Diplomarbeit

CORBA und ODMG-93
als Grundlage zur
Realisierung eines
föderierten Datenbanksystems

Stefan Sander

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

28. August 1997

Gutachter:

Dr. W. Hasselbring
Prof. Dr. E.-E. Doberkat

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1 Einleitung

1.1 Motivation und Ziel

Der zunehmende Einsatz von Datenbanksystemen (DBS) führte in den letzten Jahren dazu, daß heute in vielen Organisationen unterschiedliche Datenbanksysteme nebeneinander im Einsatz sind. Gründe hierfür liegen einerseits darin, daß verschiedene Datenbankmanagementsysteme (DBMS) sich in unterschiedlicher Weise gut für bestimmte Anwendungsbereiche eignen, andererseits unterstützen Hersteller von spezieller Anwendungssoftware im allgemeinen nur bestimmte DBMS. Hieraus resultiert, daß der eine Organisation betreffende Realweltauschnitt auf eine Menge von unterschiedlichen Datenhaltungssystemen abgebildet wird. Es zeigt sich, daß diese Systeme sehr oft inhaltlich verwandte Daten verwalten [Rah94]. Diese Situation verschärft sich auch in dem Maße, in dem Organisationen sich zusammenschließen, etwa bei Fusionierung von Unternehmen oder durch Restruktierung von Behörden.

Das Betreiben dieser voneinander isolierten DBS innerhalb einer Organisation impliziert eine Reihe von gravierenden Problemen. Die redundante Haltung der gleichen Information in verschiedenen, nicht kooperienden Systemen legt dabei den Benutzern zusätzliche Mehrarbeit auf. Das Verknüpfen von Daten unterschiedlicher DBS wird in keiner Weise unterstützt, wodurch ein Informationsverlust ensteht („das Ganze ist mehr als die Summe seiner Teile“). Die Entwicklung und Realisierung eines einzigen, unternehmensweiten Datenmodells als Ausweg aus der Krise führt häufig zu sehr komplexen, schlecht überschaubaren und ineffizienten Informationssystemen. Darüber hinaus ist dieser Weg häufig in ökonomischer Hinsicht nicht sinnvoll, da eine Migration erhebliche Kosten verursacht und die bereits bestehenden Anwendungen einen großen Teil des Unternehmens-Know-Hows darstellen.

Ein vielversprechender Ansatz zur Integration von heterogenen Datenbeständen innerhalb einer Organisation stellen föderierte Datenbanksysteme (FDBS) dar, welche aktueller Gegenstand der Forschung sind. Föderierte Datenbanksysteme versuchen mithilfe einer zusätzlichen Softwareschicht die Heterogenität der zu integrierenden Datenbanksysteme transparent zu machen und eine neue integrierte Sicht auf die verteilten Daten einer Organisation zu unterstützen [SL90]. Im Rahmen der Föderation bewahren die bestehenden Datenbanksysteme weitgehend ihre Autonomie, so daß bestehende Anwendungen unverändert weiterlaufen können. Insofern ist ein föderiertes Datenbanksystem durchaus mit einer politischen Föderation, wie etwa der *Europäischen Union* oder der *Uno*, zu vergleichen. Ähnlich wie bei einer politischen

Föderation besteht die besondere Problematik bei der Entwicklung eines föderierten Datenbanksystems in der Heterogenität der zu integrierenden Teilsysteme.

Die Zielsetzung dieser Diplomarbeit besteht darin, vor dem Hintergrund eines konkreten Anwendungs-Szenarios ein föderietes Datenbanksystem zu entwickeln, das versucht die Heterogenitäts-Problematik durch den Einsatz von Standards zu lösen. Als Anwendungs-Szenario wurde die Eingemeindung der Städte Bochum und Wattenscheid und die daraus resultierende Restrukturierung der städtischen Behörden gewählt. Das zu entwickelnde föderierte Datenbanksystem soll die datentechnische Kooperation ausgewählter Behörden unterstützen. Hierbei sollen insbesondere zwei wesentliche Aspekte der Heterogenität der zu integrierenden Datenbanksysteme betrachtet werden:

- Bezüglich der Heterogenität in der Ablaufumgebung soll in der Diplomarbeit der Ansatz verfolgt werden, die Komponenten des angestrebten FDBS als verteilte, kooperierende Objekte zu konzipieren und zu realisieren. Diese kommunizieren über einen CORBA kompatiblen Object Request Broker (ORB) miteinander. Die Common-Object-Request-Broker-Architecture (CORBA) beschreibt eine objektorientierte Infrastruktur, welche die Entwicklung von verteilten objektorientierten Systemen standardisiert. Der CORBA-Standard könnte schon bald zu einer Art *Esperanto* für verteilte heterogene Objekte avancieren.

- Die Heterogenität der Datenbankmanagementsysteme (DBMS) bzw. ihrer unterschiedlichen Datenmodelle soll durch den Einsatz des ODMG-93-Standards für objektorientierte Datenbankmanagementsysteme (ODBMS) [Cat94] überwunden werden. Das objektorientierte Datenmodell dieses Standards soll dabei als gemeinsames Datenmodell des föderierten Datenbanksystems fungieren. Die Rolle des gemeinsamen Datenmodells ist in etwa vergleichbar mit der Rolle der gemeinsamen Amtsprache in einer politischen Föderation. Ihre Ausdrucksstärke hat einen wesentliche Einfluß auf den Grad der erreichbaren Kooperation.

Das föderierte Datenbanksystem soll unter Einsatz objektorientierter Entwurfsmethoden entwickelt werden. In der Diplomarbeit soll die Unified Modeling Language [BR96], die durch das Entwurfswerkzeug Rational Rose 4.0 unterstützt wird, eingesetzt werden.

1.2 Übersicht

Ziel dieses Abschnitts ist es, einen Überblick über den Aufbau und die Struktur der Diplomarbeit zu geben und die Inhalte der einzelnen Kapitel in einen Gesamtzusammenhang zu stellen.

Die Diplomarbeit gliedert sich in zwei Teile. Im ersten Teil werden die begrifflichen Grundlagen und Konzepte der verwendeten Standards und des Themenbereiches „Föderierte Datenbanksysteme“ erarbeitet. Aufbauend auf diesem begrifflichen Fundament, wird schließlich im zweiten Teil der Arbeit die Realisierung eines föderierten Datenbanksystems dargestellt. Hierbei werden insbesondere die gewählten Ansätze, das konkrete Anwendungs-Szenario und der objektorientierte Entwurf des Systems beschrieben.

Den thematischen Einstieg in den ersten Teil der Arbeit bildet das Kapitel 2 mit einer Beschreibung der wesentlichen Begriffe und Konzepte des CORBA-Standards. In Kapitel 3 folgt eine Einführung in den ODMG-93-Standard für objektorientierte Datenbankmanagementsysteme (ODBMS). Hierbei wird insbesondere das objektorientierte Datenmodell des Standards, welches als gemeinsames Datenmodell des föderierten Datenbanksystems eingesetzt wird, vorgestellt. Das Kapitel 4 führt schließlich in den Themenbereich „Föderierte Datenbanksysteme“ ein. In diesem Kapitel wird eine Referenz-Architektur für föderierte Datenbanksysteme dargestellt, die eine wichtige Grundlage für die Realisierung des FDBS darstellt. Im Abschluß dieses Kapitels werden einige FDBS-Forschungsprojekte betrachtet. Hierbei werden bereits einige Konzepte identifiziert, die bei der Entwicklung des FDBS Verwendung finden.

Der zweite Teil der Arbeit beginnt mit Kapitel 5. In diesem Kapitel wird zunächst das konkrete Anwendungs-Szenario vorgestellt und besondere Anforderungen an das FDBS in Bezug auf dieses Szenario analysiert. Im Anschluß wird der gewählte Ansatz zur Realisierung des föderierten Datenbanksystems dargestellt und begründet. In Kapitel 6 wird schließlich die entwickelte Software-Architektur des FDBS und dessen Feinentwurf beschrieben. Das sich daran anschließende Kapitel 7 stellt das in der Arbeit verwendete Datenbank-Adapter-Konzept dar, mit dessen Hilfe die Datenmodell-Heterogenität der zu integrierenden Datenbanksysteme gelöst wurde. Die Kapitel 6 und 7 bilden den inhaltlichen Schwerpunkt der Arbeit. Sie beschreiben detailliert die objektorientierte Entwicklung des föderierten Datenbanksystems. In Kapitel 8 wird das realisierte FDBS und dessen Fähigkeiten und Möglichkeiten vor dem Hintergrund des Anwendungs-Szenarios dargestellt.

Den Abschluß der Arbeit bildet mit Kapitel 9 eine Zusammenfassung mit Ausblick. Hier werden die Ergebnisse der Arbeit betrachtet und bewertet.

Im Anhang werden die verwendeten Systeme, Werkzeuge und Methode, die in der Diplomarbeit eingesetzt wurden, dargestellt. Zudem wird der Quell-Code des realisierten föderierten Datenbanksystems angegeben.

Kapitel 2 Der CORBA-Standard

2.1 Grundlagen

Die Object Management Group (OMG) ist eine Organisation, in der sich führende Soft- und Hardwarehersteller zusammengefunden haben, um eine offene Referenz-Architektur für verteilte objektorientierte Systeme in heterogenen Umgebungen zu definieren. Im Zentrum dieser Architektur steht eine Infrastruktur, die es verteilten heterogenen Objekten ermöglicht, netzwerktransparent miteinander zu kooperieren. Die Heterogenität bezieht sich auf die Hardwareplattform, das Betriebssystem und die Implementierungssprache der verteilten Objekte. Der CORBA-Standard ist also ein objektorientierter Middelware-Ansatz. [OHE96] und geht weit über bestehende Middelware-Ansätze wie z.B. *PVM* (*Parallel Virtual Machine*) hinaus. Die OMG hat inzwischen mehr als 500 Mitglieder und gilt zur Zeit als das größte industrielle Standardisierungs-Gremium. Die OMG begann 1989 ihre Arbeit. Die erste Version des CORBA-Standards wurde im Jahre 1991 veröffentlicht. Die aktuelle Revision des Standards heißt CORBA 2.0 und erschien im Dezember 1994.

2.2 Die Object Management Architecture (OMA)

Die *Object Management Architecture (OMA)* stellt den Gesamtrahmen der OMG-Aktivitäten dar. Sie stellt eine Referenz-Architektur für verteilte objektorientierte Systeme dar und zerlegt den von der OMG adressierten Problembereich auf hoher Abstraktionsebene in fünf Komponenten. Die OMA beinhaltet das ***Objektmodell***, den ***Object-Request-Broker***, die ***Common-Object-Services***, die ***Common-Facilities*** und die ***Application-Objects*** (siehe Abbildung 1). Zentrale Komponente der Architektur ist der ***Object-Request-Broker*** (***ORB***), welcher detailliert in der ***Common-Object-Request-Broker-Architecture*** (***CORBA***) [OMG91] spezifiziert ist. Der ORB ist hierbei die Kommunikations-Infrastruktur, die es verteilten Objekten ermöglicht miteinander zu kooperieren. Er kann als logischer *Software-Bus* angesehen werden. Das ***Objektmodell*** definiert die wesentlichen Konzepte der Objektorientierung. Bei den ***Common-Object-Services*** handelt es sich eine Menge von systemnahen Basisdiensten, welche die Funktionaltät des ORB ergänzen [OMG94]. Die ***Common-Facilities*** umfassen eine Menge höherwertiger Dienste. Diese zerfallen in *Horizontal*- und *Vertical Common Facilities.* Erstere beinhalten anwendungsneutrale Dienste, wie etwa der objektorientierte Zugriff auf Betriebsystemressourcen.

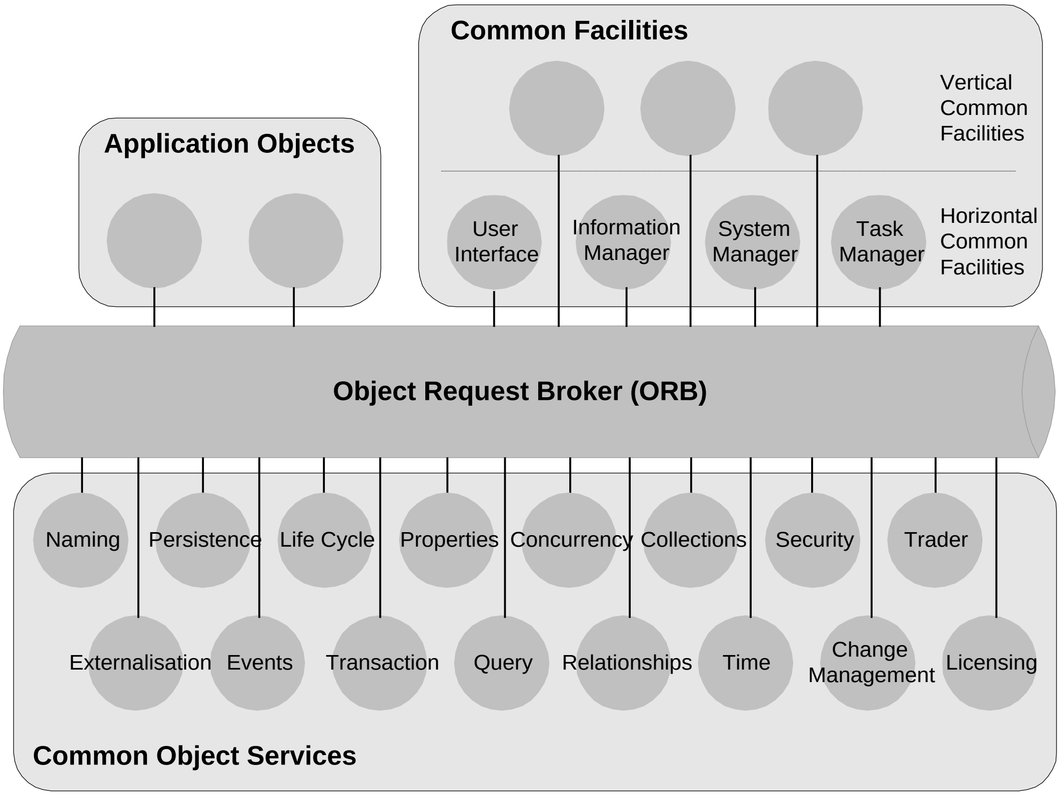


Abbildung 1 Die Object Management Architecture (OMA)

Vertical Common Facilities beinhalten Dienste, die zu bestimmten Anwendungsbereichen korrespondieren. Die ***Application-Objects*** bilden schließlich die anwendungsorientierte Funktionalität ab. Eine objektorientierte Anwendung basiert typischerweise auf einer großen Zahl von Objekten, welche sich in systemnahe, branchenspezifische und spezielle Anwendungs-Objekte klassifizieren lassen. Die Anwendungs-Objekte implementieren die spezielle Funktionalität einer Anwendung. Diese stützen sich auf Objekte ab, die bestimmte Problemlösungen eines Anwendungsbereiches, etwa der Versicherungswirtschaft, realisieren (*Vertical Common Facilities)*. Die systemnahen Objekte (*Common-Object-Services* und *Horizontal-Common-Facilities)* bilden schließlich die Basis der Anwendungsentwicklung. Diese komponenten-orientierte Softwareentwicklung läßt sich gut mit der Konstruktion von Hardware, bei der es schon lange üblich ist, Rechnersysteme aus standardisierten Komponenten zu assemblieren, vergleichen. So würde der ORB (*Software-Bus*) dem Systembus und die verteilten Objekte den vorkonfektionierten und speziellen IC-Bausteinen eines Rechners entsprechen. Die verteilten Objekte kooperieren über den ORB ähnlich wie die IC-Bausteine über den Systembus eines Rechners. Häufig fällt in diesem Zusammenhang auch der Begriff ***Software-IC*** [OHE96].

2.2.1 Das Objektmodell

Das Objektmodel, welches in der OMG-Terminologie als *Core-Object-Modell* bezeichnet wird, definiert die Konzepte der Objektorientierung der in einer ORB-Umgebung miteinander kooperierenden Objekte. Das Objektmodel basiert auf den Konzepten Objekt, Operationen, Objekttyp und Vererbung [OMG95]. Eine Objekttyp definiert den Zustandsraum und das Verhalten einer Menge von Objekten, welche als Instanzen des Objekttyps bezeichnet werden. Der Zustand eines Objektes ist gegeben durch die Werte seiner Attribute. Eine implizite Eigenschaft eines jeden Objektes ist dessen Identität, die unabhängig von dessen aktuellem Zustand ist. Diese Identität wird innerhalb einer ORB-Umgebung durch eindeutige Object-Identifier (OID) realsiert. OIDs werden in der OMG-Terminologie als Objekt-Referenzen bezeichnet. Das Verhalten eines Objektes wird durch eine Menge von Operationen bestimmt. Objekttypen können zueinander in Vererbungsbeziehung stehen. Ein Subtyp erbt dabei alle Attribute und Operationen seiner Supertypen (Mehrfachvererbung möglich). Ein Objekttyp beschreibt lediglich das Interface seiner Instanzen. Sein Interface wird in der programmiersprachenunabhängigen *Interface-Definition-Language* (*IDL*) definiert. Zu jedem Objekttyp kann es eine beliebige Anzahl von Implementierungen geben. Die Implementierungssprache spielt dabei keine Rolle. Diese strikte Trennung von Interface und Implementierung wird durch eine Reihe von Sprachanbindungen unterstützt. Das Interface dient dabei als Kontrakt zwischen kooperienden Objekten.

2.2.2 Common Object Services

Wie bereits dargestellt erweitern Common Object Services die Funktionalität des ORB [OMG94]. Die OMG definiert 16 dieser ORB-nahen Dienste, deren Interface in IDL angegeben sind. Bisher sind jedoch noch nicht alle 16 Common Object Services vollständig spezifiziert worden. Für alle diese Dienste gilt, daß es nicht Intention der OMG ist, das Rad neu zu erfinden. Vielmehr sollen standisierte Interfaces in IDL dazu beitragen, vorhandene Lösungen zu integrieren [OHE96]. An dieser Stelle soll die Funktionalität einige dieser Dienste skizziert werden:

- Der ***Naming-Service*** ist der prinzipielle Mechanismus, der es Objekten ermöglicht, andere am ORB angemeldete Objekte über einen Namen zu lokalisieren. Bestehende Naming- oder Directory-Services wie etwa ISO X.500, SUN NIS+ oder das Directory-Service von Novell könnten durch das *IDL-Naming-Service-Interface* transparent gekapselt werden. Die Assoziation zwischen einem Namen und einer Objekt-Referenz wird dabei als Name-Binding bezeichnet. Eine Name muß innerhalb eines Namens-Kontextes eindeutig sein. Namens-Kontexte sind hierarchisch organisiert. Der *Naming-Service* definiert Operationen zum Aufbau und Durchlaufen von Namens-Kontext-Hierarchien. Objekte können innerhalb eines Namens-Kontextes an einen Name gebunden werden. Über diesen gelangen wiederum andere Objekte an die entsprechende Objekt-Referenz.

- Der ***Event-Service*** realisiert einen verteilten Event-Handler. Objekte können Event-Arten spezifizieren und beim Event-Service anmelden. Andere Objekte können ihr Interesse an bestimmten Events beim Event-Service registrieren und werden benachrichtigt, wenn ein bestimmtes Event eingetreten ist.

- Objekte der realen Welt stehen in Beziehung zueinander. Das dynamische Verwalten von Beziehungen zwischen Objekten, die zuvor einander nicht kannten, wird durch den ***Relationship-Service*** unterstützt. Dieser unterstützt verschiedene Arten von Objekt-Beziehungen, wie z.B. Aggregations- oder Assoziations-Beziehungen. In Beziehung stehende Objekte werden auch als Objekt-Netze bezeichnet. Ein gutes Beispiel hierfür wäre etwa ein zusammengesetztes Dokument, welches aus einer Menge von Kapiteln, Literaturreferenzen, Abbildungen usw. besteht.

- Das Erzeugen, Kopieren, Verschieben und Löschen von Objekten wird durch die Operationen des ***Life-Cycle-Service*** unterstützt. Dieser stützt sich wiederum auf den *Relationship-Service* ab, so daß z.B. auch Deep-Copy-Semantik von komplexen Objekten (Objekt samt rekursiv aggregierten Objekten) unterstützt wird.

- Der ***Transaction-Service*** ermöglicht verteilte Transaktionen auf der Basis des ***Two-Phase-Commit-Protokolls*** [Date90]. Er unterstützt ein einfaches und ein geschachteltes Transaktionsmodell (siehe hierzu das Beispiel einer geschachtelten Transaktion im Kapitel 3.2.5). Vorgesehen ist auch die Einbeziehung von Datenbanksystemen und Transaktions-Monitoren, die das X/Open DTP-Protokoll (siehe [Rah94]) für verteilte Transaktionen unterstützen. Abbildung 2 stellt ein Beispiel einer verteilten Transaktion als Sequence-Diagramm dar. Im Beispiel soll ein Geldbetrag von einem Konto einer Bank (*Resource A*) auf ein Konto einer anderen Bank (*Resource B*) überwiesen werden.

  Ein Objekt kann in einer Transaktion eine von drei Rollen einnehmen. Die Rolle des ***transactional-client*** übernimmt das Objekt, welches die Transaktion startet. Objekte, deren Operationen innerhalb einer Transaktion aufgerufen werden, nehmen die Rolle eines ***transactional-servers*** oder eines ***recoverable-server*** ein. Ein ***recoverable-server*** ist ein Objekt, dessen Daten (oder Zustand) vom Verlauf der Transaktion abhängen. Objekte in der Rolle eines ***transactional-servers*** sind durch den Aufruf einer ihrer Operationen an der Transaktion beteiligt, haben allerdings keine zu schützenden Ressourcen, deren Zustand vom Ausgang der Transaktion abhängig ist. Sie haben aber dennoch einen Einfluß auf den Ausgang der Transaktion, da sie unter Umständen Operationen von *recoverable-server*-Objekten aufrufen. Ein ***transactional-client*** startet eine Transaktion durch den Aufruf der Operation *current::begin* (in Abbildung 2, Pfeil *(1)*). Dabei ist *current* ein mit dem ORB verbundenes Pseudo-Objekt mit den Operationen *begin*, *commit*, *rollback* und *get_control*. Ein Aufruf der Operation *current::begin* bewirkt, daß ein neuer Transaktions-Kontext und ein Objekt vom Typ *Coordinator* erzeugt wird. Das *current*-Objekt verwaltet die Assoziation zwischen dem *transactional-client*, dem Transaktions-Kontext und dem *Coordinator*-Objekt. Die Operation *current::get_control* liefert zu einem Transaktions-Kontext eine Referenz auf das korrespondierende *Coordinator*-Objekt, welches die Commit-Phase der Transaktion koordiniert. Ruft der ***transactional-client*** nun innerhalb der Transaktionsgrenzen (definiert durch *begin*, *commit / rollback*) Operationen anderer Objekte auf (in Abbildung 2, Pfeile *(2)* und *(5)*), so wird der Transaktions-Kontext als impliziter Parameter des Operations-Aufrufes mitübergeben. Wird nun eine Operation eines *recoverable-server* aufgerufen, so meldet sich dieser beim *Coordinator*-Objekt durch die Operation *Coordinator::register_resource* an (in Abbildung 2, Pfeile *(4)* und *(7)*).

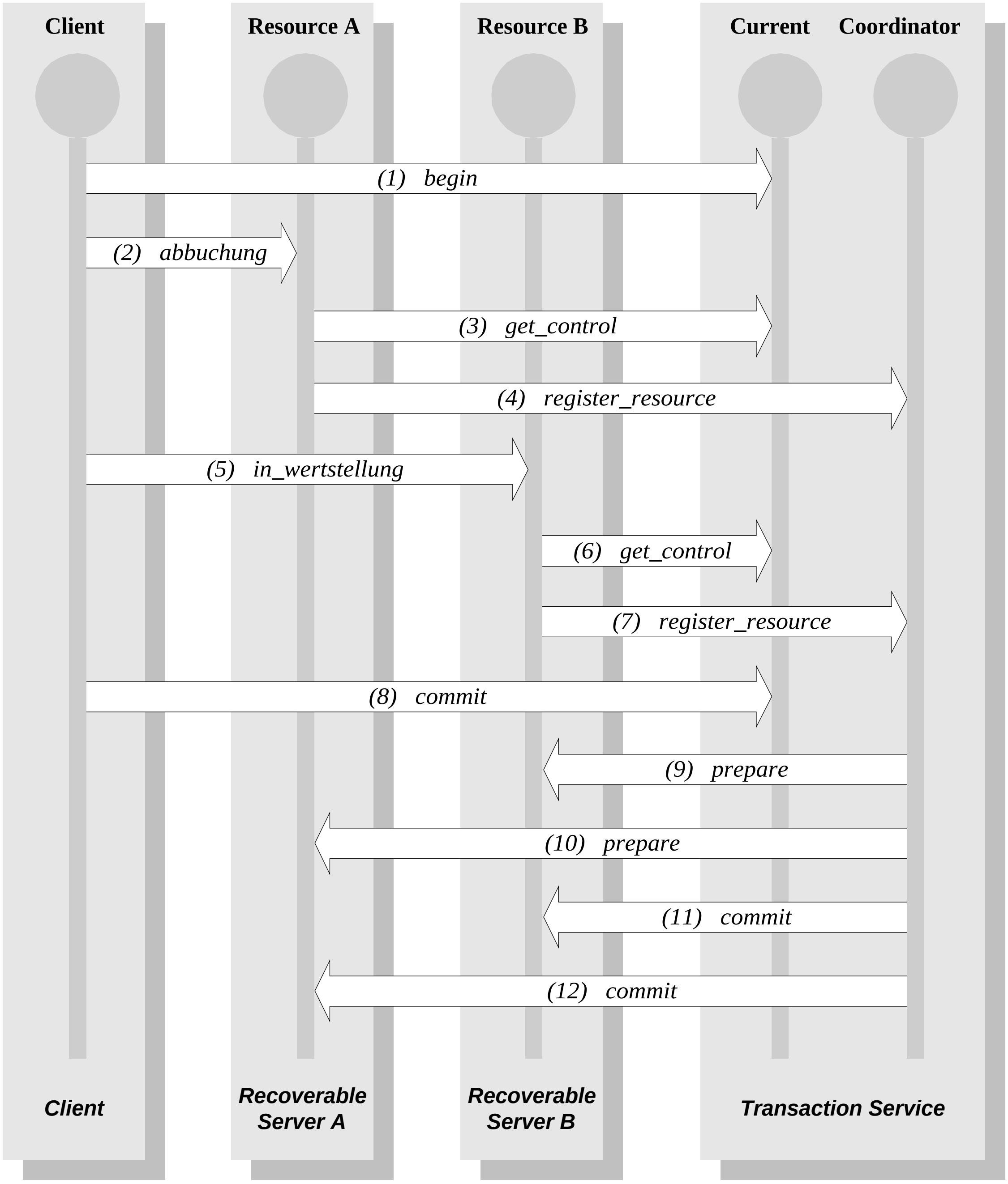


Abbildung 2 Sequence-Diagramm einer verteilten Transaktion

Die Referenz des zuständigen *Coordinator*-Objektes erhält er mittels *current::get_control* (in Abbildung 2, Pfeile *(3)* und *(6)*). Dem Koordinator der Transaktion werden somit alle an der verteilten Transaktion beteiligten *recoverable-server* bekannt gemacht. Die Commit-Phase wird schließlich durch den *transactional-client* initiiert (in Abbildung 2, Pfeil *(8)*). Das *Coordinator*-Objekt koordiniert daraufhin das Ende der Transaktion entsprechend dem bekannten *Two-Phase-Commit-Protokoll* (in Abbildung 2, Pfeile *(9)* bis *(12)*). In Phase 1 (*Prepare-Phase*) befragt er alle an der verteilten Transaktion beteiligten Server, ob sie ihre Subtransaktionen erfolgreich abschließen können. Die Server antworten mit *vote_commit* im positiven Fall und ansonsten mit *vote_rollback*. Anwortet ein Server mit *vote_commit*, so verpflichtet er sich, die Modifikationen der entsprechenden Subtransaktion auf jeden Fall festschreiben zu können. Erhält der Koordinator von allen beteiligten Servern als Antwort *vote_commit*, so teilt er ihnen in Phase 2 (*Commit-Phase*) mit, ihre Subtransaktionen festzuschreiben.

2.3 Common-Object-Request-Broker-Architecture (CORBA)

Der *Object-Request-Broker* (*ORB*) ist die zentrale Komponente in der *Object Management Architecture* (*OMA*) der OMG. Die *Common-Object-Request-Broker-Architecture* (*CORBA*) definiert die Funktionsweise eines ORBs im Detail. Innerhalb der OMG gab es ursprünglich zwei Ansätze für eine ORB-Architektur. Der eine basierte auf einem statischen, der andere auf einem dynamischen API (Application Programing Interface). CORBA vereint schließlich beide Ansätze. Hierfür steht das „*Common*“ in CORBA [OHE96].

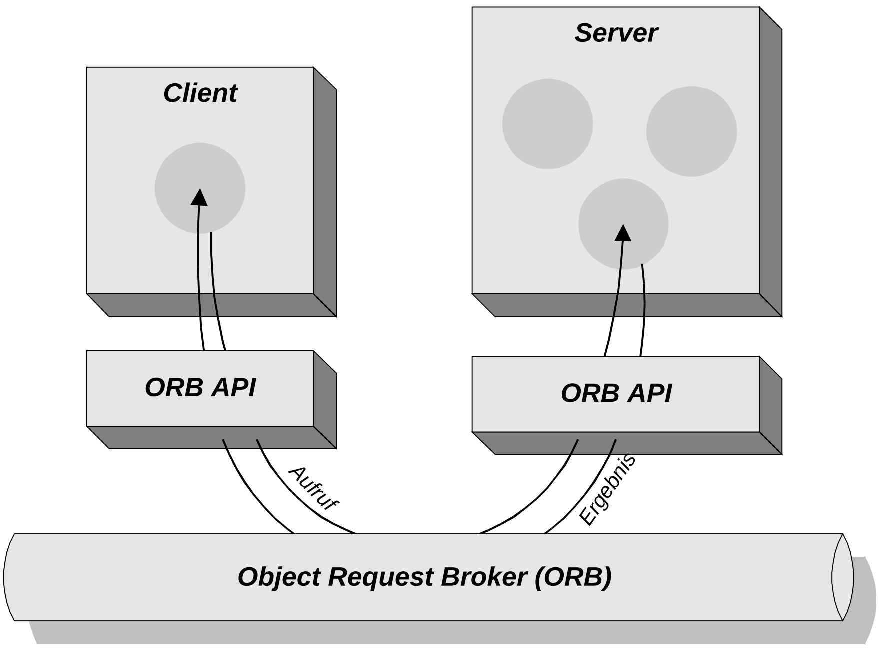


Abbildung 3 Der ORB als Vermittler von Client- und Server-Objekten

Die Aufgabe eines ORB besteht darin, die Kooperation von verteilten, möglicherweise heterogenen Objekten, zu standardisieren. CORBA stellt also einen *Middelware*-Ansatz für Objekte dar. Ein ORB stützt sich dabei auf die Dienste eines vorhandenen Netzwerkprotokolls ab. Der durch einen ORB realisierte Dienst würde der Awendungsschicht (Schicht 7) des ISO-OSI-Referenzmodells [Tan89] zugeordnet werden. Objekte in CORBA kooperieren miteinander, indem sie Operationen anderer Objekte aufrufen, um deren Zustand zu erfragen oder zu modifizieren. Innerhalb einer solchen Kooperationsbeziehung wird das aufrufende Objekt als *Client*-Objekt (Auftraggeber), das Objekt, dessen Operation aufgerufen wurde, als *Server*-Objekt (Auftragnehmer) bezeichnet (siehe Abbildung 3). Diese Rollenverteilung ist jedoch nicht statisch, sondern charakterisiert lediglich Objekte innerhalb einer bestimmten Kooperationsbeziehung. Es handelt sich somit um eine Verallgemeinerung des Client-Server-Konzeptes. Die Betriebssystem-Prozesse, in deren Adreßraum sich die *Client*- bzw. *Server*-Objekte befinden, werden als *Client(-Prozesse)* bzw. *Server(-Prozesse)* bezeichnet (siehe Abbildung 3).

Die Systemumgebung (Hardwareplattform, Betriebssystem, Implementierungssprache) und die Lokalität (Rechner, Prozeß) eines *Server*-Objektes ist für das *Client*-Objekt völlig transparent. Der ORB transformiert die Parameter und Rückgabewerte eines Operationsaufrufes entsprechend der Systemumgebungen der involvierten Objekte und dirigiert diesen zum richtigen Objekt. Alles was das Client-Objekt hierzu benötigt, ist eine Referenz auf das Server-Objekt (*Object-Indentifier* (*OID*)) und das Wissen über dessen Interface.

2.3.1 Die Komponenten von CORBA

Um die Funktionsweise eines ORB wirklich verstehen zu können, ist es notwendig, dessen Architektur einmal detaillierter zu betrachten. Abbildung 4 zeigt alle Komponenten von CORBA, die in einer Kooperationsbeziehung zwischen Client- und Server-Objekt beteiligt sind. Auf der linken Seite von Abbildung 4 befinden sich die Komponenten der Client-Seite, auf der rechten die der Server-Seite. Zwischen Client bzw. Server und dem ORB sind die Komponenten angeordnet, die die Sprachanbindung an die Implementierungsumgebung der Client- bzw. Server-Objekte leisten. CORBA beinhaltet zwei unterschiedliche Konzepte für den Aufruf einer Server-Operation. Bei der statischen Variante kennt das Client-Objekt das Interface des Server-Objektes bereits zur Übersetzungszeit (*static method invocation*). Für den dynamischen Aufruf einer Server-Operation (*dynamic method invocation*) stellt CORBA ein spezielles API zur Verfügung, mit dem Server-Interface-Beschreibungen zur Laufzeit gelesen und Operations-Aufrufe konstruiert werden können. Im folgenden soll die Funktion der einzelnen Komponenten von CORBA skizziert werden:

- Bevor ein Objekt überhaupt mit anderen Objekten kooperieren kann, muß es zunächst eine Verbindung zum ORB herstellen. Hierzu ruft es die Funktion *CORBA_ORB_init* der ***ORB-API*** auf und erhält ein ORB-Pseudo-Objekt mithilfe dessen der Client nun Zugriff auf einige wohlbekannte Server-Objekte (Common Object Services) wie z.B. dem *Naming-Service* hat.

- Für den statischen Operations-Aufruf benötigt ein Client-Objekt einen ***Client-Stub*** des entsprechenden Server-Objektes. Der *Client-Stub* ist dabei die lokale Stellvertreter-Instanz (*Proxy*) des Server-Objektes. Zu jeder Objekt-Referenz wird zur Laufzeit ein solches *Proxy*-Objekt im Client erzeugt. Der eigentliche Operations-Aufruf wird über den *Client-Stub* getätigt. Dieser enthält den nötigen Code, um den Operations-Aufruf, dessen Parameter und Rückgabewert in eine Nachricht zu kodieren und zu dekodieren. Diese wird dann von dem ORB-Kern zum Server weitergeleitet. Ein *Client-Stub* stellt also das Interface eines Server-Objektes in der Implementierungssprache des Client-Objektes dar. Der *Stub-Code* wird durch einen IDL-Compiler, der Bestandteil jeder CORBA-Implementierung ist, generiert und muß zusammen mit dem Client-Code kompiliert werden. Der IDL-Compiler setzt die IDL-Definition des Server-Interfaces in die Zielsprache des Clients um. In Abbildung 4 ist der Ablauf eines statischen Operations-Aufrufes durch die dicken durchgezogenen Pfeile dargestellt.

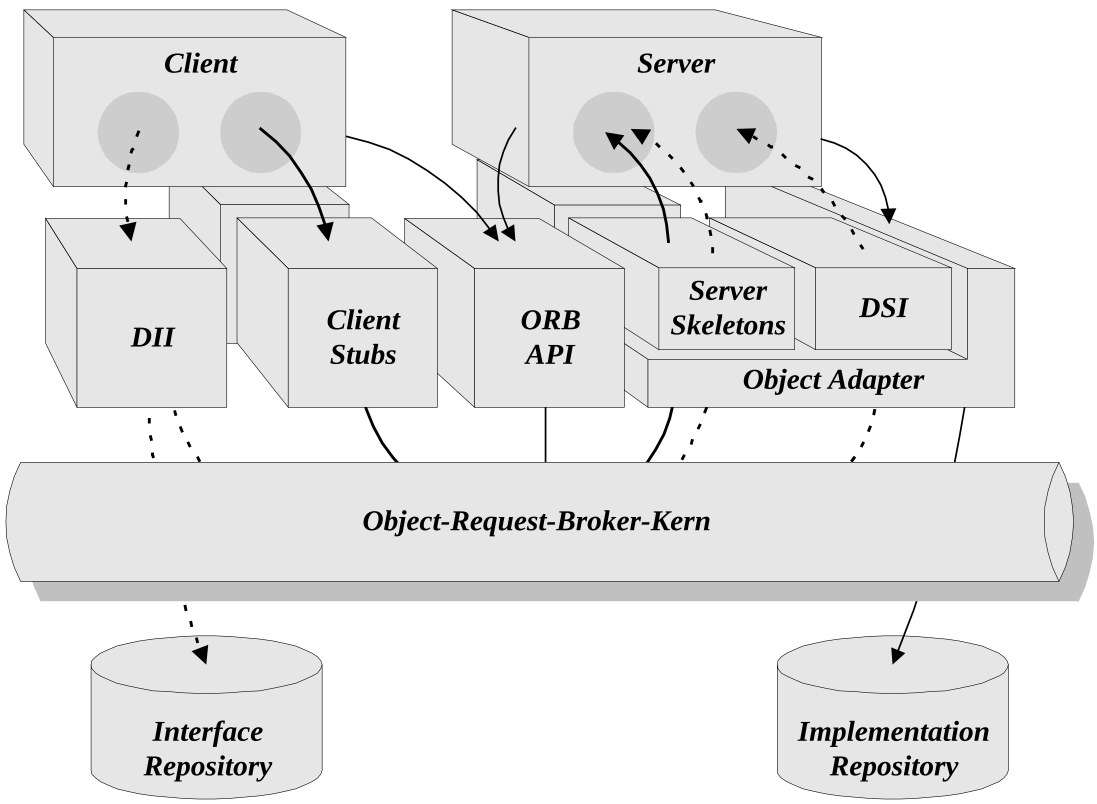


Abbildung 4 Die Common-Object-Request-Broker-Architecture im Detail

- CORBA ermöglicht es, Operationen von Server-Objekten aufzurufen, deren Interface dem Client nicht bekannt sind. Das ***Dynamic-Invocation-Interface*** (***DII***) beinhaltet die dazu notwendige Funktionalität. Die IDL-Interface-Beschreibungen von Server-Objekten sind in maschinenlesbarer Form im ***Interface-Repository*** abgelegt. Das *DII* realisiert einen navigierenden Zugriff auf diese Interface-Beschreibungen und enthält Methoden zur Konstruktion (*create_request(Object_Reference, Method, Argument_List)*) und Durchführung von Operations-Aufrufen. Dynamische Operations-Aufrufe können synchron oder asynchron getätigt werden. Die Methode *invoke* führt die Operation in RPC-Semantik aus (Remote Procedure Call). Die Kontrolle kehrt erst dann wieder zum Programm zurück, wenn das Ergebnis der Operation vorliegt. Mittels der Methoden *send* und *get_response* / *get_next_response* können Operationen asynchron aufgerufen werden. In Abbildung 4 ist die Interaktion der an einem dynamischen Operations-Aufruf beteiligten Komponenten durch die dicken gestrichelten Pfeile dargestellt.

- Die ***Server-Skeletons*** sind das Äquivalent zu den *Client-Stubs* auf Server-Seite. Sie werden ebenfalls vom IDL-Compiler in der Implementierungssprache des Server generiert. Ein *Server-Skeleton* dekodiert die Nachricht, die einen Server-Operation-Aufruf samt Parametern enthält, und ruft schließlich die entsprechende Operation der Server-Objekt-Implementierung auf (*Up-Call*). Ein Server-Objekt, dessen Dienst anderen Objekten zur Verfügung gestellt werden soll, wird mitttels der ***bind***-Funktion an ein *Server-Skeleton* und einen *Object-Identifier* (Objekt-Referenz) gebunden. Dieser Vorgang wird aus der Perspektive des ORB als Instanziierung bezeichnet. Ein *Server-Skeleton* ist sowohl bei statischen als auch bei dynamischen Operations-Aufrufen involviert (durchgezogene und gestrichelte Pfeile in Abbildung 4).

- Das ***Dynamic-Skeleton-Interface*** (***DSI***) ermöglicht es Server-Objekten, für die kein IDL-basiertes *Skeleton* existiert, ihren Dienst in der ORB-Umgebung dynamisch zu registrieren. Sie definieren dazu ihr Interface quasi *„on the fly"*. Die entsprechende Interface-Beschreibung wird hierbei im *Interface-Repository* abgelegt (siehe oben). Das DSI hat darüber hinaus die Aufgabe, hereinkommende Operations-Aufrufe (von Server-Objekten ohne *Skeleton*) an die adäquate Server-Objekt-Implementierung weiterzuleiten (*Up-Call*).

- Der ***Object-Adapter*** ist das Laufzeitsystem eines jeden Server-Prozesses. Die OMG definiert einen Standard-*Object-Adapter* (*Basic-Object-Adapter* (*BOA*)), den jede CORBA-Implementierung unterstützen muß. Der *BOA* registriert alle Server-Objekte und deren Objekttypen (durch einen Verweis auf die korrespondierende Interface-Beschreibung im *Interface-Repository*) innerhalb des Server-Prozesses im ***Implementation-Repository*** und macht sie somit in der ORB-Umgebung verfügbar. Bei der Instanzierung neuer Server-Objekte generiert er neue *Object-Identifier* (Objekt-Referenzen). Ankommende Anfragen werden vom *BOA* transfomiert und an die entsprechenden *Server-Skeletons* oder an das *DSI* weitergeleitet. Dabei werden beispielsweise Objekt-Referenzen von einer ORB-spezifischen in eine implementierungs-spezifische Repräsentation transfomiert. Aus der Sicht eines Anwendungsentwicklers realisiert der *BOA* die Hauptkontrollschleife eines Server-Prozesses, die mittels ***BOA::run*** angestoßen und durch ***BOA::shutdown*** abgebrochen wird. Diese Hauptkontrollschleife sorgt für die Aktivierung der Server-Objekt-Operationen.

2.3.2 Das Inter-ORB-Protokoll (IOP)

Das Inter-ORB-Protokoll definiert die Kommunikation zwischen ORBs verschiedener Hersteller. CORBA 1.2 ermöglichte die Portabilität von verteilten objektorientierten Systemen. Erst mit CORBA 2.0 und dem darin festgeschriebenen Inter-ORB-Protokoll wird wirkliche Interoperabiltät von verteilten Objekten erreicht. Objekte, die in verschiedenen ORB-Umgebungen residieren, können nun miteinander kooperieren.

Das Inter-ORB-Protokoll umfaßt eine Reihe von Protokollen, welche die Kommunikation zwischen verschiedenen ORBs festlegen. Das *General-Inter-Orb-Protokoll* (*GIOP*) definiert eine Reihe von Nachrichten-Typen und Datentyp-Repräsentationen, wie z.B. die *Interoperable-Object-References* (*IOR*). Das *Internet-Inter-ORB-Protokoll* (*IIOP*) ist obligatorisch für jede CORBA 2.0 Implementierung und definiert, wie die im *GIOP* definierten Nachrichten-Typen über ein TCP/IP-Netzwerk ausgetauscht werden. Neben dem *IIOP* sieht das *IOP* noch andere Protokolle vor, welche die Kooperation von ORBs, die durch andere Netzwerke als TCP/IP miteinander verbunden sind. Diese Protokolle werden als umgebungsspezifische-Inter-ORB-Protokolle (Environment-Specific-Inter-ORB-Protocols (ESIOP)) bezeichnet.

Abbildung 5 zeigt die Architektur einer möglichen *IOP*-Implementierung. Zwei ORBs kommunizieren dabei über eine Brücke miteinander. Alle Operations-Aufrufe innerhalb der ORB-Umgebung A, die sich auf Server-Objekte in der ORB-Umgebung B beziehen, werden über das Brücken-Objekt ***a*** mithilfe des *Dynamic-Skeleton-Interface* (*DSI*) weitergeleitet. Das Brücken-Objekt ***b*** ruft dann durch das *Dynamic-Invocation-Interface* (*DII*) die entsprechenden Operationen auf. Diese Late-Binding-Technik eignet sich auch sehr gut um Gateways zu ORB-ähnlichen System, wie etwa Microsofts OLE/COM, zu realisieren.

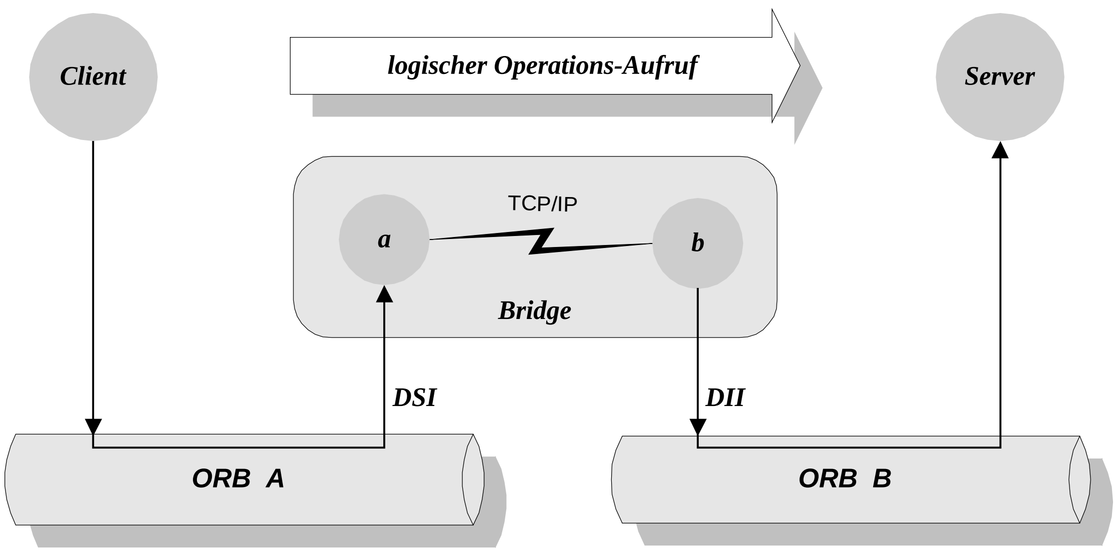


Abbildung 5 Architektur einer möglichen *IOP*-Implementierung

2.4 Die Interface Definition Language

Das wesentliche Basis-Konzept von CORBA ist die strikte Trennung der Spezifikation eines Dienstes von dessen Implementierung. Die Spezifikation eines Dienstes wird in CORBA durch die *Interface Definition Language* (IDL) vorgenommen. Dies ermöglicht zum einen die Wiederverwendung von Software-Komponenten, die in unterschiedlichen Programmiersprachen und für verschiedene Plattformen entwikkelt wurden. Schon heute existiert ein großer Markt für Software-Komponenten (Componentware) [Ho96]. Darüber hinaus ist dieser Ansatz hervoragend für die Systemintegration geeignet. Bestehende Systeme können nachträglich mit einem IDL-Interface versehen werden („***ID**ea**L**isierung*“) und in eine ORB-Umgebung eingebunden werden.

IDL ist eine deklarative programmiersprachenunabhängige Sprache zur Interface-Definition von verteilten Objekten, deren Syntax auf ANSI C++ basiert. Ein IDL-Compiler, der Bestandteil jeder CORBA-Implementierung ist, generiert aus den IDL-Interface-Beschreibungen die entsprechenden Sprachanbindungen (siehe Abschnitt 2.3.1 ***Client-Stub*** und ***Server-Skeletons***). Ein IDL-Interface beschreibt die Methoden (Operationen) und Attribute eines Objekttyps (siehe Objektmodell in Abschnitt 2.2.1). Interface-Beschreibungen sind eingebunden in einen Namens-Kontext, der deren Namens-Raum hierarchisch erweitert (siehe Abbildung 6). Die Methoden eines Objekttyps werden durch die Angabe ihrer Signatur definiert. Diese umfaßt den Namen der Methode, die formalen Parameter, den Datentyp des Rückgabewertes und die Angabe der möglichen Ausnahmen.

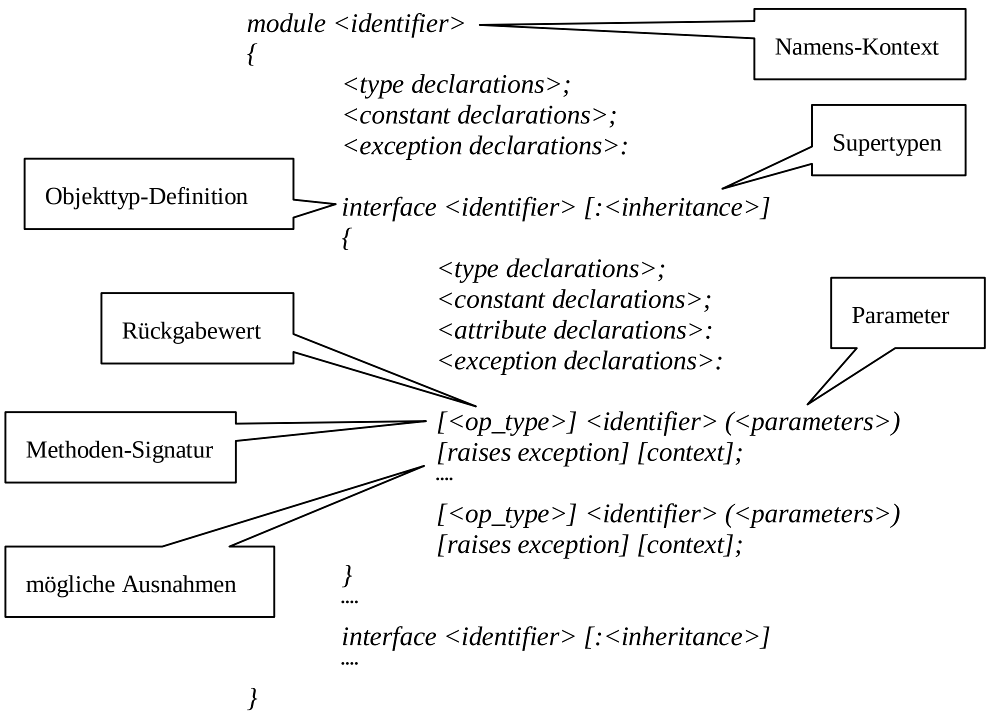


Abbildung 6 IDL-Syntax

Ein Parameter ist definiert durch seinen Datentyp und seinen Modus. Der Modus bestimmt die Richtung des Parameters beim Aufruf der Server-Operation. **In**-Parameter werden vom Client (Aufrufer) zum Server, **Out**-Parameter in die Gegenrichtung und **InOut**-Parameter bidirektional übertragen. Methoden, die keinen Rückgabewert liefern und keinem **Out**-Parameter bzw. **InOut**-Parameter haben, können als asynchrone Operationen durch das Voranstellen von ***oneway*** definiert werden.

Als Wertebereiche für Attribute, Parameter und Rückgabewerte dienen eine Menge von atomaren Datentypen (*short, long, unsigned short, unsigned long, float, double, char, octet, boolean*) und mittels Typ-Konstruktoren definierte komplexe Datentypen (*string, enum, struct, union, array, sequence, any*). Ein durch „*interface* ...“ definierter Objektyp ist ebenfalls ein zulässiger Datentyp. Der spezielle *any*-Typ-Konstruktor ermöglicht das dynamische Konstruieren beliebiger Datenstrukturen. Einem *any*-Element können Objekte aller zulässigen IDL-Datentypen (einschließlich aller komplexen Datentypen) zugewiesen werden.

Kapitel 3 Einführung in den Themenbereich „Föderierte Datenbanksysteme“

Ziel dieses Kapitels ist es, die wesentliche Konzepte und Begriffle des Themenbereichs „Föderierte Datenbanksysteme“ darzustellen.

3.1 Grundlagen

Ein föderiertes Datenbanksystem ist nach [SL90] ein System bestehend aus mehreren kooperierenden, autonomen Datenbanksystemen, welches seinen Benutzern einen homogenisierten Zugriff auf die darin enthaltenen Datenbestände bietet. Die Software, die die kontrollierte Kooperation der beteiligten autonomen und häufig auch heterogenen Datenbanksysteme realisiert, wird als föderiertes Datenbankmanagementsystem (FDBMS) bezeichnet. Abbildung 7 zeigt die Grobarchitektur eines föderierten Datenbanksystems und stellt die wichtigsten Begriffe im Zusammenhang dar.

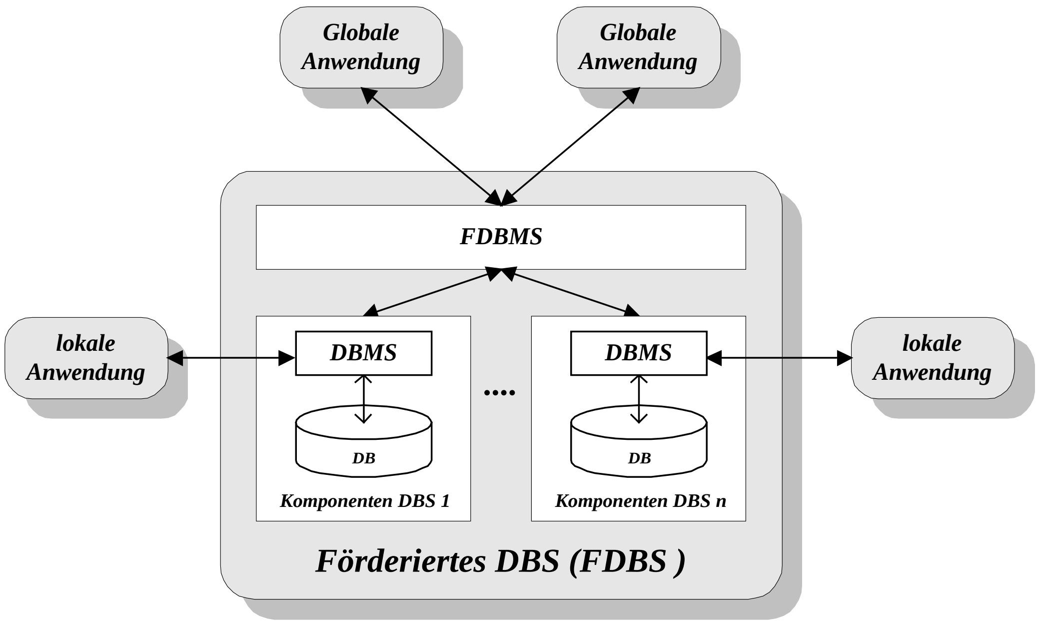


Abbildung 7 Grobarchitektur eines föderierten Datenbanksystems

Die an der Föderation beteiligten Datenbanksysteme (DBS) werden als Komponenten-DBS bezeichnet (KDBS) und bewahren im Rahmen der Föderation weitgehend ihre Autonomie. Die bestehenden Anwendungen (legacy-systems) können weiterhin auf die lokalen Komponenten-DBS zugreifen, ohne modifiziert werden zu müssen. Bereits getätigte Investitionen in Informationssysteme bleiben somit gesichert. Die bestehenden Anwendungen werden fortan als lokale Anwendungen bezeichnet.

Neue Anforderungen an die Informationsverarbeitung in einer Organisation, welche sich auf eine Menge von Daten beziehen, die in unterschiedlichen Datenhaltungssystemen vorgehalten werden, können mit Hilfe eines föderierten Datenbanksystems wesentlich leichter realisiert werden. Die daraus resultierenden Anwendungen werden als globale Anwendungen bezeichnet.

3.2 Autonomie und Heterogenität

Wie bereits im letzten Abschnitt deutlich wurde, ist die Bewahrung der Autonomie der Komponenten-Datenbanksysteme ein wichtiger Aspekt bei deren Integration. Die Autonomie der einzelnen Subsysteme ist in großen Organisationen von immenser Bedeutung. Schließlich sollen die Komponenten-Datenbanksysteme auch nach der Integration weiterhin primär den Zweck erfüllen, zu dem sie ursprünglich eingesetzt wurden. In [Rah94] werden fünf Aspekte von Autonomie beschrieben, die auf die Entwicklung eines FDBS erheblichen Einfluß haben:

- Die *Entwurfsautonomie* beinhaltet, daß jedes Komponenten-Datenbanksystem bzw. dessen Administratoren oder Abteilungen selbständig darüber entscheiden, wie der für sie relevante Realweltausschnitt (Diskurswelt) auf ein Datenbankschema abgebildet wird. Dies betrifft sowohl den logischen als auch den physischen Datenbankentwurf.

- Die *Ausführungsautonomie* besagt, daß die Ausführung von lokalen Transaktionen nicht durch die Integration eingeschränkt werden darf. Subtransaktionen globaler Transaktionen (Transaktionen, die zu Kooperationszwecken dienen) sollten möglichst wie lokale Transaktion ausgeführt werden.

- Die Komponenten-Datenbanksysteme bzw. deren Administratoren sollten selbständig über die Vergabe von Zugriffsrechten entscheiden. Dieser Aspekt wird als *Kooperationsautonomie* bezeichnet und ist nicht unerheblich in Hinblick auf den Datenschutz.

- Die *Unabhängigkeit bei der Wahl des DBMS* stellt sicher, daß die Kombination DBMS und Datenbank (DBS = DBMS + DB) den Anforderungen einer Abteilung oder eines bestimmten Anwendungsbereiches optimal gerecht wird. Die *Wahl des DBMS* impliziert zugleich die Wahl des zugrundeliegenden Datenmodells (Relational, Objektorientiert, ...), der Datenbankanfragesprache usw.

- Ebenso wie der vorherige Aspekt hat die *Unabhängigkeit bei der Wahl der Ablaufumgebung* (Hardware, Betriebssystem, Nertzwerkprotokoll) einen Einfluß auf die optimale Funktion des KDBS in Hinblick auf den primären Anwendungszweck.

Die oben beschriebenen Autonomie-Aspekte implizieren natürlich gewichtige Probleme bei der Integration der KDBS. Die Kernproblematik bei der Entwicklung eines föderierten Datenbanksystems besteht in der Überwindung der Heterogenität der beteiligten Komponenten-DBS, die eine direkte Folge deren Autonomie ist. Nach [Rah94] lassen sich in diesem Zusammenhang im wesentlichen drei Aspekte von Heterogenität unterscheiden:

- *Heterogenität bezüglich der verschiedenen DBMS* der Komponenten-DBS. Diese können sich hinsichtlich Hersteller, Version, dem zugrundeliegenden Datenmodell (Hierarchisch, Relational, Objektorientiert) und der zur Verfügung gestellten Anfragesprache (Codasyl, SQL, OQL) unterscheiden.

- *Heterogenität in der Ablaufumgebung*. Die einzelnen KDBS können auf verschiedenen Hardwareplattformen, Betriebssystemen und innerhalb verschiedener Kommunikationsnetze ablaufen.

- Die s*emantische Heterogenität* resultiert aus der Entwurfsautonomie der an der Föderation beteiligten KDBS. Semantisch gleichwertige Objekte werden unterschiedlich benannt oder repräsentiert. Diese Schemakonflikte aufzulösen ist Aufgabe der Schemaintegration.

Bei der Entwicklung eines föderierten Datenbanksystems stellt die Bewahrung der Autonomie der Komponenten-Datenbanksysteme kein Alles-oder-Nichts-Kriterium dar. Abhängig vom konkreten Systemzweck muß ein geeigneter Kompromiß zwischen Autonomie und dem notwendigen Kooperationsumfang der KDBS gefunden werden. Der nächste Abschnitt geht auf diese Problematik ein und skizziert unterschiedliche Lösungsansätze für die Integration verschiedener Datenquellen.

3.3 Einordnung von föderierten Datenbanksystemen

Neben dem FDBS-Ansatz existiert eine große Anzahl weitere Ansätze mit dem Ziel, unterschiedliche Datenbestände innerhalb einer Organisation zu integrieren. Der nun folgende Abschnitt soll die Eigenschaften und Ziele von föderierten Datenbanksystemen in Relation zu anderen Datenbanksystemen oder Ansätzen näher beleuchten. Darüber hinaus werden zwei unterschiedliche Systemkonzepte für föderierte Datenbanksysteme dargestellt.

In [SL90] wird eine Taxonomie von Mehrrechnerdatenbanksystemen beschrieben, die auf der Knoten-Autonomie der beteiligten Systeme basiert (siehe Abbildung 8). Hierin werden *föderierte Datenbanksysteme* als Teilmenge der *Multidatenbanksysteme* eingeordnet.

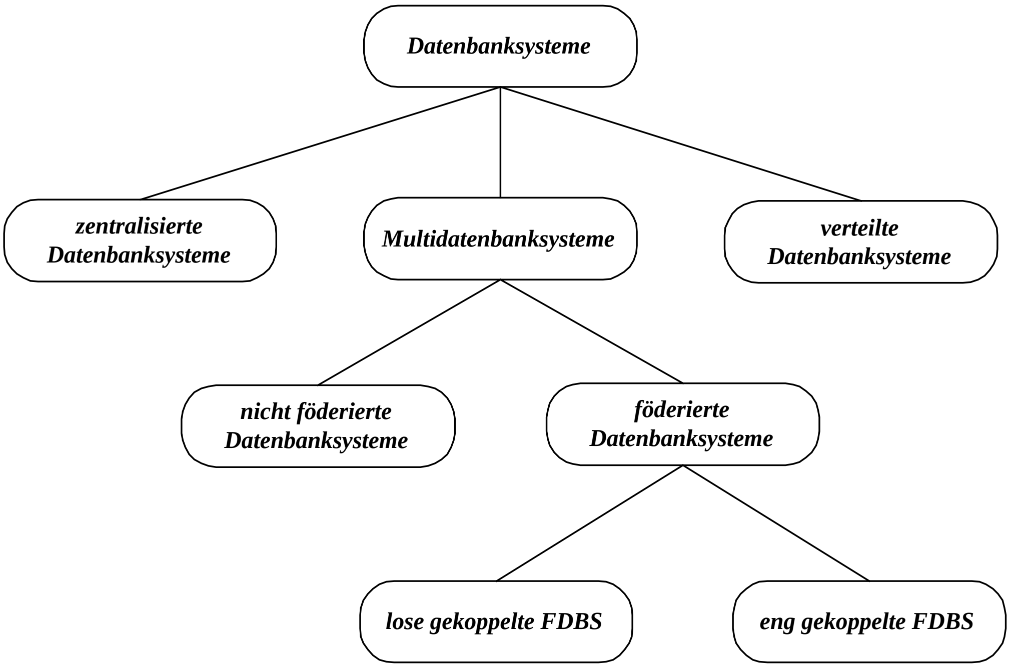


Abbildung 8 Taxonomie von Multidatenbanksystemen nach [SL90]

Ein *Multidatenbanksysteme* (*MDBS*) unterscheidet sich von *zentralisierten* und *verteilten Datenbanksystemen* dadurch, daß es Operationen auf einer Menge von möglicherweise heterogenen Komponenten-Datenbanksystemen (n * (DBMS + DB)) unterstützt. Ein *zentralisiertes Datenbanksystem* hingegen besteht aus einen Datenbankmanagementsystem (DBMS) und einer Datenbank (DB), die sich auf dem selben Rechnersystem befinden, während ein *verteiltes Datenbanksystem* aus meheren Datenbanken und einem DBMS, das über mehrere Rechnerknoten verteilt ist, besteht. Die *nicht föderierten Multidatenbanksysteme* unterscheiden sich von den *föderierten Datenbanksysteme* dadurch, daß ihre Komponenten-Datenbanksysteme nicht autonom sind. Der Datenzugriff ist bei diesen Systemen nur über eine einzige Schnittstelle (MDBMS) möglich. Im Gegensatz zu föderierten Datenbanksystemen kennen diese Systeme keine lokalen Anwendungen oder Benutzer (siehe Abbildung 7). *Föderierte Datenbanksysteme* stellen also einen Kompromiß zwischen totaler Integration, bei der die beteiligten Komponenten-Datenbanksysteme ihre Autonomie vollständig aufgeben, und isolierten Datenbanksystemen (maximale Autonomie) dar.

Wie Abbildung 8 zeigt, zerfallen die *föderierten Datenbanksysteme* in zwei Kategorien, die als ***lose*** bzw. ***eng gekoppelte FDBS*** bezeichnet werden. Ein *lose gekoppeltes FDBS* ermöglicht seinen Benutzern einen einheitlichen Zugriff auf die KDBS mittels einer Multidatenbanksprache, integriert deren konzeptuelle Schemata jedoch nicht zu einem globalen Schema. Für den Benutzer ist die Verteilung der Daten also weiterhin voll sichtbar und ihm bleibt die Aufgabe, sich selber ein integriertes Schema zu schaffen. *Lose gekoppelte FDBS* stellen einen guten Lösungsansatz für das Information-Retrieval (IR) (siehe [Fuhr96]) auf heterogenen Datenquellen dar.

Eng gekoppelte FDBS streben einen höheren Grad von Integration auf Kosten der Autonomie der KDBS an. Ein föderiertes (globales) Schema soll eine einheitliche Sicht auf alle Daten des FDBS gewährleisten und deren Verteilung transparent machen. Auf föderierte Schemata wird im weiteren Verlauf dieses Kapitels noch näher eingegangen.

Eine weitere Einordnung von föderierten Datenbanksystem wird in [Rah94] vorgenommen (siehe Abbildung 9). Unterschiedliche Arten von Datenbanksystemen werden durch die Begriffe Verteilungstransparenz, Kooperationsumfang, Knoten-Autonomie und Heterogenität zueinander in Relation gesetzt. In Abbildung 9 läßt sich sehr gut die Abhängigkeit dieser Begriffe zueinander ablesen.

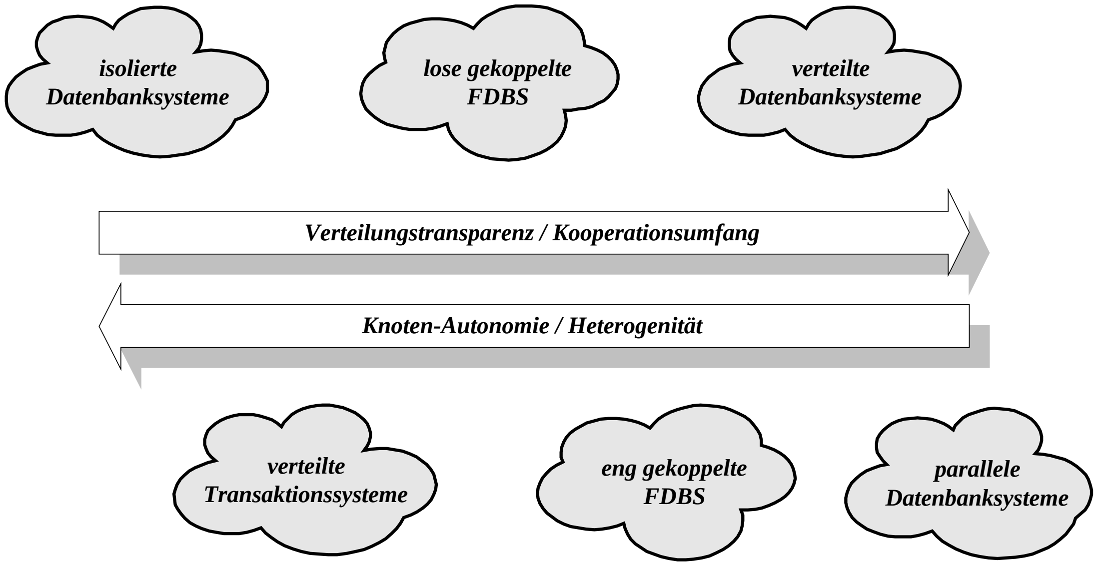


Abbildung 9 Einordnung von FBDS nach [Rah94]

Je höher der Grad der erreichten Verteilungstransparenz, desto höher ist auch der dazu nötige Kooperationsumfang. Dieser schränkt natürlich die Autonomie der beteiligten Komponenten-Datenbanksysteme ein. Isolierte Datenbanksysteme bewahren ihre volle Autonomie und bieten demzufolge überhaupt keine Verteilungstransparenz. *Verteilte Transaktionssysteme* ermöglichen die *programmierte Verteilung* von Daten mit Hilfe von *Transaktions-Monitoren* [Rah96]. Die *Transaktions-Monitore* koordinieren hierbei die Ausführung einer verteilten Transaktion gemäß einem Transaktions-Protokoll (etwa dem X/Open DTP). Die beiteiligten KDBS müssen hierzu bestimmte Schnittstellen bereitstellen, welche die Kooperation zwischen *Transaktions-Monitor* und KDBS ermöglichen. *Verteilte Transaktionssysteme* stellen eine Alternative zum FDBS-Ansatz dar, wenn der gewünschte Kooperationsumfang der KDBS begrenzt ist.

3.4 Eine Referenz-Architektur für föderierte Datenbanksysteme

In [SL90] wird eine Referenz-Architektur für föderierte Datenbanksysteme vorgestellt, die sich als Grundlage für Vergleiche unterschiedlicher FDBS-Ansätze etabliert hat. Die Architektur beinhaltet in Anlehnung an die dreischichtige ANSI/SPARC-Referenz-Architektur für Datenbanksysteme [Date90] ein 5-Schema-Ebenen-Modell. Bevor nun auf dieses Modell eingegangen wird, erscheint es sinnvoll die Begriffe Datenbank-Schema und Datenmodell zu definieren:

- Ein **Datenmodell** ist eine Sammlung von Konzepten zur Datenbeschreibung und Datenstrukturierung. Das Datenmodell bestimmt die Syntax und Semantik von Datenbankschemata [Fuhr95]. Häufig umfaßt ein Datenmodell eine Datenbanksprache (SQL, OQL+OML). Beispiele für Datenmodelle sind das relationale Modell und das ODMG-93-Objektmodell. Semantische oder Meta-Datenmodelle (z.B. das Entity-Relationship-Modell) dienen der Datenmodellierung auf abstrakterem Niveau und stellen Verallgemeinerungen anderer Datenmodelle dar.

- Ein **Datenbank-Schema** bestimmt die Struktur der Daten innerhalb einer Datenbank (DB) gemäß dem zugrundeliegenden Datenmodell. Zusätzlich zur Strukturbeschreibung der Daten beinhaltet es Integritätsbedingungen (Einschränkung der zulässigen DB-Zustände und Zustandübergänge) und die Definition von Zugriffsrechten.

Das 5-Schema-Ebenen-Modell unterscheidet fünf Arten von Schemata, welche die Ebenen des Modells ausmachen. Den Übergang von einer Schema-Ebene zur nächsten realisieren Software-Prozessoren. In Abbildung 10 ist das 5-Schema-Ebenen-Modell beispielhaft dargestellt.

Die Komponenten-Datenbanksysteme sind der untersten Schicht zugeordnet. Ihre konzeptuellen Schemata werden als ***lokale Schemata*** bezeichnet und bilden die unterste Schema-Ebene des Modells. Die *lokalen Schemata* liegen entsprechend dem Datenmodell des DBMS des KDBS vor. Zur Überwindung der Datenmodellheterogenität werden die *lokalen Schemata* durch Transformations-Prozesse in Schemata überführt, denen ein gemeinsames Datenmodell zugrunde liegt. Die dabei entstehenden Schemata werden als ***Komponenten-Schemata*** bezeichnet. Das gemeinsame Datenmodell (Common Datamodel (CDM)) wird auch als kanonisches Datenmodell (KDM) bezeichnet. Der Übergang von den *lokalen Schemata* zu den ***Komponenten-Schemata*** wird als Schema-Transformation oder auch Homogenisierung bezeichnet.

Beim Übergang von Schicht 2 zur Schicht 3 wird auf die *Komponenten-Schemata* eine Filterung angewandt, um nur die für die Föderation relevanten Daten in das globale Schema einzubeziehen. Dieser Vorgang begründet sich durch die *Kooperationsautonomie* der KDBS. Die Datenbankadministratoren der KDBS und des FDBS verhandeln darüber, in welchem Umfang dem FDBS Zugriff auf die Daten der einzelnen KDBS gewährt wird. Die dabei enstehenden Schemata werden als ***Export-Schemata*** bezeichnet.

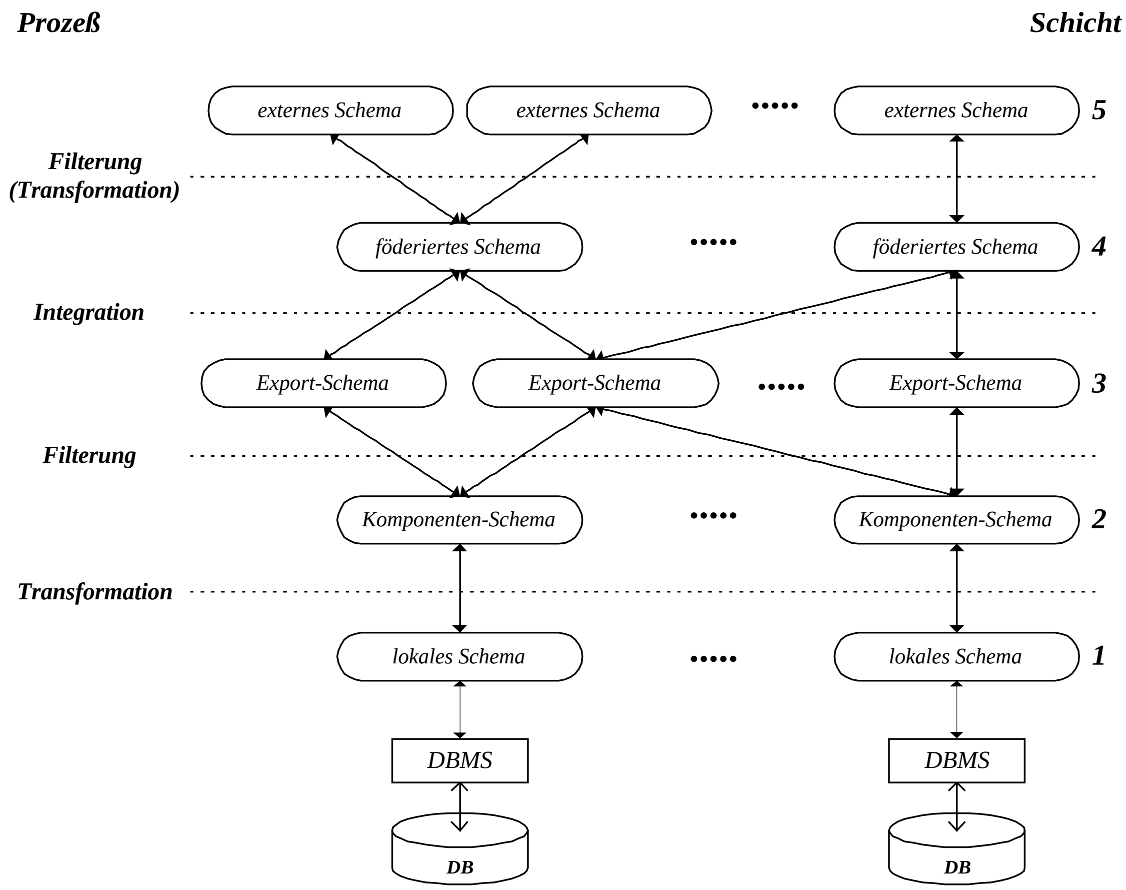


Abbildung 10 Das 5-Schema-Ebenen-Modell nach [SL90]

Ein ***föderiertes Schema*** ist das Ergebnis der Integration mehrerer *Export-Schemata* und bietet einen einheitlichen Zugriff auf die verteilten Datenbestände des FDBS. Unterschiedliche *föderierte Schemata* können die spezifischen Informationsbedürfnisse verschiedener Gruppen von globalen Benutzern bzw. Anwendungen abbilden. Verschiedene *Export-Schemata* eines *Komponenten-Schemas* unterstützen diese Diversifikation (siehe Abbildung 10). Eine wichtige Aufgabe dieser Schema-Ebene besteht darin, die Verteilung der Daten transparent zu machen. Der Übergang von *Export-Schemata* in ein *föderiertes Schema* wird als Schemaintegration bezeichnet. Die dabei auftretenden Probleme haben ihren Grund in der semantischen Heterogenität der lokalen Daten.

Die oberste Schicht des 5-Schema-Ebenen-Modells beinhaltet die ***externen Schemata***, welche in etwa mit der dritten Schicht der ANSI/SPARC Architektur [SL90] vergleichbar ist**.** Ein *externes Schema* ist eine spezifische Sicht eines einzelnen Benutzers oder einer kleinen Benutzergruppe auf ein *föderiertes Schema.* Die Aufgabe dieser Schema-Ebene liegt in erster Hinsicht in der Zugriffskontrolle. Darüber hinaus kann es unter Umständen sinnvoll sein, speziellen Benutzern bzw. Anwendungen eine Sicht auf das FDBS zu bieten, die sich in ihrem zugrundeliegenden Datenmodell von dem kanonisches Datenmodell (KDM) unterscheidet. In diesem Fall ist eine zusätzliche Transformation, vergleichbar mit jener beim Übergang von Schicht 1 zur Schicht 2, durchzuführen.

Eine konkrete Realisierung eines föderierten Datenbanksystems auf Basis des oben beschriebenen Modells beinhaltet nicht notwendigerweise Schemata in allen Ebenen des Modells. Alle Schemata mit Ausnahme der lokalen und föderierten Schemata sind optional. Stimmt beispielsweise das Datenmodell eines Komponenten-Datenbanksystems mit dem kanonischen Datenmodell überein, so würde das entsprechende lokale Schema die Aufgabe des Komponenten-Schemas übernehmen. Export-Schemata könnten entfallen, falls alle Daten eines KDBS für das FBDS zugreifbar wären oder die Zugriffskontrolle sich auf die externen Schemata beschränken würde.

3.5 Betrachtung einiger FDBS-Forschungsprojekte

3.5.1 OpenDM/Efendi

OpenDM/Efendi ist ein Projekt des C-Lab (einer Kooperation der Universität Paderborn und der Siemens-Nixdorf Informationssysteme AG) mit dem Ziel, verschiedene heterogene Datenbanksysteme und Dateisysteme durch ein föderiertes Datenbanksystem zu integrieren [Rad94]. Efendi ist eine spezielle Konfiguration des FDBS-Systems, das globalen Anwendungen den Zugriff auf die integrierten Datenbestände über ein ODMG-93-API ermöglicht.

OpenDM/Efendi verfolgt wie viele andere FDBS-Projekte den Ansatz, die Datenbankspezifika der zu integrierenden Komponenten-Datenbanksysteme in Datenbankadaptern zu kapseln. Dadurch ist der Kern des FDBS unabhängig von den Spezifika der einzelnen Komponenten-Datenbanksystemen. Abbildung 11 stellt die Architektur des OpenDM-Systems dar.

Eine Kopplungsschicht realisiert die Kommunikation zwischen dem FDBMS-Kern und den jeweiligen Datenbankadaptern und unterstützt das dynamische An- und Abkoppeln der KDBS. Die Adapter leisten die Transformation von Daten-Repräsentation und Operationen bezüglich dem Datenmodell der lokalen DBS und dem kanonische Datenmodell der Föderierungsschicht (Schematransformation). Als kanonisches Datenmodell dient ein objektorientiertes Datenmodell Namens COMIC, das auf dem ODMG-93-Objektmodell basiert. In einem Hilfs-DBS werden die Meta-Daten des föderierten Datenbanksystems verwaltet. Die Integration eines weiteren DBS beschränkt sich darauf, den entsprechenden Datenbankadapter zur Verfügung zu stellen und an das FDBS anzukoppeln. Der generische Kern des FDBS bleibt dabei unverändert. Der Administrator hat nun noch die Aufgabe, die Schemaintegration durchzuführen. Schemata werden im OpenDM-System mittels der Object-Definition-Language (ODL) des ODMG-93-Standards, welche um einige Konstrukte zur Schemaintegration erweitert wurde, definiert. Das Schlüsselwort *Schema* faßt eine Menge von Objekttyp-Definition zu einem Schema zusammen. Ein Schema kann von anderen Schemata abgeleitet werden. Das abgeleitete Schema beinhaltet dabei alle Objekttyp-Definitionen der „Import-Schemata“. Mittels einer *rename*-Klausel können Schemaelemente umbenannt werden, um Namens-Konflikte bei der Schemaintegration aufzulösen.

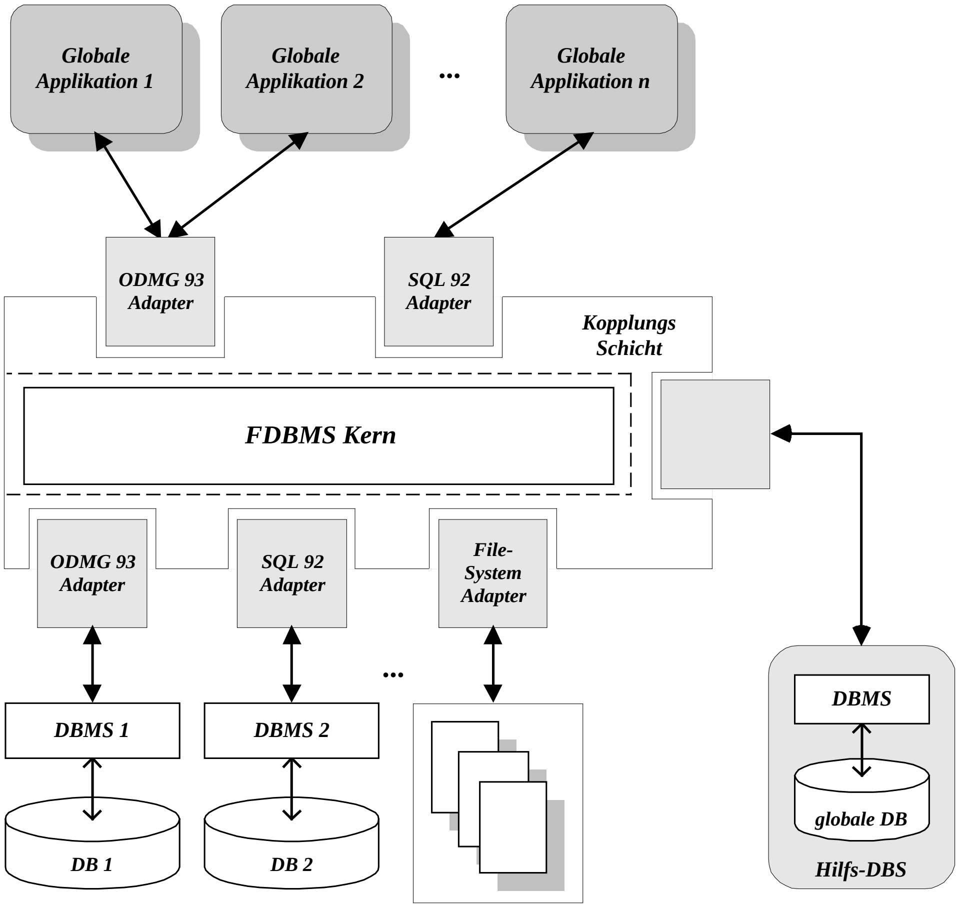


Abbildung 11 Architektur des FDBS OpenDM / Efendi nach [Rad94]

Globale Anwendungen greifen auf das FDBS über externe Adapter zu. Diese realisieren die externen Schemata der in Abschnitt 3.4 vorgestellten 5-Ebenen Referenz-Architektur. Das OpenDM-System sieht unterschiedliche externe Adapter vor, die sich in ihrem zugrundeliegenden Datenmodell voneinander unterscheiden. Die externen Adapter leisten also eine zusätzliche Schematransformation (siehe Abschnitt 3.4). Die Efendi-Konfiguration des Systems bietet globalen Anwendungen ein ODMG-93-konformes API. Dieses ist erweitert um Funktionen zur Objekt-Duplikation und -Migration.

Wie in [RS94, RS95] dargelegt, stellt Objekt-Duplikation und -Migration ein probates Mittel für eine sanfte Migration von nicht mehr zeitgemäßen, weniger robusten Datenhaltungssystemen zu *State-of-the-Art* Datenbanksystemen dar. Zunächst wird ein Datenbankadapter für das betrachtete System zur Verfügung gestellt und dessen Komponenten-Schema integriert (vergleiche hierzu Abbildung 11). Zusätzlich wird ein externer Adapter, dem das Datenmodell des alten DBS zugrunde liegt, entwickelt. Der externe Adapter muß also eine zusätzliche Schematransformation durchführen [SL90] (vergleiche Abbildung 10) und simuliert somit die API des alten DBMS. Die Daten des Alt-Systems werden dann in ein modernes DBS dupliziert. In einer ersten Phase laufen die Anwendungen, die sich auf das alte DBS beziehen, noch gegen das API des lokalen Komponenten-DBS. Während dieser Phase kann die Korrektheit und Robustheit des Zugriffs über das FDBS getestet werden. Wurde diese Testphase erfolgreich abgeschlossen, so kann das alte DBS außer Betrieb genommen werden. Eine detailierte Darstellung hierzu findet sich in [RS94].

3.5.2 IRO-DB

IRO-DB (Interoperable Relational and Object Databases) ist ein ESPRIT-III-Projekt der Abteilung IPSI (Institute for Integrated Publication and Information Systems) der Gesellschaft für Mathematik und Datenverarbeitung (GMD) in Kooperation mit meheren industriellen Partnern (GOPAS, IBERMATICA, EDS, GRAPHAEL, INTRASOFT,O_2 Technology, PRISM) [BFHK95, IRO-DB1-94, IRO-DB2-94, IRO-DB1-95]. Die wesentlichen Eigenschaften des IRO-DB-Systems sind:

- Zur Integration von relationalen und objektorientierten Datenbanksystemen verwendet das IRO-DB-System das ODMG-93-Objektmodell als kanonisches Datenmodell.
- Globale Anwendungen haben Zugriff auf das föderierte Datenbankssystem über ein standardisiertes Application Programming Interface (API). Die ODMG-93-C++-Sprachanbindung (C++OML) ist hierbei die Schnittstelle zwischen Anwendung und FDBS. Als Datenbankanfragesprache dient OQL.
- Eine Integrator Workbench realisiert eine werkzeugunterstützte Schemaintegration. Datenbank-Schemata werden dabei in ODL spezifiziert.

Abbildung 12 zeigt die Architektur des IRO-DB-Systems. Diese besteht aus drei Schichten:

- Die lokale Schicht (*Local Layer*) realisiert die Schematransformation mittels der lokalen Datenbank-Adapter (LDA). Die Adapter ermöglichen einen ODMG-93-konformen Zugriff auf die lokalen Daten. Ein OQL-Query-Processor ist Bestandteil jedes LDA. Dieser setzt DB-Anfragen in OQL in entsprechende Anfragen des lokalen DBMS um.
- Die Kommunikationsschicht (*Communication Layer*) ermöglicht dem FDBS den netzwerktransparenten Zugriff auf die lokalen Datenbanksysteme nach dem Client/Server-Ansatz. Die lokalen DBS fungieren dabei als Server, die Interoperationsschicht (*Interoperable Layer*) als Client. Der Remote-Object-Access-Dienst zerfällt in zwei Komponenten, die als *Client-Stub* under *Server-Stub* bezeichnet werden können. Die Kommunikation erfolgt mittels *Remote Procedure Call* (*RPC*). Zum Austausch von Daten wird die, in der Unix-Welt weit verbreitete, eXternal Data Representation (XDR) verwendet.

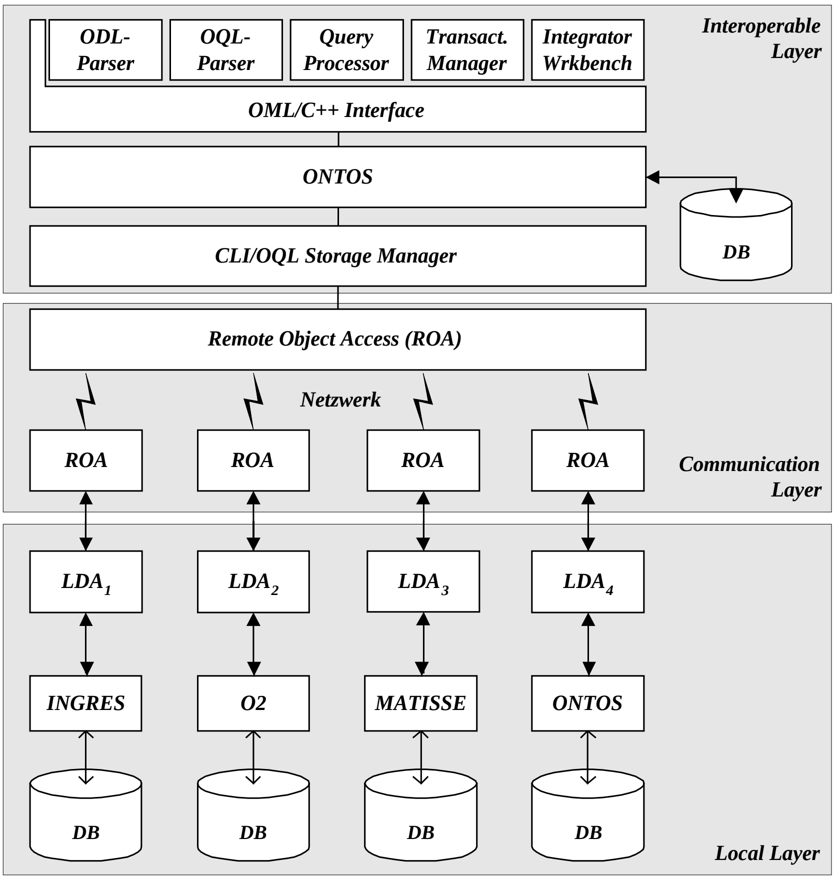


Abbildung 12 Die IRO-DB-Architektur

- Die Interoperationsschicht (*Interoperable Layer*) realisiert einen einheitlichen verteilungstransparenten Zugriff auf die Daten des FDBS. Das IRO-DB-System verhält sich nach außen wie ein ODMG-93 konformes objektorientiertes Datenbanksystem. Die Meta-Daten des IRO-DB-Systems werden intern durch das objektorientierte Datenbanksystem ONTOS verwaltet. Der globale Query-Processor zerlegt globale DB-Anfragen in lokale Subanfragen, die dann mittels der Kommunikationsschicht an die lokalen DBS weitergeleitet werden. Die Ergebnisse der Subanfragen werden als Zwischenergebnisse in der ONTOS-DB gespeichert und von dort aus weiterverarbeitet. Die Integrator Workbench unterstützt neben der Schemaintegration auch die Generierung der Schema-abhängigen C++-Laufzeit-APIs (siehe Kapitel 3.4).

Ein besonders herausragendes Merkmal des IRO-DB-Systems ist der gewählte Schemaintegrations-Ansatz [BFN94]. *Virtuelle Objekttypen* ermöglichen die Sichtenbildung auf einen oder mehrere andere Objekttypen, welche als *importierte Objekttypen* bezeichnet werden. Die Definition eines *virtuellen Objekttyps* umfaßt neben dem deklarativen ODL-Anteil noch einen *Mapping*-Teil, der die Materialisierung der Sicht definiert und auf OQL basiert. Hierdurch wird eine sehr flexible Schemaintegration ermöglicht.

Kapitel 4 Der ODMG-93-Standard für objektorientierte Datenbanksysteme

4.1 Grundlagen

Im Jahr 1991 begannen Verteter der führenden Hersteller von objektorientierten Datenbanksystemen eine Organisation Namens Object Database Management Group (ODMG) ins Leben zu rufen, um einen Standard für objektorientierte Datenbankmanagementsysteme (ODBMS) zu erarbeiten und zu etablieren. In ihren Augen war das Fehlen eines Standards der Hauptgrund für das Nischen-Dasein von objektorientierten Datenbankmanagementsystemen. Die durch die ODMG-Mitglieder angebotenen Systeme stellen heute 100 % aller verfügbaren ODBMS dar [ODMG97]. Eine erste Version des Standards wurde 1993 veröffentlich und trägt den Namen ODMG-93. Inzwischen ist die Version 2.0 dieses Standards veröffentlicht worden [ODMG97]. Die ODMG wurde 1994 als Unterorganistion in die Object Management Group (OMG) eingegliedert. Die von den ODMG-Mitgliedern angebotenen Produkte werden in naher Zukunft den Standard ODMG-93 unterstützen, falls dies nicht schon bereits geschehen ist.

Ziel der ODMG war und ist es, einen Standard für ODBMS zu schaffen, der Source-Code Kompabilität bezüglich der APIs (Application Programing Interface) von objektorientierten Datenbanksystemen bietet, so daß Anwendungen durch bloße Neuübersetzung auf ein anderes ODMG-93-Datenbanksystem portiert werden können. Vergleichbar hierzu ist etwa SQL-92 als standardisierte Anfragesprache für relationale Datenbanksysteme. Wichtiger Aspekt dieser Zielsetzung ist, daß das aus dem Standard resultierende API funktional reichhaltig genug ist, um Erweiterungen seitens der Anbieter von ODBMS erst gar nicht als notwendig erscheinen zu lassen und somit wirkliche Portabilität zu gewährleisten. Studien in den achziger Jahren zeigten, daß in Anwendungen, die sich auf relationale DBMS stützten, etwa 60 - 70% aller API Aufrufe sich auf RDBMS-anbieterspezifische Erweiterungen bezogen, also nicht im Standard SQL definiert waren [Cat94].

ODMG-93 setzt sich aus vier Hauptkomponenten zusammen: das *Object Model*, die Datendefintionssprache *ODL* (Object Definition Language), die Datenbankanfragesprache *OQL* (Object Query Language) und die Definition von standardisierten Sprachanbindungen, welche als *Object Manipulation Language* (*OML*) bezeichnet werden (derzeit definiert für C++, Smalltalk und Java). Diese vier Hauptkomponenten werden in den weiteren Abschnitten dieses Kapitels detailierter dargestellt. Die Darstellung basiert im wesentlichen auf [Cat94].

4.2 Objektmodell

Das „Object Model“ geht auf das „Core Object Model“ der OMG zurück und wurde um datenbankspezifische Eigenschaften wie z.B. „Relationships“ und Extensionen erweitert.

4.2.1 Das Kern-Objektmodell

Grundlegendes Modellierungsprinzip ist das Objekt. Objekte werden zu Objekttypen kategorisiert. Alle Objekte eines Objekttypen zeigen gleiches Verhalten und haben einen gleichen Zustandsraum. In der ODMG-93 Terminologie haben sie dieselben Charakteristika. Das Verhalten von Objekten eines Objekttyps wird durch eine Menge von Operationen, die auf diesen ausgeführt werden können, bestimmt. Der Zustand eines Objektes ergibt sich durch die Menge seiner Eigenschaften. Eigenschaften können Attribute oder Beziehungen sein. In Abbildung 13 ist die grundlegende Typhierarchie des Objektmodells dargestellt.

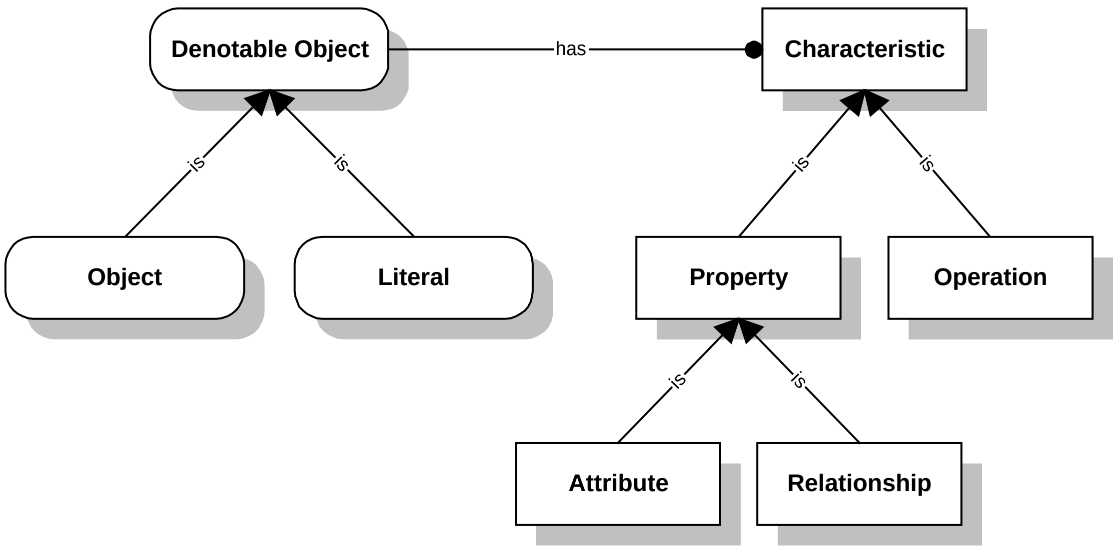


Abbildung 13 Die grundlegende Typhierarchie des ODMG-93-Objektmodells

Objekte zerfallen in zwei Kategorien, die im ODMG-93-Objektmodell als *Object* und *Literal* bezeichnet werden. Der Unterschied dieser beiden Arten von Objekten besteht in der Objekt-Identität. Objekte der Kategorie *Object* haben eine Identität, die unabhängig von ihrem aktuellen Zustand ist, weshalb sie auch als *mutable Objects* bezeichnet werden. Die Identität wird durch einen *Object-Identifier* (kurz *OID*) realisiert. Die konkrete Repräsentation der OIDs ist im Objektmodell nicht spezifiziert. Die Identität von Objekten der Kategorie *Literal* ist wertabhängig. Objekte dieser Kategorie sind identisch, falls sie in allen ihren Eigenschaften übereinstimmen. Deshalb werden sie auch als *immutable Objects* bezeichnet. Literale Objekte dienen als Wertebereich für die Attribute von veränderbaren Objekten (*mutable Objects*). Lediglich Objekte mit wertunabhängiger Identität (*mutable Objects*) können Beziehungen untereinander eingehen.

4.2.2 Eigenschaften von Objekttypen

Objekttypen beschreiben den Zustandsraum und das Verhalten ihrer Instanzen. Sie definieren somit das Interface ihrer Instanzen. Genauso wie im CORBA-Objektmodell kann es zu einem Interface mehrere Implementierungen geben. Eine konkrete Implementierung zusammen mit ihrem Interface werden in der ODMG-93-Terminologie als ***Klasse*** bezeichnet.

Objekttypen selbst werden im ODMG-93-Objektmodell auch als Objekte betrachtet. Wie alle anderen Objekte haben auch sie Eigenschaften und Operationen. Als Operationen sind auf einem Objettyp das Kreieren und Löschen von Instanzen definiert. Zu den Eigenschaften eines Objekttyps zählen die Angabe aller Supertypen, die Extension und die Definition von Schlüsseln:

- Ein Objekttyp kann einen oder mehere Objektypen als **Supertypen** haben. Die Beziehung zu seinen Supertypen ist dabei eine Eigenschaft des Objektyps. Er erbt dabei alle Charakteristika seiner Supertypen, kann neue hinzufügen oder geerbte Charakteristika redefinieren. Er spezialisiert also als Subtyp die Menge der Eigenschaften und Operationen seiner Supertypen. Durch multiple Vererbung kann es zu Namenskonflikten bei Eigenschaften oder Operationen kommen. Diese müssen durch Redefinition des Namens einer entsprechenden Eigenschaft oder Operation aufgelöst werden. Objekttypen, die nicht direkt instanziierbar sind, werden als abstrakt bezeichnet. Ihre Aufgabe besteht darin, gemeinsame Eigenschaften und Operationen an Subtypen zu vererben. Hinzu kommt der extensionale Aspekt, auf den im folgenden eingegangen wird.

- Als die **Extension** eines Objekttyps bezeichnet man die Menge all seiner Instanzen. Die Extension ist eine Eigenschaft des Objekttyps und ist optional. Die Angabe des Extensions-Namens des Objektyps im Datenbankschema bewirkt, daß das Datenbanksystem die Extension automatisch verwaltet und man über diesen Namen Zugriff auf alle Objekte des Objekttyps hat. Auf Datenbankschemata wird in Abschnitt **4.3** noch eingegangen. Der Extensions-Name eines Objekttyps ist in etwa vergleichbar mit dem Tabellennamen in einer relationalen Datenbank. Wie schon oben angedeutet hat die Supertyp- / Subtyp-Beziehung von Objekttypen auch einen extensionalen Aspekt. Zur Extension eines Objekttypen zählen neben den direkt vom Objektyp instanziierten Objekte auch rekursiv die Objekte der Extensionen seiner Subtypen. Abbildung 14 verdeutlicht diesen Zusammenhang am Beispiel. Die Objekttypen *Angestellter* und *Arbeiter* sind Subtypen des Objekttyps *Mitarbeiter*. Der Objekttyp *Angestellter* ist abstrakt und hat die Subtypen *Manager* und *Sachbearbeiter*. Seine Extension besteht in der Vereinigungsmenge der Extensionen von *Manager* und *Sachbearbeiter*. Ebenso sind die Extensionen der Objekttypen *Angestellter* und *Arbeiter* Teilmengen der Extension von *Mitarbeiter*.

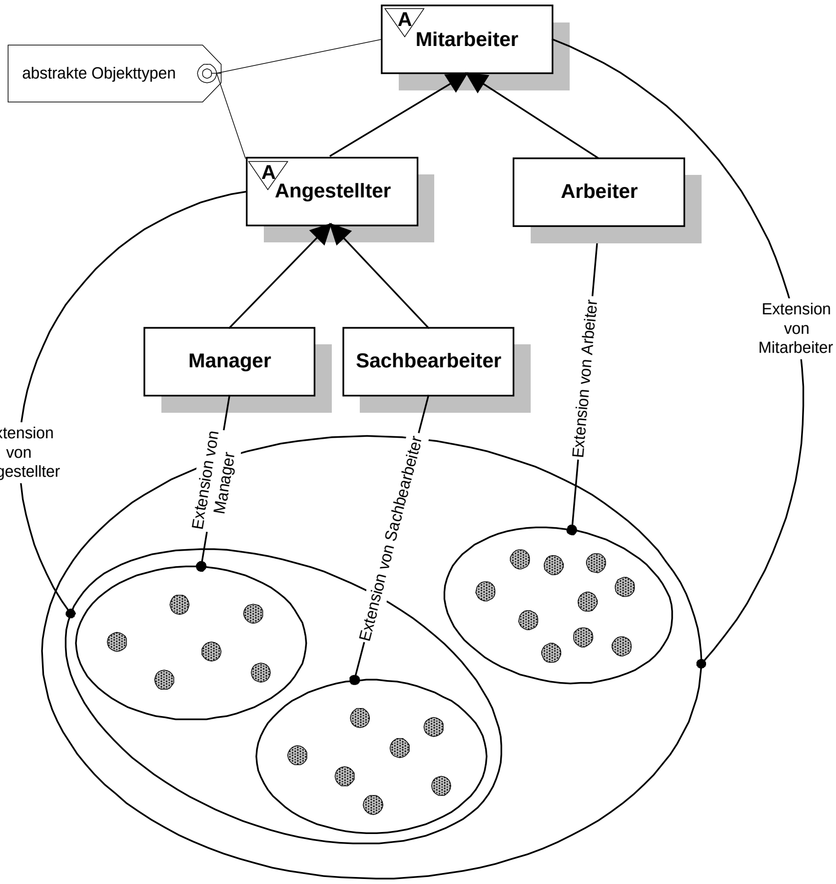


Abbildung 14 Der extensionale Aspekt der Supertyp-Beziehung

- Eine weitere optionale Eigenschaft von Objekttypen ist die Angabe von Schlüsseln. Ein Schlüssel ist gegeben durch eine Menge von Instanz-Eigenschaften, welche die Objekte des Objekttypen durch eben diese Eigenschaften eindeutig identifiziert. Im obigen Beispiel (Abbildung 14) wäre z.B. eine Mitarbeiternummer als Schlüssel-Attribut des Objektyps Mitarbeiter denkbar. In diesem Fall hätte man einen einfachen Schlüssel im Gegensatz zu einem zusammengesetzten Schlüssel, der aus meheren Eigenschaften besteht. Bestandteil eines Schlüssels können nicht nur Attribute, sondern auch Beziehungen sein.

4.2.3 Atomare und strukturierte Objekte

Wie in Abbildung 13 dargestellt unterscheidet das Objektmodell zwischen Objekten mit wertunabhängiger und wertabhängiger Identität (*mutable Object* und *Literal*). Orthogonal hierzu werden strukturierte von atomaren Objekten unterschieden. Strukturierte Objekttypen ergeben sich durch die Anwendung von Typ-Generatoren. Diese werden auch als parametrisierbare Typen bezeichnet. In der Programmiersprache

C++ werden parametrisierbare Typen als Templates bezeichnet. Das Objektmodell definiert hierzu sechs Typ-Generatoren:

- Durch den Typ-Generator ***struct***$<e_1{:}T_1,\ldots,e_n{:}T_n>$ lassen sich Objekttypen konstruieren, deren Instanzen Tupel darstellen. Die so erzeugten Objekttypen dienen als Wertebereich von Attributen oder zur Strukturierung des Ergebnisses einer Datenbankanfrage.

- Eine endliche Menge von Elementen läßt sich durch den Aufzählungs- Typ-Generator ***enum*** definieren. Solche Objekttypen können nur als Literale vorkommen und dienen somit ausschließlich als Wertebereich für Attribute.

- Die Typ-Generatoren ***List***<***T***>***, Set***<***T***>***, Bag***<***T***>***, Array***<***T***>***,*** sind Subtypen des abstrakten Typ-Generators ***collection***<***T***>. Ihre Anwendung liefert Objekttypen, die Kollektionen von Objekten des Objekttyps ***T*** entweder als Liste, Menge, Multi-Menge oder Array verwalten. Diese Objekttypen finden Verwendung, um mengenwertige Attribute zu realisieren (als *Literal*). Darüber hinaus ermöglichen sie auch die Speicherung von Mengen als Objekte (als *mutable Objects*) in der Datenbank, die unabhängig von ihrem aktuellen Wert sind. Die Identität dieser Menge ergibt sich dann aus ihrer *OID*. Die durch Anwendung der Kollektionen-Typ-Generatoren enstandenen Objekttypen definieren eine Reihe von Operationen zur Manipulation der speziellen Kollektion. Dabei definiert der abstrakte Typ-Generator ***collection***<***T***> alle Operationen, die allen Kollektionen gemeinsam sind. Die Signaturen dieser Operationen sind in der nachfolgenden Abbildung 15 angegeben. Die Semantik der Operation läßt sich zumeist schon aus ihren Namen erahnen. Die Operationen *select_element, select, query* und *exists_element* ermöglichen einen wertbasierten Zugriff auf die Elemente der Kollektion. Hierbei wird eine OQL-Anfrage (in Abbildung 15 *OQL_predicate*) als Werte-Bedingung für die zu selektierenden Objekte in der Kollektion verwandt.

```
assign_from(Collection<T>);
Boolean equal(Collection<T> cL,
              Collection<T> cR);
Integer cardinality();
Boolean is_empty();
Boolean is_ordered();
Boolean allows_duplicates();
Boolean contains_element(T element);
Boolean insert_element(T elem);
Boolean remove_element(T elem) ;
remove_all();
Iterator<T>  create_iterator();
T select_element(String OQL_predicate);
Iterator<T>  select(String OQL_predicate);
Boolean query(Collection<T>,
              String OQL_predicate);
Boolean exists_element(
              String OQL_predicate);
```

Abbildung 15 Grundlegende Operation auf Kollektionen

Ein spezielles Konzept erlaubt die Iteration über die Elemente einer Kollektion. Der Typ-Generator *Iterator<T>* stellt einen abstrakten Zeiger auf die aktuelle Position beim sequentiellen Durchlauf einer Kollektion dar. Die Operationen von *Iterator<T>* sind in Abbildung 16 angegeben. Dieses generische Konzept ermöglicht es, die Kollektion *List<T>, Set<T>, Bag<T>, Array<T>* in der gleichen Art und Weise zu durchlaufen.

first();
last():
reset();

***Boolean** not_done() **const**;*

advance();
***T** get_element() **const**;*
*replace_element(**T** obj);*

***Boolean** next(**T** obj);*

Abbildung 16 Operationen von Iterator<T>

Atomare Objektypen sind all die Objektypen, die nicht durch Anwendung der oben genannten Typ-Generatoren enstanden sind. Atomare Literale sind die Standard-Datentypen *Integer, Float, Character, Boolean.*

Das ODMG-93-Objektmodell definiert weiterhin die literalen Objekttypen *Date, Time, Timestamp* und *Interval.* Diese ergeben sich durch Anwendung des Typ-Generators ***struct***. Diese Objekttypen entsprechen ihren Pendants in der ANSI SQL Spezifikation [Cat94].

4.2.4 Instanz-Eigenschaften von Objekten

Wie in Abschnitt 4.2.1 beschrieben wurde, bestimmt ein Objekttyp die Eigenschaften und Operationen seiner Instanzen. Eine Instanz-Eigenschaft ist entweder ein Attribut oder eine Beziehung zu anderen Objekten.

Attribute haben als Wertebereich immer einen literalen Objekttypen. Dieser kann entweder atomar oder strukturiert sein. Zu jedem Attribut eines Objekttyps sind implizit zwei Operationen *set_value* und *get_value* definiert. Diese dienen dem Zugriff auf den Attributwert. Zudem ist ein spezieller Wert ***nil*** definiert, der den Wertebereich von Attributen erweitert. Dieser ist vergleichbar mit dem *NULL*-Wert in relationalen Datenbanken und hat die Semantik, daß der Wert eines Attributs undefiniert ist.

Beziehungen sind definiert zwischen veränderbaren Objekten (*mutable Objects*). Das Objektmodell sieht nur binäre Beziehungen vor, N-äre Beziehungen sind nicht vorgesehen. Eine Beziehung selbst hat keinen Namen. Sie ergibt sich vielmehr aus den Zugriffs-Pfaden, welche die Beziehung ausmachen. Diese haben einen Namen. Die Zugriffs-Pfade werden im ODMG-93-Objektmodell als *Relationship-Traversal-Paths* bezeichnet. Genau genommen ist der Zugriffs-Pfad eine Instanz-Eigenschaft eines Objektes. Er erlaubt den navigierenden Zugriff auf die durch die Beziehung assoziierten Objekte. Um den Zusammenhang von

Zugriffs-Pfaden und Beziehungen besser verstehen zu können, wird an dieser Stelle dem Abschnitt 4.3 vorgriffen. Das Beispiel in Abbildung 17 zeigt die Definition der Objekttypen *DUDSeminar (*Diplomanden- und Doktoranden-Seminar*)* und *Student,* deren Instanzen zueinander in Beziehung stehen, in ODL-Notation. In diesem Beispiel handelt es sich um eine 1:N-Beziehung zwischen der Objekttypen *DUDSeminar* und *Student.* Die eigentliche Beziehung ergibt sich aus den zueinander inversen Zugriffs-Pfaden. Ein Zugriffs-Pfad wird dabei durch die Angabe des Ziel-Objekttyps, seines Namens und gegebenenfalls durch die Angabe des zu ihm inversen Zugriffs-Pfades definiert (im Beispiel nach der ***inverse***-Klausel) . Falls eine Instanz zu meheren Instanzen des Ziel-Objekttyps in Beziehung stehen kann, so ist außerdem der Kollektions-Typ zur Repräsentation der Menge der referenzierten Objekte anzugeben (im Beispiel ***Set***<*Student*>). Ein Zugriffs-Pfad hat somit eine implizite Kardinalität. Hieraus ergeben sich drei Kategorien von Beziehungen (abgesehen von Symmetrie). Diese bezeichnet man als *1:1*-, *1:N*- und *N:M*-Beziehung.

```
interface Student
(          extent allStudents                      Name der Extension
           keys Name, Vorname
) : persistent                                     Schlüssel-Attribute
{
           attribute string    Name;
           attribute string    Vorname;
           relationship        DUDSeminar nimmt_teil_an inverse DUDSeminar:: Teilnehmer;
}
                               Ziel-Objekttyp, der
                               durch die Beziehung     Name des          inverser
                               assoziierten Objekte    Zugriffs-Pfades   Zugriffs-Pfad
interface DUDSeminar
(          extent DUDSeminare
) : persistent
{
           attribute string    Lehrstuhl;
           attribute string    Ort;
           relationship        Set<Student> Teilnehmer inverse Student:: nimmt_teil_an;
}
```

Abbildung 17 Definition von Beziehungen in ODL-Notation

Beziehungen im ODMG-93-Objektmodell beinhalten das Konzept der referentiellen Integrität. Wird ein Objekt, welches an Beziehungen teilnimmt, gelöscht, so werden auch alle Referenzen auf dieses Objekt in den Zugriffs-Pfaden der assoziierten Objekte gelöscht. Die Referenzen der inversen Zugriffs-Pfade werden immer implizit mitgesetzt. Abbildung 18 verdeutlicht diesen Zusammenhang durch ein Instanz-Diagramm des obigen *DUDSeminar-Student*-Beispiels. Die gestrichelten Pfeile im Diagramm stellen dabei die implizit gesetzten Referenzen dar.

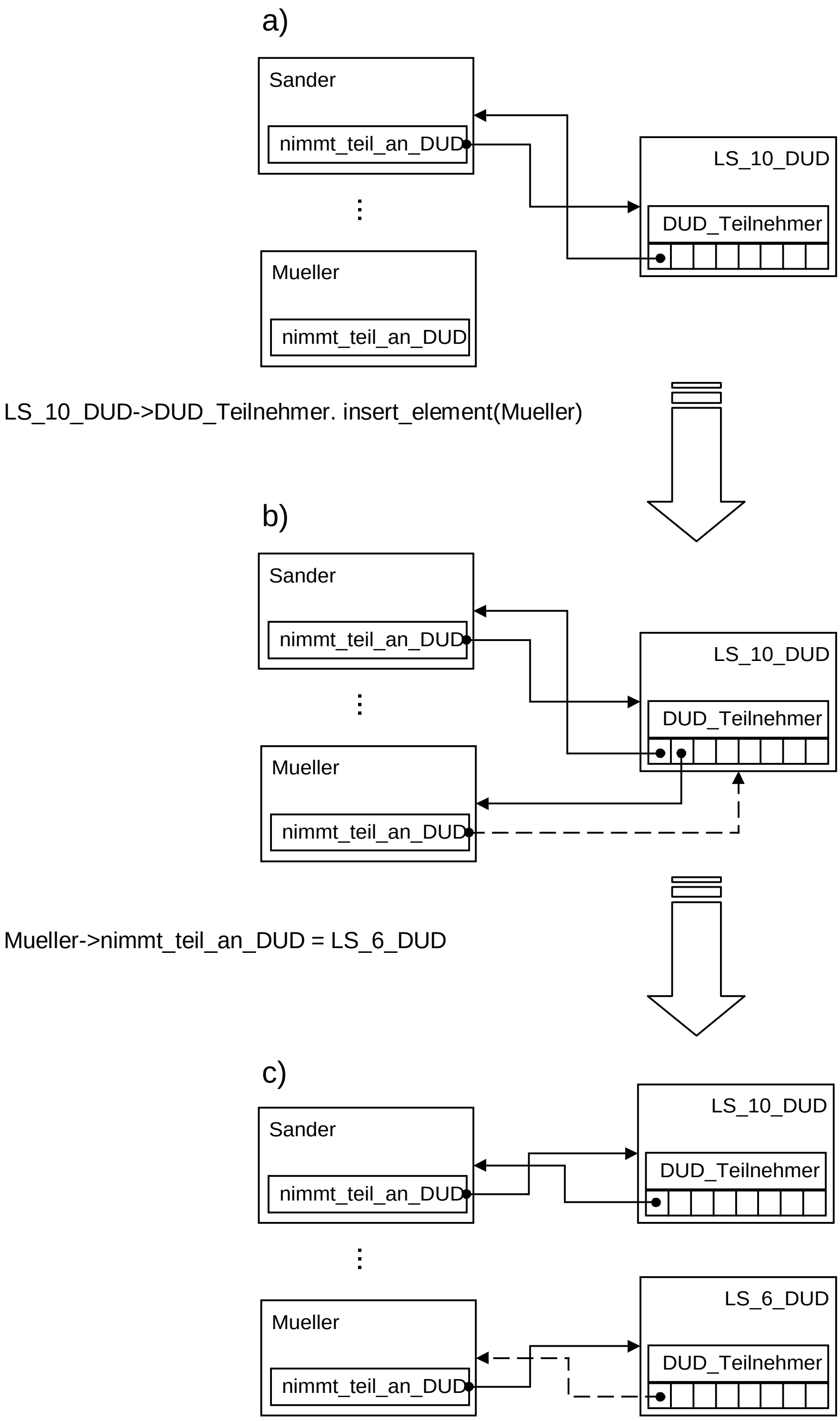


Abbildung 18 Beispiel von Beziehungen auf Instanz-Ebene

4.2.5 Der Objekt-Zugriff

Das ODMG-93-Objektmodell beinhaltet ein geschachteltes Transaktionsmodell zur Manipulation von persistenten Objekten. Abbildung 19 zeigt das Beispiel einer geschachtelten Transaktion. Das Lesen, Kreieren, Löschen und Modifizieren von persistenten Objekten ist nur in Transaktionen möglich. Transaktionen gewährleisten die ACID-Eigenschaften (Atomarität, Konsistenz, Isolation und Dauerhaftigkeit) [Date90]. Die Isolation-Eigenschaft wird durch ein pessimistisches Concurrency-Control-Protokoll realisiert, welches auf Lese- und Schreibsperren beruht. Das Objektmodell definiert einen Typ ***Transaction,*** der folgende Operationen umfaßt:

- Jede Transaction muß explizit mit ***Transaction::begin()*** gestartet werden. Jeder Lese- oder Schreib-Zugriff auf ein Objekt setzt implizit eine Lese- bzw. Schreibsperre. Alle Modifikationen von umschließenden Transaktionen, die zeitlich vor Beginn der aktuellen Transaktion durchgeführt wurden, sind für diese sichtbar.

- Verläuft die Transaktion erfolgreich, so werden durch ***Transaction::commit()*** alle in der Transaktion getätigten Modifikationen für die sie umgebende Transaktion sichtbar. Ist die Transaktion eine Top-Level-Transaktion (d.h. von keiner weiteren Transaktion umgeben), so werden die Modifikationen in die Datenbank geschrieben und sind somit für alle anderen Transaktionen (in anderen Prozessen oder Threads) sichtbar. Am Ende einer Transaktion werden alle von ihr gehaltenen Sperren beseitigt.

- Mit der Operation ***Transaction::abort()*** kann eine Anwendung eine Transaktion abbrechen. Dies bewirkt, daß der Zustand vor Beginn der Transaktion wieder hergestellt wird und alle von ihr gehaltenen Sperren freigegeben werden.

- Mit der Operation ***Transaction::abort_to_top_level()*** können Transaktion zur äußersten sie umgebenden Transaktion abbrechen. Diese Top-Level-Transaktion muß dabei nicht unbedingt ebenfalls abbrechen.

Transaction $T_{top\text{-}level}$.begin();
....
    Transaction T_{sub1}.begin();
    ....
        Transaction T_{sub2}.begin();
        ....
        if major_error then T_{sub2}.abort();
        else T_{sub2}.abort_to_top_level();
        ....
        T_{sub2}.commit();
    ....
    T_{sub2}.commit();
....
$T_{top\text{-}level}$.commit();

Abbildung 19 Beispiel einer geschachtelten Transaktion in Pseudocode

Um von einer Anwendung auf die persistenten Objekte in einer Datenbank zugreifen zu können, definiert das Objektmodell einen speziellen Typ ***Database.*** Eine Instanz dieses Typs korrespondiert zu einer logischen Datenbank. Es ist somit möglich, aus einer Anwendung heraus auf mehere Datenbanken zuzugreifen. Ein verteiltes Transaktionsprotokoll zum Zugriff auf mehrere Datenbanken innerhalb einer Transaktion ist im Objektmodell bisher nicht vorgesehen. Eine Datenbank unterliegt einem Schema, welches eine Menge von benutzerdefinierten Objekttypen beinhaltet. Die Datenbank speichert Instanzen der im Schema definierten Objekttypen. Der Typ ***Database*** umfaßt die Operationen ***open*** und ***close*** zum Öffnen und Schließen einer Datenbankverbindung, innerhalb der Transaktionen stattfinden können. Die Operationen ***set_object_name*** und ***rename_object*** dienen dazu, persistenten Objekten einen Namen zu geben, so daß dann eine Anwendung mittels der Operation ***lookup_object*** auf ein solches Objekt zugreifen kann. Objekte, die einen Namen tragen, heißen in der ODMG-93-Terminologie „*named Objects*“. Insgesamt sieht das Objektmodell drei verschiedene Möglichkeiten vor, wie eine Anwendung auf ein persistentes Objekt zugreifen kann:

- Ist ein Objekt Instanz eines Objekttyps, der einen Extensions-Namen hat, so kann die Anwendung mittels einer OQL-Datenbankanfrage auf dieses Objekt zugreifen. Diese mengenorientierte Art des Zugriffs entspricht der Vorgehensweise bei relationalen Datenbanksystemen.

- Hat ein Objekt einen Namen, so dient dieser dem Zugriff auf das Objekt. Dabei existiert innerhalb einer Datenbank ein einziger, flacher Namensraum.

- Auf Objekte, die weder einen Name haben, noch Instanz eines Objekttyps mit Extensions-Namen sind, kann nur über die Zugriffspfade von anderen zu diesen in Beziehung stehenden Objekten zugegriffen werden.

Insgesamt können in einer Datenbank nur Objekte gespeichert werden, die auch erreichbar sind. Dieses Konzept wird in der ODMG-93-Terminologie als „*Persistenz durch Erreichbarkeit*“ bezeichnet.

4.3 Object Definition Language

Bei der *Object Definition Language* (*ODL*) handelt es sich um eine deklarative Sprache, mit deren Hilfe objektorientierte Datenbankschemata programmiersprachenunabhängig definiert werden können. ODL ist in seiner Funktion vergleichbar mit dem Data-Definition-Language-Anteil von SQL (DDL). ODL ist also eine DDL für Objekttypen. ODL geht auf die *Interface Definition Language* (*IDL*) der OMG zurück. Alle Konzepte des ODMG-93-Objektmodells (siehe Abschnitt 4.2) sind in einem ODL-Schema anwendbar. Da das ODMG-93-Objektmodell eine Erweiterung des CORBA-Objektmodells darstellt, erweitert die Syntax von ODL logischerweise jene von IDL, um auch jene Konzepte, die Erweiterungen gegenüber CORBA darstellen, ausdrücken zu können. Der Einfluß der Programmiersprache C++ auf die Syntax von IDL ist natürlich auch bei ODL sichtbar, so daß ODL-Schemata C++ Header-Files sehr ähneln. Bisher sind von der ODMG die Abbildung von ODL auf die Propgrammiersprachen Smalltalk, C++ und Java spezifiziert worden. Eine vollständige Grammatik von ODL kann unter http://www.odmg.org/download/standard/ODL.yacc heruntergeladen werden und ist auch in [Cat94] angeben. An dieser Stelle soll lediglich Anhand eines kleinen Beispiels ein Gefühl dafür vermittelt werden, wie die Konzepte des Objektmodells in ODL formuliert werden. Das Beispiel modelliert eine Diplomanden- und Doktoranden-Seminarverwaltung und ist in Abbildung 20 in UML-Notation angegeben. Das dazu korrespondierende ODL-Schema ist in Abbildung 21 dargestellt.

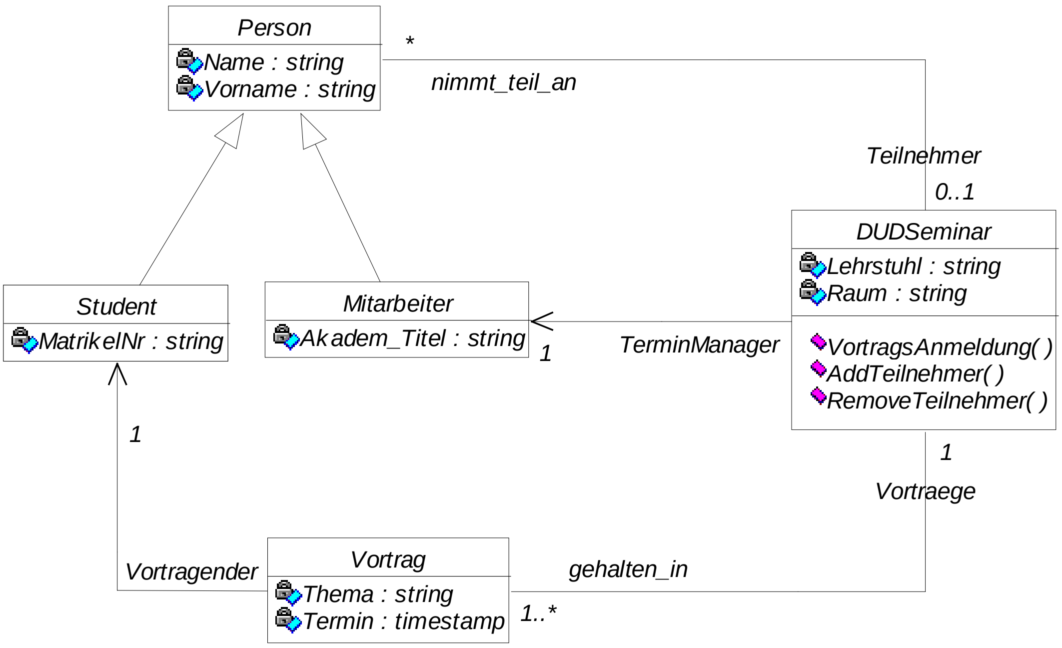


Abbildung 20 Datenbankschema in UML-Notation

```
interface Person
(       extent allPerson
        keys Name, Vorname
) : persistent
{
        attribute string     Name;
        attribute string     Vorname;
        relationship         DUDSeminar nimmt_teil_an inverse DUDSeminar::Teilnehmer;
}
interface Student : Person
(       extent allStudents
        keys MatrikelNr
) : persistent
{
        attribute string     MatrikelNr;
}
```

```
interface Mitarbeiter : Person
(           extent allMitarbeiter
) : persistent
{
            attribute string        Akadem_Titel;
}
interface DUDSeminar
(           extent DUDSeminare
            keys Lehrstuhl
) : persistent
{
            attribute string        Lehrstuhl;
            attribute string        Raum;
            relationship            Mitarbeiter TerminManager;
            relationship            Set<Person> Teilnehmer inverse Person:: nimmt_teil_an;
            relationship            List< Vortrag> Vortraege inverse Vortrag:: gehalten_in;

            boolean                 VortragsAnmeldung(in Student Std, in timestamp Termin, in string Thema);
            boolean                 AddTeilnehmer(in Student Std);
            boolean                 RemoveTeilnehmer(in Student Std);
}
interface Vortrag
(           extent Vortraege
            keys Lehrstuhl
) : persistent
{
            attribute string        Thema;
            attribute timestamp Termin;
            relationship            Student Vortragender;
            relationship            DUDSeminar gehalten_in inverse DUDSeminar:: Vortraege;
}
```

Abbildung 21 ODL-Schema der DUD-Seminarverwaltung

4.4 Object Manipulation Language

Die Transformation von ODL in die Zielprogrammiersprachen ist ebenfalls definiert (zur Zeit für C++ , Smalltalk und Java). Darüber hinaus beinhaltet die Sprachanbindung eine *Object Manipulation Language* (*OML*) für jede Zielsprache, deren Funktion es ist, den Zugriff auf die persistenten Objekte aus der Programmiersprache heraus zu ermöglichen. Oberstes Prinzip bei der Sprachanbindung von ODMG-93 an die Zielsprache ist, daß der Anwendungsentwickler mit nur einer einzigen Sprache konfrontiert ist, im

Gegensatz zu „*embedded*“ Datenbankanfragesprachen. Im folgenden soll die Sprachanbindung von C++ näher beleuchtet werden. Die konkrete Sprachanbindung für C++ ist im ODMG-93-C++-Binding definiert. Ausgehend von einem Datenbankschema in ODL, als Eingabe für einen Preprocessor, werden die Metadaten der Datenbank erzeugt (ein Data-Dictionary wird angelegt). Abbildung 22 stellt die Zusammenhänge graphisch dar. Der Preprocessor erzeugt darüber hinaus zu jedem im ODL-Schema existierenden Objekttypen eine korrespondierende Klassen-Deklaration in C++. Der Anwendungsentwickler implementiert dann die Operationen der Objekttypen als C++-Memberfunktionen der entsprechenden Klassen (siehe [Str91]). Hierzu stehen ihm eine Bibliothek von Klassen, welche die *Object Manipulation Language* realisieren, zur Verfügung.

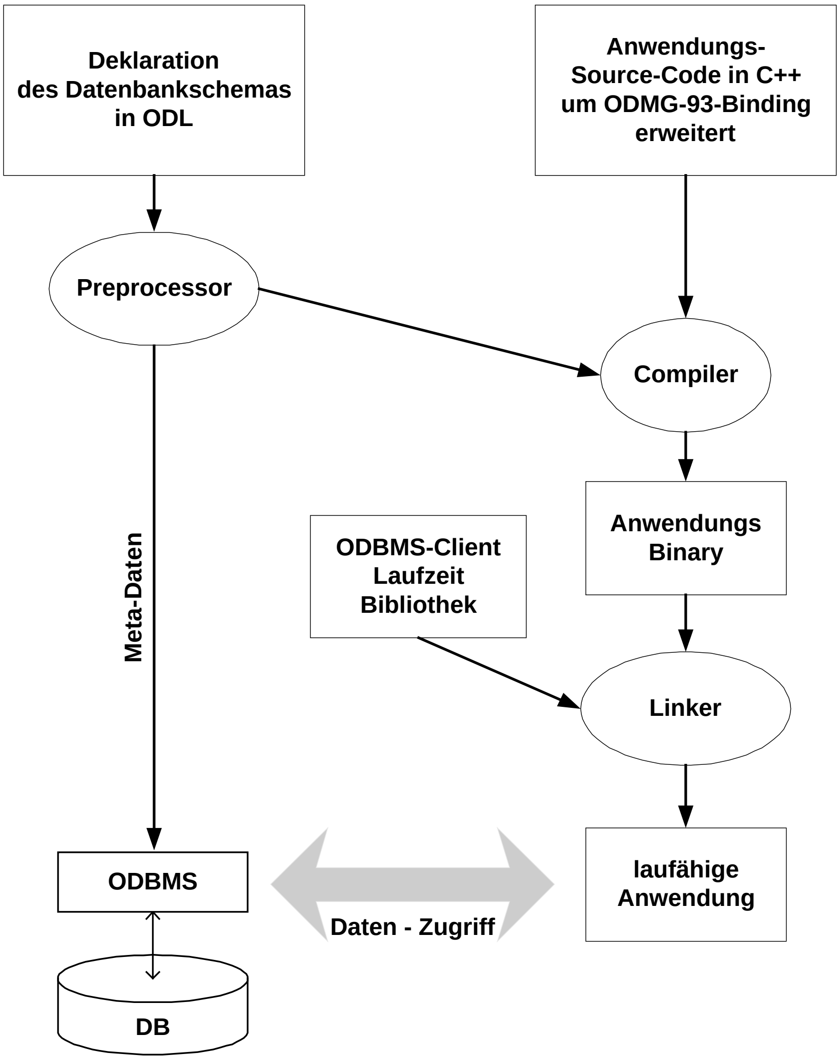


Abbildung 22 Entwicklung einer ODMG-93-Datenbank-Anwendung

Wie schon angesprochen, wird die C++-OML durch eine Menge von Klassen realisiert. Die wichtigsten dieser Klassen und deren Funktion werden im folgenden kurz skizziert:

- Die Klasse ***d_Database*** ermöglicht eine Verbindung zu einer logischen Datenbank aus einer Anwendung heraus auf- und abzubauen. Innerhalb einer solchen Datenbank-Verbindung (Instanz der Klasse ***d_Database***) können dann Transaktionen gegen die Datenbank erfolgen.

- Das Transktionsmodell, wie es im Abschnitt 4.2.5 auf Seite 35 vorgestellt wurde, wird durch die Klasse ***d_Transaction*** realisiert.

- Alle Klassen mit der Fähigkeit ihre Instanzen persistent in der Datenbank speichern zu können, sind direkt oder indirekt von der Klasse ***d_Object*** abgeleitet.

- Der Zugriff auf persistente Objekte erfolgt über sogenannte ***Smart pointer***, welche durch die Template-Klasse ***d_Ref<T>*** realisiert werden. Templates werden auch als parametrisierbare Typen oder Typ-Generatoren bezeichnet [Str91]. Eine Instanz der Klasse ***d_Ref<T>*** ist eine Referenz auf ein persitentes Objekt des Objekttyps ***T*** in der Datenbank. ***Smart pointer*** verhalten sich vom Standpunkt des Anwendungsentwicklers wie normale C++-Zeiger. Sie garantieren darüber hinaus die Integrität von Referenzen auf persistente Objekte. Verweisen zwei ***Smart pointer*** in einem Prozeß auf dasselbe Objekt, so befindet sich höchsten eine Kopie davon im Hauptspeicher. Beziehungen werden ebenfalls durch ***Smart pointer*** realisiert. Innerhalb einer Transaktion wird ein referenziertes Objekt erst dann in den Hauptspeicher geladen, wenn es zum ersten mal dereferenziert wird. Dadurch wird verhindert, daß bei einem Zugriff auf ein Objekt alle transitiv von ihm referenzierten Objekte mit in den Hauptspeicher gelangen würden. Dies hätte bei stark zusammenhängenden „Objekt-Netzen“ zur Folge, daß der Speicher des Programms sehr schnell erschöpft wäre.

- Die Kollektionen-Typ-Generatoren (siehe Abschnitt 4.2.3 auf Seite 30) werden durch die Template-Klassen ***d_Set<T>, d_Bag<T>, d_List<T>, d_Varray<T>*** realisiert. Diese sind von der virtuellen Basis-Klasse ***d_Collection<T>*** abgeleitet. Das Iterartor-Konzept wird durch die Klasse ***d_Iterator<T>*** unterstützt.

- Instanzen der Klasse ***d_OQL_Query*** ermöglichen es, OQL-Datenbankanfragen zu formulieren und auszuführen.

Nachfolgendes Beispiel in Abbildung 23 demonstriert die Funktionsweise einiger der oben aufgeführten Klassen. Das Beispiel-Programm bezieht sich auf das ODL-Datenbankschema aus Abschnitt 4.3. und gibt alle Vorträge des Diplomanden- und Doktoranden-Seminars des Lehrstuhls 10, die irgendetwas mit „ODMG“ zu tun haben, an.

```
d_Datebase db;

db.open("FIDB“);
```

logischer Name der Datenbank

```
d_Transaction tr;

tr.begin();
d_OQL_Query DBAnfrage(   "select v from v in Vortraege“
                         " where (v.Thema = \"*ODMG*\")“
                         " and (v.gehalten_in.Lehrstuhl =  \"LS10\")“  );

DBAnfrage.assign(db);

d_Set<d_Ref<Vortrag>> ODMG_Vortraege;

d_oql_execute(DBAnfrage, ODMG_Vortraege);
```

Ergebnismenge

```
d_Iterator<d_Ref<Vortrag>> iter = ODMG_Vortraege.create_iterator();

d_Ref<Vortrag> current_Vortrag;
```

Iterations-Konzept

```
while (iter.next(current_Vortrag)) {

        printf(current_Vortrag->Termin, current_Vortrag->Thema,

                current_Vortrag->Vortragender->Name);
```

Navigation über Zugriffs-Pfad. Erst zu diesem Zeitpunkt gelangt das Objekt vom Typ Student in den Speicher

```
}

tr.commit();

db.close();
```

Abbildung 23 C++OML in einem Beispielprogramm

4.5 Object Query Language

Die Datenbankanfragesprache OQL ist eine objektorientierte Erweiterung von SQL. OQL verfügt ähnlich wie SQL über wenige, sehr mächtige Sprachkonstrukte, geht dabei aber weit über SQL hinaus. Im Gegensatz zu SQL handelt es sich bei OQL um eine reine Datenbankanfragesprache. SQL hingegen umfaßt eine Datendefintionssprache (DDL) und eine Datenmanipulationssprache (DML). Im ODMG-93-Standard ist, wie schon erwähnt, ODL die DDL, die DML wird als Object Manipulation Language (OML) bezeichnet und ist durch die Sprachanbindung in die Zielsprache (etwa C++OML) gegeben.

Eine OQL-Datenbankanfrage hat die von SQL bekannte ***Select*** $t_1, ..., t_n$ ***from*** $\delta_1, ..., \delta_k$ ***where*** ϕ Form. Die Terme $t_1, ..., t_n$ berechnen dabei die Ergebnismenge. Die Ausdrücke $\delta_1, ..., \delta_k$ bestimmen die Menge(n) von Objekte, auf die sich die Anfrage bezieht. Dabei hat ein Ausdruck δ_i immer die Form ***var in S***, wobei ***var*** eine freie Variable ist (vergleichbar mit den Tupelvariablen in SQL) und ***S*** auf eine endliche Menge von Objekten verweist. ***S*** wird als mengenwertiger Ausdruck bezeichnet und kann beispielsweise ein Extensions-Name, der Name eines mengenwertigen Attributs, der Name eines mengenwertigen Zugriffs-Pfades oder selbst wieder eine Anfrage sein. Auf mengenwertige Ausdrücke lassen sich auch von SQL bekannten Aggregations-Funktionen ***{ min, max, avg, sum, count }*** anwenden. ϕ ist die Anfrage-Bedingung, welche auch wiederum Subanfragen enthalten kann. Bedingungen bestehen aus Prädikaten, die mithilfe boolescher Operatoren ***{ and, or, not }*** und Quantoren ***{ exists, for all }*** zu komplexeren Bedingungen verknüpft werden können. In Prädikaten können Objekt-Eigenschaften und Objekt-Operationen durch Vergleichsoperatoren ***{ =, <, >, ...}*** oder die Element-von-Operation (***t in S***) verwandt werden. Objekt-Eigenschaften werden durch den Zugriffs-Operator ***{ . | ->}*** dereferenziert. Hierdurch ist es auch möglich, (in einem Ausdruck) entlang von Zugriffs-Pfaden von Beziehungen zu traversieren. Solche Ausdrücke werden als ***Path-Expressions*** bezeichnet. In relationalen Datenbanken wäre hierzu ein *Join* notwendig. Anfragen können durch die Mengenoperationen ***{ intersect, union, except }*** zu komplexeren Anfragen verknüpft werden. Ähnlich wie in SQL existiert in OQL auch ein ***group by*** Konstrukt. Ergebnismengen lassen sich durch das ***sort by*** Klausel sortieren.

Obige Darstellung beschränkt sich darauf, die wichtigtsten Konzepte von OQL zu skizzieren. Eine vollständige Darstellung der Datenbankanfragesprache OQL und deren Grammatik findet sich in [Cat94]. Häufig wird kritisiert, daß die Semantik von OQL im ODMG-93 nicht formal, sondern nur durch Beispiele spezifiziert wurde [BFHK95].

Den Abschluß dieses Kapitels bilden einige OQL-Anfrage-Beispiele, die sich auf das ODL-Schema in Kapitel 4.3 auf Seite 38 beziehen.

Aggregation-Fkt angewendet auf mengenwertigen Ausdruck

Extensions-Name

select s.Lehrstuhl, count(s.Teilnehmer) from s in DUDSeminare

liefert als Ergebnis die Tupel, bestehend aus dem Namen des Lehrstuhls und der Teilnehmerzahl aller DUDSeminare.

Path-Expression

select v from v in Vortraege where (v.gehalten_in.Lehrstuhl = "LS10“)
and (Termin >= 1996-01-01) and (Termin <= 1996-12-31)

berechnet alle Vorträge, die im Jahre 1996 am Lehrstuhl "LS10“ gehalten wurden.

Existenz-Quantor

*select s from s in DUDSeminare where (exists v in s.Vortaege: v.Thema = "*ODMG*“)*

liefert als Ergebnis alle DUDSeminare, die sich mit dem Thema "ODMG“ beschäftigt haben.

Kapitel 5 Anforderungen und Ziele des entwickelten föderierten Datenbanksystems

In den folgenden Kapiteln wird das in der Diplomarbeit entwickelte föderierte Datenbanksystem detailliert dargestellt. Bevor jedoch auf die konkrete Architektur des Systems und dessen Implementierung eingegangen wird, sollen die Anforderungen an das System und die beim Systementwurf gewählten Ansätze näher beleuchtet und begründet werden.

5.1 Das Anwendungs-Szenario

Das FDBS wurde vor dem Hintergrund eines konkreten Anwendungs-Szenarios entwickelt. Die Städte Bochum und Wattenscheid werden im Rahmen einer Eingemeindung zu einer Stadt erklärt. Hiervon betroffen sind insbesondere die kommunalen Behörden. Innerhalb der Stadtverwaltungen Bochum und Wattenscheid existieren bis zur Eingemeindung jeweils zwei Einwohnermeldeämter und Standesämter. Die Einwohnermeldeämter beider Städte setzten bis dato datenbankgestützte Systeme zur Verwaltung der Einwohnerdaten ein. Abbildung 24 zeigt die Situation vor der Eingemeindung.

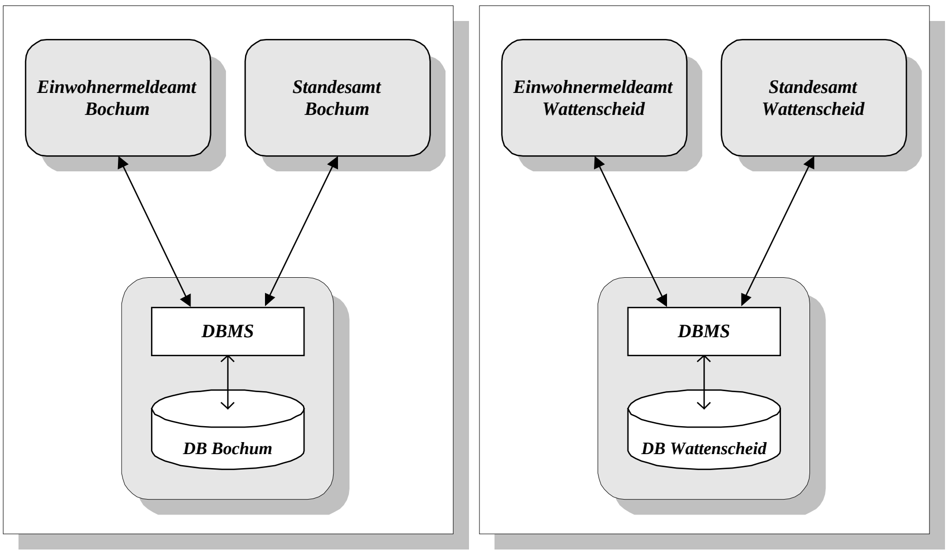


Abbildung 24 Situation vor der Eingemeindung

Die Standesämter beider Städte haben Zugriff auf die Datenbanken der Einwohnermeldeämter, um die in ihren Tätigkeitsbereich fallenden Verwaltungsvorgänge wie z.B. Hochzeiten, Scheidungen oder Geburtsanzeigen datentechnisch abzubilden.

Im Eingemeindungsvertrag hat man sich darauf geeinigt, beide Einwohnermeldeämter weiter zu betreiben, um die bestehende Bürgernähe weiterhin zu gewährleisten. Die bisherigen Standesämter sollen jedoch aus Kostengründen zu einem zentralen Standesamt fusioniert werden. Als Kompromiß einigt man sich darauf, daß das zentrale Standesamt seine Tätigkeit in neuen Räumen mit entsprechenden Ambiente für Eheschließungen aufnimmt.

Um die notwendigen organisatorischen Veränderungen möglichst schnell und kostengünstig umzusetzen, wird eine Unternehmensberatung zu Rate gezogen. Diese macht den Vorschlag, die Datenbanksysteme der oben dargestellten Behörden mithilfe eines föderierten Datenbankansatzes zu integrieren, um so die erforderliche Kooperation zu unterstützen. Die Berater untermauern ihre Systemvision mit dem Argument, daß die bestehenden Anwendungen und das darauf geschulte Personal weiterhin wie bisher ihre Tätigkeit verrichten können. Neue Anforderungen in Hinblick auf die nun notwendige Kooperation der beteiligten Behörden werde das föderierte Datenbanksystem effizient unterstützen. Abbildung 25 stellt den Lösungsvorschlag dar.

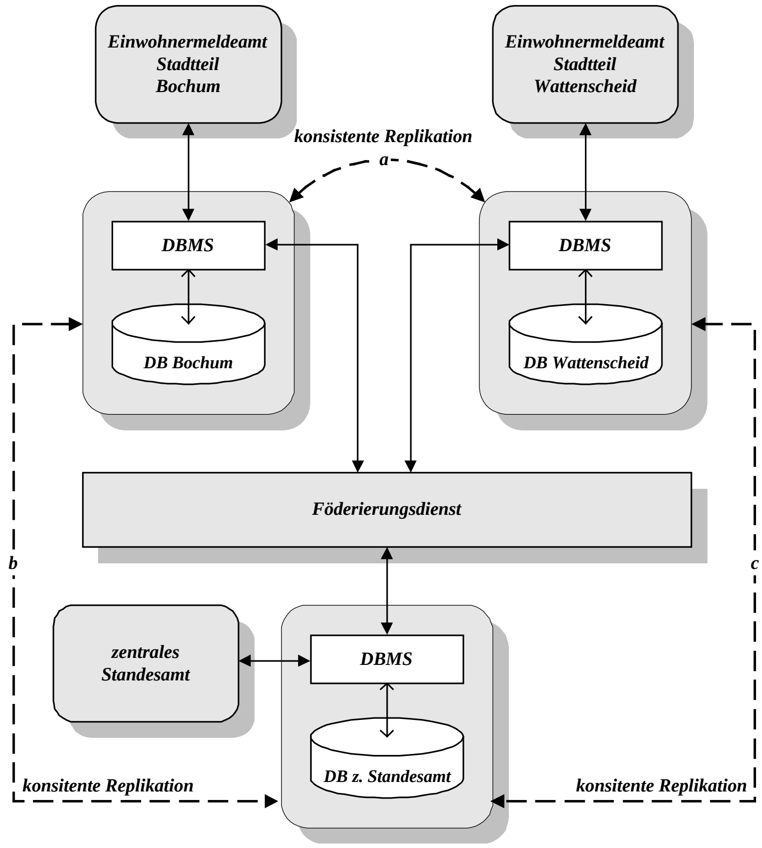


Abbildung 25 Lösungsansatz zur Integration der Behörden-DBS

Die lokalen Anwendungen der Einwohnermeldeämter der neuen Stadtteile Bochum und Wattenscheid speichern ihre Daten weiterhin in den lokalen Datenbanksystemen. Das neu geschaffene zentrale Standesamt erhält ein eigenes lokales Datenbanksystem. Der Föderierungsdienst realisiert eine konsistente Replikation der lokalen Daten der beteiligten Systeme. Dabei werden Einwohnerdaten des lokalen DBS Bochum in das lokale DBS Wattenscheid repliziert und umgekehrt (siehe Abbildung 25, Pfeil ***a***). Darüber hinaus werden alle für das zentrale Standesamt relevanten Daten der Einwohnermeldeämter Bochum und Wattenscheid in dessen lokales DBS repliziert (siehe Abbildung 25, Pfeile ***b*** und ***c***). Globale Anwendungen, etwa für statistische Auswertungen, sind in der ersten Entwicklungs-Phase des FDBS noch nicht vorgesehen.

Ein Einwohner, der beispielsweise vom Stadtteil Bochum in den Stadtteil Wattenscheid umzieht, kann sich in Zukunft immer an das Einwohnermeldeamt seiner Wahl wenden. Alle Änderungen seiner Einwohnerdaten werden durch das föderierte Datenbanksystem an alle Komponenten-Datenbanksysteme propagiert. Das zentrale Standesamt spart erheblichen Aufwand, da z.B. die durch eine Hochzeit betroffenen Personendaten bereits vorliegen. Eine Eheschließung zwischen zwei Personen, die bislang in Bochum und Wattenscheid gemeldet waren, läßt sich nun vom zentralen Standesamt mit relativ geringem datentechnischen Aufwand durchführen.

5.2 Besondere Anforderungen an das System

An das zu realisierende föderierte Datenbanksystem werden einige besondere Anforderungen gestellt. Diese werden im folgenden dargestellt:

a) Die Autonomie der zu integrierenden Komponenten-Datenbanksysteme soll nur in dem Maße eingeschränkt werden, wie es für die Kooperation der beteiligten Behörden unbedingt notwendig ist (vergleiche hierzu Kapitel 4.2). Die lokalen Anwendungen sollen ohne jede Modifikation weiterhin in ihrer gewohnten Systemumgebung (*Unabhängigkeit bei der Wahl des DBMS und der Ablaufumgebung, Ausführungsautonomie*) ihre Arbeit verrichten können. Änderungen des lokalen Datenbank-Schemas eines Komponenten-Datenbanksystems sollen durch das FDBS flexibel handhabbar sein (*Entwurfsautonomie*). Die Administratoren der Komponenten-Datenbanksysteme sollten in der Lage sein, ihre KDBS dynamisch an das FDBS an- und abzukoppeln, um beispielweise zur jeder Tageszeit ein Offline-Backup durchführen zu können (*Kooperationsautonomie*).

b) Der mögliche Ausfall eines der lokalen Datenbanksysteme oder die Unterbrechung von Netzwerkverbindungen (Netzwerkpartionierung) soll beim Systementwurf Berücksichtigung finden. Fällt beispielsweise eines der lokalen DBS aus oder ist vom FDBS aus nicht mehr erreichbar, so sollte unmittelbar nach Wiederaufnahme der entsprechenden Kooperations-Verbindung das Gesamtsystem wieder in einen konsistenten Zustand überführt werden. Der durch das FDBS realisierte Replikationsmechanismus muß für solche Situationen eine Art *asynchronen Modus* vorsehen, um Änderungen von replizierten lokalen Datenbank-Objekten nachzubearbeiten. Hierdurch wird die Verfügbarkeit und Ausfallsicherheit des Gesamtsystems erhöht. Fällt beispielsweise die Netzwerkverbindung zwischen dem lokalen DBS des zentralen Standesamtes und dem Rechner, auf dem das Föderierungssystem abläuft, aus, so können Eheschließungen dort *„offline“* durchgeführt werden. Darüber hinaus besteht auch ein enger Zusammenhang zwischen der *Kooperationsautonomie* und der oben dargestellten Anforderung an das FDBS.

c) Die Integration weiterer heterogener Datenquellen soll durch das föderierte Datenbanksystem flexibel unterstützt werden. Beispielsweise könnten im gewählten Anwendungs-Szenario zusätzliche Behörden der Kommune an der Kooperation der verteilten Datenbanken teilnehmen. Das FDBS sollte dazu Funktionen vorsehen, die eine leichte Konfigurierbarkeit des Systems unterstützen, um neue Datenquellen mit angemessenem Aufwand integrieren zu können. Dieser Aspekt schließt eine Systemunterstützung bei der Schematransformation und Schemaintegration der KDBS ein.

d) Das FDBS sollte eine möglichst große Anzahl unterschiedlicher Hardware- und Betriebssystem-Plattformen unterstützen (*Heterogenität in der Ablaufumgebung*).

e) Eine zukünftige Integration modernerer Datenbanksysteme, wie z.B. objektorientierte Datenbanksysteme, sollte ohne semantische Verluste bei der Schematransformation möglich sein. Dieser Aspekt betrifft insbesondere die Wahl des kanonischen Datenmodells (siehe Kapitel 4.4).

5.3 Der gewählte Ansatz

Im den nun folgenden Abschnitten soll der bei der Entwicklung des föderierten Datenbanksystems gewählte Ansatz skizziert und begründet werden.

5.3.1 ODMG-93 als kanonisches Datenmodell

Dem allgemeinen Trend in der Literatur folgend [BFHK95, HoKo95, HoPl95, PBE95, Rad94, Dogaz95, Xuequn95] soll ein objektorientiertes Datenmodell als kanonisches Datenmodell (KDM) des föderierten Datenbanksystems dienen. Hinsichtlich der Wahl des kanonisches Datenmodells eines föderierten Datenbanksystems werden in der Literatur zwei unterschiedliche Auffassungen vertreten. In [SCG91] wird gefordert, daß die Ausdrucksfähigkeit des KDM größer (oder gleich) als die Ausdrucksfähigkeit der den Komponenten-Datenbanksystemen zugrundeliegenden Datenmodelle sein sollte, da ansonsten ein Teil der Semantik der lokalen Schemata beim Prozeß der Schematransformation verloren geht. Da die Komponenten-Schemata, als Ergebnis der Schematransformation, die Eingabe für den Prozeß der Schemaintegration darstellen, würde sich dieser semantische Verlust auch auf die föderierten Schemata auswirken. Die hierzu konträre Auffassung schlägt ein semantisch armes Datenmodell als KDM vor [ScSa96], um die Anzahl der Konflikte bei der Schemaintegration möglichst gering zu halten. Dieser Ansatz steht vor dem Hintergrund einer automatischen Schemaintegration. Wie dabei die semantischen Verluste aufgefangen werden können, ist Gegenstand aktueller Forschung.

In dieser Diplomarbeit soll das Objektmodell des ODMG-93-Standards als kanonisches Datenmodell verwandt werden. Im folgenden werden die Vorteile und Nachteile eines objektorientierten Datenmodells, insbesondere des ODMG-93-Standards, als KDM gegenübergestellt:

- Datenbank-Schemata werden heute häufig durch OO-Techniken oder Entity-Relationship-Modelle modelliert. Viele Aspekte (z.B. Generalisierung / Spezialisierung, Beziehungen zwischen Objekten) des ursprünglichen Entwurfs gehen bei der Transformation in ein traditionelles Datenbanksystem (etwa ein relationales DBS) verloren [Kroh93]. Diese Aspekte stecken implizit im Code der Datenbank-Anwendungen und sind häufig durch die ursprünglichen Entwurfs-Dokumente noch verfügbar. Diese können im korrespondierenden „Komponenten-Schema“ wieder explizit gemacht werden. Hierbei handelt es sich um eine Art Reengineering. Dieser Prozeß wird häufig als semantischer Anreicherung bezeichnet [HoPl95].

- Die Integration von objektorientierten Datenbankschemata läßt sich weitgehend ohne semantische Verluste durchführen [HoPl95, PBE95].

- ODMG-93 stellt den zukünftigen Standard für objektorientierte Datenbanksysteme dar:

    - Das ODMG-93-Objektmodell ist generell programmiersprachenunabhängig.

    - Mit *Object-Definition-Language* (*ODL*) existiert bereits eine Sprache zur Definition von Datenbank-Schemata, welche mit der *Interface-Definition-Language* der OMG konform geht.

    - Die standardisierten Sprachanbindungen (bisher für C++, Smalltalk, Java) ermöglichen eine leichtere Portierbarkeit von globalen Anwendungen.

- Ein nicht objektorientiertes Datenmodell führt zu einem „*two level store*“ Ansatz [Cat94] (Abbildung 26 verdeutlicht diesen Sachverhalt). Anwendungen werden heute zumeist objektorientiert entworfen und implementiert. Eine zusätzliche Abbildung des Datenbank-Schemas auf die Anwendungsklassen ist nötig. Die daraus resultierenden Abbildungsfunktionen lesen die Daten gemäß dem zugrundeliegenden Schema und kopieren sie in die Attribute der Objekte. Demgegenüber verhält sich ein DBS, dem ein objektorientiertes Datenmodell zugrundeliegt, für den Entwickler von globalen Anwendungen wie ein objektorientiertes Datenbanksystem, welches einen transparenten Zugriff auf die persistenten Anwendungs-Objekte unterstützt. Die Dichotomie zwischen Datenbank-Schema auf der einen Seite und Anwendungs-Design auf der anderen Seite wird überwunden [Kroh93].

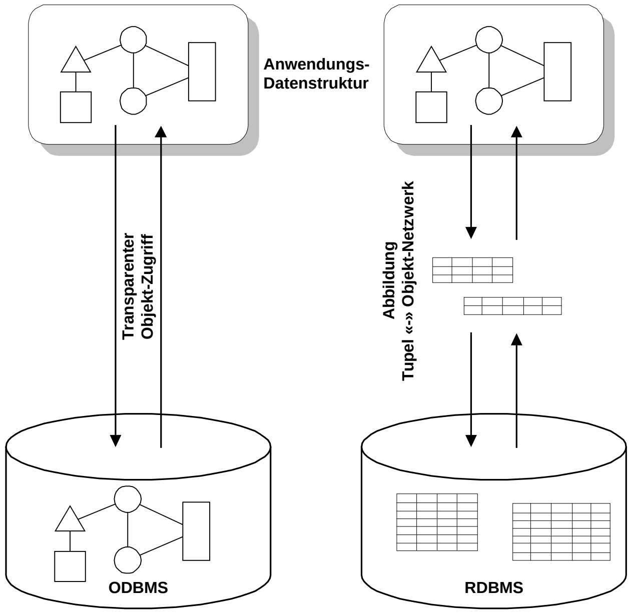


Abbildung 26 „*one level store*“ vs. „*two level store*“ Ansatz

- Eine automatische Generierung eines Komponenten-Schemas aus einem lokalen Schema, welches in einem semantisch ärmeren Datenmodell vorliegt (etwa dem relationalen Datenmodell), ist nahezu unmöglich [Hi95, HoPl95, PBE95], da bestimmte Modellierungssituationen des zu integrierenden Schemas zumeist durch unterschiedliche objektorientierte Konzepte abgebildet werden können. Darüber hinaus ist häufig in dem zu integrierenden Schema nicht offensichtlich, welches Modellierungskonzept angewandt wurde. Beispielweise kann eine Spezialisierung in einem relationalen Schema unterschiedlich umgesetzt werden und ist dabei häufig nicht von Beziehungen zwischen Datenbank-Objekten zu unterscheiden.

- Derzeit existiert im ODMG-93-Standard kein Sichtenkonzept (Probleme bei der Realisierung externer Sichten durch Filterung).

- Maßnahmen zur Realisierung einer Zugriffskontrolle sind bisher nicht im ODMG-93-Standard vorgesehen.

5.3.2 CORBA als Grundlage zur Interaktion der Systemkomponenten

Bezüglich der Heterogenität der Ablaufumgebung der Komponenten-Datenbanksysteme soll in der Diplomarbeit der Ansatz verfolgt werden, das FDBS als System verteilter, kooperierender Objekte zu konzipieren und zu implementieren. Die Kommunikation wird durch einen CORBA-kompatiblen Object Request Broker realisiert (siehe Kapitel 2). Hierdurch werden auf relativ einfache Weise die Grenzen der Ablaufumgebungen der an dem FDBS beteiligten Komponenten-DBS überwunden und ein einheitlicher

Interaktionsmechnismus der FDBS-Komponenten realisiert. Darüber hinaus erfüllt dieser Ansatz auch die Anforderung, daß das FDBS eine möglichst große Anzahl unterschiedlicher Hardware- und Betriebssystem-Plattformen unterstützen sollte, da die heute verfügbaren CORBA 2.0 Implementierungen fast alle relevanten Plattformen unterstützen und das in CORBA 2.0 verbindlich vorgeschriebene *Inter-ORB-Protokoll* (*IOP*) (siehe Kapitel 2.3.2) die Menge der erreichbaren Plattformen transitiv erweitert.

Zur Aufhebung der Datenmodell-Heterogenität soll das in Kapitel 4.5 vorgestellte Datenbank-Adapter-Konzept Verwendung finden. Abbildung 27 skizziert den kombinierten Ansatz. Die Datenbank-Adapter haben die Aufgabe, die Datenbank-Spezifika der Komponenten-Datenbanksysteme transparent zu machen (Schematransformation siehe Kapitel 4.4). Der Kern des föderierten Datenbanksystems realisiert die kontrollierte Kooperation zwischen den KDBS und kommuniziert dazu lediglich mit den Datenbank-Adaptern, welche alle dasselbe Interface haben. Die strikte Trennung von IDL-Interface-Beschreibung und Implementierung (siehe Kapitel 2.4) unterstützt diese Abstraktion. Da die Datenbank-Adapter im Zusammenhang der konsistenten Replikation aktive System-Komponenten darstellen, werden diese in Abbildung 27 auch als Datenbank-Agenten bezeichnet. Der FDBS-Kern wie auch die Datenbank-Agenten werden als CORBA-Server-Prozesse realisiert, die aus einer Menge von Objekten und dem Object-Adapter bestehen. Die Rolle (*Client / Server*) eines Objektes innerhalb einer Kooperationsbeziehung ergibt sich aus dem konkreten Kontext der Interaktion (Verallgemeinerung des *Client / Server-Konzepts*, siehe Kapitel 2.3).

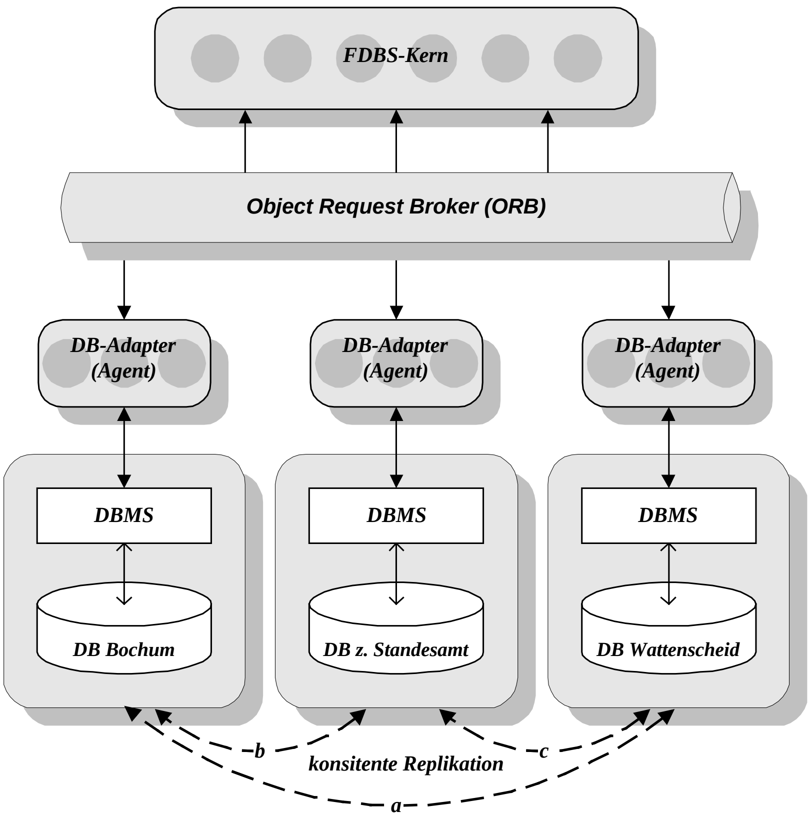


Abbildung 27 Das FDBS als System kooperierender Objekte

5.3.3 Der Publisher-Subscriber-Replikationsansatz

Zur Integration der im Anwendungs-Szenario beteiligten lokalen Datenbanksysteme soll das FDBS einen Replikationsmechanismus unterstützen, der es erlaubt, Datenbank-Objekte zwischen heterogenen Datenbanksystemen zu replizieren. Ein Datenbank-Objekt ist dabei ein Objekt im Sinne des kanonischen Datenmodells. Die besondere Problematik ergibt sich hierbei aus der Autonomie und der daraus resultierenden Heterogenität der Komponenten-Datenbanksysteme. Übliche Ansätze der Replikationskontrolle wie sie in [BeuDa96] und [Rah94] dargestellt sind, lassen sich nicht ohne weiteres anwenden, da sie für verteilte Datenbanksysteme (siehe Kapitel 4.3) konzipiert sind und ein aktives Mitwirken der beteiligten DBMS voraussetzen.

Der konzipierte Replikationsansatz wird im folgenden als *Publisher-Subscriber-Ansatz* bezeichnet und orientiert sich am Replikations-Verfahren des relationalen Datenbanksystems SQLBase [Cent1-96]. Nach der in [BeuDa96] dargestellten Klassifikation von Replikations-Verfahren, handelt es sich bei dem *Publisher-Subscriber-Ansatz*, um ein semantisches Verfahren zur Replikationkontrolle. Es ist in etwa vergleichbar mit dem Data-Patch-Verfahren, wie es in [BeuDa96] dargestellt ist. Der Ansatz verfolgt eine optimistische Kopien-Update-Strategie. Die grundlegende Annahme des Verfahrens ist, daß nicht unbedingt alle an der Replikation beteiligten Datenbanksysteme zu jedem Zeitpunkt erreichbar sind. Das Verfahren sieht eine asynchrone Propagation von DB-Objekt-Modifikationen vor. Hiermit kommt es auch den im Abschnitt 5.2 gestellten Anforderungen a) und b) nach (Autonomie der KDBS, möglicher Ausfall eines KDBS bzw. Netzwerkpartionierung).

Jedes Komponenten-Datenbanksystem wird von einem DB-Agenten überwacht (siehe Abschnitt 5.3.2), der auftretende DB-Objekt-Modifikationen entdeckt, zwischenspeichert und dem FDBS-Kern mitteilt. Der FDBS-Kern propagiert die Änderungen an die involvierten Komponenten-Datenbanksysteme bzw. deren DB-Agenten. Die Propagation der Änderungs-Operationen ist dabei von den lokalen Transaktionen der KDBS entkoppelt (*Ausführungsautonomie*). Die Granularität der betroffenen DB-Objekte ist durch das Komponenten-Schema bestimmt. Die replizierten DB-Objekte sind Instanzen der Objekttypen der Komponenten-Schemata. Die (bijektive) Schematransformations-Abbildung, welche durch die DB-Agenten realisiert wird, bestimmt, wie diese DB-Objekte in den lokalen Datenbanken gespeichert werden.

Die Objekttypen der Komponenten-Schemata stehen in einer *Publisher-Subscriber-Beziehung* zueinander (in Abbildung 28 dargestellt). Diese Beziehung spezifiziert, wie Instanzen der Extension des *Publisher-Objekttyps* in entsprechende Instanzen des *Subscriber-Objekttyps* überführt werden. Wird ein neues Objekt eines *Publisher-Objekttyps* erzeugt (*Publisher-Copy*), so werden Replikate (*Subscriber-Copies*) in allen KDBS seiner *Subscriber* erzeugt (in Abbildung 28 gestrichelte Pfeile *(1)*). Eine Änderung eines *Publisher-Copy-Objektes* wird an alle *Subscriber-Copy-Objekte* propagiert. Wird ein *Subscriber-Copy-Objekt* geändert, so unterliegen auch das *Publisher-Copy-Objekt* und dessen übrige *Subscriber-Copy-Objekte* dieser Änderung (in Abbildung 28 gestrichelte Pfeile *(2)*). Löschoperationen werden ebenfalls von einem *Publisher* zu den *Subscribern* propagiert und umgekehrt. Bezüglich Änderungs- und Lösch-

Operationen sind die Rollen von *Publisher-Copy-Objekt* und *Subscriber-Copy-Objekt* also symmetrisch. Das Erzeugen eines neuen DB-Objektes wird nur von einem *Publisher-Objekttyp* zu dessen *Subscriber-Objekttypen* propagiert. Es ist jedoch möglich, daß ein Objekttyp gleichzeitig Publisher und Subscriber eines anderen Objekttyps ist (in Abbildung 28 die Objektypen ***A*** und ***C***). Die Spezifikation aller Publisher-Subscriber-Beziehungen ist Teil des föderierten Schemas.

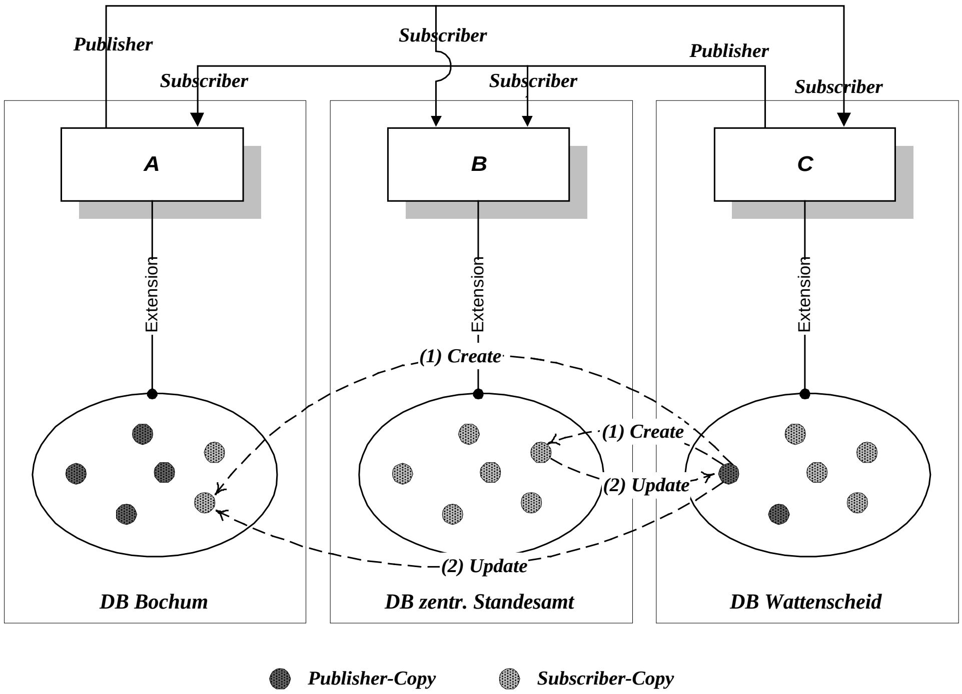


Abbildung 28 Publisher-Subscriber-Beziehung

Der Publisher-Subscriber-Ansatz verfolgt eine optimistische Update-Strategie. Aufgrund der Entkopplung von lokalen Transaktionen und der Propagation von Änderungs-Operationen können dabei Update-Konflikte entstehen. Abbildung 29 stellt einen solchen Konflikt beispielhaft dar. Konflikte treten dann auf, wenn während der Propagierung einer relevanten Änderungs-Operation lokale Benutzer betroffene Replikate modifizieren oder mehrere Kopien desselben Objektes quasi gleichzeitig geändert werden. Darüber hinaus entstehen Konflikte auch im Zusammenhang mit Netzpartionierungen oder der zeitweisen Abkopplung eines Komponenten-Datenbanksystems vom FDBS (*Kooperationsautonomie*).

Der Replikation-Mechanismus sieht einen Konflikt-Manager vor, der Konflikte durch Regeln oder durch die Delegation an andere System-Instanzen behandelt. Durch Regeln können Konflikte beispielsweise mithilfe von Prioritätslisten oder dem zeitlichen Ablauf der Änderungs-Operationen aufgelöst werden. Auf numerische Objekt-Attribute kann im Konfliktfall unter Umständen eine Aggregation-Regel angewandt werden, bei der sich der neue Wert des Objekt-Attributs aus den Differenzen der in Konflikt stehenden Kopien ergibt. Häufig ist es aber notwendig, Konflikte durch semantisches Wissen der Diskurswelt aufzulösen. Hierzu werden andere System-Instanzen (z.B. externe Programme, Administrator, Benutzer) mit in die Konfliktauflösung einbezogen.

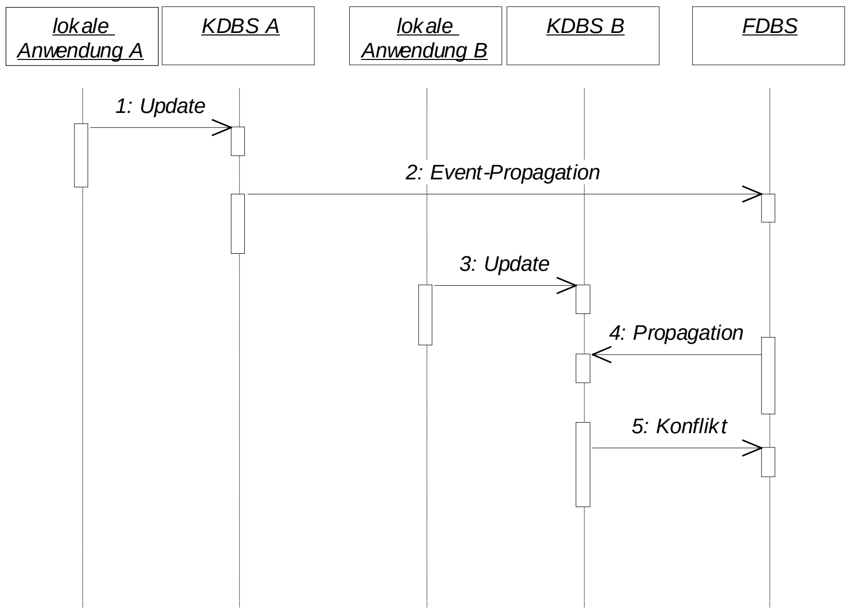


Abbildung 29 Beispiel eines Update-Konfliktes

Der gewählte Ansatz geht davon aus, daß Konflikte relativ selten auftreten und durch semantisches Wissen über die Realwelt-Objekte aufgelöst werden können, was im betrachteten Behörden-Anwendungs-Szenario der Fall ist. Um einen stärkeren Korrektheitsbegriff zu unterstützen, wäre eine Einschränkung der Autonomie der KDBS notwendig. Der in [BeuDa96] dargestellte Ein-Kopien-Äquivalenz-Korrektheitsbegriff ist nur durch eine Koppelung der Änderungs-Transaktionen und Propagations-Transaktionen zu erreichen. Dies ist aber zumeist bei hetrogenen Komponenten-Datenbanksystemen nicht möglich, wenn weiterhin lokale Transaktionen zugelassen sind, was ja gerade eine wesentliche Eigenschaft von föderierten Datenbanksystemen ist (siehe Kapitel 4).

Kapitel 6 Architektur und Entwurf des entwickelten föderierten Datenbanksystems

Das nun folgende Kapitel soll die Architektur des entwickelten föderierten Datenbanksystems und dessen Entwurf darstellen. Dabei wird zunächst im Abschnitt 6.1 die Architektur vorgestellt. Die Abschnitte 6.2 bis 6.6 beschreiben den Fein-Entwurf der Systemkomponenten des FDBS aus der Perspektive des FDBS-Kerns (siehe Kapitel 5.3.2). Hierbei wird bereits auf die Datenbank-Adapter bzw. deren IDL-Interface eingegangen. Der detailierte Entwurf der Datenbank-Adapter ist jedoch Gegenstand des Kapitels 7.

6.1 Die Systemkomponenten in der Übersicht

Dieser Abschnitt hat zum Ziel, die Systemkomponenten des in der Diplomarbeit entwickelten föderierten Datenbanksystems im Zusammenhang darzustellen und ihre Funktion zu skizzieren. Die sich hieran anschließenden Abschnitte beschreiben den Fein-Entwurf des Systems. Die Architektur des FDBS (in Abbildung 30 dargestellt) basiert auf den im Kapitel 5 vorgestellten Ansätzen. Die in Abbildung 30 schraffiert dargestellten Systemkomponenten stellen konzeptionelle Überlegungen dar, werden aber im Rahmen der Diplomarbeit nicht implementiert. Sie sind jedoch in der Architektur angegeben, um deren Vollständigkeit zu gewährleisten und stellen Anknüpfungspunkte für weitere Arbeiten dar.

Das FDBS wurde als System verteilter Objekte, die mithilfe eines CORBA-kompatiblen Object Request Brokers kooperieren, konzipiert. Im Mittelpunkt der Architektur steht der FDBS-Kern, welcher die kontrollierte Kooperation der Komponenten-Datenbanksysteme realisiert und den globalen Anwendungen eine homogenisierte Sicht auf die verteilten heterogenen Datenbestände bietet. Die Meta-Daten des FDBS (die föderierten Schemata, die Lokalisation einzelner DB-Objekte und die Komponenten-Schemata) werden in einer Hilfs-Datenbank verwaltet. Diese Aufgabe übernimmt das ODMG-93-konforme Datenbanksystem POET 4.0 (siehe Anhang A1).

Die Datenbank-Spezifika der Komponenten-Datenbanksysteme werden durch die Datenbank-Adapter gekapselt. Der FDBS-Kern hat also nur indirekt Zugriff auf die KDBS. Aufgrund ihrer aktiven Rolle im System werden diese auch als DB-Agenten bezeichnet. Die wesentliche Aufgabe der DB-Adapter besteht in der Schematransformation in das kanonische Datenmodell (Überwindung der Datenmodell-Heterogenität). Die gesamte Kommunikation zwischen dem FDBS-Kern und den Datenbank-Adaptern erfolgt über den Object Request Broker. Dies gilt ebenso für die Kommunikation zwischen FDBS-Kern und den globalen Anwendungen.

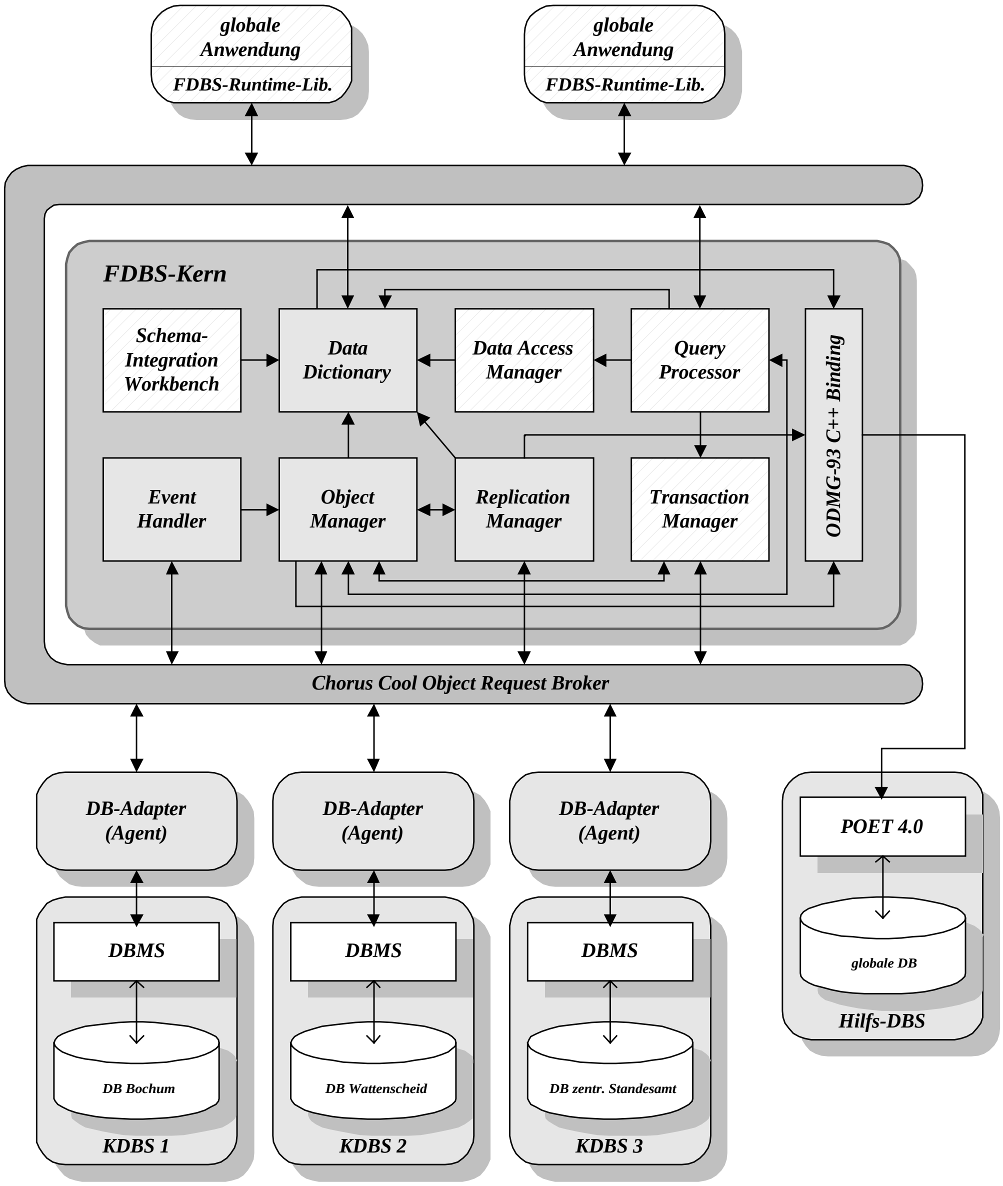


Abbildung 30 Die Architektur des entwickelten FDBS

Die innere Struktur des FDBS-Kerns ist in Abbildung 30 dargestellt. Die einzelnen Systemkomponenten und deren Aufgabe werden im folgenden skizziert:

- Im ***Data-Dictionary*** werden die Meta-Daten des FDBS verwaltet. Es beinhaltet die Komponenten-Schemata, die föderierten Schemata und die Spezifikation der Replikations-Beziehungen (siehe Kapitel 5.3.3). Das *Data-Dictionary* besteht aus eine Menge persistenter Objekte, die in der POET-Datenbank gespeichert sind und die einzelnen Schema-Elemente (Objekttypen und deren Eigenschaften, ...) beschreiben. Die im *Data-Dictionary* verwalteten Meta-Daten des FDBS werden selbst wieder durch ein Datenbank-Schema beschrieben. Dieses, im folgenden als „*Meta-Schema*“ bezeichnete Datenbank-Schema definiert, wie die Meta-Daten des FDBS in der ODMG-93-konformen POET-Datenbank abgelegt werden. Die Anwendung des POET-Schema-Precompilers auf dieses *Meta-Schema* liefert die ODMG-93-C++-Sprachanbindung (siehe Kapitel 3.4), welche das Interface des *Data-Dictionary* zum POET-System darstellt. Um neue Komponenten-Datenbanksysteme möglichst einfach in die Datenbank-Föderation zu integrieren, ermöglicht das *Data-Dictionary* einen dynamischen Import von Komponenten-Schemata (siehe auch Anforderung (c) in Kapitel 5.2).

- Der ***Event-Handler*** nimmt Ereignisse der Datenbank-Agenten entgegen und organisiert deren parallele Bearbeitung durch Delegation an die anderen Systemkomponenten des FDBS-Kerns. Mögliche Ereignisse sind hierbei die Erzeugung, die Modifikation oder das Löschen eines DB-Objektes innerhalb eines Komponenten-Datenbanksytems. Wurde ein Ereignis vom FDBS-Kern vollständig bearbeite, so benachrichtigt der *Event-Handler* den DB-Agenten, daß er dieses Ereignis vergessen kann.

- Der ***Object-Manager*** verwaltet zu allen lokalen DB-Objekten entsprechende Stellvertreter-Objekte (Proxy-Objekte), die einen eindeutigen und dauerhaften *Object-Identifier* (*OID*) haben [EK91]. Der *Object-Manager* realisiert somit das Konzept der wertunabhängigen Objekt-Identität (siehe Kapitel 3.2). Die Stellvertreter-Objekte kapseln Informationen über die Lokalisation und den Status von lokalen DB-Objekten. Darüber hinaus realisieren sie eine Abbildung auf die lokale Identität des korrespondierenden lokalen DB-Objektes. Hierdurch wird die Verteilung der DB-Objekte transparent gemacht. Der *Object-Manager* greift bei seiner Arbeit auf das *Data-Dictionary* zurück, um DB-Objekte zu materialisieren. Er ist das Bindeglied zwischen dem *Event-Handler* und dem *Replication-Manager*, der den Replikations-Mechanismus des FDBS kontrolliert.

- Der ***Replication-Manager*** realisiert das im Kapitel 5.3.3 dargestellte *Publisher-Subscriber-Replikations-Verfahren* und stützt sich bei dieser Aufgabe auf das *Data-Dictionary* und den *Object-Manager* ab. Das *Data-Dictionary* verwaltet dabei die notwendigen Informationen über die Replikations-Beziehungen von Objekttypen (Kapitel 5.3.3) mit deren Hilfe der *Replication-Manager* die Abbildung von *Publisher-Copy-Objekten* (bzw. Änderungs-Operationen auf diesen) auf die *Subscriber-Copy-Objekte* durchführt. Die *Publisher-Subscriber-Instanz-Beziehung* einzelner DB-Objekte wird im Hilfs-Datenbanksystem des FDBS (POET) persistent verwaltet. Der *Object-Manager* bietet dem *Replication-Manager* den verteilungstransparenten Zugriff auf die lokalen DB-Objekte und koordiniert die Kooperation mit den DB-Agenten.

- Die Definition von Zugriffsrechten für globale Benutzer des FDBS und deren Einhaltung ist Aufgabe des ***Data-Access-Manager***. Die Zugriffsrechte sind dabei ein Bestandteil der vom Data-Dictionary verwalteten Schemata.

- Der ***Query-Processor*** übernimmt die Aufgabe, Datenbankanfragen von globalen Anwendungen zu verarbeiten. Ankommende Anfragen werden dabei vom *Query-Processor* analysiert, mit dem *Data-*

Access-Manager abgeglichen und in lokale Subanfragen zerlegt, aus deren Ergebnissen schließlich das Ergebnis der globalen Anfrage konstruiert wird. Bei der Zerlegung von Anfragen stützt sich der *Query-Processor* auf die im *Data-Dictionary* verwalteten Meta-Daten ab. Darüber hinaus spielt der *Query-Processor* eine wichtige Rolle bei der Realisierung von Sichten, welche auf Datenbankanfragen basieren [Souza95, BFN94, Hei93].

- Die Unterstützung von globalen Transaktionen ist Aufgabe des ***Transaction-Manager***. Er koordiniert die Ausführung der lokalen Subtransaktionen mittels eines verteilten Transaktions-Protokolls und bietet den globalen Anwendungen Operationen (*begin, commit, rollback*) zur Kontrolle der globalen Transaktion [BGS92]. Ein möglicher Ansatz für die Transaktionskontrolle in einem föderierten Datenbanksystem stellt der im Kapitel 2.2.2 vorgestellte *Transaction-Service* von CORBA dar.

- Die ***Schema-Integration-Workbench*** ist eine Sammlung von Werkzeugen, die den System-Integrator bei der sehr schwierigen Aufgabe der Schemaintegration unterstützen.

Die innere Struktur der Datenbank-Adapter ist ebenso wie die des FDBS-Kerns in Abbildung 31 dargestellt.

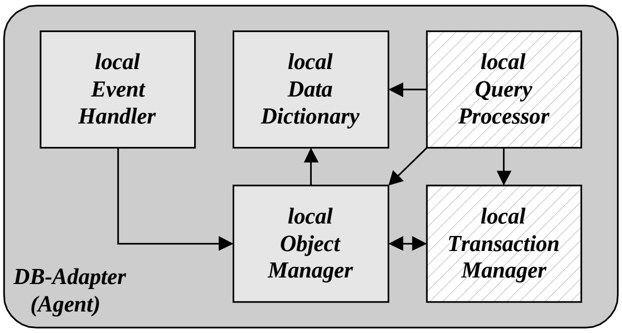


Abbildung 31 Die innere Struktur der Datenbank-Adapter

Die IDL-Interface-Beschreibung der Datenbank-Adapter ist der Kontrakt für die Kooperation mit dem FDBS-Kern. Diese impliziert eine Reihe von Systemkomponenten, die jeder Datenbank-Adapter implementieren muß:

- Das ***lokale Data-Dictionary*** verwaltet das Komponenten-Schema des Komponenten-Datenbanksystems und dessen bijektive Abbildung auf das lokale Schema (Schematransformation). Das Komponenten-Schema besteht aus einer Menge von Objekten mit IDL-Interface, welche die Schema-Elemente gemäß dem kanonischen Datenmodell beschreiben. Auf diese Objekte greift das *Data-Dictionary* des FDBS-Kern zu, um das Komponenten-Schema des KDBS zu importieren (beim ersten Ankoppeln des KDBS) oder mit einer bereits vorhandenen Kopie abzugleichen.

- Der ***lokale Object-Manager*** transformiert Operationen auf den DB-Objekten des Komponeten-Schemas entsprechend der bijektiven Schematransformations-Abbildung in Operationen bezüglich des lokalen Schemas. Darüber hinaus hat er die Aufgabe, die DB-Objekte des Komponeten-Schemas mit eindeutigen, wertunabhängigen und dauerhaften *lokalen Object-Identifiern* (*LOID*) zu versehen. Dadurch wird gewährleistet, daß alle KDBS gegenüber dem FDBS-Kern dasselbe Objekt-Identitäts-Konzept unterstützen [EK91].

- Das Entdecken von lokalen Ereignissen wie das Erzeugen, das Löschen oder die Modifikation von DB-Objekten ist Aufgabe des ***lokalen Event-Handlers***. Diese Ereignisse werden von ihm zwischengespeichert und an den *Event-Handler* des FDBS-Kerns weitergeleitet, der dann deren weitere Verarbeitung delegiert (siehe oben).

- Die durch eine globale Datenbankanfrage entstehenden lokalen Subanfragen werden vom ***local Query-Processor*** in lokale Anfragen gemäß dem lokalen Datenmodell transformiert. Diese Transformation ist bestimmt durch die bijektive Schematransformations-Abbildung, welche vom *lokalen Data-Dictionary* verwaltet wird. Dabei wird auch die Heterogenität der DB-Anfragesprache des lokalen Datenmodells überwunden.

- Der ***lokale Transaction-Manager*** ist die lokale Instanz des verteilten Transaktions-Protokolls (siehe oben).

6.2 Initialisierung des FDBS und die Ankopplung von Datenbank-Adaptern

Wie im letzten Abschnitt bereits deutlich wurde, besteht der FDBS-Kern aus einer Reihe von Systemkomponenten, die als verteilte Objekte realisiert werden. Der äußere Rahmen des FDBS-Kerns wird dabei ebenfalls als verteiltes Objekt realisiert. Dieses Objekt ist eine Instanz der Klasse *CFDBSKernel* (siehe Abbildung 32), dessen IDL-Interface in Abbildung 35 angegeben ist. Es organisiert die Initialisierung der weiteren Systemkomponenten des FDBS-Kern beim Hochfahren des Systems und ermöglicht das An- und Abkoppeln der Datenbank-Adapter. Beim Systemstart des FDBS erzeugt das *CFDBSKernel-Objekt* jeweils ein Objekt der Klassen *CFDBSDataDictManager*, *CFDBSObjectManager* und *CFDBSEvent-Handler,* welche die weitere Initialisierung des Systems vornehmen. Das *CFDBSObjectManager-Objekt* erzeugt den *Replication-Manager,* der Instanz der Klasse *CFDBSReplicationManager* ist.

Das Attribut *Name* der Klasse *CFDBSKernel* dient der Identifizierung des FDBS-Kerns mittels des *Object Requester Broker Naming Service* (siehe Kapitel 2.2.2). Der *Naming Service* verwaltet dabei die Abbildung der Objekt-Referenz des *CFDBSKernel-Objekts* und dessen Namen. Mithilfe dieses Namens erhalten die Datenbank-Adapter Zugriff auf den FDBS-Kern.

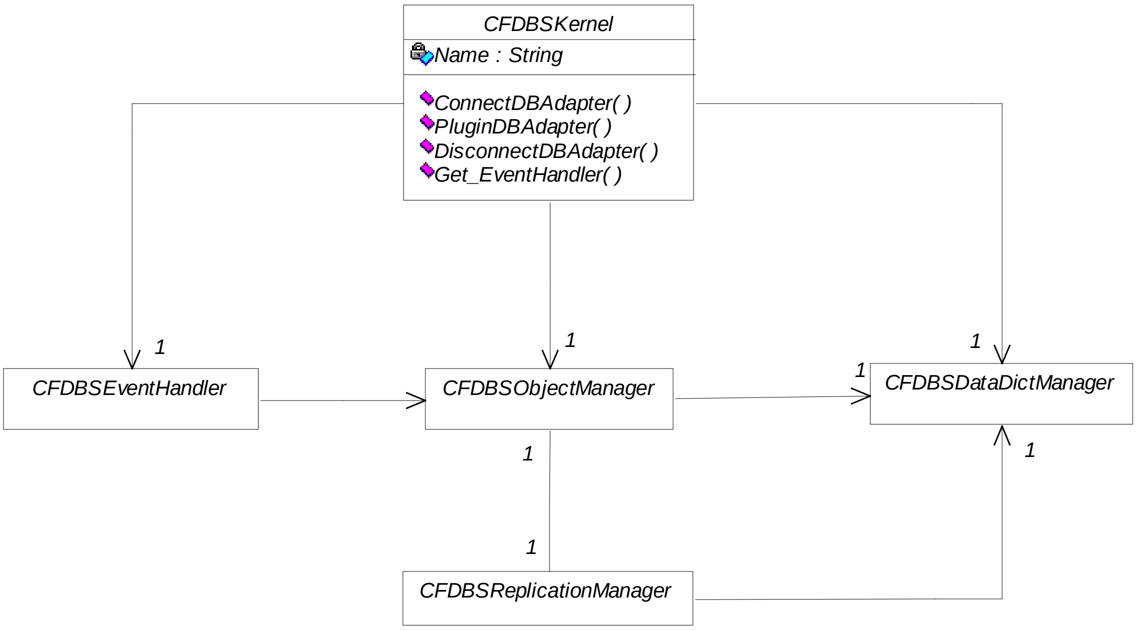


Abbildung 32 Modellierung des FDBS-Kern

Zum An- und Abkoppeln der Datenbank-Adapter stellt das *CFDBSKernel-Objekt* die Operationen *PluginDBAdapter*, *ConnectDBAdapter* und *DisconnectDBAdapter* zur Verfügung. Bevor jedoch ein Komponenten-Datenbanksystem an der Föderation teilnehmen kann, muß dessen Komponenten-Schema vom FDBS-Kern importiert und integriert werden. Abbildung 33 zeigt die Interaktion der beim Schema-Import beteiligten Objekte. Der Datenbank-Adapter besorgt sich zunächst vom *ORB* bzw. dessen *Naming Service* die Objekt-Referenz des *CFDBSKernel-Objekts*. Dies geschieht durch eine Folge von Operationen, welche in Abbildung 33 durch die Operation *Get_FDBSObjectRef* zusammengefaßt sind. Der Import eines Komponenten-Schemas wird durch die Operation *PluginDBAdapter* eingeleitet. Das *CFDBSKernel-Objekt* delegiert diese Aufgabe an den *DataDictinaryManager* (Instanz der Klasse *CFDBSDataDictManager*), indem er dessen Operation *Add_DataSource* aufruft. Dieser fragt den Datenbank-Adapter zunächst nach der Objekt-Referenz seines lokalen *DataDictinaryManager* (*Get_DataDictManager* in Abbildung 33). Nun findet der eigentliche Import des Komponenten-Schemas statt (durch *Get_ObjTypes* in Abbildung 33 angedeutet), dessen genauer Ablauf im Abschnitt 6.3.3 dargestellt wird. Verlief der Schema-Import erfolgreich, so terminiert die Operation *Add_DataSource* mit dem Rückgabewert TRUE und die Kontrolle geht wieder an die Operation *PluginDBAdapter* zurück, welche dann ebenfalls TRUE zurückliefert. Der Datenbank-Adapter weiß nun, daß sein Komponenten-Schema erfolgreich importiert wurde und beendet die Verbindung zum FDBS durch den Aufruf der Operation *DisconnectDBAdapter* des *CFDBSKernel-Objektes*. Die Administratoren des FDBS können nun mit der Schemaintegration beginnen.

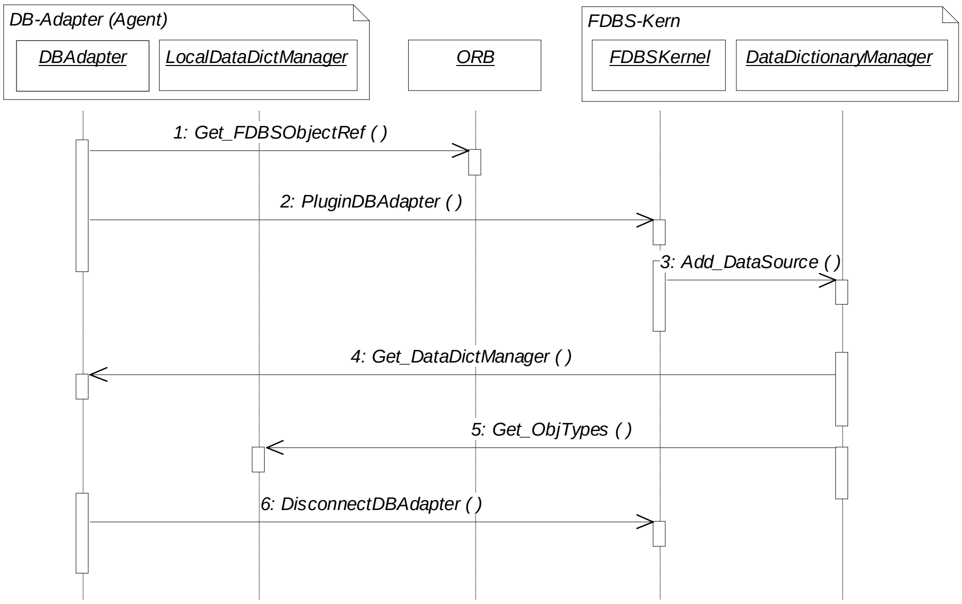


Abbildung 33 Import eines Komponenten-Schemas als Sequence-Diagramm

Das Ankoppeln eines Komponenten-Datenbanksystems, dessen Komponenten-Schema bereits erfolgreich importiert und integriert wurde, geschieht mittels der Operation *ConnectDBAdapter*. Abbildung 34 zeigt den Ablauf der Ankoppelung, bei der zunächst die importierte Kopie des Komponenten-Schemas abgeglichen wird (*CompareComponentSchema*).

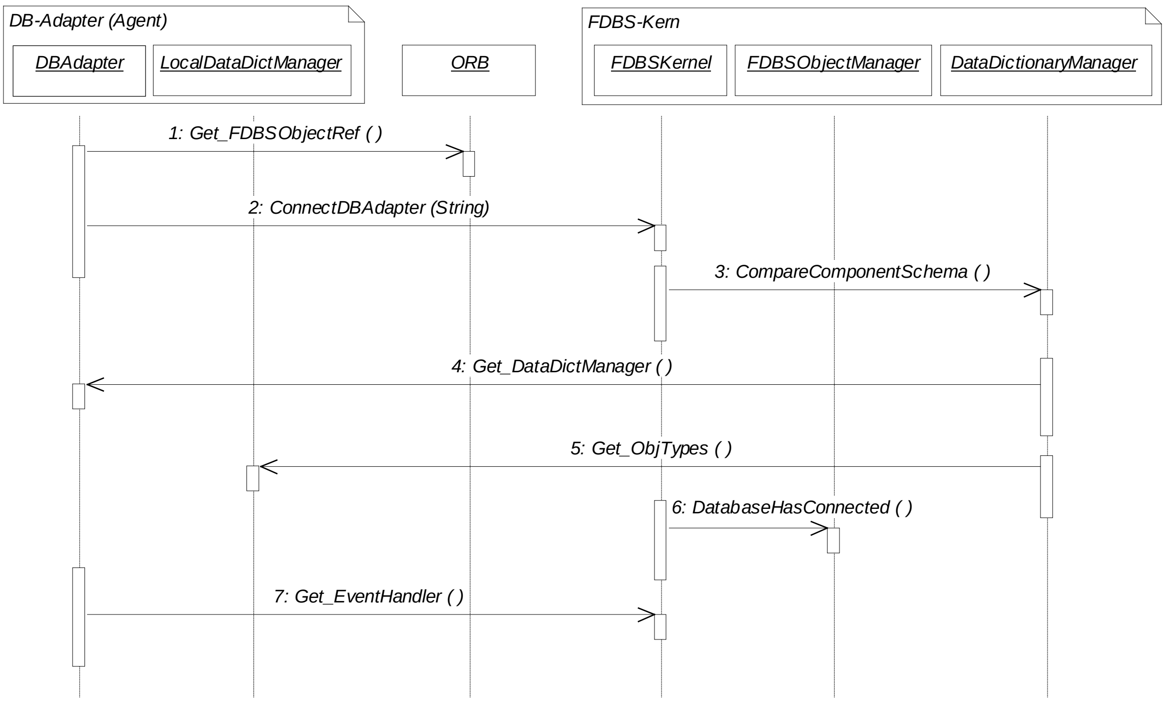


Abbildung 34 Ankoppeln eines Datenbank-Adapters als Sequence-Diagramm

Den Abgleich des Komponenten-Schemas erledigt der *DataDictionaryManager* des FDBS-Kerns in Kooperation mit seinem lokalen Pendant, wie es auch schon beim Schema-Import dargestellt wurde. Stimmen beide Schemata überein, so informiert das *CFDBSKernel-Objekt* zunächst den *FDBSObjectManager* (Instanz der Klasse *CFDBSObjectManager*) darüber, daß ein Komponenten-Datenbanksystem erfolgreich angebunden wurde (*DatabaseHasConnected*), bevor die Operation *ConnectDBAdapter* schließlich mit dem Rückgabewert TRUE terminiert. Der *FDBSObjectManager* veranlaßt, daß alle zwischenzeitlichen Änderungen von DB-Objekten im Komponenten-Datenbanksystem nachgezogen werden. Eine detaillierter Darstellung dieses Vorgangs erfolgt in Abschnitt 6.5. Nach der erfolgreichen Ankoppelung verschafft sich der Datenbank-Adapter die Objekt-Referenz des *FDBSEventHandlers* (*Get_EventHandler* siehe Abbildung 35) und weist seinen lokalen Event-Handler an, die Event-Verarbeitung aufzunehmen. Die Event-Verarbeitung erfolgt durch die Kooperation zwischen den lokalen Event-Handlern und dem *FDBSEventHandler*.

```
interface CFDBSEventHandler;

interface CFDBSKernel {
        boolean ConnectDBAdapter(in string DBName);
        boolean PluginDBAdapter(in string DBName);
        void DisconnectDBAdapter(in string DBName);
        void Get_EventHandler(out CFDBSEventHandler EvntHandler);
};
```

Abbildung 35 IDL-Interface-Beschreibung des FDBSKernel-Objektes

6.3 Das FDBS-Data-Dictionary

Eine wesentliche Komponente des föderierten Datenbanksystems ist das Data-Dictionary. Diesem kommt die Aufgabe zu, die Meta-Daten des föderierten Datenbanksystems entsprechend dem kanonischen Datenmodell zu verwalten. Die Meta-Daten beschreiben die Struktur der Daten, welche das föderierte Datenbanksystem seinen Benutzern direkt oder indirekt zur Verfügung stellt. Physisch gespeichert sind die Daten in den Komponenten-Datenbanksystemen. Diese erlauben dem FDBS den Zugriff auf einen Teil ihres Datenbestandes. Die Sicht, welche das FDBS auf ein Komponenten-Datenbanksystem hat, wird als lokales Schema bezeichnet [SL90]. Ein lokales Schema entspricht also einem externen Schema oder dem konzeptionellen Schema eines Komponenten-Datenbanksystem (vergleiche hierzu 3-Ebenen-ANSI/SPARC-Architektur [Date90]). Im letzteren Fall hätte das FDBS Zugriff auf den gesamten Datenbestand des Komponenten-Datenbanksystems. Die lokalen Schemata stellen den Ausgangspunkt für das föderierte Datenbanksystem dar. Gemäß der 5-Ebenen-Schema-Architektur nach Sheth und Larson [SL90] werden die Meta-Daten in Schema-Ebenen organisiert (siehe Kapitel 4.4). Ein Schema umfaßt dabei eine Menge von Objekttyp-Definitionen und steht in Beziehung zu einem oder mehreren Schemata der untergeordneten Ebene, aus denen es abgeleitet wurde. Direkt sichtbar für die Benutzer des FDBS sind die Objekttypen des föderierten Schemas bzw. der externen Schemata, welche durch Filterung aus dem föderierten Schema gewonnen werden. Das föderierte Schema ergibt sich aus der Schemaintegration der Export-Schemata. Diese wiederum werden aus den Komponenten-Schemata abgeleitet. Hierbei ist es auch möglich, daß ein Komponenten-Schema direkt in das föderierte Schema eingeht, sofern nicht die Notwendigkeit einer Filterung der Daten des Komponenten-Schemas besteht. Die Komponenten-Schemata bilden die lokalen Schemata der Komponenten-Datenbanksysteme auf das kanonische Datenmodell ab (Schematransformation). Auf dieser Schema-Ebene ist also bereits die Datenmodell-Heterogenität, der zu integrierenden Komponenten-Datenbanksysteme, überwunden.

Da das in dieser Diplomarbeit entwickelte FDBS keine Zugriffskontrolle und keine globalen Applikationen vorsieht, werden die Export-Schemata (dritte Schema-Ebene) und externen Schemata (fünfte Schema-Ebene) im Entwurf des Data-Dictionary nicht berücksichtigt. Die Aufgabe des Data-Dictionary besteht nunmehr darin, die föderierten Schemata und die Komponenten-Schemata entsprechend dem kanonischen Datemodell zu verwalten.

Im Objektmodell des ODMG-93-Standards [Cat94] besteht ein Datenbank-Schema aus einer Menge von Objekttyp-Definitionen, die zueinander in Beziehung stehen. Eine Objekttyp-Definition beschreibt dabei die Charakteristika einer Menge von Objekten mit ähnlichem Verhalten und gleichem Zustandsraum. Die Charakteristika eines Objekttyps zerfallen in Operationen und Eigenschaften (siehe Kapitel 3.2). Eigenschaften können Attribute oder Beziehungen sein. Das in dieser Arbeit zu konzipierende Data-Dictionary beschränkt sich darauf, die Eigenschaften, also die Attribute und Beziehungen von Objekttypen, zu verwalten. Operationen auf Objekttypen sind nicht Gegenstand dieses Systems. Diese Einschränkung wurde getroffen, um den Implementierungsaufwand in der Diplomarbeit einzuschränken.

Sämtliche Typinformationen des föderierten Datenbanksystems werden persistent in einer Hilfsdatenbank verwaltet. Diese Hilfsdatenbank wird durch das objektorientierte Datenbankmanagementsystem POET verwaltet [Poet4-96]. Abbildung 36 zeigt einen Ausschnitt des objektorientierten Datenbank-Schemas des Data-Dictionary in UML-Notation [Oest96]. Dieses Schema wurde dann in ein korrepondierendes ODMG-93-Schema umgesetzt und auf dem POET Datenbanksystem implementiert. Um Begriffskonflikten vorzubeugen, wird an dieser Stelle folgende Konvention getroffen. Der Begriff **Objekttyp** bezieht sich im folgenden auf die vom Meta-System zu verwaltenden Datentypen, während die im Datenbank-Schema des Data-Dictionary angegebenen Typen als **Klassen** bezeichnet werden. Instanzen dieser Klassen repräsentieren die vom Data-Dictionary verwalteten Objekttypen. Zwischen den Begriffen **Klasse** und **Objekttyp** besteht also nach obiger Konvention eine Typ-Instanz-Beziehung. Der Begriff Objekt wird als Synonym für die Instanzen eines Objekttyps verwendet.

6.3.1 Organisation der Schemata

Das Data-Dictionary organisiert die Objekttypen in zwei Schema-Ebenen (siehe Abbildung 36). Die Objekttypen der Komponenten-Schemata der KDBS werden als *lokale Objekttypen* bezeichnet und sind Instanzen der Klasse *CLocalObjType*. Die Klasse *CDataSource* modelliert die Komponenten-Datenbanksysteme (*Datenquellen*). Ihre Instanzen stehen in einer *1:n-Beziehung* zu den importierten *lokalen Objekttypen*. Ein *lokaler Objekttyp* ist also genau einer Instanz von *CDataSource* zugeordnet (*ImportedByDataSource*). Eine *Datenquelle* zusammen mit allen von ihr importierten *lokalen Objekttypen* (*ImportedObjTypes*) modelliert also ein Komponenten-Schema.

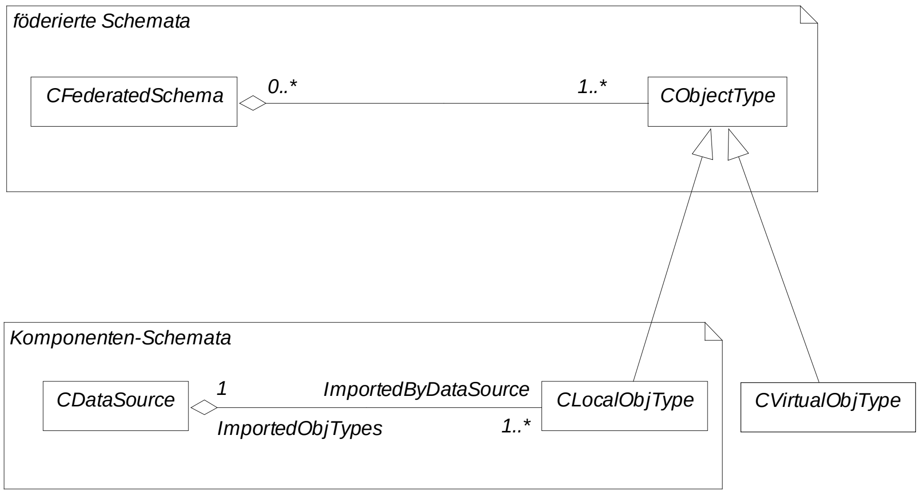


Abbildung 36 Organisation der Schemata im Data-Dictionary

Die föderierten Schemata werden durch die Klasse *CFederatedSchema* modelliert. Dabei beinhaltet ein föderiertes Schema (Instanz von *CFederatedSchema*) mehrere Objektypen, welche als Instanzen der Klasse *CObjectType* modelliert werden. *CObjectType* ist die Generalisierung der Klassen *CLocalObjType* und *CVirtualObjType* und ist eine abstrakte Klasse. Von dieser Klasse können also keine Objekte instanziiert werden. Vielmehr beschreibt sie die gemeinsamen Eigenschaften von Instanzen, der von ihr abgeleiteten Klassen. *Virtuelle Objekttypen* (Instanzen der Klasse *CVirtualObjType*) dienen der Sichtenbildung. Ein *virtueller Objekttyp* basiert dabei auf einem oder mehreren anderen Objekttypen (Instanzen der Klasse *CObjectType*), von denen er abgeleitet wurde. *Virtuelle Objekttypen* sind Gegenstand des Abschnitts 6.3.4.

6.3.2 Die Repräsentation von Objektypen

Im ODMG-93-Objektmodell lassen sich generell zwei Arten von Objekttypen unterscheiden, die Systemtypen und die benutzerdefinierten Objekttypen. Die Systemtypen dienen als Wertebereiche für die

Eigenschaften von benutzerdefinierten Objekttypen. In der ODMG-93-Terminologie werden die Systemtypen als *built-in-Types* bezeichnet [Cat94]. Die benutzerdefinierten Objekttypen sind Gegenstand des Data-Dictionary und werden durch die Klasse *CObjectType* repräsentiert. Die Spezialisierungen dieser Klasse (*CLocalObjType* und *CVirtualObjType*) modellieren die Konzepte der Datenverteilung und der Sichtenbildung, welche bisher im ODMG-93-Standard nicht vorgesehen sind.

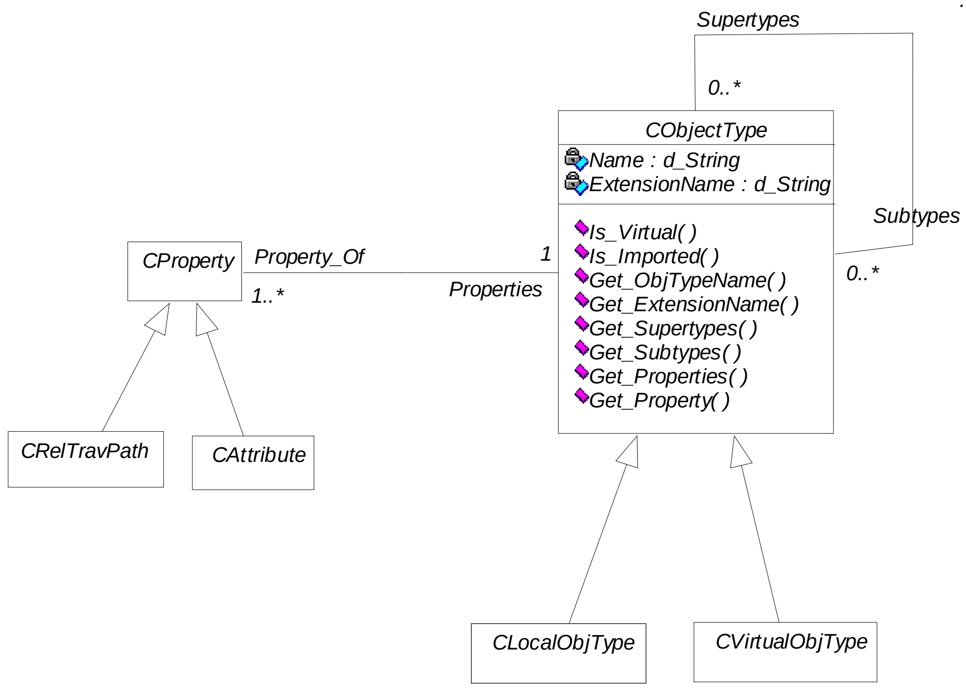


Abbildung 37 Die Repräsentation von Objekttypen-Eigenschaften

Ein benutzerdefinierter Objekttyp im ODMG-93-Objektmodell besitzt eine Menge von Eigenschaften (Instanz-Eigenschaften) und steht zu anderen Objekttypen in Vererbungsbeziehung (Typ-Eigenschaft siehe Kapitel 3.2). Die Instanz-Eigenschaften eines Objekttyps (*Properties*) werden als assozierte Instanzen der abstrakten Klasse *CProperty* verwaltet (siehe Abbildung 37). Die Supertyp/Subtyp-Assoziation modelliert die Vererbungsbeziehung zwischen Objekttypen. Ein Subtyp erbt dabei alle Eigenschaften seiner Supertypen (Mehrfachvererbung). Das Attribut *ExtensionName* der Klasse *CObjectType* bezeichnet den Namen der Extension eines Objekttyps (siehe Kapitel 3.2.2), das Attribut *Name* den Namen des Objekttyps. Die Klasse *CObjectType* definiert eine Reihe von Operationen, die einen navigierenden Zugriff auf die assoziierten Schema-Elemente ermöglichen:

- Die Operationen *Get_Property* bzw. *Get_Properties* liefern Referenzen auf eine bzw. alle Eigenschaften (*CProperty*) eines Objekttyps.

- *Get_Supertypes* bzw. *Get_Subtypes* unterstützen den navigierenden Zugriff entlang der Vererbungshierarchie von Objekttypen.

- *Is_Virtual* ist eine boolsche Operation und liefert den Wert TRUE, falls der konkrete Objekttyp Instanz ein vitueller Objekttyp ist, die Operation *Is_Imported* ist invers zu *Is_Virtual*.

- Die Operationen *Get_ObjTypeName* und *Get_ExtensionName* gestatten den lesenden Zugriff auf die Atribute *Name* und *ExtensionName* (Kapselungs-Prinzip).

Instanz-Eigenschaften eines Objekttyps sind entweder Attribute oder Beziehungen zu anderen Objekten und werden durch Instanzen der Klassen *CAttribute* und *CRelTravPath* abgebildet (siehe Abbildung 38). Die abstrakte Klasse *CProperty* ist die Generalisierung von *CAttribute* und *CRelTravPath* definiert deren gemeinsame Eigenschaften und Operationen:

- Das Attribut *Name* (Lese-Operation *Get_PropertyName*) der Klasse *CProperty* beschreibt den Namen der Eigenschaft und dient dem Zugriff auf den Wert dieser Eigenschaft. Das Attribut *Name* ist vergleichbar mit dem Namen einer Membervariable in C++.

- Das Attribut *Required* (Lese-Operation *Is_ Required*) bestimmt, ob eine Instanz des Objekttyps in dieser Eigenschaft einen definierten Wert haben muß und ist vergleichbar mit der NOT NULL-Klausel bei relationalen Datenbanken [Date90].

- Die Operation *Get_ObjectType* liefert eine Referenz auf den Objektypen (Instanz der Klasse *CObjectType*) der Eigenschaft.

- Mithilfe der Operationen *Is_Attribut* und *Is_RelTravPath* läßt sich die spezielle Art einer Instanz-Eigenschaft bestimmen.

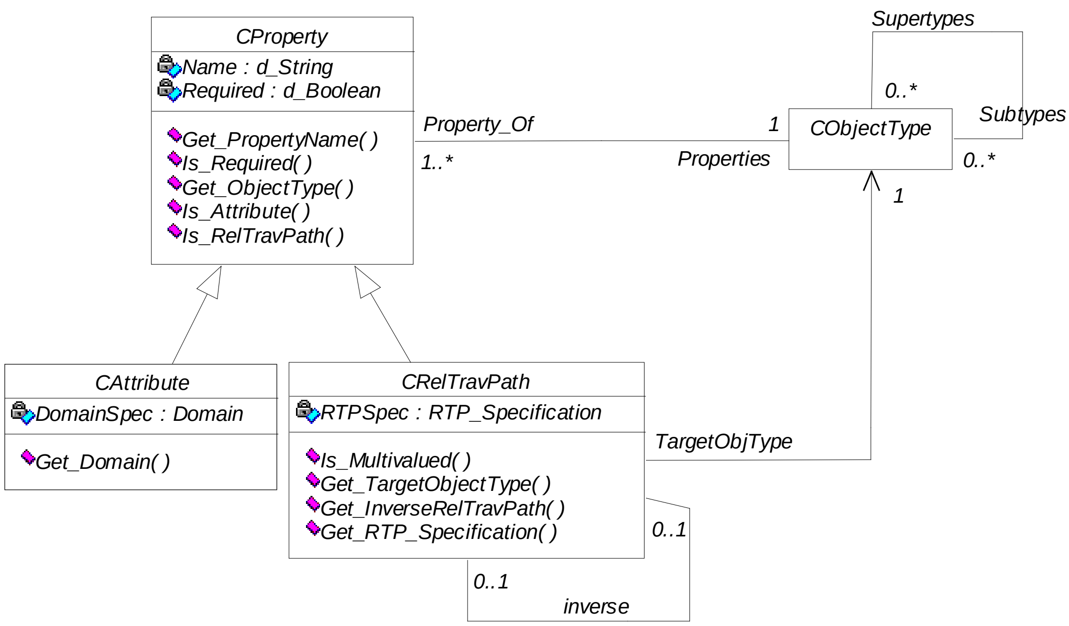


Abbildung 38 Instanz-Eigenschaften von Objekttypen

Im ODMG-93-Objektmodell ergeben sich Beziehungen aus den zueinander inversen Zugriffs-Pfaden [Cat94]. Diese erlauben einen navigierenden Zugriff auf die durch die Beziehung assoziierten Objekte (siehe Kapitel 3.2.4). Zugriffs-Pfade werden im Data-Dictionary als Instanzen der Klasse *CRelTravPath* verwaltet. Diese besitzen eine Referenz auf den Ziel-Objektyp (*TargetObjType*) und auf den inversen Zugriffs-Pfad (bidirektionale Beziehungen). Das Attribut *RTPSpec* (*Get_RTP_Specification* als dessen Lese-Operation) bestimmt die Kardinalität und die Repräsentation der referenzierten Objekte als Menge. *RTPSpec* hat als Wertebereich den Aufzählungstypen *RTP_Specification = {Ref, Set, List}* (siehe Kapitel 3.2.4). Die Operationen *Get_InverseRelTravPath* und *Get_TargetObjectType* unterstützen einen navigierenden Zugriff auf die oben beschriebenen assoziierten Schema-Elemente.

Der Wertebereich für Attribute sind literale (*immutable*) Objekte (immutable Objekte ohne wertunabhägigie Identität, siehe Kapitel 3.2.1). Diese können entweder atomar oder strukturiert sein (siehe Kapitel 3.2.3). Das in dieser Diplomarbeit zu realisierende FDBS beschränkt sich darauf, nur atomare Objekte (Datentypen) als Wertebereich für Attribute zu unterstützen, um die Komplexität des Systems im Rahmen zu halten. Das Attribut *DomainSpec* der Klasse CAttribute bestimmt den Wertebereich eines Objektyp-Attributs. Mögliche Wertebereiche sind *domain = { d_Short, d_Long, d_UShort, d_ULong, d_Float, d_Double, d_Char, d_Octet, d_Boolean, d_String, d_Date, d_Intervall, d_Time, d_Timestamp }* (siehe Kapitel 3.2.3).

6.3.3 Import von Komponenten-Schemata

Wie bereits im Abschnitt 6.2 beschrieben wurde, unterstützt das FDBS einen speziellen Ankopplungs-Modus, der es erlaubt, die Komponenten-Schemata der KDBS in das Data-Dictionary zu importieren (siehe auch Abbildung 33 in Abschnitt 6.2). Dieser Abschnitt soll nun im Detail den Import-Vorgang darstellen. Die am der Import Komponenten-Schemata beteiligten Systemkomponenten sind in Abbildung 39 dargestellt, Abbildung 40 zeigt den Import-Vorgang als Sequence-Diagramm.

Der Datenbank-Adapter bzw. dessen lokaler *Data-Dictionary-Manager* unterstützt einen navigierenden Zugriff auf die Schema-Elemente des Komponenten-Schemas (lokale Kopie), indem er diese als verteilte Objekte mit IDL-Interface für den FDBS-Kern zugänglich macht. Die in Abbildung 39 angebenen Klassen des Datenbank-Adapters *CObjectTypeDescr*, *CPropertyDescr*, *CAttributeDescr*, *CRelTravDescr* werden als Schema-Element-Deskriptoren bezeichnet, ihre Instanzen sind die verteilten Objekte mit IDL-Interface, welche die Schema-Elemente des Komponenten-Schemas beschreiben. Das lokale Data-Dictionary des DB-Adapters ist somit ein selbstbeschreibendes (*selfdescriptive*) System [OHE96]. Jeder Datenbank-Adapter, der mit dem FDBS kooperieren will, muß diese IDL-Interface-Beschreibungen implementieren. Die Semantik der Operationen der Schema-Element-Deskriptoren entspricht der Semantik der im Abschnitt 6.3.2 dargestellten Schema-Elemente.

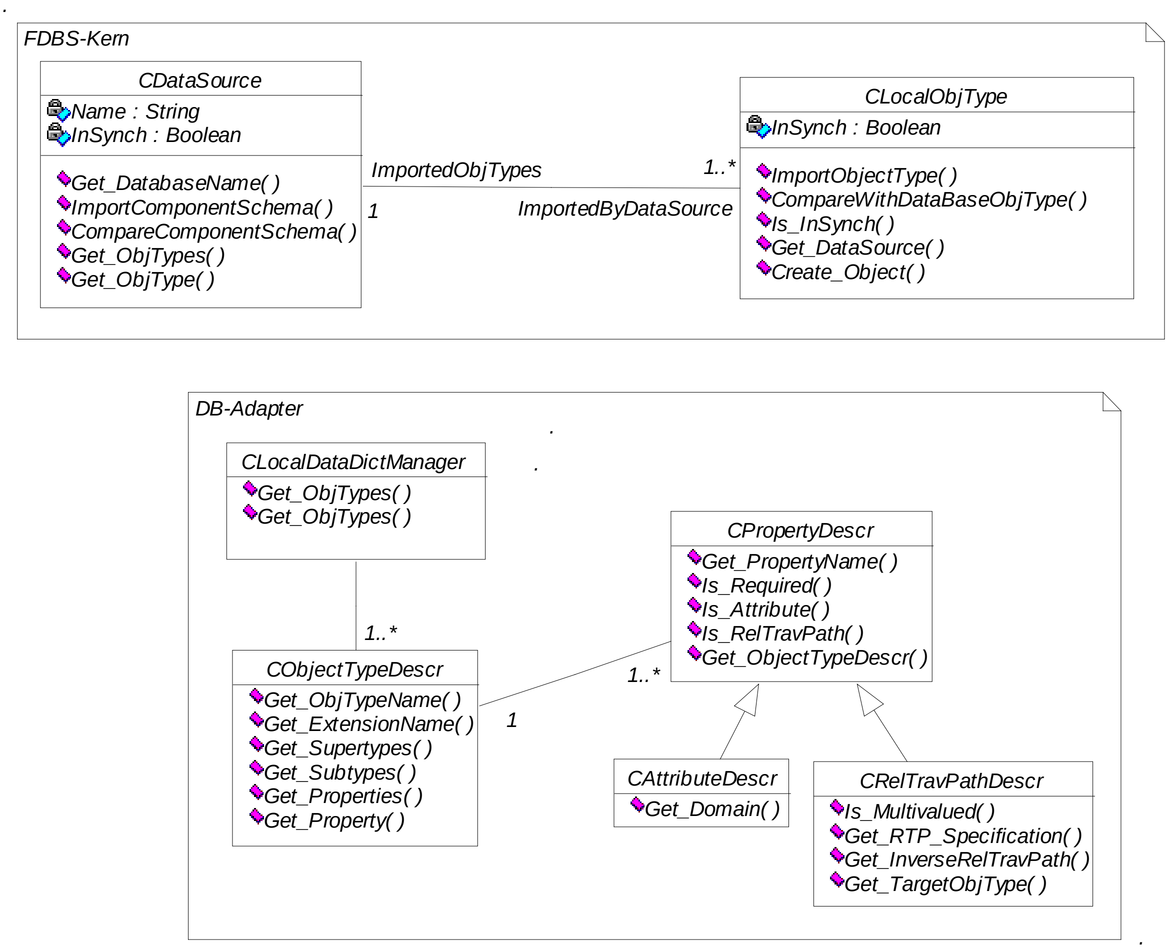


Abbildung 39 Das selbstbeschreibende System von Schema-Elementen

Der eigentliche Import-Vorgang läuft nun folgendermaßen ab (siehe Abbildung 40):

- Nach dem Ankoppeln des DB-Adapters (mittels *PluginDBAdapter*, siehe Abschnitt 6.2) delegiert das *CFDBSKernel-Objekt* den Import an den *Data-Dictionary-Manager* (*Add_DataSource*), dieser erzeugt daraufhin ein neues Objekt der Klasse *CDataSource* (in Abbildung 40 nicht dargestellt) und besorgt sich vom DB-Adapter die Objekt-Referenz (*CORBA-OID*) des lokalen *Data-Dictionary-Manager* (*Get_DataDictManager*).

- Der eigentliche Import-Vorgang wird nun vom *Data-Dictionary-Manager* an das *CDataSource-Objekt* delegiert (*ImportComponentSchema*). Das *CDataSource-Objekt* erfragt beim lokalen *Data-Dictionary-Manager* die Objekt-Referenzen aller Objekttypen der lokalen Kopie des Komponenten-Schemas (*Get_ObjTypes*) und legt zu jedem Objekttyp eine Kopie (*CLocalObjType*) im Data-Dictionary des FDBS an.

- Jeder dieser lokalen Objekttypen (*CLocalObjType*) wird nun aufgefordert, alle weiteren Schema-Elemente / Eigenschaften selbständig zu importieren (*ImportObjectType*).

- Nun kommen die oben dargestellten Schema-Element-Deskriptoren des lokalen *Data-Dictionary-Managers* zum Einsatz. Ein lokaler Objekttyp (*CLocalObjType*) erfragt sämtliche seiner Schema-Elemente beim korrespondierenden Schema-Element-Deskritor (*Get_Supertypes*, *Get_Properties*) und navigiert dabei zu weiteren Schema-Elementen (z.B. *Get_PropertyName*, ... usw.). Um Schema-Elemente im Data-Dictionary des FDBS zu verbinden (z.B. eine Supertyp/Subtyp-Beziehung), hat der lokale Objekttyp (*CLocalObjType*) durch das *CDataSource-Objekt* Zugriff auf die bereits zuvor erzeugten Schema-Elemente (z.B. *Get_ObjType* in Abbildung 40).

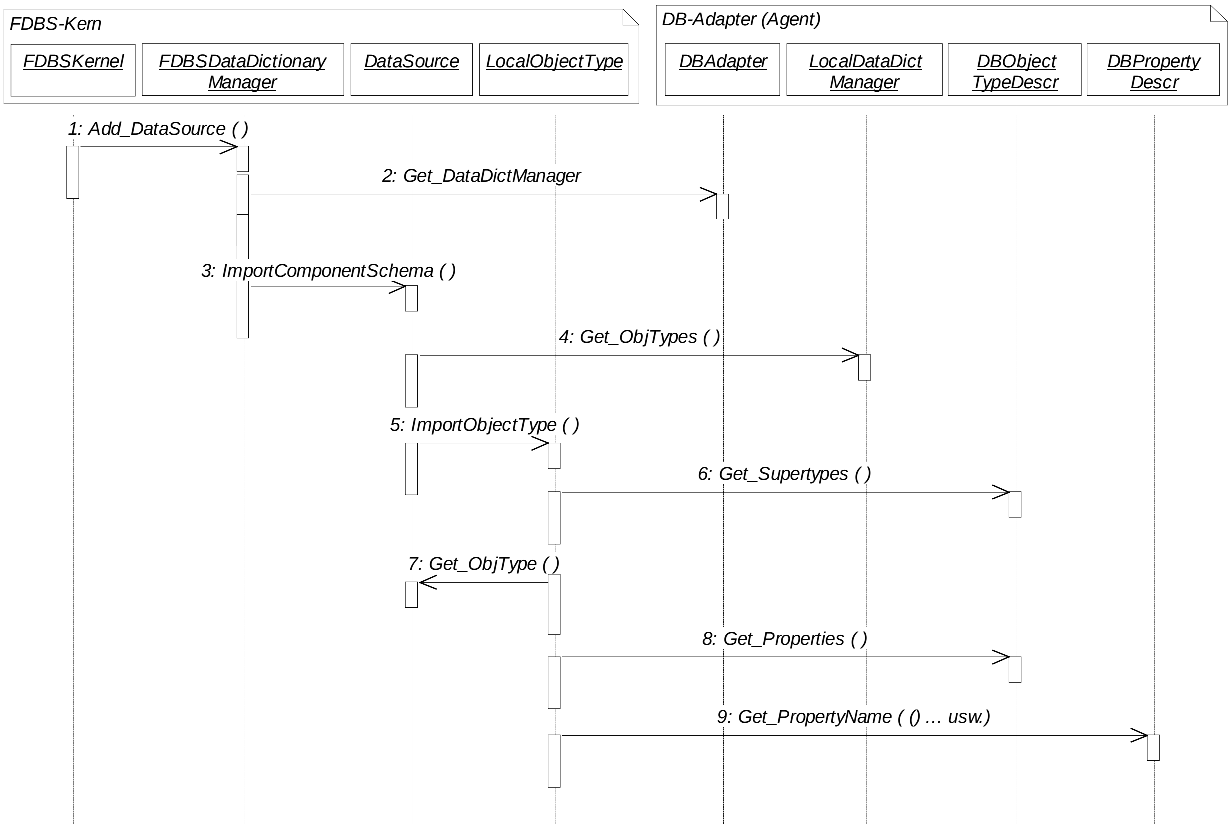


Abbildung 40 Interaktion während des Imports eines Komponenten-Schemas

Der Abgleich der Kopien des Komponenten-Schemas findet im laufenden Betrieb des FDBS beim Ankoppeln eines Datenbank-Adapters statt (siehe auch Abbildung 34 in Abschnitt 6.2) und wird analog zum Ablauf des Imports realisiert. Die Klassen *CDataSource* und *CLocalObjType* unterstützen hierzu die Operationen *CompareComponentSchema* bzw. *CompareWithDataBaseObjType* (siehe Abbildung 39).

6.3.4 Die Ableitung von virtuellen Objekttypen

Virtuelle Objekttypen dienen der Sichtenbildung und sind ein wichtiges Konzept bei der Schemaintegration von objektorientierten Datenbank-Schemata [BFN94, SpaPa95]. Dieser Abschnitt beschreibt, wie virtuelle Objekttypen abgeleitet und im Data-Dictionary verwaltet werden könnten. Virtuelle Objekttypen werden jedoch nicht im FDBS-Prototyp, der im Rahmen dieser Diplomarbeit entwickelt wurde, unterstützt, da hierzu die Implementierung eines verteilten Query-Proccesors nötig wäre und der Aufwand hierfür mehrere Mannjahre betragen würde (siehe [BFHK95]). An dieser Stelle soll jedoch das Konzept der virtuellen Objekttypen vorgestellt werden, um ein Gefühl dafür zu vermitteln, wie aus homogenisierten Schemata (Komponenten-Schemata) eine integrierte Sicht (föderierte Schemata) auf den Datenbestand des föderierten Datenbanksystems geschaffen werden kann. Die Darstellung orientiert sich dabei an den in [IRO-DB2-94, Souza95] dargestellten Konzepten.

Virtuelle Objekttypen werden von anderen Objektypen (*CObjectType*) abgeleitet (*derived_from*), welche auch als Basis-Objekttypen bezeichnet werden. Die Definition eines virtuellen Objekttypen besteht aus einem Deklarationsteil (*CObjectType*) und einem Ableitungsteil (*CVirtualObjType*), welcher die Spezialisierung gegenüber den anderen Objekttypen darstellt (siehe Abbildung 41). Die Extension eines virtuellen Objekttypen ist abhängig von den Extensionen der Basis-Objekttypen. Eine OQL-Datenbankanfrage (siehe Kapitel 3.5) definiert dabei die Ableitungsvorschrift (*extent_deriv_expr*). Die Eigenschaften der Objekte in der Extension des virtuellen Objekttyps werden ebenfalls durch OQL-Datenbankanfrage-Ausdrücke materialisiert (*CPropertyMapping->deriv_expr*). Die Materialisierung von virtuellen Objekten wird vom Object-Manager an den Query-Processor delegiert (vergleiche hierzu Abschnitt 6.1).

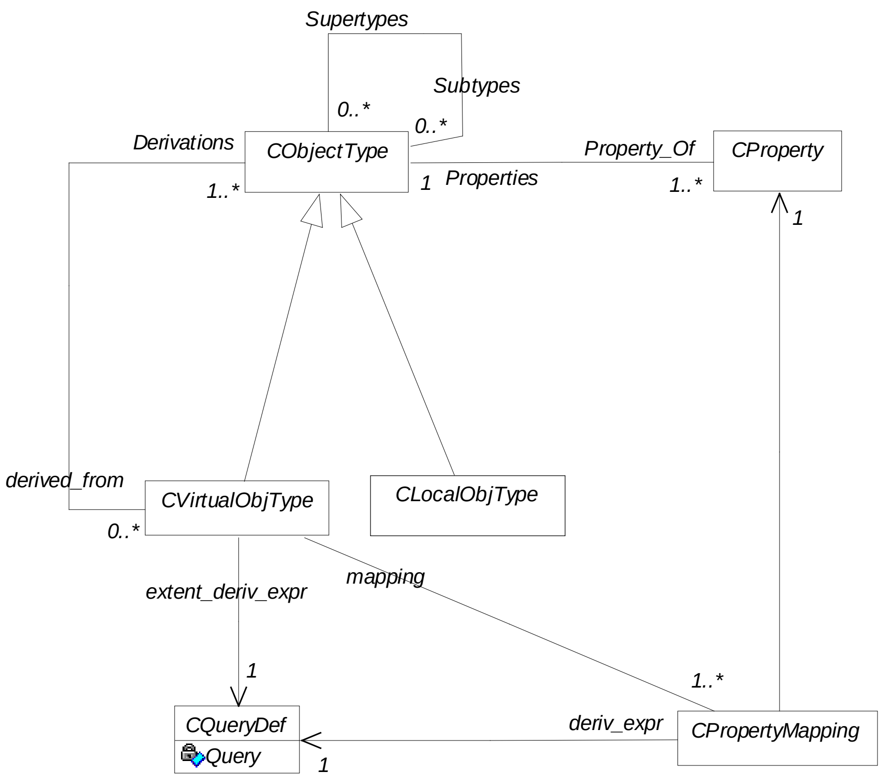


Abbildung 41 Ableitung von virtuellen Objekttypen

Ein kleines Beispiel aus dem Bankenbereich soll die Ableitung von virtuellen Objekttypen deutlich machen. Nach der Fusionierung zweier Banken, *A* und *B*, soll eine integrierte Sicht auf alle Kontoinhaber geschaffen werden. Dabei muß berücksichtigt werden, daß eine Person Konten bei beiden Banken haben kann (extensionale Überlappung [ScSa96]). Um das Beispiel zu vereinfachen, stimmen die Objektypen *KontenInhaberA*, *KontenInhaberB*, aus denen der virtuelle Objekttyp *Int_Kontoinhaber* abgeleitet wird, strukturell und semantisch vollständig überein (intensionale Überlappung). Abbildung 42 zeigt die Definition des virtuellen Objekttyps *Int_Kontoinhaber* in einer erweiterten ODL-Notation.

```
interface Int_Kontoinhaber
( extent Int_AllKontenInhaber                                      Deklarationsteil
  keys Name )
{
  attribute        string     Name;
  attribute        date       GebDatum;
  relationship     Konto Konten inverse Konto.Inhaber;

  Mapping                                                  Basis-Objekttypen
  (
   derived_from : KontoinhaberA, KontoinhaberB
   extent_deirv_expr : Select KIObj from                           Extensions-Ableitung
                        (
                        (Select KIObjA from AllKontenInhaberA
  Obj.-Mengen-          where not (exists KB in AllKontenInhaberB: KB.Name = KIObjA.Name))
  differenz A / B
                        Union
                        (Select KIObjB from AllKontenInhaberB
  Obj.-Mengen-          where not (exists KA in AllKontenInhaberA: KA.Name = KIObjB.Name))
  differenz B / A
                        Union
                        ((Select KIObjA from AllKontenInhaberA
  Obj.-Schnitt-         where (exists KB in AllKontenInhaberB: KB.Name = KIObjA.Name))
  Menge A ∩ B
                        )
  Name          deriv_expr : KIObj.Name;                            Ableitung der Eigenschaften
  GebDatum      deriv_expr : KIObj.GebDatum;
  Konten        deriv_expr :   (Select KObjA from AllKontenInhaberA.Konten
                               where (KObjA.Inhaber.Name = KIObj.Name))
                               Union
                               (Select KObjB from AllKontenInhaberB.Konten
                               where (KObjB.Inhaber.Name = KIObj.Name))

   )
}
```

Abbildung 42 Beispiel-Definition eines virtuellen Objekttyps

6.4 Der FDBS-Event-Handler

Die Verarbeitung von Ereignissen (*Events*), welche in den Komponenten-Datenbanksystemen auftreten, ist Aufgabe des *Event-Handlers* des FDBS-Kerns, welcher im folgenden als *globaler Event-Handler* bezeichnet wird. Dieser kooperiert mit den Datenbank-Adaptern, bzw deren *lokalen Event-Handlern*, die ihn über auftretende Ereignisse informieren.

Der *globale Event-Handler* wird als einzige Instanz (*Singleton*) der Klasse *CFDBSEventHandler* realisiert, sein IDL-Interface beinhaltet lediglich die Operation *Process_DBObjectEvent* (siehe Abbildung 43). Die *lokalen Event-Handler* (*CDBEventHandler*) haben die Aufgabe, lokale Ereignisse zu entdecken und dem *globalen Event-Handler* durch den Aufruf der Operation *Process_DBObjectEvent* mitzuteilen. Mögliche lokale Ereignisse sind die Erzeugung, das Löschen oder die Modifikation von DB-Objekten durch lokale Benutzer. Auftretende Ereignisse muß ein *lokaler Event-Handler* solange zwischenspeichern (wegen des möglichen Ausfalls eines Rechnerknotens oder einer Kommunikationsverbindung), bis er vom *globalen Event-Handler* über dessen vollständige Bearbeitung benachrichtigt wird. Lokale Ereignisse werden als verteilte Objekte (*CDBObjectEvent*) mit IDL-Interface repräsentiert (siehe Abbildung 43). Der *globale Event-Handler* erhält beim Aufruf der Operation *Process_DBObjectEvent* die Objekt-Referenz (*CORBA-OID*) des *CDBObjectEvent-Objektes* als Parameter. Hierdurch sind alle Ereignisse für den *globalen Event-Handler* eindeutig identifizierbar. Die Operationen von *CDBObjectEvent* beschreiben dabei ein Ereignis vollständig (Kapselungs-Prinzip).

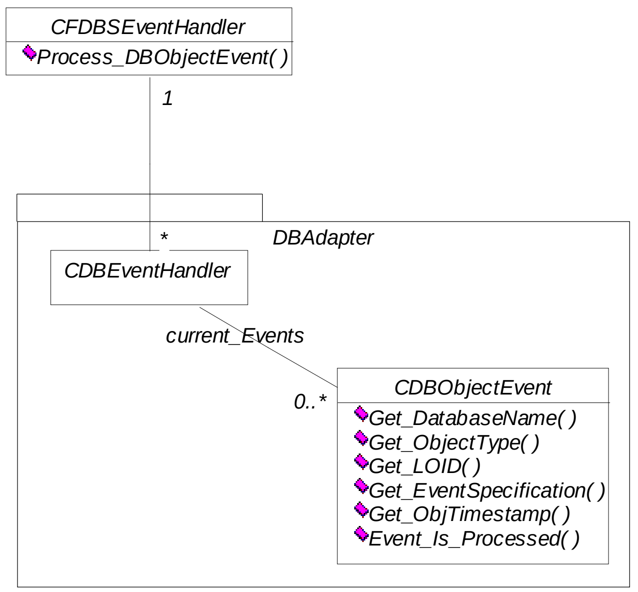


Abbildung 43 Klassen-Diagramm des Event-Handler

Der globale Event-Handler analysiert das hereinkommende Ereignis und delegiert dessen Bearbeitung an die weiteren Systemkomponenten des FDBS-Kerns. Abbildung 44 zeigt als Beispiel die Verarbeitung eines Modifikations-Ereignisses.

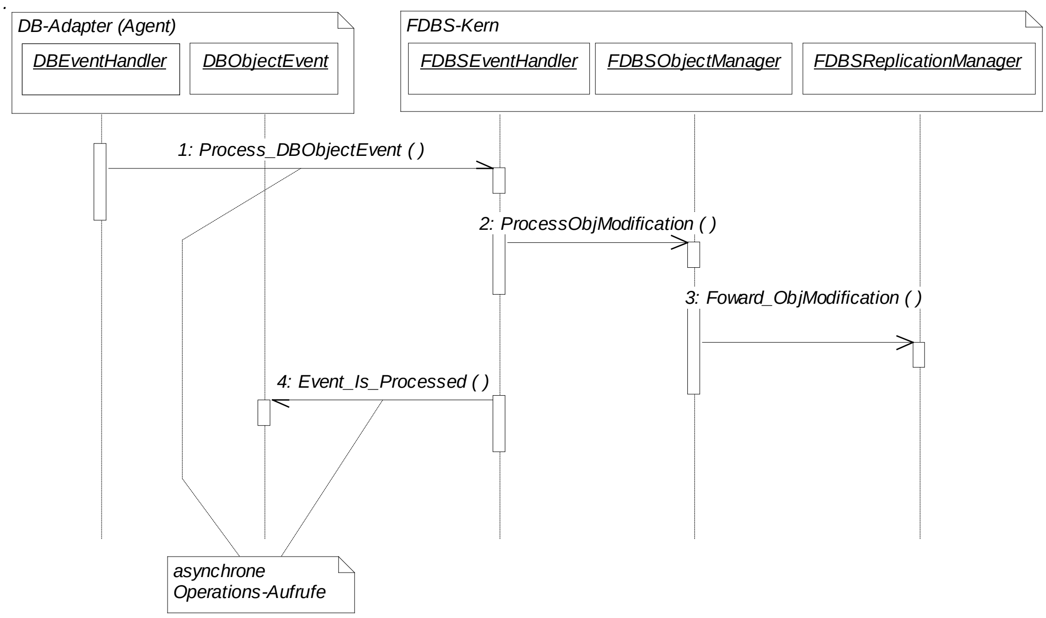


Abbildung 44 Event-Verarbeitung

Wurde ein Ereignis vollständig bearbeitet, so ruft der *globale Event-Handler* die Operation *Event_Is_Processed* des korrespondierenden *CDBObjectEvent-Objektes* auf. Dieses erledigt dann das „Vergessen" des lokalen Ereignisses.

Um die parallele Verarbeitung von Ereignissen zu optimieren, werden die Operationen *Process_DBObjectEvent* und *Event_Is_Processed* als asynchrone (*oneway*) Operationen (siehe Kapitel 2.4) im IDL-Interface spezifiziert. Darüber hinaus hat sich bei der Entwicklung des FDBS gezeigt, daß es, in Hinblick auf die Performance, sinnvoll ist, das Kapselungs-Prinzip bei den *CDBObjectEvent-Objekten* einzuschränken, um die Anzahl der Operations-Aufrufe (via ORB) zu reduzieren. Aus diesem Grund werden alle, ein Ereignis spezifizierende, Eigenschaften zusammen mit der Objekt-Referenz des korrespondierenden *CDBObjectEvent-Objektes* als Parameter an die Operation *Process_DBObjectEvent* übergeben. Abbildung 45 zeigt das optimierte IDL-Interface des *Event-Handlers*. Die Eigenschaften, die ein Ereignis charakterisieren, sind Gegenstand des nächsten Abschnitts, da sie vom Object-Manager analysiert werden. Einzige Ausnahme hierbei ist die Art des Ereignisses (in Abbildung 45 *EventSpec*), welche vom Event-Handler benötigt wird, um die Bearbeitung zu delegieren.

```
struct _Timestamp
{
 short            year;
 unsigned short   month;
 unsigned short   day;
 unsigned short   hour;
 unsigned short   minute;
 float            seconds;
 unsigned short   timezone_hour;
 unsigned short   timezone_minute;
};

interface CDBObjectEvent                                   Lokales Ereignis (DB-Adapter)
{
        oneway void Event_Is_Processed();
};

enum EventSpec {Object_Creation, Object_Modification, Object_Deletion};

struct EventInfo
{
        EventSpec EventSpecification;                      Art des Ereignisses
        string LOID;
        string DatabaseName;                               Objekt-Referenz des
        string ObjectTypeName;                             CDBObjectEvent-Objektes
        _Timestamp EventTimestamp;
        CDBObjectEvent EventObjRef;
};

interface CFDBSEventHandler                                globaler Event-Handler (FDBS-Kern)
{
        oneway void Process_DBObjectEvent(in EventInfo Event);
};
```

Abbildung 45 IDL-Interface des Event-Handlers

6.5 Der FDBS-Object-Manager

Der *Object-Manager* des FDBS-Kerns hat die Aufgabe, die Verteilung der lokalen DB-Objekte (Objekte der Extensionen der Komponenten-Schemata) transparent zu machen und eine dauerhafte und wertunabhängige Objekt-Identität [EK91] zu unterstützen. Er kooperiert mit den lokalen *Object-Managern*, die Bestandteil der Datenbank-Adapter sind, weshalb er im folgenden als *globaler Object-Manager* bezeichnet wird.

6.5.1 Die Identifizierung von DB-Objekten

Da die heterogenen Komponenten-Datenbanksysteme des FDBS unterschiedliche Konzepte zur Objekt-Identifizierung unterstützen, muß der *globale Object-Manager*, in Kooperation mit den lokalen *Object-Managern*, eine Abbildung zwischen lokaler Objekt-Identität und globaler Objekt-Identität verwalten.

Nach [EK91] lassen sich drei unterschiedliche Konzepte zur Identifizierung von Datenbank-Objekten unterscheiden. Diese lassen sich durch ihre räumliche und zeitliche Dimension charakterisieren:

- Werden Objekte durch die Werte, die sie in bestimmten Eigenschaften haben, identifiziert, so spricht man von ***wertbasierter Objekt-Identität***. Die Menge der identifizierenden Eigenschaften wird dabei als Schlüssel bezeichnet. Ein Beispiel für die *wertbasierte Objekt-Identität* sind relationale Datenbanksysteme. Die Identität ihrer DB-Objekte (Tupel) ist räumlich durch die Relation und zeitlich durch die Werte in ihren Primärschlüsseln begrenzt.

- Das Konzept der ***Session Object Identifier*** sieht eine Identität von DB-Objekten vor, die räumlich durch die Datenbank und zeitlich durch die Dauer einer Verbindung (*Session*) zum Datenbanksystem begrenzt ist. Dieses Konzept wird von objektorientierten Datenbanksystem wie z.B. O_2 [BDK92] eingesetzt. Sogenannte *Object-Handle* dienen dabei den Anwendungen zur temporären Identifizierung von DB-Objekten.

- Lassen sich DB-Objekte unabhängig von ihrem Zustand (Wertbelegung der Eigenschaften) dauerhaft und eindeutig innerhalb der gesamten Datenbank identifizieren, so spricht man von unveränderlicher (immutable) Objekt-Identität. In [EK91] wird dieses Konzept auch als ***strenge Objekt-Identität*** bezeichnet.

Um eine *strenge Objekt-Identität* zu unterstützen, wird in dieser Diplomarbeit der Ansatz gewählt, drei Ebenen von Objekt-Identität zu berücksichtigen (siehe Abbildung 46). Die Identifizierung der DB-Objekte der lokalen Schemata ist durch das DBMS der Komponenten-Datenbanksysteme bestimmt. Diese Ebene wird als *KDBMS-abhängige Object-Identität* bezeichnet. Die *lokalen Object-Manager* haben als Bestandteil eines Datenbank-Adapters die Aufgabe, das *KDBMS-abhängige Object-Identitäts-Konzept* auf eine *lokale strenge Objekt-Identität* abzubilden. Die DB-Objekte des Komponenten-Schemas lassen sich nun dauerhaft und wertunabhängig innerhalb des KDBS identifizieren.

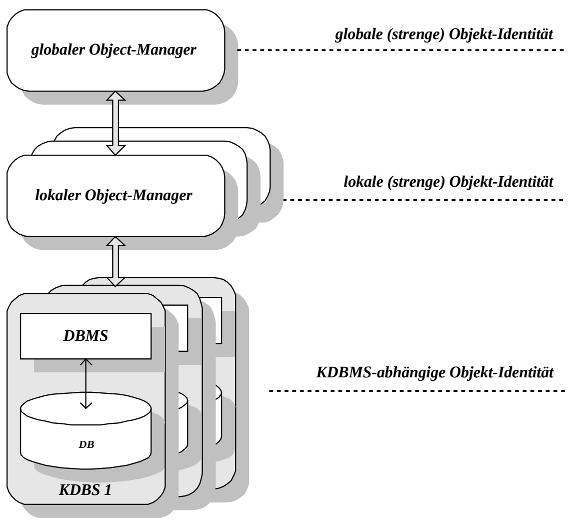


Abbildung 46 Drei-Ebenen-Objekt-Identität

Der *globale Object-Manager* erweitert durch eine zusätzliche Abbildung die räumliche Dimension der *lokalen Objekt-Identität.*

Zur Realisierung der *globalen Objekt-Identität* verwaltet der *globale Object-Manager* (*CFDBSObjectManager*) zu allen lokalen DB-Objekten (Objekte im Sinne eines Komponenten-Schemas) persistente Stellvertreter-Objekte (*Proxy-Objekte*). Diese werden in der Hilfsdatenbank (POET) als Instanz des Objekttyps *CLocalObject* gespeichert (siehe Abbildung 47 im Abschnitt 6.5.2). Die *Proxy-Objekte* bilden die *lokalen Object-Identifier* (*LOID*) auf globale *OIDs* ab.

6.5.2 Der Zugriff auf DB-Objekte

Der Zugriff auf die lokalen DB-Objekte der KDBS wird durch deren *Proxy-Objekte* (*CLocalObject*) gekapselt (Abbildung 47 zeigt alle hiebei involvierten Klassen). Wird während einer globalen Transaktion auf ein DB-Objekt zum ersten mal zugegriffen, so stellt das *Proxy-Objekt* eine Beziehung zum korrespondierenden lokalen Objekt (*CDBObject*) her. Diese Beziehung (*DBObjectRef*) besteht nur solange, wie das DB-Objekt im Zugriff ist (maximal bis zum Ende der Transaktion). Alle Zugriffe auf das lokale DB-Objekt (*CDBObject*) werden durch das *Proxy-Objekt* verborgen. Das *Proxy-Objekt* nimmt dabei die Rolle eines Mediators ein und ermöglicht somit einen verteilungstransparenten Zugriff.

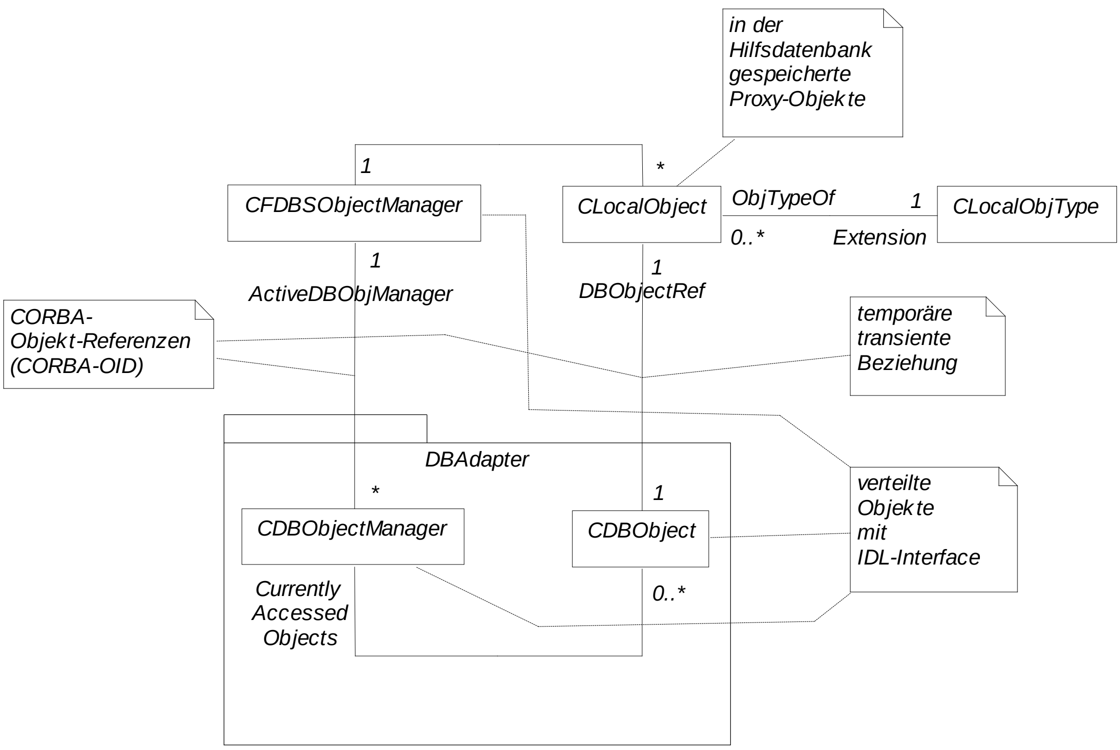


Abbildung 47 Klassen-Diagramm des Object-Manager

Die Interaktion der in Abbildung 47 dargestellten Klassen bzw. deren Instanzen läßt sich am besten anhand eines Sequence-Diagramms (Abbildung 48) darstellen. Im Beispiel erfolgt der Zugriff auf ein DB-Objekt durch ein anderes Objekt, welches in Abbildung 48 als *Object-Accessor* bezeichnet wird. Als *Object-Accessor* könnte beispielweise der *Replication-Manager* oder eine globe Anwendung (bzw. deren FDBS-Laufzeitsystem) fungieren. Zu Beginn des Beispiels hat der *Object-Accessor* bereits eine Referenz auf ein *Proxy-Objekt*.

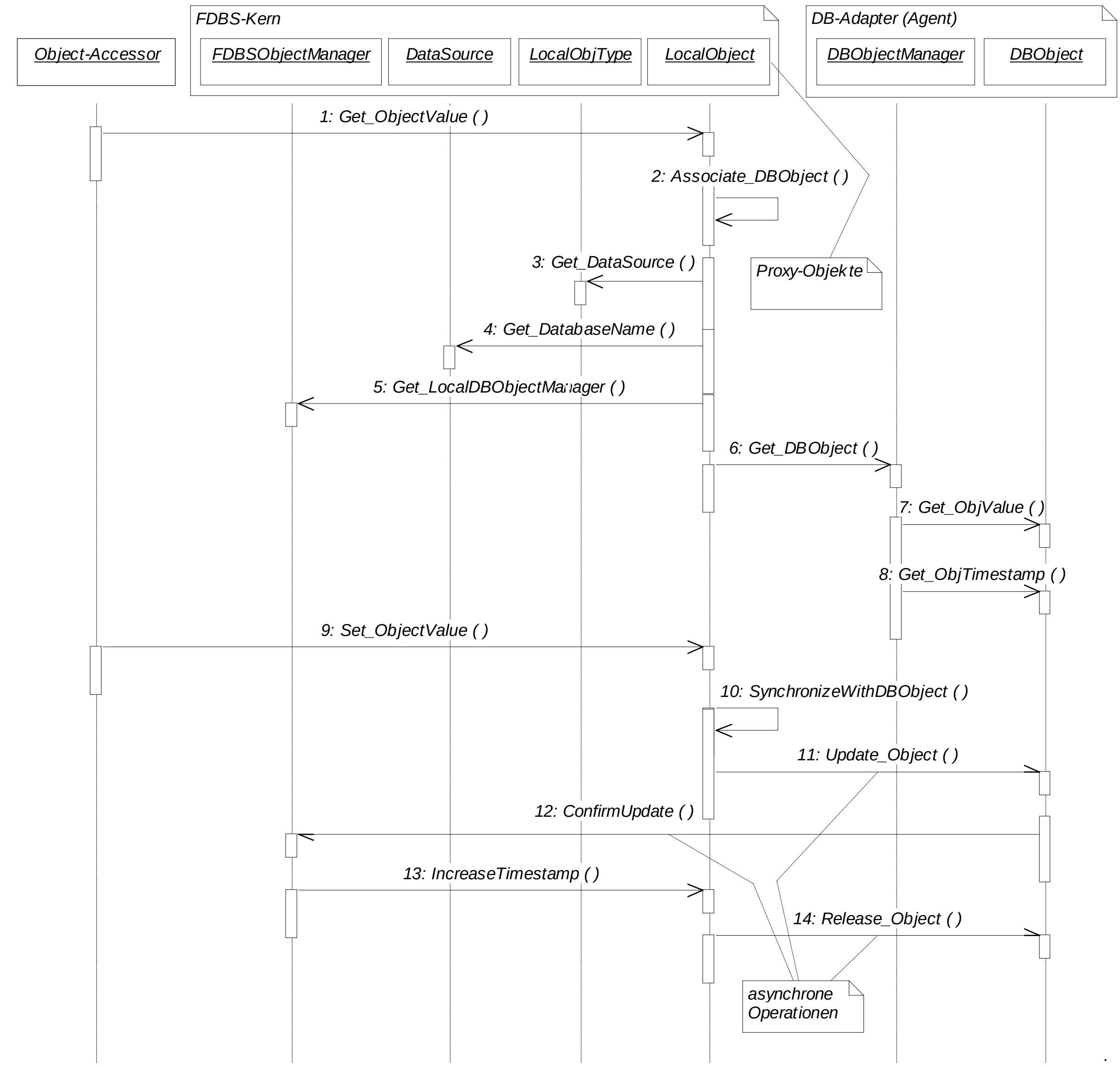


Abbildung 48 Der Ablauf beim Zugriff auf ein DB-Objekt

- Der erste Zugriff (*Get_ObjectValue*) auf das DB-Objekt (*Proxy-Objekt*) des *Object-Accessors* führt implizit dazu, daß das *Proxy-Objekt* eine temporäre Beziehung zum lokalen DB-Objekt aufbaut. Dies geschieht mittels der privaten Methode *Assoziate_DBObject* (Kapselungs-Prinzip) und löst eine Reihe weiterer Aktionen aus.

- Das *Proxy-Objekt* erkundigt sich bei seinem Objekttypen (*ObjTypeOf->Get_DataSource*, vergleiche Abbildung 47) nach seiner Datenquelle (*CDataSource*), bei dieser nach dem Namen des KDBS (*Get_DatabaseName*).

- Der *globale Object-Manager* verwaltet die Menge aller *lokalen Object-Manager*, die aktuell erreichbar sind (*ActiveDBObjManager*, siehe Abbildung 47). Hierzu bildet er die Namen der KDBS auf die Objekt-Referenzen (*CORBA-OID*) ihrer *lokalen Object-Manager* ab. Die Anfrage *Get_LocalDB-ObjectManager* liefert nun dem *Proxy-Objekt* die Object-Referenz des korrespondierenden *lokalen Object-Manager*.

- Das *Proxy-Objekt* wendet sich nun an diesen *lokalen Object-Manager* um eine Objekt-Referenz und den Objekt-Wert (Zustand des DB-Objektes, das im KDBS gespeichert ist) des lokalen DB-Objektes zu erhalten (*Get_DBObject*). Als Parameter des Operations-Aufrufes übergibt das *Proxy-Objekt* den lokalen Object-Identifier (*LOID*) des DB-Objektes.

- Der *lokale Object-Manager* verwaltet die Menge der DB-Objekte, die aktuell im Zugriff sind (*CurrentlyAccessedObjects*, vergleiche Abbildung 47). Da das betreffende DB-Objekt sich momentan nicht im Zugriff befindet, erzeugt der *lokale Object-Manager* eine Instanz von *CDBObject* (in Abbildung 48 nicht dargestellt) und liest (*Get_ObjectValue*) dessen Objekt-Wert, den er zusammen mit der Objekt-Referenz des *CDBObject-Objektes* an das *Proxy-Objekt* zurücklierfert. Zusätzlich wird der aktuelle Zeitstempel des DB-Objektes bestimmt (*Get_ObjTimestamp*). Dieser dient der Synchronisation zwischen *Proxy-Objekt* und lokalem DB-Objekt (Gegenstand von Abschnitt 6.5.3). Die Instanzen der Klasse *CDBObject* fungieren als temporäre Puffer-Objekte, welche die Transformation von DB-Objekten (bezüglich der Komponenten-Schemata) auf die lokalen Schemata kapseln.

- Der erste Zugriff auf das DB-Objekt ist nun beendet. Innerhalb der aktuellen Transaktion werden alle weiteren Zugriffe (z.B. *Set_ObjectValue*) auf das DB-Objekt direkt vom *Proxy-Objekt* behandelt, da dieses den Zustand des DB-Objektes transient speichert. Wird die laufende Transaktion beendet oder das DB-Objekt nicht mehr benötigt, so synchronisiert (*SynchronizeWithDBObject*) sich das *Proxy-Objekt* mit seinem lokalen Pendant (*CDBObject*). Hierzu übergibt es den aktuellen Zustand des DB-Objektes mittels *Update_Object* an das *CDBObject-Objekt*, welches dann die Speicherung (Transformation) im KDBS erledigt. Das lokale Puffer-Objekt (*CDBObject*) bestätigt dem *globalen Object-Manager* die erfolgreiche Speicherung durch den Aufruf von *ConfirmUpdate*. Der *globale Objekt-Manager* paßt daraufhin den Zeitstempel des *Proxy-Objektes* dem des lokalen DB-Objektes an (*IncreaseTimestamp*). Schließlich wird die temporäre Beziehung (*DBObjRef*) zwischen dem *Proxy-Objekt* und seinem lokalen Pendant durch *Release_Object* beendet. Das lokale Puffer-Objekt (*CDBObject*) beseitigt daraufhin eventuelle Datenbank-Sperren (*Locks*) und verabschiedet sich, indem es sich selbst löscht. Um die Speicherung (Synchronisierungs-Phase) von DB-Objekten zu parallelisieren, werden die Operationen *Update_Object*, *ConfirmUpdate* und *Release_Object* als asynchrone (*oneway*) Operationen (siehe Kapitel 2.4) im IDL-Interface spezifiziert. Somit kann die Speicherung von DB-Objekten, die durch die lokalen Puffer-Objekte (*CDBObject*) realisiert wird, in meheren KDBS parallel durchgeführt werden. Hiervon profitiert insbesondere der *Replication-Manager*.

6.5.3 Die Synchronisation von DB-Objekten

Die Synchronisation zwischen den *Proxy-Objekten* und den lokalen DB-Objekten basiert auf Zeitstempeln (*Timestamps*) und zusätzlichen Statusinformationen, die als Attribute von *Proxy-Objekten* gespeichert sind (siehe Abbildung 49).

Es ist die Aufgabe der Datenbank-Adapter, die lokalen DB-Objekte mit *Timestamps* zu versehen. Der *Timestamp* eines lokalen DB-Objektes erhöht sich nach jeder Änderungs-Operation innerhalb globaler und lokaler Transaktionen. Wird ein lokales DB-Objekt innerhalb einer lokalen Transaktionen geändert, so wird der Objekt-Manager durch den Aufruf von *ProcessObjModification* (siehe Abbildung 49) vom Event-Handler darüber informiert (vergleiche hierzu Abschnitt 6.4). Der *Timestamp* des *Proxy-Objektes* (*ActualObjTimestamp*) wird entsprechend erhöht. Da die lokalen Transaktionen jedoch von den globalen Transaktionen entkoppelt sind (siehe Kapitel 5.3.3), kann es zu Konflikten kommen. Durch den Vergleich der *Timestamps* von *Proxy-Objekt* und lokalem DB-Objekt läßt sich ein solcher Konflikt erkennen.

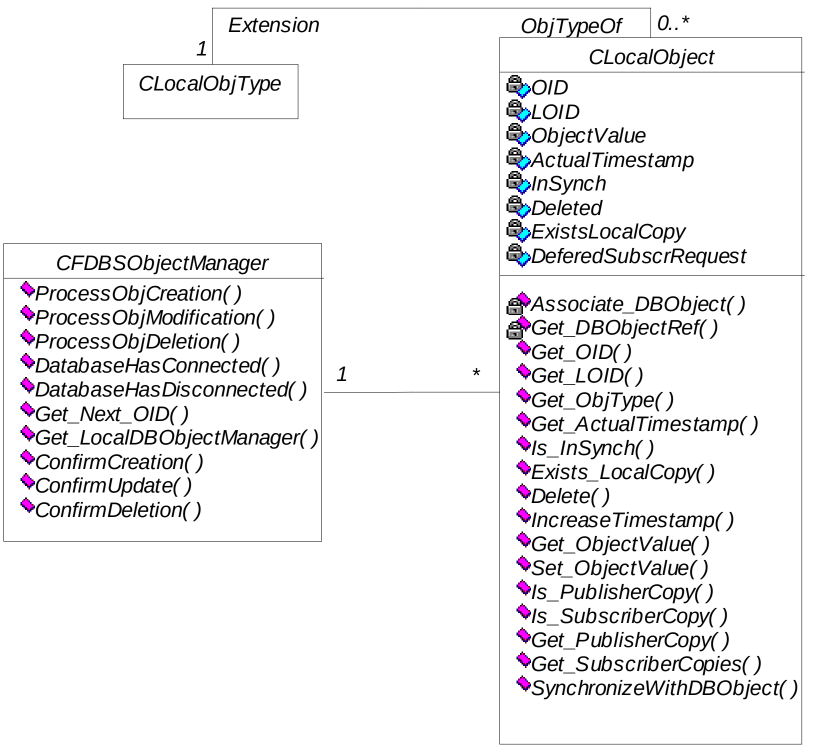


Abbildung 49 Attribute & Operationen der Klassen des Object-Manager

Die Bedeutung der zusätzlichen Statusinformationen eines Proxy-Objektes wird im folgenden dargestellt:

- Das Attribut ***InSynch*** (Lese-Operation ***Is_InSynch***) repräsentiert den Status bei der Durchführung der Synchronisierungs-Phase (siehe Abschnitt 6.5.3). Es wird am Beginn der *Synchronize-WithDBObject*-Operation auf den Wert FALSE gesetzt. Sobald der *lokale Object-Manager* des Datenbank-Adapters das erfolreiche Ende der Synchronisierungs-Phase durch *ConfirmCreation*, *ConfirmUpdate* oder *ConfirmDeletion* bestätigt, wird es vom *globalen Object-Manager* auf TRUE gesetzt. Zwischenzeitliche Ausfälle eines KDBS oder einer Netzwerk-Verbindung werden somit berücksichtigt. Der *globale Object-Manager* kann nach der Wiederaufnahme der Verbindung zwischen Datenbank-Adapter und FDBS-Kern (*DatabaseHasConnected*) die Synchronisation von Objekten, die nicht „*In_Synch*“ sind, fortsetzen. Dies ist insbesondere für den gewählten Replikation-Ansatz von Bedeutung (vergleiche Kapitel 5.3.3).

- Wird in einer globalen Transaktion ein DB-Objekt erzeugt (etwa vom *Replication-Manager*), so wird zunächst ein *Proxy-Objekt* erzeugt, welches dann die Erzeugung des lokalen DB-Objektes anstößt (*SynchronizeWithDBObject*). Bei der Initialisierung des *Proxy-Objektes* wird die Statusinformation ***ExistsLocalCopy*** auf FALSE gesetzt. Sobald der *lokale Object-Manager* die Erzeugung des lokalen DB-Objektes durch *ConfirmCreation* bestätigt, wird *ExistsLocalCopy* auf TRUE gesetzt. Der Status eines Proxy-Objektes läßt sich durch die Operation *Exists_LocalCopy* bestimmen.

- Das Löschen von lokalen DB-Objekten in globalen Transaktionen erfolgt durch die *Proxy-Objekt*-Operation *Delete* (siehe Abbildung 49). Diese Operation setzt zunächst das Attribut ***Deleted*** auf TRUE und leitet danach die Synchronisierungs-Phase durch *SynchronizeWithDBObject* ein. Erst nachdem das lokale DB-Objekt wirklich gelöscht wurde, löscht der globale Object-Manager auch das korrespondierende *Proxy-Objekt*.

- Das Attribut ***DeferedSubscrRequest*** steht im Zusammenhang mit dem Replikations-Mechanismus des FDBS, wird jedoch schon an dieser Stelle erläutert. Das *Proxy-Objekt* (eines *Subscriber-Objektes*) hält im Attribut *DeferedSubscrRequest* den Wert TRUE, falls eine lokale Änderungs-Operation des *Subscriber-Objektes* nicht zu dessen *Publisher-Objekt* propagiert werden konnte (vergleiche hierzu Kapitel 5.3.3). Nach Wiederaufnahme der Kooperations-Verbindung (*DatabaseHasConnected*) zwischen dem FDBS und dem KDBS, in dem das *Publisher-Objekt* gespeichert ist, wird der *Replication-Manager* vom Object-Manager beauftragt (*Repeat_SubscrRequest*), die Replikate wieder in einen konsitenten Zustand zu überführen. Die Statusinformation *ExistsLocalCopy* signalisiert dem *Replication-Manager*, ob die verzögerte Propagation eine Änderungs- oder Lösch-Operation als Ursache hatte.

Bisher offen geblieben ist, wie überhaupt die *Proxy-Objekte* entstehen. Die Beantwortung dieser Frage soll den Abschluß dieses Abschnitts bilden.

Wird in einem Komponenten-Datenbanksystem ein neues DB-Objekt erzeugt, so muß der *globale Object-Manager* ein korrespondierendes *Proxy-Objekt* erzeugen, nachdem er vom Event-Handler hierüber benachrichtigt wurde. Abbildung 50 stellt diesen Vorgang dar. Der *globale Object-Manager* erhält vom *Data-Dictionary-Manager* eine Referenz auf den Objekttyp (*CLocalObjType*) des DB-Objektes (*Get_LocalObjType*) und delegiert die Erzeugung des *Proxy-Objektes* an diesen Objekttypen (*Create_Object*). Das neue *Proxy-Objekt* erhält seine globale *OID* während der Initialisierung (*Konstruktor-Aufruf*) durch den Aufruf der Operation (*Get_Next_OID*). Abschließend wird der Replication-Manager angewiesen (*Forward_ObjCreation*), die eventuelle Erzeugung von Replikaten des DB-Objektes durchzuführen.

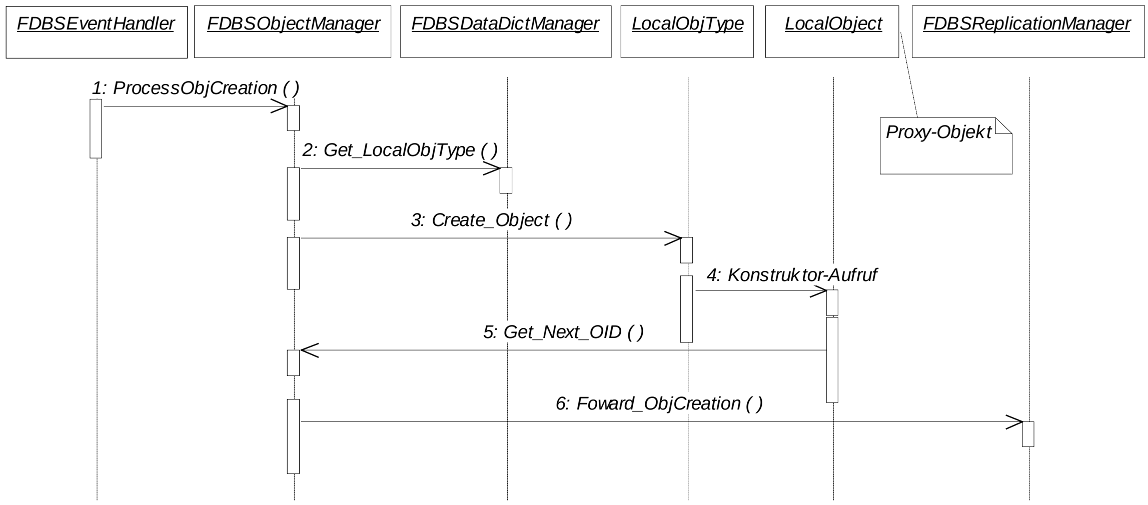


Abbildung 50 Ablauf nach der Erzeugung eines lokalen DB-Objektes

6.5.4 Die Repräsentation des DB-Objekt-Zustandes

Wie bereits im Abschnitt 6.5.2 deutlich wurde, erfolgt der Zugriff auf DB-Objekte innerhalb einer globalen Transaktion über deren *Proxy-Objekte*. Der Zustand der DB-Objekte ist jedoch physisch in den Komponenten-Datenbanksystemen gespeichert. Die lokalen Objekt-Manager transferieren den Wert eines DB-Objektes (Zustand) zum korrespondierenden *Proxy-Objekt* unmittelbar nachdem der erste Zugriff erfolgte. Die *Proxy-Objekte* fungieren dann als eine Art „*Objekt-Cache*", indem sie den Zustand der DB-Objekte während der globalen Transaktion transient speichern und alle weiteren Zugriffe direkt bearbeiten. Am Ende einer globalen Transaktion transferieren die *Proxy-Objekte* den Objekt-Zustand zürück an die lokalen Object-Manager (Synchronisierungs-Phase), die dann die Speicherung (Transformation) des Objekt-Zustandes im KDBS realisieren (vergleiche hierzu Abschnitt 6.5.2). Um die Heterogenität der Ablaufumgebungen der einzelnen Datenbank-Adapter und des FDBS-Kerns zu überwinden, ist es notwendig, eine gemeinsame und plattformunabhängige Repräsentation für die Übertragung der DB-Objekte bzw. deren Zustände zu definieren. Die Struktur der DB-Objekte ist bereits durch die Objekttyp-

Definitionen der Komponenten-Schemata beschrieben (siehe Abschnitt 6.3.2). Im folgenden wird nun beschrieben, in welchem Format der Zustand eines DB-Objektes zwischen den Datenbank-Adaptern und dem FDBS-Kern übertragen wird. Hierbei wird der Begriff „DB-Objekt“ als Synomym für den Zustand eines DB-Objektes verwendet, um die Beschreibung zu vereinfachen.

Die Repräsentation der DB-Objekte basiert auf den im Kapitel 2.4 dargestellten IDL-Datentypen und ist Bestandteil der IDL-Interface-Beschreibung der Datenbank-Adapter. Sämtliche „*built-in*“ Datentypen des ODMG-93-Objektmodells werden auf korrespondierende CORBA-IDL-Datentypen abgebildet. Dies garantiert eine plattformunabhängige Datenrepräsentation, da der Object Request Broker (ORB) bzw. die Sprachanbindungen der IDL-Interface-Beschreibungen für die notwendigen systemabhängigen Transformationen sorgen. Beispielsweise wird der *Integer*-Datentyp in verschiedenen Hardwareplattformen, Betriebssytemen und Programmiersprachen unterschiedlich repräsentiert.

In Abbildung 51 ist dargestellt, wie die DB-Objekte für die Reise durch den ORB „verpackt“ werden. Eine Instanz der Klasse *ObjExchangeRep* (Objekt-Austausch-Repräsentation) ist dabei der äußere Karton dieser Verpackung. Die Herkunft (*DatabaseName, ObjTypName*), die lokale Objekt-Identität (*LOID*) und der aktuelle Zeitstempel (*ObjTimestamp*) deklarieren dessen Inhalt, der sich aus den Werte-Belegungen der Instanz-Eigenschaften ergibt (Aggregationbeziehung *PropertyValues*).

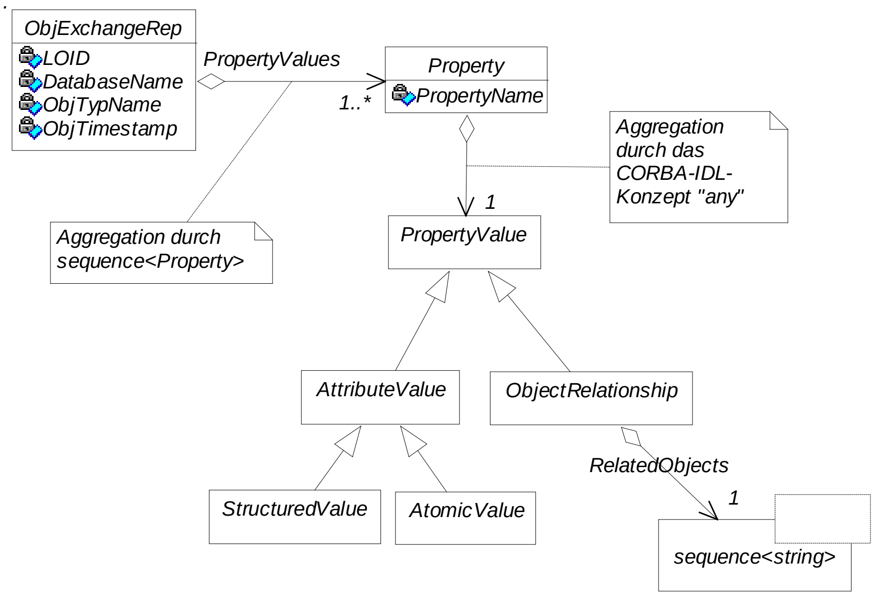


Abbildung 51 Repräsentation des DB-Objekt-Zustandes

Eine Werte-Belegung beinhaltet den Namen der Instanz-Eigenschaft (*PropertyName*) und dessen Wert (Aggregationsbeziehung zu *PropertyValue*). Der Wert der Eigenschaft wird dabei in ein CORBA-IDL ***any***-Element verpackt, welches das typsichere Ein- und Auspacken beliebiger IDL-Datentypen (einschließlich aller komplexen Datentypen) erlaubt [Siegel96].

Instanz-Eigenschaft kann ein Attribut (*AttributeValue*) oder eine Beziehung (*ObjectRelatioship*) zu anderen Objekten sein. Eine Beziehung wird als dynamisches Array (IDL-Typkonstruktor *sequence<T>*) der lokalen *OIDs* (*LOID*) der assozierten Objekte repräsentiert. Der Wertebereich der Attribut-Werte zerfällt in atomare und strukturierte Datentypen. Abbildung 52 stellt die Repräsentation der im ODMG-93-Objektmodell vordefinierten strukturierten Datentypen dar [Cat94]. Diese werden auch als „*built-in-Types*“ bezeichnet. Der Datentyp *d_string<n>* wird auf das korrespondierende CORBA-IDL-String-Konzept abgebildet (*d_string<n> := string<n> := sequence<char,n>*).

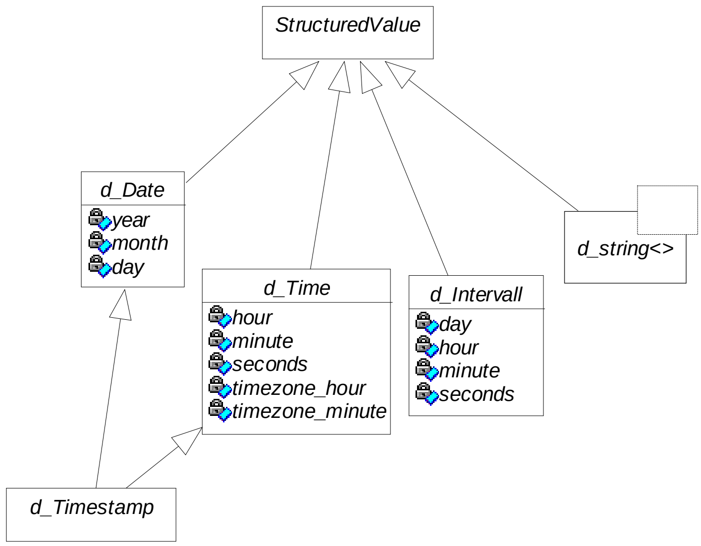


Abbildung 52 Die strukturierten „*built-in*“ Datentypen

Die atomaren Datentypen des ODMG-93-Objektmodells (siehe Abbildung 53) werden auf die entsprechenden CORBA-IDL-Datentypen abgebildet, da CORBA-IDL in diesem Punkt vollständig kompatibel zu ODMG-93-ODL ist.

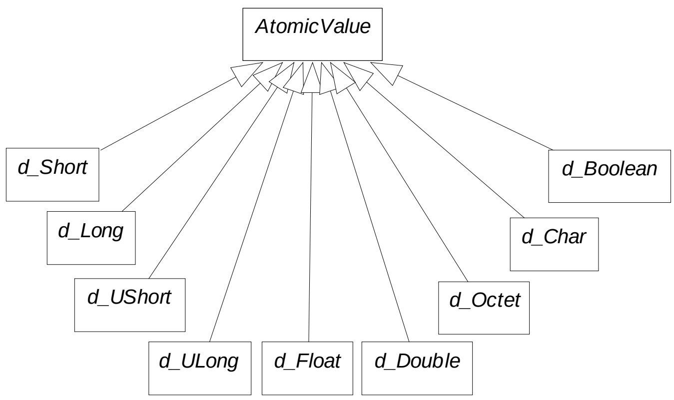


Abbildung 53 Die atomaren Datentypen

6.6 Der FDBS-Replication-Manager

Die Aufgabe des Replication-Managers besteht darin, den im Kapitel 5.3.3 dargestellten *Publisher-Subscriber-Replikationsansatz* zu realisieren. Dieser Ansatz respektiert insbesondere die Autonomie der Komponenten-Datenbanksysteme und ermöglicht eine Replikation von DB-Objekten auf Basis der Komponenten-Schemata. Wie bereits in Abschnitt 6.3.3 dargestellt wurde, exportieren die Datenbank-Adapter mithilfe eines speziellen Ankopplungs-Modus ihre Komponenten-Schemata in das Data-Dictionary des föderierten Datenbanksystems. Diese Schemata werden persistent in der Hilfsdatenbank des FDBS verwaltet und sind die Grundlage für die Spezifikation der Replikations-Beziehungen zwischen den einzelnen Komponenten-Datenbanksystemen. Die Replikations-Beziehungen zwischen den KDBS in ihrer Gesamtheit werden im folgenden als Replikations-Schema des FDBS bezeichnet. Dieses dient dabei dem Replication-Manager als Operationsvorschrift zur Durchführung der Replikation.

6.6.1 Die Verwaltung des Replikations-Schemas

Das Replikations-Schema wird, genauso wie alle anderen Meta-Daten des FDBS, in der Hilfsdatenbank (POET) verwaltet. Abbildung 54 zeigt das „Meta-Schema“ zur Verwaltung des Replikations-Schemas in UML-Notation. Die Instanzen der Klasse *CReplicationSpec* definieren die Replikations-Beziehungen zwischen den Komponenten-Datenbanksystemen des FDBS.

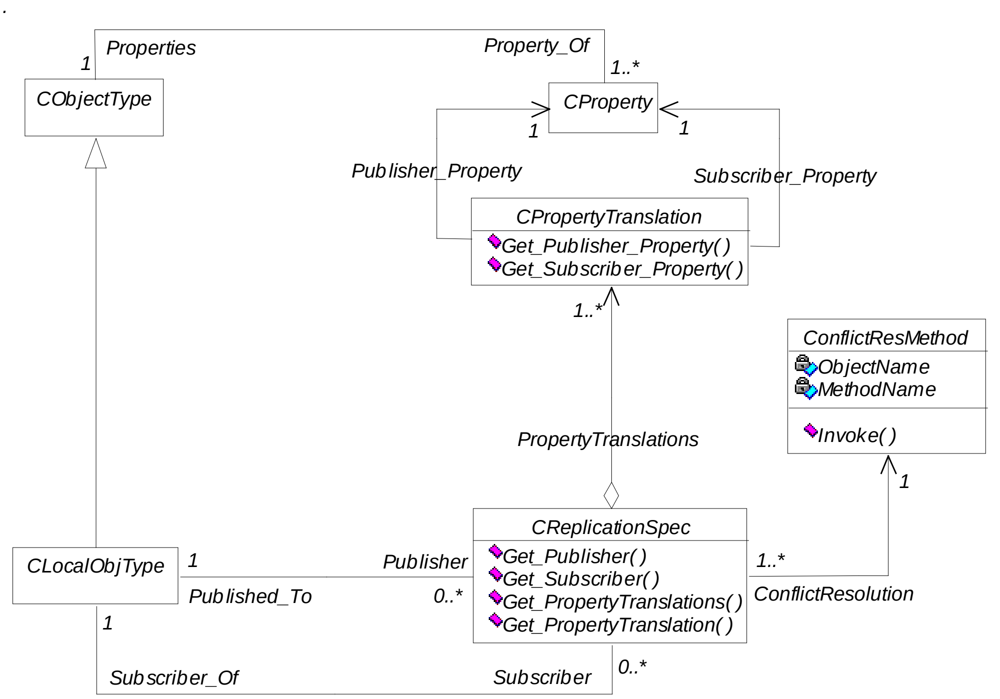


Abbildung 54 „Meta-Schema“ zur Verwaltung des FDBS-Replikations-Schemas

Gemäß dem *Publisher-Subscriber-Replikationsansatz* (vergleiche hierzu Kapitel 5.3.3) besteht eine Replikations-Beziehung zwischen zwei Objekttypen, die einen gemeinsamen Realweltausschnitt modellieren (intensionale Überlappung), deren Extensionen jedoch vor der Integration disjunkt sind (extensionale Überlappung = ∅). Die Objekttypen nehmen in einer Replikations-Beziehung die Rolle eines *Publishers* und eines *Subscribers* ein. Sowohl der *Publisher-Objekttyp* als auch der *Subscriber-Objekttyp* sind lokale Objekttypen (Instanzen von *CLocalObjType*) und somit Schema-Elemente eines Komponenten-Schemas. Die Instanzen dieser Objektypen werden also in den Komponenten-Datenbanksystemen physisch gespeichert. Für die Komponenten-Datenbanksysteme bzw. deren Datenbank-Adapter sind die *Publisher-Subscriber-Beziehungen* zwischen Objekttypen nicht sichtbar.

Der *Publisher-Objektyp bzw.* dessen Extension ist die Datenquelle der Replikations-Beziehung. Zu jedem DB-Objekt, das in der Extension des *Publisher-Objektyps* neu angelegt wird (abgesehen von Replikaten), erzeugt der Replication-Manager ein Replikat in der Extension des *Subscriber-Objektyps*. Das originäre DB-Objekt wird hierbei als *Publisher-Copy*, das Replikat als *Subscriber-Copy* bezeichnet. Änderungs-Operationen können gleichberechtigt auf dem *Publisher-Copy-Objekt* und dem *Subscriber-Copy-Objekt* angewendet werden.

Ein lokaler Objekttyp kann als *Publisher* für mehrere andere lokale Objekttypen (*Subscriber*) fungieren. Dies wird im „*Meta-Schema*" durch die 1:n-Assoziation (*Published_To / Publisher*) zwischen *CLocalObjType* und *CReplicationSpec* modelliert. Umgekehrt kann ein lokaler Objekttyp *Subscriber* von mehreren *Publisher-Objektypen* sein (1:n-Assoziation *Subscriber_Of / Subscriber*). Darüber hinaus ist es auch möglich, daß ein lokaler Objekttyp zugleich *Publisher* und *Subscriber* eines anderen lokalen Objekttyps ist. Diese Art von Replikations-Beziehung wird durch zwei Instanzen der Klasse *CReplicationSpec* ausgedrückt, bei denen jeweils die *Publisher- Subscriber-Rollen* paarweise vertauscht sind.

Die Behandlung von Update-Konflikten, die aufgrund der optimistischen Kopien-Update-Strategie des gewählte Replikationsansatzes auftreten können, wird an andere System-Instanzen (Administratoren / externe Programme) delegiert (vergleiche Kapitel 5.3.3). Die Delegation erfolgt mithilfe des *Dynamic-Invocation-Interface (DII)* des Object Request Brokers (vergleiche Kapitel 2.3.1). Eine Instanz der Klasse *ConflictResMethod* kapselt dabei den Aufruf (*Invoke*) einer Konflikt-Auflösungs-Methode welche als Bestandteil eines CORBA-Objektes von den Administratoren des FDBS zu implementieren ist. Diese wird durch ihren Namen (*MethodName*) und den Namem des CORBA-Objektes mithilfe des *CORBA-Name-Service* identifiziert.

Damit der Replication-Manager zu einem *Publisher-Copy-Objekt* überhaupt korrespondierende *Subscriber-Copy-Objekte* in den Extensionen der *Subscriber-Objekttypen* erzeugen kann, ist es notwendig die Instanz-Eigenschaften (Attribute und Beziehungen) der beteiligten Objekttypen aufeinander abzubilden. Diese Abbildung wird durch die Klasse *CPropertyTranslation* modelliert. Die Instanzen von *CPropertyTranslation* sind dabei Bestandteil einer *Publisher-Subscriber-Beziehung* (Instanz von *CReplicationSpec*). Dies wird durch die Aggregationbeziehung *PropertyTranslations* (siehe Abbildung 54)) modelliert. Sie bilden jeweils eine Eigenschaft (*CProperty*) des *Publisher-Objektyps* auf eine korrespondierende Eigenschaft des *Subscriber-Objekttyps* ab, wobei nicht unbedingt alle Eigenschaften Berücksichtigung finden müssen (intensionale Überlappung). Abbildung 55 zeigt ein Beispiel einer *Publisher-Subscriber-Beziehung* als Instanz-Diagramm. Bei der Abbildung der Instanz-Eigenschaften ist folgendes zu beachten:

- Die Wertebereiche der aufeinander abgebildeten Eigenschaften müssen kompatibel sein. Beispielsweise ist es nicht möglich, ein Attribut mit dem Wertebereich *Float* (Fließkommazahl) auf ein Attribut mit dem Wertebereich *string* (Zeichenkette) abzubilden.

- Bei der Abbildung von Beziehungen bzw. deren Zugriffs-Pfade (*CRelTravPath* siehe Abschnitt 6.3.2) muß deren Kardinalität übereinstimmen. Die Ziel-Objekttypen der Zugriffs-Pfade müssen ebenfalls durch eine *Publisher-Subscriber-Beziehung* aufeinander abgebildet werden, so daß die durch einen Zugriffs-Pfad assoziierten DB-Objekte auch als *Subscriber-Kopien* im KDBS des Subscribers vorliegen.

- Eine *Publisher-Subscriber-Beziehung* muß alle Eigenschaften, in denen die Instanzen des *Publisher-* bzw. *Subscriber-Objekttyps* einen definierten Wert haben müssen (*Required-Klausel,* vergleiche Abschnitt 6.3.2), unbedingt berücksichtigen.

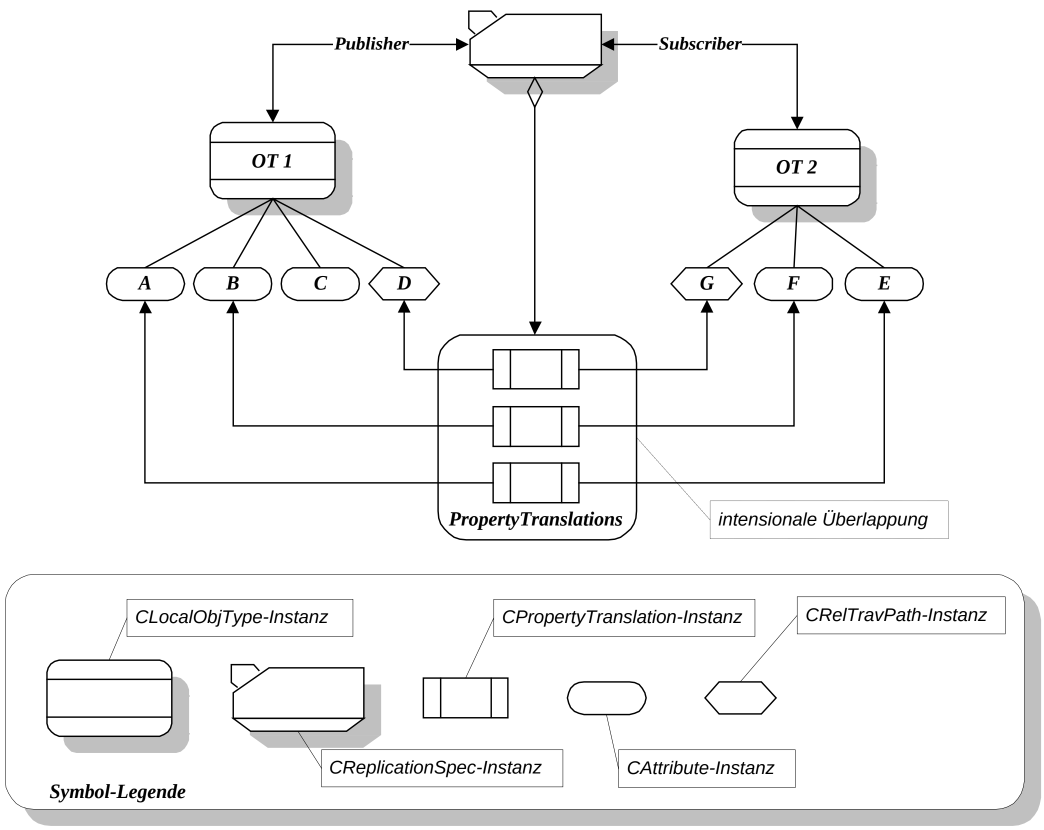


Abbildung 55 Instanz-Diagramm einer Publisher-Subscriber-Beziehung

6.6.2 Die Realisierung des Replikations-Mechanismus

Die Realisierung des Replikations-Mechanismus obliegt dem Replication-Manager. Dieser arbeitet dabei auf dem im letzten Abschnitt dargestellten Replikations-Schema und stützt sich auf die Dienste des Object-Managers und des Data-Dictionary des FDBS-Kerns ab.

Der Replication-Manager wird als einzige Instanz (*Singleton*) der Klasse *CFDBSReplicationManager* realisiert (siehe Abbildung 56). Er hält eine Referenz auf den globalen Object-Manager (*CFDBSObjectManager*), um dessen Dienste zu nutzen und verwaltet eine Menge von Transformations-Kommandos (*CTransformationCmd*). Während der Initialisierung des Replication-Managers ließt dieser das gesamte Replikations-Schema aus der Hilfsdatenbank in den Hauptspeicher und erzeugt zu jeder *Publisher-Subscriber-Beziehung* ein entsprechendes Transformations-Kommando. Diese Transformations-Kommandos verwaltet er mithilfe von Hash-Tabellen im Hauptspeicher, um nicht bei jedem Replikations-Vorgang durch das Replikations-Schema navigieren zu müssen, da sich bei der Entwicklung des FDBS zeigte, daß der (navigierende) Zugriff auf die persistenten Meta-Daten in der Hilfsdatenbank (POET) sehr viel Zeit (im Vergleich zu den übrigen Systemaktionen wie z.B. die ORB-Kommunikation) in Anspruch nimmt.

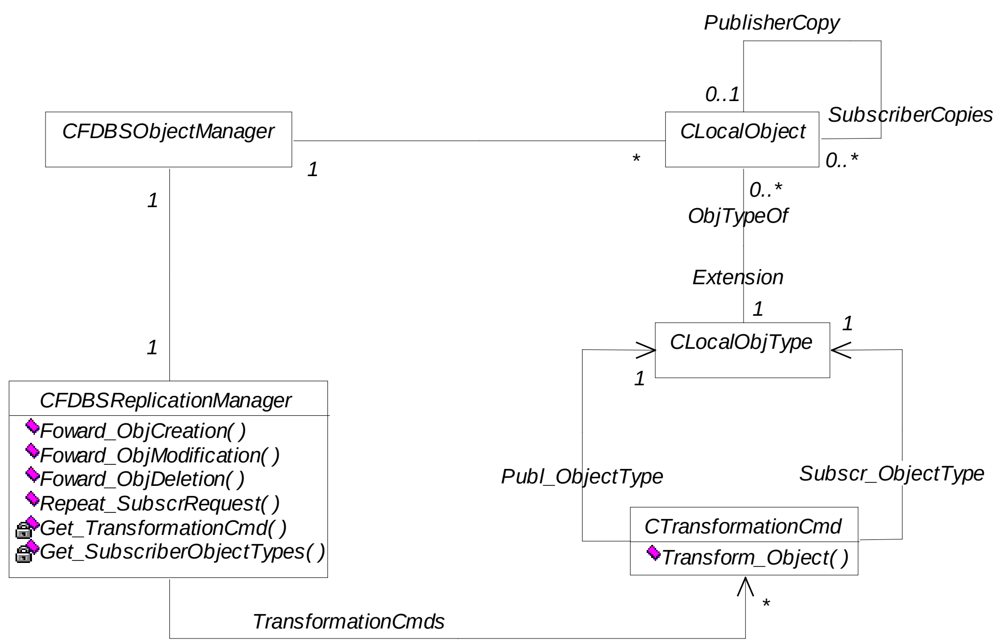


Abbildung 56 Klassen-Diagramm des Replication-Manager

Ein Transformations-Kommando (Instanz von *CTransformationCmd*) kapselt die gesamten Meta-Daten, die notwendig sind, um ein *Publisher-Copy-Objekt* in ein *Subscriber-Copy-Objekt* zu transformieren (und umgekehrt). Die Transformation der Objekt-Zustände erfolgt auf der Basis der im Abschnitt 6.5.4 dargestellten Repräsentation für DB-Objekte. Die Methode *Transform_Object* der Klasse *CTransforma-*

tionCmd realisiert die eigentliche Transformation. Parameter dieser Methode sind das Quell- und Ziel-Objekt der Transformation (Instanzen von *CLocalObject*). Je nach Replikations-Vorgang ist hierbei entweder das *Publisher-Copy-Objekt* Quell- und das *Subscriber-Copy-Objekt* Ziel-Objekt oder umgekehrt.

Die *Publisher-Subscriber-Beziehungen* des Replikations-Schemas implizieren eine Instanz-Beziehung zwischen DB-Objekten in der Extension der lokalen Objekttypen. Diese Instanz-Beziehung wird im folgenden als *Publisher-Copy / Subscriber-Copy-Beziehung* bezeichnet und wird persistent in der Hilfsdatenbank verwaltet. Die *Proxy-Objekte* (*CLocalObject*) der lokalen DB-Objekte (siehe Abschnitt 6.5.2) stehen dabei stellvertretend in Beziehung zueinander (*PublisherCopy / SubscriberCopies* siehe Abbildung 56). Die Operationen *Is_PublisherCopy*, *Is_SubscriberCopy*, *Get_PublisherCopy* und *Get_SubscriberCopies* der Klasse *CLocalObjType* (siehe Abbildung 49 in Abschnitt 6.5.3 auf Seite 77) kapseln die Navigation entlang dieser Beziehung. Im folgenden soll der dynamische Aspekt bei der Realisierung des Replikations-Mechanismus beschrieben werden. Hierzu wird die Interaktion des Replication-Managers mit den anderen Systemkomponenten am Beispiel von drei Szenarien dargestellt.

Erzeugung eines DB-Objektes

Wird ein neues DB-Objekt in der Extension eines lokalen Objekttyps erzeugt, so wird dies dem Replication-Manager durch den globalen Object-Manager mitgeteilt (*Forward_ObjCreation*), nachdem dieser vom Event-Handler darüber informiert wurde (vergleiche hierzu Abbildung 50 in Abschnitt 6.5.3).

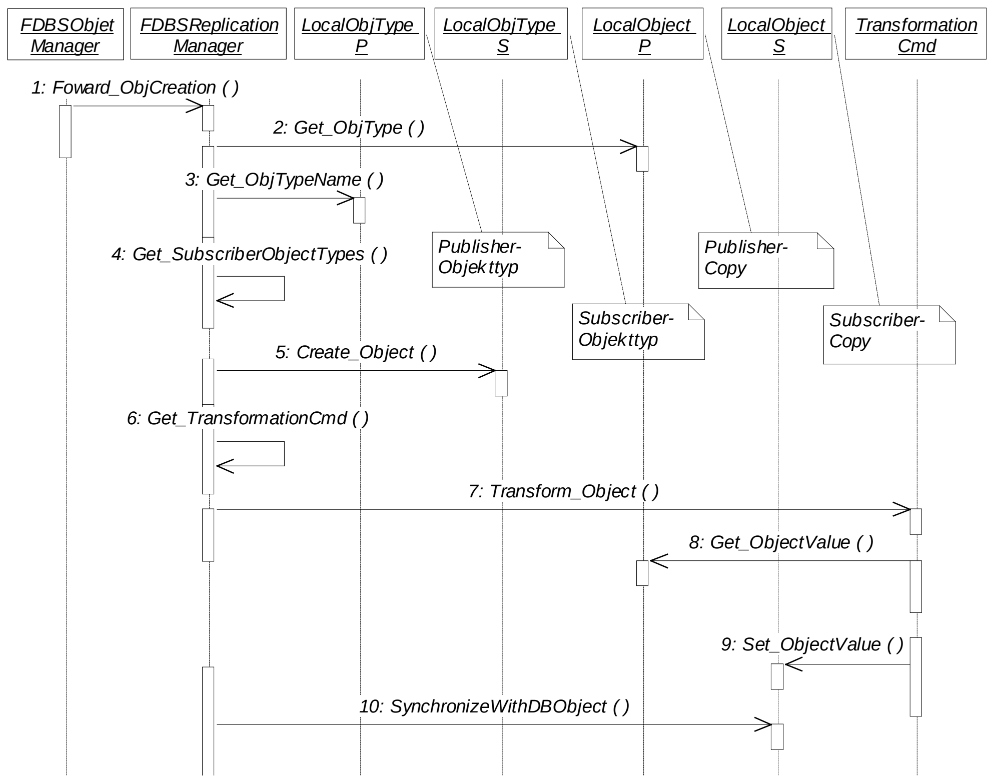


Abbildung 57 Replikations-Mechanismus nach der Erzeugung eines DB-Objektes

Abbildung 57 zeigt nun den weiteren Ablauf dieses Szenarios und führt die in Abbildung 50 dargestellte Interaktion fort:

- Der Replication-Manager stellt zuerst den Objekttypen des neuen DB-Objektes fest, indem er dessen *Proxy-Objekt* danach befragt (*Get_ObjType*). Das *Proxy-Objekt* wurde bereits vom globalen Object-Manager angelegt, bevor der Replication-Manager involviert wurde (siehe Abbildung 50).
- Durch den Namen des lokalen Objekttyps ermittelt der Replication-Manager mithilfe seiner Hash-Tabellen die möglichen *Subscriber-Objekttypen* (*Get_SubscriberObjectTypes*).
- Für jeden *Subscriber-Objekttyp* erzeugt er nun ein Replikat (*Subscriber-Copy*):
  - Hierzu erzeugt der Replication-Manager zunächst das zum *Subscriber-Copy-Objekt* korrespondierende *Proxy-Objekt* (*Create_Object*).
  - Anschließend bestimmt er das entsprechende Transformations-Kommando mithilfe seiner Hash-Tabellen (*Get_TransformationCmd*) und ruft dessen *Transform_Object*-Methode auf. Als Parameter übergibt er hierbei das *Publisher-Copy-Objekt* und das *Subscriber-Copy-Objekt* (bzw. deren *Proxy-Objekte*) als Quell- bzw. Ziel-Objekt.
  - Die *CTransformationCmd*-Instanz leitet nun aus dem Zustand des *Publisher-Copy-Objektes* (Get_ObjValue) den Zustand des *Subscriber-Copy-Objektes* (*Set_ObjectValue*) ab.
  - Abschließend leitet der Replication-Manager die Synchronisierungs-Phase des *Proxy-Objektes* (*Subscriber-Copy*) mit dem lokalen Pendant ein (siehe Abschnitt 6.5.2).

Die Propagation von Änderungs-Operationen

Eine Änderungs-Operation kann sowohl auf einem *Publisher-Copy-Objekt*, als auch auf einem *Subscriber-Copy-Objekt* innerhalb einer lokalen Transaktion ausgeführt werden. Im folgenden wird der Fall eines Updates auf einem *Subscriber-Copy-Objekt* betrachtet. Das Update-Ereignis gelangt über den Weg Event-Handler ⇒ globaler Object-Manager zum Replication-Manager (*Forward_ObjModification*). Abbildung 58 stellt die weitere Behandlung des Ereignisses durch den Replication-Manager dar:

- Der Replication-Manager stellt zunächst fest, ob es bei dem geänderten DB-Objekt um eine Subscriber-Copy handelt (*Is_SubscriberCopy*). Da dies der Fall ist, navigiert (*Get_PublisherCopy*) er zuerst zum entsprechenden *Publisher-Copy-Objekt* (bzw. dessen *Proxy-Objekt P*). Über dessen Referenz gelangt der Replication-Manager an die übrigen *Subscriber-Copy-Objekte* (*Get_SubscriberCopies*).
- Im folgenden vergleicht der Replication-Manager für alle Kopien die Zeitstempel der *Proxy-Objekte* mit den Zeitstempeln der lokalen DB-Objekte (Instanzen von *CDBObject* siehe Abschnitt 6.5.2). Ein solcher Vergleich impliziert eine Folge von Operations-Aufrufen, die ist in Abbildung 58 durch die Pseudo-Operation *CompareTimestamps* angegeben ist, um die Übersicht zu Bewahren. Hierbei wer-

den insbesondere die lokalen DB-Objekte in den KDBS durch die Datenbank-Adapter gelesen und dabei Sperren (*Locks*) gesetzt. Stimmen die lokalen Zeitstempel mit denen der *Proxy-Objekte* überein, so liegt kein Update-Konflikt vor. Im anderen Fall wäre eine Konflikt-Auflösung notwendig, die in Abschnitt 6.6.3 beschrieben wird.

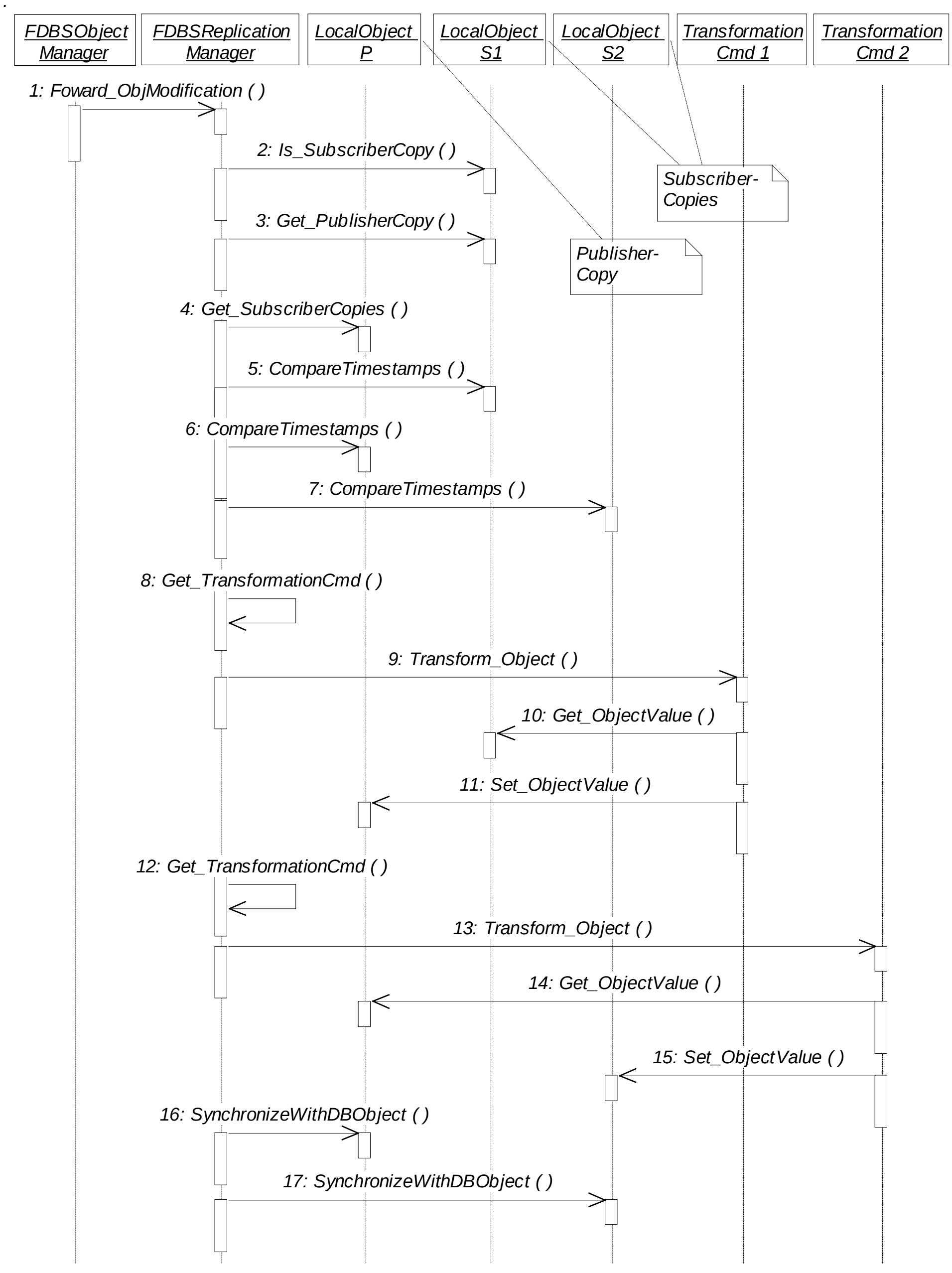


Abbildung 58 Replikations-Mechanismus nach dem Update einer Subscriber-Copy

Wird an dieser Stelle festgestellt, daß das KDBS (bzw. dessen Datenbank-Adapter), in dem die *Publisher-Copy* gespeichert ist, momentan nicht erreichbar ist, so wird das Status-Attribut ***Defered-SubscrRequest*** des *Proxy-Objektes S1* (*Subscriber-Copy*) auf TRUE gesetzt, um die Propagation der

Änderungs-Operation bis zur Wiedererreichbarkeit dieses KDBS zu verzögern (vergleiche hierzu Abschnitt 6.5.3).

- Da im Beispiel vom Replication-Manager kein Update-Konflikt festgestellt wurde, überträgt dieser zunächst die Zustands-Änderung der *Subscriber-Copy S1* auf den Zustand der *Publisher-Copy P*:

  - Hierzu bestimmt er mithilfe seiner Hash-Tabellen das entsprechende Transformations-Kommando (*Get_TransformationCmd*) und übgibt diesem als Parameter das *Subscriber-Copy-Objekt S1* als Quell-Objekt und das *Publisher-Copy-Objekt P* als Ziel-Objekt.

  - Das Transformations-Kommando (*CTransformationCmd*) leitet nun aus dem Zustand des *Subscriber-Copy-Objektes S1* (Get_ObjValue) den neuen Zustand des *Publisher-Copy-Objektes P* (*Set_ObjectValue*) ab.

- Anschließend wird der Zustand der weiteren *Subscriber-Copy-Objekte (*im Beispiel *S2*) aus dem neuen Zustand des *Publisher-Copy-Objektes P* abgeleitet. Die Folge der Operations-Aufrufe *12* bis *15* in Abbildung 58 wiederholt sich dabei für jedes weitere *Subscriber-Copy-Objekt*. Ist jedoch ein *Subscriber-Copy-Objekt* bzw. dessen lokale Instanz (*CDObject*) aktuell nicht erreichbar, so wird das Attribut ***InSynch*** des korrespondierenden Proxy-Objektes auf FALSE gesetzt (vergleiche hierzu Abschnitt 6.5.3). Nach Wiederaufnahme der Verbindung zwischen dem entsprechenden Datenbank-Adapter und dem FDBS-Kern wird die Änderungs-Operation dann nachgezogen.

- Nachdem nun die Änderungs-Operation auf die *Proxy-Objekte* aller Kopien übertragen wurde, stößt der Replication-Manager abschließend die Synchronisierungs-Phase (*SynchronizeWithDBObject*) der *Proxy-Objekte* an (siehe Abschnitt 6.5.2).

Eine Änderungs-Operation auf einem *Publisher-Copy-Objekt* wird analog zum obigen Szenario behandelt. Der einzige Unterschied hierbei besteht darin, daß die Aktionsfolge *8* bis *11* in Abbildung 58 entfällt, da lediglich die Zustands-Änderung des *Publisher-Copy-Objektes* an alle *Subscriber* zu propagieren ist.

Die Propagation von Lösch-Operationen

Wird ein lokales DB-Objekt gelöscht, so wird der Replication-Manager durch den globalen Object-Manager darüber informiert (*Forward_ObjDeletion*). Das Szenario einer Lösch-Operation ist in Abbildung 59 dargestellt. In diesem Beispiel wurde ein *Subscriber-Copy-Objekt* innerhalb einer lokalen Transaktion gelöscht. Wird ein *Publisher-Copy-Objekt* gelöscht, so wird dies vom Replication-Manager analog behandelt. Der weitere Ablauf der Behandlung der Lösch-Operation durch den Replication-Manager stellt sich nun folgendermaßen dar:

- Der Replication-Manager bestimmt genau wie im obigen *Update-Szenario* alle involvierten Kopien (*Publisher-Copy* und *Subscriber-Copies*) und vergleicht die Zeitstempel der *Proxy-Objekte* (*P* und

S2) mit denen der lokalen DB-Objekte (Instanzen von *CDBObject*) um eventuelle Update-Konflikte zu entdecken.

- Wurden keine Konflikte festgestellt, so werden alle verbleibenden Kopien vom Replication-Manager gelöscht:

    - Hierzu ruft der Replication-Manager für jede verbleibende Kopie die *delete*-Operation des entsprechenden *Proxy-Objektes* (*P* und *S2*) auf.

    - Die *delete*-Operation der *Proxy-Objekte* setzt deren Status-Attribut ***deleted*** auf TRUE (vergleiche Abschnitt 6.5.3) und leitet unmittelbar die Synchronisierung (*SynchronizeWithDBObject*) mit dem korrespondierenden lokalen DB-Objekt (*CDBObject*) ein.

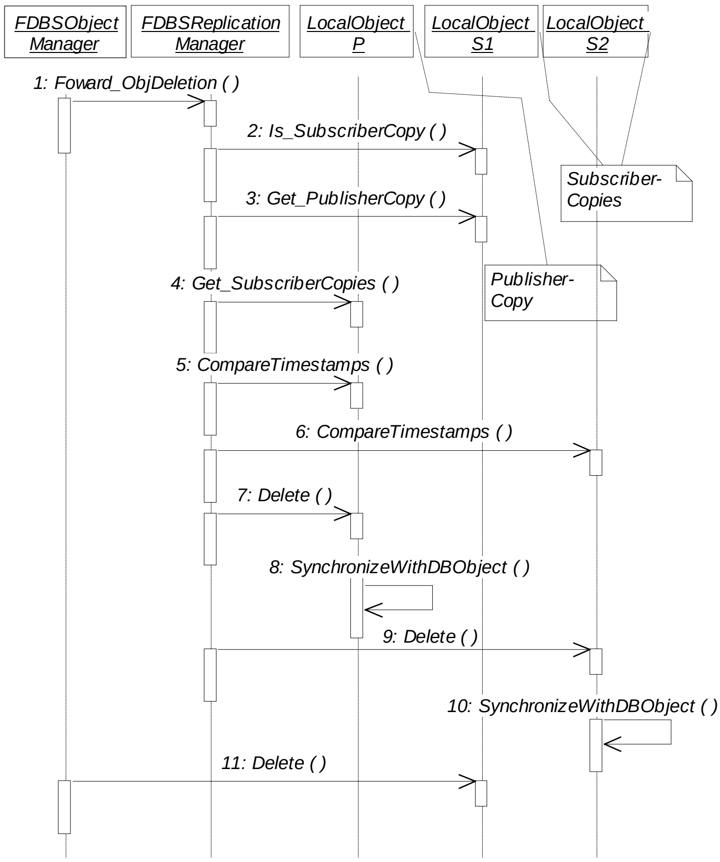


Abbildung 59 Replikations-Mechanismus nach einer Lösch-Operation

- Nachdem die *Forward_ObjDeletion*-Operation terminiert, löscht der globale Object-Manager das *Proxy-Objekt* (*S1*) des lokalen DB-Objektes dessen Lösch-Operation ursächlich war.

- Die *Proxy-Objekte* der übrigen Kopien (*P* und *S2*) werden vom globalen Object-Manager erst dann gelöscht, wenn er die Bestätigung (*ConfirmDeletion*) der lokalen Object-Manager erhält, daß die lokalen DB-Objekte gelöscht wurden (in Abbildung 59 nicht dargestellt, siehe Abschnitt 6.5.3).

6.6.3 Die Auflösung von Update-Konflikten

Aufgrund der optimistischen Kopien-Update-Strategie des gewählte Replikationsansatzes können Update-Konflikte auftreten (vergleiche Kapitel 5.3.3). Mögliche Update-Konflikte werden vom Replication-Manager erkannt (siehe Abschnitt 6.6.2) und an andere System-Instanzen (Administratoren / externe Programme) delegiert, um deren semantisches Wissen über die Daten bei der Auflösung der Konflikte auszunutzen. Die Delegation geschieht hierbei mithilfe des *Dynamic-Invocation-Interface (DII)* des Object Request Brokers (vergleiche Kapitel 2.3.1). Die Administratoren des FDBS haben die Aufgabe, externe CORBA-Objekte zu implementieren, welche die Konflikt-Auflösung oder Weiterleitung realisieren.

Im Falle eines Update-Konfliktes ruft der Replication-Manager die im Replikations-Schema hinterlegte Konflikt-Auflösung-Methode eines solchen externen CORBA-Objektes auf (siehe Abbildung 56 in Abschnitt 6.6.1). Die Instanzen der Klasse *ConflictResMethod* kapseln dabei das Aufinden des externen CORBA-Objektes (*CORBA-Name-Service*) und das dynamische Konstruieren des Methodenaufrufs. Die *invoke* Methode der Klasse *ConflictResMethod* dient hierbei als „*Wrapper*“ des dynamische Methodenaufrufs (*Dynamic-Method-Invocation*) der Konflikt-Auflösungs-Methode. Als ***inout*** Parameter (siehe Kapitel 2.4) werden die in Konflikt geratenen DB-Objekt-Kopien in der im Abschnitt 6.5.4 dargestellten plattformunabhängigen CORBA-IDL-Austausch-Repräsentation übergeben. Die in Konflikt geratenen Kopien werden jedoch zuvor in das Format (Struktur) des *Publisher-Objektyps* transformiert, um deren Vergleichbarkeit zu gewährleisten. Dies geschieht mithilfe der im Abschnitt 6.6.2 dargestellten Transformations-Kommandos (*CTransformationCmd*). Diese Transformation ist deshalb notwendig, weil der *Publisher-Objektyp* den gemeinsamen Nenner (intensionale Überlappung) aller Kopien darstellt. Die Konflikt-Auflösungs-Methode erhält also als Parameter quasi unterschiedliche Versionen des Publisher-Objekt-Zustandes.

Für die Konflikt-Auflösungs sind nun generell zwei Möglichkeiten denkbar:

- Die Konflikt-Auflösungs-Methode löst den Update-Konflikt selbst, indem sie semantisches Wissen der Diskurswelt implementiert und die in Konflikt geratenen Kopien (***inout*** Parameter des Methoden-Aufrufs) in einen konsistenten Zustand überführt. Sie signalisiert dies dem Replication-Manager, indem sie den Wert TRUE als Rückgabewert liefert.

- Ein auftretender Konflikt wird lediglich protokolliert, etwa in einem *Logfile*. Die Konflikt-Auflösungs-Methode läßt die in Konflikt geratenen Kopien unverändert und liefert FALSE als Rückgabewert an den Replication-Manager. Hierbei ist es die Aufgabe der Administratoren bzw. Benutzer, den Konflikt aufzulösen.

Falls der Konflikt durch die Konflikt-Auflösungs-Methode gelöst wurde, so leitet der Replication-Manager den Zustand aller *Subscriber-Copy-Objekte* aus dem neuen Zustand des *Publisher-Copy-Objektes* (***inout*** Parameter) ab.

Kapitel 7 Der Entwurf und die Realisierung von Datenbank-Adaptern

In dem nun folgenden Kapitel soll der Entwurf der Datenbank-Adapter des föderierten Datenbanksystems dargestellt werden. Die Entwicklung eines Datenbank-Adapters ist dabei durch das kanonische Datenmodell und das Interface zum FDBS-Kern determiniert. Die Schnittstellen zwischen den Datenbank-Adaptern und dem FDBS-Kern sind in der CORBA-Interface-Definition-Language (IDL) spezifiziert und wurden bereits in Kapitel 6 dargestellt. Diese stellen die Kooperationsgrundlage (*Kontrakt*) zwischen dem FDBS-Kern und den DB-Adaptern dar. Als kanonisches Datenmodell des föderierten Datenbanksystems dient der ODMG-93-Standard. Entsprechend der 5-Ebenen-Schema-Architektur nach Sheth und Larson [SL90] besteht die Aufgabe eines Datenbank-Adapters nun darin, den Übergang vom lokalen Datenmodell (lokales Schema) zum kanonischen Datenmodell (Komponenten-Schema) für den FDBS-Kern transparent zu machen. Dieser Prozeß wird als Schema-Transformation oder Homogenisierung bezeichnet [BFN94, HoPl95, KlaFiAb94].

7.1 Der gewählte Schema-Transformations-Ansatz

Der nun folgende Abschnitt beschreibt einen Schema-Transformations-Ansatz für relationale Komponenten-Datenbanksysteme, der zum Ziel hat, einen Großteil der impliziten Semantik in relationalen Datenbank-Schemata in den Komponenten-Schemata wieder explizit zu machen, um so die Integration der heterogenen Datenbestände der KDBS zu erleichtern. Dieser Aspekt der Schema-Transformation wird als semantische Anreicherung [HoPl95] bezeichnet. Dies ist natürlich nur möglich, wenn die Ausdrucksstärke des kanonischen Datenmodells größer als die des relationalen Datenmodells ist [SCG91].

Relationale Datenbank-Schemata werden heute häufig mithilfe semantischer Datenmodelle, wie z.B. dem Entity-Relationship-Modell modelliert. Viele semantische Aspekte der ursprünglichen Modellierung, wie z.B. Generalisierungs / Spezialisierungs oder Assoziations-Beziehungen gehen bei der Transformation in ein relationales Datenbank-Schema verloren [Kroh93, JaSchäZü96], da das relationale Datenmodell nur einige wenige Datenstrukturierungs-Konzepte unterstützt. Ein weiterer Grund für diesen semantischen Verlust ist der Prozeß der Normalisierung [HoPl95], der zur Folge hat, daß die ursprünglichen Objekttypen des logischen DB-Schemas jeweils durch eine Vielzahl von Relationen repräsentiert werden und der ursprüngliche Entwurf sich häufig nicht mehr erkennen läßt. Ziel der Normalisierung ist es, Redundanz durch die Berücksichtigung von funktionalen Abhängigkeiten zu beseitigen [Date90]. Demgegenüber steht, daß Redundanz manchmal bewußt in relationalen DB-Schemata eingesetzt wird, um den Lese-Zugriff auf Daten zu beschleunigen [RaHo96]. Insgesamt läßt sich feststellen, daß es eine Vielzahl unterschiedlicher Auffassungen bezüglich der Modellierung von relationalen Datenbank-Schemata gibt. Hierauf muß der Schema-Transformations-Ansatz Rücksicht nehmen.

7.1.1 Der Übergang von Tupeln zu Objekten

Bei der Entwicklung eines Schema-Transformations-Ansatzes stellt sich zunächst die Frage, welche Datenstrukturierungskonzepte des kanonischen Datenmodells überhaupt bei der Transformation der relationalen Schemata von Interesse sind. Erst dann sind Überlegungen anzustellen, wie die verschiedenen Modellierungs-Konzepte eines relationalen DB-Schemas darauf abgebildet werden können und wie diese Abbildung repräsentiert wird.

Im folgenden werden nun die Konzepte des kanonischen Datenmodells, die im Schema-Transformations-Ansatz Berücksichtigung finden, in Bezug auf das relationale Modell erläutert:

- Die *Objekttypen* des ODMG-93-Objektmodells entsprechen am ehesten den Relationen eines relationalen DB-Schemas. Jedoch entspricht nicht jede Relation einem *Objekttypen*. Darüber hinaus unterscheidet sich die Identifizierung von *DB-Objekten* und Tupeln völlig (siehe Kapitel 6.5.1). Im Prozeß der Schema-Transformation müssen also solche Relationen identifiziert werden, welche auf *Objekttypen* abbildbar sind. Zudem muß eine Abbildung gefunden werden, die Tupel auf *Objekte* abbildet und dabei die unterschiedlichen Konzepte der *Objekt-Identität* berücksichtigt.

- *Supertyp / Subtyp-Beziehungen* werden im relationalen Modell nicht direkt unterstützt. Vererbungsbeziehungen werden jedoch in relationalen Schemata häufig indirekt angewandt. Dabei existieren unterschiedliche Ansätze für die Repräsentation von Vererbungsbeziehungen in einem relationalen Schema [HoPl95].

- *Beziehungen* zwischen Objekten (Tupeln) werden im relationalen Modell ebenfalls nicht direkt unterstützt. Sie werden jedoch mithilfe von Fremdschlüsseln in einem relationalen Schema umgesetzt. Das Navigieren entlang dieser „relationalen Beziehungen“ ist ebenfalls nicht möglich, sondern erfolgt indirekt durch den Wertevergleich von Fremdschlüsseln und Primärschlüsseln (Join-Condition) innerhalb einer Datenbankabfrage (Verbundanfrage / Join). Es ist also die Aufgabe der Schema-Transformation, die eigentlichen Beziehungen wieder explizit zu machen.

- Wertebereich der *Attribute* eines Objektes im ODMG-93-Objektmodell sind wiederum Objekte (literale Objekte, siehe Kapitel 3.2.1). Diese können, im Gegensatz zu den Attributen einer Relation, bei denen nur atomare Datentypen als Wertebereich zulässig sind (1. Normalform), unter Umständen auch strukturiert sein. Ein Objekt mit strukturierten Attributen wird auch als komplexes oder aggregierendes Objekt bezeichnet. Komplexe Objekte zerfallen im relationalen Modell in eine Menge von

Tupeln, die über unterschiedliche Tabellen verstreut und durch Fremdschlüssel quasi zusammengekittet sind [Froese95]. Die Schema-Transformation könnte diese komplexen Objekte wieder explizit machen.

7.1.2 Skizzierung der Vorgehensweise

In diesem Abschnitt wird nun der realisierte Schema-Transformation-Ansatz dargestellt. Die grundlegenden Konzepte dieses Ansatzes sind in Abbildung 60 graphisch dargestellt. Die in Abbildung 60 angedeuteten ***Meta-Schemata*** visualisieren das Ergebniss des Abschitts 7.1 und sind in Abbildung 73 und Abbildung 74 am Ende dieses Abschnitts auf Seite 109 vollständig dargestellt. Der Ansatzes laßt sich nun folgendermaßen skizzieren:

- Die Konzepte des relationalen Datenmodells und des kanonischen Datenmodells (ODMG-93) werden durch die Klassen eines objektorientierten ***Meta-Schemas*** modelliert. Die Instanzen dieser Klassen entsprechen dabei den Schema-Elementen des lokalen Schemas (relationales Datenmodell) und des Komponenten-Schemas (ODMG-93-Objektmodell). Beispielsweise wird das Konzept *Relation* auf eine Klasse abgebildet, welche die Eigenschaften von Tabellen-Definitionen kapselt. Die Instanzen dieser Klasse beschreiben die Struktur von konkreten Relationen (Tabellen) eines lokalen Schemas. Die Instanzen der Klassen des *Meta-Schemas* beschreiben also die *Meta-Daten* des lokalen Schemas und des Komponenten-Schemas.

- Dieses Meta-Schema zerfällt entsprechend der Unterscheidung zwischen dem Komponenten-Schema und dem lokalen Schema in zwei Teilbereiche. Die Beziehungen zwischen den Klassen dieser Teilbereiche des *Meta-Schemas* modellieren die ***Schema-Transformations-Abbildung***. Das *Meta-Schema* beschreibt also bereits, wie die Konzepte des relationalen Datenmodells auf die des ODMG-93-Objektmodells abgebildet werden können. Die Beziehungen zwischen Instanzen der Klassen des *Meta-Schemas* repräsentieren eine konkrete ***Schema-Transformations-Abbildung*** eines existierenden relationalen DB-Schemas auf ein Komponenten-Schema.

- Die persistente Speicherung der Meta-Daten bzw. der *Schema-Transformations-Abbildung* erfolgt im relationalen Komponenten-Datenbanksystem. Hierzu wird das objektorientierte *Meta-Schema* auf ein korrespondierendes relationales *Meta-Schema* abgebildet. Für diese ***„two level store“***-Vorgehensweise (vergleiche Kapitel 5.3.1) gibt es mehrere Gründe:

  - Um einen möglichst effizienten Datenbank-Adapter zu entwickeln, müssen die Möglichkeiten (*Features*) eines relationalen Komponenten-Datenbanksystem ausgenutzt werden. Dies betrifft insbesondere, wie sich im weitern Verlauf dieses Kapitel noch zeigen wird, die Verlagerung eines Teils der Datenbank-Adapter-Logik auf den relationalen Datenbank-Server. Dies geschieht mithilfe von *Stored-Procedures* und *Triggern*. Da dieser Teil der Datenbank-Adapter-Logik ebenfalls auf den *Meta-Daten* bzw. der *Schema-Transformations-Abbildung* arbeitet, ist eine Speicherung der *Meta-Daten* im relationalen Komponenten-Datenbanksystem naheliegend.

- Die eigentlichen Datenbank-Adapter werden in der objektorientierten Programmiersprache C++ [Str91] entwickelt. Die Klassen des objektorientierten *Meta-Schemas* bzw. deren Instanzen sind die adäquate Repräsentationsform der *Meta-Daten* bzw. der *Schema-Transformations-Abbildung* in dieser Systemumgebung. Darüber hinaus realisieren sie den objektorientierten Zugriff des FDBS-Kerns auf die Schema-Elemente des Komponenten-Schemas beim Schema-Import. Die IDL-Interfaces der im Kapitel 6.3.3 dargestellten Schema-Element-Deskritoren sind hierbei eine Projektion der C++-Interfaces der entsprechenden Klassen des objektorientierten *Meta-Schemas*.

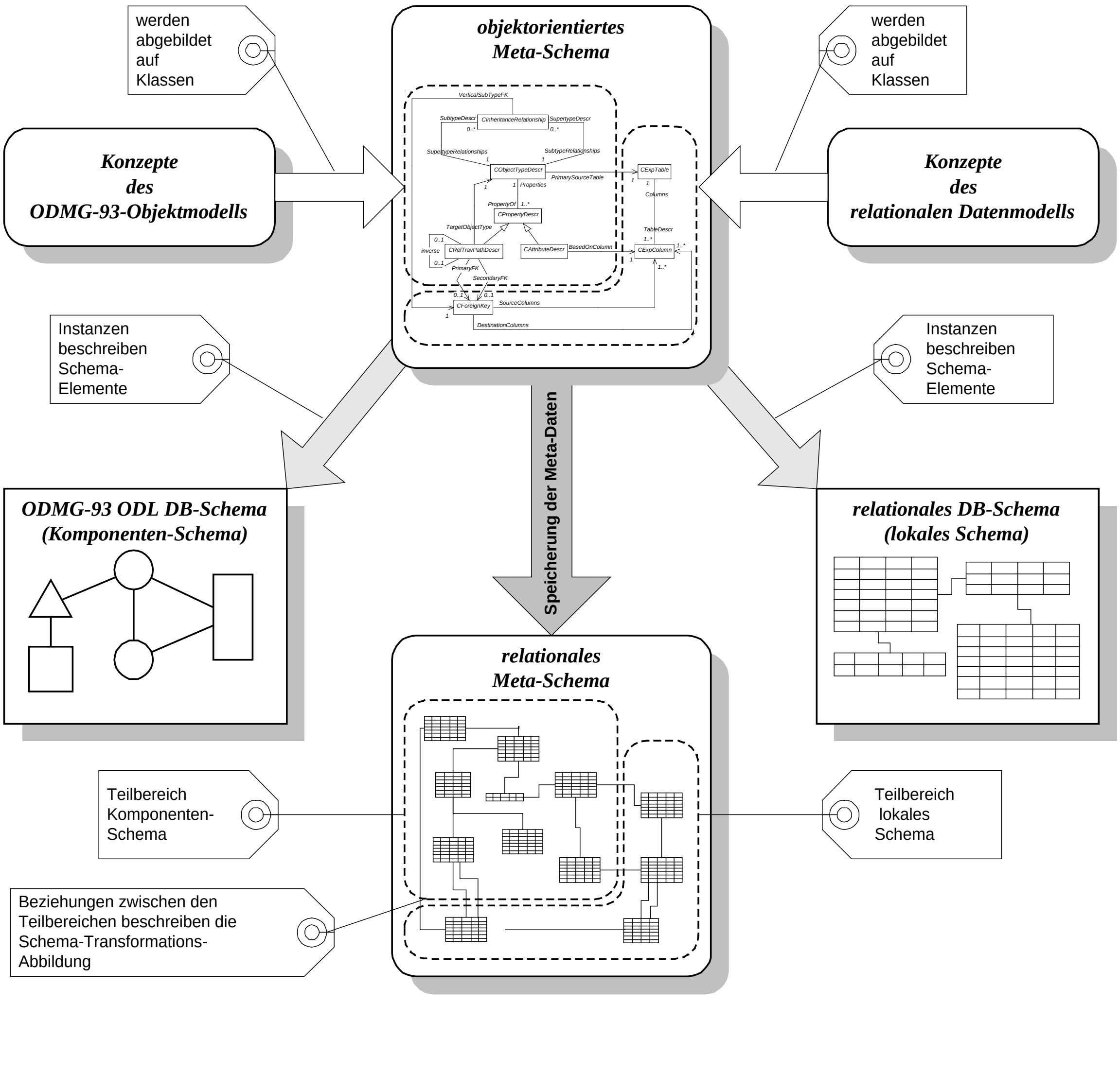


Abbildung 60 Graphische Darstellung des Schema-Transformations-Ansatzes

7.1.3 Abbildung der Konzepte des relationalen Datenmodells

Die Konzepte Relation, Attribut, Primärschlüssel und Fremdschlüssel des relationalen Datenmodells werden durch die Klassen *CExpTable*, *CExpColumn* und *CForeignKey* des objektorientierten Meta-Schemas abgebildet (siehe Abbildung 61). Das *Valid*-Attribut (*Is_Valid*) dieser Klassen dient beim Systemstart dazu, den Abgleich der *Meta-Daten* mit dem Systemkatalog des relationalen DBS zu dokumentieren. Wurden Änderungen am lokalen Schema vorgenommen, so liefert die *Is_Valid*-Methode der hiervon betroffenen Schema-Elemente den Wert FALSE. Diese Differenzen werden dem Administrator beim Start des Datenbank-Adapters detailliert mitgeteilt.

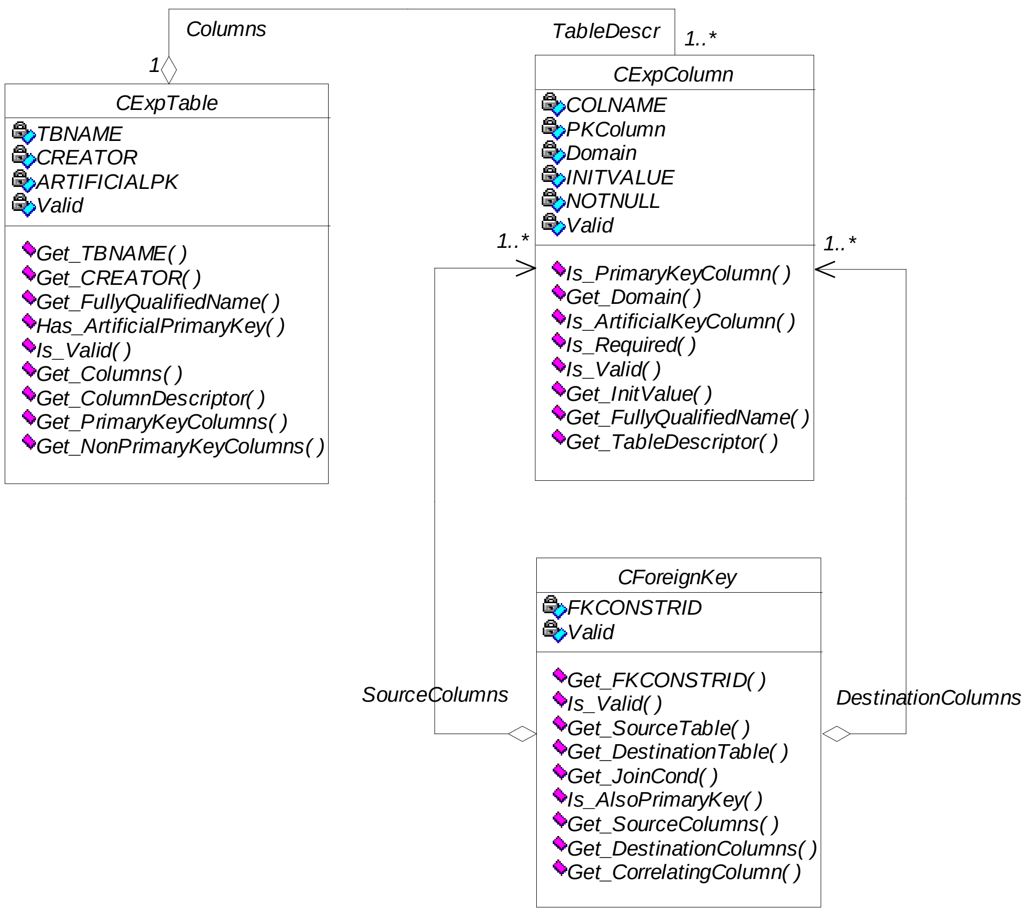


Abbildung 61 Abbildung der relationalen Konzepte im objektorientierten Meta-Schema

- Relationen (Tabellen) werden durch die Klasse *CExpTable* modelliert:
  - Eine Tabelle in einer relationalen Datenbank wird durch die Angabe ihres Namens (*TBName / Get_TBName*) und den Namen des Benutzers (*Creator / Get_Creator*), der sie angelegt hat, eindeutig identifiziert (*Get_FullyQualifiedName := TBName.Creator*).

    - Eine Tabellen-Definition beinhaltet die Angabe ihrer Attribute (Aggregations-Beziehung *Columns / TableDescr*).

    - Der Primärschlüssel ist durch die Menge der Primärschlüssel-Attribute gegeben (*Get_PrimaryKeyColumns*). Hierbei kann es sich auch um einen künstlichen Schlüssel handeln (*Has_ArtificialPrimaryKey*), bei dem die Werte der Schlüssel-Attribute durch das Anwendungs-System vergeben werden. Dieses Konzept ist nicht Bestandteil des relationalen Modells, dient jedoch später dazu, die Integration der KDBS zu erleichtern.

- Die Attribute einer Tabelle werden als Instanzen der Klasse *CExpColumn* beschrieben. Die Attribute dieser Klasse beschreiben deren Eigenschaften:

    - *COLNAME* := Name des Attributes.

    - *PKColumn* bestimmt, ob das Attribut Bestandteil des Primärschlüssels ist.

    - *Domain* bestimmt den atomaren Wertebereich (z.B. *Integer*, *Float*, *Varchar* etc.) des Attributes (*Get_Domain*).

    - *InitValue* ist der Initialisierungswert eines künstlichen Schlüssel-Attributes (*Is_ArtificialKeyColumn*).

    - Die *NOTNULL*-Klausel bestimmt, ob ein Tupel der Relation in diesem Attribut einen Wert haben muß (*Is_Required*) oder auch undefiniert sein kann.

- Das Konzept Fremdschlüssel wird durch die Klasse *CForeignKey* modelliert. Fremdschlüssel dienen dazu, Beziehungen zwischen Tupeln verschiedener Relationen auszudrücken. Abbildung 62 stellt ein Beispiel einer sogenannten „*Fremdschlüsssel-Beziehung*“ dar. Ein Fremdschlüssel besteht aus einer Menge von Attributen, deren Wertebereiche durch die aktuelle Ausprägung der Primärschlüssel-Attribute der „referenzierten“ Relation bestimmt sind (*referenzielle Integrität*).

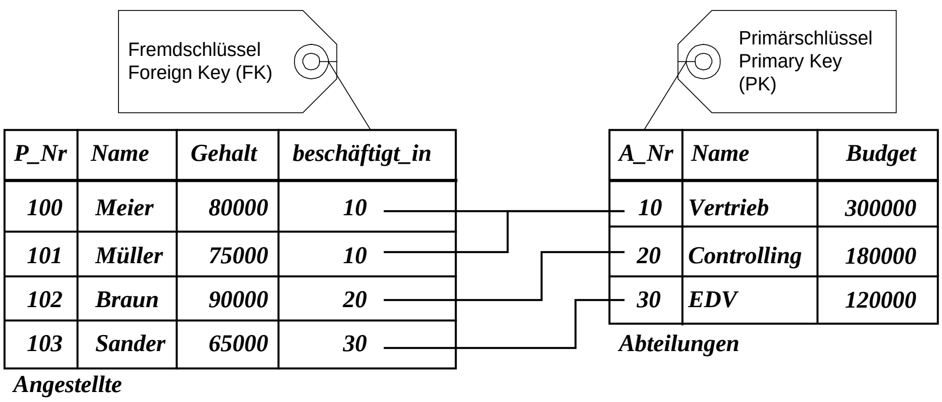


Abbildung 62 Beispiel einer Fremdschlüsssel-Beziehung

Im Beispiel sind mögliche Werte des Fremdschlüssel-Attributs *beschäftigt_in 10, 20* und *30*. Die Referenzierung von assozierten Tuppeln basiert dabei auf dem Wertevergleich der Fremdschlüssel- und Primärschlüssel-Attribute. Fremdschlüssel werden durch die Klasse *CForeignKey* folgendermaßen modelliert:

- Jeder Fremdschlüssel (FK) wird eindeutig durch einen Name (*FKCONSTRID*) bezeichnet.

- Die Aggregations-Beziehungen *SourceColumns* und *DestinationColumns* bestimmen die Mengen der Fremdschlüssel- bzw. Primärschlüssel-Attribute der referenzierten Tabelle. Sie werden hierbei als Listen verwaltet, um die Zuordnung der einzelnen Fremdschlüssel-Attribute zu den Primärschlüssel-Attributen aufrecht zu erhalten. Die Methoden *Get_SourceColumns*, *Get_Destination-Columns* und *Get_CorrelatingColumn* sind die hierzu korrespondierenden Lese-Operationen (Kapselungs-Prinzip).

- Die Relation, welche den Fremdschlüssel enthält, wird im folgenden als Quelle (*SourceTable*), die referenzierte Relation als Ziel (*DestinationTable*) bezeichnet. Diese lassen sich durch die Methoden *Get_SourceTable* und *Get_DestinationTable* bestimmen. Im obigen Beispiel sind dies die Tabellen *Angestellte* und *Abteilungen.*

- Ob ein Fremdschlüssel auch Bestandteil des Primärschlüssels der Relation ist, läßt sich mithilfe der Methode *Is_AlsoPrimaryKey* bestimmen.

- Die Methode *Get_JoinCond* liefert die zum Fremdschlüssel korespondierende Wertevergleichs-Bedingung. Als Parameter dienen hierbei Tupelvariablen *SrcTblVar* und *DestTblVar*, um auch komplexere Beziehungen bei der Schematransformation behandeln zu können. Für den Fremdschlüssel des obigen Beispiels würde die Methode *Get_JoinCond* als Ergebnis folgenden Ausdruck zurückliefern *(SrcTblVar.beschäftig_in = DestTblVar.A_Nr).*

Wie bereits im vorigen Abschnitt beschrieben wurde, werden die Meta-Daten des *objektorientierten Meta-Schemas* im relationalen KDBS persistent gespeichert. Abbildung 63 zeigt den zu Abbildung 61 korrespondierenden Teil des *relationalen Meta-Schemas*. Die Modellierung und Implementierung (Schema-Generierung) des *relationalen Meta-Schemas* erfolgte mithilfe des Entity-Relationship Entwurfswerkzeugs ERwin [ERw96]. Die durch das in Abbildung 63 dargestellte Schema beschriebenen *Meta-Daten* sind teilweise redundant zum Systemkatalog des KDBS. Hierfür gibt es mehrere Gründe:

- Das lokale Schema und somit auch der Systemkatalog des KDBS unterliegen Änderungen. Das *relationale Meta-Schema* soll jedoch den Zustand des lokalen Schema zum Zeitpunkt, in dem die Schema-Transformations-Abbildung definiert wurde, reflektieren. Beim Start eines Datenbank-Adapters werden das *relationale Meta-Schema* mit dem Systemkatalog des KDBS verglichen, um Änderungen des lokalen Schemas zu entdecken und durch die Administratoren des KDBS behandeln zu lassen.

- Da die Schema-Elemente des lokalen Schemas explizit im *relationalen Meta-Schema* beschrieben werden, ist es nun möglich, die Korrektheit einer Schema-Transformations-Abbildung durch die Definition von referenziellen Integritätsbedingungen [Date90] zu gewährleisten. Beispielsweise ist hierdurch sichergestellt, daß ein Objektyp des Komponenten-Schemas nur existierende Relationen referenzieren kann.

- Häufig werden Fremdschlüssel nicht explizit im konzeptionellen Schema spezifiziert (*ALTER TABLE T1 FOREIGN KEY (...) REFERENCE...*), sei es aus Nachlässigkeit oder um die Performance zu optimieren. Im relationalen Teilbereich des *Meta-Schemas* können nun solche Fremdschlüssel für die Schema-Transformation wieder explizit gemacht werden.

- Die Administratoren des KDBS können nun explizit die Menge der Relationen (des konzeptuellen Schemas) bestimmen, die bei der Schema-Transformation ins Komponenten-Schema berücksichtigt werden sollen (oder dürfen).

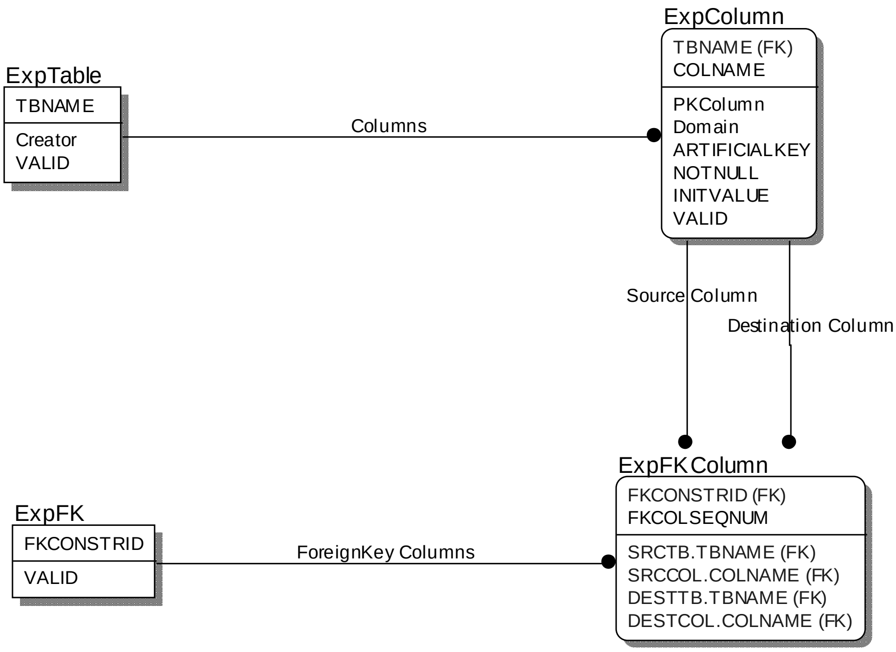


Abbildung 63 Abbildung der relationalen Konzepte im relationalen Meta-Schema

7.1.4 Abbildung der Konzepte des kanonischen Datenmodells (ODMG-93)

Die Abbildung der Konzepte des ODMG-93-Objektmodells durch ein objektorientiertes Meta-Schema wurde bereits ausführlich in Kapitel 6.3.2 beschrieben. Dieser Abschnitt stellt nun die unterschiedlichen Repräsentationen der Schema-Elemente eines Komponenten-Schemas („***two level store Ansatz***“) in Abbildung 64 und Abbildung 65 gegenüber.

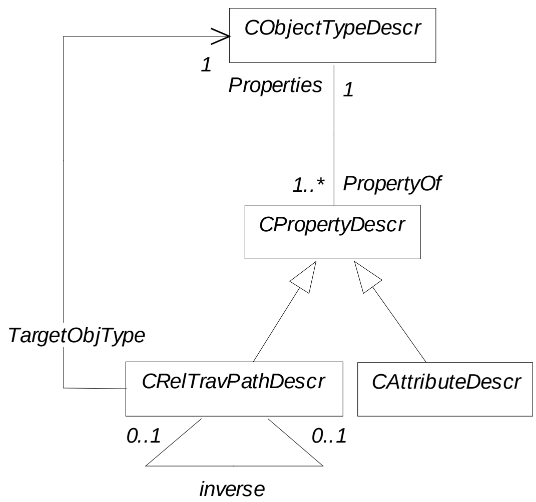


Abbildung 64 Abbildung der ODMG-93-Konzepte im objektorientierten Meta-Schema

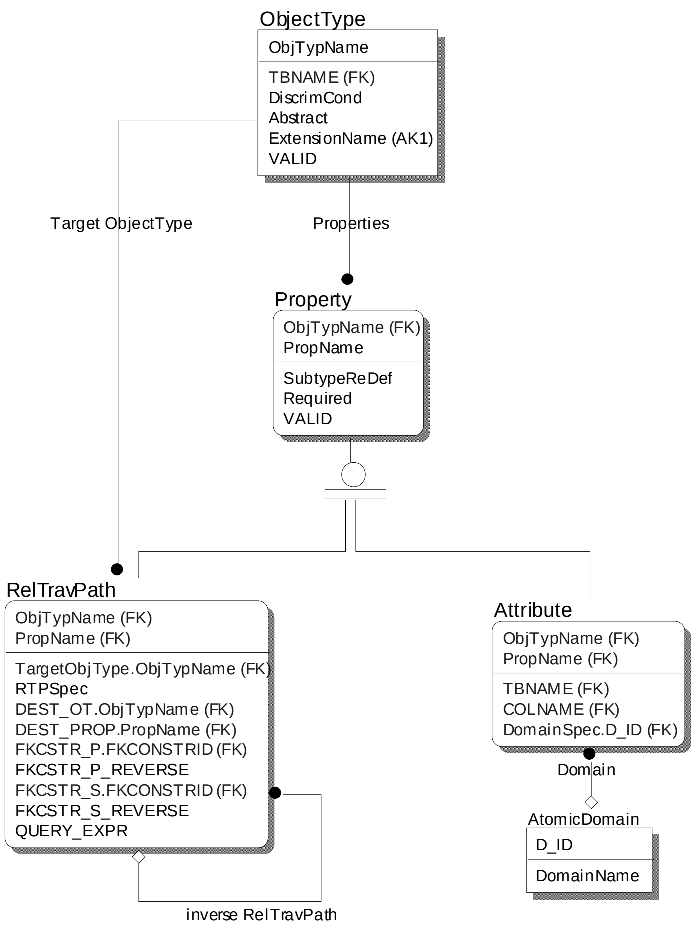


Abbildung 65 Abbildung der ODMG-93-Konzepte im relationalen Meta-Schema

Die Beziehungen zwischen den Schema-Elementen eines Komponenten werden, wie nicht anders zu erwarten war, im relationalen Meta-Schema durch Fremdschlüssel (FK) modelliert. Die Relation *AtomicDomain* beschreibt den Aufzählungstyp, der alle zulässigen ODMG-93 *built-in*-Datentypen enthält. Die eigentliche Schema-Transformations-Abbildung wird nun im nächsten Abschnitt dargestellt.

7.1.5 Die Schema-Transformations-Abbildung

Die Schema-Transformations-Abbildung soll mithilfe eines relationalen Beispiel-Schemas beschrieben werden. Abbildung 66 stellt dieses Beispiel-Schema in der Notation des Entity-Relationship-Entwurfswerkzeugs ERwin dar.

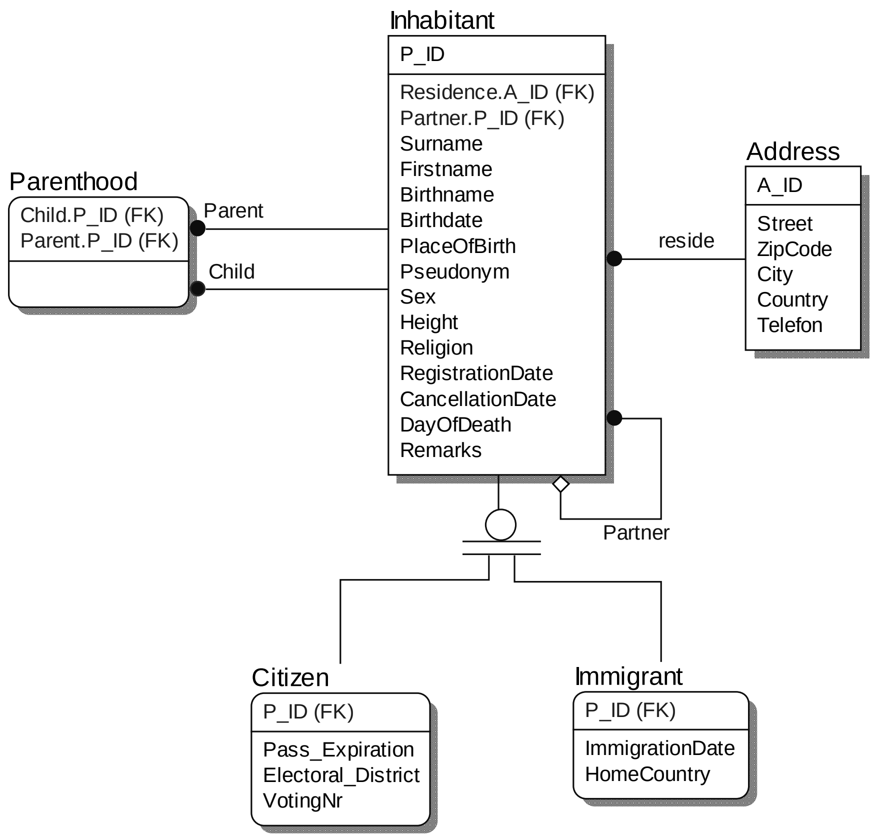


Abbildung 66 ER-Modell des konzeptuellen Schemas eines Einwohnermeldeamtes

Durch dieses Beispiel wird zugleich das konzeptuelle Schema eines Einwohnermeldeamtes des Anwendungs-Szenario (siehe Kapitel 5.1) eingeführt:

- Die Einwohner (*Inhabitant*) einer Stadt können sowohl wahlberechtigte Staatsbürger (*Citizen*) als auch Einwanderer (*Immigrant*) sein. Die Relation *Inhabitant* speichert die Daten, die für alle Einwohner verwaltet werden. In den Tabellen *Citizen* und *Immigrant* werden die speziellen Daten von wahlberechtigten Staatsbürgern und Einwanderern verwaltet. Der Primärschlüssel (*P_ID*) dieser Tabellen ist zugleich Fremdschlüssel auf die Tabelle *Inhabitant*. Diese Art der Repräsentation von Vererbungsbeziehungen wird als *vertikaler Ansatz* bezeichnet [HoPl95, RaHo96].

- Zu jedem Einwohner wird dessen Ehegatte (*Partner*) verwaltet. Das Fremdschlüssel-Attribut *Partner* in der Relation *Inhabitant* modelliert diese *1:1-Beziehung.*

- Die Eltern-Kind-Beziehung (*n:m-Beziehung*) zwischen den Einwohnern wird in der Tabelle *Parenthood* verwaltet.

- Die Relation *Address* speichert alle Adressen der Einwohner der Stadt und steht in *1:n-Beziehung* (*reside*) zur Relation *Inhabitant.* Hierbei ist *Residence* das Fremdschlüssel-Attribut dieser Beziehung.

Im folgenden wird nun die konkrete Abbildung des obigen relationalen Schemas auf ein Komponenten-Schema dargestellt.

7.1.5.1 Die Abbildung von Relationen und impliziten Vererbungsbeziehungen

Alle Relationen des relationalen Schemas werden daraufhin untersucht, ob ihre Tupel eigenständige Objekte darstellen. Die Relationen, die lediglich „relationale“ Beziehungen repräsentieren, wie z.B. die Relation *Parenthood* im obigen Beispiel, erfüllen diese Eigenschaft also nicht. Für diejenigen Relationen, die obige Eigenschaft erfüllen, ist nun zu prüfen, ob sie nicht verschiedene Arten von Objekten mit gemeinsamen Eigenschaften repräsentieren. An diesem Punkt angelangt, muß nun das Problem der impliziten Vererbungsbeziehungen gelöst werden. Allgemein werden in der Literatur drei unterschiedliche Ansätze zur Repräsentation von Vererbungsbeziehungen in relationalen Schemata unterschieden [HoKo95, HoPl95, Ontos96, RaHo96]:

- Die am häufigsten verwendete Methode ist der *vertikale Ansatz*. Hierbei werden die gemeinsamen Eigenschaften der Subtypen in einer Relation, welche den Supertypen repräsentiert, gespeichert. Die speziellen Eigenschaften der Subtypen werden jeweils in einer eigenen Relation gespeichert. Der Primärschlüssel dieser „speziellen“ Relationen ist dabei gleichzeitig Fremdschlüssel auf die Relation des Supertypen. Entspricht dem Supertypen kein Objekt der Diskurswelt, so wird dieser als abstrakter Supertyp bezeichnet. Dies ist im obigen Einwohnermeldeamt-Schema der Fall. Bei der Analyse der „*vertikalen“* Vererbungsbeziehungen ist es notwendig, die möglichen „Extensionen“ (Ausprägungen) der potentiellen Supertyp- und Subtypen-Relationen zu untersuchen. Diese müssen folgende Inklusions-Bedingung erfüllen:

  $$\forall\ \mathbf{i, j} \in \{1,..,\mathbf{n}\} : \mathbf{(}\ \pi_{\mathbf{PK}}(\mathbf{Super}) \supseteq \pi_{\mathbf{PK}}(\mathbf{Sub_i})\ )\ \wedge\ (\ \pi_{\mathbf{PK}}(\mathbf{Sub_i}) \cap \pi_{\mathbf{PK}}(\mathbf{Sub_j}) = \emptyset\ )$$

  wobei **PK** den Primärschlüssel der Relationen **Super** und $\mathbf{Sub_i}$ bezeichnet und $\pi_{\mathbf{PK}}$ die Projektion auf die Primärschlüssel-Attribute darstellt**.** Folgende, in Abbildung 67 beispielhaft dargestellte Relationen, würden diese Bedingung nicht erfüllen, wenn ein Kunde gleichzeitig auch Lieferant und umgekehrt sein könnte ($\pi_{G\_ID}(\mathbf{Kunde}) \cap \pi_{G\_ID}(\mathbf{Lieferant}) \neq \emptyset$).

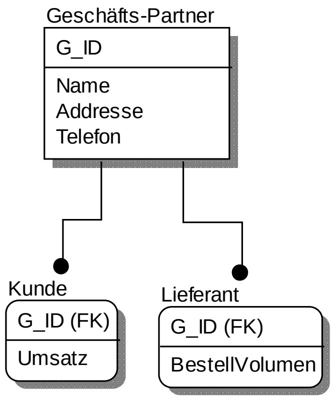


Abbildung 67 Beispiel einer vermeintlichen Vererbungsbeziehung

Dies kann man dem relationalen Schema allein jedoch nicht ansehen, vielmehr muß bei der Analyse eines lokalen Schemas zusätzliches semantisches Wissen über die Diskurswelt hinzugezogen werden.

- Werden alle Eigenschaften der Subtypen (und des Supertyps) jeweils vollständig in verschiedenen Relationen gespeichert, so spricht man vom *horizontalen Ansatz*. Der Zugriff auf die Tupel in der „Extension“ des Supertypen erfolgt durch mehrere SQL-Teil-Anfragen, welche durch das SQL-Konstrukt *Union* zu einer DB-Anfrage veknüpft werden.

- Werden die Tupel der Subtypen (und des Supertyps) in einer einzigen Relation gespeichert und ist die Typ-Zugehörigkeit dabei in einem oder mehreren Attributen kodiert, so wurde der *Discriminator-Ansatz* oder *Flag-Ansatz* angewandt. Beispielweise könnte es im obigen Beispiel von Interesse sein, die männlichen (Attribut *Sex* = TRUE) und die weiblichen Einwohner (Attribut *Sex* = FALSE) zu unterscheiden und durch unterschiedliche Objekttypen abzubilden.

Wurden die Vererbungsbeziehungen im relationalen Schemata nun in Hinblick auf die oben dargestellten Ansätze analysiert, so wird folgende Abbildung vorgenommen:

- Jeder identifizierte Objekttyp, inclusive der abstrakten Supertypen, wird auf eine Instanz der Klasse *CObjectTypDescr* (siehe Abbildung 68) abgebildet. Werden die Eigenschaften der DB-Objekte eines Objekttypen in einer Relation gespeichert, so hält dieser eine Referenz (*PrimarySourceTable*) auf die Tabelle (Instanz von *CExpTable*), die seine „speziellsten“ Eigenschaften speichert. Beim *horizontalen Ansatz* wird ein abstrakter Supertyp durch keine Tabelle repräsentiert. In diesem Fall hält der abstrakte Supertyp auch keine Referenz auf eine Tabelle. Im obigen Beispiel lassen sich die Objekttypen *Inhabitant*, *Citizen*, *Immingrant* und *Address* identifizieren. Zu jedem dieser Objekttypen gibt es genau eine korrespondierende Relation (*vertikaler Ansatz*).

- Bei dieser Abbildung können unter Umständen einer Tabelle mehrere Objekttypen zugeordnet werden (*Discriminator-Ansatz*). In diesem Fall definiert der Wert des Attributs *DiscrimCond* der Klasse *CObjectTypDescr* die *Discriminator-Bedingung* zur Selektion der speziellen Objekte.

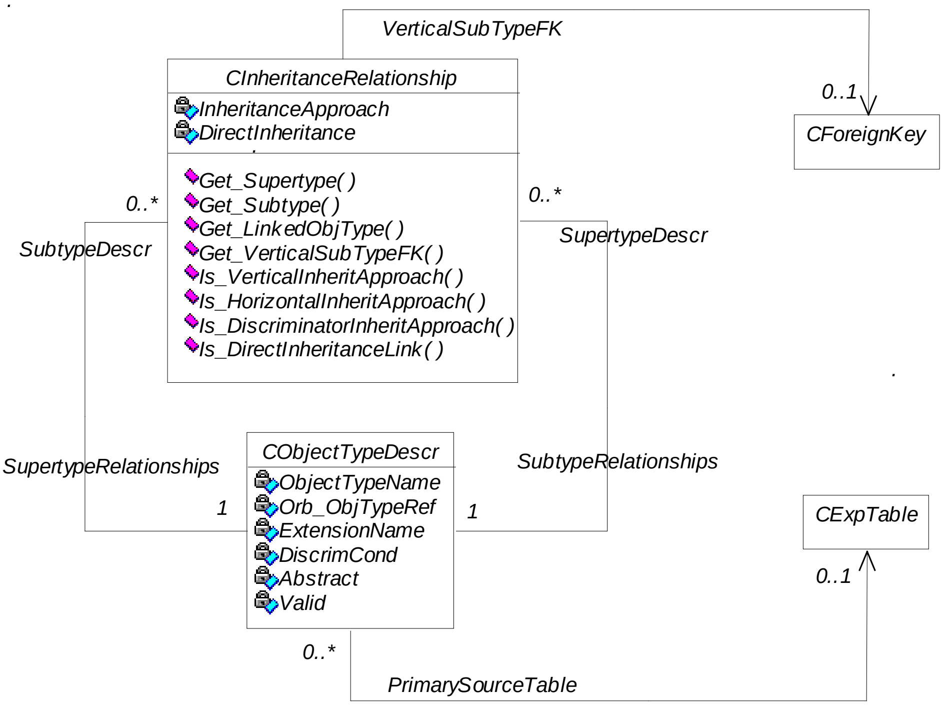


Abbildung 68 Die Abbildung von Relationen auf Objekttypen

- Die Vererbungsbeziehungen zwischen den identifizierten Objekttypen werden als Instanzen der Klasse *CInheritanceRelationship* repräsentiert. Das Attribut *InheritanceApproach* dieser Klasse (Wertebereich *{vertical, horizontal, discriminator}*) markiert dabei die Kanten in der Vererbungs-Hierarchie mit dem angewendeten Ansatz. Wurde der vertikale Ansatz angewandt, so referenziert die Vererbungsbeziehung (Instanz von *CInheritanceRelationship*) den dazu korrespondierenden Fremdschlüssel (*VerticalSubTypeFK*).

- Um auch Hierarchien zu unterstützen, bei denen die obigen drei Ansätze beliebig frei kombiniert wurden, ist es notwendig, zusätzliche Vererbungs-Kanten einzuführen. Ein Beispiel, in Abbildung 69 dargestellt, soll dieses Problem verdeutlichen. Für den Objekttypen *Manager* und seine Subtypen ist nicht eindeutig festgelegt, ob die indirekt geerbten Eigenschaften des Objekttyps *Mitarbeiter* in der zum Objekttypen *Manager* korrespondierenden Relation gespeichert werden, da der Objekttyp *Manager* durch den *horizontalen Ansatz* von seinem direkten Supertyp *Angestellter* abgeleitet wurde. Aus diesem Grund werden die indirekten Vererbungsbeziehungen, bei denen der *vertikale Ansatz* angewandt wurde, dann angegeben, wenn ein Objekttyp von seinem direkten Vorgänger in der Vererbungs-Hierarchie durch den *horizontalen Ansatz* abgleitet wurde. Durch das Attribut *DirectInheri-*

tance der Klasse *CInheritanceRelationship* lassen sich die direkten von den indirekten Vererbungsbeziehungen unterscheiden.

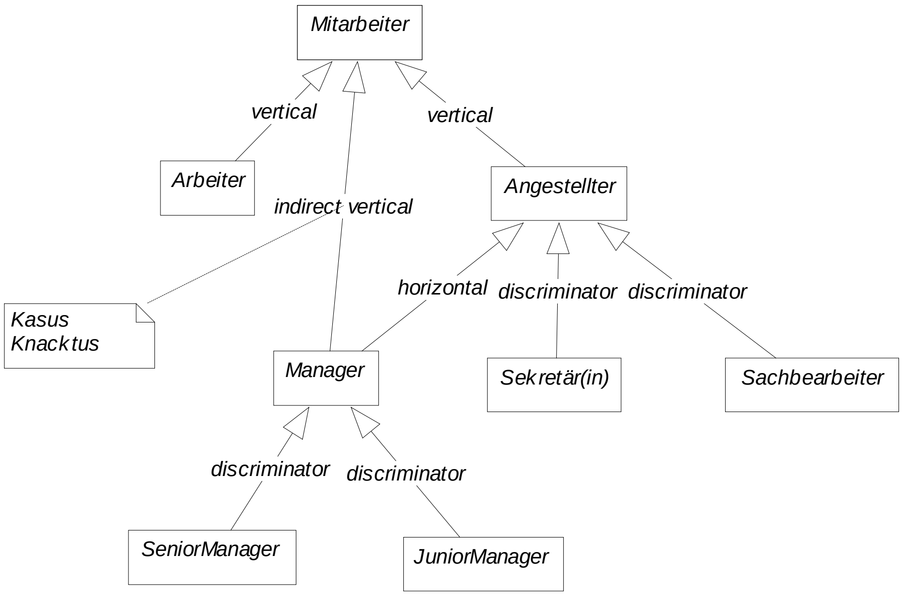


Abbildung 69 Kombinierter Einsatz der relationalen Vererbungsansätze

7.1.5.2 Die Abbildung von Attributen und impliziten Beziehungen

Die Abbildung der Attribute und impliziten Beziehungen des lokalen (relationalen) Schemas wird nun folgendermaßen durchgeführt. Jede Eigenschaft (Attribut oder Beziehung) wird auf eine Instanz der Klasse *CPropertyDescr* abgebildet (siehe Abbildung 70). Hierbei ist die Vererbungs-Hierarchie zu berücksichtigen. Entspricht einer Eigenschaft, die ein Subtyp von einem Supertypen erbt, ein anderes lokales Schema-Element als das des Supertypen, so wird diese Eigenschaft redefiniert. Eine zusätzliche Instanz der Klasse *CPropertyDescr* (Attribut *SubTypeReDef* = TRUE) repräsentiert diese Redefinition. Hierbei müssen die Wertebereiche (D_1 und D_2) der Eigenschaft und ihrer Redefinition kompatibel sein ($D_1 \supseteq D_2$). Redefinitionen sind eine Folge des *horizontalen Vererbungs-Ansatzes* (siehe Abschnitt 7.1.5.1).

Die Abbildung von Attributen

Bei der Abbildung der Attribute der Relationen des lokalen Schema (Instanzen von *CExpColumn*) ist nun für jedes Attribut zu entscheiden, ob es für die Diskurswelt relevante Attribut-Werte repräsentiert oder lediglich ein künstliches Konzept, als Folge des relationalen Datenmodells, darstellt. Die Attribute, die nicht Bestandteil eines *Primärschlüssels*, eines *Fremdschlüssels* oder eines *Discriminators* sind, haben in den meisten Fällen eine relevante Semantik. Für alle übrigen Attribute ist jeweils der konkrete Einzelfall zu betrachten. Diejenigen Attribute, die als relevant identifiziert wurden, werden auf Attribute (Instanzen der Klasse *CAttributeDescr*) des entsprechenden Objekttyps abgebildet (*BasedOnColumn*). In Folge der unterschiedlichen *Vererbungs-Ansätze* kann hierbei ein relationales Attribut unter Umständen auf mehrere Attribute des Komponenten-Schemas abgebildet werden (*Discriminator-Ansatz*). Wurde der *horizontalen Ansatz* angewandt, so enspricht einem Attribut eines abstrakten Supertypen des Komponenten-Schemas kein relationales Attribut. Bei der Abbildung eines relationalen Attributes mit dem Wertebereich D_{rel} ist zu beachten, daß der Wertebereich des Komponenten-Schema-Attributes $\mathrm{D}_{\mathrm{ODMG\text{-}93}}$ den Wertebereich D_{rel} zumindest einbetten kann ($D_{ODMG\text{-}93} \supset D_{rel}$).

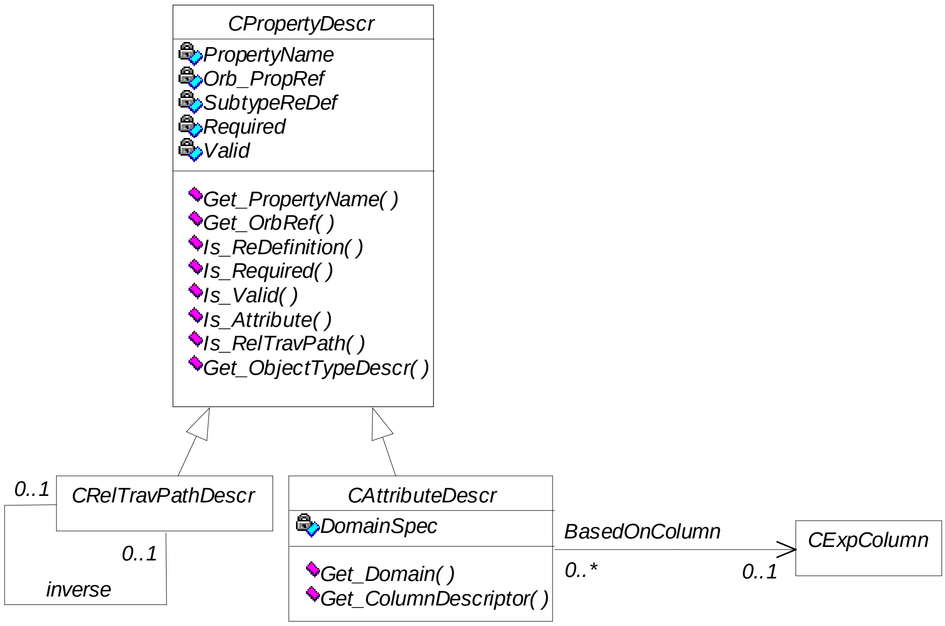


Abbildung 70 Die Abbildung von Attributen bei der Schema-Transformation

Die Abbildung von Beziehungen

Ziel der Schema-Transformations-Abbildung ist es, die *impliziten* Beziehungen zwischen den identifizierten Objekttypen (siehe Abschnitt 7.1.5.1) wieder explizit zu machen. Diese werden unterschiedlich, je nach Art der Beziehung (*1:1, 1:n, n:m*), mithilfe von Fremdschlüsseln (und zusätzlichen Relationen) repräsentiert.

Wie bereits im Abschnitt 7.1.3 deutlich wurde, hat ein Fremdschlüssel eine implizite Richtung (*SourceTable, DestinationTable*). Diese Eigenschaft soll nun bei der Ableitung der Beziehungen (*Relationship-Traversal-Pathes*) des Komponenten-Schemas ausgenutzt werden. Zunächst werden die *„eigentlichen“* Beziehungen des relationalen Schemas zusammen mit den daran beteiligten Schema-Elementen identifiziert. Die Abbildung auf Beziehungen des Komponenten-Schemas erfolgt entsprechend den Kardinalitäten der „*relationalen*“ Beziehungen:

- An der Repräsentation einer *1:n-Beziehung* sind ein Fremdschlüssel und zumeist zwei Relationen beteiligt. Solch eine Beziehung wird auf zwei zueinander inverse Zugriffs-Pfade (Instanzen von *CRelTravPath*, siehe Abbildung 71) abgebildet. Beide Zugriffs-Pfade halten Referenzen auf den korrespondierenden Fremdschlüssel (*PrimaryFK*), den Ziel-Objekttypen (*TargetObjType*) und den inversen Zugriffs-Pfad (*inverse*). Hierbei unterstützt der eine Zugriffs-Pfad den navigierenden Zugriff in der Richtung des Fremdschlüssels (Instanz-Attribut *PrimaryFKReverseDirection* = FALSE), der andere Zugriffs-Pfad die Navigation in Gegenrichtung (Instanz-Attribut *PrimaryFKReverseDirection* = TRUE). Die Angabe der Richtung ist hierbei unbedingt notwendig, um auch *1:n-Beziehungen* zwischen Tupeln (in verschiedenen Rollen) einer Relation zu unterstützen. Ein Beispiel hierfür wäre eine Relation *Bauteil*, deren Tupel in einer *Baugruppe-Unterteil-Beziehung* zueinander stehen.

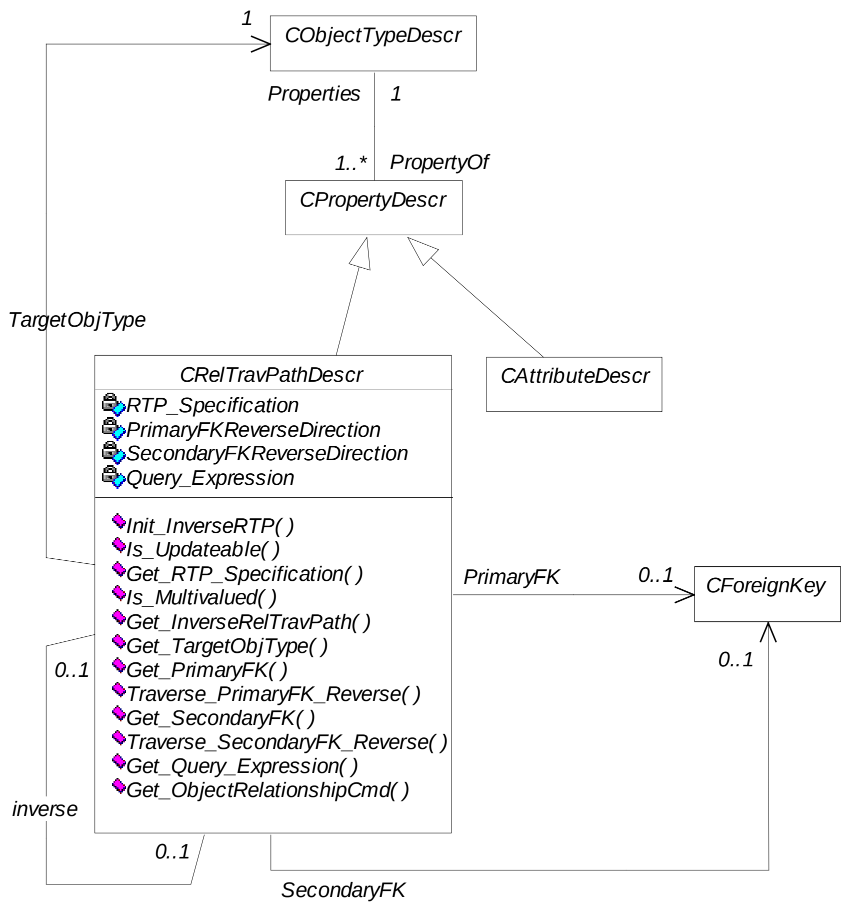


Abbildung 71 Die Abbildung von „relationalen“ Beziehungen bei der Schema-Transformation

- Eine *1:1-Beziehung* ist ein Spezialfall einer *1:n-Beziehung* und kann analog zu der oben dargestellten Vorgehensweise abgebildet werden.

- Eine *n:m-Beziehungen* wird mithilfe einer zusätzlichen Relation *R* dargestellt. Der Primär-Schlüssel dieser Relation besteht dabei aus zwei Fremdschlüsseln ($PK_R = \{ FK_1 , FK_2 \}$), welche die in Beziehung stehenden Tupel referenzieren. Eine *n:m-Beziehung* wird nun wie folgt auf zwei zueinander inverse Zugriffs-Pfade (*CRelTravPathDescr*) abgebildet. Jeder Zugriffs-Pfad verwaltet ein geordnetes Paar von Referenzen *P* (*<u>PrimaryFK</u>*, *SecondaryFK*) auf die beiden Fremdschlüssel, welche die Beziehung repräsentieren. Sei RTP_1 der Zugriffs-Pfad eines Objekttypen OT_1 auf die DB-Objekte eines Objekttyps OT_2 und R_1 , R_2 die dazu korrespondierenden Relationen, dann beschreibt das Paar $P_1 = (\underline{FK_1} , FK_2)$ den Pfad, entlang der Fremdschlüssel von R_1 nach R_2, für RTP_1, wobei der erste Fremdschlüssel in Gegenrichtung zu durchlaufen ist (*PrimaryFKReverseDirection* = TRUE). Analog gilt dies für den Zugriffs-Pfad $RTP_2 : P_2 = (\underline{FK_2} , FK_1)$.

In Abbildung 72 ist die Ableitung der Eltern-Kind-Beziehung (*n:m-Beziehung*) des Einwohnermeldeamt-Schemas als Instanz-Diagramm beispielhaft dargestellt.

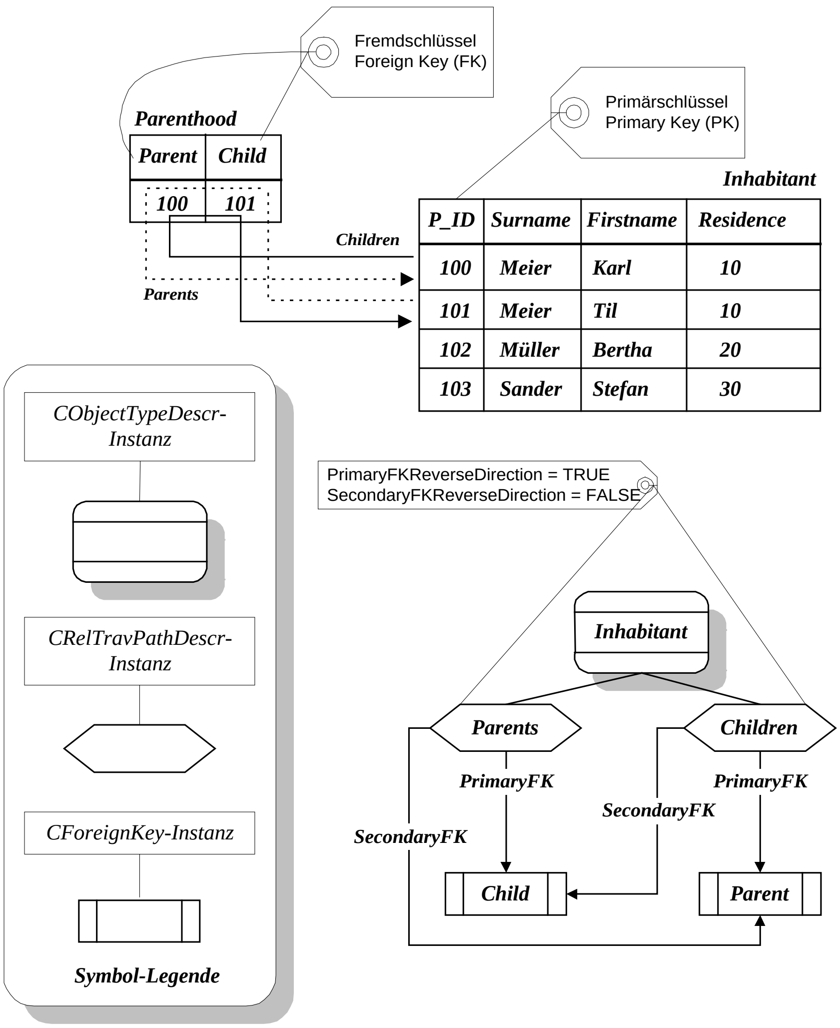


Abbildung 72 Die Abbildung einer n:m-Beziehung

Die bisher dargestellten Beziehungen eines Komponenten-Schemas verwalten die korrespondierenden Fremdschlüssel des relationalen Schemas und sind daher update-fähig (*Is_Updateable*, siehe Abbildung 71). In manchen Situationen kann es jedoch auch sinnvoll sein, Beziehungen im Komponenten-Schema zu definieren, die auf einem DB-Anfrage-Ausdruck *Query_Expression* basieren und deshalb nicht update-fähig sind. Ein Beispiel für eine solche Beziehung wird im nächsten Abschnitt dargestellt.

Den Abschluß des Abschnitts 7.1 bildet die graphische Darstellung des vollständigen *objektorientierten Meta-Schemas* (Abbildung 73) der Schema-Transformations-Abbildung. Das hierzu korrespondierende *relationale Meta-Schema* zur persistenten Speicherung der *Meta-Daten* ist in Abbildung 74 dargestellt (vergleiche hierzu Abschnitt 7.1.2).

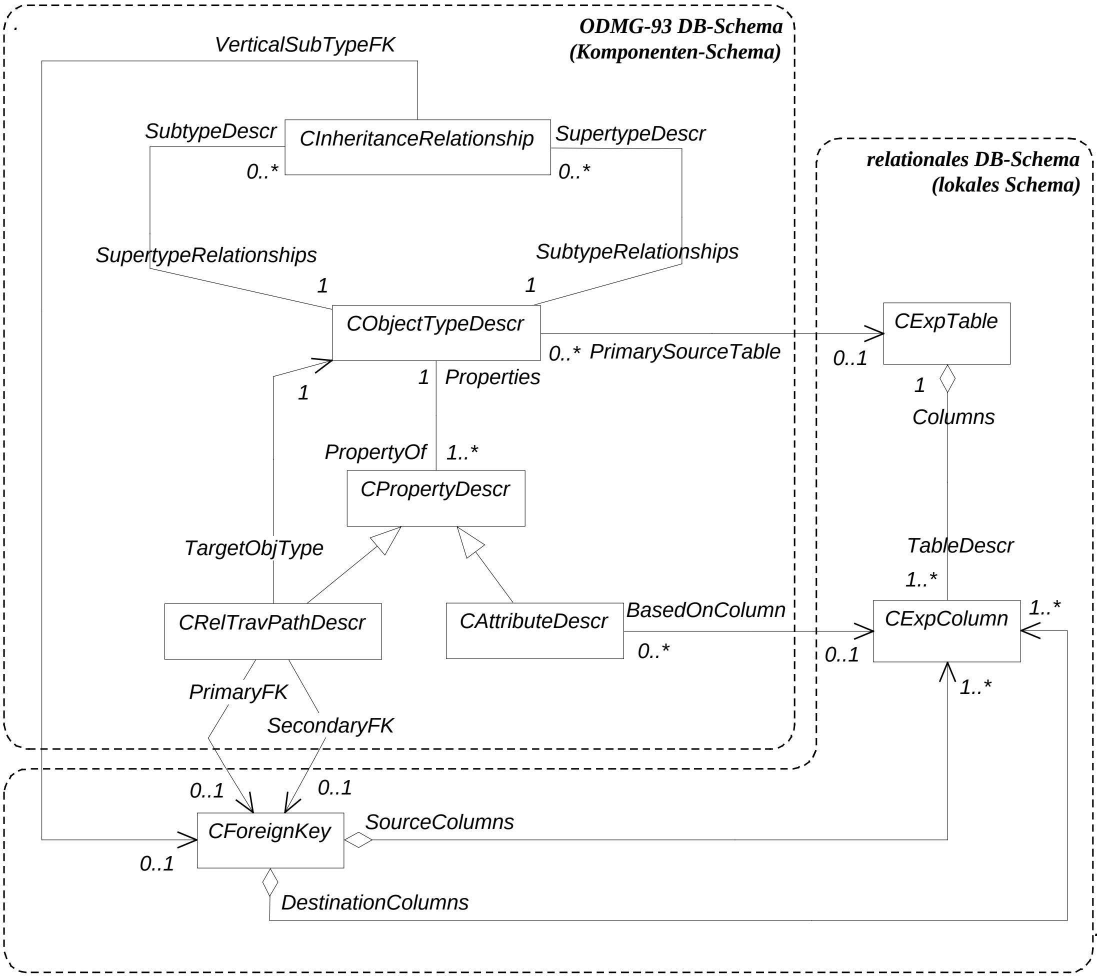


Abbildung 73 Das objektorientierte Meta-Schema

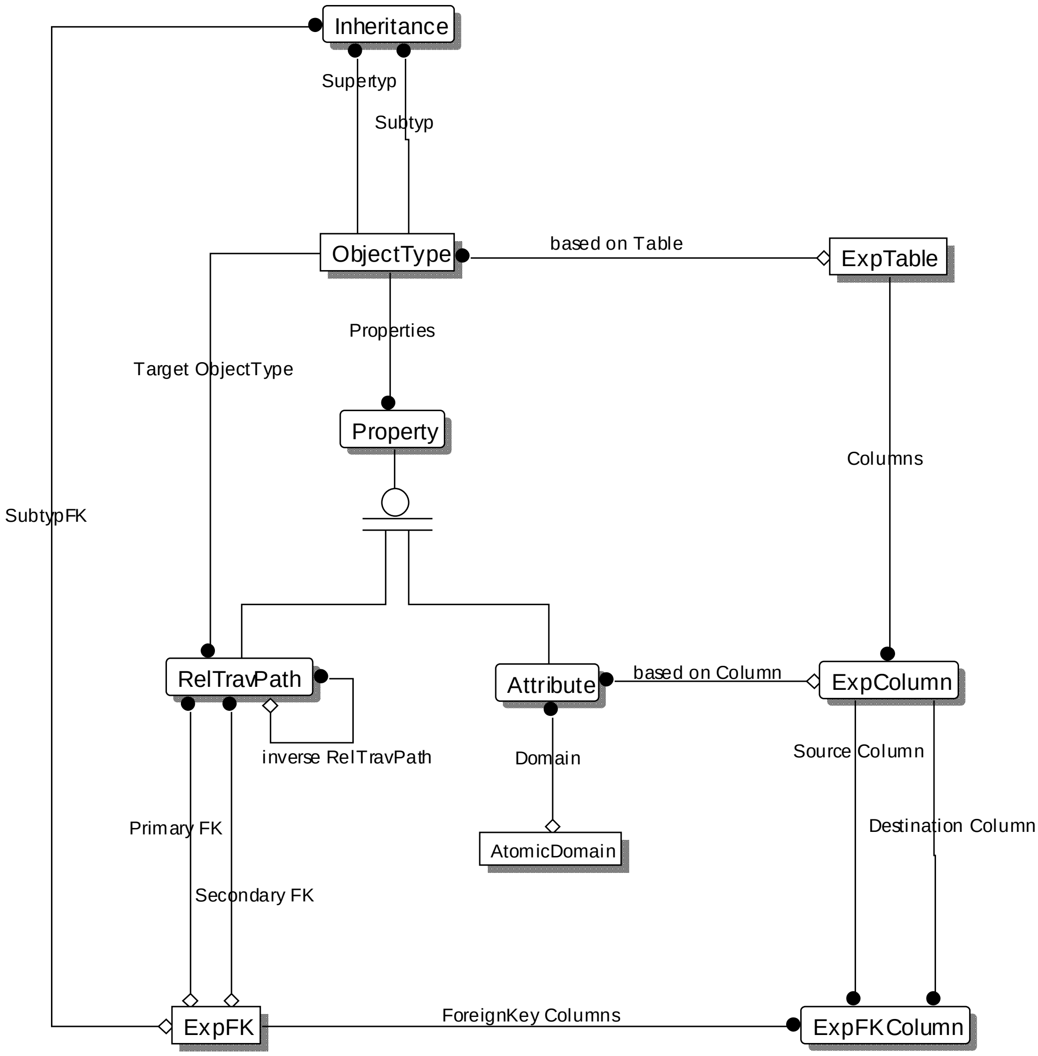


Abbildung 74 Das relationale Meta-Schema

7.2 Die Realisierung der lokalen Objekt-Identität

Wie bereits im Kapitel 6.5.1 dargestellt wurde, ist es die Aufgabe des Datenbank-Adapters eine wertunabhängige und dauerhafte Objekt-Identität der DB-Objekte des Komponenten-Schemas zu unterstützen. Dieses Konzept wurde in Kapitel 6.5.1 als *strenge lokale Objekt-Identität* [EK91] bezeichnet.

Um eine *strenge lokale Objekt-Identität* zu realisieren, ist es notwendig das Identifizierungs-Konzept (*KDBMS-abhängige Objekt-Identität*) des relationalen Komponenten-Datenbanksystems zu erweitern. Hierbei muß die im Abschnitt 7.1 dargestellte Schema-Transformations-Abbildung berücksichtigt werden, da die DB-Objekte des Komponenten-Schemas Gegenstand der *Objekt-Identität* sind. Zur Realisierung der *lokalen Objekt-Identität* wird folgender Ansatz verfolgt:

- In einer zusätzlichen Relation *Extension* (siehe Abbildung 75) wird zu jedem DB-Objekt des Komponenten-Schemas ein dauerhafter und wertunabhängiger *lokaler Object-Identifier* (*LOID*) verwaltet. Diese Relation hält einen Fremdschlüssel auf die Relation *ObjectType*, in der alle Objekttypen des Komponenten-Schemas gespeichert sind. Hierdurch ist es möglich, die zum Objekttypen korrespondierende Tabelle (repräsentiert in *ExpTable*) zu identifizieren (vergleiche hierzu Abschnitt 7.1.5.1). Das Attribut *RegisteredByFDBS* vermerkt zu jedem DB-Objekt, ob der FDBS-Kern schon über dessen Existenz benachrichtigt wurde (siehe Abschnitt 7.3).

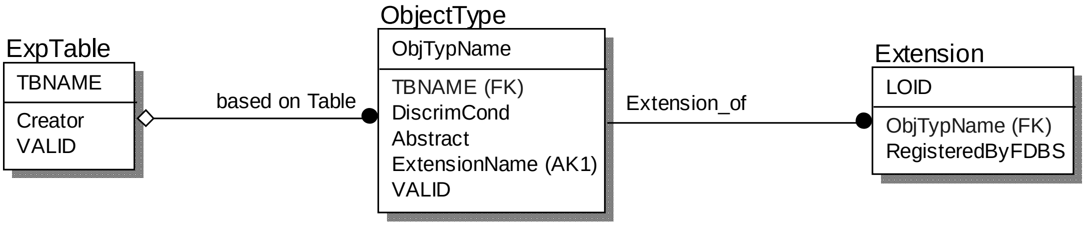
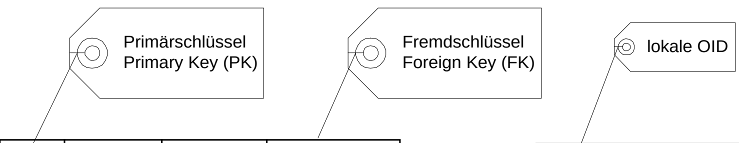


Abbildung 75 Die Verwaltung der Objekt-Identität

- Zusätzlich wird jede Tabelle, die zu einem Objekttypen korrespondiert, um den Fremdschlüssel (*LOID*) auf die Relation *Extension* erweitert (siehe Abbildung 76). Hierdurch ist es nun möglich, jedes DB-Objekt dauerhaft und wertunabhängig zu identifizieren, da eine Wertänderung des Primärschlüssels keinen Einfluß auf diese Abbildung hat. Das zusätzliche Fremdschlüssel-Attribut *LOID* wird auch als *surrogate-key* bezeichnet [Kroh93]. Dieser Ansatz ist zwar eine Einschränkung der *Entwurfsautonomie* der KDBS (vergleiche Kapitel 4.2), stellt jedoch keine Einschränkung für die lokalen Applikation des KDBS dar (logische Datenunabhängigkeit [Date90]).



P_ID	***Surname***	***Firstname***	***LOID***
100	*Meier*	*Karl*	*1000*
101	*Meier*	*Til*	*1001*
102	*Müller*	*Bertha*	*1005*
103	*Sander*	*Stefan*	*1010*

Citizen

LOID	***ObjTypname***	***Reg.FDBS***
1000	*Citizen*	*Y*
1001	*Citizen*	*Y*
1005	*Citizen*	*Y*
1010	*Citizen*	*Y*
1011	*Address*	*Y*
1200	*Address*	*Y*
1442	*Immigrant*	*Y*

Extension

Abbildung 76 Beispiel-Ausprägung im Einwohnermeldeamts-Datenbank

- Die Verwaltung der *lokalen Object-Identifier* (*LOID*) erfolgt mithilfe von *Triggern* und *Stored Procedures* [Cent3-96, Froese95]. Ein Trigger (auf einer Relation) ist definiert durch ein Ereignis, welches durch eine Änderungs-Operation ausgelöst wird (*Insert, Update, Delete*), den Zeitpunkt der Feuerung (vor oder nach der Operation) und die Angabe der Aktion, welche das Ereignis behandelt. Die Aktion wird in Form einer *Stored Procedure* definiert. Für jede Relation, die einem Objekttypen entspricht, werden drei *Trigger* definiert, die jeweils nach dem Einfügen, Ändern oder Löschen eines Tupels feuern, wobei für die Verwaltung der Objekt-Identität nur die *Insert-* und *Delete-Trigger* von Bedeutung sind. Der *Update-Trigger* dient zur Entdeckung von Zustands-Änderungen (siehe Abschnitt 7.3.1). Das Einfügen eines Tupels wird durch die *Stored Procedure „RegisterObj"* behandelt. Diese sorgt dafür, daß dem DB-Objekt eine neue *LOID* zugewiesen wird, indem sie ein entsprechendes Tupel in der Relation *Extension* anlegt und den Fremdschlüssel (*LOID*) des eingefügten Tupels entsprechend anpaßt (siehe Abbildung 76). Hierbei wird die Vererbungs-Hierachie des Komponenten-Schemas berücksichtig (vergleiche hierzu Abschnitt 7.1.5.1). Das Löschen eines Tupels wird durch die *Stored Procedure „ObjDelete"* behandelt (näheres hierzu in Abschnitt 7.3).

7.3 Der lokale Event-Handler

Der lokale Event-Handler hat die Aufgabe, lokale Ereignisse, die infolge von Änderungs-Operationen innerhalb lokaler Transaktionen auftreten, zu entdecken und den FDBS-Kern hierüber zu informieren. Die Interaktion des lokalen Event-Handlers mit seinem globalen Pendant im FDBS-Kern wurde bereits hinreichend in Kapitel 6.4 darstellt. In diesem Abschnitt sollen nun lediglich einige Aspekte der Realisierung des lokalen Event-Handlers beschrieben werden.

7.3.1 Die Entdeckung von Ereignissen

Wie bereits im letzten Abschnitt dargestellt wurde, erfolgt die Entdeckung von Ereignissen mithilfe von *Triggern*. Zusätzlich zur Aufrechterhalten der Objekt-Identitäts-Abbildung haben die *Trigger* bzw. die von ihnen aufgerufenen *Stored Procedures* die Aufgabe, alle auftretenden Ereignisse zu protokollieren. Die aufgetretenden Ereignisse werden hierbei in der zusätzlichen Relation *Obj_Event* vermerkt (siehe Abbildung 77).

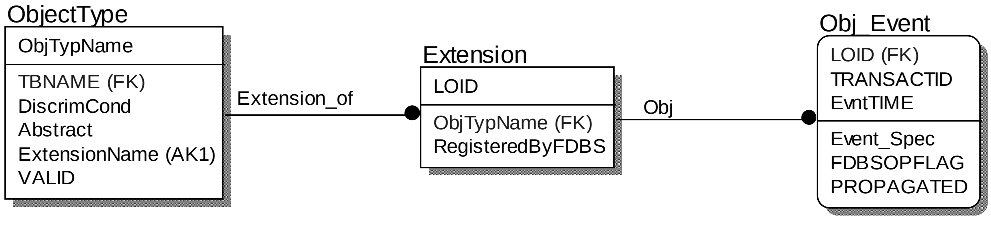


Abbildung 77 Die Verwaltung von Ereignissen

Ein Ereignis wird hierbei folgendermaßen protokolliert:

- *LOID* identifiziert das vom Ereignis betroffene DB-Objekt eindeutig.

- Zu jedem Ereignis wird die ID der verursachenden Transaktion (*TRANSACTID*) [Cent396] vermerkt. Hierdurch ist es möglich, mehrere Einfüge- oder Lösch-Operationen bezogen auf ein DB-Objekte innerhalb **einer** lokalen Transaktion zusammenfassend zu behandeln. Dies gilt insbesondere in Hinblick auf die *vertikalen Vererbungsbziehungen* (siehe Abschnitt 7.1.5.1). Um beispielsweise ein DB-Objekt des *vertikalen* Subtypen *Citizen* des Einwohnermeldeamts-Schemas (vergleiche hierzu Abbildung 66 im Abschnitt 7.1 auf Seite 101) anzulegen, muß zunächst (wegen der referenziellen Integrität) ein Tupel in die Supertyp-Relation *Inhabitant* eingefügt werden, bevor schließlich das *Citizen*-Tupel eingefügt werden kann. Hierbei existiert also zwischenzeitlich ein *Inhabitant*-Objekt. Werden jedoch beide Einfüge-Operationen innerhalb **einer** Transaktion (durch *TRANSACTID* bestimmbar) vorgenommen, so werden diese auf **ein** Ereignis abgebildet.

- *EvntTime* enthält die Systemzeit beim Auftreten des Ereignisses und dient gleichzeitig als Zeitstempel des lokalen DB-Objektes.

- *Event_Spec* $\in$ { IO, UO, DO } spezifiziert die Art des Ereignisses wird vom *Insert-, Update-* oder *Delete-Trigger* entsprechend gesetzt.

- Im Attribut *FDBSOPFLAG* wird vermerkt, ob das Ereignis durch einen lokalen Benutzer des Komponenten-Datenbanksystems oder durch den Datenbank-Adapter, als Stellvertreter des FDBS, ausgelöst wurde. Der Datenbank-Adapter wird dabei durch eine spezielle User-Kennung „*FDAGENT*“ identifiziert. Ereignisse, die vom FDBS initiiert wurden, werden vom lokalen Event-Handler nicht an den Event-Handler des FDBS-Kerns propagiert. Wäre dies nicht der Fall, so würden infolge des Replikations-Mechanismus (siehe Kapitel 6.6) endlose Ketten von Ereignissen der unterschiedlichen Replikate eines DB-Objektes entstehen (bzw. ein *Deadlock* verursacht).

- Das Attribut *Propagated* spezifiziert den Status der Bearbeitung eines Ereignisses entsprechend des in Kapitel 6.4 dargestellten Interaktions-Protokolls zwischen dem lokalen und dem globalen Event-Handler. Wurde ein Ereignis vom globalen Event-Handler vollständig bearbeitet (*Event_Is_Processed*), so wird der Status-Attribut *Propagated* entsprechend auf TRUE gesetzt.

7.3.2 Die Berarbeitung von Ereignissen

Die in der Relation *Obj_Event* protokollierten Ereignisse werden vom „eigentlichen“ Event-Handler in zyklischen Abständen aus der Datenbank gelesen und dann weiterbearbeitet. Der Event-Handler wird als einzige Instanz (*Singleton*) der Klasse *CDBEventHandler* realisiert (siehe Abbildung 78) und erbt von der Klasse *COOL_TimerCallback*, welche Bestandteil der Chorus-Cool-ORB-Implementierung [Cho296] ist, das Konzept einer abstrakten „Zeitschaltuhr“. Der Event-Handler wird vom ORB in zyklischen Abständen über den Ablauf eines Zeit-Intervalls benachrichtigt (virtuelle Methode *notifyTimer*). Das Zeit-Intervall wird hierbei durch die Methode *setTimeout* definiert, nachdem der Event-Handler vom Datenbank-Adapter gestarted wurde (*Start_EventProcessing*). Innerhalb der *notifyTimer*-Methode werden nun die protokollierten und noch nicht bearbeiteten Ereignisse gelesen und eventuell zusammengefaßt (*Get_UnhandledDB-Events*) und schließlich an den Event-Handler des FDBS-Kerns propagiert (*PropagateNewEvent*). Hierbei werden unter Umständen mehrere unbehandelte Ereignisse eines DB-Objektes zusammengefaßt (z.B. Insert eines DB-Objektes mit anschließendem Update). Die Ereignisse werden, wie in Kapitel 6.4 bereits dargestellt wurde, als CORBA-Objekte der Klasse *CDBObjectEvent* repräsentiert und können somit vom FDBS-Kern durch ihre CORBA-OID eindeutig über alle KDBS identifiziert werden.

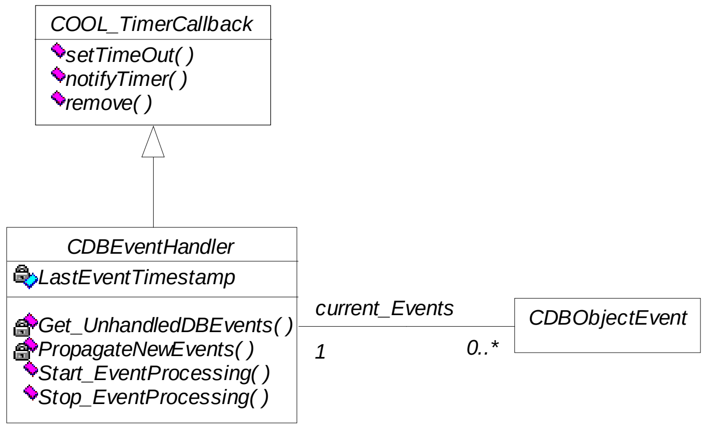


Abbildung 78 Klassendiagramm des lokalen Event-Handler

Wird der Datenbank-Adapter heruntergefahren, so wird der Event-Handler aufgefordert die Arbeit einzustellen (*Stopt_EventProcessing*). Er meldet sich hierzu beim Benachrichtigungsdienst ab (*COOL_TimerCallback::remove*) und deaktiviert alle noch nicht quittierten Ereignisse (Instanzen von *CDBObjectEvent*).

7.4 Der lokale Object-Manager

In diesem Abschnitt sollen die wesentlichen Ideen bei der Realisierung des lokalen Object-Managers beschrieben werden. Die Interaktion mit seinem globalen Pendant im FDBS-Kern wurde bereits in Kapitel 6.5 dargestellt. Die wesentliche Aufgabe des lokalen Object-Managers besteht darin, einen homogenen Zugriff (Datenmodellheterogenität, Heterogenität der Ablaufumgebung des KDBS) auf die lokalen DB-Objekte zu unterstüzen.

Der lokale Object-Manager wird als einzige Instanz der Klasse *CDBObjectManager* (siehe Abbildung 79) realisiert. Er ist der erste Ansprechpartner des globalen Object-Managers beim Zugriff auf ein lokales DB-Objekt (vergleiche hierzu Kapitel 6.5). Der Zugriff auf die lokalen DB-Objekte wird durch die Klasse *CDBObject* gekapselt und erfolgt durch die Methoden *Get_DBObject* und *Get_DBObjectRef*. Diese Methoden liefern jeweils eine Referenz (CORBA-OID) auf eine *CDObject*-Instanz, die als lokale Stellvertreter-Instanz des eigentlichen DB-Objektes während des Zugriffs fungiert. Die Methode *Get_DBObject* liefert dabei zusätzlich den Zustand des DB-Objektes in der im Kapitel 6.5.4 dargestellten plattformunabhängigen CORBA-IDL-Austausch-Repräsentation. Die Methode *Create_DBObject* erzeugt ein neues DB-Objekt auf Anforderung des globalen Object-Managers.

Bei seiner Arbeit stützt sich der lokale Object-Manager auf die *Meta-Daten* des Komponenten-Schemas ab. Der lokale DataDictinary-Manager (einzige Instanz der Klasse *CLocalDataDictManager*) ist dabei für den Object-Manager der Eintrittspunkt beim navigierenden Zugriff auf die Meta-Daten.

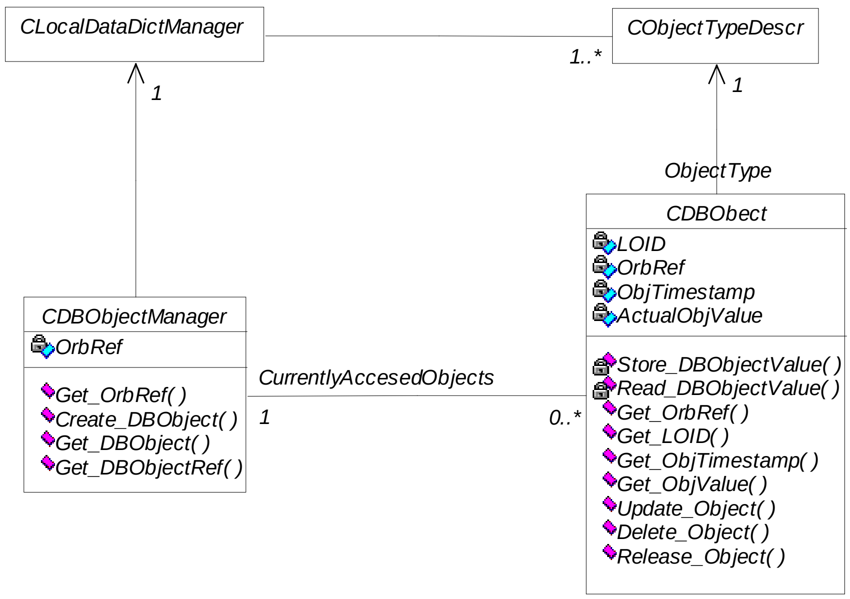


Abbildung 79 Klassendiagramm des lokalen Object-Manager

Der Zugriff auf die Daten (Zustand der DB-Objekte des Komponenten-Schemas) und Operationen auf diese, wird durch die Klasse *CDObject* gekapselt. Der Zustand eines DB-Objektes liegt dabei in der im Kapitel 6.5.4 dargestellten plattformunabhängigen CORBA-IDL-Austausch-Repräsentation vor.

- Die private Methode *Store_DBObjectValue* realisiert das physische Anlegen eines DB-Objektes (Konstruktor-Aufruf der Klasse *CDBObject*) in der Komponenten-Datenbank. Hierbei wird der Zustand des DB-Objektes gemäß der Schema-Transformations-Abbildung auf eine Menge von Tupeln des lokalen (relationalen) Schemas transformiert.

- Der Zustand eines DB-Objektes wird durch die private Methode *Read_DBObjectValue* beim ersten Zugriff auf das DB-Objekt implizit aus der Komponenten-Datenbank gelesen.

- Die Methoden *Update_Object* und *Delete_Object* realisieren das Speichern einer Zustands-Änderung bzw. das physische Löschen eines DB-Objektes in der Komponenten-Datenbank.

Die gerade dargestellten Methoden stützen sich auf eine Menge von Objekten ab, welche die eigentliche Transformation der Daten und Operationen entsprechend der in Abschnitt 7.1.5 dargestellten Schema-Transformations-Abbildung realisieren. Die Klassen dieser „Transformations-Objekte" sind in Abbildung 80 dargestellt.

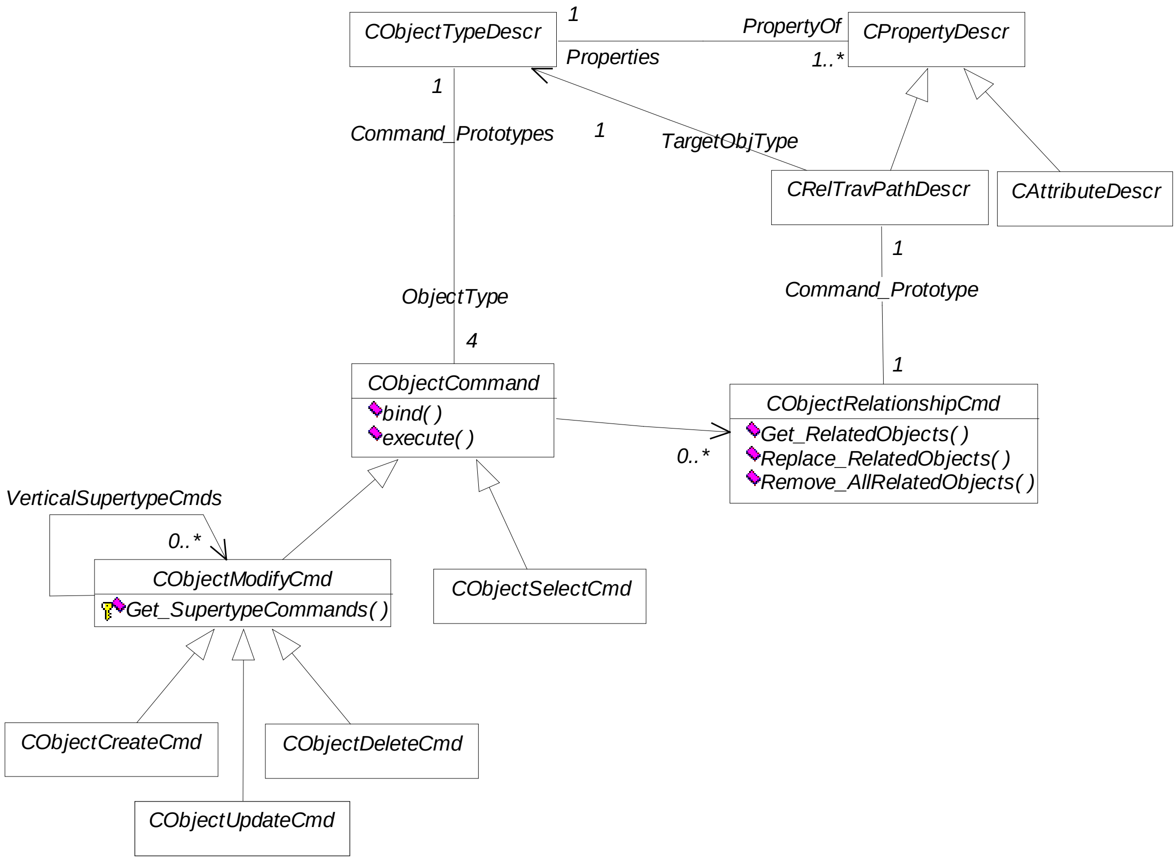


Abbildung 80 Die Schema-Transformations-Kommandos

Die Transformations-Objekte sind für die DB-Objekte (Instanzen von *CDBObject*) über deren Objekttypen (Instanz von *CObjectTypeDescr*) ereichbar (siehe Abbildung 79). Hierbei wird das Entwurfsmuster ***Prototyp*** [GamHeJoV94] angewandt. Jeder Objekttyp des Komponenten-Schemas verwaltet vier Transformations-Objekte (*Command_Prototypes*), welche Instanzen der von *CObjectCommand* abgeleiteten Klassen sind (siehe Abbildung 80). Diese Transformations-Objekte dienen dabei als *Prototypen* für eine konkrete Transformation und werden bei der Initialisierung der Objekttypen (beim Start des DB-Adapters) im voraus berechnet:

- Im Kontext einer konkreten Transformation liefern die Methoden *Get_ObjectSelectCommand, Get_ObjectCreateCommand, Get_ObjectUpdateCommand, Get_ObjectDeleteCommand* der Klasse *CObjectTypeDescr* (in Abbildung 80 nicht dargestellt) eine Kopie der bereits berechneten Transformations-Objekte (*Command_Prototypes*). Diese realisieren das Einlesen (*CObjSelectCmd*), Erzeugen (*CObjCreateCmd*), Speichern von Zustands-Änderungen (*CObjUpdateCmd*) und Löschen (*CObjDeleteCmd*) von DB-Objekten im Komponenten-Datenbanksystem. Die Operationvorschrift der Transformations-Objekte ist durch die Schema-Transformations-Abbildung gegeben.

- Die Abwicklung einer Transformation erfolgt dabei für alle Transformations-Objekte auf die gleiche Art und Weise (Polymorphie). Zunächst wird der Zustand des DB-Objektes bzw. dessen CORBA-IDL-Austausch-Repräsentation an das Transformations-Objekt gebunden (***bind***), bevor die eigentliche Transformation ausgeführt wird (***execute***).

Die Transformations-Objekte realisieren die von ihnen geforderte Funktionalität folgendermaßen:

- Jedes Transformations-Objekt verwaltet intern eine Menge von (dynamischen) SQL-Statements, die durch den *Prototypen* bereits berechnet wurden. Diese SQL-Statements dienen dazu die Attribut-Werte des DB-Objekt-Zustandes zu speichern bzw. zu lesen.

- Darüber hinaus verwaltet jedes Transformations-Objekt eine Menge von Instanzen der Klasse *CObjectRelationshipCmd*, welche die Transformation der Beziehungen (Zugriffs-Pfade) des DB-Objekt-Zustandes kapseln und ebenfalls als Transformations-Objekt bezeichnet werden. Hierbei wurde wiederum das Entwurfsmuster ***Prototyp*** angewandt. Die Methode *Get_ObjectRelation-shipCmd* (in Abbildung 80 nicht dargestellt) der Klasse *CRelTravPathDescr* liefert eine Kopie des entsprechenden Prototypen.

- Infolge des vertikalen Ansatzes zur relationalen Repräsentation von Vererbungsbeziehungen (siehe Abschnitt 7.1.5.1) verwalten die Transformations-Objekte der Klassen *CObjCreateCmd, CObjUpdateCmd und CObjDeleteCmd* (rekursiv) entsprechende Transformations-Objekte (*VerticalSupertypeCmds*) für ihre vertikalen Supertypen. Die Klasse *CObjectModifyCmd* ist die Generalisierung dieser gemeinsamen Eigenschaft. Die Ausführung einer Transformation (***execute***) geschieht dabei durch den rekursiven Aufruf der ***execute***–Methode der vertikalen „Supertyp-Transformations-Objekte“. Wegen der ***referenziellen Integrität*** wird bei der Erzeugung (*CObjCreateCmd*) eines DB-Objektes die Vererbungshierarchie von oben nach unten durchlaufen (vom Supertyp zum Subtyp). Eine Lösch-Operation (*CObjDeleteCmd:: execute*) durchläuft die Vererbungshierarchie von unten nach oben.

Kapitel 8 Das realisierte FDBS in Aktion

Das nun folgende Kapitel beschreibt das Ergebnis der Implementierung des föderierten Datenbanksystems vor dem Hintergrund des in Kapitel 5 dargestellten Anwendungs-Szenarios.

8.1 Die Komponenten-Datenbanksysteme des FDBS-Anwendungs-Szenarios.

Nachdem das föderierte Datenbanksystem rechtzeitig zum Termin der Eingemeindung der Städte Bochum und Wattenscheid fertig gestellt wurde, ist nun der Zeitpunkt des Stapellaufs gekommen.

Die Abbildung 81 zeigt das lokale Schema eines Einwohnermeldeamtes. Da dieses Schema bereits in Kapitel 7.1.4 beschrieben wurde, sollen in diesem Abschnitt nur die für die Integration der KDBS relevanten Aspekte erörtert werden. Beide Einwohnermeldeämter (Bochum und Wattenscheid) des Anwendungs-Szenarios haben das gleiche lokale Schema. Diese Vereinfachung wurde getroffen, um den Aufwand beim Einrichten des FDBS zu reduzieren. Die Fähigkeiten des FDBS, insbesondere der Replikations-Mechanismus für heterogene DB-Objekte (siehe Kapitel 6.6), kommen jedoch trotzdem zur Geltung, wie der weitere Verlauf dieses Kapitels zeigen wird.

Die Komponenten-Datenbanken des Anwendungs-Szenarios werden alle durch ein relationales Datenbankmanagementsystem (RDBMS) verwaltet (SQLBase 6.1 [Cent196]). Die Entwicklung eines Datenbank-Adapters für das SQLBase 6.1-System wurde in Kapitel 7 detailliert beschrieben.

Es wäre wünschenswert auch einen Datenbank-Adapter für das POET-Datenbanksystem als KDBS zur Verfügung zu stellen. Wie das Kapitel 7 zeigte, ist es eine sehr aufwendige Angelegenheit, einen Datenbank-Adapter zu entwickeln. Die Entwicklung eines POET-Datenbank-Adapters wäre im Vergleich zur Entwicklung eines Datenbank-Adapter für ein relationales Datenbanksystem weniger aufwendig aber auch gleichzeitig weniger interressant gewesen, da das Datenmodell des lokalen Schemas und das kanonische Datenmodell identisch sind.

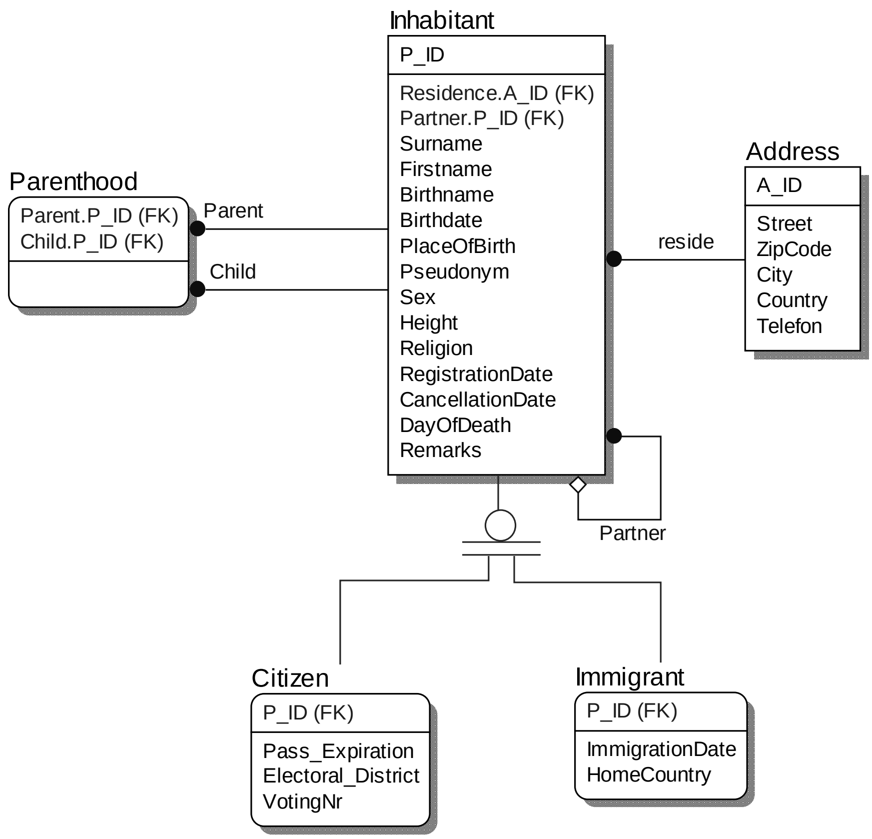


Abbildung 81 Das lokale Schema eines Einwohnermeldeamtes

Die lokalen Schemata der Datenbanken der Einwohnermeldeämter Bochum und Wattenscheid werden nun genauso wie es in Kapitel 7 beschrieben wurde, auf ein Komponenten-Schemata abgebildet (Schema-Transformation-Abbildung). Dabei wird die Schema-Transformations-Abbildung mithilfe der in Kapitel 7.1 beschriebenen Meta-Daten definiert. Dieser Vorgang wird manuell (bzw. durch SQL-Skripte) durchgeführt. Das Komponenten-Schema zu dem lokalen Schema in Abbildung 81 ist in Abbildung 82 dargestellt. Hierbei sind folgende Dinge hervorzuheben:

- Die in Abbildung 82 dargestellten Elemente *Parents, Children, Address, AddressOf, Partner* entsprechen den Zugriffs-Pfaden der wieder explizit gemachten Beziehungen (siehe Kapitel 7.1.5.2). Dabei sind die Zugriffs-Pfade *Partner* zu sich selbst invers.

- Der Primärschlüssel der Relation *Address* (*A_ID*) ist ein künstlicher Schlüssel und hat keine Semantik für die Diskurswelt. Deshalb ist er auch kein Attribut des Objekttyps *Address* in Abbildung 82. Unterschiedliche Kopien (*Publisher* und *Subscriber* siehe Kapitel 6.6) desselben *Address*-Objektes (bzw. des entsprechenden Tupels) werden dabei in verschiedenen KDBS unterschiedliche Werte des Primärschlüssel-Attribut *A_ID* zugewiesen. Die Werte werden dabei durch einen *before-Insert-Trigger* erzeugt und sind für die eigentlichen Objekttypen irrelevant.

- Die Wertebereiche der Primärschlüssel der Relationen *Inhabitant* (*P_ID*) in Bochum und Wattenscheid werden folgendermaßen eingeschränkt, wobei ***Integer₊*** nur die positiven ganzen Zahlen enthält:

$$\textit{\textbf{dom(Bochum.Inhabitant.P\_ID) := x+1, x}} \in \textit{\textbf{Integer}}_{+}$$

$$\textit{\textbf{dom(Wattenscheid.Inhabitant.P\_ID) := x+2, x}} \in \textit{\textbf{Integer}}_{+}$$

  Auf die existierenden Daten ist gegebenenfalls eine Konvertierung auszuführen. Analog gilt dies auch für die entsprechende *Inhabitant*-Relation in der zentralen Standesamt-Datenbank und sei schon hier vorweg genommen:

$$\textit{\textbf{dom(ZStandesamt.Inhabitant.P\_ID) := x+3, x}} \in \textit{\textbf{Integer}}_{+}$$

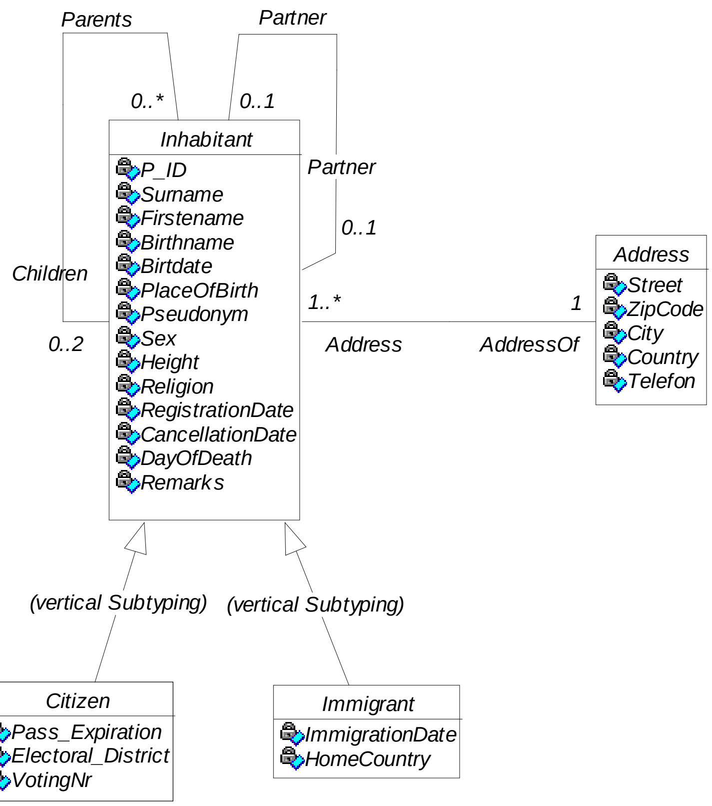


Abbildung 82 Das Komponenten-Schema eines Einwohnermeldeamtes

Für das in Abbildung 83 dargestellte lokale Schema des zentralen Standesamtes wird analog verfahren (siehe Abbildung 84). Hierbei ist folgendes hervorzuheben:

- Die Relation *Person* dient dazu, die Daten von Personen zu speichern, die nicht Einwohner der Städte Bochum oder Wattenscheid sind, aber trotzdem im zentralen Standesamt heiraten wollen.

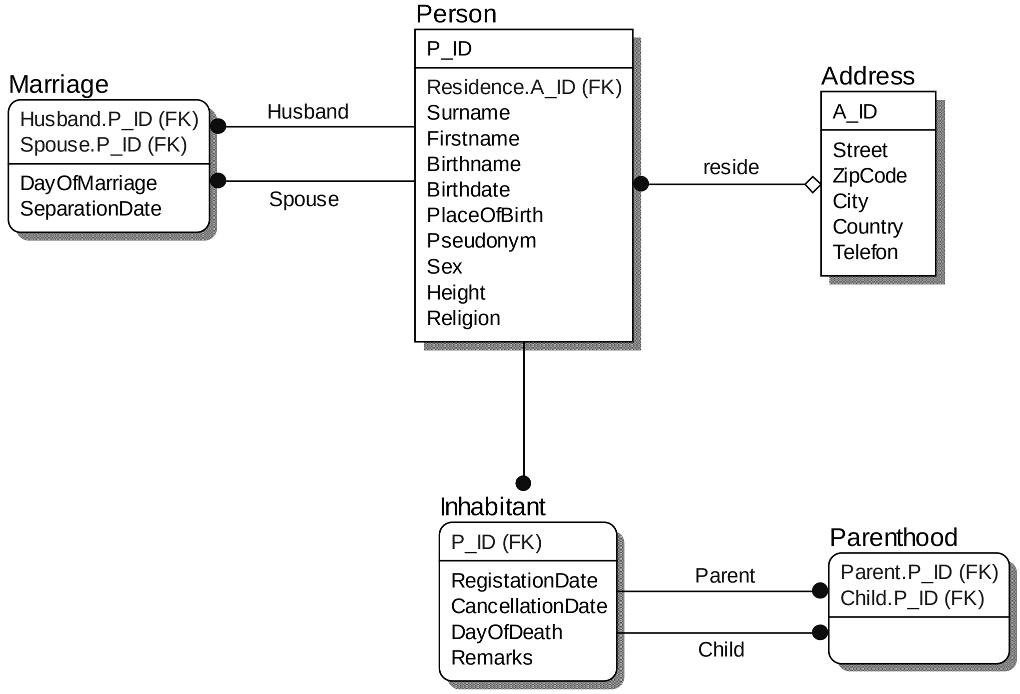


Abbildung 83 Das lokale Schema des zentralen Standesamtes

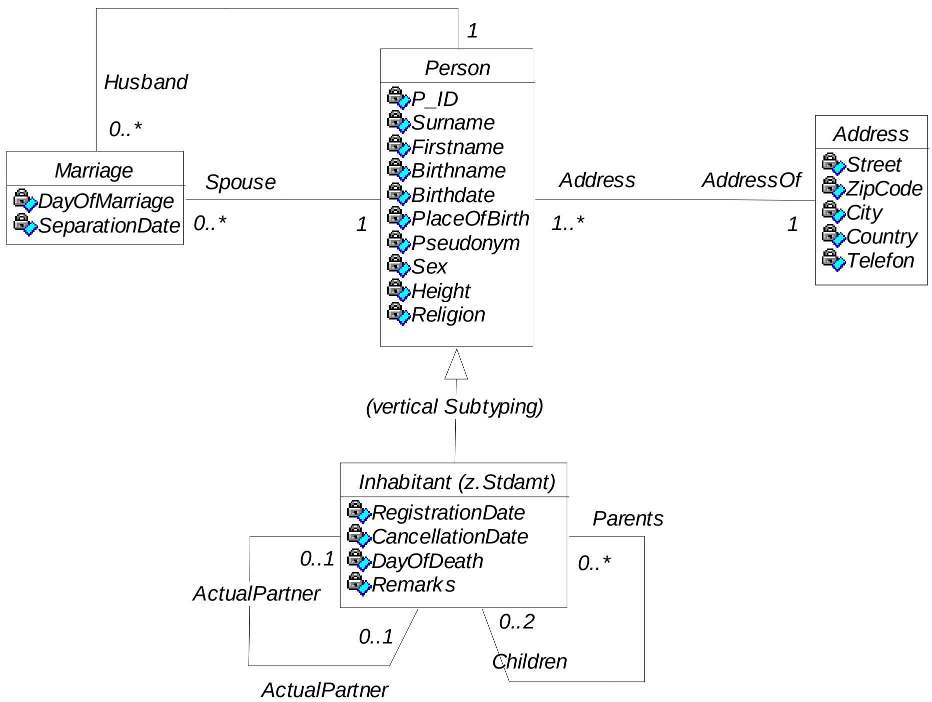


Abbildung 84 Das Komponenten-Schema des zentralen Standesamtes

- Zwischen den Relationen *Person* und *Inhabitant* (Einhwohner) besteht eine vertikale Vererbungsbeziehung (siehe Kapitel 7.1.5.1).

- Die *ActualPartner*-Beziehung zwischen DB-Objekten des Objekttyps *Inhabitant* ist eine Beziehung, die auf einem Datenbankanfrage-Ausdruck (*Query_Expression*, siehe Kapitel 7.1.5.2) basiert und setzt die gerade aktuell miteinander verheirateten Einwohner der Städte Bochum und Wattenscheid zueinander in Beziehung. Der Datenbankanfrage-Ausdruck (*Query_Expression*) zur Materialisierung der Zugriffs-Pfade der Beziehung hat folgende Gestalt:

```
Select DEST.LOID from
SYSADM.INHABITANT SRC,
SYSADM.MARRIAGE RTBL,
SYSADM.INHABITANT DEST
where (SRC.LOID = :1)
and (     ((RTBL.HUSBAND = SRC.P_ID) and
           (RTBL.SPOUSE = DEST.P_ID))
or        ((RTBL.SPOUSE = SRC.P_ID) and
           (RTBL.HUSBAND = DEST.P_ID))     )
and (SeparationDate IS NULL)
```

Die *ActualPartner*-Zugriffs-Pfade werden später im Replikations-Schema auf die korrespondierenden *Partner*-Zugriffs-Pfade der Objekttypen *Inhabitant* der Einwohnermeldeämter Bochum und Wattenscheid abgebildet.

8.2 Die Konfiguration des Systems

Nachdem die Schema-Transformations-Abbildungen der drei Komponenten-Datenbanksysteme definiert wurden, werden die dabei entstandenen Komponenten-Schemata in das *Data-Dictionary* des FDBS-Kerns importiert (Kapitel 6.3.3).

Die Datenbank-Adapter, als *Windows NT Console-Prozesse* implementiert, werden hierzu mit dem Parameter ***–R*** (wie Registrierung des Komponenten-Schemas) aufgerufen. Weitere Parameter sind ***-D*** [Name der Datenbank], ***-U*** [Name des speziellen DB-Benutzers „*FDAGENT*“ (siehe Kapitel 7.3.1)], ***-P*** [Password von „*FDAGENT*“].

Nachdem die Komponenten-Schemata nun (einmal) durch die Datenbank-Adapter exportiert wurden, haben die Administratoren des FDBS die Aufgabe, das Replikations-Schema (siehe Kapitel 6.6.1) zu spezifizieren. Diese geschieht (mithilfe des POET-ODMG-93-C++-APIs) durch kleine C++-Programme, welche das Replikations-Schema in der Hilfsdatenbank des FDBS anlegen.

Im folgenden soll nun das Replikations-Schema des Anwendungs-Szenarios graphisch dargestellt werden.

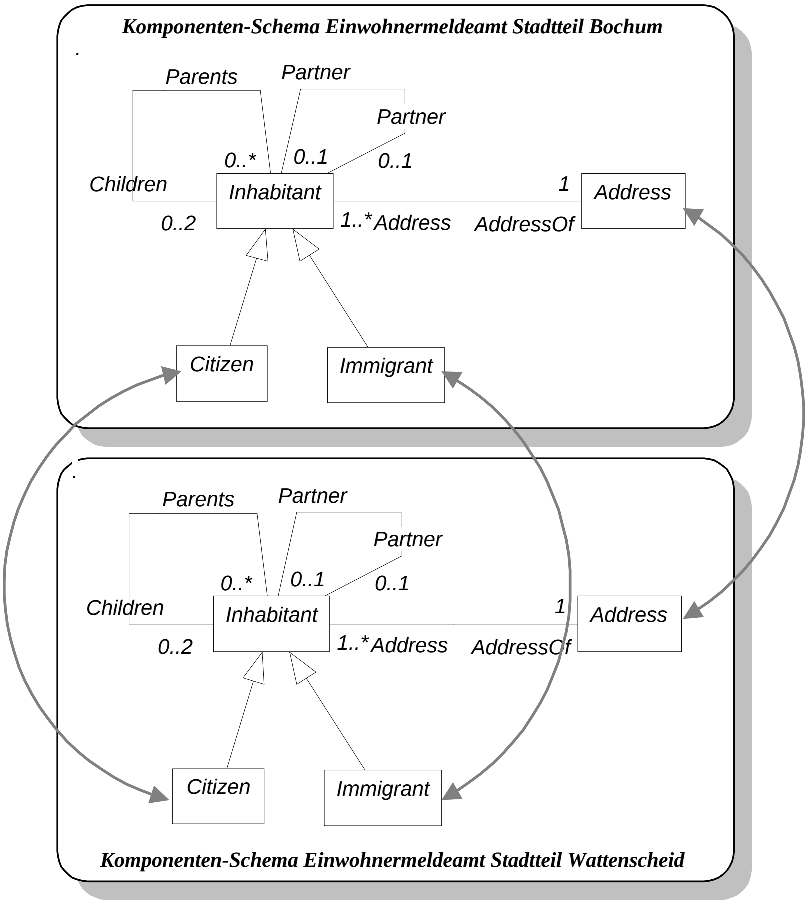


Abbildung 85 Replikation zwischen den Einwohnermeldeämtern (EWMA) Bo. und Wa.

Wie in Abbildung 85 dargestellt, werden alle DB-Objekte beider Datenbanken jeweils vom Einwohnermeldeamt (EWMA) Bochum zum EWMA Wattenscheid repliziert und umgekehrt. Die zweispitzigen Pfeile in Abbildung 85 haben dabei die Semantik, daß beide verbundenen Objekttypen zugleich als *Publisher* und *Subscriber* des jeweils anderen Objekttyps fungieren (vergleiche hierzu Kapitel 5.3.3 und 6.6).

In Abbildung 86 sind die Replikations-Beziehungen zwischen den Komponenten-Schemata des Einwohnermeldeamtes Bochum bzw. Wattenscheid und dem zentralen Standesamt dargestellt. Es werden alle relevanten Einwohnermeldeamtsdaten (Extension des Objekttyps *Inhabitant*) in den entsprechenden Objekttyp des zentralen Standesamt repliziert. Dabei fungiert der Objekttyp *Bochum.Inhabitant* bzw. *Wattenscheid.Inhabitant* als *Publisher*, der Objekttyp *Standesamt.Inhabitant* als *Subscriber*.

Da das zentrale Standesamt jedoch auch für Geburten-Anzeigen zuständig ist, wird der Objekttyp *Standesamt.Inhabitant* als *Publisher* des Objekttyps *Bochum.Citizen* bzw. *Wattenscheid.Citizen* definiert. Dies steht vor dem Hintergrund, daß alle Neugeborenen automatisch Bürger mit allen Rechten sind (Citizen).

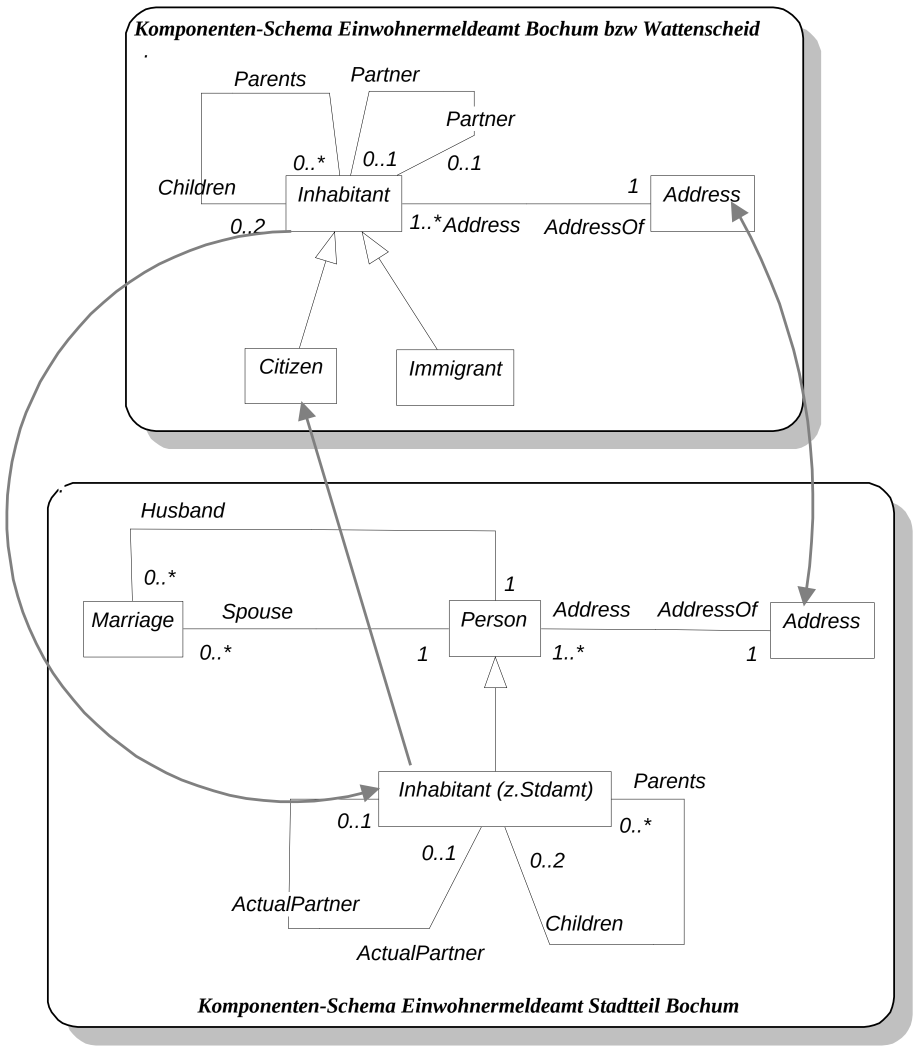


Abbildung 86 Replikation zwischen dem EWMA Bo. bzw. Wa. und dem Standesamt

8.3 Die Beispiel Applikationen des Anwendungs-Szenarios

Für die lokalen Schemata der Komponenten-Datenbanksysteme wurden mithilfe der Centura-4GL-Entwicklungsumgebung [Centura97] lokale Applikationen erstellt, welche die Arbeit der betrachteten Behörden simulieren sollen. Das nun folgende Beispiel soll ein Eindruck über die Fähigkeiten des föderierten Datenbanksystems vermitteln und entspricht der Demonstration als Teil des Abschlußvortrages dieser Diplomarbeit:

- Im Jahr 1996 entschließt sich eine Bürgerin aus Düsseldorf in die Stadt Bochum zu ziehen. Sie meldet diesen Wohnsitzwechsel dem Einwohnermeldeamt des Stadtteils Bochum (siehe Abbildung 87).

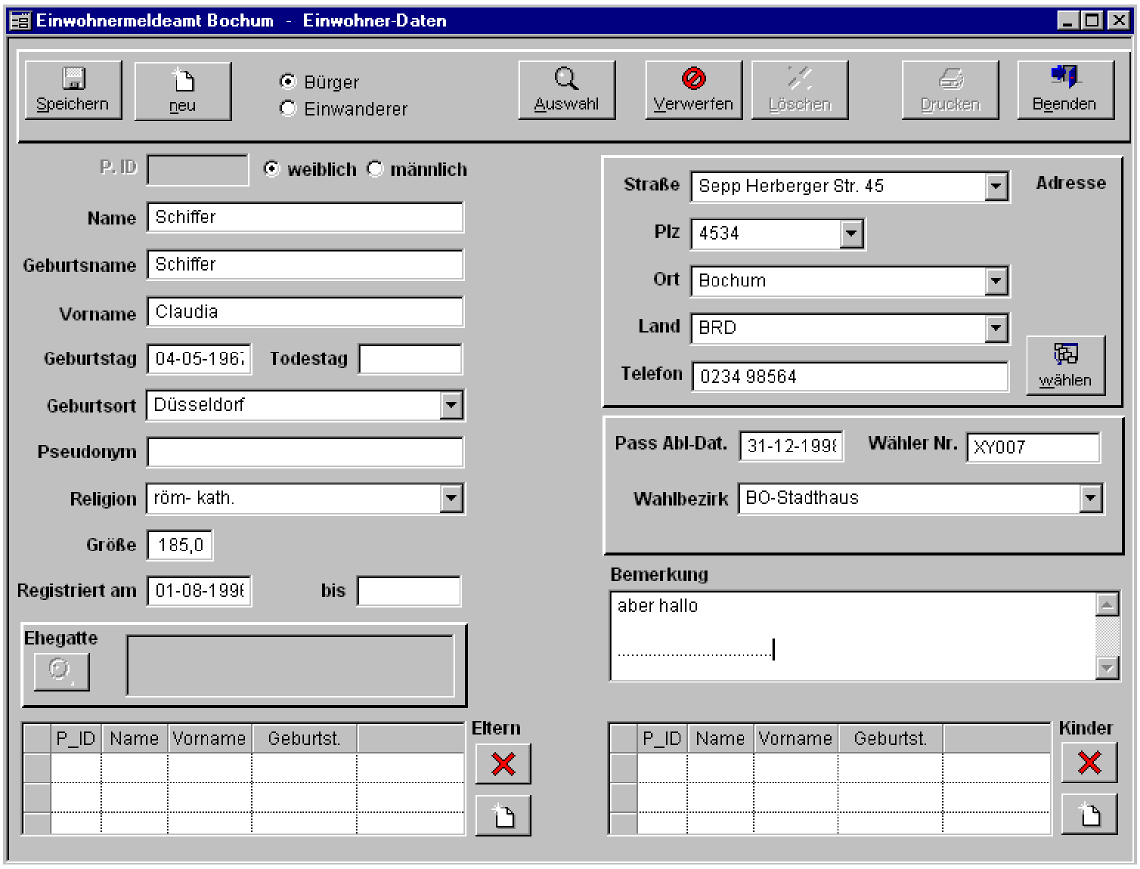


Abbildung 87 Erfassung einer neuen Bürgerin im Einwohnermeldeamt Bochum

- Kurze Zeit darauf meldet sich ein Einwanderer aus den USA im Einwohnermeldeamt Wattenscheid, um seinen neuen Wohnsitz registrieren zu lassen (siehe Abbildung 88).

- Wie das Leben so spielt, verlieben sich die gerade erwähnten Personen und heiraten am Freitag den 13. März 1998 im zentralen Standesamt. Wie es der Zufall so will, fällt gerade an diesem Tag die Netzwerkverbindung, die das zentrale Standesamt mit den übrigen Behörden der Stadt verbindet, aus. Der Standesbeamte will die Hochzeit schon absagen, als ihm sein Assistent mitteilt, daß seit der Eingemeindung Eheschließungen nun auch „*offline*“ durchgeführt werden können (siehe Kapitel 6.6). Die Zeremonie nimmt ihren Lauf (siehe Abbildung 89).

- Da die beiden Ehegatten nun seit der Hochzeitsnacht in einer gemeinsamen Wohnung leben, will die Ehegattin am Tag nach der Hochzeit ihren Wohnungswechsel im Einwohnermeldeamt des Stadtteils Wattenscheid melden. Ganz erstaunt muß sie dort feststellen, daß die Behörde bereits vollständig (datentechnisch) im Bilde ist (siehe Abbildung 90) und nichts mehr zu tun ist.

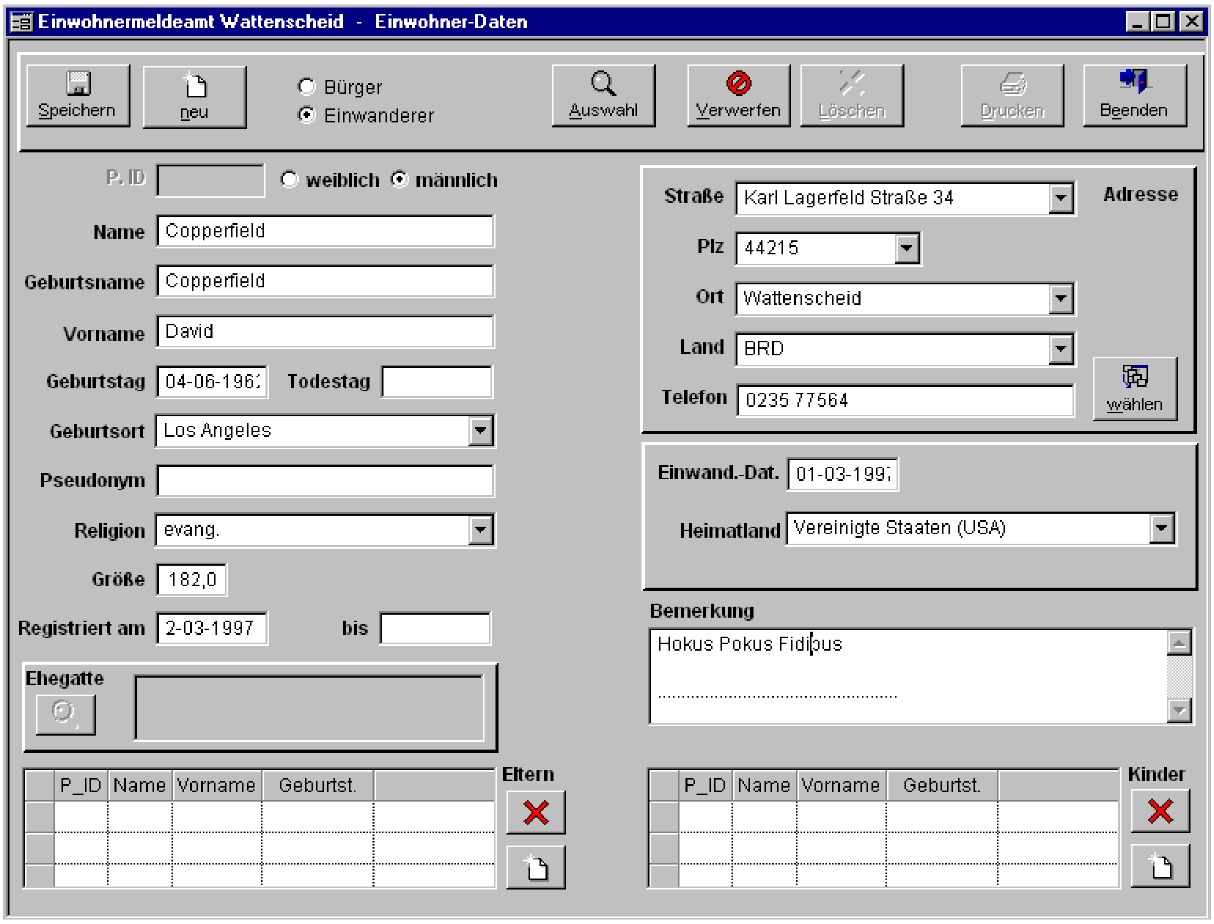


Abbildung 88 Erfassung eines Einwanderes im Einwohnermeldeamt Wattenscheid

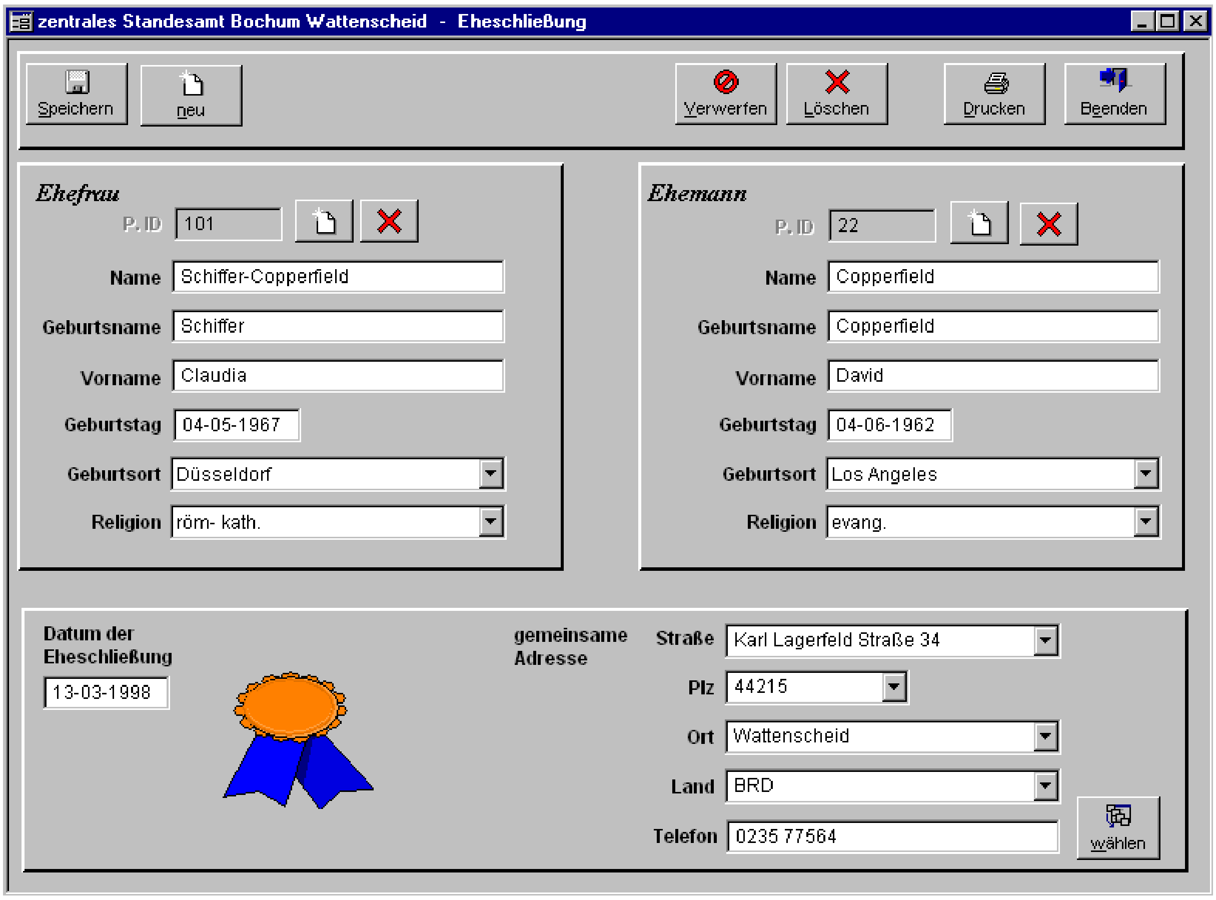


Abbildung 89 Eine Eheschließung im zentralen Standesamt

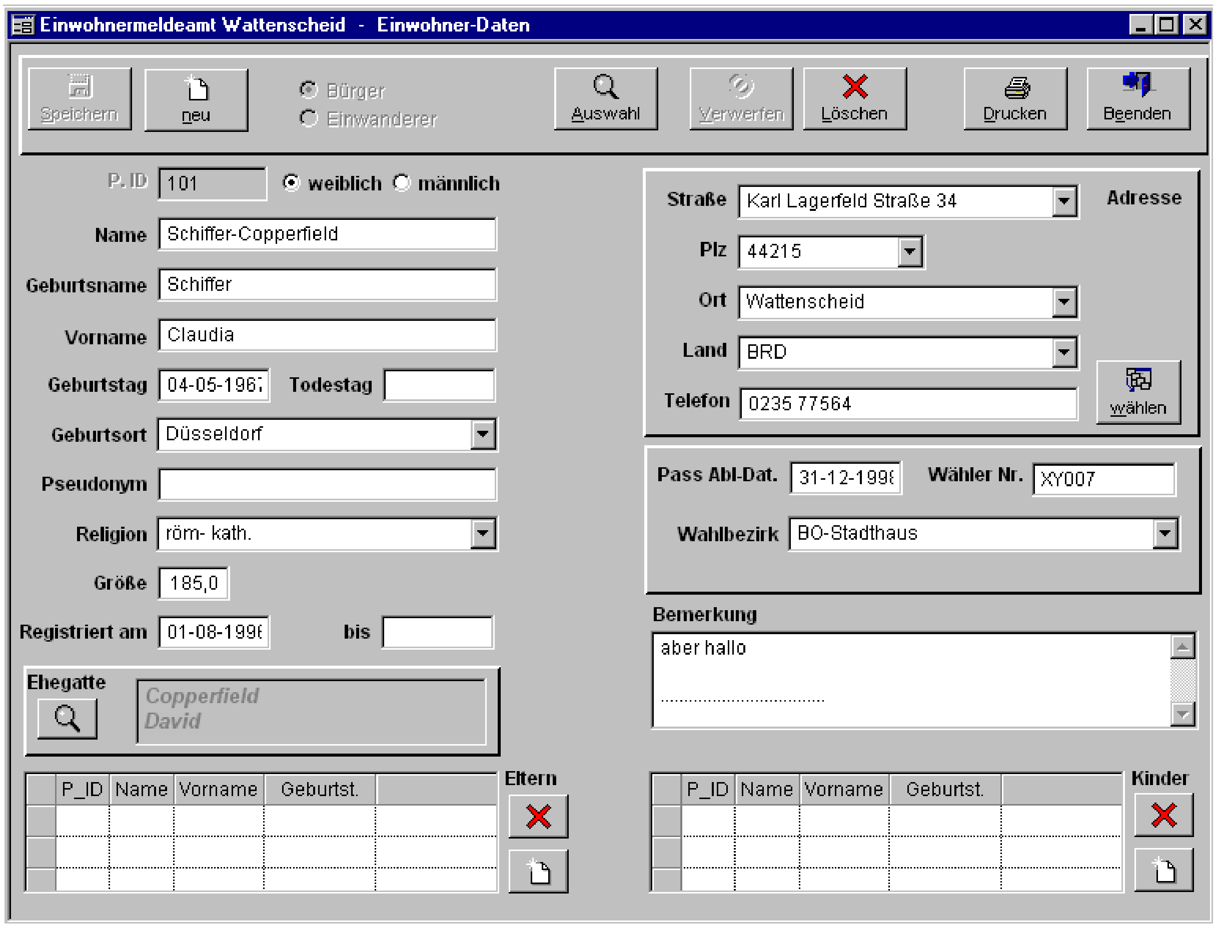


Abbildung 90 Replizierte Daten aus Bochum mit Änderung des zentr. Standesamtes

Kapitel 9 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde ein föderiertes Datenbanksystem unter Anwendung von modernen Standards entwickelt.

Im ersten Teil der Arbeit wurden die wichtigsten Begriffe und Konzepte für die Durchführung dieses Vorhabens erarbeitet. Hierbei wurden insbesondere die verwendeten Standards ausführlich dargestellt.

Der ODMG-93-Standard für objektorientierte Datenbankmanagementsysteme (ODBMS) bzw. dessen Objektmodell diente als kanonisches Datenmodell des entwickelten föderiertes Datenbanksystems. Wie in Kapitel 7 deutlich wurde, lassen sich die Konzepte des relationalen Datenmodells sehr gut auf das ODMG-93-Objektmodell abbilden. Vielmehr noch wurde gezeigt, das man einen großen Teil der impilziten Semantik eines relationalen Datenbankschemas bei der Schema-Transformation wieder explizit machen kann. Insofern hat sich die Forderung in [SCG91], daß die Ausdrucksfähigkeit des kanonisches Datenmodells größer (oder gleich) als die Ausdrucksfähigkeit der den Komponenten-Datenbanksystemen zugrundeliegenden Datenmodelle sein sollte, als sinnvoll erwiesen. Hinsichlich der Schema-Transformation, also der Beseitgung der Datenmodell-Heterogenität, hat sich der Einsatz des ODMG-93-Objektmodell als kanonisches Datenmodell bewährt.

Der in der Diplomarbeit entwickelte Schema-Transformations-Ansatz ließe sich, durch die Unterstüzung von komplexen Attributen, noch weiter verbessern. Hierbei wäre insbesondere eine Unterstützung von mengenwertigen Attributen sinnvoll. Darüber hinaus läßt sich dieser Ansatz durchaus auf andere relationale Datenbanksysteme übertragen. Wünschenswert wäre eine Werkzeugunterstüzung bei der Durchführung der Schema-Transformation. Ansätze hierfür werden bereits in der Literatur beschrieben [JaSchäZü96, RaHo96]. Eine vollständig automatische Schema-Transformation wird jedoch nicht sinnvoll sein, wie sich in Kapitel 7.1 bei der Identifizierung von Vererbungsbeziehungen gezeigt hat.

Der Einsatz des CORBA-Standards hat sich in vielerlei Hinsicht bewährt. Ein Object Request Broker stellt eine sehr gute Infrastruktur für die Entwicklung von verteilten, objektorientierten Systemen zur Verfügung. Die eigentliche Intention der Verwendung eines CORBA-Object Request Brokers in dieser-Diplomarbeit, war die Überwindung der Heterogenität der Ablaufumgebungen der Komponenten-Datenbanksysteme. Dieses Ziel wurde durch den eingesetzten ORB in vollem Unfang erreicht. Darüber hinaus zeigte sich im Verlauf der Entwicklung des FDBS, daß sich die Spezifikation der IDL-Interfaces sehr gut mit der eingesetzten objektorientierten Entwurfsmethode UML verbinden läßt.

Der Entwurf und die Entwicklung des FDBS wäre ohne die Unterstüzung der in dieser Arbeit verwendeten Werkzeuge garnicht möglich gewesen. Insbesondere das Entwurfwerkzeug ERwin, welches sämliche relationalen Schemata bis ins kleinste Detail (*Trigger, Stored Procedures* usw.) generierte, leistete in der Diplomarbeit wertvolle Dienste.

Literatur

[Ba95] H. Balzert. Methoden der objektorientierten Systemanalyse. Wissenschaftsverlag, Mannheim, 1995

[BDK92] F. Bancilhon. C. Delobel. und P. Kanellakis. Building *an Object-Oriented Database* System: The *Story of* 0_2. Morgan Kaufman, 1992.

[BeuDa96] T. Beuter, P. Dadam. Prinzipien der Replikationskontrolle in verteilten Datenbanksystemen. Informatik Forschung und Entwicklung Nr. 11. Springer Verlag, 1996.

[BFN94] R. Busse, P. Frankhauser und E. J. Neuhold. Federated Schemata in ODMG. In Proc. of 2nd Int. East/West Database Workshop Klagenfurt, Austria, S. 356-379. Workschop in Computing, Springer Verlag, September 1994.

[BFHK95] R. Busse, P. Frankhauser, Gerald Huck, Wolfgang Klas. Iro-DB: An object-oriented Approach towards federated and interoperable DBMS. Integrated Publication and Information Systems Institute GMD-IPSI, Darmstadt, 1995.

[BGS92] Y. Breitbart, H. Gracia-Molina, A. Silberschatz. Overview of multidatabase transaction managment, 1992. The VLDB Journal, 1(2) S. 181-240, Oktober 1992.

[BLN86] C. Batini, M. Lenzerine, und S.B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys, 18(4):323-364, Dezember 1986.

[BR96] G. Booch, J. Rumbaugh. Unified Modelling Language for object-oriented development, documentation set, Technischer Bericht, Rational Software Corporation, Santa Clara, CA, 1996.

[Cat94] R.G.G. Cattell. The Objectdatabase Standard: ODMG‘93. Morgan Kaufmann Publisher, 1994.

[Centura97] Centura Team Developer Online Manual. Centura Software Corporation, 1997.

[Cent1-96] SQLBase Database Administrator´s Guide. Centura Software Corporation, 1996.

[Cent2-96] SQLBase Language Reference. Centura Software Corporation, 1996.

[Cent3-96] SQLBase Advanced Topic Guide. Centura Software Corporation, 1996.

[Cent4-96] SQLBase Application Programming Interface Reference. Centura Software Corporation, 1996.

[Cent5-96] SQLBase SQLBase++ Class Library Reference. Centura Software Corporation, 1996.

[Cho1-96] CHORUS/COOL-ORB Programmer´s Guide. Chorus Systems, Frankreich, 1996.

[Cho2-96] CHORUS/COOL-ORB Programmer´s Reference Manual. Chorus Systems, Frankreich, 1996.

[Date90] C. J. Date. An Introduction to Database Systems. Volume 1. Addison Wesley, 5. Edition, 1990.

[Dogac95] A. Dogac. METU Interoperable Database System. Middle East Technical University (METU), Ankara, 1995.

[EK91] F. Eliassen, und R. Karlsen. Interoperability and Object Identity. ACM Sigmod Record, 20 (4): 25 - 29, Dezember 1991.

[ERw96] Erwin User´s Guide. Logic Works Inc., Princeton, 1996.

[Fuhr95] Norbert Fuhr. Skriptum zur Vorlesung Informationssysteme. Universität Dortmund, 1995, http://ls6-www.informatik.uni-dortmund.de/ir/teaching/courses/is/

[Fuhr96] Norbert Fuhr. Skriptum zur Vorlesung Information Retrieval. Universität Dortmund, 1996, http://ls6-www.informatik.uni-dortmund.de/ir/teaching/courses/ir/

[Froese95] J. Froese. Effiziente Systementwicklung mit Oracle7. Addison Wesley, 1995.

[GamHeJoV94] E. Gamma, R. Helm, R. Johnson und J. Vlissides. Design Pattern – Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

[Has97] Federated Integration of Replicated Information within Hospitals. International Journal on Digital Libraries, 1997 (im Druck)

[Hei93] W. Heijenga. „Vom Schema zur Sicht: Änderungsmöglichkeiten bei der Sichtableitung in einem objektorientierten Datenbanksystem". Cadlab Report *22*/.93. Paderborn. 1993.

[Hi95] E. Hildebrandt. Eignungsuntersuchung des ODMG-93 Objektmodells als Objektmodell einer Föderierungsschicht zur Integration heterogener, autonomer Datenbanksysteme. Diplomarbeit and der Fakultät für Informatik der Otto von Guericke Universität Magdeburg, 1995.

[Ho96] T. Holzer. Objektorientierte Standardsoftware ? in Objekt Spektrum Nr. 1 Jan/Feb 1996, S. 68 - 71.

[Hornig96] P. Hornig. Die Kinder der OMA – Überblick über CORBA-Implementierungen in Objekt Spektrum Nr. 1 Jan/Feb 1996, S. 38 - 43.

[HoKo95] U. Hohenstein and C. Körner. Semantische Anreicherung relationaler Datenbanken. In G. Lausen, editor, *Proc. GI-Fachtagang „Datenbanksysteme in Büro, Technik and Wissenschaft" (BTW'95), Dresden,* S. 130-149. Informatik aktuell, Springer-Verlag, Mai 1995.

[HoPl95] U. Hohenstein V. Plesser. Semantic Enrichment: A First Step to Provide Database Interoperability. Corperate Research & Development Siemens AG, ZT, ZFE T SE 3, München, 1995.

[HTJ+95] M. Höding, C. Türker, S. Janssen, K.-U. Sattler, S. Conrad,G. Saake, und I. Schmitt. Föderierte Datenbanksysteme - Grundlagen und Ziele des Projektes $SIGMA_{FDB}$. Preprint Nr. 12, Fakultät für Informatik der Otto von Guericke Universität Magdeburg, 1995.

[IRO-DB1-94] IRO-DB: Interoperable Object Manager Design Specification. Version 1.5, Document Nr. D4-2/1, GMD 1994.

[IRO-DB2-94] IRO-DB: Design Document for Data Dictionary. Version 1.3, Document Nr. D4-3/1, GMD 1994.

[IRO-DB1-95] IRO-DB: Design Specification for the Global Transaction Management. Version 1.1, Document Nr. D4-6/1, GMD 1995.

[JaSchäZü96] J. Jahnke, W. Schäfer und A. Zündorf. A Design Environment for Migrating Relational to Object-Oriented Database Systems. International Conference on Software Maintance (ICSM), IEEE Computer Society, 1996.

[JJC9I] C. Jacquemot. P. Strarup Jensen, und S. Carrez. CHOHUS/COOL: CHORUS Object Oriented Technology. In *Object-Based Parallel and Distributed Computation (OBPDC ´95)*, Band 116 der Reihe Lecture Notes in Computer Science, Seiten 187-204. Springer Verlag 1991.

[KlaFiAb94] W. Klas, G. Fischer und Karl Aberer. Integrating Relational and Object-Oriented Database Systems using a Metaclass Concept. Journal of System Integration, Vol. 4 No. 4, 1994, Kluwer Academic Publisher.

[KoKraNi96] A. Koschel, R. Kramer und R. Nikolai. A Federation Architecture for an Environmental Information System incorporating GIS, the World Wide web and CORBA. Forschungszentrum Informatik, Karlsruhe, 1996. http://www.fzi.de

[Kroh93] P. Kroha. Objects and Databases. The McGraw-Hill International Series in Software Engineering, 1993.

[ODMG97] ODMG Document. The Standard for Object Databases 1997. http://www.odmg.org

[$O_2$95] O_2 Programmer´s Reference Guide. O_2 Software, 1995

[OHE96] R. Orfali, Harkey D. und J. Edwards. The Essential Distributed Object Survival Guide. John Wiley & Sons New York ,1996.

[OMG91] Object Management Group. The Common Object Request Broker: Architeture and Spezification, OMG Document Number 91.12.1, Dezember 1991

[OMG94] Object Management Group. Common Object Services Specification, Volume 1. OMG Document Number 94.3.1, März 1994

[OMG95] Object Management Group.OMG´s Object Model. OMG Document Number 95.3.22, März 1995

[Oest96] Objektorientierte Softwareentwicklung: Analyse und Design mit der UML. Oldenbourg Verlag, 1996.

[Ontos96] ONTOS Inc. ONTOS Object Integration Server: Integrating Objects with Relational Databases. http://www.ontos.com, 1996.

[PBE95] E. Pitoura, O. Bukhres, und A. Elmagarmid. Object Orientation in Multidatabase Systems. ACM Computing Surveys, 27(2):141-195, Juni 1995.

[Poet1-96] POET 4.0 Developer´s Workbench Guide. POET Software Hamburg, 1996.

[Poet2-96] POET 4.0 Administrator´s Workbench Guide. POET Software Hamburg, 1996.

[Poet3-96] POET 4.0 Generic Programming Guide. POET Software Hamburg, 1996.

[Poet4-96] POET 4.0 Programmer´s Guide. POET Software Hamburg, 1996.

[Poet5-96] POET 4.0 ODMG Programmer's Guide. POET Software Hamburg, 1996.

[Poet6-96] POET 4.0 ODMG Reference Guide. POET Software Hamburg, 1996.

[Poet7-96] POET 4.0 ODMG C++ Binding Guide. POET Software Hamburg, 1996.

[Rad94] E. Radeke. Efendi: Federated Database System of Cadlab.Technical Report 39-94, University Paderborn & Siemens Nixdorf, Oktober 1994.

[Rah94] E. Rahm. Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenverarbeitung. Addison Wesley, Bonn, 1994.

[RaHo96] S. Ramanathan und J. Hodges. Extraction of Object-Oriented Structures from existing Relational Databases. Department of Computer Science, Mississipi State University, 1996.

[RS94] E. Radeke und M. Scholl. Federation and Stepwise Reduction of Database Systems. In Conf. on Application on Databases (ADB), Vadstena, 1994.

[RS95] E. Radeke und M. Scholl. Functionality for Object Migration among Distributed, Heteregenous, Autonomous Database Systems. In Workschop Research Issues on Data Engineering - Distributed Object Management (RIDE-DOM), Tapei, 1995.

[SCG91] F. Saltor, M. Castellanos und M. Garcia-Solaco. Suitability of Data Models as Canonical Data Models in Federated Databases. ACM SIGMOD Record 20 (2), S. 44-48, 1991.

[Sch95] I. Schmitt. Flexible Integration and Verivation of Heterogeneous Schemata in Federated Database Systems. Preprint Nr. 10, Fakultät für Informatik, Universität Magdeburg, November 1995

[ScSa95] I. Schmitt, G. Saake. Managing Object Identity in Federated Database Systems. Institut für technische Informationssysteme, Fakultät für Informatik der Otto von Guericke Universität Magdeburg, 1995.

[ScSa96] I. Schmitt and G. Saake. Flexible Generation of Global Integrated Schemata using GIM. In S. Conrad, M. Höding, S. Janssen, and G. Saake, editors, *Kurzfassungen des Workshops ‚Föderiete Datenbanken", Magdebnrg, 22.04-23.04.96,* pages 29-43. Bericht 96-01, Institut für Technische Informationssysteme, Universität Magdeburg, April 1996.

[Siegel96] J. Siegel. CORBA Fundamentals and Programming. John Wiley & Sons New York, 1996.

[Souza95] Cassio Souza. Design and Implementation of an Object-Oriented View Mechanism. Le Chesnay Cedex France, 1995.

[SpaPa95] S. Spaccapietra, C. Parent. View Integration: A Step Forward in Solving Structural Conflicts. IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2, April 1996

[SL90] A. Sheth und J. Larson. Federated database systems for managing distributed, heterogeneous,and autonomous databases. ACM Computing Surveys, 22(3):183-236,1990.

[Str91] Stroustrup, B.. *The C++ Progrornrning Longuage.* 2nd Edition. New York: Addison Wesley. 1991.

[SV95] D. C. Schmidt, S. Vinoski. Objects Interconnections, Introduction to Distributed Object Computing. C++ Report Magazine Januar 1995.

[Tan89] A. S. Tanenbaum. Computer Networks. Prentice-Hall, 1989.

[Vis95] Visual C++ Online Documentation System. Microsoft Corporation, 1995

[Xuequn95] Xuequn Wu. An Architecture for Interoperation of Heterogeneous Database Systems. Fraunhofer Institut für System- und Software-Technik. Dortmund, 1995.