Konzeption und Implementierung einer erweiterten ORB/DBMS-Schnittstelle (ODA+) in einer CORBA-basierten CSCW-Umgebung

Diplomarbeit

Universität Rostock, Fachbereich Informatik
Lehrstuhl für Datenbank- und Informationssysteme

vorgelegt von: Karsten Wendt
geboren am 29.04.1973 in Perleberg

Betreuer: Prof. Dr. Andreas Heuer
Prof. Dr. Clemens H. Cap
Dr. Holger Meyer
Dipl.-Inf. Guntram Flach
Dipl.-Inf. Uwe v. Lukas

Abgabedatum: 31.5.1998

Zusammenfassung

In allen Bereichen der Wirtschaft, Ausbildung, Forschung und Verwaltung gewinnt der Einsatz von Techniken des computerunterstützten Arbeitens (CSCW) an Bedeutung. Für die Zusammenarbeit von verteilten Teams auf der Basis von Computern muß Rücksicht auf ein heterogenes Umfeld genommen werden. Durch den CORBA-Standard der Object Management Group lassen sich CSCW-Techniken plattform- und programmiersprachenunabhängig umsetzen. Die Wiederverwendung spielt aufgrund der ständigen Zunahme der Komplexität und der gleichzeitigen Verringerung der Entwicklungszeiten eine wesentliche Rolle in CSCW-Anwendungen. Die Integration von Datenbanksystemen in CSCW-Werkzeuge stellt eine geeignete Ergänzung für die Nutzung von Datenbankkonzepten wie konkurrierender Zugriff, Recovery, Anfragen und Persistenz dar.

Eine sehr anspruchsvolle Anforderung in CSCW-Anwendungen ist der konkurrierende Zugriff auf gemeinsame Objekte. Deshalb ist es nicht ausreichend, nur Basis-DBMS-Funktionalität bereitzustellen. Beispielsweise wird durch die Verwendung flacher Transaktionen in den meisten Fällen nur isoliertes Arbeiten unterstützt. Daher werden die Transaktionen und Basismechanismen erweitert, z.B. durch die Versionierung von Objekten, die flexible Sperrenverwaltung auf Objektebene sowie die Unterstützung geschachtelter und langer Transaktionen, so daß auch kooperatives Arbeiten auf Datenbankobjekte möglich ist. Die entwickelten Konzepte werden innerhalb eines Object Database Adapters umgesetzt und anhand eines Anwendungsprototypen validiert.

Abstract

Computer Supported Cooperative Work becomes more and more essential in economy, education, science and administration. For the cooperation of distributed teams heterogeneous environment must be taken into account. The CORBA standard of the Object Management Group is a promising way to provide platform and programming language transparency. Reuse of software components play an important role because of increasing complexity and the decreasing development times. Integration of Database Systems is a good example for reuse and provide database-concepts such as concurrency control, recovery, queries and persistence.

A demanding requirement to CSCW-applications is the cooperation of different users. For this it is not sufficient to provide the basic functionality of Database Management Systems. For instance in the majority of cases, flat transactions will isolate the users. Hence, transactions and basic techniques are extended in various ways such as support of versions and flexible locks of objects and support of nested and long transactions in order to enable cooperative work on database objects. The developed concepts are realized by an Object Database Adapter and are validated in the context of a prototyp.

CR-Klassifikation

C.2.4	Distributed Systems (H.2.4)	verteilte Systeme
	distributed Databases	verteilte Datenbanken
	Distributed Applications	verteilte Anwendungen
H.2.0	Database Management General (E.5)	allgemeine Datenbankverwaltung
H.2.4	Database Management Systems	Datenbank-Management-Systeme
	Transaction processing	Transaktionsverarbeitung
H.2.8	Database Management Systems	Datenbank-Management-Systeme
	Distributed Databases	verteilte Datenbanken
H.4.3	Communications Applications	Kommunikationsanwendungen

Keywords

Computer Supported Cooperative Work, Common Object Request Broker Architecture, Object Oriented Database Management System, Object Database Adapter, Persistence Object Service, Transaction management.

Danksagung

Ich möchte mich bei Prof. Dr. Andreas Heuer und bei Dr. Holger Meyer für ihre nützliche Kritik, hilfreichen Hinweise und Anregungen sowie bei Prof. Dr. Clemens H. Cap für das Gutachten bedanken.

Ferner bin ich Dipl.-Inf. Guntram Flach und Dipl.-Inf. Uwe v. Lukas für die sehr geduldige und motivierende Betreuung, die fachlichen Diskussionen sowie für den Stil der Arbeit dankbar.

Meinen besonderen Dank gilt Dipl.-Ing. Thomas Runge für die zahlreichen Ratschläge und Hilfen, die er mir bei der Problembewältigung in der Realisierungsphase gab.

Inhaltsverzeichnis

1 Einführung **10**

1.1 Motivation . . . 10

1.2 Das DICE-Projekt . . . 11

1.3 Grundlagen und Begriffe . . . 12

1.3.1 Computer Supported Cooperative Work . . . 12

1.3.2 Der CORBA-Standard . . . 13

1.3.3 Transaktionen und Sperren . . . 15

1.3.4 ODA im DICE-Projekt . . . 17

1.4 Das Anwendungsszenario . . . 23

1.5 Anforderungen des ODA+ im Anwendungsszenario . . . 25

1.6 Vorgehensweise . . . 26

2 Theorie zur Synchronisation von Transaktionen **28**

2.1 Klassifikation . . . 29

2.1.1 Optimistische Verfahren . . . 30

2.1.2 Pessimistische Verfahren . . . 31

2.2 Sperrmodelle . . . 31

2.2.1 Die Innen- und Außenwirkung von Sperren . . . 32

2.2.2 Flexibilisierung von Sperren . . . 33

2.2.2.1 Hierarchische Sperren . . . 34

2.2.2.2 Anderweitige Sperrkonzepte . . . 35

2.3 Synchronisation von Transaktionen . . . 36

2.3.1 Basisprotokolle . . . 37

2.3.2 Multiversionsverfahren . . . 38

2.3.3 Geschachtelte Transaktionen . . . 39

2.3.4 Synchronisation in objektorientierten Systemen . . . 43

2.3.5 Workspace-Konzept . . . 46

2.3.6 Synchronisation über Konsistenzbedingungen 48
2.4 Fazit 48

3 Lösungsansatz 50
3.1 Werkzeugbetrachtung Datenbanksysteme 50
3.1.1 Das OODB **O_2** 52
3.1.2 Das OODB Versant 52
3.1.2.1 Transaktionen 54
3.1.2.2 Lange Transaktionen 54
3.1.2.3 Kurze Transaktionen 54
3.1.2.4 Optimistisches Locking 55
3.1.2.5 Geschachtelte Transaktionen 55
3.1.2.6 Sperren in Versant 55
3.1.2.7 Sperrmodellerweiterung 57
3.1.3 Fazit der Datenbankanalyse 57
3.2 Die transparente Umsetzung erweiterter Datenbankkonzepte 57
3.2.1 Die Gesamtarchitektur 58
3.2.2 Die Beschreibung der entwickelten IDL-Spezifikationen 59
3.2.2.1 Das Interface für persistente CORBA-Objekte 60
3.2.2.2 Anfragen und Beziehungen 61
3.2.2.3 Die Fabrikschnittstelle 62
3.2.3 Die allgemeine Transaktionsschnittstelle 64
3.2.3.1 Die erweiterte Transaktionsschnittstelle 65
3.2.3.2 Die Unterstützung geschachtelter Transaktionen 66
3.2.3.3 Lange Transaktionsunterstützung 67
3.2.4 Vorteile und Restriktionen des Lösungsansatzes 67

4 Realisierung und Anwendung 69
4.1 Die Realisierung der vorgestellten Schnittstellen 69
4.1.1 Die Datenbankservergenerierung 69
4.1.2 Der direkte Persistenzansatz 70
4.1.3 Kollektionen und Referenzen 71
4.1.4 Die Fabrikrealisierung 73
4.1.5 Die Realisierung der verschiedenen Transaktionstypen 74
4.1.5.1 Die Umsetzung allgemeiner und erweiterter Transaktionen 74
4.1.5.2 Die Umsetzung geschachtelter Transaktionen 74

4.1.5.3 Die Umsetzung langer Transaktionen 75
4.1.6 Der Wrapper-Ansatz 76
4.2 Auswirkungen auf das Anwendungsszenario 77
4.2.1 Unterstützung der vollen Kooperation 79
4.2.2 Eingeschränkte Kooperation 80
4.3 Diskussion und Grenzen des Anwendungsszenarios 82

5 Zusammenfassung und Ausblick 83

Abbildungsverzeichnis 85

Tabellenverzeichnis 86

Literaturverzeichnis 87

Abkürzungen 91

A Ausschnitte aus der Realisierung der Basisklassen 93
A.1 Die Realisierung für den Einstiegsserver 93
A.2 Die Realisierung für persistente CORBA-Klassen 93
A.2.1 Direkter Persistenzansatz 94
A.2.1.1 Die Realisierung für Kollektionen 94
A.2.1.2 Die Realisierung für Fabriken 95
A.2.2 Die Realisierung mit dem Wrapperansatz 95

B Ausschnitte aus der Transaktionsrealisierung 98
B.1 Die allgemeine Transaktionsrealisierung 98
B.2 Die geschachtelte Transaktionsrealisierung 99
B.3 Die lange Transaktionsrealisierung 100

C Das Anwendungsszenario 102
C.1 Relevante IDL-Schnittstellen 102
C.2 Auszug aus dem relevanten Teil eines Whiteboardes 104
C.3 Auszug aus der Wbgroup-Implementierung 104

D Allgemeine Anleitung für die Nutzung 105

Thesen 107

Kapitel 1

Einführung

Fast alle Büroarbeitsplätze sind heute mit Computern ausgestattet, welche im zunehmenden Maße auch vernetzt sind. Die Ausnutzung dieser Ressourcen führt zur effektiveren Auslastung der Arbeitskraft Mensch. Viele zur Zeit noch notwendige Tätigkeiten können dadurch ganz oder teilweise entfallen, wie zum Beispiel die Dienstreise. Konferenzen und Diskussionsforen können vom Arbeitsplatz einer Person via Netz abgehalten werden und ersetzen die zeitaufwendigen Dienstreisen. Diese Personen sind verteilt auf unterschiedliche Standorte und wollen gemeinschaftlich Entscheidungen fällen bzw. Lösungen erarbeiten. Dazu reichen die heutigen Kommunikationsmechanismen, wie z.B. Telefon, noch nicht aus. Für die Entscheidungsfindung und die Erarbeitung von Lösungen sollen Kooperationswerkzeuge für die gemeinsame Bearbeitung von Dokumenten unterstützt werden. Leider ist die Hardware-Ausstattung des Arbeitsplatzes nicht das einzig Ausschlaggebende. Die Netzwerkmechanismen stehen zwar in sehr entwickelter Form zur Verfügung, aber gerade bei den Kooperationswerkzeugen und ihrer Unterstützung besteht ein sehr großer Nachholbedarf.

1.1 Motivation

Um diesen Nachholbedarf zu überwinden, hat sich das Forschungsgebiet des **Computer Supported Cooperative Work** (CSCW) entwickelt. Dieses muß Rücksicht nehmen auf eine heterogene Computerlandschaft. Begründet wird dieser Sachverhalt durch die Weiterverwendung existierender Systeme, die sich in den letzten Jahren auf den verschiedensten Rechnern unterschiedlichster Architektur entwickelt haben. Um dem Rechnung zu tragen, wurden Standards entwickelt, die diese Heterogenität verbergen. Ein solcher, der auch in dieser Arbeit eine tragende Rolle spielt, ist die von der **Object Management Group** (OMG) verabschiedete **Common Object Request Broker Architecture** (CORBA).

Die Kommunikation und die Kooperation sind die wesentlichen Elemente von CSCW-Anwendungen. Die Aspekte der Kommunikation in einem heterogenen Umfeld sind in CORBA transparent realisierbar. Darum sind Entwicklungen, die auf dem CORBA-Standard basieren, sehr offen. Durch Abstrahierung wiederkehrender Probleme lassen sich Dienste realisieren, die nicht nur für ein Anwendungsgebiet nutzbar sind. Viele dieser abstrahierten Komponenten sind durch einfache Mechanismen wiederverwendbar.

Die heutigen Standards der OMG bieten uns eine sehr geringe Unterstützung von persistenten Objekten. Weil in CSCW-Anwendungen Persistenz, aufgrund der asynchronen Bearbeitung von Dokumenten, eine wichtige Anforderung ist, müssen entsprechende Mechanismen zur Verfügung gestellt werden. Datenbanksysteme (DBS) bieten dafür die ideale Lösung an. Sie können eine große Menge von Daten speichern und bieten für den Zugriff auf diese eine Reihe von Möglichkeiten an. Zumeist werden diese Zugriffe auch noch intern optimiert, was die Informationssuche erheblich beschleunigt. Darum ist die Kombination von Datenbankmanagementsystemen (DBMS) und CSCW-Anwendugen sehr bedeutend. Da uns mit CORBA ein sehr guter Kommunikationsbus zur Verfügung steht, reicht es aus, wenn ein DBMS an diesen angeschlossen wird. Durch die Komplexität der Anwendungen reicht es in den meisten Fällen nicht aus, nur Standardfunktionalität bereitzustellen. Deswegen ist es wichtig, die Probleme, die mit der Kooperation zusammenhängen, schon in das DBMS zu integrieren und darüberhinaus den CORBA-Anwendungen nutzbar zu machen.

1.2 Das DICE-Projekt

Database In Cooperative Environments (DICE), siehe [FlMe97a, Kanu97, Kuna97, Ruiz97], wurde 1996 begonnen. Datenbanksysteme haben sich heutzutage in vielen Standardanwendungen bewährt. Aber im zunehmenden Maße ändern sich die Datenstrukturen, die in den heutigen Anwendungen zu speichern sind. Sie sind viel komplexer und umfangreicher geworden. Die Unterstützung dieser komplexen und multimedialen Datenstrukturen und die Verbindung von Techniken aus dem CSCW-Bereich mit denen aus Datenbanksystemen ist Ziel dieses Projektes. Darüber hinaus werden die Anforderungen an Datenbankmanagementsysteme für deren Einsatz in CSCW-Tools untersucht. Dabei werden die Schwerpunkte auf folgende Gebiete gelegt:

- **Erarbeitung eines flexibel verwendbaren und erweiterbaren Transaktionskonzeptes**
  Die Arbeiten im DICE-Projekt beschäftigen sich unter anderem damit, inwieweit der **Objekt Transaktion Service** (OTS) der OMG den Anforderungen an Transaktionen (TA) für die Nutzung in CSCW-Anwendungen gerecht wird. Dabei wird das lokale DBS in das verteilte, kooperative Transaktionsprotokoll auf der Basis einer Middleware integriert.

- **Bereitstellung von Datenbankfunktionalität für Workflow Management Systeme**
  Bestimmte Arbeitsabläufe zu automatisieren, ist Ziel von Workflow Management Systemen. In Datenbanksystemen existiert der Trigger-Mechanismus. Diese Technik soll auf Modellierbarkeit von Workflows untersucht werden.

- **Persistenzunterstützung in einer CORBA-Umgebung**
  In diesem Punkt ist nicht nur reine Persistenz einzuordnen, sondern auch die Abbildung der meisten Datenbankkonzepte. Dazu zählen auch die Transaktionsnutzung, die Sperrverwaltung und die Nutzung von Recovery-Verfahren innerhalb einer CORBA-basierten Middleware.

Diese Diplomarbeit ist innerhalb dieses Projektes in den dritten Unterpunkt mit dem Schwerpunkt der Unterstützung von Kooperation in CSCW-Anwendungen eingeordnet. Die Kooperationsunterstützung wird in den heutigen CSCW-Anwendungen zumeist noch in den Bereich der Middleware gelegt. Dabei werden Konsistenzsicherungsmaßnahmen in vielen Fällen entweder unzureichend angeboten oder dem Anwender überlassen. Das heißt, zum Teil geschieht die Synchronisation von Nutzern durch Aktionsrechtvergabe auf der *Graphical User Interface* (GUI)-Ebene. Dies ist insofern wichtig, weil damit eine Rückmeldung auf der Benutzerebene erreicht wird.

Datenbanksysteme bieten für die Gewährleistung der Konsistenz eine, auf einem fundierten Theoriemodell basierende, Lösung an. Aus diesem Grund wird die Verlagerung bzw. Verankerung dieser Mechanismen in das DBS Schwerpunkt dieser Arbeit sein. Ausgehend von einem Anwendungsszenario, siehe Abschnitt 1.4, wird eine erweiterbare Komponente zur Unterstützung von Datenbankfunktionalität entwickelt.

1.3 Grundlagen und Begriffe

In diesem Kapitel werden grundlegende Begriffe für das weitere Verständnis dieser Arbeit eingeführt. Dazu gehören die CSCW-Systeme, der CORBA-Standard und die **Object Database Adapter** (ODA)-Umsetzung im DICE-Projekt. Für die intensivere Beschäftigung mit der jeweiligen Thematik sind Literaturreferenzen angegeben.

1.3.1 Computer Supported Cooperative Work

In den letzten Jahren wuchs die Dezentralisierung und als nächste Stufe davon die Globalisierung von Unternehmen. Verbunden mit immer kürzeren Entwicklungszeiten für neue Produkte nimmt dadurch die Bedeutung von CSCW ständig zu. Bedingt durch leistungsfähige Infrastruktur werden die Techniken aus dem CSCW-Bereich zunehmend realisierbarer.

Die computergestützte kooperative Arbeit befaßt sich mit der Rechnerunterstützung mehrerer Personen bei ihrer Aufgabenbewältigung und macht diese effektiver. Die Problemlösung ist aufgrund der heutigen Komplexität ein arbeitsteiliger Prozeß. Darum müssen die darin involvierten Personen miteinander kommunizieren und kooperieren. CSCW-Anwendungen unterstützen diesen Prozeß. Um diese CSCW-Anwendungen zu klassifizieren, wird die in Abbildung 1.1 angegebene Raum-Zeit-Matrix verwendet.

Personen, die miteinander kommunizieren, können sich entweder im selben Raum oder an verschiedenen Orten befinden. Diese Eigenschaft wird in der Raumdimension der Matrix dargestellt. Ähnlich zu dieser räumlichen kann auch eine zeitliche Verteilung auftreten. Geschieht die Kommunikation zum gleichen Zeitpunkt z.B. Telefonat, spricht man von synchroner, anderenfalls von asynchroner Kommunikation z.B. Mailsysteme. Viele CSCW-Anwendungen sind Mischformen, die asynchrone bzw. synchrone sowie entfernte und benachbarte Arbeit unterstützen. In [BoSc95, BMST95] werden einige von ihnen vorgestellt.

Im Zusammenhang von CSCW-Anwendungen spielt die Konsistenz von Daten eine wichtige Rolle und beruht auf den folgenden Begriffen.

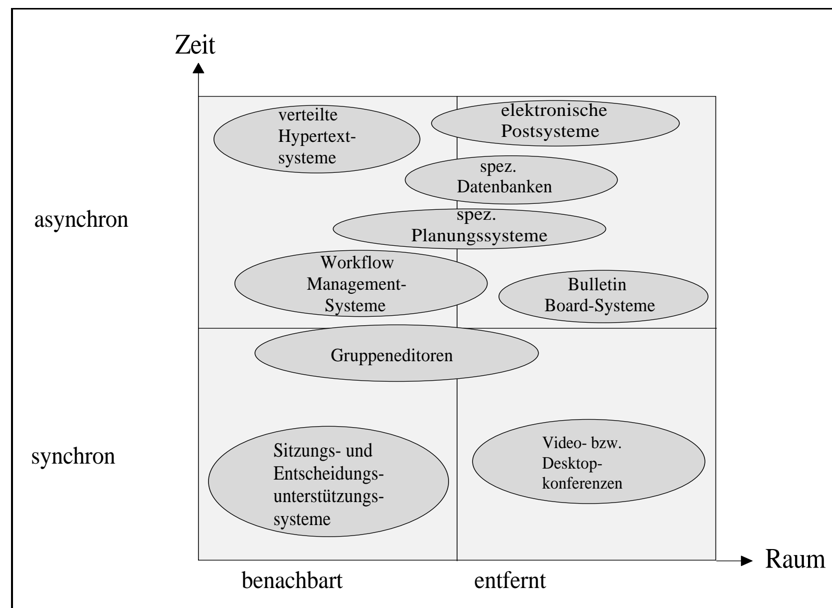

Abbildung 1.1: Klassifikation der CSCW-Systeme nach Raum und Zeit [BMST95]

- **Präzedenz:** Die Aktionen der Teilnehmer an einem CSCW-System besitzen eine Reihenfolge und dementsprechend ist das Ergebnis aller Aktionen.
- **Konvergenz:** Werden keine Änderungen gemacht, konvergiert das Ergebnis in einen Ruhezustand (Fixpunkt). Dies heißt, jeder Teilnehmer sieht den gleichen Zustand zu diesem Zeitpunkt.

In Transaktionen, siehe Kapitel 1.3.3, existieren semantisch äquivalente Begriffe.

1.3.2 Der CORBA-Standard

In diesem Kapitel wird ein Einblick, siehe auch in [MoZa95], in den CORBA-Standard gegeben. Eine Spezifikation wird in [OMG96] gegeben. Ziel dieses Standards ist die Nutzung, die Bereitstellung und die Wiederverwendung von verteilten Anwendungen. Dazu werden Komponenten definiert, welche die Handhabung verteilter Applikationen wesentlich vereinfachen. Dies geschieht durch Nutzung der Objektorientierung wie abstrakte Datentypen, Vererbung, Polymorphismus usw. Mit diesen Mitteln kann von den Unterschieden in der Nutzung unterschiedlicher Programmiersprachen und Betriebssysteme abstrahiert werden. Abbildung 1.2 zeigt die **Object Management Architecture** (OMA), die aus vier Teilen besteht.

- Die **Object Services** stellen Dienste zur Verfügung, die durch alle Objekte genutzt werden können. Darunter fallen Leistungen wie Namens-, Persistenz- und Sicherheitsdienste. Eine Architektur und die bereits verabschiedeten Spezifikationen werden in [OMG94a, OMG94b] vorgestellt. Der Persistence Object Service (POS) spezifiziert allgemeine Schnittstellen zur Unterstützung von Persistenz in einer verteilten CORBA-Umgebung. Damit ist es möglich, Objektzustände zu speichern und wieder zu restaurieren.

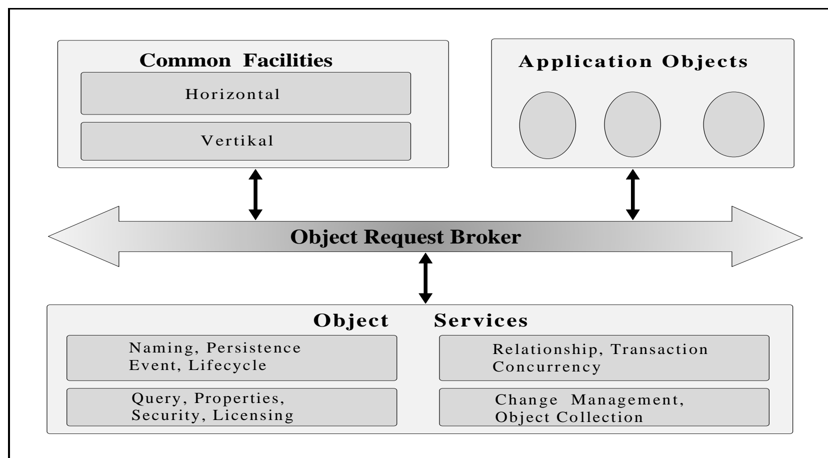

Abbildung 1.2: Object Management Architecture [OMG90]

- Die **Common Facilities** sind Komponenten, welche anwendungsspezifische standardisierte Schnittstellen bereitstellen, z.B. für die Finanzbuchhaltung. Sie werden in [OMG95a, OMG95b] beschrieben.

- Die **Application Objects** sind spezielle für das jeweilige Problem entwickelte Anwendungen. Um ihre Aufgaben zu realisieren, nutzen sie die *Object Services* und *Common Facilities*.

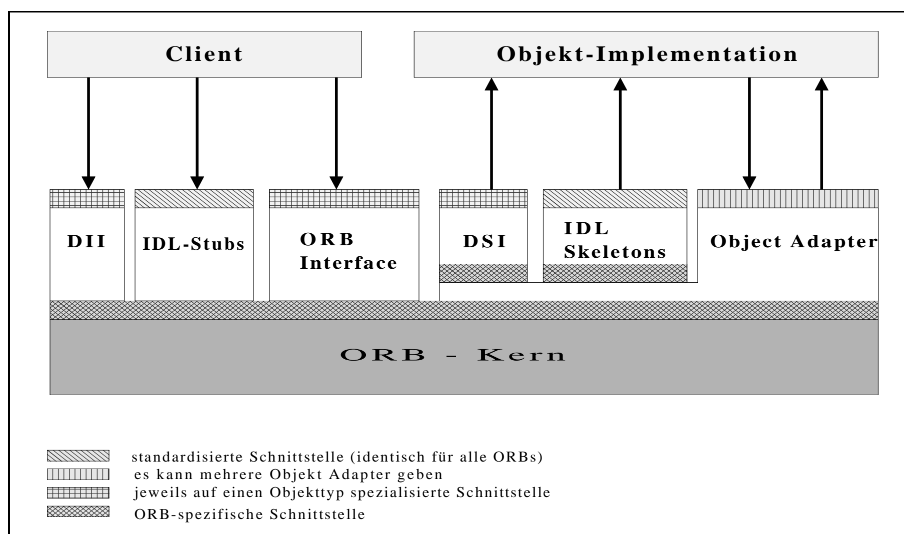

Abbildung 1.3: Common Object Request Broker Architecture [OMG96]

- Der **Object Request Broker** (ORB) ist das Medium, das die Kommunikation zwischen den oben erwähnten Komponenten realisiert. Er ist verantwortlich für die transparente Verteilung der Objekte im Netz. Um diese Komponente zu konkretisieren, wird die Common Object Request Broker Architecture eingeführt. In Abbildung

1.3 ist die Struktur eines CORBA-konformen ORB's angegeben.

Der Client ist der Nutzer verteilter Objekte. Dafür kann er zwei Schnittstellen, zum einen eine statische, die **Client-Stubs**, durch direkten Aufruf einer Methode und zum anderen eine dynamische, das **Dynamic Invocation Interface** (DII), verwenden. Diese leiten den Request über den **ORB-Kern** an den **Object Adapter** weiter. Dieser sucht das passende Objekt und aktiviert es wenn nötig. Danach wird der Request über die statische Form der **Skeletons** oder dynamisch über das **Dynamic Skeleton Interface** (DSI) an die Objektimplementierung weitergeleitet. Desweiteren wird die programmiersprachenunabhängige **Interface Definition Language** (IDL) und deren **Binding** an verschiedene Programmiersprachen definiert. Somit kann jeder Client- bzw. Server-Entwickler die vom ihm bevorzugte Sprache verwenden.

1.3.3 Transaktionen und Sperren

Transaktionen besitzen die Aufgabe, in Beziehung stehende Operationen zusammenzufassen. Gerade bei parallel ablaufenden Transaktionen treten Probleme auf, weil eine Transaktion das Ergebnis einer gleichzeitig ablaufenden TA beeinflussen kann. Unter dem **"lost update"**-Problem wird verstanden, daß zwei parallel ablaufende Transaktionen Änderungen an dem selben Objekt machen. Die Modifikation der zuerst schreibenden Operation geht hierbei verloren. Bei der **"dirty read"**-Anomalie liest eine Transaktion Werte, die gleichzeitig innerhalb einer anderen modifiziert werden und arbeitet mit diesen. Dabei besitzen diese Werte nur vorläufigen Charakter, weil die modifizierende TA z.B. noch abgebrochen werden kann. Um diese und andere Probleme zu beheben, werden die ACID-Transaktionen verwendet. In diesem Zusammenhang werden gewisse Eigenschaften von Transaktionen definiert:

- **Atomarität** (A): Sie beinhaltet, daß eine Transaktionen entweder ganz oder gar nicht ausgeführt wird.
- **Konsistenz** (C): Diese bedeutet, daß nach Beendigung einer Transaktion alle Integritätsbedingungen, die auf der Datenbank definiert sind, erfüllt sind, z.B. daß keine Objektreferenz einen unbestimmten Wert enthält.
- **Isolation** (I): Eine Transaktion läuft parallel zu anderen ab. Mit Isolation ist gemeint, daß das Ergebnis einer Transaktion nicht von der Ausführung anderer Transaktionen abhängt und entspricht damit der alleinigen seriellen Ausführung. Das heißt, Transaktionen dürfen einander nicht beeinflussen.
- **Dauerhaftigkeit** (D): Es wird garantiert, daß das Ergebnis einer erfolgreich beendeten Transaktion dauerhaft in der Datenbank (DB) gespeichert ist und nicht z.B. durch nachträgliches Rücksetzen verändert wird.

Um diese Eigenschaften zu gewährleisten, besitzen die Datenbanksysteme Komponenten, wie unteranderem die Transaktionsverwaltung und die Sperrenverwaltung. Auf zwei weitere soll im folgenden eingegangen werden.

Concurrency Control-Komponente

Die oben angeführten Probleme können vermieden werden, wenn keine Transaktion parallel zu einer anderen abläuft. Die Reihenfolge des Ablaufs der Transaktionen wird vorher festgelegt. Diese Situation ist eine **serielle** Ausführung der Transaktionen. Da dies wegen schlechter Auslastung zu restriktiv ist, können andere verzahnte Operationsreihenfolgen zugelassen werden, wenn diese gewissen Bedingungen genügen.

- Eine verzahnte Operationsreihenfolge ist **view serializable** [AbAg92], wenn eine mögliche serielle Ausführung der Transaktionen existiert und wenn für die verzahnte Operationsreihenfolge sowie für die serielle Ausführung jede beteiligte Transaktion die gleichen Werte liest und die gleichen Endwerte liefert. Die Überprüfung auf View-Serialisierbarkeit ist NP-vollständig. Daraus folgt, daß dieser Begriff keine praktische Relevanz besitzt.

- Ein Konflikt zwischen zwei Operationen auf dem gleichen Objekt tritt auf, wenn mindestens eine von beiden eine Schreiboperation ist. Eine verzahnte Operationsreihenfolge ist **conflict serializable** [AbAg92], wenn eine mögliche serielle Ausführung der Transaktionen existiert und in beiden Fällen die in Konflikt stehenden Operationen in der selben Reihenfolge ausgeführt werden.

Ein **pessimistisches** Concurrency Control-Verfahren gewährleistet die Konfliktserialisierbarkeit, indem es überprüft, ob eine auszuführende Operation in Konflikt steht mit Operationen anderer Transaktionen. Wenn dies der Fall ist, wird die Operation verzögert oder die entsprechende Transaktion abgebrochen. Die meisten pessimistische Verfahren arbeiten auf der Grundlage von Sperren.

Sperren

Eine Sperre schützt ein Objekt vor dem gleichzeitigen Zugriff mehrerer Zugriffsoperationen, wenn diese in Konflikt zueinander stehen. In diesem Fall darf nur eine Operation auf das Objekt zugreifen. Stehen die Operationen nicht in Konflikt zueinander, so ist gleichzeitiger Zugriff erlaubt. Eine Sperre erlaubt dem Inhaber, bestimmte Operationen auf dem Objekt auszuführen. Dabei unterscheidet man bei einer Sperre die Granularität. So existieren Sperren auf alle Instanzen einer Klasse oder nur auf einer Instanz der Klasse. Im einfachsten Fall gibt es zwei Arten von Sperren, eine *shared*-Sperre erlaubt den nur lesenden Zugriff und eine *exclusive*-Sperre erlaubt lesenden sowie schreibenden Zugriff. Dies bedeutet, eine Sperre wird gesetzt, wenn auf dem Objekt eine entsprechende Operation ausgeführt werden soll. Äquivalent zu den Operationen existieren auch zwischen Sperren Konflikte, siehe Tabelle 1.1. Daraus folgt, daß eine Sperre auf einem Objekt nicht gewährt werden darf, wenn eine inkompatible, gekennzeichnet durch ein Minuszeichen, auf diesem gesetzt ist.

	shared	exclusive
shared	+	-
exclusive	-	-

Tabelle 1.1: Kompatibilitätsmatrix für Paare von Sperren

Eine Synchronisation erfolgt nun aufgrund der gesetzten Sperren. Leider kann die Serialisierbarkeit nicht immer garantiert werden. Darum werden Protokolle eingeführt, die gewisse Restriktionen an die Sperrenvergabe und -freigabe stellen. Als Beispiel sei hier das **Zwei-Phasen-Sperrprotokoll** genannt. Dieses verlangt, daß keine Sperrenvergabe nach einer Sperrenfreigabe erfolgt.

Ein **optimistisches** Concurrency Control-Verfahren überprüft am Ende einer Transaktion, ob ein Konflikt aufgetreten ist und bricht im negativen Fall ab, siehe Kapitel 2.1.1.

Recovery-Komponente

Die Recovery-Komponente befaßt sich einerseits mit der Rücksetzung schon gemachter Änderungen und andererseits mit der Wiederholung von verloren gegangenen Änderungen. Diese Mechanismen müssen greifen, wenn z.B. Systemfehler oder der Abbruch einer Transaktion, von deren Ergebnisse andere parallele Transaktionen abhängen, eintreten. Der Abbruch einer Transaktion kann mehrere Ursachen haben. Es kann eine Operation innerhalb der Transaktion fehlschlagen z.B. durch Nichteinhalten der Konsistenzbedingungen. Die Hauptaufgabe beim Eintreten solcher Ereignisse ist, die Datenbank wieder in den letzten konsistenten Zustand zu bringen.

Um einen tieferen Einblick in diese Thematik zu erhalten, kann in [BGH87, GrRe92] nachgelesen werden.

1.3.4 ODA im DICE-Projekt

Aufsetzend auf dem CORBA- und Object Database Management Group (ODMG)[1] –Standard, siehe [Catt97], soll ein Object Database Adapter (ODA) entwickelt werden, der es ermöglicht, unabhängig vom verwendeten Datenbanksystem zu sein und trotzdem Datenbankfunktionalität bereitzustellen. Die Transparenz schließt auch unterschiedliche Modelle (relational bzw. objektorientiert) ein. Es existieren verschiedene Realisierungsvorschläge für einen ODA, siehe Tabelle 1.2.

Kurzbezeichnung	Beschreibung	Literatur
CORM	- CORBA Object Relational Mapper - Zugriff auf relationale Datenbestände - Abbildung Relationen- in Objektmodell	[Chri96]
OMG-Objektservice	- Austausch vom BOA durch einen ODA	[Catt97]
OOSA	- kommerzieller Orbix-ObjectStore Adapter	[IONA98]
Telemed-ODA	- transparente und orthogonale Persistenz - BOA-Erweiterung zur Interpretierung von CORBA-Referenzen auf persistente Objekte	[Reve96]
DICE-ODA	- siehe Kapitel 1.3.4	[Neuw97]

Tabelle 1.2: existierende Object Database Adapter

Der von der ODMG gemachte Vorschlag, den Basic Object Adapter (BOA) durch einen ODA zu ersetzen, ist zur heutigen Zeit nicht sinnvoll [Neuw97], weil kein bestehendes

[1] ein Normungsgremium für die Standardisierung von objektorienterten Datenbanksystemen

DBS bislang CORBA-Referenzen, welche auch für die Identifikation von Objekten aus der Datenbank genutzt werden sollen, interpretieren kann. Ein weiterer Nachteil ist, daß nur reine Persistenzmechanismen unterstützt werden. Der Client hat keine Möglichkeit in die Transaktionssteuerung einzugreifen, weil der ODA in diesem Fall selber festlegt, was eine Transaktion ist[2].

Der ODMG-Standard definiert mit der Objekt Definition Language (ODL) eine Sprache mit der Objekte modelliert werden können. Sie erweitert die IDL-Sprache der OMG um einige datenbankspezifische Konstrukte und ist deshalb mächtiger. Die Object Query Language (OQL) ist die Anfragesprache der ODMG. Der in DICE verwendete ODA soll zum einen alle ODL-Konstrukte und zum anderen die dynamische Nutzung der OQL unterstützen.

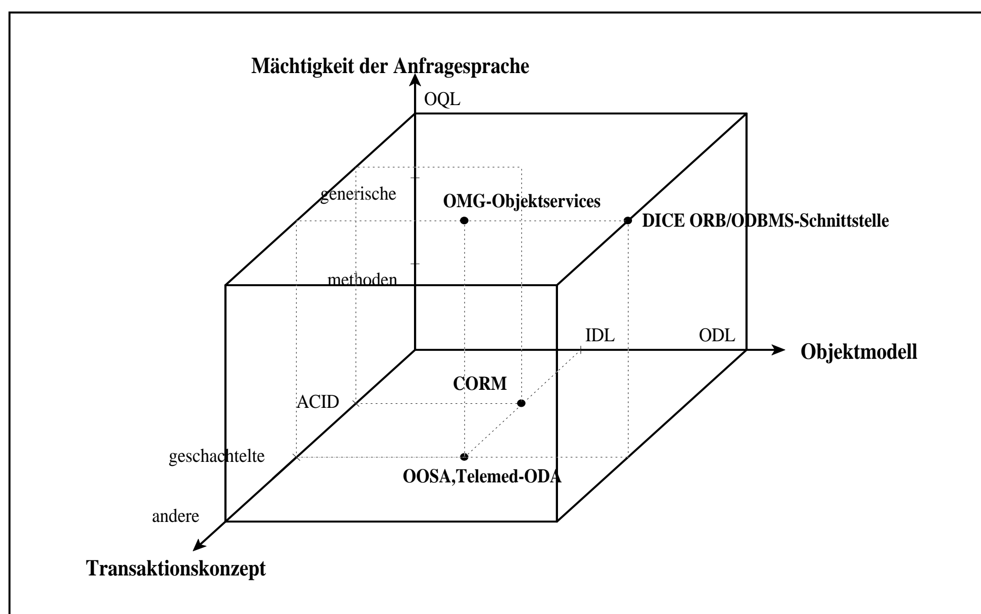

Abbildung 1.4: Einordnung der existierenden Prototypen nach [Neuw97]

Die Abbildung 1.4 zeigt die Einordnung der verschiedenen ODA-Realisierungen bezüglich der Kriterien Transaktion, Anfrageunterstützung und dem Objektmodell. Es sind unteranderem die Adapter CORBA Object Relational Mapper (CORM), welcher relationale Daten bereitstellt und der Telemed-ODA, der am Los Alamos National Laboratory [Reve96] entwickelt wurde, eingetragen. Welche Funktionalität mit dem ODA im DICE bereitgestellt werden soll, geht demnach auch aus Abbildung 1.4 hervor.

Die Abbildung 1.5 zeigt die Architektur, welche im DICE-ODA verwendet wird. In dieser Abbildung ist der ODA als CORBA-Server realisiert. Der Adapter ist Server bezüglich eines CORBA-Clients, aber wiederum auch Datenbank-Client. Somit können die Vorteile von CORBA und des Datenbanksystems miteinander kombiniert werden. CORBA stellt die Transparenz- und das DBS die Datenbankfunktionalität zur Verfügung. Die Datenbankfunktionalität wird über entsprechende IDL-Schnittstellen gekapselt. Somit kann

[2]Die Abarbeitung einer Methode kann eine solche Transaktionseinheit sein

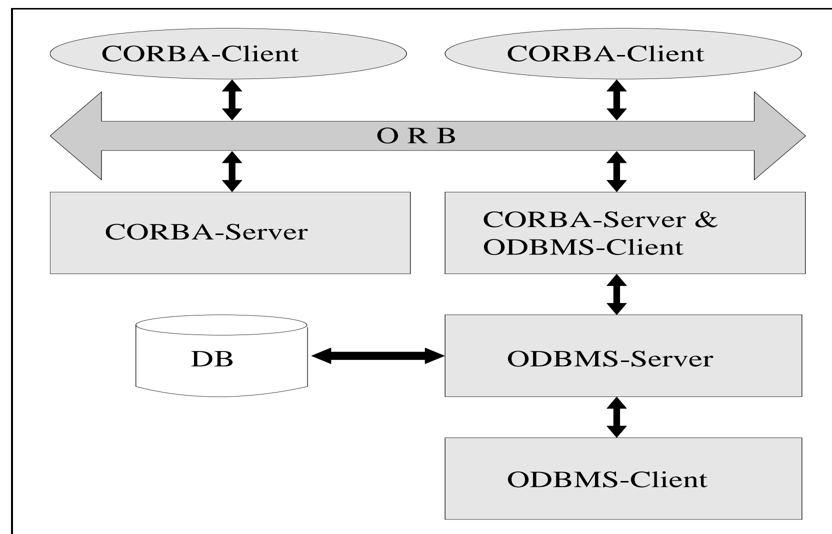

Abbildung 1.5: Prozeßarchitektur [Neuw97]

jeder CORBA-Server sowie Client über diese Schnittstelle die gewünschte Funktionalität verwenden.

Die folgenden Punkte geben eine Überblick in die Realisierungen des DICE-ODA's. Sie bilden die Grundlage für die Erweiterungen bezüglich der Konzepte aus Kapitel 3 und der Realisierungen angegeben in Kapitel 4.

- **orthogonale, implizite Persistenz:**
  Dieser Punkt bedeutet, daß innerhalb der verteilten Umgebung aus Client-Sicht kein Unterschied zwischen normalen CORBA-Objekten und solchen, die in einem DBMS gespeichert sind, bestehen. Dies schließt die Existenz von `store`- und `restore`-Routinen aus. Weiterhin müssen beliebige CORBA-Objekte persistent realisierbar sein.

  Lösungsansatz:
  Es existieren generell zwei Möglichkeiten CORBA-Objekte persistent zu machen. Die erste besteht in der direkten Persistenz der CORBA-Objekte[3] selbst, was aber zu einigen Nachteilen führt. Die CORBA-Objekte besitzen nicht nur die Eigenschaften, die innerhalb des IDL-Interfaces beschrieben wurden, sondern erben noch eine Menge von Daten, die für die Netzwerkkommunikation erforderlich sind. Damit erhöht sich der Speicherplatzbedarf für das jeweilige Objekt unnötig. Für jedes CORBA-Objekt wird ein Zähler mitgeführt, welcher die Anzahl der Referenzen auf dieses angibt. Jedesmal wenn eine Referenz erzeugt bzw. freigegeben wird, wird der Zähler inkrementiert oder dekrementiert. Für beide Operationen muß das betreffende Objekt in der Datenbank exklusiv gesperrt werden, was wiederum andere Clients bei der Arbeit auf diesem behindert.

  Um dieses Problem zu lösen, werden im DICE-ODA nur die Implementierungssobjekte persistent gemacht. Dazu wird der TIE-Ansatz[4] innerhalb von CORBA ge-

[3]zwingend erforderlich bei Nutzung des Vererbungsansatzes
[4]auch Delegationsmethode genannt

nutzt. Hierbei wird streng zwischen den kommunikations- und den IDL-spezifischen Daten und Funktionen getrennt. In diesem Fall ist nur der letztere Teil persistent in der Datenbank gespeichert. Dadurch muß unterschieden werden, ob der Speicher für das Implementierungsobjekt oder für das transiente CORBA-Objekt freigegeben wird. Für das persistente Implementierungsobjekt ist dazu eine spezielle Methode `ReleaseInstance()` erforderlich. Für das transiente CORBA-Objekt ist weiterhin die traditionelle `_Release`-Routine zu verwenden. Ein Nachteil dieser Methode ist, daß die Persistenztransparenz aufgegeben wird.

Weiterhin wird nicht verhindert, daß mehrere transiente CORBA-Objekte auf das selbe persistente Implementationsobjekt zeigen. Dies kann gelöst werden durch Protokollierung und Zuordnung der CORBA-Referenz zur Datenbankreferenz für die aktiven Objekte. Damit wird bei der Erzeugung der Instanz eines transienten CORBA-Objektes in der tabellarischen Umsetzung nachgeschaut, ob für das persistente Implementationsobjekt schon ein CORBA-Objekt existiert und in diesem Fall die entsprechende CORBA-Referenz zurückgegeben.

Clients arbeiten nur mit CORBA-Objekten und möchten deren implizite Persistenz durch deren Konvertierung in einen String erreichen. Leider sind CORBA-Objektreferenzen nicht persistent, so daß die Zuordnung Implementierungsobjekt zum transienten CORBA-Objekt verloren geht, wenn beide aus dem Speicher und somit das Paar auch aus der oben erwähnten Tabelle entfernt werden. Dieses Problem kann gelöst werden, indem in der Datenbank eine Tabelle gehalten wird. Diese ist nach der CORBA-Referenz gehasht und muß durch die entsprechenden Aktivierungsroutinen berücksichtigt werden.

- **Unterstützung von Kollektionen und Anfragen:**
  Viele objektorientierte Datenbanksysteme bieten innerhalb eines C++–Bindings eine Möglichkeit, auf Kollektionen zuzugreifen und auf diese Anfragen zu stellen. CORBA bietet über ein `sequence<element>`–Konstrukt ein Alternative zu Kollektionen an. Leider ist eine solche Umsetzung von Kollektionen aus Performanzgründen [Neuw97] nicht akzeptabel, da alle Objekte in der Sequenz aktiviert sein müssen.

  Lösungsansatz:
  Im ODMG-Standard [Catt97] und dem darin definierten C++–Binding werden unterschiedliche Typen von Kollektionen unterstützt. Leider ist das darin verwendete Template-Konzept [Stro92] nicht innerhalb von IDL-Konstrukten anwendbar. Aus diesem Grund wird für jedes Paar (Kollektionstyp, Interface-Typ) eine eigene Interface-Definition mit entsprechenden Funktionen (z.B. `query(in string predicate)`, `add_element()`, ...) bereitgestellt. Die `query`-Methode liefert konform zum ODMG-Standard eine Unterkollektion der aufrufenden Kollektion. Für die Unterstützung von beliebigen Anfragen ist ein Interface zur Verfügung zu stellen, ähnlich dem OQL-Binding an die Programmiersprache C++.

- **Beziehungen:**
  Aus dem Entity-Relationship-Modell sind uns die 1:1-, 1:n- und n:m-Beziehungen [HeSa95] bekannt. So ist die Vater:Sohn-Beziehung eine 1:n-Beziehung. Ein Vater kann bis zu n Söhne besitzen und einem Sohn ist ein Vater zugeordnet. Diese Beziehungen werden **bidirektional** genannt. Zur Unterstützung solcher Relationships

gehört die Überprüfung der referentiellen Integrität. Das heißt, wird ein Sohn gelöscht, muß er automatisch aus der Söhne-Kollektion des Vaters genommen werden. Eine einfachere Form sind die **unidirektionalen** Beziehungen. Der Unterschied besteht darin, daß keine Rückverweise existieren.

Lösungsansatz:
Umgesetzt werden Beziehungen durch Attribute. Innerhalb der Implementation zum Setzen bzw. Lesen des Attributs werden vom Datenbanksystem bereitgestellte parameterisierte Kollektion-Templates genutzt. Die Parameter sind der Typ des Beziehungsobjektes und in Stringform das inverse Beziehungsattribut. Bei der Abbildung unidirektionaler Beziehungen entfällt der Parameter für das inverse Beziehungsattribut.

- **Transaktionen:**
  In vielen Anwendungen möchte man eine Anzahl von Operationen innerhalb einer Transaktion ausführen. Im CORBA-Umfeld wird dabei unterschieden zwischen **verteilten** und **lokalen** Transaktionen. Zumeist existiert eine Reihe von Servern, die Objekte zur Verfügung stellen. Wenn innerhalb einer Transaktion Operationen von Objekten nur eines Servers genutzt werden, wird diese als **lokal** bezeichnet. Werden dagegen Operation von Objekten unterschiedlicher Server aufgerufen, heißt sie **verteilt**. Der Begriff der lokalen sowie verteilten Transaktion kann um das ACID-Prinzip erweitert werden. Eine lokale ACID-Transaktion ist somit eine lokale Transaktion, die die in Kapitel 1.3.3 aufgeführten Eigenschaften besitzt. Alle im Kapitel 2 vorgestellten Transaktionskonzepte können so um den Lokalitätsbegriff erweitert werden. Ein Beispiel für eine verteilte TA ist das Buchen einer Reise. Nur wenn jeweils beide Teiltransaktionen für die Buchung des Fluges und des Hotels positiv verlaufen sind, ist die verteilte TA auch positiv verlaufen.

  Lösungsansatz:
  Für verteilte Transaktionen ist der ORB-Implementator verantwortlich, der den in [OMG94b] spezifizierten Object Transaction Service bereitstellen muß. Innerhalb jedes Servers, in welchem Teiltransaktionen ablaufen, muß eine Schnittstelle bereitgestellt werden, damit ein verteiltes Zwei-Phasen-Commit-Protokoll realisiert werden kann. Diese Schnittstelle stellt Grundfunktionen für Transaktionen und ihre Operationen bereit.

  Die lokalen Transaktionen sind verantwortlich, um eine Anzahl von IDL-Operationen, deren Objekte sich innerhalb eines CORBA-Servers befinden, zu einer semantischen Einheit zusammenzufassen. Dazu werden die Transaktionskonzepte des Datenbanksystems innerhalb eines Transaktions-Interfaces gekapselt.

- **Sperrenverwaltung:**
  Der Client möchte auf einem Objekt arbeiten. Dazu braucht er Rechte wie das Schreiben und Lesen. Dies erreicht man über die Sperrenzuordnung zu Objekten. Die Verstärkung oder Abschwächung von Sperren sollten unterstützt werden.

  Lösungsansatz:
  Für die Sperrenverwaltung erhält jedes Interface eine spezielle Routine, die als Parameter den Lockmodus enthält. Eine Alternative ist, innerhalb einer Transaktion

eine Methode zur Verfügung zu stellen, die einen Default-Lockmodus setzt. In beiden Fällen wird die Sperrenverwaltung an das DBS weitergeleitet.

Probleme und Restriktionen

- Zwingend erforderlich für die letzten beiden Punkte ist eine eindeutige Zuordnung vom Client zum ODA-Server, was durch den nicht OMG-konformen Aktivierungsmodus `perclient` von Orbix realisiert werden kann. Konforme Aktivierungsmodi sind `permethod`, `perobject` und `shared`. In den ersten beiden Aktivierungsarten wird zum einen für jeden Methodenaufruf oder zum anderen für jedes Objekt ein neuer Server gestartet. Damit ist aber keine Zusammenfassung von Operation auf verschiedenen Objekten in einer Transaktion möglich, da kein Transaktionsobjekt im Kontext zu den Operationen steht.

  Bei der `shared`-Aktivierung werden alle Objekte in einem Server behandelt. Somit befindet sich jedes Client-Transaktionsobjekt im selben Server, und die Operationszuordnung zum Client-Transaktionsobjekt ist nicht mehr gegeben. Genauso ergeht es der unterschiedlichen Sperrenzuordnung pro Client für das selbe Objekt.

- Die Abbildung von Beziehungen zwischen CORBA-Objekten, die sich nicht im selben Server befinden, ist nicht vollständig gelöst. Durch Einführung einer speziellen Beziehungsklasse sind diese Beziehungen in der folgenden Weise abbildbar. Die Beziehungsklasse besitzt eine Zustandsvariable vom Typ String, in der die CORBA-Objektreferenz als Zeichenkette abgespeichert ist, und zwei Methoden zum Setzen und Lesen der Beziehung. Beim Setzen wird die CORBA-Referenz in eine Zeichenkette umgewandelt und in die Zustandsvariable eingetragen. Beim Lesen wird dieser Prozeß wieder umgekehrt. Leider kann das Datenbanksystem die referentielle Integrität der Beziehung nicht mehr überprüfen, weil es den String nicht mehr als Beziehung interpretieren kann. Für eine zufriedenstellende Lösung kann daher nur auf die Spezifikation des Relationshipservices der OMG gewartet werden.

- Generische Anfragemöglichkeiten werden nicht unterstützt, weil die dynamische Ergebnistyperzeugung nicht vorgesehen ist. Dazu sind weitergehende Betrachtungen in Bezug auf Einsatz des Interfacerepositories, welche die dynamische Typenerzeugung innerhalb eines CORBA-Servers zuläßt, notwendig.

- Viele Datenbanksysteme stellen nur einen begrenzten Funktionsumfang gerade bei der Unterstützung von Anfragen, Kollektionen, Transaktionen und Sperren bereit. So unterstützt Objectstore nur Sperren auf Seitenebene und nicht auf Objektebene. Der ODA-Implementator ist aber auf diese Funktionalität angewiesen, oder er verlagert diese aus dem Datenbanksystem durch Eigenumsetzung.

In diesem Kapitel wurden die Realisierung von Basiskonzepten und Restriktionen in einem allgemeinen DICE-ODA beschrieben. Zusätzlich zur Bereitstellung von erweiterten Konzepten sollen diese Basiskonzepte und die Verringerung der Restriktionen realisiert und in dem folgenden Anwendungsszenario integriert werden.

1.4 Das Anwendungsszenario

Dieses Anwendungsszenario ist in das TOBACO-Projekt [DLMR97] des ZGDV eingegliedert. In diesem werden Konzepte für CORBA-basierte Dienste zum synchronen und kooperativen Arbeiten entwickelt und realisiert. Diese Mechanismen werden in der 3D-Modellierung bzw. der Beschränkung auf den zweidimensionalen Fall angewendet, siehe Abbildung 1.6.

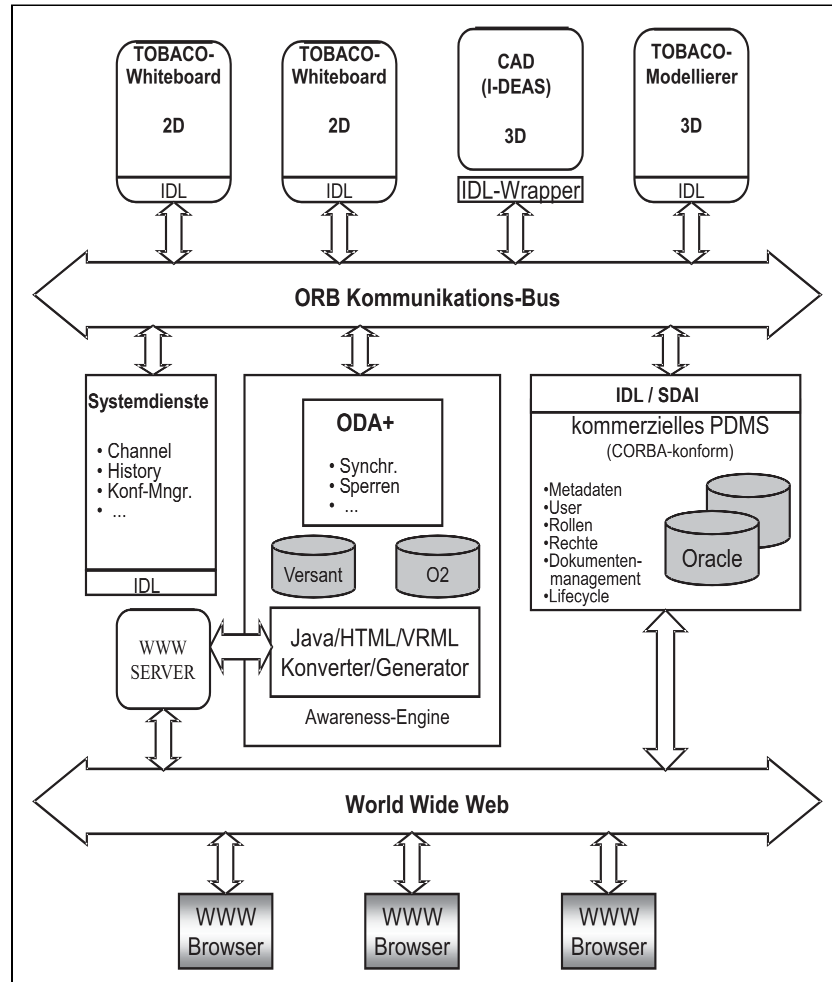

Abbildung 1.6: TOBACO-Architektur

Die Vision dieser Architektur ist, daß neben Konferenzdaten und Gruppendaten auch die 3D-Modellierungsdaten durch einen **erweiterten ODA** (ODA+) zur Verfügung gestellt werden, so daß jeder Modellierungsschritt auf persistenten CORBA-Objekten[5] realisiert wird. Über eine Ankopplung der Datenbank an das WWW, sollen entsprechend aufbereitete Daten auch auf einem WWW-Browser anzeigbar sein. Das Datenbanksystem muß in dieser Architektur verschieden Aufgaben realisieren.

- Abbildung der 3D- und 2D-Daten in Datenbankobjekte

[5]richtig transientes CORBA-TIE mit persistentem Implementierungsobjekt

- Abbildung der Gruppendaten (Rollen, Konferenzen, ...)
- Aktionsverwaltung der verschiedenen kooperativen Modellierer

Im Whiteboard (WB) wird die Komplexität der Daten auf den 2D-Fall beschränkt. Die Whiteboard-Architektur mit ODA+ ist in Abbildung 1.7 dargestellt. Die Gruppenkommunikation erfolgt hier über ein spezielles Objekt.

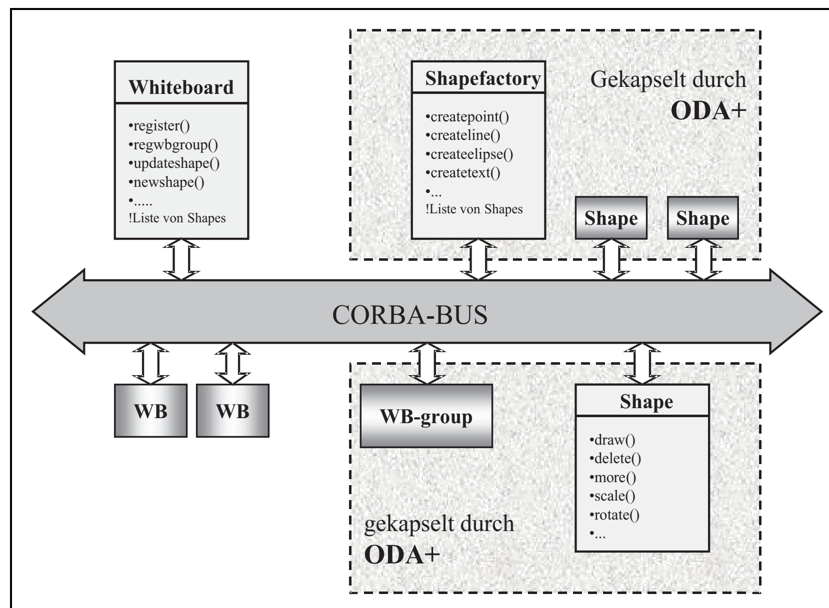

Abbildung 1.7: Whiteboard-Architektur

Das Whiteboard ist die Benutzungsoberfläche, die dem Anwender zur Verügung steht, um mit dem System zu interagieren. Beim Starten des Whiteboards bindet es sich an eine Shapefactory und ein WB-Group-Objekt, falls diese existieren. Ansonsten wird eine neue Shapefactory bzw. ein neues WB-Group-Objekt erzeugt. Die Shapefactory ist für die Verwaltung von Shapes[6] verantwortlich. Zu diesem Zweck hält sie sich eine Liste der erzeugten Shapes. Das WB-Group-Objekt ist für die Verwaltung der Whiteboards im System zuständig und hält sich dafür eine Liste der angemeldeten Whiteboards. Bei der Erzeugung einer 2D-Primitive wird vom Whiteboard eine Nachricht an die Shapefactory gesendet. Diese erzeugt das entsprechende Shape-Objekt und fügt es in die Liste der Shapes ein und informiert das WB-Group-Objekt über die Shape-Erzeugung. Dieses leitet das erzeugte Shape-Objekt an alle eingetragenen Whiteboards weiter und realisiert damit einen **Multicast**. Das Whiteboard trägt dieses in seine lokale Liste der aktuell zu zeichnenden Shapes ein ruft nun die `draw()`-Routine auf dem Shape-Objekt und übergibt seine Objektreferenz. Das Shape-Objekt ruft jetzt die `draw()`–Routine des Whiteboards auf, wodurch letztendlich auf allen Whiteboards die Änderung zu sehen ist.

Dem zu entwickelnden ODA kommt die Aufgabe zu, alle Shapefactories mit ihren Shapes persistent zu machen. Dazu soll auf den Objekten über geeignete Transaktionsmechanismen zum einen synchrone Arbeit, daß heißt jeder Whiteboard-Nutzer arbeitet zur gleichen

[6]Diese sind 2D-Grundprimitiva

Zeit mit diesem, aber auch zum anderen die asynchrone Arbeit erlaubt werden. Diese Arbeit befaßt sich mit der Aufgabe diesen erweiterten ODA zu realisieren.

1.5 Anforderungen des ODA+ im Anwendungsszenario

Im kooperativen Umfeld reichen die Leistungen eines Object Database Adapters, die im Kapitel 1.3.4 vorgestellt worden sind, nicht aus. Deshalb muß vom ODA Funktionalität gekapselt werden, die den kooperativen Anforderungen gerecht wird. Nachfolgend werden weitergehende zu realisierende Konzepte genannt.

- **Umsetzung der Datenstrukturen aus dem Whiteboard und dem CSCW-Bereich**
  Im Anwendungsszenario werden für die Darstellung der Graphiken bestimmte Primitive verwendet. Diese sowie die Konferenzdaten und auch Rollendaten sollten in entsprechende Datenbanktypen abgebildet werden und mittels des Object Database Adapters bereitgestellt werden.

- **Transaktionen**
  Zum einen müssen die Möglichkeiten lokaler Transaktionen erweitert werden. Hierarchien, bedingt durch Zerlegung von Aufgaben, kommen in der Gruppenarbeit häufig vor. Diese lassen sich aber auf flache Transaktionen nicht abbilden. Damit gelten für die Transaktionsmodelle folgende Anforderungen, siehe [Meck96].

  - **Langlebigkeit**:
    Im Anwendungsszenario werden komplexe Modelle entwickelt. Dieser Prozeß ist im allgemeinen sehr zeitaufwendig und kann mitunter Jahre dauern. Reine Sperrprotokollverfahren sind damit nicht anwendbar, da dies zu sehr langen Blockaden führt. Ein vollständiges Rücksetzen einer solchen Transaktionen ist nicht gewünscht, weil sehr weitreichende Änderungen verloren gehen.
  - **Kooperation**:
    An der Entwicklung des Modells sitzen mehrere Personen, die an der Teilmodellierung anderer Entwickler partizipieren möchten. Damit lesen diese Entwickler inkonsistente Zustände, was möglicherweise zu kaskadierenden Abbrüchen bei Standardtransaktionen führt.
  - **Konsistenz**:
    Konsistenzbedingungen flacher Transaktionen sind in vielen Fällen zu einschränkend. Aus diesem Grund werden erweiterte Konsistenzbegriffe eingeführt.
  - **Kontrollfluß**:
    In konventionellen Transaktionen ist der Kontrollfluß eine Folge von *begin of transaction*(BOT), *end of transaction*(EOT) und einer Menge von Schreib- und Leseoperationen. Leider sind damit Konzepte, wie Arbeitsabläufe, die immer wieder kehren bzw. dynamisch erzeugt werden, nur ungenügend abbildbar. Für diese Zwecke muß das Transaktionsmodell um Checkpoints, welche Sicherungs- bzw. auch Konsistenzpunkte bezeichnen, erweitert werden.

- **Erweiterung der Sperrmodelle**
  Sperrmodelle, wie das READ/WRITE-Modell aus 1.3.3, sind nur syntaktisch. Darum geht das Wissen aus der Anwendung verloren. Zu der Erweiterung der Lockmodi sollte außerdem Funktionalität bereitgestellt werden, die ein "upgrade" der Lockmodi oder auch eine frühe Freigabe der Sperre enthält. Für den Fall der Verwendung sehr komplexer Objekte sollte eine möglichst feinkörnige Granularität der Sperren unterstützt werden.

- **Rollenunterstützung**
  In klassischen CSCW-Anwendungen sind Rollen gewissen Personen zugeordnet, die damit bestimmte Rechte und Pflichten besitzen. Die Rollenvergabe kann sowohl statisch als auch dynamisch erfolgen. Für die Zuordnung von einer Rolle zu einer Person ist eine Authentifizierung notwendig. Z.B. darf ein Projektleiter, der sich einen Eindruck über den Projektstand bilden will, Sperren brechen ohne Notifikation des Sperreninhabers.

- **aktive Mechanismen**
  Die Modellierer arbeiten synchron sowie asynchron an der gemeinsamen Entwicklung eines Modells. Diese Arbeit muß koordiniert werden, damit nicht eventuell "das Rad zweimal erfunden" wird. Über Notifikationstechniken sind andere Gruppenmitglieder z.B. über Änderungen zu informieren. Notifikationsmechanismen sollten dabei frei konfigurierbar sein. Um solche Mechanismen abzubilden, kann das Trigger-Konzept von Datenbanksystemen verwendet werden. Hierbei werden bei bestimmten Situationen (Ereignisse und Bedingungen) Aktivitäten (Aktionen) eingeleitet.

- **Unterstützung von Versionen und komplexen Objekten**
  Ein sehr extremes Beispiel für komplexe Objekte ist die Entwicklung eines Autos. Das Auto besteht wiederum aus Motor, Karosserie usw. Objektorientierte Datenbanksysteme besitzen unter anderem die Möglichkeit, Versionen zu verwalten. Weil in der Modellierung Versionen eine sehr große Bedeutung haben, sollte der ODA+ auch entsprechende Mechanismen anbieten.

Diese Anforderungen ergeben sich direkt oder indirekt aus dem Anwendungsszenario, sollen abgebildet und entsprechend in eine CORBA-Umgebung integriert werden. Innerhalb des nächsten Kapitels wird beschrieben, wie in dieser Arbeit für diesen Zweck vorgegangen wird.

1.6 Vorgehensweise

Im folgenden Kapitel wird zunächst die Synchronisation von erweiterten Transaktionsmodellen erklärt. Anhand der Kriterien für lokale Transaktionen im kooperativen Umfeld wird entschieden, ob sie einsetzbar sind. Außerdem werden erweiterte Sperrmodelle eingeführt und auf ihre Anwendbarkeit untersucht.

Im Kapitel 3 werden zuerst bestehende Datenbanksysteme auf eine breite Unterstützung von erweiterten Sperr- und Transaktions-Features untersucht. Danach wird eine Entscheidung bezüglich der Wahl eines Datenbanksystems getroffen. Weiterhin soll eine Ar-

chitektur gefunden werden, in der die im Kapitel 1.5 angegebenen Kriterien einfließen. Gleichzeitig werden Möglichkeiten offen gelassen, den ODA+ zu erweitern. Dabei werden erweiterte Transaktionen und Sperrmodelle entsprechend Kapitel 2 als Grundbestandteil umgesetzt.

Im Kapitel 4 wird eine Umsetzung anhand von Versant vorgesstellt. Weiterhin wird eine Alternative entwickelt, wie sich die Konzepte aus Kapitel 3 noch realisieren lassen.

Kapitel 5 gibt eine Zusammenfassung und schließt mit einem Ausblick ab.

Es sei angemerkt, daß für das Verständnis dieser Arbeit Grundkenntnisse in Bezug auf Transaktionen, CORBA und die C++–Programmiersprache notwendig sind.

Kapitel 2

Theorie zur Synchronisation von Transaktionen

In dem folgenden Kapitel werden grundlegende Verfahren erläutert, wie die Transaktionsoperationen synchronisiert werden. Für die Synchronisation von Transaktionen sind prinzipiell zwei Komponenten erforderlich.

Zum einen zählt dazu der Transaktions-Manager. Er ist verantwortlich für die Verwaltung der Transaktionen und deren Zustände. Für jede Transaktion werden Warteschlangen geführt, welche die ausführbereiten Operationen beinhalten. Der Beginn einer neuen Transaktion ist mit BOT und das Ende mit EOT gekennzeichnet. Dabei gilt für das Ende einer Transaktion, daß entweder mit *commit*, für den Fall das aus Sicht des Transaktions-Managers keine Fehler aufgetreten sind oder mit *abort*, durch impliziten Nutzerabbruch oder falls eine Fehlersituation entdeckt wurde, beendet wird.

Zum anderen existiert der Scheduler. Dieser nimmt die Operationen, welche der Transaktions-Manager ausgibt, entgegen und bringt sie in eine Reihenfolge, die bestimmten Eigenschaften genügen. Dazu stehen ihm Mittel zur Verfügung, auf den Transaktions-Manager bzw. auf die Zustände der Transaktionen, siehe Abbildung 2.1, Einfluß zu nehmen.

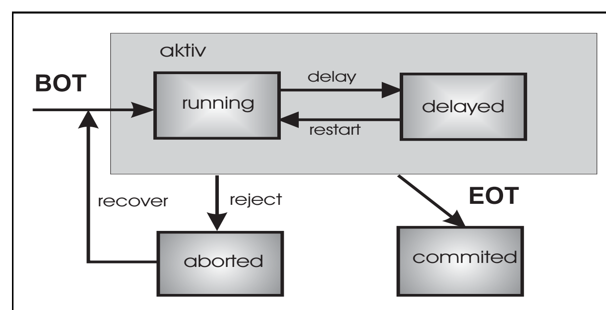

Abbildung 2.1: Transaktionszustände

Jede Operation kann entweder sofort weitergegeben (*execute*), zurückgewiesen (*reject*) oder verzögert (*delay*) werden. Das Zurückweisen einer Operation ist mit dem Abbruch der entsprechenden Transaktion verbunden. Verzögert wird eine Operation, wenn diese

momentan nicht ausführbar ist, aber durchaus zu einem späteren Zeitpunkt in den Outputschedule eingefügt werden kann. Die betreffende Transaktion wird in den "delayed"-Zustand überführt. Damit ist der Scheduler ein Filter, welcher nur die bestimmten Bedingungen entsprechenden Operationen durchläßt. Solche Bedingungen können z.B. die Serialisierbarkeit des Ausgabe-Schedules oder die Vermeidung von kaskadierenden Abbrüchen sein. Unter dem letzten Punkt versteht man, daß keine Transaktion Objekte von noch nicht beendeten Transaktionen liest. Eine weitere wichtige Bedingung, die ein Scheduler erfüllen muß, ist, daß sichergestellt wird, daß alle Transaktionen auch wirklich zu einem definierten Ende kommen (Deadlock- und Livelock-Freiheit für alle Eingabe-Schedules).

2.1 Klassifikation

Eine allgemeingültige Klassifizierung für die Synchronisation von Transaktionen zu finden, erweist sich als sehr schwierig. Deshalb seien hier nur verschiedene Sichtweisen angegeben. In [GrVo93] werden Scheduler danach charakterisiert, wie "kulant" diese die Operationen im Zusammenhang mit dem schon erzeugten Ausgabe-Schedule bzw. diesen selbst testen. Demnach ist er:

- **aggressiv**, falls er die Operationen überwiegend weitergibt. In diesem Fall sind Eigenschaften wie die Serialisierbarkeit für den Ausgabe-Schedule und die Deadlock-Freiheit zumeist nicht gewährleistet.
- **konservativ**, wenn er Operationen überwiegend verzögert oder sogar Transaktionen abbricht. Der Ausgabe-Schedule kann im extremen Fall zu einer seriellen Ausführung der Transaktionen degenerieren.

Damit sind die Klassifikationskriterien in diesem Fall die Eigenschaften, die der erzeugte Ausgabe-Schedule besitzt.

Eine anderere Einteilung nach [GrVo93] klassifiziert nach **sperrenden** und **nicht sperrenden** Schedulern. In dem ersten Fall wird davon ausgegangen, daß sich eine Transaktion zuerst das Recht (Sperre) für die Ausführung einer Operation mit der jeweiligen Semantik erwerben muß. Damit werden auch gleichzeitig alle anderen in Konflikt stehenden Operationen bzw. dafür benötigte Sperren bis zu der entsprechenden Sperrfreigabe blockiert. Die zweite Klasse kommt dagegen ohne die Verwaltung von Sperren aus. In diese Klasse fallen Verfahren, wie optimistische Concurrency-Control, Serialisierbarkeitsgraphen-Tester oder auch Zeitstempelverfahren. Alle Verfahren besitzen ihre Vor- und Nachteile. Aus diesem Grund sind auch hybride Techniken entwickelt worden.

Die letzte Klassifikation und hier verwendete Einteilung unterscheidet nach dem Zeitpunkt, wann eine Überprüfung des Schedules auf gewisse Eigenschaften des zu erzeugenden bzw. erzeugten Schedules stattfindet. In den **pessimistischen** Verfahren wird bei jeder Operation davon ausgegangen, daß Fehler auftreten können. Sobald eine Operation, bzw. schon bei der Sperroperation für diese, beim Scheduler eintrifft, wird der Ausgabe-Schedule auf z.B. Serialisierbarkeit überprüft. Dagegen wird in **optimistischen** Verfahren der Ausgabe-Schedule nach dem Ende der aktiven Phase einer Transaktion auf Serialisierbarkeit überprüft. Die entsprechenden Verfahren werden in den folgenden Kapiteln vorgestellt.

2.1.1 Optimistische Verfahren

Bei diesen Verfahren wird davon ausgegeangen, daß keine Konflikte auftreten bzw. nur selten vorkommen. Deshalb wird jede Operation, welche von einer Transaktion ausgeführt wird, direkt weitergeleitet. Erst vor dem Commit einer Transaktion wird überprüft, ob der so entstandene Ablaufplan (Schedule) serialisierbar ist. Ist dies nicht der Fall, wird die TA abgebrochen. Eine optimistische Transaktion besteht aus drei typischen Phasen, siehe Abbildung 2.2. Es wird ein *deferred-update* vorrausgesetzt. Das heißt, alle Änderungen einer Transaktion werden auf einem privatem Bereich ausgeführt.

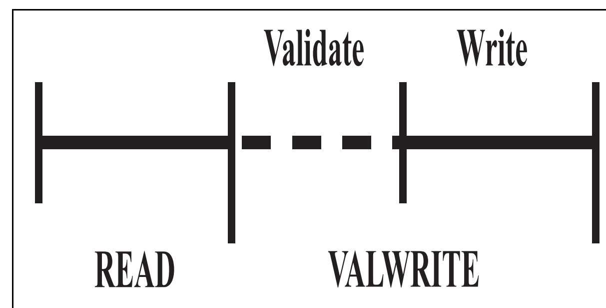

Abbildung 2.2: Transaktionsphasen in einer optimistischen TA [Weik88]

- Die **Read-Phase** ist die Phase, in der die Transaktion ausgeführt wird, alle Schreiboperationen werden nur auf den temporären Speicher der TA ausgeführt
- In der **Validation-Phase** wird überprüft, ob die Transaktion korrekt abgelaufen ist.
- In der **Write-Phase** werden die temporären Ergebnisse der TA zurückgeschrieben, wenn der zweite Punkt bejaht werden konnte. Es wird hier davon ausgegangen, daß die Write-Phase und die Validation-Phase nicht unterbrochen wird ("kritischer Bereich"), welche dann als Valwrite-Phase bezeichnet wird. Zumindest für diese Phase sind Sperren vorrauszusetzen. Ein Nachteil ist, daß die Valwrite-Phase eine sehr lange Dauer besitzt, wenn eine Transaktion sehr viele Änderungen macht.

Jetzt existieren nach [Hard84] zwei Möglichkeiten, wie die Validierung zu erfolgen hat. Zum einen existiert die Rückwärtsvalidation (*backward-oriented certification*-BOCC), siehe auch [KuRo81]. Hier wird gegenüber den Transaktionen t_j validiert, deren Valwrite-Phase innerhalb der Read-Phase der zu validierenden TA t_i lag. In diesem Fall wird überprüft, ob eines der Objekte, welche t_i gelesen hat, von einer t_j geschrieben wurde. Zum anderen gibt es die Vorwärtsvalidation (*forward-oriented certification*-FOCC), nachzulesen in z.B. [Schl81]. In diesem Verfahren wird gegenüber Transaktionen t_j validiert, welche ihre Valwrite-Phase noch nicht erreicht haben. Hier wird auf Disjunktheit der von t_i geschriebenen und der von t_j gelesenen Objekte überprüft. Sind beide Objektmengen disjunkt, wird erfolgreich abgeschlossen. Algorithmen zu beiden Verfahren stehen in [GrVo93].

2.1.2 Pessimistische Verfahren

Die Klasse der pessimistischen Verfahren kann einerseits in die nicht- und die sperrenden Verfahren untergliedert werden. Zu den nichtsperrenden Verfahren zählen die **Zeitstempelverfahren** und die **Serialisierbarkeitsgraphen-Tester**.

Beim ersteren Verfahren wird jeder Transaktion ein Zeitstempel zugewiesen. Deshalb wird es auch *Timestamp-Ordering*-Protokoll (TO-Protokoll) genannt. Jeder Operation wird der Zeitstempel ihrer Transaktion zugeordnet. Ein sehr vereinfachtes Zeitstempel-Verfahren, bezeichnet als *Basic*-TO (BTO), funktioniert, indem alle Operationen abgewiesen werden, die "zu spät" sind. Eine Operation ist "zu spät", wenn schon eine Operation auf dem selben Objekt ausgeführt wurde, die einen größeren Zeitstempel besitzt. Dies wird erreicht, indem auch jedes Objekt ein Zeitstempelattribut mit dem Zeitstempel der letzten Transaktionsoperation erhält. Die Transaktion einer "zu späten" Operation wird zurückgesetzt und zu einem späteren Zeitpunkt mit einem größeren Zeitstempel neu gestartet. Die andere Möglichkeit ist, daß Operationen mit einem hohen Zeitstempel verzögert werden.

Dagegen arbeitet das zweite Verfahren mittels der dynamischen Verwaltung eines Graphen (kurz SGT- für Serialisierbarkeitsgraphentester) aller beteiligten Transaktionen. Für jede ankommende Transaktionsoperation werden die Konflikte mit schon ausgeführten Operationen anderer Transaktionen ermittelt und in den Graphen als neue Kante eingefügt. Die Transaktionen bilden dabei die Knotenmenge. Durch Überprüfung auf Azyklizität des Graphen wird nach jeder Kanteneinfügung ermittelt, ob ein serialisierbarer Outputschedule entstanden ist. Im negativen Fall wird die Operation zurückgewiesen, die Transaktion abgebrochen und damit aus der Knotenmenge des Graphen und alle inzidenten Kanten entfernt. Ein Nachteil dieses Verfahrens ist exponentieller Speicherplatzbedarf in Abhängigkeit von der Anzahl der Transaktionen. In dieser Arbeit werden diese Verfahren keine Rolle spielen, da sie in einem kommerziellen DBMS in Reinform selten verwendet werden.

In Tabelle 2.1 werden die bereits angeführten Verfahren und die Klassifikationskriterien gegenübergestellt. Ein Plus (Minus)-Zeichen bedeutet, daß das Protokoll in diese Kategorie (nicht) eingeordnet ist.

Die Sperrverfahren sind die praktikabelsten Synchronisationsverfahren. Deshalb werden sie in dieser Arbeit eingehender beleuchtet. Zunächst werden die Sperrmodelle um Mechanismen erweitert, welche besonders in kooperativer Umgebung wichtig sind. Darauf aufbauend werden danach die verschiedenen Synchronisationsmechanismen für verschiedenste Transaktionen vorgestellt.

2.2 Sperrmodelle

Sperren werden nicht nur im Zusammenhang von Transaktionen verwendet. Das heißt, es existieren Locks, welche nicht innerhalb einer Transaktion (*transaction related lock* kurz TR-lock) gesetzt wurden und nur bestimmten Objekten (*object related lock* kurz OR-lock) oder übergeordneten Besitzern (*subjekt related lock* kurz SR-lock) zugeordnet werden. Die **objektbezogenen** werden verwendet, um Operationsmöglichkeiten auf dem Objekt einzuschränken, z.B. Nur-Lese-Objekte. In einigen noch näher zu erläuternden Verfahren ist es notwendig, transaktionsübergreifende temporäre Sperren, die bestimmten

	opt.	pess.	sperr.	nichtsperr.	aggr.	kons.
2PL	-	+	+	-	-	+
S2PL	-	+	+	-	-	+
C2PL	-	+	+	-	-	+
N2PL	-	+	+	-	-	+
FOCC	+	-	-	+	+/-	+/-
BOCC	+	-	-	+	+/-	+/-
MGL	-	+	+	-		
BTO	-	+	-	+	-	+
MVTO	-	+	-	+	-	+

Tabelle 2.1: Protokolleigenschaften

Legende:

2PL	Zwei-Phasen-Protokoll
S2PL	striktes Zwei-Pasen-Protolkoll
C2PL	konservatives Zwei-Phasen-Protokoll
N2PL	Nicht-Zwei-Phasen-Protokoll
FOCC	vorwärtsgerichtete Validierung
BOCC	rückwärtsgerichte Validierung
MGL	Multi-Granularitäts-Sperrverfahren
BTO	Basis-Zeitstempelverfahren
MVTO	Mehrversionen-Zeitstempelverfahren

Subjekten gehören, zu besitzen. Ein Subjekt kann alles sein, was durch den Transaktions-Manager identifiziert werden kann, z.B. eine Anwendung, ein Prozeß, eine Nutzergruppe etc. Sie werden benötigt, um Objekte von einer Transaktion in eine andere zu verschieben. Damit während dieser Phase keine andere Transaktion auf das selbe Objekt zugreifen kann, werden die **subjektbezogenen** Sperren verwendet. Sperren können mittels eines Sperr-Managers verwaltet werden, indem er allen Objekte eine Menge von Paaren (Sperrmodus, Sperrinhaber) zuordnet und entsprechend aktualisiert.

Zuerst wird das Verständnis von Sperren erweitert. Anschließend werden Sperrmodelle eingeführt, die eine größere Nebenläufigkeit zulassen.

2.2.1 Die Innen- und Außenwirkung von Sperren

Ausgegangen wird von dem Grundprinzip, daß eine Transaktion zu Beginn keine Rechte hat, um auf Datenbankdaten zu arbeiten. Um diese Erlaubnis zu erhalten, werden Sperren vorrausgesetzt. Mit einer Sperre erhält die Transaktion die Rechte, bestimmte Operationen auszuführen. Aber nicht allein auf die besitzinhabende Transaktion hat eine Sperre Einfluß. Es wird unterschieden zwischen dem Inhaber (*internal effect*) und den Konkurrenten (*external effect*) einer Sperre, siehe [ScUn92]. Die externe Auswirkung reglementiert den Zugriff auf das Objekt von anderen Nutzern als dem Inhaber. Eine Schreibsperre hat damit die Innenwirkung, daß der Besitzer lesen, verändern und löschen darf. Die Außenwirkung bewirkt, daß kein Konkurrent das Objekt mit irgendeiner Sper-

re versehen darf. Eine Lesesperre besitzt die gleiche Innen- sowie Außenwirkung. Der Inhaber und die Konkurrenten dürfen auf dem Objekt nur lesen. Aus dieser Sicht läßt sich eine Sperre über das Paar (*internal effect, external effect*) definieren. Dabei sind nur Paare zugelassen, die auch einen Sinn ergeben. Im Standardsperrmodell ist z.B. nur das Paar (s,s) sinnvoll. Daraus ergibt sich folgende Kompatibilitätsdefinition. Das heißt zwei Sperren sind kompatibel, wenn

- die Außenwirkung des Besitzers der Sperre erlaubt die Innenwirkung des Konkurrenten
- die Außenwirkung des Konkurrenten erlaubt die Innenwirkung des Besitzers

Nachdem das Verständnis von Sperren beleuchtet wurde, folgt im nächsten Abschnitt eine Erweiterung von Sperren um mehr Semantik.

2.2.2 Flexibilisierung von Sperren

Im Kapitel 1.3.3 wurden Sperren eingeführt, die in konventionellen Transaktionen verwendet werden. Um mehr Nebenläufigkeit zu erlauben, können diese Lockmodi erweitert werden. Diese Lockmodi erlauben einen größeren Einbezug der Semantik der Operationen.

Für das nun folgende Lockmodell setzen wir voraus, daß die Versionierung[1] von Objekten erlaubt ist. Die uns bekannten Lockmodi **shared (S)** und **exclusive (X)** werden um

- **update lock (U)**, erlaubt das Lesen und Modifizieren aber nicht das Löschen von Objekten
- **derivation lock (D)**, erlaubt das Lesen von Objekten und das Ableiten von neuen Objekten
- **browse lock (B)**, erlaubt das "dirty read" von Objekten (Lesen ohne Sperrenberücksichtigung)

erweitert, siehe [ScUn89]. Setzen wir dieses Lockmodell in Beziehung zu der in Kapitel 2.2.1 angegeben Sichtweise, ergibt sich die Tabelle 2.2. Die Zeichen *+(-)* indizieren,

		Außen				
		B	S	D	U	X
I	B	+	+	+	+	+
n	S	+	+	*	*	-
n	D	+	+	*	!	-
e	U	+	*	!	!	-
n	X	+	-	-	-	-

Tabelle 2.2: Innen-Außenwirkungs-Matrix

welche korrespondierende Innen/Außenwirkungspaare immer (nie) kombiniert werden dürfen. Das Paar, gekennzeichnet durch ***, kann erlaubt werden, ohne daß es zu "lost update"

[1] Dazu gehört die Verwaltung von unterschiedlichen Zustände des selben Objektes

führt. Ein *!* darf nur in speziellen Anwendungen unterstützt werden, weil es zu schwerwiegenden Konsistenzproblemen führen kann. Die Konsistenzsicherung der beiden letzten beschriebenen Punkte muß dann in die Anwendung verlagert werden.

2.2.2.1 Hierarchische Sperren

Die Sperrmodelle, die bis jetzt betrachtet wurden, abstrahierten davon, daß ein Objekt gesperrt wurde. Erweitern wir unsere Sicht, so besitzt ein Objekt eine Struktur, ein Verhalten, einen Zustand und eine Identität. Einen Einblick in die Objektorientierung allgemein und speziell in Datenbanksystemen wird in [Heue97] gegeben. In CAD-Anwendungen beinhalten Objekte vielfach wieder andere Objekte bzw. deren Referenzen, z.B. ein Auto besitzt eine Karosserie, einen Motor und eine Innenausstattung. Aus diesem Grund werden Sperren auf verschiedenen Stufen (Granularität) unterschieden. Zum einen soll eine Sperre möglichst wenig restriktiv sein, so daß noch parallele Arbeit möglich ist. Aber auch der negative Fall, daß zu viele Sperren gesetzt sind, auch als Sperreskalation bezeichnet, sollte nicht eintreten, da jede Sperre auch verwaltet werden muß. Aus diesem Grund müssen Sperren zusammengefaßt werden. Das heißt, viele Sperren auf der selben Granularitätsstufe werden gegen eine auf höherer Ebene ausgetauscht.

In [DHKS89] wird das Problem, des Setzens von Sperren mit optimaler Granularität durch **geeignete Vorwegnahme von Sperreskalation** automatisch realisiert. Die sperrbaren Einheiten sind hier:

- **basic lockable units (BLU)** stellen die kleinsten Einheiten, wie Basisdatentypen integer, real usw. dar.
- **homogenous lockable units (HoLU)** bestehen aus Mengen und Listen (Daten selben Typs).
- **heterogenous lockable units (HeLU)** bestehen aus Teilobjekten verschiedenen Typs.

Ausgegangen wird von dem Fall, daß nur BLU's gesperrt werden. Da dies einen zu hohen Aufwand bei der Verwaltung darstellt, erfolgt folgende Zusammenfassung. Wird eine bestimmte Anzahl i von Komponenten der HeLU's bzw. der HoLU's gelesen oder geändert, wird eine Sperre auf der entsprechenden HeLU oder HoLU vorweggenommen. Die Zahl i kann dabei eine bestimmte Prozentzahl von der Gesamtanzahl sein und ist sehr wesentlich für die Leistungsfähigkeit eines solchen Systems.

Ein Problem bei Sperren auf verschiedenen Hierarchiestufen ist die Konflikterkennung zwischen diesen. Aus diesem Grund werden zwei Sperrenarten unterschieden.

- **normale Sperren** sperren das Objekt der betreffenden Granularitätsstufe explizit und implizit alle darin enthaltenen Objekte niedriger Granularitätsstufen. Hier sind die uns bekannten Schreib- und Lesesperren einzuordnen.
- Mit der **Intention-Sperre** erhält die Transaktion das Recht, Objekte der nächst niederen Stufe zu sperren. Eine gesetzte Intention-Sperre sagt aus, daß Objekte niederen Abstraktionsniveaus gesperrt sein können. Damit sind die beiden Sperren wie folgt erklärt.

- *intention-shared* (IS) erlaubt der Transaktion, Teilobjekte zum Lesen zu sperren.
- *intention-exclusive* (IX) erlaubt auf Teilobjekte, Schreibsperren zu setzen.

In der Tabelle 2.3 ist die Verträglichkeitsmatrix für Intentionsperren angegeben [GLPT76].

	S	X	IS	IX
S	+	-	+	-
X	-	-	-	-
IS	+	-	+	+
IX	-	-	+	+

Tabelle 2.3: Kompatibilitätsmatrix für Intention-Sperren

Um die Konflikterkennung für den Scheduler zu vereinfachen, darf eine Sperre, sowohl normal als auch *intention*, nur gesetzt werden, falls die Transaktion eine Intention-Sperre auf dem nächst höheren Abstraktionsniveau besitzt. Ausgenommen davon ist die Wurzel der Hierarchie. Das bedeutet, daß die Sperranforderung immer von der Wurzel zu den Blättern und die Freigabe von den Blättern zur Wurzel zu erfolgen hat. Diese Vorgehensweise wird auch **Multiple Granularity Locking** (MGL) genannt.

Leider geht dieses Verfahren davon aus, daß gemeinschaftlich genutze Objekte nicht vorkommen. Für diesen Fall müssen, wenn eine Sperre auf ein Objekt gesetzt wird, auch alle Vorgänger mittels einer Intention-Sperre davon in Kenntnis gesetzt werden.

2.2.2.2 Anderweitige Sperrkonzepte

Sperren können semantisch angereichert werden. Das Sperrverhalten kann z.B. verändert werden, indem es auf unterschiedliche Situationen dynamisch reagiert.

- **semantische Sperren:**
  In Transaktionen werden die eigentlich benutzten Operationen auf die syntaktischen Operationen (read und write) abgebildet. Leider geht so die eigentliche Bedeutung und ihre Beziehungen zueinander verloren. Nehmen wir als Beispiel zwei Increment-Operationen zweier Transaktionen. Innerhalb einer Transaktion wird diese Operation in eine Lese- mit darauffolgender Schreibaktion des Increments des gelesenen Wertes abgebildet. Das heißt, daß auf Scheduler-Ebene ein Konflikt auftritt. Dagegen sind zwei Increment-Operationen kommutativ und damit konfliktfrei. Ein Verfahren ist die Erweiterung der Operationen und entsprechender Sperren um z.B. Increment- bzw. Decrement-Operationen. Damit führt dieser Weg dazu, die Kommutativität von Operationen und die Operationssemantik in das Sperrmodell miteinzubeziehen.
- **Unterscheidung von Konkurrenten:**
  Mit der Innen- und Außenwirkung wird zwischen dem Inhaber und dem Konkurrenten einer Sperre differenziert. Unterschiedliche Konkurrenten werden in diesem

Fall gleich behandelt. Ein anderer Ansatz geht davon aus, daß unterschiedliche Konkurrenten auch unterschiedliche Rechte besitzen. Z.B. darf ein Projektleiter trotz Sperre sich den derzeit aktuellen Zustand anschauen. Für solche Zwecke werden in [BaSc93] Mechanismen zur Verfügung gestellt, um die allgemeine und spezielle Außenwirkung einer Sperre zu definieren.

- **Notifikationssperren:**
  In diesem Fall werden Sperren um aktive Mechanismen erweitert. Ähnlich zu Trigger-Mechanismen lassen sich bestimmte Sperrereignisse spezifizieren bei deren Auftreten bestimmte Aktionen ausgeführt werden. In [HoZd87] wird eine Transaktion A informiert, wenn eine Transaktion B auf die von A gesetzte Notifikationssperre zugreifen möchte.

2.3 Synchronisation von Transaktionen

Die folgenden Transaktionsmodelle und ihre Synchronisationsarten werden unter dem Gesichtspunkt der Verwendbarkeit in Entwicklungsumgebungen betrachtet. Deshalb müssen sich daraus ergebende Vorraussetzungen in Bezug auf die Kooperation erfüllt werden. Dazu werden in [Meck96] folgende Kriterien angeführt:

- **unsichere Freigabe:** Die Objekte können vor dem Ende der Transaktion freigegeben werden. Sollte die betreffende Transaktion abbrechen, wird jedes vorzeitig freigegebene Objekte zurückgesetzt. Zu diesem Zweck behält die Transaktion das Rücksetzbarkeitsrecht für das Objekt.

- **sichere Freigabe:** Objekte werden frühzeitig freigegeben und werden auch bei Abbruch der TA nicht mehr zurückgesetzt.

- **flexible Sperrmodi:** Z.B. die in Kapitel 2.2.2 angeführten Sperren sollten in das Sperrprotokoll integrierbar sein.

- **vorzeitiges Commit von Komponententransaktionen:** In geschachtelten Transaktionen werden Komponententransaktionen vorzeitig beendet und geben damit ihre Objekte frei.

- **Kooperationsmechanismen:** Darunter sind Mechanismen zu verstehen, wie das Verleihen und die Weitergabe von Objekten bzw. deren Sperren. Unter Verleihen bzw. Weitergabe wird verstanden, daß eine Transaktion, welche eine Sperre auf einem Objekt besitzt, diese einer anderen Transaktion zur Verfügung stellen kann. Diese gehört dann dieser oder muß nach der Beendigung dem ursprünglichen Besitzer wieder zurückgegeben werden.

- **Spezifikation kooperativer Abläufe:** Ein Beispiel dafür ist die Festlegbarkeit von Zeitpunkten innerhalb einer Transaktion, wann diese von anderen parallel ablaufenden unterbrochen werden darf. Für diesen Zweck sind Erweiterungen des Transaktionsmodells um Operationen wie *Checkpoint* oder *Savepoint* notwendig.

- **Kooperationsmengen:** Bestimmte Transaktionen können zusammengefaßt werden. Zwischen diesen existiert ein höherer Freiheitsgrad bezüglich der Isolationseigenschaft, weil nicht die restriktivsten Maßnahmen der Synchronisation greifen.

Die angeführten Kriterien werden in den nachfolgenden Kapiteln zur Einordnung und näheren Charakterisierung für die Synchronisationsarten verwendet.

2.3.1 Basisprotokolle

In diesem Kapitel werden zunächst nur die Protokolle vorgestellt, welche in flachen Transaktionen vorkommen. Diese Mechanismen lassen sich auch bei der Synchronisation in weitergehenden Transaktionsmodellen wiederfinden. Zu den gebräuchlichsten Protokollen gehören die in der Einführung schon kurz erwähnten 2Phasen-Protokolle (2PL-*two-phase locking*). Eine 2PL–Transaktion ist gekennzeichnet durch eine Wachstumsphase, in der sie alle benötigten Sperren anfordert und durch eine Schrumpfungsphase, in der alle Sperren wieder freigeben werden, siehe Abbildung 2.3. Durch die Restriktion, daß die Sperranforderung nach einer Sperrfreigabe verboten ist, wird die Konfliktserialisierbarkeit des Ausgabe-Schedules sichergestellt.

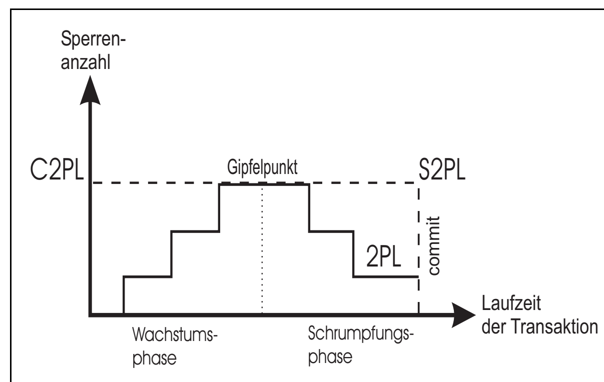

Abbildung 2.3: Vergleich der Basisprotokolle

Dieses Protokoll vermeidet nicht den kaskadierenden Abbruch von Transaktionen, denn während der Schrumpfungsphase können andere Transaktionen wieder Sperren erwerben. Aus diesem Grunde wird das 2Phasen-Protokoll so erweitert (S2PL-*strict two-phase locking*), daß alle Sperren[2] bis zum Ende der Transaktion gehalten werden. Deswegen treten auch hier nicht erwünschte Nebeneffekte (*Livelock* und *Deadlock*) auf, die gerade bei der Sperrenkonvertierung auftreten. Halten zwei Transaktionen auf dem selben Objekt eine Lesesperre und wollen diese zu einer Schreibsperre umwandeln, warten beide aufeinander. Auf Protokollebene (C2PL-*conservative two-phase locking*) wird die Behebung dieses Problems erreicht, indem alle Sperren, welche eine Transaktion benötigt, zu Beginn der Transaktion gesetzt werden. Wird diese Variante nicht gewählt, kann der gleiche Effekt erzielt werden, wenn man solche Deadlock-Situationen zwar zuläßt, sie aber mit geeigneten Mitteln behebt. Man ermittelt die Transaktionen, welche daran beteiligt sind und

[2]ausreichend alle Schreibsperren

bricht eine von ihnen ab. Für die Ermittlung, welche abgebrochen wird, existieren auch wiederum verschiedene Verfahren. Diese müssen realisieren, daß nicht immer die selbe TA abgerochen wird (*Livelock*) und so wenig wie möglich Rücksetzkosten entstehen.

Für die Deadlock-Erkennung ist eine Lösung [Meye96] die Zyklenerkennung innerhalb eines gerichteten Graphen, in dem die Knotenmenge die Transaktionen und die Kanten durch Konfliktpaare der Operationen der entsprechenden Transaktionen gebildet werden. Eine zweite besteht mittels Timeout-Verfahren. Dabei wird generell die Timeout-Zeit, innerhalb derer eine Transaktion keine Sperre mehr bewilligt bekommen hat, abgewartet.

Ein anderes Protokoll verzichtet auf die Zweiphasigkeit (N2PL- *non two-phase locking*) von Transaktionen. Die Sperrenanforderung bzw. -freigabe kann beliebig innerhalb der Transaktion erfolgen. Damit werden die Sperren nur für die jeweilige Operation gesetzt und nach deren Ausführung wieder freigeben. Der Vorteil ist ein sehr hoher Kooperationsgrad zwischen den Transaktionen. Leider ist das Konsistenzkriterium Serialisierbarkeit nicht mehr gegeben und es kann zu schon beschriebenen Problemen wie "lost update" kommen. In diesem Fall ist der Entwickler selbst für die Konsistenz der Daten verantwortlich.

In Bezug zu den aufgestellten Kriterien ergeben sich folgende Aussagen.

- **sichere, unsichere Freigabe und flexible Sperrmodi:**
  Die Anwendbarkeit der beiden Protokolle S2PL und C2PL in Nichtstandardanwendungen ist sehr stark begrenzt. Zum einen ist die Vorhersage, welche Objekte man schreiben und nur lesen möchte, in Entwurfsumgebungen nicht realisierbar. Und zum anderen werden Objekte unnötig lange mit Sperren belegt, so daß die beiden ersten Punkte nicht erfüllt werden.

  Sowohl das 2PL- als auch das N2PL-Protokoll sind für die Mechanismen sichere und unsichere Freigabe geeignet. Für das 2PL-Protokoll gilt aber die Restriktion, daß nach einer Sperrfreigabe keine Anforderung mehr auftreten darf. Im Fall einer unsicheren Freigabe besteht die Möglichkeit des kaskadierenden Abbruchs von Transaktionen. Darüberhinaus lassen sich alle Protokolle so erweitern, daß sie flexible Sperrmodi unterstützen. Eine Erweiterung um Transaktionsoperationen für die Unterstützung von Checkpoints und Savepoints ist möglich.

- Weil das basierende flache Transaktionsmodell die weiteren Punkte nicht unterstützt, können diese auch nicht realisiert werden.

2.3.2 Multiversionsverfahren

Bis jetzt wurde davon ausgegangen, daß jede Schreiboperation das alte Objekt überschreibt. Aufgrund von Recovery-Techniken, welche mit zwei Zuständen von Objekten (inkonsistente und konsistente) arbeiten, wird schon für den Nutzer transparent mit Versionen[3] gearbeitet. Darauf basierend, daß durch eine Schreiboperation eine neue Version eines Objektes erzeugt wird, muß demnach für eine Leseoperation der jeweilige konsistente Objektzustand herausgesucht werden. Dies geschieht über Zeitstempel, welche eine Transaktion und eine Objektversion erhalten. Daraus ableitend, wird ein abgeändertes

[3] unterschiedliche Zustände des selben Objektes

Korrektheitskriterium, welches Versionen berücksichtigt, für Schedules verwendet. Äquivalent zum Mono-Versionsfall existieren die Korrektheitskriterien der View- bzw. Konflikt-serialisierbarkeit, beinhalten aber nicht mehr die selbe Klasse von zulässigen Schedules. Der View-Serialisierbarkeitsbegriff ist aber auch hier aus praktischer Sicht, durch exponentielle Tests bedingt, unbrauchbar.

Ein Multiversionsschedule ist **multiversions-konflikt-serialisierbar**, falls es einen seriellen Mono-Versions-Schedule gibt, in dem alle Konfliktpaare von Operationen im letzteren Schedule in der gleichen Reihenfolge ausgeführt werden wie im Multi-Versions-Schedule. Eine etwas ausführlichere Beschreibung ist in [GrVo93] bzw. in den Originalarbeiten [BeGo83, KaPa84] zu finden.

Ein Protokoll das auf der Basis dieses Kriteriums arbeitet ist das **Multi-Version-TimestampOrdering-Verfahren** (MVTO). Der MVTO-Scheduler arbeitet folgendermaßen:

- Eine Leseaktion der Transaktion i auf dem Objekt x (kurz r_i(x)) wird in eine Leseaktion auf der Version von x (kurz $r_i(x_k)$ transformiert, welche die größte Zeitmarke kleiner oder gleich der Zeitmarke der Transaktion i (kurz ts(t_i)) besitzt.
- Der Schritt w_i(x) wird folgendermaßen bearbeitet. Wenn ein Schritt $r_j(x_k)$ mit ts(t_k)<ts(t_i)<ts(t_j) bereits verarbeitet ist, wird w_i(x) abgelehnt. Sonst wird er in $w_i(x_i)$ transformiert und weitergeleitet. Damit auch gleichzeitig eine neue Version angelegt.

Für die Rücksetzbarkeit ist das Commit einer Transaktion solange zu verzögern, bis die Transaktionen, von denen eine Version gelesen wurde, noch nicht beendet sind. Der Vorteil dieses Verfahrens ist, daß nur lesende Transaktionen immer durchlaufen und nie abgebrochen oder blockiert werden. Zur Synchronisation von Multiversions-Schedules existieren auch Sperrverfahren [BHR80].

- **unsichere Freigabe:** Für nur lesende Transaktionen ist eine Freigabe von Objekten implizit im Verfahren enthalten, da für jede TA immer der aktuellste konsistente Wert bereitgestellt wird. Die Transaktion behält das Rücksetzbarkeitsrecht und realisiert dadurch automatisch eine unsicherer Freigabe.
- **sichere Freigabe:** Das Verfahren kann so modifiziert werden, daß für solche Objekte die Transaktion das Rücksetzbarkeitsrecht nicht behält.
- **flexible Sperrmodi:** Es ist schwierig, erweiterte Sperrmodi zu integrieren, da für die Sperrprotokollarten spezielle Sperren eingeführt werden, so daß primär nur R/W-Schedules synchronisiert werden.
- Die restlichen Punkte brauchen auch hier aufgrund des flachen Transaktionsmodell nicht erläutert werden.

2.3.3 Geschachtelte Transaktionen

Das geschachtelte Transaktionskonzept basiert auf der Tatsache, daß die meisten Anwendungsobjekte eine Struktur besitzen und wiederum mit einer Anzahl von struktu-

rierten Objekten in Beziehung stehen. Um solche Aspekte auch innerhalb eines Transaktionssystems modellieren zu können, wurden die flachen Transaktionen um das Prinzip der Schachtelung erweitert. Damit kann der Abstraktion in Softwaresystemen besser Rechnung getragen werden. Demnach ist eine Transaktion entweder ein Baum von Sub-Transaktionen oder eine flache Blatttransaktion. Es wird in dem Prinzip der **geschlossenen** und der **offen geschachtelten** Transaktionen unterschieden. Bei den ersteren ist das Commit einer Sub-Transaktion[4] immer an das Commit aller Vorgängertransaktionen[5] bis hin zur Wurzeltransaktion[6] gebunden.

Das offenen Schachtelungsprinzip basiert darauf, daß Sub-Transaktionen auch ohne Commit der Vorgängertransaktionen beendet werden können. Dies heißt, sie muß nicht generell abgebrochen werden, wenn eine der Vorgängertransaktionen abgebrochen wurde. Für solche Situationen können inverse Transaktionen auch **Kompensationstransaktionen** aufgerufen werden, welche einen aus semantischer Sicht zum Anfangszustand äquivalenten Zustand erzeugen. Ein anderer Vorteil geschachtelter Transaktionen ist die Möglichkeit, **Kontingenttransaktionen** aufzurufen, falls eine Transaktion abgebrochen wurde.

Für die Synchronisation von geschlossen geschachtelten Transaktionen werden die bekannten Basisprotokolle mit einigen Erweiterungen verwendet. Eine Sub-Transaktion hat während ihrer Laufzeit eine Menge von Sperren, unter anderem auch **zurückgehaltene** Sperren, erworben. Bei der Verwendung des S2PL-Protokoll werden diese am Ende (commit) der Sub-Transaktion der Eltertransaktion vererbt und dort als zurückgehaltene Sperren geführt. Sollte diese Vorkehrung nicht getroffen werden, führt dies möglicherweise zum inkonsistenten Lesen, weil die Sub-Transaktion an das Commit aller Vorgängertransaktionen gebunden ist und es deswegen noch zum Rücksetzen dieser kommen kann. Im Falle eines Abbruchs können alle gehaltenen Sperren verworfen werden. Die Sperranforderung wird abgelehnt, wenn eine Transaktion existiert, welche eine inkompatible Sperre besitzt und nicht direkter Vorgänger der Sperranforderungstransaktion ist. Für die Serialisierbarkeit ist die Striktheit ein notwendiges Kriterium. Denn während der Schrumpfungsphase der Eltertransaktion kann bei reinem 2PL-Protokoll innerhalb einer Kindtransaktion noch eine Sperranforderung geschehen. Damit ist durch die Aufwärtsvererbung auch die Zweiphasigkeit der Eltertransaktion nicht mehr gegeben.

Die offen geschachtelten Transaktionen werden in der Synchronisation unterschiedlich behandelt. Die gesetzten Sperren einer Sub-Transaktion müssen nicht am Ende an die Eltertransaktion als zurückgehaltene Sperre weitervererbt werden.

- **unsichere und sichere Freigabe:** Durch die Striktheit der Sub-Transaktionen im geschlossenen Fall ist die frühzeitige Freigabe nicht möglich. Jede Sperre wird bis zum Ende der Transaktion gehalten und dann erst an den Vater weitervererbt. Dagegen ist im offenen Fall sowohl unsichere als auch sichere Freigabe realisierbar.

- **flexible Sperrmodi:** Ableitend von der Aussage über S2PL können erweiterte Lockmodi verwendet werden.

- **vorzeitiges Commit von Komponententransaktionen:** Durch das Schachtelungskonzept bedingt wird dieser Punkt erfüllt.

[4]bei einer Hierarchiestufe gleich Kindtransaktion
[5]eine Stufe bedeutet Elter- oder Vatertransaktion
[6]besitzt selbst keine Vorgängertransaktion

- **Kooperationsmechanismen:** Weitergabe von Sperren kann innerhalb eines Transaktionsbaumes durch schrittweise Weitergabe der Sperre realisiert werden.
- **Spezifikation kooperativer Abläufe:** Kooperative Abläufe sind nicht spezifizierbar.
- **Kooperationsmengen:** In diesem Fall nicht möglich, weil alle Transaktionen im Baum gleich behandelt werden.

Mehrschichtentransaktionen

Eine spezielle Variante der offen geschachtelten Transaktionen sind die **Mehrschichtentransaktionen**. In diesem Fall ist an den Transaktionsbaum die Bedingung geknüpft, daß alle Blatttransaktionen die gleiche Entfernung zur Wurzeltransaktion besitzen. Damit ist eine Mehrschichtentransaktion ein Baum mit n Ebenen L_0,...,L_{n-1} und der Höhe n. Jeder Knoten des Baumes ist eine Transaktion bzw. Aktion, die durch die Kindtransaktionen realisiert wird. Zum Beispiel kann ein Update eines Wertes als Transaktion, welche zuerst den Wert liest und danach schreibt, aufgefaßt werden. Damit implementieren die Blatttransaktionen die eigentlichen Operationen und bilden die Schicht L_0.

Der Vorteil der Beschränkung auf n Schichten ist, daß für diese Klasse wieder Serialisierbarkeitsbegriffe existieren, welche von den Schedulern benutzt werden. Ein ausführliche Beschreibung der Theorie mit entsprechenden Beispielen kann in [Weik88] nachgelesen werden.

Definition 1:
Ein Mehrschichten-Schedule besteht aus einer Menge von Mehrschichtentransaktionen M_1, ...,M_k *und einer partiellen Ordnung* $<_i$ *auf den Aktionen der Ebene* L_0. △

Definition 2:
Die Menge aller Aktionen auf Ebene L_i*, deren Vorgänger a ist, bezeichne man mit* $act_i(a)$*. Die* L_i*-Aktionsordnung ist eine partielle Ordnung* $<_i$ *auf den Aktionen der Ebene* L_i *und definiert durch:* $a <_i a :\Leftrightarrow \forall b \in act_0(a), b' \in act_0(a') : b <_0 b'$*. Die Relation* CON_i *beinhaltet alle Konfliktaktionspaare der Ebene* L_i. △

Definition 3:
Die Serialisierungsordnung $\lesssim_i$ *ist folgendermaßen definiert:*
1. $\lesssim_0 = <_0 \cap CON_0$
2. $für\, 0 < i \leq n\, gilt : a \lesssim_i b :\Leftrightarrow \exists a' \in act_{i-1}(a)\, \exists b' \in act_{i-1}(b) :$
   $CON_{i-1}(a', b') \wedge a' \lesssim_{i-1} b'$. △

Auf der Basis eines Graphen läßt sich jetzt die Mehrschichten-Serialisierbarkeit definieren.

Definition 4:
Ein Mehrschichtenschedule ist ***mehrschichten-serialisierbar****, wenn die Relation* $\lesssim_i \cup (<_i \cap CON_i)$ *auf allen Ebenen azyklisch ist.* △

Nach [BBG87] kann das Mehrschichten-Serialisierbarkeits-Problem in das schon bekannte Einschichten-Serialisierbarkeits-Problem zerlegt werden. Die Zerlegung erfolgt dabei so, daß die Aktionen einer Schicht mittels uns schon bekannter Scheduler-Techniken synchronisiert werden. Die Synchronisation beginnt mit den Transaktionen des obersten

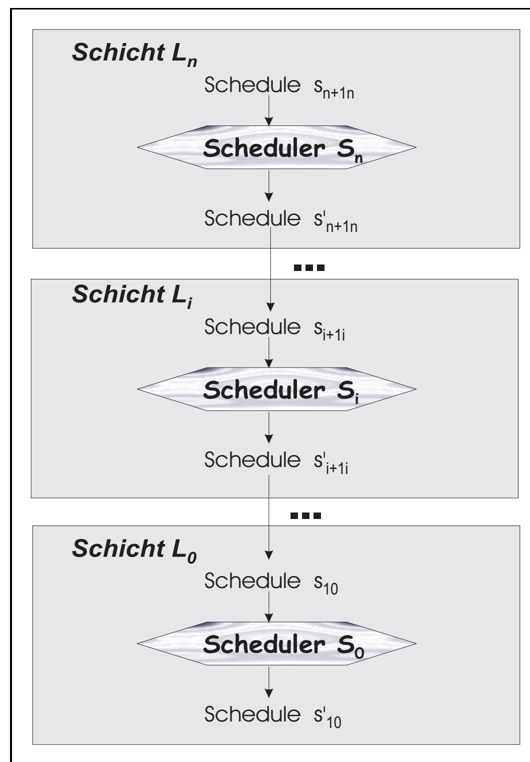

Abbildung 2.4: Mehrschichten-Schedule-Strategie

Abstraktionsniveaus, siehe Abbildung 2.4. Der Scheduler für eine Schicht erzeugt immer den Input-Schedule für den nächst niederen Scheduler. Dieses Verfahren wird bis zu dem Scheduler der untersten Schicht sukzessiv weitergeführt. Die Synchronisation erfolgt auf einer Schicht allein in Abhängigkeit von der Konfliktrelation dieser Schicht. Daran erkennt man schon einen großen Vorteil dieser Verfahrensweise, daß auf jeder Schicht ein anderes Grundsynchronisationsverfahren verwendet werden kann.

Nehmen wir an, daß auf der i-ten Schicht das strikte Zweiphasen-Sperrverfahren verwendet wird. Das heißt, man spezifiziert für jede Aktion dieser Schicht eine eigene Sperre mit der dazugehörenden Kompatibilitätstabelle. Wird eine Aktion ausgeführt, muß für diese die entsprechende Sperre gesetzt werden. Diese wird erst wieder freigegeben, wenn die Aktion beendet wurde.

- **unsichere und sichere Freigabe:** Realisierbar, wenn N2PL oder 2PL als Synchronisationsprotokoll verwendet wird. Es sind aber auch optimistische Concurrency-Strategien denkbar.
- **flexible Sperrmodi:** In den verschiedenen Abstraktionsstufen existieren bereits unterschiedlichste Sperrmodelle, die die Semantik der auf dieser Schicht angewendeten Operationen beinhalten.
- **vorzeitiges Commit von Komponententransaktionen:** Wie alle Transaktionen, die der Klasse der geschachtelten Transaktionen angehören, ist vorzeitiges Commit realisierbar.

- **Kooperationsmechanismen:** Die Weitergabe bzw. das Verleihen von Sperren kann nur auf Transaktionen der selben Ebene erfolgen, weil nur in dieser das gleiche Sperrmodell angenommen werden darf.
- **Spezifikation kooperativer Abläufe:** Schwer zu realisieren, aufgrund der Unvereinbarkeit mit dem Mehrschichtenserialisierbarkeitsbegriff.
- **Kooperationsmengen:** Jede Schicht kann mit einer eigenständigen Protokollstrategie synchronisiert werden, was eine Ungleichbehandlung der Transaktionen unterschiedlicher Ebenen erzeugt. Denkbar ist, daß eine Schicht z.B. durch das N2PL-Protokoll synchronisiert wird.

2.3.4 Synchronisation in objektorientierten Systemen

In objektorientierten Datenbanksystemen sind Methodenaufrufe auf Objekten immer als eigenständige geschachtelte Transaktion zu sehen. Innerhalb der Implementierung einer Methode besteht die Möglichkeit einer beliebigen Schachtelung von Methodenaufrufen und damit verbundener Objekte. In [GNR90] wird für die Konsistenzsicherung in objektorientierten (OO) Datenbanksystemen der Begriff der OO-Serialisierbarkeit eingeführt.

Definition 5:
*Eine **Methode** oder **Aktion** m eines Objektes O wird in parameterisierter Form folgendermaßen geschrieben: $O.m(parameter)$ oder $O.m()$ ohne Parameter. Wird in m eine andere Methode m' aufgerufen, wird dies mit dem Ausdruck $m \rightarrow m'$ bezeichnet. Die **transitive Hülle** bezeichnet $\rightarrow^+$. Ist die Methode m selbst wieder in der Hülle, wird dieser Fall mit $\rightarrow^*$ gekennzeichnet. Eine Methode $O.m$ wird **primitiv** genannt, wenn sie keine weiteren Methodenaufrufe beinhaltet. Die Menge der primitiven Aktionen auf einem Objekt O wird mit PRI_O bezeichnet.* △

Der Transaktionsbegriff muß an die Aufrufschachtelung von Methodenaufrufen angepaßt werden. Dies spiegelt sich auch in Definition 6 wider:

Definition 6:
*$O.t_i$ ist eine Aktion auf dem Objekt O. Die **OO-Transaktion** $O.t_i$ besteht aus:*

- *der Aktion $O.t_i$ selber*
- *eine Menge von Aktionen A_{iw} der Form*
  $A_{iw} = \{ a_{iwj} | t_i \rightarrow^* a_{iw} \wedge a_{iw} \rightarrow a_{iwj} \}$ *und*
- $\forall A_{iw}$ *existiert eine partielle Ordnung (precedence relation)* $\pi_{iw} \subseteq A_{iw} \times A_{iw}$ △

Aus dieser Definition ist erkennbar, daß eine OO-Transaktion einen Baum, siehe Abbildung 2.5, äquivalent zu den geschachtelten Transaktionen bildet. Im Gegensatz zu den Schichtentransaktionen sind die Schichten hier variabel. Die partielle Ordnung ist definiert durch eine links nach rechts Besuchsstrategie der Knoten bzw. Blätter.

Definition 7:
*Ein **OO-Transaktionssystem** ist das Paar TS=(OBJ,TOP) bestehend aus einer Menge von Objekten OBJ und einer Menge von OO-Transaktionen TOP, auch als top-level-Transaktionen bezeichnet. Die Menge ACT_O der Aktionen auf dem Objekt $O \in OBJ$ ist*

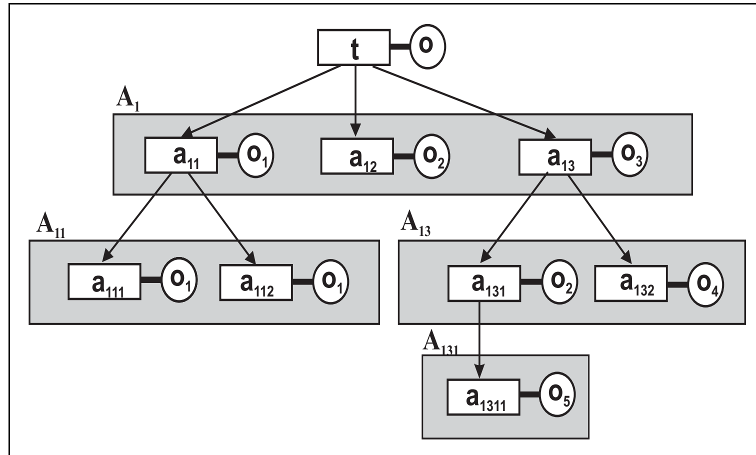

Abbildung 2.5: Transaktion in objektorientierten Systemen

definiert durch:
$ACT_O := \{a | O.a, T \rightarrow^* a, T \in TOP\}$. △

Um zwischen Transaktionen und Aktionen zu unterscheiden, werden virtuelle Objekte eingeführt. Eine Transaktion ist demnach eine Aktion, welche zuerst auf ein Objekt zugreift. Greift eine Aktion a, welche direkt oder indirekt von einer Transaktion t aufgerufen wird, auf das selbe Objekt O wie t zu, wird ein virtuelles Objekt O' siehe Definition 8 konstruiert.

Definition 8:
Sei a,b,t,u $\in ACT_O$ wie aus Definition 7. Für jede Aktion a mit $O.t \rightarrow^+ O.a$ und $\nexists$ u mit $O.t \rightarrow^+ O.u \rightarrow^+ O.a$ wird TS=(OBJ,TOP) folgendermaßen erweitert.

- *ein Objekt O' mit $ACT_{O'}$:=ACT_O-{t} wird hinzugefügt,*
- *ACT_O ist definiert durch $ACT_O := ACT_O - \{a\}$ und*
- *$\forall b \in ACT_O, b' \in ACT_{O'}$: b=b' und b→b' wird hinzugefügt.*

Mit dieser Erweiterung läßt sich die Menge der Transaktionen auf dem Objekt O definieren: $TRA_O := \{t | T \rightarrow^* t \rightarrow^+ a, T \in TOP, a \in ACT_O\}$ △

Transaktionen werden mit t,t'... und Aktionen mit a,a',b... bezeichnet. Mittels dieser Definitionen läßt sich ein Serialisierbarkeitsbegriff für die Aktionen und Transaktionen auf einem Objekt herleiten.

Definition 9:
*Ein **Objekt-Schedule** für eine Objekt O besteht aus dem 4-Tupel Sch_O=(TS,O,<,$\ll$) mit*

- *TS=(OBJ,TOP) und O∈OBJ,*
- *einer Aktionsordnung $< \subseteq ACT_O \times ACT_O$ und*
- *einer Transaktionsordnung $\ll \subseteq TRA_O \times TRA_O$*

△

In Definition 10 folgt nun der Begriff Serialität für Objekt-Schedules. Damit wird ein Bezug zur konventionellen Theorie hergestellt.

Definition 10:
$Sch_O = (TS, O, <, \ll)$ *ist* ***seriell***[7]*, wenn*

$$\forall t, t' \in TRA_O, t \neq t' : t \rightarrow^+ a, t' \rightarrow^+ a' \Rightarrow \forall O.b, O.b' : t \rightarrow^+ b, t' \rightarrow^+ b' : b < b'$$

△

Ableitend aus der konventionellen Theorie kann für die Transaktions- bzw. Aktionsordnung der Begriff des Konfliktes bzw. der Kommutativität genutzt werden. Für zwei Aktionen a, a' gilt a⊕a', wenn sie kommutieren bzw. a◇a', wenn sie in Konflikt stehen. Der Nachteil gegenüber Schichtentransaktionen besteht darin, daß zwischen Operationen, die auf verschiedenen Schichten ausgeführt werden, Konfliktrelationen definiert werden müssen. Daraus ergibt sich die Definition 11:

Definition 11:
Die ***Aktionsordnung*** $<\subseteq ACT_O \times ACT_O$ *ist definiert als: für a,a'∈* ACT_O *ist*

a<a' :⇔ a,a'∈ $PRI_O \wedge a \Diamond a' \Rightarrow$ *entweder a<a' oder a'<a oder*
es existiert ein Sch'=(TS,P,<',≪') mit $a \ll' a'$[8].

Die ***Transaktionsordnung*** $\ll\subseteq TRA_O \times TRA_O$ *ist definiert als:*

t≪t' :⇔ $(t \rightarrow, t' \rightarrow a' : a < a' \wedge a \Diamond a')$ *oder*
$(t \rightarrow u, t' \rightarrow u' : u \ll u' \wedge u \Diamond u')$.

△

Die Aktionsordnung gibt alle Paare von Aktionen an, bei denen die Ergebnisse der einen Aktion sich auf die Effekte der anderen Aktion bzw. umgekehrt auswirken. Analog zur konventionellen Serialisierbarkeitstheorie folgt nun der Äquivalenzbegriff zweier Objekt-Schedules und daraus ableitend der OO-Serialisierbarkeitsbegriff.

Definition 12:
Zwei Objekt-Schedules Sch=(TS,O,<,≪) und Sch'=(TS,O,<',≪') sind ***äquivalent****, wenn sie die gleiche Transaktionsordnung besitzen. Damit ist ein Objekt-Schedule* ***oo-serialisierbar****, wenn*

- *es existiert ein äquivalenter serieller Objekt-Schedule Sch'=(TS,O,<',≪') und*
- *< ist azyklisch*

△

In einem Transaktionssystem existieren eine Menge von Objekt-Schedules, zwischen denen auch Abhängigkeiten bestehen. Aus diesem Grund wird das Verständnis eines Schedules und der Abhängigkeitsordnung erweitert, siehe Definition 13. Die Menge ADD_O sind die Aktionen, welche von den Aktionen auf O abhängen.

Definition 13:
Ein ***Systemschedule*** *ist definiert als SSch={Sch(TS,O,<,≪)|(O∈OBJ)}. Die Menge* ADD_O *ist definiert als:*

[7] zusätzlich muß < total sein
[8] a,a' können in Sch' Transaktionen sein

$ADD_O := \{b|P.b \rightarrow Q.c', O.a \rightarrow Q.c\}$.
Die ***erweiterte Aktionsordnung*** $<\!+ \subseteq (ACT_O \cup ADD_O) \times (ACT_O \cup ADD_O)$ *eines Objekt-Schedules ist wie folgt definiert: für a∈* $ACT_O, b \in ADD_O$
a<+b:⇔ ein Schedule Sch'=(TS,R,<',≪') existiert mit a≪'b,
b<+a:⇔ ein Schedule Sch"=(TS,R,<",≪") existiert mit a≪"b, △

Aus der erweiterten Aktionsordnung wird die Definition 14 für OO-Serialisierbarkeit für System-Schedules hergeleitet.

Definition 14:
Ein System-Schedule ist serialisierbar, wenn für alle Objekt-Schedules Sch=(TS,O,<,≪) gilt:

- *Sch ist oo-serialisierbar und*
- *<+ ist azyklisch*

△

Die Synchronisation auf der Basis des oo-Serialisierbarkeitsbegriff erfolgt transparent bei dem Aufruf einer Methode. Aus diesem Grund wird auf eine Einordnung bezüglich der Kriterien verzichtet.

2.3.5 Workspace-Konzept

Workspaces sind motiviert durch die hierarchische Zerlegung von Entwurfsaufgaben und die oft langen Bearbeitungsphasen. Sie sind logische Datenbanken, welche die Objekte der entsprechenden Anwendungsebene enthalten. Es existieren verschiedene Arten von Workspaces, siehe [KSUW85]. Der **öffentliche** Workspace (*public* DB) enthält Objekte, welche allen Entwicklern zugänglich sind. Weiterhin existieren noch **private** Workspaces, welche nur einem einzigen Entwickler zugeordnet sind und **Gruppen**-Workspaces. Diese gehören einer Gruppe von Entwicklern, die mit der gleichen Teilaufgabe betraut wurden. Ein Beispiel für Workspaces ist in Abbildung 2.6 angegeben.

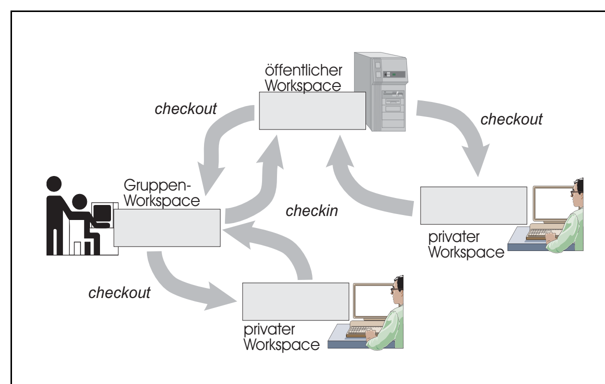

Abbildung 2.6: Workspaces und Operationen

Um Objekte zwischen Workspaces auszutauschen, stehen zwei Operationen siehe [DRR90] zur Verfügung:

- **check-out** transferiert ein Objekt aus einem Vorgänger-Workspace [9] in den Workspace, in dessen Kontext die Operation aufgerufen wurde, um es dort zu bearbeiten. Der Vorteil dieser Operation ist, daß mit jeder `checkout`-Operation ein persistenter Lock auf dem Objekt im Eltern-Workspace verbunden ist. Dieser überlebt Systemabstürze und wird erst wenn die lokalen Arbeiten vollständig beendet sind oder explizit mit der nächsten Operation freigegeben.

- **check-in** transferiert ausgecheckte Objekte zurück in den Original-Workspace. Dies ist meistens verbunden mit der Erzeugung einer neuen Version des Objektes und der Freigabe für andere `checkout`-Operationen.

Die Synchronisation erfolgt auf der Basis von geschachtelten Transaktionen. Jede `checkout`-Operation stellt dabei eine Sperranforderung dar bzw. das Checkin beinhaltet die Freigabe der Sperre. In [KSUW85] werden basierend auf privaten und Gruppen-Workspaces zwei Transaktionstypen, die Gruppen- und Nutzertransaktionen, eingeführt. Die Gruppentransaktionen mit Gruppen-Workspace holen sich aus der öffentlichen Datenbank über die `checkout`-Operation die Objekte. Die Nutzertransaktionen mit privatem Workspace beziehen ihre Objekte aus einer Gruppendatenbank und sind zudem Sub-Transaktionen der Gruppentransaktionen. Die Synchronisation erfolgt zweistufig, indem beide Transaktionsarten unterschiedlich behandelt werden. Die Gruppentransaktionen werden untereinander isoliert. Das heißt, hier greift der Serialisierbarkeit über entsprechende strenge Protokolle. Dagegen greifen bei den Nutzertransaktionen weniger restriktive Protokolle, so daß unter ihnen gute Kooperationsmöglichkeiten bestehen.

- **unsichere Freigabe:** Innerhalb der Nutzertransaktionen erfolgt aufgrund, daß nicht S2PL auf dieser Ebene verwendet wird, stets eine unsichere Freigabe[10].

- **sichere Freigabe:** Für einzelne Objekte kann ein explizites Commit gesetzt werden, so daß ein Rücksetzen nur noch von Hand erfolgen kann.

- **flexible Sperrmodi:** Es werden flexible Sperrmodi und auch Sperrkonvertierung unterstützt siehe auch [KSUW85].

- **vorzeitiges Commit von Komponententransaktionen:** Die Resourcen von Nutzertransaktionen können jederzeit durch vorzeitiges Commit freigegeben werden.

- **Kooperationsmechanismen:** Es werden die Vergabe und das Verleihen von Objekten unterstützt, indem Objekte von einem Workspace in einen anderen Workspace übertragen werden.

- **Spezifikation kooperativer Abläufe:** In diesem Modell ist eine solche Spezifikation nicht vorgesehen.

- **Kooperationsmengen:** Durch die Unterscheidung von Gruppen- und Nutzertransaktionen ist eine solche Modellierung schon impliziert.

[9] siehe die Terminologie bei geschachtelten Transaktionen
[10] wenn kaskadierender Abbruch implementiert

2.3.6 Synchronisation über Konsistenzbedingungen

Bis jetzt sind wir davon ausgegangen, daß die Maßeinheit der Konsistenz immer eine Transaktion ist. Dazu haben wir den Begriff der Serialisierbarkeit eingeführt. In vielen Anwendungen ist die Serialisierbarkeit nicht notwendig, um Konsistenz der Daten zu erreichen, sondern es reicht aus, die Konsistenz an Bedingungen zu knüpfen. In [KoSp88] werden diese Bedingungen an explizite Konsistenzprädikate, welche in konjunktiver Normalform vorliegen, gebunden. Eine Transaktion ist dann konsistenzerhaltend, wenn sie den Datenbankzustand in einen anderen überführt und beide den Konsistenzprädikaten genügen.

2.4 Fazit

Aufgrund der zwingenden Unterstützung von hierarchischen Aufgabenstrukturen und langen Entwicklungsphasen müssen die Schachtelung von Transaktionen sowie eine Unterstützung langer Transaktionen gefordert werden. Geeignete Verfahren, um diesen Bedingungen gerecht zu werden, basieren auf dem Workspace-Konzept und zumindest auf geschlossen geschachtelten Transaktionen. Einen Überblick verschafft dazu noch einmal Tabelle 2.4.

	UF	SF	FS	VC	KMc	KA	KMn
Basisprotokolle	+	+	+	-	-	-	-
Mehrversionen	+	+	-	-	-	-	-
geschlossen	-	-	+	-	+	-	+
offen	+	+	+	+	+	-	+
Mehrschichten	+	+	+	+	+	-	+
Workspace	+	+	+	+	+	-	+

Tabelle 2.4: Eigenschaften der Synchronisationsarten

Legende:

UF	unsichere Freigabe
SF	sichere Freigabe
FS	flexible Sperrmodi
VC	vorzeitiges Commit von Subtransaktionen
KMc	Kooperationsmechanismen
KA	kooperative Abläufe
KMn	Kooperationsmengen

Die Synchronisation basiert in den semantisch stärkeren Transaktionsmodellen auf der Grundlage der Protokolle, welche in flachen Transaktionen verwendet werden. Deshalb sind auch ihre Möglichkeiten abhängig von den Basissynchronisationsmechanismen. Aus diesem Grund werden unterschiedliche Transaktionstypen unterstützt, welche unterschiedlich synchronisiert werden. Beispiele dafür sind in [ScUn92, BKK85] gegeben. Mit diesem Mittel lassen sich gut Kooperationsmengen abbilden.

Wird Wert auf die Unterstützung von Kooperation auf Datenbankebene gelegt, sollten offen geschachtelte Transaktionen bzw. Mehrschichtentransaktionen, wenn eine feste Struk-

tur gegeben ist, verwendet werden. Als Synchronisationsverfahren ist entweder das 2PL- oder das N2PL-Protokoll zu verwenden, da nur sie direkt eine frühe Freigabe von Objekten beinhalten. Wird nicht kaskadierendes Zurücksetzten gefordert, kann das S2PL so modifiziert werden, daß bestimmte Sperren frühzeitig freigegeben werden. Ist dagegen die Konsistenz der Daten wichtig, müssen geschlossen geschachtelte Transaktionen, mit einem S2PL-Protokoll verwendet werden.

In der Regel sind bei den verwendeten Synchronisationstechniken immer Kompromisse zwischen Leistungsverhalten, Kooperationsmöglichkeiten, Konsistenzkriterien und zu betreibenden Aufwand zu schließen. Denn ein höheres Kooperationsverhalten wirkt sich sofort auf die Konsistenz und den zu betreibenden Aufwand aus.

Kapitel 3

Lösungsansatz

In diesem Kapitel werden zunächst die zur Verfügung stehenden Datenbanksysteme untersucht. Danach folgt ein Lösungsansatz, in dem die erweiterten Datenbanksystemkonzepte in einem ODA+ realisiert werden.

3.1 Werkzeugbetrachtung Datenbanksysteme

Aufgrund der komplexen Strukturen der Daten in der dreidimensionalen Modellierung und deren Einbettung in eine CORBA-Umgebung fiel die Entscheidung für das grundlegende Datenmodell des Datenbanksystems sehr leicht. Relationale Datenbanksysteme eignen sich vorwiegend für flache Strukturen. Zudem wird in CORBA mittels der *Interface Definition Language* objektorientiert modelliert. Aus diesen Gründen sind objektorientierte Datenbanksysteme für diesen Anwendungsfall ausgewählt worden.

Das zu verwendene OODBS muß eine breite Unterstützung der in Kapitel 1.5 und 2 aufgestellten Kriterien anbieten. Weil in unserem Anwendungsfall stark synchron kooperiert wird, muß die Behinderung, die durch das Setzen von Sperren auftritt, relativ klein sein. Es kann von niemandem verlangt werden, daß er erst einige Stunden warten muß, bevor er seine Objekte erhält. Zu diesem Zweck muß die Zeit, für die ein Objekt gesperrt ist, sehr kurz gehalten werden. Dies wird durch eine möglichst frühe Freigabe einer Sperre erreicht, sei sie nun sicher oder unsicher.

Weiterhin sind aus praktischer Sicht die Schachtelung von Transaktionen, die Unterstützung flexibler Sperrmodi auf Objektebene und lange Transaktionen wichtig und sollten deshalb für die Auswahl eines der angeführten objektorientierten Datenbanksysteme als wichtige Kriterien[1] gewertet werden.

Die Hauptvertreter objektorientierter Systeme sind **Objectstore**, **Objectivity**, **O_2**, **Poet** und **Versant**. Sie unterscheiden sich vor allem in der verwendeten Architektur. Die ersten drei sind Vertreter einer Pageserver-Architektur. Die beiden letzteren besitzen eine Objectserver-Architektur.

[1]Die Verfügbarkeit des Datenbanksystems in seiner neuesten Version (**O_2**, **Versant**) besaß auch einen großen Einfluß bei der Auswahl

In allen Datenbanksystemen werden die Objekte nicht 1:1 vom transienten in den persistenten Speicher abgebildet, sondern zerlegt und in Seiten (Pages) eingebettet. Dadurch muß bei einer Wiederherstellung eines Objektes, dieses aus seinen physischen Bestandteilen zusammengesetzt werden muß. Die Architekturen unterscheiden sich dort, wo diese Transformation erfolgt. In dem Fall der Pageserver-Strategie werden dem Client vom Datenbankserver Seiten geschickt, welcher damit auf Client-Ebene eine Transformation vornimmt. Die Objectserver-Strategie realisiert dagegen die Transformation schon auf Server-Seite. Der Client besitzt nur noch einen Objekt-Cache. Jede Architektur hat, wie aus den folgenden Punkten zu ersehen, ihre Vor- und Nachteile.

- Ein Nachteil der Objektstrategie ist, daß der Server für die korrekte Speicherung und Rekonstruierung von Objekten Informationen über die Client-Plattform und Client-Sprache besitzen und eine entsprechende Transformation realisieren muß. Die Pageserver-Architektur hat diesen Nachteil nicht, weil die Transformation von Seitenebene auf Objektebene erst im Client realisiert wird. Für unsere Anwendung braucht dieser Nachteil aber nicht betrachtet werden, weil die Datenbank-Clients nur auf einer Plattform und nur in der C++–Programmiersprache entwickelt worden sind.

- Ein weiterer Nachteil der Objectserver-Architektur besteht darin, daß der Server sehr viel über das Aussehen der Objekte wissen muß. Aus diesem Grund ist die Definition des Datenbankschemas häufig auf die vom System definierten Klassen begrenzt bzw. müssen von diesen abgeleitet sein. Ein Pageserver hat dieses Problem nicht, da er vom Aussehen eines Objektes nichts weiß. Diese Einschränkung ist aufgrund des gesteigerten Implementierungsaufwands nur störend zu werten, weil viele CORBA-Strukturen z.B. `sequence` und CORBA-Pointer auf jeden Fall separat abgebildet werden.

- Ein Vorteil der Objektstrategie besteht darin, daß die Objekte schon auf der Server-Seite existieren. Die Netzwerkauslastung sinkt, weil immer nur die wirklich benötigten Objekte übertragen werden. Durch die direkte Anfragebearbeitung auf der Server-Seite ist die Auswertung von Anfragen wesentlich effizienter. Außerdem besitzen die Pageserver mit Ausnahme von O_2 ein Problem mit dem Locking auf Objektebene, weil sie nur Sperren auf der Page-Ebene unterstützen.

Für die Entscheidung für oder gegen ein System werden die unterstützten Features den Ausschlag geben. Die genannten Systeme mit ihren Eigenschaften werden in [Stei97] verglichen und kurz vorgestellt. Die für uns wichtigen Punkte sind in Tabelle 3.1 angegeben.

Die Entscheidung über die Datenbankarchitektur fiel auf einen Objectserver bzw. einen Pageserver, welcher Objektsperren erlaubt, weshalb Objectstore und Objectivity aus der engeren Auswahl fielen. Poet unterstützt auf Transaktionsebene kein 2-Phasen-Protokoll und Rollback, was in Bezug auf Transaktionen eine erhebliche Einschränkung ist. Aus diesem Grund werden nur noch die Systeme O_2 und Versant weitergehend auf die noch geforderten Kriterien untersucht. Dies erfolgt in den beiden folgenden Abschnitten.

Produkt	Objectivity	Objectstore	O_2	Poet	Versant
Architektur	Page	Page	Page	Object	Object
Optimistisches Sperren	+	-	+	+	+
Objektversionierung	+	-	+	-	+
Ereignisverwaltung	+	+	+	+	+
2PL	+	+	+	-	+
Lange Transaktionen	+	+	+	+	+
Dirty Read	+	+	+	+	+
Rollforeward	+	+	+	+	+
Rollback	+	+	+	-	+

Tabelle 3.1: Vergleich von ODBMS nach [Stei97]

3.1.1 Das OODB O_2

O_2 [O$_2$96b] ist ein objektorientiertes DBMS basierend auf einem Client-Server-Modell. In diesem können sich mehrere Clients an einen Datenbankserver binden. In O_2 werden Programmiersprachen nicht um Persistenkonzepte erweitert wie in Versant und Objectstore, sondern O_2 besitzt ein eigenständiges orthogonales Typsystem und eine eigenständige Sprache – O_2C. Zusätzlich besitzt O_2 eine Anfragesprache, woraus die OQL der ODMG hervorgegangen ist. Weitere Informationen können aus [HeSc95, O$_2$96b] entnommen werden.

Für die Nutzung von O_2–Objekten und der Anfragesprache in C++ wird ein ODMG-konformes Binding an diese Programmiersprache unterstützt. In diesem Zusammenhang wird allerdings nur ein flaches Transaktionsmodell bereitgestellt. Durch einen Umweg kann man eine Schachtelung erreichen, indem Transaktionsobjekte geschachtelt erzeugt werden. Daß heißt, ein Transaktionsobjekt wird deklariert und erzeugt innerhalb der Operationsaufrufe BOT und EOT ein zweites Transaktionsobjekt. Die Ergebnisse der ersten Transaktion werden erst sichtbar, wenn das Commit der zweiten positiv verlaufen ist. Parallelität zwischen den Kindtransaktionen kann aber nicht erreicht werden. Synchronisiert werden die Transaktionen über ein 2PL-Protokoll. Es existieren aber keine Möglichkeiten, in den Synchronisationsprozeß einzugreifen.

O_2 unterstützt eine Sperrenvergabe auf Objektebene. Analog zu Versant werden auch hier mit bestimmten Operationen automatisch Sperren auf dem Objekt gesetzt. Es ist aber auch möglich, eine Sperre explizit zu setzen. Die Sperrmodi beschränken sich auf *shared*- und *exclusive*-Sperren. Eine *browse*-Sperre kann über einen Versionsmechanismus erreicht werden.

3.1.2 Das OODB Versant

Versant ist ein objektorientiertes Datenbanksystem, in dem die C++–Programmiersprache um Persistenzkonzepte erweitert wurde. Das Objektmodell von Versant unterstützt neben den Konzepten wie Klassen, Vererbung, Polymorphismus und Datenkapselung auch komplexe Beziehungen den damit verbundenen navigierenden Zugriff und das Stellen von

Ad-Hoc-Anfragen. Für diesen Zweck wird eine SQL-ähnliche Anfragesprache, die um die Bearbeitung von mengenwertigen Attributen erweitert wurde, verwendet. Das notwendige Schema-Mapping vollzieht sich automatisch.

Ein Vorteil bieten die Verteilungsmöglichkeiten in Versant. Für diesen Zweck basiert Versant auf dem Client/Server-Modell, siehe Abbildung 3.1.

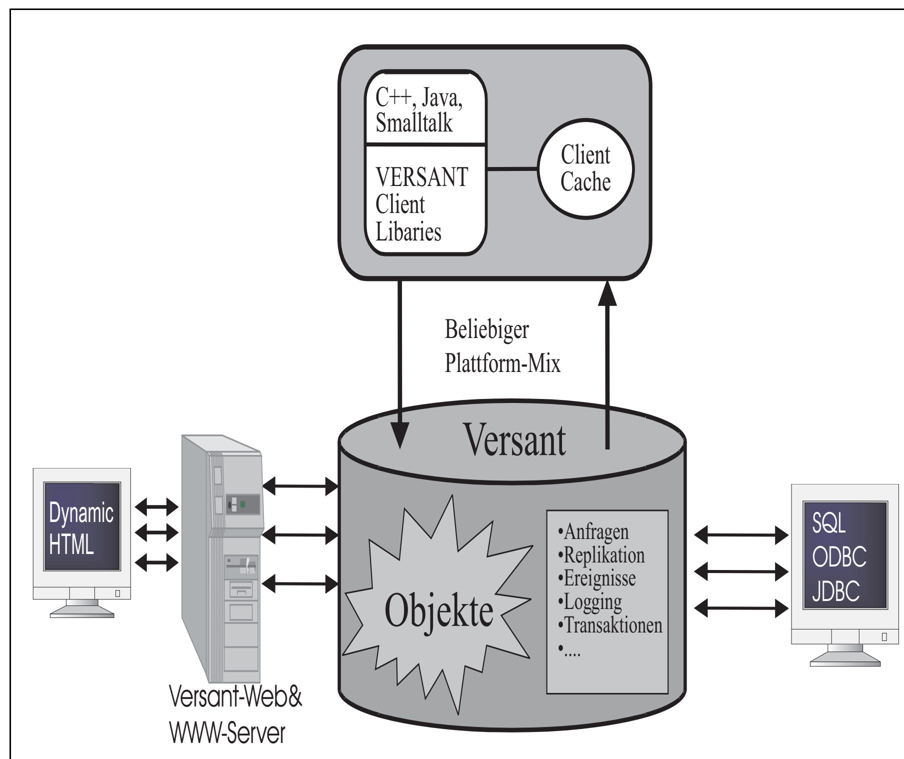

Abbildung 3.1: Versant Client/Server-Architektur

Clients können z.B. C++-, Smalltalk- oder Java-Applikationen sein. Für diese Programmiersprachen existiert aus diesem Grund auch ein Binding. Für C++ kann mit diesem auch ODMG-konform gearbeitet werden. Weiterhin existieren Schnittstellen für einen SQL-Zugriff (SQL-92), für *Open DataBase Connectivity* (ODBC) und für *Java DataBase Connectivity* (JDBC). Datenbanken und daraus folgend auch Objekte können auf verschiedene Knoten verteilt werden. Die knotenübergreifende Beziehungsmodulierung wird durch den Einsatz von unveränderbaren, ortsunabhängigen und systemweit eindeutigen Objektidentifikatoren erreicht. Der transaktionale Zugriff auf Objekte verteilter Datenbanken erfolgt transparent, indem die betroffenen Datenbanken miteinander kommunizieren.

Hervorzuhebende Eigenschaften sind die Ereignisverwaltung und die Datenbankreplikation. Eine Synchronisation der Replikate erfolgt automatisch, womit eine ständige Verfügbarkeit bei Hardware-, Software- und Netzwerkfehlern gegeben ist. In den folgenden Unterkapiteln wird auf spezielle Datenbank-Features, siehe [Vers97a, Vers97b, Vers97c], eingegangen, welche für die Auswahl maßgeblich sind.

3.1.2.1 Transaktionen

Im Rahmen des ODMG-konformen C++–Bindings, welches auch Versant bereitstellt, werden nur kurze Transaktionen gefordert. Für die Nutzung von weitergehenden Konzepten werden in Versant zusätzliche Mechanismen zur Verfügung gestellt. Es existieren generell zwei Transaktionskonzepte in Versant, die kurzen und langen Transaktionen. Beide sind als Einheit zu sehen. Die Parameter, mit der eine lange Transaktion gestartet wird, bestimmt das Verhalten der eingebetteten kurzen Transaktion. Deswegen folgt zuerst ein Beschreibung der langen Transaktionen.

3.1.2.2 Lange Transaktionen

Lange Transaktionen können mit unterschiedlichen Parametern gestartet werden. Diese bestimmen, ob mit geschachtelten oder flachen Transaktionen, mit welchem Sperrmodell oder mit welcher Synchronisationsmethode gearbeitet wird. Zusätzlich unterstützen sie das Konzept lang andauernder Aktivitäten.

In langen Transaktionen werden lang andauernde Aktivitäten über ein eingeschränktes Workspace-Konzept realisiert. Dazu existieren private und öffentliche Workspaces, zwischen denen Objektmengen[2] und notwendige Schemainformationen mit den aus Kapitel 2.3.5 bekannten Operationen transferiert werden. Die Begriffe Workspace und physische Datenbank sind hier äquivalent. Es gilt die Restriktion, daß nur aus öffentlichen in private Workspaces ausgecheckt werden darf. Aus diesem Grund darf ein ausgechecktes Objekt nicht wieder ausgecheckt werden. Diese Restriktion ist für die Bildung von Projektteams sehr einschränkend, weil die hierarchische Zergliederung von Aufgaben nicht modelliert werden kann. Für jedes ausgecheckte Objekt wird in der öffentlichen Datenbank eine persistente Sperre gesetzt, siehe Kapitel 3.1.2.6. Es ist erlaubt, aus verschiedenen öffentlichen Workspaces Objekte in ein privates Workspace zu transferieren.

Eine lange Transaktion erhält einen Namen, um sie bei einer eventuellen Weiterführung der Arbeit zu identifizieren. So ist es möglich, eine lange Transaktion auf unterschiedliche Weise zu beenden. Zum einen kann die Arbeit mit den normalen Operationen Commit und Rollback beendet werden. Dabei werden automatisch alle ausgecheckten Objekte in die Vater-Workspaces zurücktransferiert und die lange Transaktion beendet. Zum anderen existiert die Möglichkeit, eine lange Transaktion so zu beenden, daß die Arbeit zu einem späteren Zeitpunkt über die Angabe des langen Transaktionsnamen wieder aufgenommen werden kann. Die gesetzten persistenten Sperren bleiben in diesem Fall erhalten.

3.1.2.3 Kurze Transaktionen

Kurze Transaktionen besitzen in Versant die ACID-Eigenschaften und können zusätzlich noch verteilt sein. Dies bedeutet, daß eine Transaktion auf Objekten arbeiten kann, die sich in verschiedenen Datenbanken befinden. Diese Datenbanken können sich in einer verteilten Datenbankumgebung befinden. Wird eine lange Transaktion gestartet, beginnt automatisch auch eine kurze Transaktion. Es werden aber zusätzliche Transaktionsoperationen bereitgestellt, wie das Setzen von Sicherungspunkten bzw. ein Rücksetzen auf

[2] Es ist die Angabe einer Bedingung unterstützt

den Zustand zum Zeitpunkt des Sicherungspunktes und Erweiterungen für die normalen Endeoperationen, welche die Verwaltung des Objekt-Caches unterstützen. Kurze Transaktionen werden standardmäßig über ein striktes 2-Phasen Protokoll synchronisiert.

3.1.2.4 Optimistisches Locking

Treten in der Anwendung eine geringe Anzahl von Konflikten auf, so sind optimistische Verfahren den Sperrverfahren überlegen. Im voreingestellten Fall werden Transaktionen über Sperren synchronisiert. In Versant existiert nun die Möglichkeit, auf ein optimistisches Verfahren umzuschalten. Zum einen sind für die Umschaltung beim Start der umgebenden langen Transaktion die entsprechenden Parameter zu übergeben. Zum anderen ist darüberhinaus für jede Klasse von zu benutzenden Objekten eine Schemaänderung durchzuführen, indem ein *timestamp*-Attribut in die Klassendefinition eingefügt wird. Über dieses Attribut erfolgt die Synchronisation unter optimistischen Transaktionen. Es wird auch gegenüber anderen sperrenden Transaktionen synchronisiert. Optimistisches Locking ist nicht kombinierbar mit dem folgenden geschachtelten Transaktionskonzept.

3.1.2.5 Geschachtelte Transaktionen

Man muß zwei Konzepte in Versant kombinieren, um geschachtelte Transaktionen zu nutzen. Das erste besitzt syntaktischen Charakter. Mit der Nutzung dieses Konzeptes kann in einer solchen geschachtelten Transaktion wiederholt durch Methodenaufruf eine geschachtelte Transaktion mit der entsprechenden Nestungsstufe erzeugt bzw. eine geschachtelte Transaktion beendet werden. Datenzugriffe erfolgen nicht nur in den Blatttransaktionen. Für die Synchronisation innerhalb geschachtelter Transaktion werden die Sperren erweitert. Dazu wird ein *combined* Lock-Modus eingeführt, mit dem spezifizierbar ist, ob eine Sperre nach Ende der Transaktion an die Vatertransaktion weitervererbt wird.

Mit dem ersten Konzept läßt sich keine Parallelität zwischen Kindtransaktionen einer Vatertransaktion erreichen. Darum werden im zweiten Konzept mehrere Prozesse unterstützt. Für jede geschachtelte Transaktion wird ein neuer Kindprozeß erzeugt und der Vorgängertransaktion als geschachtelte Kindtransaktion zugeordnet.

3.1.2.6 Sperren in Versant

Versant bietet ein umfangreiches Sperrkonzept, bei dem alle Arten von Objekten[3] gesperrt werden können. Bei dem Aufruf von versantspezifischen Methoden werden automatisch Sperren gesetzt, es sei denn, die automatische Sperrung ist ausgeschaltet oder es wird ein benutzereigenes Sperrmodell verwendet.

Zwei Arten von Sperren werden in Versant unterschieden.

- **kurze Sperren:**
  Diese Sperren werden freigegeben, wenn eine kurze Transaktion endet. Es werden die Lockmodi (*browse*, *shared*, *update*, *exclusive*) unterstützt. Sie besitzen die in Kapitel 2.2 angegeben Eigenschaften.

[3]Hier sind Datenbankobjekte, versionierte Objekte und Klassenobjekte gemeint

- **persistente Sperren:**
  Diese Sperren besitzen alle Eigenschaften kurzer Locks, eingeschränkt auf die Modi (*browse*, *shared*, *exclusive*) und einige zusätzliche. Das Setzen einer persistenten Sperre erfolgt zunächst über das Setzen einer kurzen Sperre, die bei der nächsten Transaktionsoperation in eine persistente Sperre umgewandelt wird, um so das Ende der Transaktion zu überleben. Die zusätzlichen Features von persistenten Sperren sind:

  - **Lock Level:**
    Für eine persistente Sperre kann angegeben werden, ob andere Konkurrenten eine gesetzte Sperre brechen dürfen. Dazu besitzt eine persistente Sperre ein *lock level*-Attribut mit den zwei Zuständen *hard lock* und *soft lock*. Ein *soft lock* darf durch ein *hard lock* gebrochen werden. Dagegen darf ein *hard lock* nicht gebrochen werden.
  - **Notification:**
    Es sind drei Ereignisse spezifizierbar, um auf bestimmte Situationen zu reagieren. So werden Nachrichten verschickt, wenn eine Sperre durch einen anderen Nutzer gebrochen wird, jemand auf eine von mir gesetzte Sperre wartet oder ein Objekt wieder verfügbar ist. Alle drei Möglichkeiten sind verknüpfbar.
  - **Lock Queuing:**
    Ist eine Sperre momentan nicht verfügbar, läßt sich eine Reaktion auf diese Situation angeben. Es kann spezifiziert werden, daß die betreffende Transaktion zum einen wartet bis die *timeout*-Zeit abgelaufen ist oder zum anderen daß die Sperranforderung sofort abgelehnt wird. Der dritte Fall entspricht einer Reservierung für das Objekt. Mittels Notifikation werden in diesen Fall Nachrichten versendet, wenn eine Sperre wieder aufgehoben wird.

Eine Lock auf einem Objekt sperrt in Versant nicht automatisch alle über eine Referenz erreichbaren Objekte, setzt aber automatisch eine Sperre auf dem dazugehörenden Klassenobjekt. Für diesen Zweck existieren die Intention-Sperren (*shared, exclusive* und *shared with intention to exclusive-SIX*). Die letzte Sperre schützt ein Klassenobjekt vor Änderungen und entspricht dem Setzen einer Lesesperre auf allen Instanzen. In Tabelle 3.2 ist die Kompatibilität von normalen und Intention-Locks in Versant angegeben.

	B	IS	S	U	IX	SIX	X
B	+	+	+	+	+	+	+
IS	+	+	+	+	+	+	-
S	+	+	+	+	-	-	-
U	+	+	+	-	-	-	-
IX	+	+	-	-	+	-	-
SIX	+	+	-	-	-	-	-
X	+	-	-	-	-	-	-

Tabelle 3.2: Kompatibilitätsmatrix für Versant-Sperren

Weiterhin bietet Versant an, durch bestimmte Operationen auf Objekten innerhalb einer Transaktion Sperren zu verschärfen oder abzuschwächen.

3.1.2.7 Sperrmodellerweiterung

Reichen die in Tabelle 3.2 angegebenen Sperrmodi nicht aus, können diese um eigene erweitert werden. Zu diesem Zweck lassen sich neue Sperrmodi als Konstanten definieren und in das von Versant bereitgestellte Sperrmodell integrieren. Um diese Sperren zu synchronisieren, müssen einige Angaben über die Interaktionen der neuen Sperrmodi untereinander bzw. mit den zu integrierenden Standardsperren gemacht werden. Dazu werden mehrere Matritzen gefordert, die der Anwender spezifiziert. Diese beinhalten die Information, in welchem Verhältnis zwei Lockmodi zueinander stehen, z.B. welcher von den beiden stärker oder schwächer ist, sind die beiden kompatibel und was passiert, wenn eine Sperrkonvertierung geschieht.

3.1.3 Fazit der Datenbankanalyse

In den Betrachtungen zu den Datenbanksystemen hat sich ergeben, daß Versant die weitreichendsten Eigenschaften in Bezug auf Kooperationsunterstützung besitzt. Hervorzuheben sind dabei die Unterstützung von langen und geschachtelten Transaktionen und deren Synchronisation. Im letzteren Fall kann dabei sogar noch auf die Sperrvererbung Einfluß genommen werden.

Vergleicht man die Features von Versant und O_2 in Bezug auf Transaktions- und Sperrmodellunterstützung kann die Wahl nur auf Versant fallen. Z.B. kann mit den zwei Sperren, die O_2 anbietet, ein "dirty read" nur über Versionierung modelliert werden. Es existiert in O_2 nämlich keine Sperre, die niemals durch irgendeine andere Sperre blockiert wird. Bei den geschachtelten Transaktionen in O_2 werden alle Sperren automatisch an die Elterntransaktion weitervererbt. Es existieren keine Möglichkeiten, bestimmte Objektsperren von diesen Mechanismus auszuschließen, wodurch die kooperierende Arbeit mehrerer Personen an einem Objekt verhindert wird. Ein dritter Negativpunkt ist, daß lange Aktivitäten nur über Versionierung unterstützt werden.

Für die Granularität von Sperren sollte mindestens das Sperren auf Objekte unterstützt werden. Denn wenn auf zwei Objekte, welche sich auf der selben Seite befinden, von unterschiedlichen Transaktionen zugegriffen werden soll, wird eine von beiden unnötig behindert und schränkt damit den Kooperationsgrad ein.

3.2 Die transparente Umsetzung erweiterter Datenbankkonzepte

Ein primäres Ziel für den Nutzer von Datenbankkonzepten innerhalb einer CORBA-Umgebung ist eine hohe Unabhängigkeit gegenüber Änderungen auf der Datenbankseite. Diese Unabhängigkeit zeigt sich in den folgenden Punkten:

- **Datenbanksystemunabhängigkeit**: Darunter ist zu verstehen, daß keine Änderungen für den Client entstehen, wenn auf der Server-Seite das Datenbanksystem ausgetauscht wird.

- **logische Datenunabhängigkeit**: Nehmen wir an, daß in einer Datenbank Klassen eingefügt werden. Dann sollte sich dies nicht[4] auf bestehende Clients auswirken.
- **Datenmodellunabhängikeit**: Der Client hat bedingt durch CORBA eine objektorientierte Sicht auf die Daten der Datenbank. Aus diesem Grund sollte er unabhängig vom Datenmodell, welches das Datenbanksystem verwendet, sein.
- **Datenbankentwurfsunabhängigkeit**: In vielen Anwendungsfällen existieren Datenbanken mit den entsprechenden Anwendungen darauf. Der Entwurf dieser Datenbanken erfolgte ohne Berücksichtigung einer späteren Bereitstellung innerhalb eines CORBA-Servers. Die Nutzung solcher Datenbestände innerhalb eines CORBA-Servers darf sich nicht auf bestehende Anwendungen auswirken.

Weiterhin müssen alle Basiselemente eines Datenbanksystems äquivalent dem in [Neuw97] angebenen Abbildungsmechanismus umgesetzt werden. Dazu gehören Anfragen, Transaktionen, Beziehungen, Kollektionen und die persistenten Objekte selber. Dies soll in den folgenden Kapiteln erfolgen.

3.2.1 Die Gesamtarchitektur

Die Prozeßarchitektur, siehe Abbildung 1.5, für einen ODA in DICE wird zunächst um eine Komponente erweitert. Mit der abgeleiteten Architektur, siehe Abbildung 3.2, wird eine Zuordnung eines CORBA-Clients zu einem CORBA-Datenbankserver[5] erreicht, ohne die nicht OMG-konforme `per-client`-Registrierung eines Servers von Orbix zu verwenden. Mit dieser Art der Registrierung von Servern und den darin enthaltenen Objekten läßt sich die eindeutige Zuordnung eines Clients zu einer Transaktion realisieren. Sie besitzt aber drei entscheidende Nachteile:

- CORBA-Referenzen auf persistente Implementierungsobjekte dürfen nicht an andere Clients oder Server weitergegeben werden, weil Methodenaufrufe auf der CORBA-Referenz von anderen Clients fehlschlagen. In diesem Fall wird automatisch ein neuer Server hochgefahren, wodurch der Methodenaufruf nicht mehr im Kontext der selben Transaktion steht.
- Die gleichzeitige Nutzung einer Transaktion von unterschiedlichen Clients ist nicht vereinbar mit einer `per-client`-Registrierung des Servers, der das Transaktionsobjekt zur Verfügung stellt.
- Ein Client darf nicht zwei Transaktionsobjekte parallel besitzen, weil stets nur ein Transaktionsobjekt für jeden Client existiert.

Damit ist die gleichzeitige Verwendung der selben Transaktion durch mehrere Clients, als auch die richtige Zuordnung einer Transaktionsoperation zum richtigen Transaktionsobjekt, mit reinen Aktivierungsmechanismen nicht möglich. Aus diesem Grund wird in die Architektur 3.2 ein Server eingefügt, der die Verwaltung aller CORBA-Datenbankserver übernimmt.

[4] in Form von Neuübersetzung der Client-Anwendung
[5] Client bezüglich eines Datenbankservers

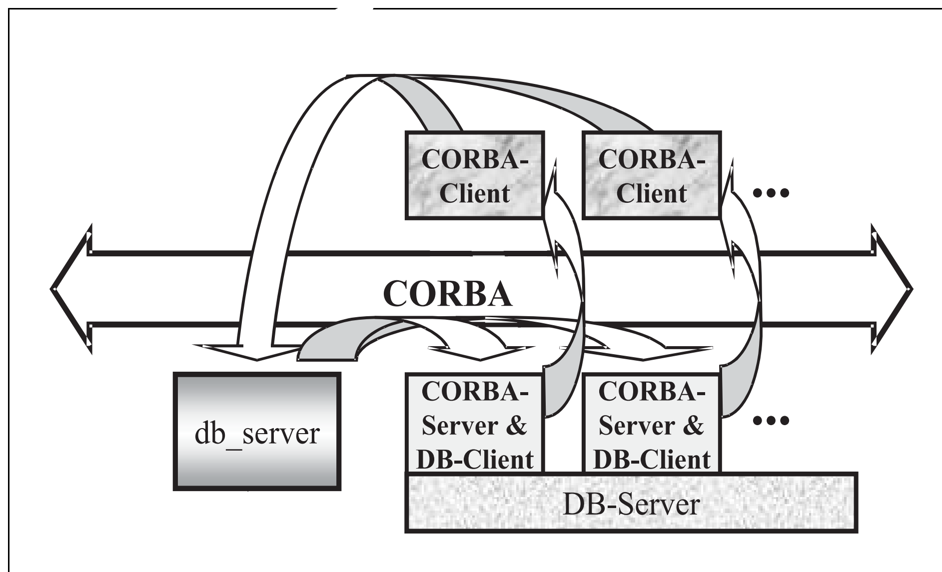

Abbildung 3.2: Die erweiterte Prozeßarchitektur

Der `db_server`-Server stellt ein Objekt bereit, welches jeder auf Datenbankobjekte zugreifende Client benutzt. Durch diesen Server wird eine eindeutige Zuordnung vom Client-Transaktionsobjekt zur Datenbanktransaktion erreicht. Es besitzt dazu folgendes Interface:

```
interface db_server {
  opennestedtransaction get_opennestedttransaction();
  closednestedtransaction get_closednestedtransaction();
  flattransaction get_flattransaction();
  void delete_transaction(in transaction ta);
};
```

Diese Schnittstelle des Server-Objektes ist der Einstiegspunkt für alle Clients, die auf persistente Objekte zugreifen. Es stellt unterschiedliche Transaktionsobjekte bereit. In diesem Fall sind es flache und offen bzw. geschlossen geschachtelte Transaktionen. Die Schnittstellen zu diesen Transaktionstypen werden in Kapitel 3.2.3 eingeführt. Zunächst wird anhand eines Beispiels die Realisierung von Persistenzkonzepten aufgezeigt. Dazu wurden die Konzepte aus [Neuw97] erweitert.

3.2.2 Die Beschreibung der entwickelten IDL-Spezifikationen

In den folgenden Abschnitten werden die Schnittstellen in der Spezifikationssprache der OMG beschrieben. Für eine generische Nutzung von Datenbankobjekten werden die notwendigen Schnittstellen vorgestellt. Das heißt, die Schnittstellen sind nicht begrenzt auf die Klassen eines bestimmten Anwendungsfalls.

3.2.2.1 Das Interface für persistente CORBA-Objekte

Persistente Objekte besitzen mehr Eigenschaften als transiente Objekte. Im Programmiersprachen-Binding von ODMG wird dazu eine Klasse eingeführt, von der alle persistenten Klassen erben müssen. In dieser Arbeit wird ein analoger Mechanismus verwendet. Es wird ein Interface eingeführt, welches die Änderung von Eigenschaften persistenter CORBA-Objekte zuläßt.

```
enum lockmodus{B,R,U,X}
interface pers_CORBA {
 void upgrade_lock(in lockmodus sperre);
 void downgrade_lock(in lockmodus sperre);
 void make_versioned();
 pers_CORBA create_version();
 list_of_pers_CORBA all_kind_version();
 pers_CORBA get_parent_version();
 void release_instance();
};
```

In diesem Interface werden zunächst drei Eigenschaften unterstützt:

- **Die Sperrenunterstützung**:
  Das Setzen von Sperren[6], siehe Kapitel 2.2.2, wird durch die Unterstützung von Sperroperationen auf dem Interface für persistente CORBA-Objekte unterstützt. Zum einen kann eine Sperre verschärft (*upgrade*) oder abgemindert (*downgrade*) werden. Diese beiden Methoden können aber im Zusammenhang mit der Transaktionssynchronisation des Datenbanksystems zu Problemen führen, so daß eine umfangreiche Ausnahmebehandlung mit diesen beiden Methoden verbunden sein muß. Zum Beispiel tritt eine Fehlersituation ein, wenn eine bestehende Sperre derartig verstärkt werden soll, daß die Zielsperre inkompatibel zu anderen bestehenden Sperren wäre.

- **Die Versionierungsunterstützung**:
  Die Eigenschaft, daß ein Objekt versioniert ist, leitet zusätzliches Verhalten von persistenten Objekten ab. So muß ein Objekt innerhalb des Versionsgraphen auf die Vaterversion und auf alle Kindversionen Zugriff besitzen. Zusätzlich darf ein versioniertes Objekt eine neue Version von sich erzeugen. Eine Versionierungsunterstützung wird mit den vier Versionierungsmethoden unterstützt und ist auch nicht vollständig, z.B. das Setzen einer *default*-Version innerhalb eines Versionsgraphen fehlt.

- **Die Speicherfreigabe**:
  Es existieren gleichzeitig zwei Objekte, zum einen das persistente Implementierungsobjekt und zum anderen das CORBA-TIE-Objekt, welches die CORBA-Client-Aufrufe an das Implementierungsobjekt weiterleitet. Aus diesem Grund wird unterschieden, welches Objekt freigegeben wird. In [Neuw97] wurde für die Freigabe

[6] unterstützt sind Browse-, Read-, Update- und Exclusive-Sperren

des Implementierungsobjektes in jeder IDL-Schnittstelle bzw. auch in der Implementierungsklasse dazu eine neue Methode eingeführt. Damit ergibt sich der Vorteil, daß die Funktion für die Freigabe des persistenten Objektes in die Methodendefinition dieses Interfaces wechselt. Es ändert sich nichts in der Verwendung dieser Funktion.

Für jede persistente Klasse, die eine CORBA-Repräsentation[7] besitzt, existiert ein Interface, welches vom `pers_CORBA`-Interface abgeleitet wird. Demzufolge sieht eine beliebige Schnittstelle **K** folgendermaßen aus.

```
interface K:pers_CORBA {
// Attribute und Methoden des Interfaces K
};
```

Jedes Objekt mit der Schnittstelle von K, das persistent erzeugt wurde, besitzt aufgrund der Vererbung zu den eigenen Eigenschaften noch ein Verhalten, welches es als persistentes Objekt auszeichnet. Der Nutzer eines derartigen Objektes besitzt dadurch z.B. die Möglichkeit, innerhalb der umgebenden Transaktion die Sperre auf dem Implementierungsobjekt zu verstärken oder abzuschwächen. Die Realisierung des zusätzlichen Verhaltens wird in Kapitel 4.1.2 vorgestellt und ist wiederum vom Datenbanksystem abhängig. Für den Client sind diese Unterschiede aber transparent.

3.2.2.2 Anfragen und Beziehungen

Für Anfragen, die Ergebniskollektionen unterschiedlicher Objekttypen erhalten, muß eine globale Oberklasse für jeden Kollektionstyp (Bag, List, Set) existieren. Der Vollständigkeit halber ist nachfolgend ein einfaches `query`-Interface angegeben. Dieses besitzt zwei Methoden. Die erste initialisiert das Query-Objekt mit einer Anfragezeichenkette. Die Syntax des Strings ist abhängig von der Realisierung. Für eine hohe Client-Transparenz muß hier ein OQL-Anfragestring unterstützt werden.

```
interface query {
 void init(in string pred);
 list_of_pers_CORBA execute();
// ...
}
```

Die Ergebnisse von Anfragen beschränken sich auf Kollektionen persistenter CORBA-Objekte. Als Ergebnis der `execute()`-Methode ist nur die Oberklasse (Interface) der Liste angegeben. Die folgende Schnittstellendefinition für eine Liste, siehe [Neuw97], ist nach ODMG nicht vollständig, kann aber um eine Iterator-Funktion oder um Mengenoperationen erweitert werden.

```
interface list_of_pers_CORBA {
 void remove_element_at(in long position);
 void add_element(in pers_CORBA object);
```

[7] auch für jede CORBA-Repräsentation mit persistenter Implementierungsklasse

```
 pers_CORBA get_element_at(in long position);
 long list_size();
 list_of_pers_CORBA query(in string pred);
};
```

Die Abbildung der anderen Kollektionstypen erfolgt analog mit entsprechenden Methodenänderungen. Mit diesem allgemeinen Interface kann aber auf der Client-Seite nicht gearbeitet werden. Aus diesem Grund existiert für jedes konkrete persistente Interface, entsprechend auch für jeden Kollektionstyp, ein konkretes Kollektions-Interface.

```
interface list_of_K:list_of_pers_CORBA {
};
```

Die Liste von K-Objekten besitzt damit die gleichen Methoden aber mit einem konkreten Verhalten. Der Client-Nutzer der `get_element_at()`-Methode der konkreten Instanz einer Liste kann sich über das Abwärts-Casting der verwendeten CORBA-Implementierung[8] ein CORBA-Objekt des konkreten Typs erzeugen. Solange der Client genau weiß, daß auf Implementierungsobjektebene auch eine Instanz des konkreten Typs existiert, treten keine Probleme auf. Im anderen Fall muß das Abwärts-Casting an das Datenbanksystem weitergereicht werden, welches besondere Methoden für diesen Zweck zur Verfügung stellt. Eine CORBA-Implementierung führt dadurch das Abwärts-Casting nur auf dem CORBA-TIE-Objekt durch. Daraus entsteht ein Laufzeittypfehler.

Die Abbildung von Referenzen erfolgt auf eine ähnliche Art und Weise. Auch hier existiert die entsprechende Oberklasse und die konkreten Unterklassen. Mit Hilfe dieser Klassen werden Beziehungen zwischen Datenbankobjekten abgebildet, siehe [Neuw97].

3.2.2.3 Die Fabrikschnittstelle

Es existieren mehrere Möglichkeiten, Objekte in CORBA zu aktivieren. Normalerweise geschieht einen Objektaktivierung automatisch durch den Basic Object Adapter (BOA), bevor der erste Methodenaufruf auf einem Objekt erfolgt. Eine Objektaktivierung wird erreicht, wenn:

- Das Objekt wird in der Server-Anwendung erzeugt. Diese Methode ist für die Realisierung von Persistenzkonzepten nicht verwendbar, weil nicht alle Datenbankobjekte in der Art bereitgestellt werden können.
- Das Objekt wird durch einen Loader-Mechanismus bereitgestellt. Falls eine Objektaktivierung fehlschlägt, wird ein Loader aufgerufen, der den Zustand des Objektes[9] restauriert. Ein Loader kann aber nur so realisiert werden, daß er die benötigten Objekte innerhalb einer eigenen Transaktion oder einer Server-Transaktion bereitstellt. Dadurch ist aber keine Verknüpfung des vom Loader bereitgestellten Objekt zur Client-Transaktion möglich. Dieser Fall ist aber nicht gewollt.
- Objekte werden über eine Objektfabrik aktiviert.

[8] `_narrow`-Methode in Orbix
[9] wenn der Loader so realisiert ist

Eine Objektfabrik besitzt eine analoge Aufgabe zum Loader bis auf den Unterschied, daß hier die Initiative direkt vom Client ausgeht und nicht automatisch durch den BOA erfolgt. Dieser Mechanismus wird in dieser Arbeit unterstützt und erweitert. Die Idee gleicht dem virtuellen Konstruktor in [GHJV94].

Daraus ergibt sich zunächst, daß eine Fabrik bestimmte persistente Objekte bereitstellt. Welches Objekt bereitgestellt wird, muß über einen Anfragemechanismus[10] an die Datenbank realisiert werden. Das entsprechende persistente Objekt wird aus der Datenbank geholt. Damit ist aber automatisch das Setzen einer Sperre verbunden. Um darauf Einfluß zu nehmen, wird zusätzlich die Spezifikation einer Sperre unterstützt, mit der das entsprechende Objekt aus der Datenbank geholt wird.

Nur auf einen existierenden Datenbestand zuzugreifen, ist in den meisten Fällen zu einschränkend. Aus diesem Grund wird dem Nutzer einer Fabrik die Möglichkeit offen gelassen, neue Objekte zu erzeugen und auch wieder zu entfernen. In ODMG wird eine automatische Unterstützung eines Extents gefordert. Mit dem Extent einer Klasse sind alle Instanzen einer Klasse in der Datenbank gemeint. Die Bereitstellung aller Objekte einer Klasse wird mit der letzten Methode unterstützt.

```
interface pers_CORBA_factory {
 pers_CORBA create_pers_CORBA();
 pers_CORBA get_pers_CORBA(in string pred, in lockmodus sperre);
 void delete_pers_CORBA(in pers_CORBA del_Object);
 set_of_pers_CORBA get_extent();
};
```

Die Fabrik für persistente CORBA-Objekte steht mit einer Transaktion in Beziehung. Innerhalb einer Transaktion kann eine Anzahl von Datenbanken unterstützt werden. Dieser Sachverhalt wird in diesem Interface nicht berücksichtigt. Deswegen ist es nicht eindeutig, in welcher Datenbank ein persistentes Implementierungsobjekt erzeugt[11] wird. Eine einfache Realisierung ist die Übergabe des Datenbanknamens als String. Sie besitzt den Nachteil, daß die Verbindung zu dieser Datenbank bezüglich der umgebenden Transaktion überhaupt nicht existieren kann. In ODMG wird für eine Datenbankverbindung eine separate Klasse verwendet. Analog läßt sich ein IDL-Interface entwickeln, welches die Verbindung zu einer Datenbank gewährleistet. In diesem Ansatz erhalten die Methoden der Fabrik dieses Datenbank-Interface als zusätzlichen Parameter.

Analog zu den konkreten Listen existiert für jedes konkrete Interface, welches von dem `pers_CORBA`-Interface abgeleitet wurde, wiederum eine konkrete Factory. Als Beispiel ist wieder die Factory zu der Schnittstelle K angegeben.

```
interface K_factory:pers_CORBA_factory {
};
```

Diese konkrete Factory besitzt ein verändertes Verhalten bezüglich der Methodenimplementierung. Der Client-Nutzer der Methoden einer Instanz einer konkreten Factory erzeugt sich über das CORBA-Abwärts-Casting den konkreten Typ.

[10] Zeichenkettenübergabe eines Prädikats

[11] entsprechendes gilt für die anderen Funktionen

3.2.3 Die allgemeine Transaktionsschnittstelle

Für die Unterstützung transaktionaler Konzepte wurden verschiedene Transaktionstypen bereitgestellt. In Abbildung 3.3 sind diese unterstützten Transaktionstypen und ihre Beziehungen untereinander dargestellt. Es wurde eine Vererbungshierarchie entwickelt, um eine transparente Client-Nutzung der Transaktionstypen zu gewährleisten.

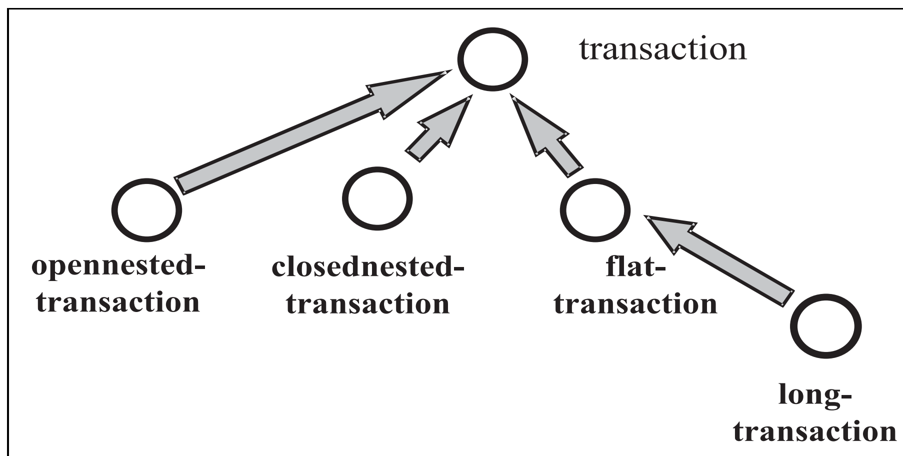

Abbildung 3.3: Die Transaktionshierarchie

Jede Datenbankoperation muß an eine Transaktion[12] gebunden sein. Das gleiche gilt für die Unterstützung persistenter Objekte innerhalb einer CORBA-Umgebung. Zum einen kann dies transparent geschehen, indem jeder Methodenaufruf eine separate Transaktion darstellt. Zum anderen soll der Client mehrere Operationen zu einer Transaktion zusammenfassen dürfen. Der zweite Fall wird mittels der folgenden Schnittstellendefinition realisiert.

```
interface transaction {
 void commit();
 void abort();
 void set_db(in string db_name);
 pers_CORBA_factory get_factory(in string which);
 query get_query_object();
};
```

Jedes Datenbanksystem besitzt ein Minimum an Transaktionunterstützung, was sich im allgemeinen Transaktion-Interface widerspiegelt. Dazu zählen die Operationen `commit` und `abort`. Um diese Schnittstelle möglichst klein zu halten, wird definiert, daß eine Transaktion automatisch mit der Erzeugung eines solchen Objektes beginnt. Auch das Ende einer Transaktion kann automatisch über den Destruktor erfolgen und ist als Abbruch definiert. Für die beiden Transaktionsoperationen wird festgelegt, daß nach ihrem Aufruf die laufende Transaktion beendet und eine neue Transaktion gestartet wird.

Im Fall der `commit`-Transaktionsoperation müssen zwei Fälle unterschieden werden:

[12] siehe ODMG-Standard

- Das Datenbanksystem konnte das Commit ausführen. In diesem Fall sind keine Maßnahmen zu treffen.
- Das Datenbanksystem konnte das Commit nicht ausführen und hat die Transaktion zurückgesetzt. In diesem Fall ist eine Ausnahmebehandlung erforderlich. Dies kann über das *Exceptionhandling* von CORBA oder durch Rückgabeparameter erfolgen.

Ein Datenbanksystem verwaltet mitunter eine Anzahl von Datenbanken. Aus diesem Grund muß spezifiziert werden, mit welcher Datenbank gearbeitet wird. Darum wird innerhalb einer Transaktion der Verbindungsaufbau zu einer Datenbank unterstützt.

Die Einordnung von Fabrik- und Anfragebereitstellung in eine Transaktion erlaubt die Zuordnung von Objekten bzw. Methodenaufrufen zu einer bestimmten Transaktion. Denn der Zugriff auf Objekte der Datenbank erfolgt innerhalb einer Transaktion entweder auf den Objekten, die uns eine konkrete Fabrik bzw. das Ergebnis einer Anfrage bereitstellt, oder auf den Objekten, die von den ersteren als das Ergebnis von Methodenaufrufen oder aufgrund der Traversierung von Beziehungspfaden geliefert wurden. Welche Fabrik man zurückgeliefert bekommt, wird mittels der Übergabe einer Zeichenkettenrepräsentation bestimmt.

Durch die Übergabe einer Zeichenkette für den Typ in der Fabrikmethode sind bestehende CORBA-Anwendungen mit einem persistenten Objektzugriff unabhängig gegenüber Erweiterungen bezüglich neuer Datenbanken und damit verbundenen Klassenerweiterungen. Dies schließt die Neuübersetzung entsprechender CORBA-Anwendungen ein.

3.2.3.1 Die erweiterte Transaktionsschnittstelle

Für eine gute Handhabung von Transaktionen ist die unterstützte Funktionalität nicht ausreichend. Die Operationen `commit` und `abort` geben automatisch alle gehaltenen Sperren frei und invalidieren damit alle Datenbankobjekte der Transaktion. Möchte der Nutzer mit der Arbeit fortfahren, muß er sich die entsprechenden Objekte von neuem holen. Aus diesem Grund werden die allgemeinen Transaktionen, siehe Interface-Beschreibung, erweitert. Eine Operation, die auch die Objektzustände persistent macht[13], aber die Sperren nicht freigibt, ist die `checkpoint`-Operation. Es ist eine Ausnahmebehandlung analog zum Commit erforderlich. Die entsprechende Operation für `abort` ist im Savepoint-Konzept enthalten.

```
interface flattransaction:transaction {
 long savepoint();
 void undosavepoint(in long savepoint_id);
 void checkpoint();
 void connect_db(in string db_name);
 void disconnect_db(in string db_name);
};
```

Das Savepoint-Konzept erlaubt es, auf einen Zustand innerhalb der Transaktion zurückzusetzen, der durch die `savepoint`-Operation definiert wurde. Jeder Sicherungspunkt erhält

[13] beendet die laufende Transaktion

einen Identifikator, der für das Rücksetzen[14] der Transaktion auf einen bestimmten Sicherungspunkt verwendet wird. Dieses Konzept kann benutzt werden, um einen Abbruch der gesamten Transaktion zu unterstützen, der aber nicht gleichzeitig die Sperren freigibt. Dazu wird der Zustand zu Beginn einer Transaktion als Savepoint definiert.

Im allgemeinen Transaktions-Interface kann immer nur mit einer Datenbank gearbeitet werden. Verteilt man seine Daten auf mehrere Datenbanken, müssen aber auch Verbindungen zu diesen aufgebaut werden. Aus diesem Grund wird die gleichzeitige Nutzung mehrerer Datenbanken durch den entsprechenden Aufbau und Abbau von Verbindungen unterstützt.

3.2.3.2 Die Unterstützung geschachtelter Transaktionen

In Kapitel 2.3.3 wurde das Konzept der geschachtelten Transaktionen eingeführt. Es wurde dabei unterschieden zwischen offen und geschlossen geschachtelten Transaktionen. In beiden Fällen muß es einer Transaktion erlaubt werden, mehrere Sub-Transaktionen zu erzeugen. Dies spiegelt sich auch in der entsprechenden Abbildung wider. Zunächst besitzt eine geschachtelte Transaktion die gleichen Eigenschaften[15] wie eine allgemeine Transaktion. Als Beispiel ist die Schnittstelle für geschlossen geschachtelte Transaktionen[16] angegeben.

```
interface opennestedtransaction:transaction {
 opennestedtransaction new_opennestedtransaction();
};
```

Die ererbten Transaktionsoperationen werden auf das geschachtelte Transaktionskonzept angepaßt. Sie bedeuten wieder die Beendigung einer Sub-Transaktion bzw. der Wurzeltransaktion, sind aber an die Bedingung geknüpft, daß alle ihre Sub-Transaktionen beendet sein müssen. Deshalb sind diese Operationen, um eine entsprechende Ausnahmebehandlung zu erweitern.

Offen und geschlossen geschachtelte Transaktionen unterscheiden sich in der Semantik der Endeoperationen. Das heißt, geschlossen geschachtelte Transaktionen vererben ihre Sperren an die Vatertransaktion, wogegen offen geschachtelte ihre Sperren freigeben.

Ein Transaktions-Server für geschachtelte Transaktionen verwaltet mehrere Transaktionen (Sub-) pro CORBA-Server. Im Client wird die Zuordnung vom Objektmethodenaufruf zu einer Transaktion durch die Aufrufschachtelung erreicht. Diese Art der Zuordnung geht durch die CORBA-Kommunikationsmechanismen verloren. Dadurch müßten aber persistente CORBA-Objekte Transaktionssemantik erhalten, was nicht erwünscht ist. Daraus ergibt die Einordnung der Fabrik- und Anfragebereitstellung in eine Transaktion den Sinn, daß wiederum eine eindeutige Abbildung von einem Objekt zu einer geschachtelten Transaktion erreicht wird.

[14] durch Aufruf der `undosavepoint`-Operation
[15] durch Vererbung erreicht
[16] der offene Fall sieht äquivalent aus

3.2.3.3 Lange Transaktionsunterstützung

In langen Transaktionen wird ein allgemeines Workspace-Konzept, siehe Kapitel 2.3.5, realisiert. Zu diesem Zweck werden öffentliche und private Workspaces[17] (Datenbanken) unterstützt. Die Transferierung von Objekten übernehmen die bekannten Operationen `checkout`[18] und `checkin`, siehe Schnittstellendefinition.

```
interface longtransaction:flattransaction {
 void create_public_workspace(in string name);
 void create_private_workspace(in string name);
 void delete_workspace(in string name);
 void set_workspace(in string name);
 list_of_pers_CORBA checkout(in list_of_pers_CORBA co, in string db_name);
 void checkin(in list_of_pers_CORBA back, in string db_name);
 void set_long_tra_name(in string long_tra_name);
 void commit_n();
 void abort_n();
}
```

Die flachen Transaktionen unterstützen den Verbindungsaufbau zu bestimmten (öffentlichen) Datenbanken. Aus diesen können nun Objekte in den gesetzten Workspace (Datenbank) transferiert werden.

Im Gegensatz zu kurzen Transaktionen kann nach Beendigung einer langen Transaktion die Arbeit fortgesetzt werden. Dies erfolgt über das Setzen eines langen Transaktionsnamens. Diese lange Transaktion muß aber explizit[19] beendet worden sein. Wird die lange Transaktion dagegen mit einer der ererbten Endeoperationen beendet, werden alle ausgecheckten Objekte an ihren Herkunfts-Workspace zurücktransferiert.

3.2.4 Vorteile und Restriktionen des Lösungsansatzes

In dem Lösungsansatz wurde sich streng an eine Abstraktionsstufe gehalten. Diese entspricht der Client-Sicht für die Nutzung erweiterter Datenbankfunktionalität. Daraus ergeben sich zunächst neben den unterstützten Konzepten einige Vorteile.

- Datenbanksystemunabhängigkeit wurde realisiert, indem nur Konzepte abgebildet wurden.
- Logische Datenunabhängigkeit ist durch die Verwendung von Fabriken erreicht.
- Transparente Nutzung unterschiedlicher Transaktionstypen wird durch die Vererbungshierarchie unterstützt. Z.B. kann einem Client eine offen geschachtelte Sub-Transaktionreferenz übergeben werden, ohne etwas über Spezifika dieser zu wissen.

Leider sind einige Restriktionen festzustellen. Diese liegen aber zum Teil auch an dem gewählten Abstraktionsniveau.

[17] Gruppen-Workspaces nur durch Zuordnung von öffentlichen zu Gruppen auf der Anwendungsebene
[18] ausgenommen von privaten Workspaces
[19] über `commit_n` oder `abort_n`

- Mit diesem Konzept wird zwar eine Client-Transparenz bezüglich eines bestimmten Datenbank Management Systems gewährleistet. Das heißt aber auch, daß bestimmte Schnittstellen mit einigen nicht realisierbar sind, weil sie diese Funktionalität nicht besitzen.

- Spezielle Mechanismen von Datenbanksystemen werden nicht berücksichtigt. Eine Erweiterung um diese kann erfolgen, indem eigene Transaktionsklassen umgesetzt und in die Hierarchie eingliedert werden. Dabei sollte aber aus Gründen der Client-Transparenz berücksichtigt werden, daß die Grundoperationen zumindest ein äquivalentes Verhalten besitzen.

- Datenmodellunabhängikeit und Datenbankentwurfsunabhängigkeit kann nur in der Realisierung der Schnittstellen erreicht werden.

- Die Unterstützung von CORBA-Objektreferenzen in Datenbanksystemen wird auf dieser Ebene nicht betrachtet.

In dem folgenden Realisierungskapitel werden einige dieser Nachteile noch eingehender beleuchtet.

Kapitel 4

Realisierung und Anwendung

In Kapitel 3 wurde mit der Definition der IDL-Schnittstellen die Grundlage für die Nutzung von Datenbankkonzepten in einer CORBA-basierten verteilten Anwendung geschaffen. Um die Arbeit mit beliebigen Datenbankklassen zu unterstützen, wird in den folgenden Kapiteln erläutert, welche zusätzlichen Klassen mit entsprechender Implementierung generiert bzw. wie vorhandene Klassen angepaßt werden.

4.1 Die Realisierung der vorgestellten Schnittstellen

Für ein besseres Verständnis der in den folgenden Kapiteln gemachten Aussagen, ist es ratsam, sich die im Anhang angegebenen Auszüge der Implementierung durchzulesen.

Die Nutzung von Datenbankobjekten innerhalb einer C++–Anwendung basiert auf den Schnittstellen, die uns das entsprechende Datenbanksystem zur Verfügung stellt. Eine standardisierte Schnittstelle ist das C++–Binding der ODMG, welche nur Basisfunktionalität unterstützt. Deshalb wurde in der Realisierung auf die C++–Schnittstelle von Versant zurückgegriffen.

4.1.1 Die Datenbankservergenerierung

Jede CORBA-Implemetierung stellt für die Registrierung von neuen Servern und den darin enthaltenen Objekten Kommandozeilenbefehle bereit. Diese enthalten als Parameter die Aktivierungsdatei mit vollem Pfadnamen, Aufrufrechte und Art der Aktivierung (`shared, unshared, per-method, per-object`). Diese Informationen werden in das Implementationrepository, dessen Benutzung in einer der nächsten Revisionen von CORBA standardisiert wird, eingetragen. Dieses Implementationrepository verwaltet alle registrierten Server.

In der CORBA-Implementierung Orbix entspricht die Handhabung des Implementationrepositorys dem anderer registrierter Server. Der Unterschied besteht nur darin, daß der Server für das Implementierungsverzeichnis direkt in den Objektadapter, vergleiche mit Abbildung 1.3, eingegliedert ist. Die Funktionalität des bereitgestellten Server-Objektes wird für die dynamische Generierung von Einträgen in das Implementationrepository verwendet.

Es wird vorrausgesetzt, daß die entsprechenden Server mit dem jeweils bereitgestellten Objekt für einen konkreten Transaktionstyp realisiert[1] sind.

Dieser Server realisiert die Registrierung und das Löschen von Transaktionsservern und gibt die entsprechenden Transaktionsobjektreferenzen zurück. Für die Registrierung eines neuen Servers wird zuerst ein eindeutiger Server-Name erzeugt. Unter dem generierten Namen wird mit entsprechendem Objektmarker und Aufrufkommando der neue Transaktions-Server registriert und Orbix-spezifische Rechte auf dem registrierten Server vergeben.

Durch diesen Mechanismus wird erreicht, daß jeder CORBA-Client einen eigenen Datenbank-Client besitzt. Dadurch kommt es nicht zu der ungewollten Nutzung der selben Transaktion durch zwei unterschiedliche Clients.

4.1.2 Der direkte Persistenzansatz

In dieser Arbeit werden zwei Ansätze für die Bereitstellung von persistenten Objekten vorgestellt. Beide besitzen die gemeinsame Eigenschaft, zwischen ORB-spezifischen Daten und dem eigentlichen Datenbankobjekt genau zu trennen. Der erste zu beschreibende Ansatz wurde innerhalb des Anwendungsszenarios realisiert und entspricht im wesentlichen dem Grundansatz aus [Neuw97]. Er wird in dieser Arbeit als direkter Persistenzansatz bezeichnet, siehe Abbildung 4.1.

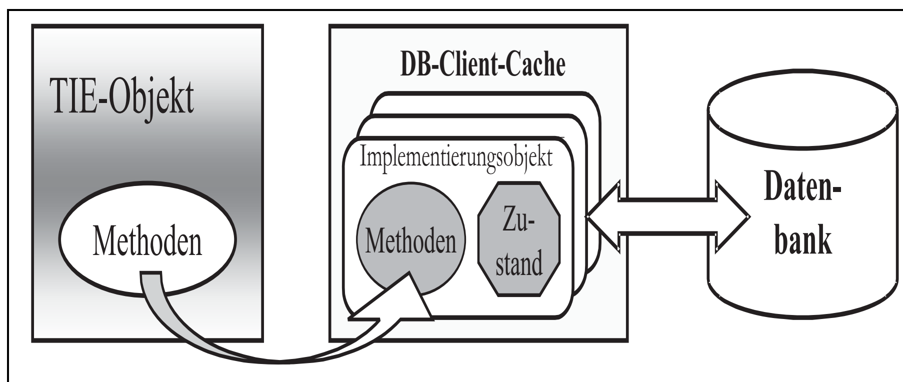

Abbildung 4.1: Persistenz der Implementierungsobjekte

Jede Klasse, die in ein Datenbankschema aufgenommen wird, muß von einer Basisklasse[2] direkt oder indirekt erben. Um eine Kopplung von CORBA-Objekten mit Datenbankobjekten zu erreichen, erbt die Implementierungsklasse vom `pers_CORBA`-Interface von dieser Datenbankklasse und bildet gleichzeitig die Basisklasse aller konkreten persistenten Implementierungsklassen. Deshalb muß diese Basisklasse in alle Datenbankschemas eingefügt werden. Jede Datenbankklasse, die im CORBA-Server zugreifbar gemacht werden soll, erbt daraufhin direkt oder indirekt[3] von dieser CORBA-Basisimplementierungsklasse. Dieser Zusammenhang wird in Abbildung 4.2 noch einmal verdeutlicht.

[1] die Aktivierungsdatei liegt vollständig implementiert und übersetzt vor
[2] PVirtual in Versant
[3] alle Basisklassen - Mehrfachvererbung wird nicht berücksichtigt

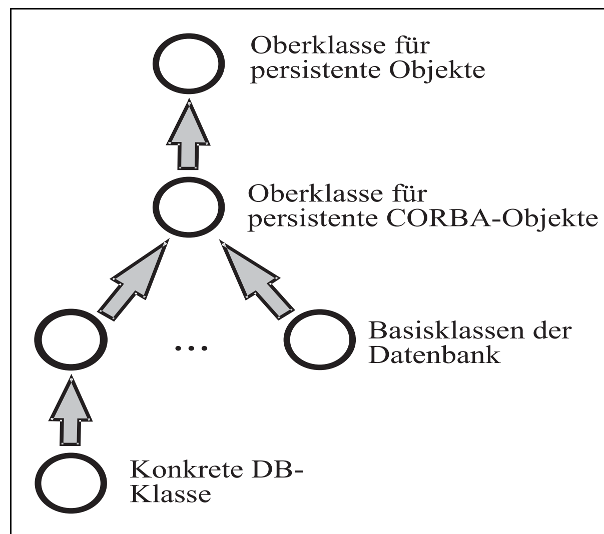

Abbildung 4.2: Hierarchie für persistente Implementierungsobjekte

In der Implementierung der Basisklasse wird zwischen zwei Methodenarten unterschieden[4].

- Methoden besitzen eine konkrete Implementierung. Zu diesen gehören die Methoden, die keinen konkreten Unterklassentyp brauchen, z.B. die Verschärfung und Entschärfung von Sperren.
- Methoden besitzen keine konkrete Implementierung[5]. Diese Methoden dürfen keine virtuellen Funktionen[6] sein, weil Instanzen der Basisklasse existieren. Sie müssen aber von der konkreten Datenbankklasse überschrieben werden, weil sie einen konkreten Typ zurückliefern müssen. Die Erzeugung einer neuen Version von sich entspricht einer solchen Methode. Diese Methoden besitzen in allen zu unterstützenden Klassen eine analoge Implementierung mit dem Unterschied, daß die entsprechenden konkreten Klassentypen eingesetzt werden. Deshalb ist eine automatische Klassenschemaänderung bzw. Implementierungsangabe für die Erweiterung um diese Methoden möglich.

Einige Datenbanksysteme besitzen für den Zweck der Initialisierung von transienten Daten die Unterstützung einer speziellen Methode[7]. Das muß in der Basisimplementierungsklasse berücksichtigt werden, indem diese Methode auch als überschreibbare Funktion bereitgestellt wird.

4.1.3 Kollektionen und Referenzen

Die Realisierung von konkreten Kollektionen ist äquivalent zu dem gemachten Ansatz in [Neuw97]. Der Unterschied besteht darin, daß eine globale Implementierungsklasse für

[4]zu erkennen an `virtual` siehe Anhang A

[5]Diese besitzen dann eine Lehrimplementierung

[6]Es existiert keine Implementierung

[7]In Versant heißt diese `init`

jeden Kollektionstyp[8] existiert, von der alle konkreten Kollektionen erben, siehe Abbildung 4.3. Dadurch ist die Verwendung des CORBA-Any-Typs bei der Unterstützung von Anfragen nicht notwendig, weil jedes Ergebnis automatisch vom Oberkollektionstyp ist. Dafür ist es aber erforderlich, das besagte Oberklassen nicht als virtuelle Klassen realisiert sind, da sie als Ergebnistyp verwendet werden.

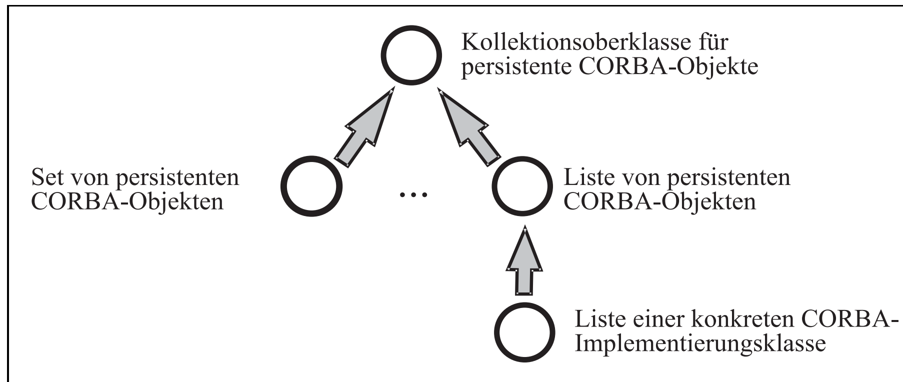

Abbildung 4.3: Hierarchie für Kollektionsrealisierung

Eine Erweiterung ergibt sich für die Unterstützung von Anfragen und des Extents in der Fabrikrealisierung. Als Ergebnis einer Anfrage erhält man eine Datenbankkollektion. Dadurch bedingt, muß eine konkrete CORBA-Kollektionsrealisierung einen zusätzlichen Konstruktor bereitstellen. Mit diesem ist es möglich, eine CORBA-Kollektion mit einer Datenbankkollektion zu initialisieren.

Datenbanksysteme besitzen zur Verwaltung von Kollektionen effektive Mechanismen, welche in der Realisierung durch die direkte Weitergabe der Methoden verwendet werden. Datenbankreferenzen werden äquivalent realisiert. In dieser Art der Abbildung wird aber immer nur innerhalb einer Transaktion gearbeitet.

Um innerhalb eines persistenten Implementierungsobjektes CORBA-Referenzen[9] zu speichern, wurde für jedes Interface eine eigene persistente Referenzklasse erzeugt. Diese kann als Member einer persistenten Implementierungsklasse verwendet werden. Auf diese Weise können beliebige CORBA-Referenzen in einem persistenten Implementierungsobjekt gespeichert werden. Diese Klasse spiegelt somit eine Beziehung von persistenten Objekten zu transienten oder wiederum persistenten CORBA-Objekten[10] wider. Die Datenbankklasse besitzt zwei wesentliche Funktionen, welche die Umwandlung aus oder in einen String realisieren. Nur auf diese Weise kann eine CORBA-Referenz in der Datenbank gespeichert werden, weil automatisch zum Zeitpunkt des Commits oder Rollbacks alle transienten Objektreferenzen invalidieren.

Der Fall, daß als Zeichenkette abgespeicherte CORBA-Referenzen nicht mehr gültig sind, kann natürlich auch eintreten. Aus diesem Grund wird, wenn ein Referenzobjekt aus der Datenbank geholt wird, überprüft, ob die gespeicherte CORBA-Referenz noch gültig ist. Ist die Referenz nicht mehr gültig, wird jeweils ein leeres CORBA-Objekt zurückgegeben.

[8] Darin ist auch eine allgemeine Kollektion eingeschlossen
[9] oder auch Beziehungen
[10] innerhalb eines anderen Servers und Transaktion

Das heißt, das entsprechende Datenbankobjekt hat dafür zu sorgen, daß die Referenz neu gesetzt wird. Befindet sich sich das betreffende persistente CORBA-Referenzobjekt im Cache, wird allerdings keine Prüfung mehr vorgenommen. Der Nutzer muß dann einen Test der CORBA-Referenz durchführen.

Damit ist aber keine transparente Lösung für die Speicherung von CORBA-Referenzen in einem persistenten Objekt realisiert. Mit den Konzepten dieser Arbeit ist dies auch nicht möglich. Nehmen wir einmal an, es existiert eine eindeutige Zuordnung eines CORBA-Objektes zu einem Datenbankobjekt. Diese Zuordnung sagt aber nicht aus, mit welcher Transaktion das entsprechende persistente CORBA-Objekt in Beziehung steht. Diese Entscheidung trifft stets die Anwendung.

4.1.4 Die Fabrikrealisierung

Die Fabrik gewährleistet die Verwaltung von Objekten einer konkreten Klasse. Dazu erbt jede konkrete Fabrikimplementierungsklasse von einer nicht abstrakten Basisfabrikklasse, siehe Abbildung 4.4. Dies ist notwendig, um eine Client-Transparenz in Bezug auf Klassen zu erreichen. Dadurch ist es nicht erforderlich, in der Schnittstelle, welche die unterschiedlichen Fabriken bereitstellt, für jede konkrete Fabrik eine zusätzliche Funktion einzufügen.

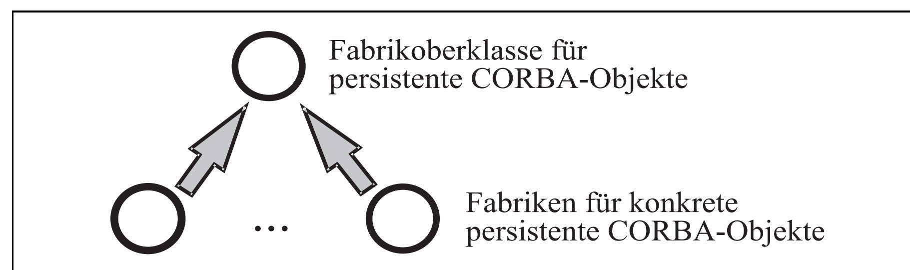

Abbildung 4.4: Hierarchie für die Fabrikrealisierung

Die Basisfabrik realisiert die Umwandlung der Anfragezeichenkette in ein Versant-Prädikat. Dieses Versant-Prädikat wird in allen konkreten Fabriken benötigt, um bestehende persistente Objekte zu erhalten.

Für die Erzeugung eines persistenten Objektes existieren verschiedene Möglichkeiten. In ODMG wird für diesen Zweck der `new`-Operator mit den optionalen Parametern der Datenbank und des Typs des zu erzeugenden Objektes überladen. In der Realisierung der Fabrikmethode zur Erzeugung eines CORBA-Objektes mit persistenten Implementierungsobjekt wird ein von Versant bereitgestelltes Makro verwendet. Es erzeugt das persistente Objekt automatisch in einer festgelegten Datenbank. Danach wird noch der entsprechende CORBA-TIE mit dem entweder erzeugten oder erhaltenen persistenten Implementierungsobjekt initialisiert und zurückgegeben. Das Löschen von persistenten Objekten erfolgt nach erfolgreichem Commit der umgebenden Transaktion durch direkte Weiterleitung an die vom Datenbanksystem bereitgestellte Funktionalität.

Versant besitzt keine automatische Verwaltung von Extents. Um trotzdem alle Datenbankobjekte einer Klasse zu erhalten, besitzt Versant Klassenobjekte, an die eine entsprechend

formulierte Anfrage gestellt wird. Die erhaltene Versant-Liste[11] wird für die Initialisierung der CORBA-Liste weiterverwendet.

4.1.5 Die Realisierung der verschiedenen Transaktionstypen

Im Kapitel 3.2.1 wurde die Schnittstelle eines Serverobjektes vorgestellt, welches neue Server nach Client-Aufruf registriert. Die Grundlage der in diesem neu registrierten Servern bereitgestellten Objekte bilden die Implementierungsrealisierungen in den folgenden Abschnitten.

4.1.5.1 Die Umsetzung allgemeiner und erweiterter Transaktionen

Die Realisierung dieser Implementierungsklasse erfolgt durch eine Abbildung auf das Konzept der kurzen Transaktionen in Versant. Im Gegensatz zum C++–Binding der ODMG existieren keine Transaktionsklassen. In Versant existiert pro Datenbank-Client nur jeweils eine aktive kurze Transaktion[12]. An diese werden die Transaktionsoperationen durchgereicht. Weil jeweils durch die Nutzung des Datenbankserver-Interfaces nur ein Transaktionsobjekt pro Server existiert, liegt darin keine Einschränkung.

Die Realisierung des Savepoint-Konzeptes muß in Versant auf die Unterstützung nur eines Sicherungspunktes eingeschränkt werden. Jeder Savepoint überschreibt automatisch den vor ihm gesetzten. Das Zurücksetzen der Transaktion erfolgt automatisch auf den gesetzten Sicherungspunkt oder auf den Zustand, der zu Beginn der Transaktion existierte.

Der Verbindungsaufbau zu verschiedenen Datenbanken wird an die versantspezifischen Methoden durchgereicht. Eine verbundene Datenbank hat die besondere Eigenschaft, daß in dieser automatisch Objekte erzeugt bzw. auf diese Anfragen gestellt werden.

Weiterhin werden Anfragen und Fabriken realisiert. Die Methode zum Erhalt einer konkreten Fabrik[13] testet den Übergabeparameter, der die konkrete Fabrik spezifiziert, erzeugt sie und gibt diese zurück. Der Vorteil dieser Realisierung ist, daß der Client unabhängig von Klassen[14] ist. Für eine Erweiterung um eine Klasse wird die Methodenimplementierung so verändert, daß auch eine Fabrik dieser Klasse erzeugt werden kann.

4.1.5.2 Die Umsetzung geschachtelter Transaktionen

Einführend sei bemerkt, daß Versant keine saubere Unterstützung der Konzepte der offen und geschlossen geschachtelten Transaktion besitzt. Es existiert nur ein Konglomerat aus beiden Konzepten. Das Prinzip der Kompensationstransaktion wird überhaupt nicht unterstützt. Zunächst besitzt das geschachtelte Transaktionskonzept von Versant folgende Eigenschaften:

- Der CORBA-Serverprozeß mit bereitgestelltem Transaktionsobjekt stellt für Versant eine Session dar. In diesem Prozeß wird die erwähnte Session mit den Parametern

[11] Durch die Anfrage bedingt wird keine Menge als Extent zurückgeliefert.
[12] auch wenn ein Commit bzw. ein Abort aufgerufen wurde
[13] Dies wird über die Angabe einer Zeichenkette als Parameter erreicht.
[14] Hier ist im Sinn von Ergänzung um Klassen gemeint.

für geschachtelte Transaktionen begonnen. Im Fall geschachtelter Transaktionen befinden sich sowohl die transienten als auch persistenten Objekte im *shared Memory*, auf dem über Prozeßgrenzen zugegriffen werden kann.

- Jede geschachtelte Transaktion erhält ihren eigenen Prozeßbereich. Dieser wird von Versant verwaltet.
- Eine geschachtelte Transaktion darf erst beendet werden, wenn sie keine Kindtransaktionen mehr besitzt.
- Verbundene Datenbanken gelten für alle Schachtelungsstufen.

Das Verhalten einer Wurzeltransaktion unterscheidet sich von einer geschachtelten Kindtransaktion. Dies macht sich in den Transaktionsoperationen bemerkbar. Im Gegensatz zu einer Sub-Transaktion wird durch die Operationen Commit und Abort der Wurzeltransaktion die gesamte geschachtelte TA beendet. Dies zeigt sich zwar in der Implementierung, wirkt sich aber nicht auf den Client aus.

Die Realisierung für das Erzeugen einer CORBA-Sub-Transaktion benutzt zunächst den Mechanismus von Versant und erzeugt danach ein geschachteltes CORBA-Transaktionsobjekt innerhalb der Versant-Sub-Transaktion und gibt dieses zurück.

Das Commit und Abort einer geschlossen geschachtelten Transaktion wird auf die Versant-Endeoperationen einer geschachtelten Transaktion derart abgebildet, daß alle Sperren an die Vatertransaktion vererbt werden. Entsprechend entgegengesetzt werden offen geschachtelte Transaktionen abgebildet, siehe Anhang B.

4.1.5.3 Die Umsetzung langer Transaktionen

Die langen Transaktionen von Versant besitzen die Funktionalität, um ein Workspace-Konzept umzusetzen. Eine Einschränkung ergibt sich daraus, daß Objekte nur von öffentlichen Arbeitsbereichen in private Arbeitsbereiche ausgecheckt werden dürfen.

Zunächst müssen Workspaces verwaltet werden. Dazu zählt das Erzeugen und Löschen von Workspaces[15].

Für die Transferoperationen des Datenbanksystems werden Datenbanklisten benötigt. Aus diesem Grund muß die CORBA-Liste in der Implementierung in eine solche Liste umgewandelt werden. Dazu ist es notwendig, eine neue Datenbankliste zu erzeugen und die persistenten Implementierungsobjekte aller Objekte der CORBA-Liste[16] in diese einzufügen. Mit Hilfe dieser Liste erfolgt durch das Datenbanksystem die Transferoperation. Die erhaltene Datenbankliste der `checkout`-Operation wird danach zur Initialisierung der entsprechenden CORBA-Liste verwendet und zurückgegeben. Die Realisierung der Eincheck-Operation erfolgt analog. Die Weiterführung einer langen Transaktion erfolgt durch Setzen des langen Transaktionsnamens.

[15] Dies bedeutet das Anlegen und Löschen von privaten Datenbanken, denn öffentliche Datenbanken sind nicht sinnvoll aufgrund der Restriktion.

[16] Dies wird über die `_deref()`-Routine von Orbix nach Anwendung der `get_element()`-Funktion auf der CORBA-Liste realisiert.

Die Anwendung der `checkout`-Operation sollte nur auf versionierte Objekte[17] geschehen, weil im nicht versionierten Fall mit der `checkout`-Operation eine Schreibsperre der entsprechenden Objekte verbunden ist.

4.1.6 Der Wrapper-Ansatz

Der direkte Persistenzansatz besitzt einige Nachteile. Falls eine bestehende Datenbank für CORBA-Clients zugreifbar gemacht wird, ergeben sich entscheidende Auswirkungen auf vorhandene Anwendungen der Datenbank:

- **Schemaänderung**:
  Damit erweiterte Datenbankfunktionalität auch für einen CORBA-Client verfügbar ist, muß jede nutzerdefinierte Oberklasse in der Datenbank von der Implementierungsoberklasse für persistente Objekte abgeleitet werden. Zusätzlich muß diese Klasse in das Klassenschema der entsprechenden Datenbank eingefügt werden.

- **Implementierungsänderung**:
  Für die Unterstützung von Beziehungen zwischen Objekten müssen die Methoden für das Setzen bzw. Erhalten von Referenzen oder Kollektionen in der Art geändert werden, daß die Datenbankreferenzen in CORBA-Referenzen umgewandelt werden.

Dadurch ergibt sich, daß die Datenbankentwurfsunabhängikeit, siehe Kapitel 3.2, nicht gewährleistet ist.

Weiterhin ist der vorgestellte Ansatz nicht anwendbar, wenn das Datenbanksystem kein objektorientiertes Datenhaltungsmodell besitzt. Es existieren dann keine Objekte, an die man die Methodenaufrufe direkt weiterleiten kann. Ein Ansatz der ohne Auswirkungen auf vorhandene Anwendungen auskommt und auch für andere Datenhaltungsmodelle anzupassen ist, wird nachfolgend beschrieben.

In diesem Ansatz wird für jede persistente Klasse ein *Wrapper*, siehe Abbildung 4.5, generiert und zwischen dem CORBA-TIE und dem persistenten Implementierungsobjekt eingefügt. Der Wrapper selber ist nicht persistent, sondern leitet nur die CORBA-Aufrufe

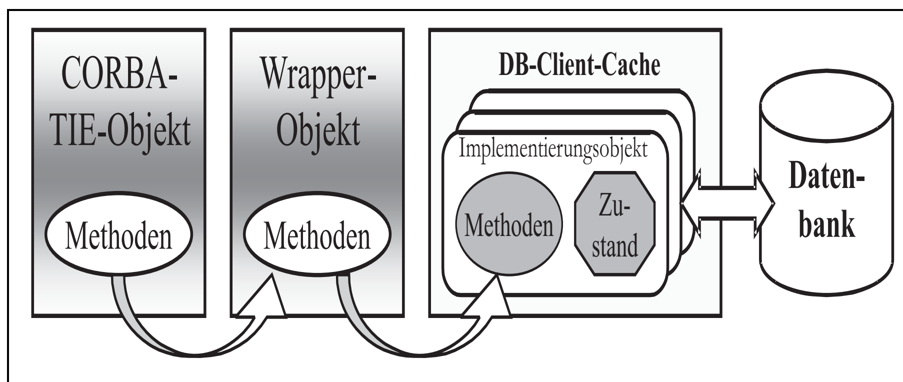

Abbildung 4.5: Der Wrapperansatz

[17] durch den Aufruf der `make_versioned()`-Methode auf dem Objekt

an das persistente Objekt weiter. Deswegen kann in diesem Ansatz für die Realisierung des Wrappers die Vererbungsmethode verwendet werden. Die Trennung vom persistenten Datenbankobjekt erfolgt automatisch durch den Wrapper. Die IDL-Interfaces entsprechen denen im direkten Persistenzfall.

Auch in diesem Ansatz existiert ein globaler Wrapper, der die Funktionsbeschreibung bzw. auch die Implementierung aller bereitzustellenden Datenbankobjekte beinhaltet. Für alle Datenbankklassen wird ein konkreter Wrapper, der vom globalen Wrapper erbt, generiert. Ein konkreter Wrapper bildet alle Methoden und Beziehungen der entsprechenden Datenbankklasse ab. Als privaten Daten-Member besitzt der spezielle Wrapper eine Referenz[18] auf das eigentliche Datenbankobjekt. Aus diesem Grund werden alle CORBA-Aufrufe an die Methoden des Datenbankobjektes durchgereicht. Dies gilt auch zunächst für die Methoden, welche z.B. Datenbankobjektreferenzen zurückliefern. Die erhaltene Datenbankreferenz oder auch Kollektion wird daraufhin für die Initialisierung des entsprechenden Wrappers bzw. eines Kollektion-Wrappers verwendet. Dieser entspricht gleichzeitig dem Rückgabeparameter[19].

Auch in diesem Ansatz wird erweiterte Funktionalität auf Objektebene realisiert. Sie wird im globalem Wrapper definiert, teilweise implementiert[20] oder in den konkreten Wrappern überschrieben. Die Realisierung erfolgt in diesem Ansatz über die Datenbankreferenz.

Für die Initialisierung mit einen Datenbankobjekt bzw. Referenz muß ein Wrapper einen entsprechenden Konstruktor unterstützen. Dieser Konstruktor eines konkreten Wrappers wird außerdem auch für die Unterstützung von Beziehungen, Kollektionen und der Fabrik verwendet, falls als Ergebnis ein Datenbankobjekt geliefert wurde.

Für die Kollektionsunterstützung existieren zum einen die Oberklassen für alle Kollektionsarten und zum anderen für jede konkrete Klasse die entsprechenden Kollektionen. Zu beachten ist, daß nicht mehr der CORBA-TIE eines persistenten Objektes, sondern der eines Wrappers um diesen, zurückgeliefert wird. Ansonsten entspricht die Realisierung dem persistenten Realisierungsansatz. Weitere Datenbankkonstrukte werden mit entsprechenden Anpassungen auf den Wrapper-Ansatz äquivalent zu [Neuw97] abgebildet.

Die Realisierung einer Fabrik erfolgt in analoger Weise zum direkten Persistenzansatz. Sie unterscheidet sich darin, daß hier jeweils Wrapper oder Wrapper-Kollektionen zurückgegeben werden.

4.2 Auswirkungen auf das Anwendungsszenario

Der erste Schritt für die Erweiterung des Anwendungsszenarios um Persistenzkonzepte liegt darin, eine Analyse durchzuführen, welche Objekte persistente Daten beinhalten. Im Anwendungsszenario existieren drei Server-Objekte: das Whiteboard-, das Wbgroup- und das Shapefactory-Objekt. Für das Whiteboard ist die Bereitstellung von Persistenzmechanismen, aufgrund vollständig transienter Daten, nicht notwendig. Das Gruppenobjekt besitzt dagegen einerseits transiente Daten. Diese Daten entsprechen einer Liste von Whiteboards, die in bestimmten Situationen zu informieren sind. Andererseits verfügt es auch

[18] in Versant `Link`

[19] falls der BOA-Approach verwendet wird

[20] Dazu zählt auch eine Leerimplementierung.

über persistente Eigenschaften in Form von Konferenzdaten, wie zum Beispiel den Namen der Konferenz bzw. des Erzeugers und die Datenbankreferenz auf die Shapefactory. Die Shapefactory spiegelt in Form einer Liste von Shapes den aktuellen Konferenzzustand wieder. Das heißt, sowohl Whiteboard-Gruppenobjekt als auch Shapefactory mit den dazugehörenden Shapes müssen um Persistenzmechanismen erweitert werden. Dies erfolgt in den nächsten Schritten.

- Zunächst müssen die ausgewählten Objekte persistent gemacht werden. Dazu erben die IDL-Interfaces bzw. die entsprechenden Implementierungsklassen von den vorgestellten Schnittstellen und Klassen. Die Implementierungsklassen müssen an die Nutzung von Persistenz angepaßt werden. Zusätzlich werden für jede konkrete Klasse die Kollektionen und die Fabriken benötigt.

- In diesem Szenario wurde eine bestehende Anwendung um Persistenzkonzepte erweitert. Dadurch ergeben sich Änderungen innerhalb von Schnittstellen. Die IDL-Interfaces sind im Anhang C nachzulesen.

  Die Shapefactory des Wbgroup-Objektes wird im persistenten Fall innerhalb einer Transaktion bearbeitet. Deswegen muß die Wbgroup-Schnittstelle um Operationen ergänzt werden, die den Whiteboards den Zugriff auf die umgebende Transaktion der Shapefactory erlauben.

  In der nicht persistenten Realisierung existierte jeweils nur eine Konferenz. Damit ist die Zuordnung vom Gruppenobjekt zum Shapefactory-Objekt gegeben. In der Erweiterung des Anwendungsszenarios wird eine Zuordnung durch die entsprechende Datenbankbeziehung realisiert, die durch eine entsprechende Methode im Wbgroup-Interface unterstützt wird. Über diese Methode muß ein Whiteboard sich die entsprechende Shapefactory holen.

- Eine Transaktionskontrolle[21] wurde für den Server, der die Gruppenobjekte bereitstellt, nicht realisiert. Außer dem Erzeugen, dem damit verbundenen Initialisierungsvorgang und dem Löschen werden keine Änderungen am persistenten Teil eines Gruppenobjektes vorgenommen. Aus diesem Grund wird im dazugehörenden Server nur das Fabrikobjekt zum Wbgroup-Interface, siehe auch Kapitel 3.2.2.3, erzeugt. Es wird trotzdem transparent innerhalb einer Transaktion gearbeitet, welche nach dem Herunterfahren des Servers automatisch beendet wird.

- Für die Bearbeitung einer Shapefactory ist eine Transaktionskontrolle, z.B. wenn Änderungen zurückgesetzt werden sollen, sinnvoll. In diesem Fall sind verschiedene Kooperationsstufen denkbar:

  - **volle Kooperation**: Das Wbgroup-Objekt erzeugt sich über den Server aus Kapitel 3.2.1 einen neuen Datenbankserver mit bereitgestelltem flachen Transaktionsobjekt. Jedes Whiteboard erhält das selbe Transaktionsobjekt in dessen Kontext er seine Objekte erzeugt, verändert und löscht. In diesem Fall sind die Rechte für Transaktionsoperationen unter den Whiteboard-Nutzern abzustimmen.

[21] bedeutet die bewußte Bearbeitung innerhalb einer Transaktion

  - **geschachtelte Kooperation**: Das Wbgroup-Objekt erzeugt sich über den Server aus Kapitel 3.2.1 einen neuen Datenbankserver mit bereitgestelltem geschachtelten Transaktionsobjekt. Für jedes Whiteboard, das sich anmeldet, wird eine eigene geschachtelte Transaktion gestartet. In diesem Fall muß das Gruppenobjekt zusätzliche Funktionalität erhalten, um die verschiedenen Whiteboards wieder zu synchronisieren, wenn alle ihre geschachtelte Transaktion beendet haben.
  - **Gruppenkooperation**: Mitglieder einer Gruppe erhalten das selbe geschachtelte Transaktionsobjekt, und Mitglieder unterschiedlicher Gruppen bekommen dagegen unterschiedliche geschachtelte Transaktionsobjekte.
  - **keine Kooperation**: Jedes Whiteboard erhält ein eigenes flaches Transaktionsobjekt innerhalb eines separaten Servers.

  Das Gruppenobjekt entscheidet über den Kooperationsgrad, welcher Transaktionstyp verwendet wird. Die nächsten Abschnitte geben Realisierungsmöglichkeiten und die entsprechenden Kooperationseinschränkungen an.

- Es wird generell davon ausgegangen, daß die Shapefactory nicht in der Transaktion bearbeitet wird, in der sich die Wbgroup-Objekte befinden. Dies entspricht einer separaten Transaktion für die Bearbeitung einer Shapefactory. Daraus ergibt sich ein Problem für die Beziehung vom Gruppenobjekt zum Shapefactory-Objekt, da Datenbankreferenzen nicht übertragbar sind[22]. Teilweise gelöst wird dieses Problem, indem die Klasse zur Speicherung von CORBA-Referenzen in Datenbankobjekten aus Kapitel 4.1.3 verwendet wird. Für das Setzen der abgespeicherten Referenz ist das Gruppenobjekt zuständig.

Durch die Erweiterung des Anwendungsszenarios um Persistenzkonzepte, existiert die Möglichkeit, eine Vielzahl von Konferenzen zu unterstützen. Ein Whiteboard bindet sich zunächst an ein Gruppenfabrikobjekt. Durch die Nutzung der von diesem Objekt bereitgestellten Schnittstelle kann eine bestehende Konferenz zum einen über den Extent der Wbgroup-Klasse oder zum anderen direkt über eine Fabrikmethode geladen werden. Weiterhin kann aber auch eine neue Konferenz erzeugt werden.

4.2.1 Unterstützung der vollen Kooperation

Jede Konferenz bearbeitet ihre Shapefactory in einer separaten kurzen Transaktion. Aus diesem Grund existiert auch ein separater Server für diese Transaktion. Zu unterscheiden sind zwei Fälle.

- Die Konferenz, zunächst das Wbgroup-Objekt, wird über die Konferenzfabrik neu erzeugt. In diesem Fall muß eine Shapefactory in einer eigenständigen Transaktion erzeugt werden. Dies erfolgt durch die Nutzung des in Kapitel 3.2.1 vorgestellten Servers. Das erhaltene Transaktionsobjekt des neu registrierten Servers wird weiterverwendet, um sich eine neue Shapefactory zu erzeugen.

[22] Für diesen Zweck muß eine Datenbankreferenz in einen String umgewandelt werden.

- Eine bestehende Konferenz wird über die Gruppenfabrik geladen. Damit ist die CORBA-Referenz auf die Shapefactory ungültig und muß neu gesetzt werden. Das Setzen dieser Referenz erfolgt in der Initialisierungsfunktion, siehe Kapitel 4.1.2. Zu diesem Zweck wird von dem Server, der die Transaktionen verwaltet, eine neue Transaktion angefordert. Über die Fabrikfunktion wird letztendlich die richtige Shapefactory aus der Datenbank geholt.

In beiden Fällen wird die CORBA-Referenz der Shapefactory umgewandelt, so daß die Datenbank diese auch speichern kann, siehe Kapitel 4.1.3.

Diese Art der Realisierung hat im Vergleich zum nicht persistenten Fall keine Einschränkungen zur Folge. Für die Transaktionsunterstützung auf der Whiteboard-Ebene werden Transaktionsoperationen in der Wbgroup-Schnittstelle eingeführt. Einen Sinn machen aber nur die Operationen `checkpoint`, `savepoint` und `undosavepoint`. Die Operationen `commit` und `rollback` sind mit dem Seiteneffekt verbunden, den Objekt-Cache zu invalidieren. Diese Operationen dürfen nur vom Wbgroup-Objekt aufgerufen werden, wenn alle Whiteboards sich abgemeldet haben.

Außer den verbundenen Whiteboards existieren für eine Konferenz mit der vollen Kooperationsunterstützung drei Server. Der Server für die Erzeugung eines neuen Servers mit darin bereitgestelltem Transaktionsobjekt existiert dabei nur für diesen Zeitraum. Die zwei weiteren Server beinhalten verschiedene Objekte. In dem einen wird das Gruppenobjekt über das Gruppenfabrikobjekt bereitgestellt. In dem anderen existieren zu der Shapefactory und den Shapes noch das umgebende Transaktionsobjekt und das Fabrikobjekt zu einer Shapefactory.

4.2.2 Eingeschränkte Kooperation

In diesem Anwendungsszenario erweist es sich als sehr schwierig, die unkontrollierte Kooperation sinnvoll durch Transaktionsmechanismen einzuschränken, ohne erheblichen Aufwand zu betreiben. Es sprechen aber einige Gründe für eine solche Einschränkung.

- Das gleichzeitige Verändern des selben Shapes von zwei Whiteboards wird nicht abgefangen:
  Es können ungewollte Nebeneffekte auftreten. Einerseits kann sich bei zwei gleichen Relativoperationen, wie z.B. Skalieren, die Änderung verstärken, abschwächen oder aufheben. Andererseits setzt sich bei zwei gleichen Absolutoperationen, z.B. Setzen der Farbe, die als letzte gemachte Änderung durch.

  Über das Setzen von Sperren kann ein exklusiver Zugriff auf Shapes realisiert werden. Dafür muß jedes Whiteboard für die Bearbeitung von vorhandenen Shapes eine separate geschachtelte Transaktion erhalten und eine entsprechende Sperre anfordern. Für die Unterstützung von Gruppen erhalten die Mitglieder[23] die selbe geschachtelte Transaktion. Damit ist die gemeinsame Bearbeitung einzelner Shapes realisierbar. Die Bearbeitung der Shapefactory wird in diesem Fall nicht berücksichtigt. Das heißt, das Erzeugen und Löschen von Shapes erfolgt immer noch in der selben Wurzeltransaktion. Diese Lösung ergibt aber verschiedene und sehr aufwendige Implementierungsänderungen.

[23] in Form des Whiteboards

- **Shapefactory**:
  Eine Funktion der Shapefactory gibt die Liste der Shapes zurück. Diese Funktion wird von Whiteboards verwendet, um den aktuellen Konferenzzustand zu erhalten. Im Normalfall sind damit Lesesperren auf den einzelnen Shapes verbunden. Dies führt aber zu Konflikten mit gesetzten Schreibsperren in den Kindtransaktionen, wenn ein Whiteboard ein Shape bearbeitet. Aus diesem Grund müssen in der Wurzeltransaktion in der Realisierung dieser Funktion nur *Browse*-Sperren auf Shapes angefordert werden.
- **Wbgroup**:
  Für die Bearbeitung der Shapefactory wird ein Server verwendet, der ein Wurzeltransaktionsobjekt bereitstellt. Über das Wurzeltransaktionsobjekt wird für jedes sich anmeldende Whiteboard eine eigene Kindtransaktion erzeugt und diesem zugeordnet. In der Kindtransaktion erfolgt über die Nutzung der entsprechenden Schnittstellen das Setzen von Sperren auf Shapes. Zu diesem Zweck muß die Wbgroup-Schnittstelle um Funktionen erweitert werden, die es einem Whiteboard-Objekt erlaubt, eine Sperre auf ein Shape innerhalb seiner Kindtransaktion zu setzen.
- **Shape**:
  Jedes Shape besitzt eine Datenbankreferenz auf seine Shapefactory, welche verwendet wird, um der Shapefactory Veränderungen der Shapes mitzuteilen. Diese Referenz darf nicht verwendet werden, weil Shape und Shapefactory nicht in der selben geschachtelten Transaktion bearbeitet werden. Deshalb muß eine in eine Zeichenkette umgewandelte CORBA-Referenz, welche in der Initialisierung[24] der Shapefactory gesetzt wird, verwendet werden.
- **Whiteboard**:
  Für das Setzen von Sperren müssen Shapes auf der Whiteboard-Ebene selektiert und das betreffende Gruppenobjekt informiert werden. Weiterhin muß eine Rückmeldung erscheinen, wenn ein Objekt schon gesperrt ist.

Das gleichzeitige Bearbeiten und Löschen eines Shapes von zwei Whiteboards wird ebenfalls gelöst, weil für das Löschen eines Datenbankobjektes eine Schreibsperre für das Shape in der Wurzeltransaktion angefordert wird. Diese Operation schlägt aber fehl, wenn ein anderes Whiteboard dieses Objekt bearbeitet.

- Das gleichzeitige Einfügen von neuen Shapes in die Shapefactory kann zu Problemen führen:
  Aufgrund der Abhängigkeit einer Zeichnung von der Reihenfolge der Darstellung der Shapes kann es passieren, daß das zuletzt eingefügte Shape das vorherige überdeckt.

  Zur Lösung dieses Problems darf nur exklusiver Zugriff auf die Shapefactory erlaubt werden. Realisiert werden kann dies mittels einer separaten geschachtelten Transaktion, in der exklusiv die Shapefactory für ein Whiteboard gesperrt wird. Andere Whiteboards dürfen während dieser Zeit nur über *Browse*-Sperren sich den Konferenzzustand ansehen ihn aber nicht bearbeiten. Kooperation ist dadurch zwischen den Whiteboards allerdings nicht möglich. Darum sollten die Transaktionen auch von kurzer Dauer sein.

[24] im Konstruktor oder `init`-Methode

4.3 Diskussion und Grenzen des Anwendungsszenarios

Ein Nachteil der gewählten Architektur im Anwendungsszenarios ist die Verwendung von aktiven Objekten. Dies bedeutet, daß alle Objekte an der Kommunikation teilnehmen. Dazu beinhalten sie aber CORBA-Referenzen, an die Nachrichten verschickt werden. In der Unterstützung von Persistenz sind CORBA-Referenzen nur schlecht auf Datenbankkonzepte abbildbar. Datenbankreferenzen dürfen in den meisten Fällen, aufgrund unterschiedlicher Transaktionen, nicht verwendet werden. Desweiteren sind die Objekte selber für die Korrektheit der abgebildeten CORBA-Referenzen verantwortlich, was zu einem sehr hohen Verwaltungsaufwand führt. Aus diesem Grund sollten Kommunikation und Zustand streng getrennt werden. Dies kann erreicht werden, indem die Shapefactory und die Shapes keine Referenzen beinhalten und die Kommunikation nur über das Gruppenobjekt läuft.

Für die Wiederverwendung von Konferenzzuständen innerhalb verschiedener Konferenzen muß eine zusätzliche Funktion eingeführt werden. Diese garantiert über eine separate geschachtelte Transaktion, daß entweder der aktuelle Zustand, wenn die Bearbeitung momentan abgeschlossen ist, oder ein vorläufiger Zustand, wenn die Konferenz zur Zeit bearbeitet wird, geliefert wird. Auch hier tritt das Problem der aktiven Objekte auf, weil für die Shapefactory mit den Shapes die entsprechenden Referenzen für die neue Konferenz geändert werden müssen.

Kapitel 5

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, erweiterte Datenbankfunktionalität innerhalb eines Object Database Adapters bereitzustellen und in ein Anwendungsszenario zu integrieren. Die zu unterstützende erweiterte Datenbankfunktionalität sollte sich dabei durch einen hohen Kooperationsgrad auszeichnen. Aus diesem Grund wurden Transaktionen und ihre Synchronisation untersucht und inwieweit sie eine hohe Kooperation ermöglichen. Die Entscheidung fiel auf Sperrtechniken, weshalb Sperren auch gesondert betrachtet wurden. Diese Untersuchungen hatten zum Ergebnis, daß es notwendig ist, ein geschachteltes Transaktionskonzept und erweiterte Sperrmodi zu unterstützen. Dies spiegelt sich unter anderem auch in der Funktionalität wider, die ein CORBA-Client für den Zugriff auf Datenbankobjekte besitzt:

- **flexible Arbeit mit persistenten Objekten**: Durch die Realisierung einer Fabrik besitzt ein Client die Möglichkeit, nicht nur auf bestehende Datenbankobjekte zuzugreifen, sondern auch persistente Objekte zu erzeugen und zu löschen. Die Arbeit mit einem persistenten CORBA-Objekt[1] kann völlig transparent für den Client geschehen, in dem ihm die Persistenz verborgen bleibt.

- **generische Anfragen**: Durch die Umsetzung von globalen Kollektion-Interfaces kann eine Unterstützung von Anfragen ohne den Any-Typ von CORBA auskommen. Das Ergebnis einer Anfrage entspricht dem globalem Kollektions-Interface.

- **Versionierung von Objekten**: Es wird eine Klasse (Interface) bereitgestellt, von der alle persistenten CORBA-Objekte erben. Diese Klasse definiert Versionierungsmethoden, welche von einem Client benutzt werden.

- **erweiterte Sperren auf Objekte**: Es werden nicht nur Schreib- und Lesesperren unterstützt, sondern auch Browse- und Update-Sperren. Darüberhinaus ist es möglich, Sperren zu verstärken und abzuschwächen.

- **Transaktionskonzepte**: Es werden allgemeine Transaktionen, ohne die `per-client`-Aktivierung von Servern zu verwenden, unterstützt. Das heißt, es ist eine eindeutige Zuordnung von Client-Transaktion zur Datenbanktransaktion erreicht. Außerdem

[1] das Implementierungsobjekt ist persistent

wird eine eindeutige Zuordnung von Datenbankobjekten zu einer Transaktion durch die entsprechende Fabrikrealisierung erreicht.

- **erweiterte Transaktionen**: Es werden zusätzliche Transaktionsoperationen bereitgestellt. Dazu zählt ein Savepoint-Konzept und die Möglichkeit, auf die Sperrfreigabe Einfluß zu nehmen.
- **geschachtelte Transaktionen**: Offen und geschlossen geschachtelte Transaktionen werden unterstützt. Sie besitzen die Fähigkeit, Sub-Transaktionen zu erzeugen und je nach geschachtelten Transaktionstyp vererben sie ihre Sperren bzw. geben diese frei.
- **lange Transaktionen**: Eine lange Aktivitätsunterstützung wird durch ein allgemeines Workspace-Konzept realisiert. Es besteht die Möglichkeit, private und öffentliche Arbeitsbereiche zu definieren. In diese können aus unterschiedlichen Datenbanken sich die benötigten Objekte geholt werden, um längere Zeit lokal mit ihnen zu arbeiten.

Weiterhin wurde ein Lösungsansatz vorgestellt, der unabhängig vom Datenmodell des verwendeten Datenbank Management Systems ist. Er ermöglicht es auch, bestehende Datenbanken ohne Auswirkungen auf deren Anwendungen innerhalb eines Object Database Adapters zur Verfügung zu stellen. Zu diesem Zweck sollte eine Automatismus entwickelt werden, der aus bestehenden Datenbankklassen die entsprechenden Wrapper, Kollektionen und Fabriken mit der entsprechenden Implementierung generiert.

In der kooperativen dreidimensionalen Modellierung herrschen komplexere Bedingungen[2]. Deshalb sollte die Anwendbarkeit der entwickelten Konzepte in dieser Rubrik überprüft werden. Vor allem hier kann ein derart realisiertes System sehr schnell an seine Leistungsgrenzen stoßen. Ein entsprechender Laufzeittest sollte darüber Auskunft geben.

Die Einschränkungen der Kooperation müssen praktisch realisiert werden. Dazu ist das Anwendungsszenario momentan nicht oder nur eingeschränkt geeignet. Mit der entsprechenden Abstraktion zwischen Kommunikation und Zustand ist dies aber zu erreichen.

Ein sehr wichtiger Aspekt ist die Fehlerbehandlung. Dies betrifft z.B. die Sperroperationen. Dazu sind sowohl die IDL-Schnittstellen als auch die Implementierung sinnvoll zu erweitern.

Die Integration der aktiven Mechanismen muß noch vollständig betrachtet werden. Dazu wäre wohl eine Kopplung von den Trigger-Mechanismen und dem standardisierten Ereignis-Service der OMG geeignet. Die Persistenz von CORBA-Objekten sollte eine solche Kopplung nicht beeinflussen.

[2] Vor allem durch die Nutzung von Geometriekernen

Abbildungsverzeichnis

1.1 Klassifikation der CSCW-Systeme nach Raum und Zeit [BMST95] 13
1.2 Object Management Architecture [OMG90] 14
1.3 Common Object Request Broker Architecture [OMG96] 14
1.4 Einordnung der existierenden Prototypen nach [Neuw97] 18
1.5 Prozeßarchitektur [Neuw97] 19
1.6 TOBACO-Architektur 23
1.7 Whiteboard-Architektur 24

2.1 Transaktionszustände 28
2.2 Transaktionsphasen in einer optimistischen TA [Weik88] 30
2.3 Vergleich der Basisprotokolle 37
2.4 Mehrschichten-Schedule-Strategie 42
2.5 Transaktion in objektorientierten Systemen 44
2.6 Workspaces und Operationen 46

3.1 Versant Client/Server-Architektur 53
3.2 Die erweiterte Prozeßarchitektur 59
3.3 Die Transaktionshierarchie 64

4.1 Persistenz der Implementierungsobjekte 70
4.2 Hierarchie für persistente Implementierungsobjekte 71
4.3 Hierarchie für Kollektionsrealisierung 72
4.4 Hierarchie für die Fabrikrealisierung 73
4.5 Der Wrapperansatz 76

Tabellenverzeichnis

1.1 Kompatibilitätsmatrix für Paare von Sperren 16
1.2 existierende Object Database Adapter 17

2.1 Protokolleigenschaften 32
2.2 Innen-Außenwirkungs-Matrix 33
2.3 Kompatibilitätsmatrix für Intention-Sperren 35
2.4 Eigenschaften der Synchronisationsarten 48

3.1 Vergleich von ODBMS nach [Stei97] 52
3.2 Kompatibilitätsmatrix für Versant-Sperren 56

Literaturverzeichnis

[AbAg92] A.E.Abbadi, D.Agrawal. *Transaction Management in Database Systems.* in: Elma92, S.1-32.

[BaSc93] G.Barbian, G.Schlageter. *CODA- a GROUPBASE-System for Cooperative Applications.* in: Proc. ICICIS, Mai, 1993.

[BBG87] C.Beeri, P.A.Bernstein, N.Goodman. A Model for Concurrency in Nested Transaction Systems. Technical Report, Hebrew University of Jerusalem, erschienen in: JACM, 1987.

[BeGo83] P.Bernstein, N.Goodman. *Multiversion Concurrency Control - Theory and Algorithms.* in: TODS, Vol.8, No.4, 1983.

[BGH87] P.Bernstein, N.Goodman, V.Hadzilacos. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BHR80] R.Bayer, H.Heller, A.Reiser. *Parallelism and Recovery in Database Systems.* in: TODS, Vol.5, No.2, 1980.

[BKK85] F.Bancilhon, F.Kim, H.Korth. *A Model of CAD Transaction.* in: Proc. 11th Conf. on Very Large Data Bases, 1985,S.25-33.

[BMST95] K.Bauknecht, T.Mühlherr, C.Sauter, S.Teufel. *Computerunterstützung für die Gruppenarbeit.* Addison-Wesley, 1995.

[BoSc95] U.Borghoff, J.Schlichter. *Rechnergestützte Gruppenarbeit.* Springer Verlag, 1995.

[Catt97] R.G.G. Cattel. *The Object Database Standard: ODMG 93, Release 1.2.* Morgan Kaufmann Publisher, 1997.

[Chri96] A.Christiansen. *Integration relationaler Datenbanken in CORBA-Umgebungen.* Diplomarbeit, Fachbereich Informatik, Institut Technische Informationssysteme, Universität Magdeburg, August, 1996.

[DHKS89] P.Dadam, U.Herrmann, K.Küspert, G. Schlageter. *Sperren disjunkter, nichtrekursiver komplexer Objekte mittels objekt- und anfragespezifischer Sperrgraphen.* in: Datenbanksysteme in Büro, Technik und Wissenschaft, S. 98-113, 1989.

[Diet96] J.Dietrich. *Synchronisation kooperierender Transaktionen in Entwurfsumgebungen.* Diplomarbeit, Studiengang Informatik HI, Fachbereich Mathematik-Informatik, Universität-Gesamthochschule Paderborn, 1996.

[DiLM95] U.Dietrich, U. von Lukas, I.Morche. *Rechnergestütztes kooperatives Arbeiten im CAD-Umfeld auf der Basis standardisierter Werkzeuge.* In: CG topics 6/1995, Seiten 13-17.

[DLMR97] U.Dietrich, U. von Lukas, I.Morche, T.Runge. *Abschlußbericht TOBACO I: Werkzeuge für das kooperative Arbeiten.* ZGDV, ZGDV-Bericht 3/97, 1997.

[DRR90] K.R.Dittrich, M.A.Ranft, S.Rehm. *How to Share Work on Shared Objects in Design Database.* in: Proc. 6th International Conference on Data Engineering, Los Angeles, 1990, S.575-583.

[Elma92] N.M.Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publisher, 1992.

[FlMe97a] G.Flach, H.Meyer. *Das DICE-Projekt: Datenbankunterstützung für kooperative Anwendungen.* Rostocker Informatik-Berichte, 20, 1997.

[GHJV94] E.Gamma, R.Helm, R.Johnson, J.Vlissides.. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GLPT76] J.N.Gray, R.A.Lorie, G.R.Putzolu, I.L.Traiger. *Granularity of Locks and Degrees in a Shared Data Base.* in: Proc. IFIP Working Conf. on Modelling in DBMS, North-Holland, 1976.

[GNR90] J.GU, E.J.Neuhold, T.C.Rakow. *Serializability in Object-Oriented Database Systems.* in: Proc. 6th International Conference on Data Engineering, Los Angeles, 1990.

[GrRe92] J.Gray, A.Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1992.

[GrVo93] M.Groß-Hardt, G.Vossen. *Grundlagen der Transaktionsverarbeitung.* Addison-Wesley, 1993.

[Hard84] T.Härder. *Observations on Optimistic Cocurrency Control Schemes.* in: Information Systems, Vol.9, No.2, 1984.

[Heue97] A.Heuer. *Objektorientierte Datenbanken- Konzepte, Modelle, Standards und Systeme.* 2. aktualisierte und erweiterte Auflage, Addison-Wesley, 1997.

[HeSa95] A.Heuer, G.Saake. *Datenbanken - Konzepte und Sprachen* . International Thomson Publishing, 1995.

[HeSc95] A.Heuer, J.Schlegelmilch. *Made in France: Erfahrungen mit O_2.* in: Datenbank-Fokus, Januar, 1995, S.71-74.

[HoZd87] M.Hornick, S.Zdonik. *A shared Segmented Memory System for an Object-Oriented Database.* in: ACM Transactions on Office Information Systems, 5(1), Januar, 1987, S.70-95.

[IONA97] IONA Technoligies Ltd. *Orbix Programmer's Guide, Release 2.2.* Dublin, 1997.

[IONA98] IONA Technoligies Ltd. *Orbix+ObjectStore Adapter.* in: http://www-usa.iona.com/Developers/whitepaper/oosa/osatext.html, Januar, 1998.

[Kanu97] J.Kanuschin. *Versions- und Kooperationsverwaltung in einer verteilten, kooperativen CSCW-Umgebung.* Diplomarbeit am Lehrstuhl für Datenbank- und Informationssysteme, Fachbereich Informatik Uni-Rostock, 1997.

[KaPa84] P.C.Kanellakis, C.H.Papadimitriou. *On Concurrency Control by Multiple Versions.* in: TODS, Vol.9, No.1, 1984.

[KSUW85] P.Klahold, G.Schlageter, R.Unland, Wilkes. *A transation model supporting complex applications in integrated information systems.* in: Proc. ACM-SIGMOD, Austin, 1985, S.388-401.

[Kuna97] M. Kunas. *Transaktionssteuerung in einer verteilten Kooperativen CSCW-Umgebung.* Diplomarbeit am Lehrstuhl für Datenbank- und Informationssysteme, Fachbereich Informatik, Universität-Rostock, 1997.

[KuRo81] H.T.Kung, J.T.Robinson. *An Optimistic Method for Concurrency Control.* in: TODS, Vol.6, No.2, 1981.

[KoSp88] H.F.Korth, G.D.Speegle. *Formal Model of Correctness Without Serializability.* in: Proc. ACM-SIGMOD, International Conference on Management of Data, Chicago, 1988, S.379-386.

[KTW96] J.Klingemann, T.Tesch, J.Wäsch. *Semantic-Based Transaction Management for Cooperative Applications.* in: Proc. International Workshop on Advanced Transaction Models and Architectures(ATMA), Goa, 1996, S. 234-252.

[Maye94] E.Mayer. *Synchronisation in kooperativen Systemen.* Vieweg Verlag, 1994.

[Meck96] A.G.Meckenstock. *Synchronisations- und Recoverykonzepte für Transaktionen in kooperativen Entwurfsumgebungen.* Dissertation, Fachbereich Mathematik/Informatik, Universität-Gesamthochschule Paderborn, 1996.

[Meye96] H.Meyer. *Vorlesungsskripte Datenbanken III - Transaktionsverarbeitung.* Universität Rostock, 1996.

[MoZa95] T.J.Mowbray, R.Zahavi. *The Essential CORBA System Integration Using Distributed Objects.* John Wiley & Sons Inc., 1995.

[Neuw97] E.Neuwirt. *Konzeption und Implementation einer ORB/DBMS-Schnittstelle in einer CORBA-basierten CSCW-Umgebung.* Diplomarbeit am Lehrstuhl für Datenbank- und Informationssysteme, Fachbereich Informatik, Universität Rostock, 1997.

[$O_2$96b] O_2-Technology Ltd. *ODMG C++ Binding Guide.* Release 4.6, Mai 1996.

[OMG90] Object Management Group. *Object Management Architecture Guide.* in: OMG Publication, Boulder (CO), 1990.

[OMG94a] Object Management Group. *Object Services Architecture.* OMG-Dokument 94.11.12, 1994.

[OMG94b] Object Management Group. *Common Object Service Spezifikation.* Jon Wiley & Sons,Inc., 1994.

[OMG95a] Object Management Group. *Common Facilities Architecture.* OMG-Dokument 95.1.2, 1995.

[OMG95b] Object Management Group. *Common Facilities Roadmap.* OMG-Dokument 95.1.32, 1995.

[OMG96] Object Management Group. *CORBA 2.0 Specification.* OMG-Dokument 96.03.04, 1996.

[Reve96] F.Reverbel. *Persistence in Distributed Object Systems: ORB/ODBMS-Integration.* Ph.D. Dissertation, University of New Mexico, Computer Science Department, 1996.

[Ruiz97] T.S.Ruiz Rugama. *Integration von Workflow-Management-Funktionalität in eine CORBA-basierte CSCW-Umgebung.* Diplomarbeit am Lehrstuhl für Datenbank- und Informationssysteme, Fachbereich Informatik Universität-Rostock, 1997.

[Schl81] G.Schlageter. *Optimistic Methods for Concurrency Control in Distributed Database Systemss.* in: VLDB, 1981.

[ScUn89] G.Schlageter, R.Unland. *Ein allgemeines Modell für Sperren in nicht-konventionellen Datenbanken.* in: Datenbanksysteme in Büro, Technik und Wissenschaft, S. 114-118, 1989.

[ScUn92] G.Schlageter, R.Unland. *A Transaction Manager Development Facility for Non Standard Database Systems.* in: [Elma92], S.399-466.

[Stei97] B.Stein. *Sanfte Revolution* in: Software Entwicklung, Oktober 1997, S.20-34.

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

[Vers97a] Versant Object Technology Corporation. *VERSANT ODBMS 5.0 C++ Tutorial Manual.* Menlo Park, 1997.

[Vers97b] Versant Object Technology Corporation. *VERSANT ODBMS 5.0 C++ Concepts and Usage.* Menlo Park, 1997.

[Vers97c] Versant Object Technology Corporation. *VERSANT ODBMS 5.0 C++ Reference Manual.* Menlo Park, 1997.

[Weik88] G.Weikum. *Transaktionen in Datenbanksystemen.* Addison-Wesley, 1988.

Abkürzungsverzeichnis

ACID Atomicity (Atomarität), Consistency (Konsistenz), Isolation (Isolation) und Durability (Dauerhaftigkeit).

BLU Basic Lockable Unit.

BOA Basic Object Adapter.

BOCC Backward Oriented Certification.

BOT Begin of Transaction.

BTO Basic Timestamp-Ordering.

CINK CSCW für Industriekooperation.

CORBA Common Object Request Broker Architecture.

CORM CORBA Object Relational Mapper.

CSCW Computer Supported Cooperative Work.

DB Database.

DBMS Database Management System.

DBS Database System.

DII Dynamic Invocation Interface.

DSI Dynamic Skeleton Interface.

EOT End of Transaction.

FOCC Forward Oriented Certification.

GUI Graphical User Interface.

HeLU Heterogeneous Lockable Unit.

HoLU Homogeneous Lockable Unit.

IDL Interface Definition Language.

JDBC Java Database Connectivity.

MVTO	Multi Version Timestamp Ordering.
ODA	Object Database Adapter.
ODA+	Extended Object Database Adapter.
ODBC	Open Database Connectivity.
ODBMS	Object Database Management System.
ODL	Object Definition Language.
ODMG	Object Database Management Group.
OMA	Object Management Architecture.
OMG	Object Management Group.
OQL	Object Query Language.
OOSA	Orbix+ObjectStore Adapter.
ORB	Objekt Request Brocker.
OTS	Object Transaction Service.
POS	Persistence Object Service.
RDBMS	Relational Database Management System.
SGT	Serialisierbarkeitsgraphentester.
TA	Transaction.
WB	Whiteboard.
WWW	World Wide Web.

Anhang A

Ausschnitte aus der Realisierung der Basisklassen

A.1 Die Realisierung für den Einstiegsserver

```
class db_server_i {
private:
 stringSeq gen_launch_command_flat();
 stringSeq gen_marker();
 IT_daemon_var daemon;
public:
 db_server_i(){daemon = IT_daemon::_bind();};
 ~db_server_i();
 flattransaction_ptr get_flattransaction (){
  char *server_name = new char[30];
  sprintf(server_name,"server%ld",time(0));
  daemon->newSharedServer
   (server_name, gen_marker(), gen_launch_command_flat(),0);
  daemon->addLaunchRights("all",server_name);
  daemon->addInvokeRights("all",server_name);
  flattransaction_ptr dummy_tra_ptr = short_tra::_bind(server_name);
  return flattransaction::_duplicate(dummy_tra_ptr);
 };
// ...
};
DEF_TIE_db_server(db_server_i)
```

A.2 Die Realisierung für persistente CORBA-Klassen

In diesem Abschnitt des Anhangs wird anhand eines konkreten persistenten CORBA-Objekts – Wbgroup-Objekt – aus dem Anwendungsszenario gezeigt, wie die Implementierungen zu gestalten sind.

A.2.1 Direkter Persistenzansatz

```
class pers_CORBA_i:public PVirtual{
public:
 void upgrade_lock (lockmodus sperre) {
   if (sperre == B) dom->upgradelock(this,NOLOCK,O_LK_IMPROMOTE); /...};
 void downgrade_lock (lockmodus sperre) {// ...};
 void make_versioned () {this->preptoversion(); };
 virtual pers_CORBA_ptr create_version () {
  return pers_CORBA::_nil(); };
 virtual list_of_pers_CORBA_ptr all_kind_version () {
  return list_of_pers_CORBA::_nil(); };
 virtual pers_CORBA_ptr get_parent_version () {
  return pers_CORBA::_nil(); };
 virtual void release_instance () {this->releaseobj(); };;
// fuer die Initialisierung transienter Daten konkreter Klassen
 virtual void init () {};
};
DEF_TIE_pers_CORBA{pers_CORBA_i}


class Wbgroup_i:public pers_CORBA_i{
// ...
public:
 pers_CORBA_ptr create_version() {
  Link<Wbgroup_i> versionlink = L_AS(Wbgroup_i, this->createvsn());
  return Wbgroup::_duplicate(new TIE_Wbgroup
     (Wbgroup_i)(versionlink)); };
// ...
}
DEF_TIE_Wbgroup(Wbgroup_i)
```

A.2.1.1 Die Realisierung für Kollektionen

```
class list_of_Wbgroup_i:public list_of_pers_CORBA_i{
  LinkVstr<Wbgroup_i>  wbgroup_liste;
public:
  list_of_Wbgroup_i(LinkVstr<Wbgroup_i>  liste_wbgroup) {
    wbgroup_liste = liste_wbgroup;};
  void remove_element_at (long position) {
    wbgroup_liste.remove(position); };
  void add_ellement_at (pers_CORBA_ptr object){
    wbgroup_liste.add((Wbgroup::_narrow(object))->_deref()); };
  pers_CORBA_ptr get_element_at (long position); {
     return Wbgroup::_duplicate(new TIE_Wbgroup
        (Wbgroup_i)(wbgroup_liste[position])); };
  long list_size () {return wbgroup_liste.size(); };
```

```
// usw.
};
DEF_TIE_list_of_pers_CORBA(list_of_pers_CORBA_i)
```

A.2.1.2 Die Realisierung für Fabriken

```
class Wbgroup_factory_i:public pers_CORBA_factory_i {
protected:
 PPredicate umwandeln(const char* pred) {// ...};
public:
 pers_CORBA_ptr create_pers_CORBA () {
  Wbgroup_i* dummy = O_NEW_PERSISTENT(Wbgroup_i)();
  return Wbgroup::_duplicate(new TIE_Wbgroup(Wbgroup_i) (dummy)); };
 pers_CORBA_ptr get_pers_CORBA(const char* pred, lockmodus sperre) {
  if (sperre == B) {
  LinkVstr<Wbgroup_i> dummy = PClassObject<Wbgroup_i>::Object().select
     (0,TRUE,NOLOCK,umwandeln(pred));
  return Wbgroup::_duplicate(new TIE_Wbgroup
     (Wbgroup_i) (dummy[0].pointer());}; \\ ... };
 void delete_pers_CORBA (pers_CORBA_ptr del_Object);
 virtual list_of_pers_CORBA_ptr get_list_of_pers_CORBA () {
  LinkVstr<Wbgroup_i> dummy = PClassObject<Wbgroup_i>::Object().select
      (0,TRUE,NULL_PREDICATE);
  return list_of_Wbgroup::_duplicate(new TIE_list_of_Wbgroup
      (list_of_Wbgroup_i)(dummy)); };
};
DEF_TIE_Wbgroup_factory(Wbgroup_factory_i)
```

A.2.2 Die Realisierung mit dem Wrapperansatz

```
// C++
extern PDOM *dom; // Transaktion

class pers_CORBA_wrapper_i
{
protected:
 virtual Link<PObject> get_Implementation_Object() { return 0; };
public:
 void release_instance() {
 get_Implementation_Object()->releaseobj(); };
// Funktionalitaet fuer Datenbankobjekte
 void upgrade_lock(lock_modus lock) {
  if (lock == B)
   dom->upgradelock(get_Implementation_Object(), NOLOCK, O_LK_IMPROMOTE);
 //... };
 void downgrade_lock(lock_modus lock) {// ... };
```

```
// Versionierungsfunktionalitaet
 void make_versioned() {
   get_Implementation_Object()->preptoversion(); };
 virtual pers_CORBA_wrapper_ptr create_version() { //...};
// ...
};
DEF_TIE_pers_CORBA_wrapper(pers_CORBA_wrapper_i)

class Wbgroup_wrapper_i:public pers_CORBA_wrapper_i {
 Link<Wbgroup> ref;
protected:
 Link<Pobject> getImplementation_Object() { return ref; };
public:
 Wbgroup_wrapper_i(){};
 Wbgroup_wrapper_i(Link<Wbgroup> uref){ref = uref};
 pers_CORBA_wrapper_ptr create_version() {
  Link<Wbgroup> versionlink = L_AS(Wbgroup, ref->createvsn());
  return Wbgroup_wrapper::_duplicate(new TIE_Wbgroup_wrapper
     (Wbgroup_wrapper_i)(new Wbgroup_wrapper_i(versionlink))); };
 //...
// Methoden von Wbgroup
 void registerWB (Whiteboard_ptr wb){
   ref->registerWB(Whiteboard_ptr wb);
 };
// ...
};

class list_of_Wbgroup_wrapper_i:public list_of_pers_CORBA_wrapper_i
{
private:
 LinkVstr<Wbgroup> list_wbgroup;
public:
 list_of_Wbgroup_wrapper_i () {
  list_wbgroup = new LinkVstr<Wbgroup>();};
 list_of_Wbgroup_wrapper_i (LinkVstr<Wbgroup> wbgroup_list) {
  list_wbgroup = wbgroup_list};
 ~list_of_Wbgroup_wrapper_i( ){};
 pers_CORBA_wrapper_ptr getelement_at(long position) {
  Wbgroup_wrapper_i *dummy =new Wbgroup_wrapper_i
       (list_wbgroup[position]);
  return Wbgroup_wrapper::_duplicate(new TIE_Wbgroup_wrapper
   (Wbgroup_wrapper_i)(new Wbgroup_wrapper_i(list_wbgroup[position])));
  };
// usw.
}
DEF_TIE_list_of_Wbgroup_wrapper(list_of_Wbgroup_wrapper_i)
```

```
class factory_of_Wbgroup_wrapper_i:public factory_of_pers_CORBA_wrapper_i
{
public:
 pers_CORBA_wrapper_ptr create_pers_CORBA_wrapper() {
  Link<Wbgroup> object_ref = O_NEW_PERSISTENT(Wbgroup)();
  return Wbgroup_wrapper::_duplicate(new TIE_Wbgroup_wrapper
   (Wbgroup_wrapper_i)(new Wbgroup_wrapper_i(object_ref)); };
 Wbgroup_wrapper_ptr get_pers_CORBA_wrapper(const char* Pred) {// ...};
// ...
}
DEF_TIE_factory_of_Wbgroup_wrapper(factory_of_Wbgroup_wrapper_i);
```

Anhang B

Ausschnitte aus der Transaktionsrealisierung

B.1 Die allgemeine Transaktionsrealisierung

```
extern PDOM *dom;

class flattransaction_i:public transaction_i{
public:
 void commit () { dom->commit(); };
 void abort () { dom->abort();};
 void savepoint () { dom->savepoint();};
 void undosavepoint () { dom->undosavepoint();};
 void checkpoint () { dom->checkpoint();};

 void set_default_db (const char * db_name) {
  dom->set_default_db(db_name);};
 void connect_db (const char * db_name) {
  dom->connectdb(db_name);};
 void disconnect_db (const char * db_name) {
  dom->disconnectdb(db_name);};

 pers_CORBA_factory_ptr get_factory (const char * which) {
  if (strcmp(which,"Wbgroup"))
   return Wbgroup_factory::_duplicate(new TIE_Wbgroup_factory
     (Wbgroup_factory_i)());
  // f"ur jede vorhandene Klasse die entsprechede Factory erzeugen
  };
  query_ptr get_query_object () {
    return query::_duplicate(new TIE_query(query_i)()); };
};
DEF_TIE_flattransaction(flattransaction_i)
```

B.2 Die geschachtelte Transaktionsrealisierung

```
class opennestedtransaction_i:public transaction_i{
 char* session_name;
 o_uid ancestor;
public:
 opennestedtransaction_i (const char* name, o_uid anc) {
  session_name = new char[strlen(name)+1];
  strcpy(session_name, name);
  ancestor = anc;
 };
 ~opennestedtransaction_i() {
   delete[] session_name;
// ...
   dom->exit();
 };
 void commit() {
  if(ancestor!=0) {
  o_opts opt; opt[0] = O_CUST_INHERIT
  dom->endtransaction(O_COMMIT, opt, NULL, NULL);
  } else { //...
  };
 };
 void abort() {
  if(ancestor!=0) {
  o_opts opt; opt[0] = O_CUST_INHERIT
  dom->endtransaction(O_ABORT, opt, NULL, NULL);
  } else { //...
  };
 };
 opennestedtransaction_ptr new_opennestedtransaction(){
  o_uid transid; dom->gettransid(&transid);
  dom->begintransaktion();
  o_4b pid = dom->forkprocess();
  dom->joinsession(session_name, &transid, TRUE);
// ...
  nested_tra_i *dummy = new opennestedtransaction_i(session_name, transid);
  return opennestedtransaction::_duplicate(
    new TIE_opennestedtransaction(opennestedtransaction_i)(dummy));
// ...
 };
}

class closednestedtransaction_i:public transaction{
 char* session_name;
 o_uid ancestor;
public:
```

```
 closednestedtransaction_i(const char* name, o_uid anc) {
  session_name = new char[strlen(name)+1];
  strcpy(session_name, name); ancestor = anc;
 };
 ~closednestedtransaction_i() {
   delete[] session_name;
   dom->exit();
  };
 void commit() {
  if(ancestor!=0) {
   o_opts opt; opt[0] = O_CUST_INHERIT;
   dom->endtransaction(O_COMMIT, opt, NULL, !NULL);
  } else { //...
  };
 };
 void abort() {
  if(ancestor!=0) {
   o_opts opt; opt[0] = O_CUST_INHERIT;
   dom->endtransaction(O_COMMIT, opt, NULL, !NULL);
  } else { //...
  };
 };
 closednestedtransaction_ptr new_closednestedtransaction(){
  o_uid transid;
  dom->gettransid(&transid);
  dom->begintransaktion();
  o_4b pid = dom->forkprocess();
  dom->joinsession(session_name, &transid, TRUE);
// ...
  nested_tra_i *dummy = new closednestedtransaction_i
      (session_name, transid);
  return closednestedtransaction::_duplicate(
    new TIE_closednestedtransaction(closednestedtransaction_i)(dummy));
// ...
 };
}
```

B.3 Die lange Transaktionsrealisierung

```
class longtransaction_i:public flattransaction_i{
 LinkVstr<pers_CORBA_i> gen_dblist(list_of_pers_CORBA_ptr) {//...};
public:
 void create_workspace(in string name) {
  Vstr<char> ret_val;
  dom->utility(O_UT_CREATEDB, "", name, &ret_val);
  dom->utility(O_UT_MAKEDB, "", name, &ret_val); };
```

```
 void delete_workspace(in string name) {
  Vstr<char> ret_val;
  dom->utility(O_UT_REMOVEDB, "", name, &ret_val); };
// Transferoperationen
 void checkin(list_of_pers_CORBA_ptr back, const char* db_name) {
  dom->checkin(gen_dblist(back),db_name,NULL_VSTR,NULL_VSTR);
 list_of_pers_CORBA_ptr checkout(list_of_pers_CORBA_ptr out,
                                const char* db_name)
 {// ...};

 void set_long_tra_name(const char* long_tra_name) {
    dom->set_long_trans_name(long_tra_name); };
// ...
}
```

Anhang C

Das Anwendungsszenario

C.1 Relevante IDL-Schnittstellen

```
interface Wbgroup:pers_CORBA
{
 oneway void registerWB(in Whiteboard wb);
 oneway void unregisterWB(in Whiteboard wb);
 ShapeFactory get_ShapeFactory();
 oneway void updateShape(in Shape s);
 oneway void newShape(in Shape s);
 oneway void deleteShape(in long s);
 oneway void deleteAllShapes();
 oneway void setSlider(in short which, in long pos);
 oneway void refresh();

// Transaktionsoperationen
 void savepoint();
 void undosavepoint();
 void checkpoint();
 void ping();
};

interface list_of_Wbgroup:list_of_pers_CORBA
{
};

interface Wbgroup_factory:pers_CORBA_factory
{
};

interface ShapeFactory:pers_CORBA
{
 long get_oid();
```

```
 void set_wbgroup_ptr(in Wbgroup gruppe);

 oneway void createPoint(in Shape::Point position,
                   in Attribs attr, in string name, in string email);
 oneway void createPolyline(in PointList plist,
                   in Attribs attr, in string name, in string email);
 oneway void createLine(in Shape::Point begin, in Shape::Point end,
                   in Attribs attr, in string name, in string email);
 oneway void createEllipse(in Shape::Point left_top, in long height,
                   in long width, in long angle,
                   in Attribs attr, in string name, in string email);
 oneway void createRectangle(in Shape::Point left_top, in long width,
                   in long height, in long angle,
                   in Attribs attr, in string name, in string email);
 oneway void createText(in Shape::Point position,
    in string text, in long angle,
                   in TextAttribs text_attribs, in Attribs attr,
                   in string name, in string email);
 oneway void createArrow(in Shape::Point position,
                   in Attribs attr, in string name, in string email);
 oneway void createLocalPic(in Shape::Point position, in long width,
                   in long height, in string filename,
                   in Attribs attr, in string name, in string email);
 oneway void createRemotePic(in Shape::Point position, in long width,
                   in long height, in string filename,
    in RemotePic remPic, in Attribs attr,
      in string name, in string email);

 oneway void deleteAllShapes();
 oneway void removeShape(in Shape s, in boolean quiet);
 oneway void updateShape(in Shape s);

 readonly attribute AllShapes shapeList;
 readonly attribute long shapeCount;
};

interface list_of_ShapeFactory:list_of_pers_CORBA
{
};

interface ShapeFactory_factory:pers_CORBA_factory
{
};
```

C.2 Auszug aus dem relevanten Teil eines Whiteboardes

```
Wbgroup_factory_var fac_var = Wbgroup_factory::_bind(":Wbgroup");
// Zugriff auf vorhandenen Konferenz
list_of_Wbgroup_var extent_var = list_of_Wbgroup::_narrow
           (fac_var->get_list_of_pers_CORBA());
Wbgroup_var wb_var = Wbgroup::_narrow
           (extent_var->get_element_at(1));
// Erzeugung einer neuen Konferenz
Wbgroup_var wb_var = Wbgroup::_narrow
           (fac_var->create_pers_CORBA());
ShapeFactory_var sf_var = wb_var->get_ShapeFactory();
```

C.3 Auszug aus der Wbgroup-Implementierung

```
// Initialisierung des Transaktionsservers im Konstruktor
 db_server_var dummy_server = db_server::_bind(":Db_server");
 flattransaction_ptr dummy_tra_ptr = dummy_server->get_flattransaction();
 ShapeFactory_factory_var dummy_fac = ShapeFactory_factory::_narrow
     (dummy_tra_ptr->get_pers_CORBA_factory("ShapeFactory"));
// Erzeugen einer persistenten CORBA-Shapefactory-Referenz
 ref_fac_ptr = O_NEW_PERSISTENT(ref_of_ShapeFactory_i)();
 ShapeFactory_var shape_fac = ShapeFactory::_narrow
    (dummy_fac->create_pers_CORBA());
 fac_oid = shape_fac->get_oid();
// Holen der Shapefactory nur in der init-Routine
 ShapeFactory_var shape_fac = ShapeFactory::_narrow
    (dummy_fac->get_pers_CORBA(fac_oid));
// Setzen der Referenz im Konstruktor sowie in der init-Routine
 ref_fac_ptr->set_ref_of_CORBA(shape_fac);
```

Anhang D

Allgemeine Anleitung für die Nutzung

Ich möchte in diesem Teil des Anhangs vorstellen, wie ein beliebiger CORBA-Client vorzugehen hat, um die Funktionalität des entwickelten Object Database Adapters zu nutzen.

Angenommen ein Client möchte normale Transaktionsfunktionaliät verwenden und will auf Objekte einer Datenbank mit dem Namen `Personen`, die Personen verwaltet, zugreifen. Das vereinfachte IDL-Interface für die Client-Sicht sieht dazu folgendermaßen aus.

```
interface Person:pers_CORBA{
 attribute string Vorname;
 attribute string Nachname;
};
interface list_of_Person:list_of_persCORBA{};
interface Person_factory:pers_CORBA_factory{};
```

Um nun z.B. eine Person zu erzeugen und die entsprechenden Attribute zu setzen, muß der Client sich zunächst eine Transaktion anfordern. Weiterhin braucht er die entsprechende Fabrik zum Personen-Interface. Dies und das Setzen der Datenbank erfolgt über die Transaktion.

```
db_server_var oda_ref = db_server::_bind(":Db_server");
flattransaction_ptr ta_ptr = oda_ref->get_flattransaction();
ta_ptr->set_db("Personen");
Person_factory_ptr fab_ptr = Person_factory::_narrow
   (ta_ptr->get_pers_CORBA_factory("Person"));
Person_ptr pers_ptr = Person::_narrow(fab_ptr->>create_pers_CORBA());
pers_ptr->Vorname("Peter");
pers_ptr->Nachname("Corba");
ta_ptr->commit();
```

Selbständigkeitserklärung

Hiermit versichere ich, daß ich die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen angefertigt habe.

Karsten Wendt

Rostock, den 31.05.1998

Thesen

I In den letzten Jahren hat die Dezentralisierung bei einer gleichzeiten Verkürzung der Produktentwicklungszeiten stark zugenommen. Aus diesem Grund nimmt die Bedeutung von CSCW-Anwendungen stetig zu. Mit dem CORBA-Standard existiert die Möglichkeit, diese Anwendungen effektiv zu realisieren, denn gerade in den Punkten der Unterstützung von Transparenz bezüglich Programmiersprachen, Netzwerke und Betriebssysteme gibt es zur Zeit keine Alternative.

II Nachdem mit dem Internet Inter ORB Protocoll (IIOP) eine Kopplung von CORBA-Implementierungen verschiedener Hersteller gewährleistet ist, wird im zunehmenden Maß eine Integration anderer Technologien z.B. WWW, erfolgen.

III Der Stellenwert der Integration von Persistenzdiensten in eine CORBA-Umgebung wird für eine effektive Wiederverwendung von Daten jeglicher Art zunehmen. Dies schließt neben der Persistenz auch eine generische Anfrage- und eine Beziehungsunterstützung ein. Damit soll es ermöglicht werden, auch von einem beliebigen CORBA-Client effizient auf persistente Daten zuzugreifen.

IV Um bestimmte Fehlersituationen zu vermeiden, müssen Synchronisationsmaßnahmen getroffen werden. Aus diesem Grund sind exklusiver Zugriff auf Objekte und die Zusammenfassung mehrerer IDL-Operationen zu einer Transaktion besonders wichtig. Datenbanksysteme besitzen für diesen Zweck geeignete Mechanismen.

V In CSCW-Anwendungen liegt ein Schwerpunkt auf der Unterstützung von Kooperation, und es kommt häufig auf die Abbildung von Projektstrukturen (Gruppen) an. Dafür erweisen sich normale flache Transaktionen als zu restriktiv. Aus diesem Grund wird es notwendig, mittels geschachtelter Transaktionen diese Einschränkungen zu minimieren. Eine Integration solcher Datenbanktransaktionen wird deshalb besonders wichtig.

VI In der Modellierung kommt es häufig auf die Unterstützung von mehreren Zuständen für ein Objekt an. In einem objektorientierten DBMS ist die Versionierung schon meistens enthalten. Die Nutzung der Funktionalität, die das DBMS bereitstellt, ist effizienter als eine aufwendige eigene Realisierung, besonders wenn benutzte Objekte persistent sind.

VII In einer verteilten CORBA-Umgebung existieren eine Anzahl von CORBA-Server, welche Datenbankfunktionalität bereitstellen. Jeder dieser Server besitzt einen eigenständigen Transaktionskontext. Deshalb ist es nicht sinnvoll, eine automatische

Objektaktivierung zu realisieren. Dies liegt in der nicht eindeutigen Zuordnung von Objekt zu einer Datenbanktransaktion begründet.

VIII Eine eindeutige Zuordnung von Client-Transaktion zu einer Datenbanktransaktion wird mit dem gemachten Ansatz realisiert. Diese Zuordnung durch `per-client`-Aktivierung zu erreichen, ist nicht geeignet, aufgrund der fehlenden Unterstützung der Weitergabe von CORBA-Objektreferenzen dieses Servers an unterschiedliche Clients.

IX In vielen Fällen existieren schon Datenbanken. Die Objekte dieser Datenbanken ohne Auswirkungen auf bestehende Anwendungen bereitzustellen, wird mit dem Wrapper-Ansatz erreicht. In diesem Fall geht es vor allem darum, daß sich das Datenbankschema und auch Implementierung nicht ändert. Dies kann erreicht werden, indem die notwendigen Anpassungen an eine Nutzung der Datenbankobjekte in einem CORBA-Umfeld durch die Wrapper geschehen. Durch den Wrapperansatz kann auch vom Datenhaltungsmodell des Datenbanksystems abstrahiert werden.