktion schlägt fehl, wenn versucht wird, ein mit einem Exclusive Lock gesperrtes Objekt zu lesen.

Sperren überdauern Transaktionen, können aber bei Beendigung derselben aufgehoben werden. Sperren können auf einzelne Objekte (also auch Klassenobjekte) oder Mengen von Objekten gesetzt werden. Über eine Datenbanksitzung hinaus kann nicht gesperrt werden.

Transaktionen

Die Voreinstellung ist, daß der Beginn einer Sitzung automatisch eine Transaktion startet, die spätestens bei Sitzungsende beendet wird [GSI 41, 4-2]. Transaktionen können explizit begonnen, committed oder abgebrochen werden. GemStone bietet die optimistische und die pessimistische Transaktionsstrategie an [GProg 41, 12-5].

Es ist möglich, auf Objekte außerhalb einer Transaktion zuzugreifen; das Ändern und Anlegen von persistenten Objekten ist dagegen nur innerhalb einer Transaktion möglich.

Zugriffsbeschränkung und Autorisierung

Namensräume bestimmen die Sichtbarkeit von Objekten. Namensräume sind Zuordnungslisten von Namen (*Symbolen*) zu Verweisen auf Objekte. Alle Smalltalk–Objekte sind über den System–Namensraum erreichbar — falls nicht, werden sie Opfer der Garbage Collection. In einer Mehr–Benutzer–Smalltalk–Umgebung wird jedem Benutzer ein eigener Namensraum zur Verfügung gestellt. Damit ist ein Objekt, obwohl es im gemeinsamen Objektraum existiert, nur für den zugänglich, für den ausgehend von den Symbolen im Namensraum ein Pfad dorthin existiert. Neben privaten Namensräumen gibt es auch solche, die für Gruppen oder alle Benutzer offen sind. Wer einen Verweis auf ein Objekt besitzt, kann es verändern, egal aus welchem Namensraum es stammt.

Unabhängig vom Namensraumkonzept kann der Zugriff auf Objekte beschränkt werden. Durch Zuordnung der zu schützenden Objekte zu Autorisationsobjekten, den sogenannten *Segments*, ist eine Zugriffsbeschränkung bis zur Objektebene möglich [GProg 41, 11-5].

Die Autorisierung erfolgt beim Server bei Sitzungsanmeldung. Neben der Autorisierung beim Datenbankverwalter ist je nach Einrichtung des Servers noch eine Autorisierung beim Betriebssystem des Server–Rechners notwendig [GAdm 41, 3-9].

lange Transaktionen, Check–In/Out, Objekt–Versionen

GemStone unterstützt keine langen Transaktionen. Der Check–In/Out–Mechanismus kann allerdings mit wenig Aufwand nachgebildet werden.

Ereignismeldungen

Watch/Notify: Ereignisse bezüglich einzelnen Objekten und Mengen von Objekten können von den Gem Servern erkannt, behandelt und an die Anwendung weitergereicht werden. Die zu beobachtenden Objekte werden in einen speziellen Container, *Notify Set*, eingetragen [GProg 41, 12-34].

Signals: Anwendungen können sich gegenseitig Ereignismeldungen, die sogenannten *Signals*, schicken, die von den Gem Servern vermittelt werden [GProg 41, 12-40].

5.4.4 Heterogene Systeme

Sprachschnittstellen

Objektzugriff: Smalltalk–Umgebungen verschiedener Hersteller
C++

Datenzugriff: C
Hierbei ist zu betonen, daß auch über die C–Schnittstelle Smalltalk–Methoden auf dem Server ausgeführt und Objekte (und Klassenobjekte) erzeugt werden können. Ergebnisse können nur als Daten, nicht als Objekte übertragen werden. Die C–Schnittstelle war die erste verfügbare Sprachschnittstelle für GemStone.

Unterschiedliche Betriebssysteme oder Plattformen

Als interpretierte Sprache ist Smalltalk von Natur aus portabel, weswegen der Server mit verschiedenartigen Clients zurechtkommt. Bei Zugriff aus C++ oder C gibt es keine Probleme, da die Objekte bzw. Daten auf Seite des Clients in das jeweilige Format gebracht werden. Die Netzwerkschicht wird gekapselt, heterogene Netzwerke sind damit kein Problem.

5.4.5 Skalierbarkeit

Ein System, in dem GemStone benutzt wird, kann in drei Ebenen unterteilt werden:

- Die obere Ebene bildet die Smalltalk–Anwendung, die GemStone benutzt und für die Präsentation der Daten und Ein/Ausgabe zuständig ist.

- Die mittlere Ebene besteht aus Gem Servern, die Methoden auf persistente Objekte ausführen können und damit zu komplexen Operationen auf großen Datenmengen fähig sind. Gem Server Prozesse können auf mehrere Rechner verteilt werden.

- Die untere Ebene besteht aus dem Object Repository, der eigentlichen Objektdatenbank, welche ebenfalls auf mehrere Rechner verteilt werden kann. Weiterhin ist vorgesehen, die Kapazität durch den Anschluß relationaler Datenbanken an ein GemStone System zu erhöhen.

Verweise zwischen Objekten verschiedener Datenbanken, Multi-Server-Betrieb

Ein Object Repository kann auf mehrere verschiedenartige Rechner verteilt werden, wobei Extents zur Sicherheit gespiegelt werden können. Dadurch sind große Datenbanken möglich. Gem Server können auf verschiedenen Rechnern laufen, lediglich der Transaktionsverwalter Stone darf nur einmal pro Object Repository vorhanden sein. Eine Anwendung kann auf mehrere Repositories zugreifen [GSI 41, 2-3].

Verweise zwischen Objekten verschiedener Object Repositories sind nicht möglich.

Netzwerkbelastung, Server-Queries

Die Netzwerkbelastung wird bei geschickter Verteilung der Anwendungslogik auf Client und Server stark verringert. Methoden können dort ausgeführt werden, wo auch die Daten abgelegt sind. Für hohen Durchsatz zwischen Object Repository und Gem Servern sorgen Puffer.

Verfügbarkeit und Sicherheit

Wichtige administrative Operationen wie Schema–Evolution und Backup sind im laufenden Betrieb möglich. GemStone kommt mit großen Datenmengen (Terabyte–Bereich) zurecht und unterstützt die redundante Speicherung (*Replication Support*, [GProg 41, 13-2] und [GAdm 41, 1-9]) zur höheren Verfügbarkeit.

5.4.6 Administration

Die Administration erfolgt teilweise über die Smalltalk–Entwicklungsoberfläche auf dem Client (Benutzerverwaltung, Setzen der Zugriffsrechte auf Objekte, Verwaltung von Namensräumen), teilweise über ein Kommandozeilenprogramm auf dem Server (*topaz*).

Auf dem Server–Rechner benötigt der GemStone–Administrator keine besonderen Rechte.

5.4.7 Entwicklung mit GemStone

Integration in Entwicklungsumgebungen:

Das GemStone Smalltalk Interface bietet eine nahezu nahtlose Integration in die Smalltalk–Entwicklungsumgebung des Clients. Der Entwickler findet in seinem Smalltalk–System Entwicklungswerkzeuge für GemStone Smalltalk vor und kann innerhalb seines eigenen und der öffentlichen Namensbereiche mit Browsern den Objektbestand anzeigen und manipulieren.

Versionen von Klassen:

Klassenbeschreibungen werden in Smalltalk in eigenen Klassenobjekten abgelegt. Wird eine Klasse modifiziert, bleibt die alte Version erhalten. Die Versionen einer Klasse werden in einer Liste abgelegt. Jedes Objekt kennt sein Klassenobjekt, so daß sichergestellt ist, daß auch nur passende Methoden aufgerufen werden. Eine Anwendung kann auf alle Versionen einer Klasse zugreifen [GProg 41, 5-2].

Objektmigration:

Die Objektmigration ist automatisch möglich durch Senden einer Nachricht an das Klassenobjekt oder an die zu migrierenden Objekte. Jeder Benutzer, der Schreibrechte für die zu migrierenden Objekte besitzt, kann eine Migration durchführen. Die Migration kann auf jede Version der Klasse erfolgen, nicht nur auf die neueste [GProg 41, 5-7].

5.4.8 Einschätzung

GemStone ist direkt auf Smalltalk zugeschnitten und für diese Sprache die erste Wahl. Die Möglichkeit, Teile der Anwendung auf dem Server auszuführen, hebt GemStone von anderen Produkten ab. Es ist auch das einzige System, bei dem auf Wunsch auch die Methoden zusammen mit dem Objekt abgespeichert werden können. Durch Einsatz der interpretierten Programmiersprache Smalltalk erreicht GemStone eine akzeptable Plattformunabhängigkeit. Das Konzept der aktiven Datenbank mit Objektanschluß bietet die größte Vielfalt an Einsatzmöglichkeiten.

Kapitel 6

Vergleich der Produkte

Es gibt kein Idealprodukt, das für alle Anwendungen gleich gut geeignet wäre. Bei der Auswahl sind stets die Anforderungen zu untersuchen, die für das spezielle Einsatzgebiet gelten. Die vier untersuchten Produkte haben ausnahmslos Stärken und Schwächen. Für jedes gibt es besonders geeignete Einsatzgebiete.

In Kapitel 4 wurden Anforderungen vorgestellt, die sich beim Einsatz in Client–Server–Systemen ergeben. Unter Berücksichtigung dieser Anforderungen erfolgte die Vorstellung der Produkte in Kapitel 5. Zu jedem Produkt wurde eine Einschätzung abgegeben.

Dieses Kapitel faßt die Ergebnisse des vorigen zusammen und bietet einen Überblick über die Produkte.

6.1 Erfüllung der Anforderungen

Tabelle 6.1gibt einen groben Überblick darüber, wie die Produkte die Anforderungen erfüllen:

+	Anforderung erfüllt
o	Anforderung läßt sich nur teilweise oder umständlich erfüllen
-	Anforderung läßt sich nicht oder nur unbefriedigend erfüllen

Erläuterung der Einträge:

Mit **transparenter Objektzugriff** ist vor allem die Sprachschnittstelle gemeint. ObjectStore bietet einen nahezu transparenten Objektzugriff, für den dank der impliziten Speicherung kaum Änderungen am Quelltext nötig sind. Bei Versant ist das Erben von einer Versant–Klasse und die Erweiterung der Zugriffsmethoden innerhalb der Klassen nötig. Poet verlangt neben der Vererbung von einer Poet–Klasse die explizite Speicherung, wodurch größere Änderungen im Quelltext nötig werden. GemStone ermöglicht auf Wunsch den transparenten Zugriff auf beliebige Objekte.

Granularität der Sperren: Versant, Poet und GemStone bieten die Möglichkeit, auf Objektebene zu sperren. ObjectStore kann nur auf Seitenebene sperren, bietet aber immerhin eine Deadlock Detection an.

Zugriffsbeschränkungen bezüglich Benutzer: ObjectStore kann den Zugriff nur auf ganze Datenbankdateien oder, bei Verwendung des eigenen Dateisystems (raw fs), auf Segmente durch die Mechanismen des unterliegenden Dateisystems beschränken. Bei Versant ist eine Beschränkung nur auf ganze Datenbankdateien möglich, wobei eine feinere Abstufung der Zugriffsmöglichkeiten (Rollen) unabhängig vom Dateisystem möglich ist. Poet kann Exemplare von Klassen im Zugriff beschränken, wobei die Benutzerverwaltung und Autorisation komplett unabhängig vom Betriebssystem sind. GemStone kann den Zugriff für einzelne Objekte beschränken.

Lange Transaktionen werden nur von GemStone nicht unterstützt. Bei Poet gibt es keine Versionen von Objekten und ein Check-In ist nur für alle Objekte eines Workspace zusammen möglich.

Ereignismeldungen sind bei ObjectStore nur zwischen Clients möglich, bei den anderen Systemen kann der Server Clients beim Eintreten von Ereignissen bezüglich Objekten benachrichtigen.

Heterogene Systeme: Bedingt durch seine Technik kann ObjectStore Anforderungen heterogener Systeme nicht oder nur unter Aufwand erfüllen. Poet und GemStone benutzen ein plattformunabhängiges Objektmodell für C++ bzw. Smalltalk, Versant ein generisches Objektmodell.

Skalierbarkeit und Verfügbarkeit: ObjectStore und Poet Datenbanken müssen zur Migration von Objekten offline betrieben werden. Die beste Skalierbarkeit bietet Versant, auch GemStone hat durch die Verteilung der Anwendungslogik und der Extents des Object Repositories einen guten Stand.

Server Queries bezeichnen die Eigenschaft des Datenbanksystems, Suchanfragen auf dem Datenbank–Server durchzuführen, um dadurch den Transport großer Datenmengen zum Client zu verhindern. Der ObjectStore Server kennt keine Objekte und bietet daher diese Möglichkeit nicht an. Unter Versant können Anwendungen Anfragen an den Server schicken, der sie

	ObjectStore	Versant	Poet	GemStone
Transparenter Objektzugriff	+	o	-	+
Granularität der Sperren	o	+	+	+
Zugriffsbeschränkungen	-	-	+	+
Lange Transaktionen	+	+	o	-
Ereignismeldungen	o	+	+	+
Heterogene Systeme	-	+	+	+
Skalierbarkeit und Verfügbarkeit	o	+	o	+
Server Queries	-	+	o	+
Geringe Netzwerkbelastung	o	+	o	+
Administrations- & Entwicklungswerkzeuge	o	o	+	+

Tabelle 6.1: Vergleich der vier Objektdatenbanken

ausführt. Die Anfragen können Container von Objekten oder über Pfadausdrücke spezifizierte Objekte betreffen. Unter Poet werden Anfragen an Container gestellt, welche sie an den Server durchreichen, der sie vor Ort ausführt. GemStone ist eine aktive Datenbank und kann komplexe Operationen auf persistenten Objekten ausführen.

Eine **geringe Netzwerkbelastung** ist für hohen Durchsatz zwingend nötig. Bei ObjectStore kann eine Query trotz optimierter Suchstrategien zur Übertragung eines Großteils der Elemente eines Containers führen. Bei Poet führt jeder `Store`–Aufruf zur Übertragung des Objekts, unabhängig von Transaktionen.

Gute **Administrations- und Entwicklungswerkzeuge** vereinfachen den Umgang mit der Objektdatenbank, vor allem, wenn sie von einem Client–Rechner aus funktionieren. Hier bieten GemStone und Poet die beste Einbindung.

6.2 Eignung der Produkte für Client–Server–Umgebungen

ObjectStore ist für Client–Server–Systeme, in denen Heterogenität eine große Rolle spielt, wenig geeignet. Der Grund dafür liegt in der Technik, Objekte durch Speicherung ihres Abbilds im Hauptspeicher persistent zu machen (siehe 5.1.2). Die Probleme kommen beim Einsatz verschiedener Plattformen und verschiedener Programmiersprachen. Da der Server keine Operationen auf Objekten, wie etwa das Durchsuchen von Containern, durchführen kann, ist die Gefahr einer hohen Netzwerkbelastung groß.

Versant zielt mit seinen Konzepten auf den Client–Server–Markt. Durch das generische Objektmodell wird Heterogenität in jeder Beziehung unterstützt, die eindeutige Objekt–ID *loid* erlaubt die Verteilung der Daten. Versant ist keine aktive Datenbank, der Server kann jedoch Queries ausführen.

Poet zeigt seine Stärken im Desktop–Bereich. Heterogenität ist für Poet unproblematisch; große Datenmengen, auf verschiedene Server verteilt, werden durch den expliziten Zugriff zum Problem hinsichtlich Netzwerkbelastung und Durchsatz.

GemStone spielt als aktive Objektdatenbank eine besondere Rolle. Die Möglichkeit, Applikationen auf Client und GemStone zu verteilen und GemStone als persistenten Objektspeicher verwenden zu können, bietet ein breites Einsatzspektrum.

Kapitel 7

Zusammenfassung und Ausblick

Für die Entwicklung von Objektdatenbanken ist noch kein Ende abzusehen. Im Gegensatz zu relationalen Datenbanken gibt es erhebliche Unterschiede in Technik und Sprachschnittstelle. Für letztere gibt es seit kurzer Zeit einen Standard [ODMG 93], nach dem sich die Hersteller im Laufe der Zeit richten werden. Es ist momentan nicht abzusehen, daß Objektdatenbanken relationale Datenbanken verdrängen werden. Durch den Einsatz objektorientierter Technologien gibt es aber immer häufiger Anwendungen, die mit einer Objektdatenbank besser bedient sind. Die relationale Datenbanktechnologie ist über 20 Jahre alt und konnte über viele Produktversionen hinweg ausgefeilt werden. Die ersten Objektdatenbanken kamen dagegen vor 9 Jahren auf den Markt und werden erst allmählich in größerem Maßstab eingesetzt.

Die Untersuchung von Objektdatenbanken ist ein umfangreiches Unterfangen. Der Großteil der im Rahmen dieser Arbeit präsentierten Ergebnisse stammt aus den Handbüchern der einzelnen Produkte, die im Literaturverzeichnis aufgeführt sind, sowie aus Schulungen und Fragen an die Support–Mitarbeiter der Hersteller. Anwendungen, die die Objektdatenbanken benutzen, sind in der Zeit nicht entstanden, nur Code–Fragmente, die die Sprachschnittstelle und die Containerkonzepte demonstrieren. Aus diesem Grund konnten auch keine Messungen und Tests bezüglich Durchsatz (Performance) und großen Datenmengen gemacht werden.

Das Software–Technologie–Labor bietet die Möglichkeit, eine Evaluation von Objektdatenbanken im Zusammenhang mit dem Bau eines Prototyps durchzuführen. Die Betrachtungen der vorigen Kapitel können als Grundlage für die Entscheidung für ein bestimmtes Produkt dienen. Gerade Objektdatenbanken bieten noch einen großen Raum für Erweiterungen und Verbesserungen. Neue Produktversionen bieten zum Teil erhebliche Neuerungen, weswegen es sich lohnt, die Entwicklung weiter zu verfolgen.

Literaturverzeichnis

[GIntro 41] An Introduction to GemStone Version 4.1
Oktober 1995, GemStone Systems, Inc.

[GProg 41] GemStone Version 4.1 Programming Guide
Juli 1995, GemStone Systems, Inc.

[GSI 41] GemStone Version 4.1 Smalltalk Interface for Visualworks 2.0
Juli 1995, GemStone Systems, Inc.

[GAdm 41] GemStone Version 4.1 for Unix System Administrator's Guide
Juli 1995, GemStone Systems, Inc.

[GSchul] GemStone Schulung: Introduction to the Use of GemStone
1995, Servio Corp.

[Hahn 95] Wolfgang Hahn et al.: Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken
1995, Informatik Spektrum 18: 143-151; Springer-Verlag

[Lausen 96] Georg Lausen; Gottfried Vossen: Objekt-orientierte Datenbanken: Modelle und Sprachen
1996, Oldenbourg-Verlag, München/Wien

[Manola 94] Frank Manola: An Evaluation of Object-Oriented DBMS Developments
1994, GTE Laboratories Incorporated, Waltham
http:// info.gte.com/ftp/doc/tech-reports.html, MANO94a

[Meyer 90] Bertrand Meyer: Objektorientierte Softwareentwicklung
1990, Carl Hanser Verlag München, Prentice-Hall International Inc., London

[ODMG 93] R.G.G. Cattel (Editor): The Object Database Standard: ODMG-93 Release 1.2
1996, Morgan Kaufmann Publishers, Inc.

[OSUser 4] ObjectStore C++ API User Guide Release 4
Juni 1995, Object Design Inc., Burlington

[OSRef 4] ObjectStore C++ API Reference Release 4
Juni 1995, Object Design Inc., Burlington

[OSBuild 4] ObjectStore C++ Building Applications Release 4
Juni 1995, Object Design Inc., Burlington

[OSManag 4] ObjectStore Management Release 4
Juni 1995, Object Design Inc., Burlington

[OSTech 3] ObjectStore Technical Overview Release 3
März 1994, Object Design Inc., Burlington

[PProg 30] POET 3.0 Programmer's Guide
Januar 1995, POET Software GmbH, Hamburg

[PRef 30] POET 3.0 Reference Guide
Januar 1995, POET Software GmbH, Hamburg

[PAdm 30] POET 3.0 Administrator's Workbench Introduction and Reference Guide
Januar 1995, POET Software GmbH, Hamburg

[PDev 30] POET 3.0 Developer's Workbench Introduction and Reference Guide
Januar 1995, POET Software GmbH, Hamburg

[PTech 30] POET 3.0 Technical Overview
Januar 1995, POET Software GmbH, Hamburg

[Shekar 95] Chandrashekar Ramanathan: Providing Object-Oriented Access to a Relational Database
1995, Mississippi State University, Department of Computer Science
shekar@CS.MsState.edu

[VConcept 40] VERSANT ODBMS Release 4.0: VERSANT Concepts and Usage Manual
Juli 1995, Versant Object Technology

[VSystem 40] VERSANT ODBMS Release 4.0: System Manual
Juli 1995, Versant Object Technology

[VRef 40] VERSANT ODBMS Release 4.0: C++/VERSANT Reference Manual
Juli 1995, Versant Object Technology

[Wendt 91] Siegfried Wendt: Nichtphysikalische Grundlagen der Informationstechnik: Interpretierte Formalismen
1991, 2.Auflage, Springer–Verlag Berlin Heidelberg New York

[Wiegert 95] Oliver Wiegert: Änderbarkeit durch Objektorientierung
1995, Friedr. Vieweg & Sohn Verlagsgesellschaft mbH Braunschweig/Wiesbaden