Universität Leipzig

Fakultät für Mathematik und Informatik

Institut für Informatik

Modellierung und prototypische Implementierung eines Repository-Ansatzes für technische Metadaten in Data Warehouse-Umgebungen

Diplomarbeit

Aufgabenstellung und Betreuung: Prof. Dr. habil. Erhard Rahm,
Dipl.-Math. Robert Müller,
Dipl.-Inf. Thomas Stöhr,
Dipl.-Inf. Do Hong Hai

Vorgelegt von: Nguyen Thi Minh Thu

Leipzig, Februar 2000

Vorwort

Das Thema der vorliegenden Arbeit entstand im Rahmen des Projektes „Evaluierung von Metadaten-Produkten“, das die Abteilung Datenbanken des Instituts für Informatik der Universität Leipzig im Auftrag der R+V-Versicherung Wiesbaden durchgeführt hat.

Bei folgenden Personen möchte ich mich besonders bedanken:

- Dem Institutsleiter Herrn Prof. Dr. E. Rahm für die Aufgabenstellung und die Unterstützung der Arbeit.
- Herrn R. Müller, Herrn Th. Stöhr und Herrn Do Hong Hai für die engagierte Betreuung der Arbeit und viele nützliche Anregungen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CASE	Computer Aided Software Engineering
CLI	Call Level Interface
COM	Common Object Model
CORBA	Common Object Request Broke Architecture
CWM	Common Warehouse Model
CWMI	Common Warehouse Meta Data Interchange
DBMS	Datenbank-Managementsystem
DDL	Data Definition Language
DML	Data Manipulation Language
DTD	Document Type Definition
ERM	Entity Relationship Model
ESQL	Embeded Structureal Query Language
ETL	Extraktion, Transformation, Laden
IUS	Informix Universal Server
MDC	Meta Data Coalition
MDIS	Meta Data Interchange Specification
MDX	MetaData eXchange
MOF	Meta Object Facility
MX2	Metadata eXchange 2
ODBC	Open Database Connectivity
OIM	Object Information Model
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
OMG	Object Management Group
ORDBS	Objektrelationales Datenbanksystem
ORM	Objektrelationales Modell
RDBS	Relationales Datenbanksystem
SMIF RFP	Stream-based Model Interchange Request For Proposal
SQL	Structural Query Language

UML	Unified Modeling Language
XML	Extensible Markup Language
XMI	XML Metadata Interchange
XIF	XML Interchange Format
W3C	World Wide Web Consortium

Abbildungsverzeichnis

1 Einleitung

1.1 Data Warehousing

Die allererste Bestimmung des Begriffes „Data Warehouse“ wird auf Inmon zurückgeführt. Nach ihm wird ein Data Warehouse als eine „subjekt-orientierte, integrierte, zeitvariante und nicht-flüchtige Sammlung von Daten zur Entscheidungs-Unterstützung“ charakterisiert ([Inm93]). Mit Hilfe eines solchen „Informationslagers“ können wichtige Geschäftstendenzen entdeckt und erforscht werden, damit bessere und schnellere Entscheidungen bezüglich unterschiedlicher Geschäftsaspekte getroffen werden können.

Die Daten eines Data Warehouse kommen aus einer Vielzahl von heterogenen Datenquellen wie z.B. Datenbanksystemen, Dateisystemen, Anwendungen. Erst nachdem sie extrahiert, bereinigt und transformiert werden, können sie für Analyseaufgaben zur Verfügung stehen. Die Datenquellen werden häufig als operative oder operationale Systeme und das Data Warehouse als dispositives System bezeichnet. Ein charakteristischer Unterschied zwischen beiden Systemen besteht darin, daß während das erstere auf sogenanntes Online Transaction Processing (OLTP) basiert, d.h. strukturierte, oft wiederholte Tasks mit kurzen, atomaren und isolierten Transaktionen, ist das letztere ein zum Analysezweck genutztes OLAP-System (Online Analytical Processing). Die Daten eines OLTP-Systems sind in der Regel normalisiert. Dagegen sind die Data Warehouse-Daten oft unnormalisiert, damit eine bessere Performanz für komplexe Adhoc-Anfragen gewährleistet wird.

Die Architektur eines Data Warehouse besteht aus drei Ebenen: der operationalen Ebene, der Data Warehouse-Ebene und der Benutzer-Ebene (siehe Abbildung 1-1). Hier wird die Datenbewegung innerhalb einer Data Warehouse-Umgebung deutlich: von der operationalen Ebene zum Data Warehouse, vom Data Warehouse zu den Data Marts und dann zu den Anwendungen. Obwohl der Data Warehousing Prozeß verschiedene Phasen umfaßt wie Entwurf, Aufbau, Nutzung und Pflegen, kann man sie in zwei Kategorien einordnen, nämlich den Extraktions-, Transformations- und Laden-Prozeß (ETL-Prozeß) hinsichtlich der Datenbereitstellung und den Prozeß der Analyse, Abfrage und Navigation hinsichtlich des Datenzugriffs.

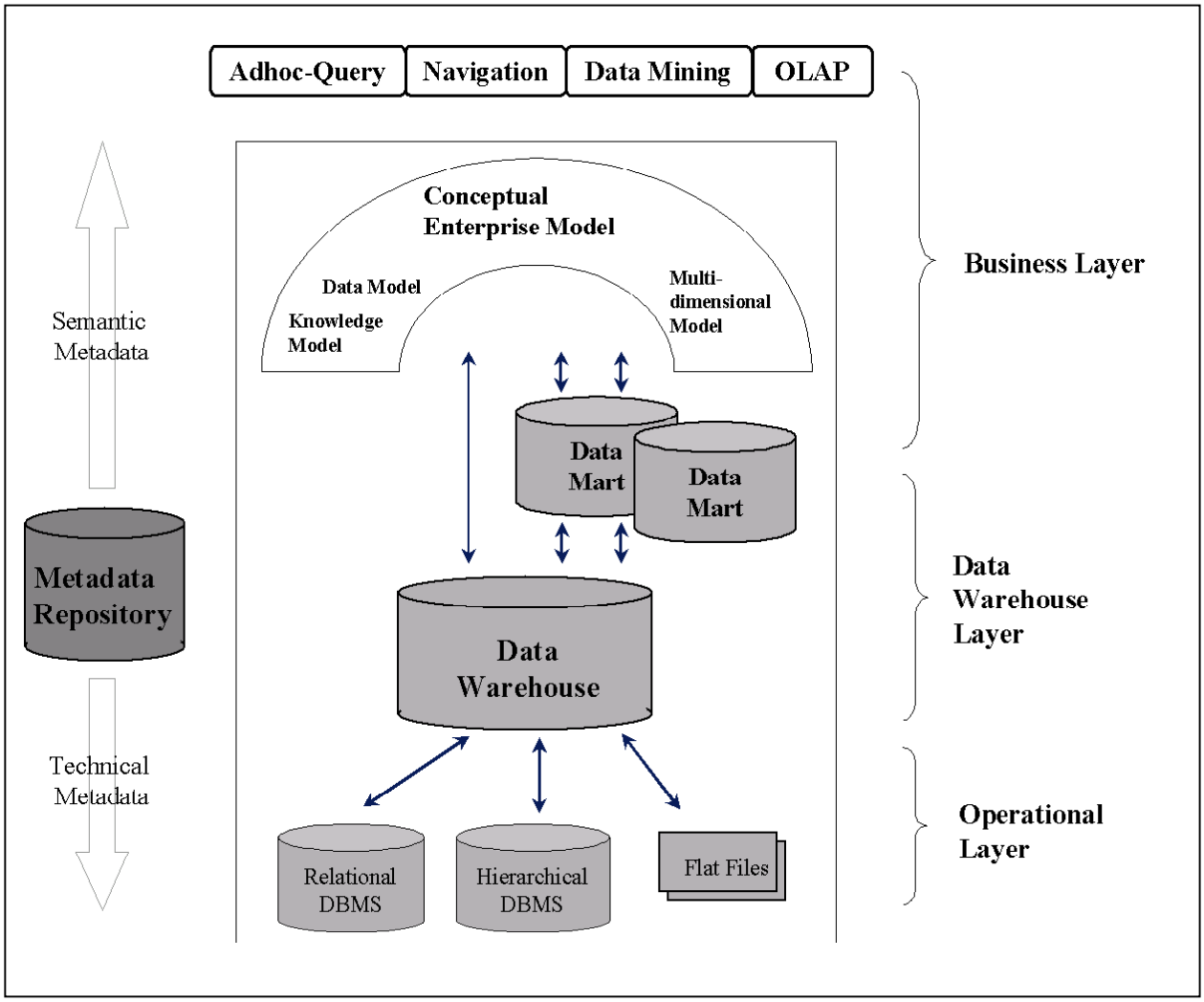


Abbildung 1-1: Dreiebenen-Architektur einer Data Warehouse-Umgebung ([MSR99])

Die Rolle und die Bedeutung von Metadaten für ein Informationssystem wurden längst erkannt. Insbesondere für eine Data Warehouse-Umgebung, wo eine Unmenge von Daten aus mehreren heterogenen Quellen abgelegt wird, ist ein Metadaten-Management von noch größerer Bedeutung. Deshalb sollen Metadaten eines Data Warehouse unter neuen Gesichtpunkten betrachtet werden. Entsprechend der oben geschilderten Architektur des Data Warehouse können Metadaten in dieser Umgebung in zwei Klassen unterteilt werden, die mit beiden Data Warehouse-Prozessen zusammenhängen. Metadaten, die mit dem Analyse- und Navigationsprozeß verbunden sind, können als semantische Metadaten bezeichnet werden. Und Metadaten, durch die sich der ETL-Prozeß beschreiben läßt, werden technische Metadaten genannt. Diese Unterteilung von Metadaten kann auch auf die Sichtweisen der Benutzer zurückgeführt werden: während technische Metadaten zum Arbeitsumfeld

der Administratoren bzw. Entwickler gehören, interessieren sich Business-Benutzer hauptsächlich für semantische Metadaten.

Obwohl jede dieser Metadaten-Klassen speziellen Aufgaben unterliegt und unter verschiedenen Gesichtpunkten betrachtet wird, besteht zwischen ihnen ein enger Zusammenhang. Durch technische Metadaten wird es Business-Benutzern ermöglicht, richtige Daten zu finden und darauf zuzugreifen. Andererseits müssen die im Data Warehouse gespeicherten physikalischen Daten mit den Business-Begriffen des Unternehmens übereinstimmen. Dieser Zusammenhang spiegelt auch die Verbindung zwischen beiden Hauptprozessen der Datenbewegung in einem Data Warehouse. Es wird deshalb angestrebt, ein Metadaten-Management-System zu konstituieren, das diese Prozesse umfassen und die Metadaten integrieren kann. Eine notwendige Voraussetzung dafür ist die Speicherung und Verwaltung der Metadaten in einem Repository. Durch die Verwendung des Repository wird nicht nur die Integration von Metadaten, sondern auch die Integration unterschiedlicher Werkzeuge, die Wiederverwendung von Software u.a. ermöglicht.

1.2 Aufgabenstellung

Die Integration von Metadaten stellt große Anforderung an ein leistungsfähiges Metadaten-Management des Data Warehouse. Da das ein komplexes Problem-Gebiet ist, kann im Rahmen dieser Diplomarbeit nicht die ganze Problematik behandelt werden. Es werden vor allem nur die technischen Metadaten und der damit verbundene Prozeß der Datenbereitstellung (ETL-Prozeß) untersucht.

Das Ziel der vorliegenden Diplomarbeit umfaßt die Modellierung und prototypische Implementierung eines Repository-Ansatzes für technische Metadaten in Data Warehouse-Umgebungen. Daraus ergeben sich die zwei folgenden Hauptaufgaben:

1. Modellierung eines Repository-Ansatzes für technische Metadaten in Data Warehouse-Umgebungen.

2. Prototypische Implementierung des konzeptuellen Metadaten-Modells in einem DBMS (Datenbankmanagementssystem).

Die erste Aufgabe schließt sowohl die semantische als auch die konzeptuelle Modellierung ein: zunächst wird mittels einer Modellierungssprache ein Repository-Metamodell entworfen, das alle relevanten Objekte und Aufgaben eines technischen Metadaten-Managements

umfassen kann; anschließend soll dieses Metamodell in das Datenmodell der zugrundeliegenden Repository-Datenbank transformiert werden. Eine Anforderung an das entworfene Metamodell besteht darin, daß es möglichst erweiterbar, insbesondere um semantische Eigenschaften, sein soll, weil diese Arbeit trotz der Beschränkung auf technische Metadaten ein Grundstein für weitere Arbeit bezüglich der Metadaten-Integration bilden muß.

Im Implementierungsschritt soll eine Repository-Datenbank entwickelt werden, die in der Lage ist, möglichst alle im konzeptuellen Schema modellierten Daten abbilden zu können. Zu den Aufgaben der Implementierung gehören außerdem die Definition und Speicherung einiger Test-Mappings durch das Repository. Diese Mapping-Beispiele wurden für das Projekt der Evaluierung von Metadaten-Produkten der Abteilung Datenbanken verwendet und befinden sich im Anhang C. Der Schwerpunkt dieser Arbeit liegt vor allem in der Modellierung eines Repository-Ansatzes und in der Implementierung des modellierten Schemas. Beim Testen kann keine tatsächliche Mapping-Durchführung (Datenbewegung) vorgenommen werden.

Diese Diplomarbeit wird wie folgt gegliedert:

Im nächsten Kapitel (Kapitel 2) werden die Rolle der Metadaten sowie die Funktionalitäten des Metadaten-Repository, insbesondere in Data Warehouse-Umgebungen, erörtert. Zu den Hauptpunkten dieses Kapitels gehören die Klassifizierung und die Standardisierung von Data Warehouse-Metadaten. Es folgt ein kurzer Überblick über den heutigen Repository-Produktmarkt bzw. über die führenden Repository-Hersteller.

Das dritte Kapitel behandelt die erste Hauptaufgabe der Diplomarbeit: die konzeptuelle Modellierung des Repository-Datenbankschemas. Zunächst wird dieses Schema mittels der Modellierungssprache UML, ein Standard von OMG, entworfen. Alle Klassen, die relevante Objekte eines technischen Metadaten-Managements repräsentieren, und ihre Beziehungen zueinander werden ausführlich erläutert. Das UML-Schema wird anschließend in ein objektrelationales Datenmodell transformiert, das die Grundlage für die Implementierung auf dem objektrelationalen Informix-DBMS bildet.

Im vierten Kapitel wird die Implementierung des Schemas auf dem DBMS Informix Universal Server (Version 9.1) behandelt. Es wird mit ODBC auf der Plattform Sun Solaris programmiert. Nach einer kurzer Einführung in Informix-DBS und ODBC wird das Ausfüllen der Repository-Datenbank vorgeführt. Dabei werden insbesondere repräsentative Funktionen der Repository-Engine implementiert, wie z.B. Importieren/Definieren, Lö-

schen und Anzeigen von Metadaten bzw. Mappings. Dafür werden die im Anhang A und Anhang B angegebenen Datenstrukturen der zwei relationalen Datenbanken (Quell: die Filmdatenbank auf dem DB2-DBS; Ziel: die Informix-Filmdatenbank) und die im Anhang C beschriebenen Test-Mappings verwendet.

Das fünfte Kapitel faßt die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

2 Metadaten für Data Warehouse-Umgebungen

2.1 Rolle von Metadaten in Data Warehousing

Der Nutzen von Metadaten besteht darin, eine konsistente Dokumentation der Struktur, des Entwicklungsprozesses und der Nutzung eines rechnergestützten Systems bereitzustellen. Dadurch kann das System besser verstanden und die Arbeit von Administratoren, Entwicklern und Endbenutzern effizienter unterstützt werden. In einer Data Warehouse-Umgebung ist die Rolle von Metadaten von großer Bedeutung aufgrund der Komplexität und Funktionalität eines Data Warehouse.

In einem Data Warehouse wird eine Unmenge von Informationen aus mehreren heterogenen Quellen abgelegt. Mit der regelmäßigen Vermehrung von Informationen können Schwierigkeiten und Probleme bei der Entwicklung, Verwaltung und Nutzung des Data Warehouse entstehen. Dieses Problem kann mit Hilfe eines mächtigen Metadaten-Managements gelöst werden. In [SVV99] haben die Autoren folgende zwei Hauptziele des Metadaten-Managements hervorgehoben:

1. **Verringerung der Administrationstätigkeit eines Data Warehouse**

   Dies schließt folgende Punkte ein:

   - *Automatisierung der verschiedenen Administrationsprozesse für Informationssysteme.*
     Während der Ausführung der Prozesse (wie Laden und Auffrischen der Data Warehouse-Systeme) werden Metadaten automatisch zugegriffen und genutzt. Ihre Aufgabe ist es, diese Prozesse zu überwachen und einen Überblick über sie zu erhalten.

   - *Unterstützung der Systemsintegration.*
     Sowohl Schema- als auch Datenintegration basieren auf Metadaten, die Information über die Struktur und die Bedeutung des Quell- und Zielsystems enthalten. Zusätzlich sind auf Datenquellen angewandte Transformationsregeln erforderlich und können auch als Metadaten abgelegt werden.

   - *Durchsetzung der komplexen Sicherheitsmechanismen.*
     Zugriffskontrolle in einem Data Warehouse-System ist oftmals kompliziert und erfordert intelligente Methoden. Die meisten Probleme hängen mit der Datentrans-

formation zusammen, die den Wert der Daten im Data Warehouse im Vergleich mit den operationalen Quelldaten ändert. Beispielsweise sind manche operationale Quelldaten Geheiminformation, eine Aggregation dieser einzelnen Daten kann jedoch keine kritische Information sein und umgekehrt. Metadaten sollen daher die Zugriffsregeln und Nutzerrechte für das ganze Data Warehouse-System bereitstellen.

- *Unterstützung von Analyse und Entwurf neuer Anwendungen und Modellierung von Geschäftsprozessen.*
  Metadaten erhöhen die Kontrolle und die Zuverlässigkeit des Prozesses der Anwendungsentwicklung durch die Bereitstellung von Information über die Bedeutung der Daten, deren Struktur und Ursprung. Außerdem können Metadaten bezüglich der Entwurfsentscheidungen, die für vorhandene Anwendungen aufgenommen wurden, auch für neuere wiederverwendet oder aktualisiert werden.

- *Verbesserung der Flexibilität des Systems und der Wiederverwendung von existierenden Software-Module.*
  Semantische Aspekte, die voraussichtlich häufig zu verändern sind (sog. „Bussines-Regeln“), werden explizit als Metadaten außerhalb der Anwendungsprogramme gespeichert. Diese Business-Regeln können ohne große Anstrengungen in Übereinstimmung mit möglichen neuen Anforderungen aktualisiert werden.

2. Verbesserung der Datenextraktion aus dem Data Warehouse

Dazu gehören folgende Punkte:

- *Verbesserung der Datenqualität.*
  Datenqualität umfaßt Dimensionen wie Konsistenz, Vollständigkeit, Genauigkeit, Zeit-Variante und Korrektheit. Außerdem soll Daten-Historie unterstützt werden: Zeit der Erzeugung und der Autor der Daten, die Quelle der Daten, die Bedeutung der Daten etc.

- *Verbesserung der Abfrage-, Retrieval- und Antwortqualität.*
  Metadaten können den Aufwand der Benutzer beim Zugriff, Evaluieren und Nutzen der passenden Information reduzieren. Durch die Metadaten können Beziehungen zwischen Datenelementen entdeckt werden.

- *Verbesserung der Datenanalyse.*
  Metadaten sollen den Anwendungsbereich und seine Repräsentation im Datenmodell verstehen, um die Ergebnisse angemessen anzuwenden und zu interpretieren. Außerdem können Metadaten durch Analyseanwendungen automatisch zugegriffen und zur Durchführung ihrer spezifischer Berechnungen genutzt werden.

Die Verfügbarkeit eines Metadaten-Managements bringt noch andere Vorteile mit sich: Teilen von Wissen und Erfahrungen, Wiederverwendung der Fachkenntnisse, Eliminierung der Zweideutigkeit und Garantieren der Datenkonsistenz innerhalb des Unternehmens. Kurz gefaßt: Metadaten werden zur Schlüssellösung bei der Reduzierung sowohl technischer als auch geschäftlicher Probleme innerhalb eines Data Warehouse-Systems.

2.2 Klassifizierung von Metadaten

Allgemein werden Metadaten als Daten *über* Daten definiert, d.h. Daten, die andere Daten beschreiben. Wie alle anderen Daten, müssen Metadaten auch verwaltet und gepflegt werden und zwar typischerweise in einem Repository. Beim Etablieren der Architektur eines Metadaten-Repository spielt die Charakterisierung von Metadaten eine bedeutende Rolle. Obwohl es verschiedene Klassifizierungen gibt, beruhen sie meistens auf der Architektur oder auf den Hauptfunktionen des Data Warehouse (siehe [AM97], [Wie98], [Wie99], [SVV99]. In [SVV99] werden die Metadaten nach unterschiedlichen Kriterien klassifiziert.

1. **Kriterium „Typ“.**

   Es gibt zwei Typen von Metadaten:

   - *Metadaten für primäre Daten.*
     Primäre Daten bestehen aus allen Daten, die von den Datenquellen, dem Data Warehouse und den Anwendungen verwaltet werden. Die entsprechenden Metadaten schließen daher die Information bezüglich der Struktur solcher Bestände ein, wie z.B. Schemabeschreibung, statistische Werte, Anzahl der Einträge in der Datenbank etc.

   - *Daten-verarbeitende Metadaten.*
     Das sind Informationen, die sich auf die Datenverarbeitung beziehen, beispielsweise Informationen bezüglich des Laden- und Auffrischen-Prozesses, des Analyse-Prozesses und der Administration.

2. **Kriterium „Abstraktionsebene“.**

   Durch dieses Kriterium wird zwischen konzeptuellen, logischen und physikalischen Metadaten unterschieden. Die konzeptuelle Perspektive umfaßt die geschäftliche Beschreibung mittels der natürlichen Sprache. Logische Metadaten bilden die konzeptuelle Perspektive auf einer tieferen Ebene ab, wie z.B. das relationale Datenbankschema, die Beschreibung der Extraktions-/Transformationsregeln in Pseudocode. Die physikalische Perspektive ist mit der Implementationsebene verbunden. Sie enthält den entsprechenden SQL-Code von Business-Regeln, die Indexdateien der Relationen und den Code der Analyseanwendungen u.a.

3. **Kriterium „Herkunft“.**

   Metadaten können nach ihrer Herkunft unterschieden werden. Das sind z.B. die Werkzeuge, die sie produzieren (ETL-Komponenten, CASE-Tools u.a.), die Quellen, die sie bereitstellen (Data Dictionaries) oder das System, aus dem Metadaten importiert werden.

4. **Kriterium „Zweck“.**

   Durch dieses Kriterium können Metadaten entsprechend der Aktivitäten, wie z.B. Daten-Extraktion, oder –Transformation, Data Mining, Report etc., klassifiziert werden. Jedoch ist die Unterscheidung zwischen verschiedenen Klassen manchmal nur relativ, da manche Metadaten (z.B. Schemabeschreibung) für mehrere Zwecke verwendet werden können: Administration und Verwaltung, Auffrischen und Analyse.

5. **Kriterium „Benutzersicht“.**

   Die gleiche Information und Struktur kann aus unterschiedlichen Perspektiven betrachtet werden, abhängig von den sie gebrauchenden Benutzern. In diesem Zusammenhang wird zwischen Business-Metadaten und technischen Metadaten unterschieden. Entsprechend der Themenstellung der vorliegenden Arbeit soll diese Klassifizierung näher betrachtet werden (siehe auch [MSR99], [Wie98], [Whi99] u.a.).

   Technische Metadaten

   Technische Metadaten werden von den sogennanten technischen Benutzern (Datenbank-Administratoren, Entwicklern, Programmierern) verwendet und gepflegt. Sie werden normalerweise aus Copybook Libraries, DBMS-Katalogen bzw. Data Dictiona-

ries und unterschiedlicher Tools extrahiert. Technische Metadaten umfassen in der Regel Informationen über:

- Die Schemastruktur und der Inhalt der beteiligten operativen und dispositiven Systeme. Das umfaßt sowohl die Strukur der physikalisch gespeicherten Daten als auch die technische Beschreibung der zugehörigen Datenhaltungssysteme. Zu schematischen Metadaten gehören z.B. Informationen über Tabellen- oder Rekordstruktur, Constraints, Referenzbeziehungen, Triggers, etc. in verschiedenen Datenbank- und Dateisystemen.

- Abhängigkeiten und Abbildungen zwischen operativen und dispositiven Systemen. Das schließt z.B. Informationen über die an einer Abbildung beteiligten Quell- bzw. Zielsysteme sowie die zugehörigen Abbildungsregeln und Transformationscodes ein.

- Temporale Information und Benutzer-Aktionen („Wer hat was wann gemacht? Was ist wann passiert?).

Semantische Metadaten

Semantische oder Business-Metadaten werden vorrangig von (Business-) Endbenutzern (Manager, Analysten u.a.) gebraucht, die sich vor allem nur für den Business-Inhalt des Data Warehouse interessieren. Dazu gehören u.a.:

- Konzeptuelle Datenmodelle von operativen und dispositiven Systemen auf einer hohen Abstraktionsebene (z.B. Business-Konzepte, Beziehungen zwischen Business-Konzepten, Business-Regeln, Dimensionen, Fakten).

- Abhängigkeiten zwischen konzeptuellen Datenmodellen und physikalischen Daten (z.B. Ableitungsregeln der Business-Konzepte aus operationalen Datenmodellen).

Eine explizite Repräsentation von solchen semantischen Metadatentypen unterstützt die Endbenutzer bei den Navigations-, Analyse- und Anfrage-Tasks, um die Anwendungen und daher das Informationssystem besser zu verstehen, ohne daß sie mit den technischen Datenbeschreibungen oder einer Abfragensprache wie SQL vertraut sein müssen. Aufgrund der gewissermaßen ähnlichen Funktionalitäten stehen die semantischen Metadaten in einem engen Zusammenhang zu CASE-Tools und Werkzeugen der Künstlichen Intelligenz.

2.3 Funktionalitäten des Metadaten-Repository

Ein Schlüssel zum Erfolg eines jeden Data Warehouse-Projektes ist zweifellos ein mächtiges und durchgängiges Metadaten-Management. Eine wesentliche Anforderung an ein solches ist die Verwaltung der Metadaten in einem Repository. Das Repository ist der Mechanismus für die Definition, Speicherung und Verwaltung aller Informationen über ein Unternehmen, seine Daten und seine Softwaresysteme sowie den Zugriff darauf.

Im Abschnitt 2.1 wurden bereits die Rolle sowie die Ziele eines Metadaten-Managements erörtert. Um diese Ziele zu erreichen, hat das Metadaten-Repository bestimmte Funktionalitäten zu erfüllen: Bereitstellung von Information, Metamodell, automatischer Repository-Zugriff, Versions- und Konfigurationsmanagement, Wirkungsanalyse und Nutzerbenachrichtigungen (siehe auch [Ber93] und [SVV99]).

1. **Bereitstellung von Information.**

   Das Repository soll geeignete Mechanismen für Abfragen, Filterung, Navigation und Browsen anbieten, um die gespeicherten Informationen (Metadaten) den Benutzern bereitzustellen.

2. **Metamodell.**

   Ein Metamodell ist das konzeptuelle Schema eines Metadaten-Repository, in dem die Metadaten-Elemente und die Beziehungen zwischen ihnen festgelegt werden. Bei der Erstellung des Metamodells spielt die Klassifizierung der Metadaten eine bedeutende Rolle, weil dadurch z.B. die richtigen Elemente und Beziehungen entsprechend ihren Typen oder Abstraktionsebenen bestimmt werden können. Das Metamodell muß erweiterbar sein und sich fortlaufend an die Veränderung der Anwendungsanforderungen anpassen.

3. **Repository-Zugriff.**

   Für einen effizienten Zugriff von Benutzern auf das Metadaten-Repository werden geeignete Schnittstellen benötigt. Insbesondere sind Schnittstellen für Interoperabilität und API sehr relevant für den Metadaten-Austausch mit anderen Repositories bzw. Tools. Das führt zur Notwendigkeit, einen hersteller-übergreifenden Standard für den Austausch von Metadaten zu erarbeiten.

4. **Versions- und Konfigurationsmanagement.**

   Wichtige Änderungen von Metadaten fordern die Erzeugung unterschiedlicher Versionen und ihre Abspeicherung im Repository.

5. **Wirkungsanalyse.**

   Mögliche Veränderungen im Data Warehouse-System sollen durch den Administratoren vorher abgeschätzt werden. Beispielsweise können die Änderungen im Quellschema Folgen für Transformationsregeln haben.

6. **Benachrichtigungen.**

   Das Repository soll einen Benachrichtigungsmechanismus für sowohl interessierte Benutzer als auch Tools bereitstellen, z.B. über wichtige Veränderungen im Repository.

2.4 Metadaten-Standards

In diesem Abschnitt werden verschiedene Standards behandelt, die für Metadaten-Management, insbesondere in Data Warehouse-Umgebungen, relevant sind. Diese Metadaten-Standards werden demnächst wie folgt eingegliedert:

- Standards für Metadaten-Repräsentation
- Standards für Metadaten-Austausch
- Metamodell-Standards für Data Warehousing

Eine andere Klassifizierung von Metadaten-Standards wird auf [SVV99] verwiesen.

2.4.1 Standards für Metadaten-Repräsentation

MOF (Meta Object Facility)

MOF ist ein Repräsentations-Metastandard von OMG (Object Management Group). Dieser objektorientierte Standard definiert ein Metametamodell mit ausreichender Semantik für die Beschreibung der Metamodelle in verschiedenen Bereichen ([OMG97]). Der Hauptzweck von OMG MOF ist die Bereitstellung einer Menge von CORBA-Schnittstellen, die für die Definition und das Manipulieren einer Menge von Interoperabilitäts-Metamodellen genutzt werden können. MOF wird in die drei folgenden Paketen geordnet:

- Das *Modell-Paket* enthält die Hauptteile des MOF-Modells. Die abstrakte Klasse *ModelElement* klassifiziert die elementaren, atomaren Konstrukte des Modells. Sie ist in die Klassen *TypedElement*, *Feature*, *Tag*, *Constraint*, *Namespace* und *Import* unterteilt.

- Das *Reflective-Paket* enthält den MOF-Teil, der die Reflektion unterstützt, d.h. die Fähigkeit der Objekte, Informationen über ihre Struktur und Eigenschaften zu entdecken und anzuwenden.

- Das *Facility-Paket* enthält Modellelemente, die MOF-Modell ausnutzende Repositories, Tools, etc. unterstützen.

UML (Unified Modeling Language)

UML ist ein Repräsentations-Standard von OMG. UML ist eine Sprache zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme. Sie umfaßt drei Blocktypen ([BRJ99]):

- *Dinge* bestehen aus strukturellen Dingen (Klasse, Schnittstelle etc.), Verhaltens-Dingen (Nachrichten, Zustände), Gruppierungen (Pakete) und Kommentare (Notizen).

- *Beziehungen* bestehen aus Abhängigkeit, Assoziation, Generalisierung und Realisierung.

- *Diagramme* ermöglichen die graphischen Präsentationen von Dingen und ihren Beziehungen.

2.4.2 Standards für Metadaten-Austausch

MDIS (Meta Data Interchange Specification)

MDIS ist ein von MDC (Meta Data Coalition) entwickelter Austauschstandard. MDIS stellt ein Metamodell für die Haupttypen von Datenmodellen (relational, hierarchisch, Netzwerk) und einen Standard-Import/Export-Mechanismus für den Austausch dieser Metadaten-Objekte zwischen verschiedenen Tools bereit ([MDC97]). Durch folgende MDIS-Objekte können verschiedene Typen und Ebenen der Metadaten-Objekte bzw. ihre Beziehungen zueinander repräsentiert werden:

- *Database* repräsentiert relationale, objektorientierte, hierarchische, Netzwerk-Datenbank, Dateisystem.

- *Record* repräsentiert Tabelle, Klasse/Objekt, Segment.
- *Subschema* ist eine logische Gruppierung von Komponenten innerhalb einer Datenbank.
- *Element* repräsentiert Spalte/Kolumne, Attribut/Methode, Feld.
- *Relationship* stellt Beziehungen zwischen den Objekten dar. Es gibt neun Relationship-Typen: EQUIVALENT, DERIVED, INHERITS-FROM, CONTAINS, INCLUDES, LINK-TO, REDEFINES, GROUP-EQUIVALENT, USER-DEFINED.
- *Dimension/Level* beschreibt mehrdimensionale Datenbanken, wird jedoch noch nicht vollständig definiert.

Ein Nachteil von MDIS besteht darin, daß obwohl Abbildungen zwischen unterschiedlichen Datenbanken durch gegebene Relationship-Typen beschrieben werden können, werden jedoch Abbildungs-Regeln nicht als Meta-Klassen dargestellt, sondern nur als String abgespeichert.

XML (Extensible Markup Language)

XML (aufgenommen von W3C) wird zum Standard für die Repräsentation von Daten auf dem Web. XML ermöglicht es, die tatsächlich gebrauchten Tags im gegebenen Kontext zu definieren und Metadaten-Marken innerhalb eines Web-Dokuments einzubetten. Dadurch können manche Typen von Metadaten repräsentiert und ausgetauscht werden. Eine XML-Dokumentstruktur wird mittels einer Document Type Definition (DTD) definiert, die in die Webseite eingebunden oder durch einen Link zu einer anderen Seite extern gespeichert werden kann.

XMI (XML Metadata Interchange)

XMI wurde von IBM, Oracle, Unisys u.a. für Stream-based Model Interchange Request For Proposal (SMIF RFP) von OMG vorgeschlagen und ist ein Standard für Daten-Austausch via Internet ([O98/10]). XMI benutzt MOF als Metadatenmodell, der Austausch der eigentlichen Modelle basiert jedoch auf XML und deshalb kann es für den Metadaten-Austausch zwischen den nicht-CORBA-basierten Repositories und Tools genutzt werden. Der Vorschlag besteht aus:

- Einer Menge von XML DTD-Produktionsregeln für die Transformation der auf MOF basierten Metamodelle in die XML DTDs.
- Einer Menge von XML Document-Produktionsregeln für die Kodierung/ Dekodierung der auf MOF basierten Metadaten.
- Entwurfsprinzipien für XMI-basierte DTDs und XML-Strömungen.
- Konkrete DTDs für UML und MOF.

2.4.3 Metamodell-Standards für Data Warehousing

Obwohl eine Vielzahl von Standards im Bereich des allgemeinen Metadaten-Managements existiert, befindet sich die Standardisierung der Metadaten im Data Warehouse-Bereich noch in der Anfangsphase. Mit ihren Vorschlägen und Aktivitäten haben die Organisationen MDC und OMG einen bedeutenden Beitrag dazu geleistet. Ihre komplexen Metamodelle, OIM und CWM, werden demnächst diskutiert.

OIM (Open Information Model)

OIM, der Metamodell-Standard von MDC ([MDC99]), ist eine Hierarchie von Modellen, die bedeutende Bereiche von Information, Data Warehouse und Wissensmanagement in der integrierten Metadaten-Umgebung eines Unternehmens adressieren. Sie stellen eine allgemeine Menge von Metadaten-Typen bereit, die einen generischen Zugriff und Austausch von Metadaten ermöglichen:

- ***Analyse- und Entwurfs-Modell***: umfaßt den Bereich von objektorientierter Modellierung und Entwurf von Software-Systemen. Das Modell stellt Konzepte bereit, um Probleme und Lösungen im gesamten Software-Lebenszyklus zu beschreiben. Der Kern des Modells ist das UML-Metamodell.
- ***Objekt- und Komponent-Modell***: intendiert, verschiedene Aspekte einer objektorientierten Entwicklung zu umfassen.
- ***Business-Engineering-Modell***: stellt in Kombination mit dem UML-Modell alle notwendigen Metadaten-Typen bereit, um sowohl Ziele, Organisation und Infrastuktur eines Business als auch die Prozesse und die das Business regierenden Regeln zu beschreiben.

- ***Wissensmanagement-Modell***: stellt die notwendigen Metadaten-Typen zur Erzeugung der Katalogstrukturen von Unternehmensinformation und zur Erfassung der Business-Begriffe, ihrer semantischen Beziehungen und Abbildungen zu Speicherungsstrukturen bereit.

- ***Datenbank- und Warehousing-Modell***: stellt Metadaten-Konzepte für Datenbankentwurf, Schemawiederverwendung und Data Warehousing bereit. Dieses Untermodell enthält folgende Schemas:

  - Das *Database Schema* beschreibt Informationen über die in den relationalen Datenbanken verwalteten Daten einer Organisation (z.B. Tabellen, Sichten, Spalten, Constraints, Triggers etc.).

  - Das *Data Transformation Paket* beschreibt Datentransformationen für relational-zu-relational Translation. Innerhalb einer Transformation können Skripts, textuelle Beschreibungen oder Programmcodes gespeichert werden.

  - Das *OLAP Schema* beschreibt mehrdimensionale Datenbanken (z.B. Cubes, Dimensionen, Fakten, Aggregationen).

  - Das *Record Oriented Database Schema* beschreibt rekord-orientierte Information, d.h. Information über die z.B. in Dateien und Legacy-Datenbanken einer Organisation verwalteten Daten (z.B. Rekords, Gruppierungen, Felder).

CWM (Common Warehouse Model)

Im September 1998 hat OMG ein Request For Proposal (RFP) für „Common Warehouse Metadata Interchange“ (CWMI) ausgegeben (siehe [O98/9]). Es wird gefordert, eine vollständige Spezifikation von Syntax und Semantik für den Export/Import der Warehouse-Metadaten und das allgemeine Warehouse-Metamodell zu umfassen. Aufgrund dieses RFP wurde CWM gegenwärtig als die „initial submission“ etabliert, die zukünftig noch von beteiligten Anbietern überarbeitet und abgestimmt werden soll. CWM besteht aus Definitionen von Metadaten-Modellen aus folgenden Bereichen:

- ***CWM Foundation*** enthält Modellelemente, die gemeinsame Konzepte und Strukturen von anderen CWM-Untermodellen repräsentieren.

- ***Warehouse Deployment*** enthält Klassen, die erfassen, wie Data Warehouse-Tools auf verschiedenen Rechner-Plattformen installiert bzw. verteilt werden.

- ***Relational Data Sources*** beschreibt Daten, die durch eine relationale Schnittstelle wie ein RDBMS, ODBC oder JDBC zugreifbar sind.

- ***Record-oriented Data Source*** umfaßt die Basiskonzepte eines Rekords und seiner Struktur (z.B. Datenrekords von Dateien oder Datenbanken, Datentypen von Programmiersprachen).

- ***Multidimensional Data Source*** stellt eine generische Beschreibung von mehrdimensionalen Datenbanken bereit. Basiskonzepte sind *Dimension* und *Dimensioned Object*, die die physische mehrdimensionale Speicherung von Daten adressieren.

- ***XML Data Source*** enthält Typen und Assoziationen, die allgemeine, XML-Datenquellen beschreibende Metadaten repräsentieren.

- ***Data Transformations*** adressiert allgemeine, im Data Warehousing verwendete Transformations-Metadaten. Es umfaßt Basistransformationen aller Typen von Datenquellen und -zielen: objekt-orientiert, relational, rekord-orientiert, mehrdimensional, XML und OLAP.

- ***OLAP*** enthält wesentliche semantische OLAP-Konstrukte, die allgemein für die meisten OLAP-Tools sind, wie z.B. Dimension, Cube, Dimension Hierarchy, Member, etc.

- ***Warehouse Process*** dokumentiert den Prozessfluß, der für die Ausführung der Transformationen verwendet wird.

- ***Warehouse Operation*** adressiert die day-to-day Operation der Warehouse-Prozesse, die von drei getrennten Bereichen konstituiert wird: Transformations-Ausführung, Abmessungen, Änderungs-Anforderungen.

2.4.4 Weitere Ansätze

Neben den Standardisierungsaktivitäten von Hersteller-Koalitionen gibt es auch verschiedene Forschungen, die sich mit der Modellierung von Data Warehouse-Metadaten beschäftigen. Im folgenden werden die Metamodelle aus [Wie98] und [MSR99] kurz vorgestellt.

In [Wie98] definierte Wieken ein Metamodell, das spezifisch für das Repository Viasoft Rochade entwickelt wurde. Die Strukturierung dieses Metamodells enthält neben den fundamentalen Bereichen der operativen und dispositiven Systeme insbesondere die Metadaten-Strukturierung entsprechend den Data Warehouse-Prozessen: Access, Navigation. Akquisition und Preparation.

- ***Die Meta-Komponente Access*** dient der Abbildung der Metadaten, die für die Funktion „Abfrage“ bzw. „Delivery“ benötigt werden. Alle relevanten Informationsobjekte werden durch den entsprechenden Typen Business_Element repräsentiert, der auf einer Spalte/einem Attribut basieren oder über eine Ableitungsregel aus anderen Business Elementen abgeleitet werden kann.

- ***Die Meta-Komponente Navigation*** ermöglicht das Auffinden und die Nutzung von Informationen auf der Grundlage einer semantischen Hierarchie: Business_Area, Business_Directory, Business_Element.

- ***Die Meta-Komponenten Akquisition und Preparation*** enthalten die Informationen über den Zusammenhang zwischen dem operativen System und dem Data Warehouse bzw. den Data Marts. Der Abbildungsprozeß wird in zwei Ebenen beschrieben: der Feld/Attribut-Ebene und der Tabelle-Ebene.

In [MSR99] entwickelten Müller et al. ein einheitliches und integriertes Modell für Data Warehouse- Metadaten. Das Modell verwendet einen einheitlichen Repräsentierungsansatz (UML), um technische und semantische Metadaten und ihre gegenseitige Abhängigkeit zu integrieren.

- ***UML-Schema für technische Metadaten***: umfaßt zwei Typen von technischen Metadaten:

  - Die Beschreibung der Datenstruktur in operativen und dispositiven Systemen durch die Klassen *Data Store*, *DBMS Store*, *File Store*, *Schema*, *Entity*, *Attribute* u.a.

  - Die Beschreibung der Abhängigkeiten und Abbildungen zwischen operativen und dispositiven Systemen durch die Klassen *Mapping*, *Transformation*, *Aggregation*, *Filter*, *Function* u.a.

- ***UML-Schema für semantische Metadaten***: neben den gemeinsamen Klassen mit dem technischen Modell (*Entity*, *Attribute*, *Assoziation*, *Mapping*, *Transformation*, *Aggre-*

gation u.a.) stellt dieses Schema weitere Klassen für Repräsentation von Warehouse-bezogene Metadaten aus dem Business-Gesichtpunkt wie z.B. *Business-Concept*, *Data-Cube* u.a.

2.5 Kommerzielle Metadaten-Tools für Data Warehouse-Umgebungen

Für alle Komponenten eines Data Warehouse-Systems sind viele kommerzielle Tools verfügbar. Jedes dieser Tools ist ein Metadaten-Konsumer und/oder -Produzent. In [MSR99] werden die Metadaten-bezogenen Tools in einer Data Warehouse-Umgebung in vier Gruppen unterteilt

- ***ETL-Tools***: diese Gruppe von Tools richtet sich auf die Extraktion, Transformation und das Laden der Daten, und unterstützt daher hauptsächlich nur die technischen Metadaten. Dazu gehören z.B. Platinum Decision Base, Ardent DataStage, ETI Extract, Informatica Powermart.

- ***OLAP-Tools***: einige Tools wie MicroStrategy DSS Agent, Cognos Impromptu, Business Objects unterstützen mehrdimensionale Analysen (drill-down, roll-up) der Data Warehouse-Daten. Sie brauchen Schemainformationen, um Query/Reports zu erzeugen.

- ***CASE-Tools***: Modellierungstools, z.B. ERWin (Platinum), E/R Studio (Embarcadero), GDPro (Advanced Software) und Rose (Rational Software), beschreiben Business-Daten auf einer abstrakten semantischen Ebene, d.h. die Daten werden gewöhnlich durch ERM oder objektorientierte Ansätze wie UML modelliert. Mit Hilfe dieser Tools werden die Datenbankschemata für Data Warehouses und Data Marts generiert.

- ***Repository***: viele Hersteller (wie z.B. Microsoft, Platinum, Viasoft und Ardent) bieten Repository-Produkte für Metadaten-Management an.

Aufgrund der Vielzahl der Data Warehouse-Tools erfordern die Erzeugung und Verwaltung von Metadaten häufig große Anstrengungen. Eine Anforderung an ein effizientes Metadaten-Management ist die Integration von Metadaten zwischen unterschiedlichen Tools. Um das zu erreichen, entwickeln die Hersteller zahlreiche zentralisierte und verteilte Ansätze für Metadaten-Management. Nach White können diese Ansätze in die drei folgenden Kategorien eingeordnet werden ([Whi99]):

- ***Zentrale Repositories für Metadaten-Austausch***

  Repository-Produkte für Metadaten-Austausch existierten seit vielen Jahren für die Unterstützung von Entwicklungs-Tools und Anwendungen der Transaktions-Prozesse. Die meisten Hersteller erweiterten ihre Repository-Produkte, um Entscheidungsprozesse zu unterstützen. Beispiele für Anbieter und Produkte in dieser Kategorie sind Platinum (Platinum Repository), Softlab (Enabler), Unisys (UREP) und ViaSoft (Rochade).

  Ein wichtiges neues Produkt hier ist das Microsoft Repository und sein OIM. Microsoft und andere Tools der dritten Parteien tauschen Metadaten mit dem Repository aus, das eine COM-basierte Architektur verwendet. Dateien, die in Microsoft's XIF definiert sind, können auch für den Import/Export von Repository-Metadaten genutzt werden. OIM ist eine Menge von UML-Metadaten-Modellen, die den Tools ermöglichen, ein gemeinsames Verständnis von den im Repository gespeicherten Metadaten zu haben.

- ***Von Hersteller-Koalitionen definierte Metadaten-Standards***

  Solche Standards erfordern, daß sich die Koalitionsmitglieder auf ein gemeinsames Sub-Modell für den Metadaten-Austausch zwischen ihren Tools einigen. Jeder Hersteller kann zusätzliche Metadaten für seine eigenen Produkte festlegen.

  Das Standard MDIS von MDC (siehe Abschnitt 2.4.2) wird von den Herstellern nicht viel unterstützt, weil es heute bei weitem noch nicht für den Austausch aller wichtigen Metadaten ausreicht. Als wichtige Metamodell-Standards für Data Warehousing gelten gegenwärtig das OIM von MDC und das CWM von OMG (siehe Abschnitt 2.4.3).

- ***Hersteller-spezifische „offene" APIs für Metadaten-Austausch***

  In diesem Ansatz bieten die Hersteller "offene" APIs an, die anderen Parteien ermöglichen, Metadaten aus ihren Produkten zu importieren und/oder exportieren. Beispiele von solchen Hersteller-Schnittstellen sind IBM Meta Data Interchange Language, Informatica Metadata Exchange (MX/MX2) und ETI Meta Data Exchange (MDX).

Obwohl diese drei Ansätze eine Lösung für das Problem der Metadaten-Intergration in Data Warehouse-Umgebungen anbieten, hat der Standardisierungs-Ansatz eine bessere Chance auf Erfolg gegenüber anderen Ansätzen. Gründe dafür sind u.a.: einheitliche Repräsentation von Metadaten, Erweiterbarkeit für Tool-spezifische Metadaten, verlustfreier Metadaten-Austausch zwischen Tools, etc.

3 Konzeptioneller Entwurf des Repository für technische Metadaten

Dieses Kapitel befaßt sich mit der Modellierung eines Repository für technische Metadaten. Ein technisches Metadaten-Repository soll in der Regel Informationen über die Architektur der beteiligten Datenhaltungssysteme, Abbildungsdefinitionen zwischen Quell- und Zielsystemen und zusätzlich auch temporale und Benutzer-Informationen enthalten. Es muß zunächst ein Metadaten-Modell mit Hilfe einer Modellierungssprache entwickelt werden, um die relevanten Objekte und Beziehungen zwischen ihnen darzustellen (Abschnitt 3.2). Anschließend wird eine Umsetzung dieses Modells in das Datenmodell der zugrundeliegenden Repository-Datenbank vorgenommen (Abschnitt 3.3). Ein Standard auf der Metamodell-Ebene, erwähnt im Abschnitt 2.4 über die Metadaten-Standards, UML, wird für die semantische Modellierung verwendet. Aufgrund der zunehmenden Bedeutung der objektrelationalen Datenbanksysteme wird Informix Universal Server Version 9.1 als das Datenbanksystem für das Repository ausgewählt. Deshalb soll das erstellte UML-Metamodell anschließend in ein objektrelationales Modell transformiert werden.

3.1 UML als Modellierungsansatz

In diesem Abschnitt wird UML als Modellierungsansatz für das Metadaten-Modell vorgestellt. UML (Unified Modeling Language) ist eine standardisierte, visualisierte objektorientierte Modellierungssprache. Sie wurde von den drei „Amigos“ Booch, Rumbaugh und Jacobson als eine Kombination unterschiedlicher Modellierungskonzepte erarbeitet ([BRJ99]):

- Datenmodellierungskonzepte (Entity Relationship Diagramme)
- Business-Modellierung (work flow)
- Objektmodellierung
- Komponentmodellierung

Aufgrund ihrer Modellierungsmächtigkeit wird UML von vielen Herstellern genutzt, um das Metamodell ihres Repository zu modellieren. Beispielsweise basieren Microsoft OIM, Informatica MX2 u.a. auf UML.

UML enthält eine Vielzahl von Modellelementen, die in den folgenden graphischen Diagrammtypen verwendet werden:

- ***Klassendiagramm***: zeigt Klassen und ihre Beziehungen untereinander.
- ***Anwendungsfalldiagramm***: beschreibt eine begrenzte Arbeitssituation im Anwendungsbereich.
- ***Zustandsdiagramm***: zeigt Zustände, Zustandsübergänge und Ereignisse.
- ***Aktivitätsdiagramm***: eine spezielle Form des Zustandsdiagramms, das überwiegend Aktivitäten enthält.
- ***Interaktionsdiagramm***: zeigt Objekte und ihre Beziehungen inkl. ihres Nachrichtenaustausches.
- ***Komponentendiagramm***: zeigt die Organisation und Abhängigkeiten von Komponenten (Softwaremodule, Pakete).
- ***Verteilungsdiagramm***: zeigt die Konfiguration der Knoten und ihrer Komponenten, Prozesse und Objekte.

Zur Repräsentation des technischen Metadaten-Modells wird das Klassendiagramm verwendet. Andere Diagramme werden hinsichtlich des Zweckes dieser Arbeit nicht gebraucht. In einem Klassendiagramm gibt es folgende Modellierungselemente (siehe auch Tabelle 1):

Klassen

Eine *Klasse* ist eine Sammlung von Objekten mit gemeinsamer Struktur, Verhalten, Beziehungen und Semantik. Das Klassensymbol ist ein Rechteck mit drei Fächern, das die Angabe vom **Klassennamen** (fettgedruckt), von Attributen bzw. zugehörigen Datentypen (optional) und von Methoden (optional) enthält. Die Struktur einer Klasse wird durch ihre Attribute und ihr Verhalten durch die Operationen/Methoden repräsentiert.

Beziehungen und Kardinalität bzw. Rollennamen

Die Beziehungen stellen den Pfad für die Kommunikation zwischen den Objekten dar. Es gibt im allgemeinen folgende Beziehungen: *Assoziation, Aggregation/Komposition, Abhängigkeit* und *Vererbung*, die jeweils durch verschiedene Linienarten modelliert werden.

Eine *Assoziation* beschreibt eine Relation zwischen den Klassen und ist als eine die Klassen verbindende Linie anzusehen. Es werden *gerichtete Assoziationen* (nur einseitig direkt navigierbar) und *bidirektionale Assoziationen* (beidseitig direkt navigierbar) unterschieden. Die beiden Enden einer Assoziation sind *Assoziationsrollen*, die die Klassen in derjenigen Assoziation spielen. Außerdem werden auf beiden Assoziationsenden auch die *Kardinalität*, d.h. die Anzahl der an der Assoziation beteiligten Elemente, angegeben. Für die Angabe von Kardinalitätsrestiktionen wird folgende Notation verwendet:

x.. y	mindestens x, maximal y Objekte nehmen an der Beziehung teil
*	„viele“
1	genau 1

Eine *Aggregation* ist eine Sonderform der Assoziation, nämlich eine Ganzes-Teile-Beziehung zwischen den beteiligten Klassen. Eine Aggregation ist als eine Linie mit einer hohlen Raute auf die das Ganze repräsentierende Klasse zu sehen. Wenn die Teile vom Ganzen existenzabhängig sind, wird die Aggregation eine *Komposition* mit dem Symbol einer Linie mit einer gefüllten Raute.

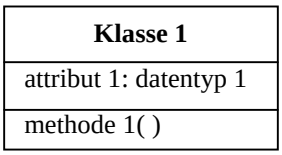
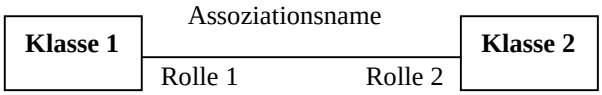
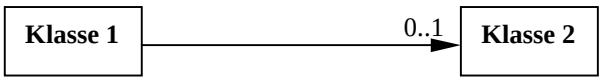
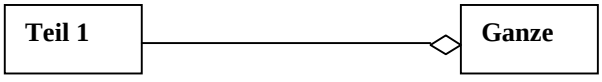
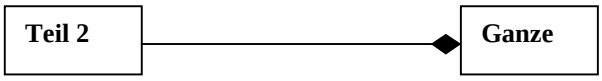
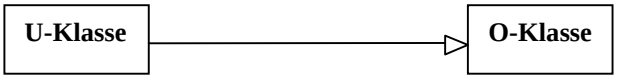
Element	Symbol
Klasse	
(bidirektionale) Assoziation	
gerichtete Assoziation	
Aggregation	
Komposition	
Vererbung	

Tabelle 1: Darstellung von Klassen und Beziehungen in UML

Zwischen einer Oberklasse und ihren Unterklassen existiert eine *Vererbungsbeziehung*, das heißt, die Unterklassen können die Eigenschaften der Oberklasse nehmen. Das Symbol der Vererbung ist eine Linie mit einem hohlen Dreieck auf der Oberklasse.

Für weitergehende Informationen über UML bzw. objektorientierte Modellierung wird auf [BRJ99], [Bur97], [FS98], [Neu98] und [Oes98] verwiesen.

3.2 Das UML-Modell für technische Metadaten

Bei der Erstellung eines Modells müssen vor allem die Hauptobjekte dieses Modells bestimmt werden. Für das Metamodell der vorliegenden Arbeit sind insbesondere solche Objekte relevant, die für die Beschreibung der Datenstruktur und der Abhängigkeit der Datenhaltungssysteme verwendet werden. Außerdem soll dieses Modell die Abbildungen zwischen den Systemen möglichst detailliert beschreiben können, sodaß das Repository diese Abbildungen aufgrund deren Definition implementieren könnte. Um alle Objekte des Modells und ihre Beziehungen zueinander übersichtlich und vollständig darzustellen, wird es in drei zusammenhängende Schemata unterteilt:

- Das Gesamt-Schema,
- Das Transformations-Schema und
- Das Filter-Schema

3.2.1 Das Gesamt-Schema

Im **Gesamt-Schema** sind die Hauptklassen enthalten, die zur Abbildung der Struktur der beteiligten operativen und dispositiven Datenhaltungssysteme und ihrer Abhängigkeiten insbesondere bezüglich der Datenbewegung benötigt werden. Das sind u.a. die Klassen *Data_Store, Schema, File_Structure, Database, File, Record, Element, Mapping_Folder, Mapping, Filter* und *Transformation* (siehe Abbildung 3-1).

Klasse Data_Store

Die Daten in einer Data Warehouse-Umgebung werden normalerweise in den Datenbanken oder den Dateien abgelegt. Die wichtigsten Quell-/Ziel-Datenhaltungssysteme sind daher Datenbanksysteme und Dateisysteme. Eine neue, „obere“ Klasse wird definiert, die diese beiden Arten umfassen und sie verallgemeinern kann. Die Klasse **Data_Store** repräsentiert also entweder ein System bzw. eine Gruppe von Dateien, die unterschiedliche Strukturen

haben können, oder ein Datenbank-Managementsystem. Im **Gesamt-Schema** werden die Beziehungen der Klasse **Data_Store** zu den Klassen **Database** und **File** durch zwei Assoziationen dargestellt: ein Data Store repräsentiert entweder eine Instanz von Database oder mehrere Instanzen von File. Zur Identifizierung eines Data Store wird das Attribut *name* der Klasse zugewiesen, dessen Wert eindeutig sein muß. Ein weiteres Attribut, *aliasname*, kann für eine genauere Beschreibung des Data Store gebraucht werden[1].

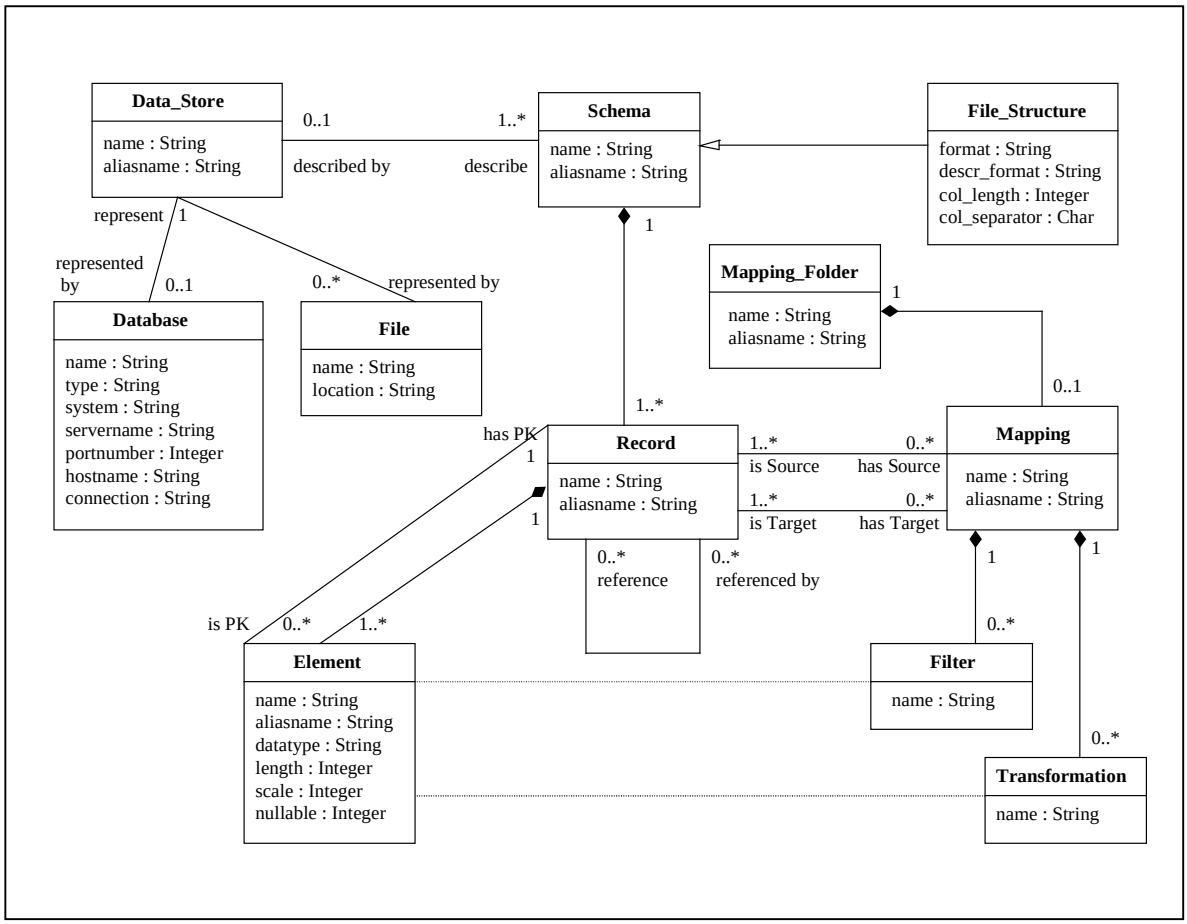


Abbildung 3-1: Das Gesamt-Schema (1)

Klasse File

Die Klasse **File** repräsentiert eine Datei als ein physikalisches Objekt. Die Dateien können unterschiedliche Formate haben. Ihre Struktur kann durch ein Beschreibungsformat, wie z.B. ein Cobol Copybook[2], beschrieben werden. Die logische Struktur einer Datei wird dennoch durch eine andere Klasse, nämlich die Klasse **File_Structure**, dargestellt. Um

[1] Die Festlegung der Klassenattribute ist flexibel und die Attributenmenge ist erweiterbar und veränderbar.
[2] Ein Cobol Copybook ist ein Dateistruktur-Deskritor, der von Cobol-Programme zur Interpretation der flachen Dateien genutzt wird

eine Datei identifizieren zu können, braucht man das Attribut *name,* was aber nicht identisch mit dem originalen Dateinamen sein muß, und das Attribut *location*, was besagt, wo die Datei liegt.

Klasse Database

Eine Instanz von Database kann ein relationales, objektorientiertes, hierarchisches Datenbanksystem sein. Da ein Data Store nur ein Database repräsentiert, kann der Name des Database identisch mit dem Namen des entsprechenden Data Store sein. Bei der Implementierung spielt das Attribut *name* sowohl die Rolle des Primärschlüssels, als auch die des Fremdschlüssels (Name von Data Store). Andere relevante Informationen, insbesondere bezüglich des Zugriffes auf die Datenbank, sollen aus folgenden Attributen entnommen werden:

- *type*: Typ des Database, Werte: “RDB”, “OODB”, “HDB”, “NWDB”
- *system*: Name und Version des DBMS
- *servername*: Name von Server oder Rechner, auf dem das Database abgelegt wird
- *portnumber*: Identität zur Kommunikation mit dem Server
- *hostname*: Name des Datenbankservers
- *connection*: Art und Weise des Zugriffes aufs Database, z.B. über native API, über ODBC oder andere Middleware-Standards

Klasse Schema

Während die Klassen **Data_Store**, **Database** und **File** physikalische Objekte repräsentieren, läßt sich die logische Organisation bzw. Struktur dieser Objekte durch die beiden weiteren Klassen **Schema** und **File_Structure** darstellen. Ein Data Store kann z.B. ein oder mehrere Schemas referenzieren. Jedes Schema ist eine logische Gruppierung der Strukturen von Tabellen, von Segmenten oder einer Gruppe von Dateien. Es können „reine“ logische Strukturen sein und deshalb auch nicht-physikalische Daten beschrieben werden. In diesem Fall ist das Schema nicht an ein Data Store gebunden. Bei der Assoziation zwischen beiden Klassen im **Gesamt-Schema** beträgt die Kardinalität der Klasse **Data_Store** 0..1 und der Klasse **Schema** 1..*. Die Attribute sind dieselben wie die des Data Store (*name* und *aliasname*).

Klasse File_Structure

Wenn ein Data Store ein Dateisystem ist, muß man mehr über seine Struktur erfahren, um auf die gespeicherten Daten zugreifen zu können. Die Klasse **File_Structure** soll eine Unterklasse der Klasse **Schema** sein, d.h. ein Dateisystem kann durch eine oder auch mehrere Instanzen von File_Structure beschrieben werden. Dazu braucht man vor allem Informationen über die Dateiformate (Attribut *format*), z.B.: flat file, ASCII file, VSAM, ISAM. Viele Dateien benötigen ein besonderes Format zur Beschreibung der Dateistruktur (Attribut *desc_format*) wie ein Cobol Copybook. Die Struktur einer Datei kann nicht nur aus ihrem Format entnommen werden. Falls die Dateien kein bestimmtes Format oder Beschreibungsformat haben, müssen ihre Spalten durch ein besonderes Merkmal charakterisiert werden, damit sie voneinander unterschieden werden können. Für die Dateien mit einer festen Spalten-Länge ist diese Länge (Attribut *col_length*) erforderlich. Andernfalls müssen die Spalten durch ein Zeichen, wie z.B. Komma, Semikolon, Tabulator, getrennt werden (Attribut *col_separator*).

Klasse Record

Die Klasse *Record* repräsentiert:

- Rekord-Layout einer Datei
- Struktur einer relationalen Tabelle
- Definition einer Klasse in einer objektorientierten Datenbank
- Segmentstruktur innerhalb einer hierarchischen Datenbank
- Rekordstruktur innerhalb einer Netzwerk-Datenbank

Mehrere Instanzen der Klasse **Record** können in einem logischen Schema zusammengefaßt werden. Andererseits enthält ein Record wiederum eine oder mehrere Instanzen der Klasse **Element** und eine oder einige von denen können Primärschlüssel oder Schlüsselkandidaten der Record-Instanz sein. Außerdem existieren zwischen den Instanzen der Klasse **Record** auch Beziehungen zueinander: ein Record kann andere Records referenzieren und/oder wieder von anderen Records referenziert werden. Ein referenzierender Record hat Fremdschlüsselkandidaten, die Primärschlüsselkandidaten in den referenzierten Records sind und umgekehrt.

Jede Record-Instanz hat einen Namen zur Identifizierung (Attribut *name*), der innerhalb eines Schemas eindeutig sein soll. Der genaueren Beschreibung des Records dient das Attribut *aliasname*.

Klasse Element

Zwischen den Klassen **Record** und **Element** gibt es eine Kompositionsbeziehung, d.h., mehrere Instanzen der letzteren Klasse sind in einer Instanz der ersteren Klasse enthalten und sind existenzabhängig von dieser. Wenn z.B. eine Tabelle einer relationalen Datenbank gelöscht wird, werden naturgemäß die Spalten/Attribute dieser Tabelle auch gelöscht.

Entsprechend der von einer Record-Instanz beschriebenen Objektstruktur gibt es „Unter"-Objekte, deren Struktur von einer Element-Instanz beschrieben werden kann. Also repräsentiert die Klasse **Element**:

- Felder innerhalb eines Dateirekords
- Spalten innerhalb einer relationalen Tabelle
- Attribute oder Methoden einer Klasse
- Felder innerhalb eines hierarchischen Segments
- Members innerhalb einer Dimension

Neben den zwei Attributen *name* und *aliasname* besitzt die Klasse **Element** noch weitere spezifische Attribute. Eine Element-Instanz hat normalerweise einen Datentyp (Attribut *datatype*), wie z.B. String, Integer, Numeric etc. Informationen über die Länge des Attributwertes, d.h. die gesamte Anzahl der Zeichen bzw. der Ziffer, und im Fall einer numerischen Zahl auch die Anzahl der Ziffer nach dem Dezimalpunkt (Attribute *length* und *scale*) sind ebenfalls wichtig. Das Attribut *nullable* gibt an, ob das Element einen Nullwert haben kann.

Die Klassen User und Privileg

Das Metadaten-Management soll auch Information über Benutzer bzw. ihre Zugriffsrechte enthalten. Für die Repräsentation der Benutzer wird die Klasse **User** gebildet (siehe Abbildung 3-2). Jeder User wird durch seinen (Login-)Namen (Attribut *name*) und seinen Gruppennamen (Attribut *group*) charakterisiert. Er kann über verschiedene Zugriffsrechte auf mehrere Instanzen von Record und/oder auf mehrere Instanzen von Element verfügen. Umgekehrt kann eine Record- bzw. Element-Instanz von mehreren Benutzern zugegriffen

werden. In manchen Fällen darf ein Benutzer z.B. nur lesend auf eine Tabelle, auf einige Spalten dieser Tabelle aber ändernd und/oder löschend zugreifen. Da die Assoziation zwischen der Klasse **User** und der Klasse **Record** bzw. der Klasse **Element** eigene Attribute hat, wird diese Assoziation durch eine eigene Klasse dargestellt, die Privileg genannt wird. Die Klasse **Privileg** hat das Attribut *name*, das Name des Zugriffsrechts (z.B.: read, write, delete, etc.) bedeutet.

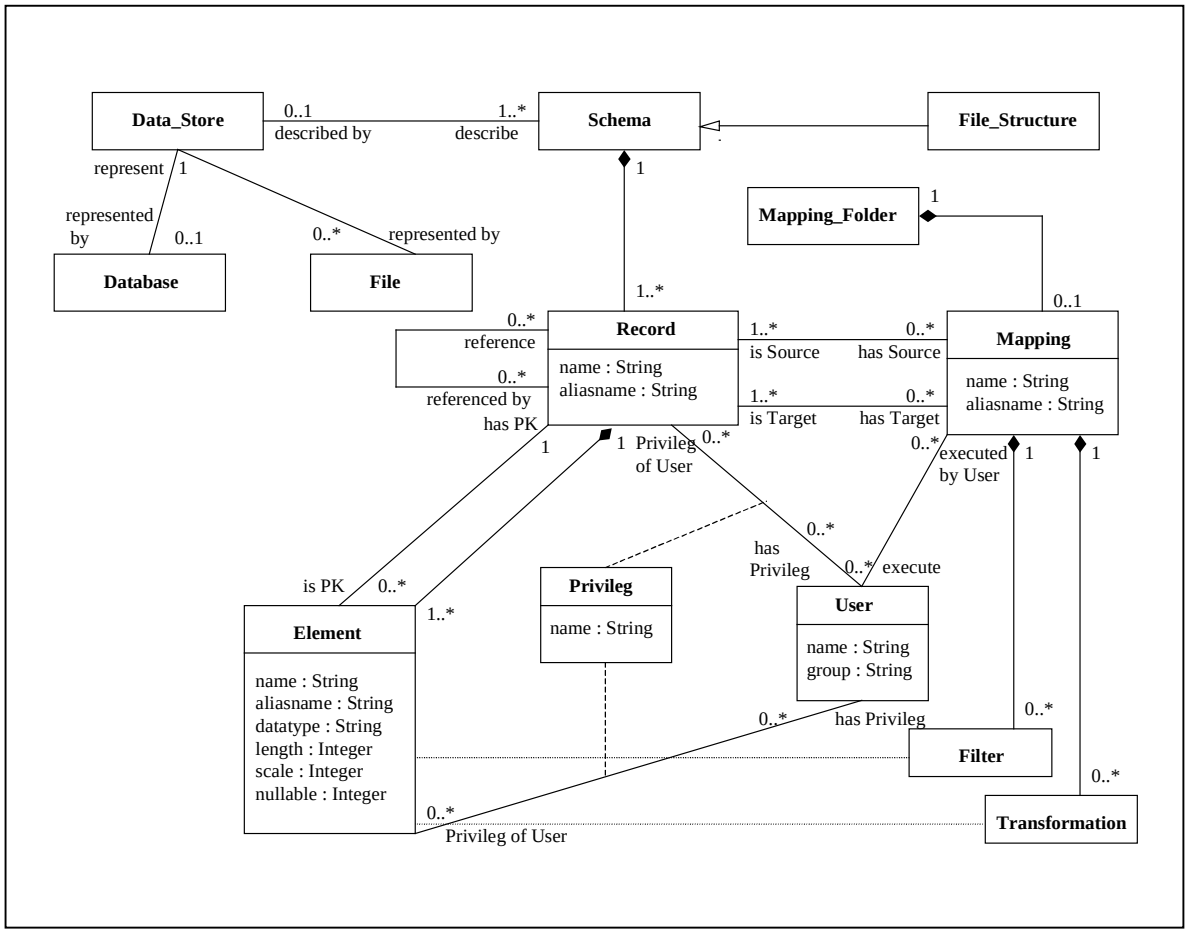


Abbildung 3-2: Das Gesamt-Schema (2)

Die Klassen Mapping_Folder und Mapping

Mittels der obigen Klassen des Metamodells kann die Struktur der Quell- bzw. Zielsysteme einschließlich der Benutzerinformation beschrieben werden. Ein Repository für technische Metadaten hat noch die Aufgabe, den ETL-Prozeß eines Data Warehouse zu verkörpern. Innerhalb dieses Prozesses werden die Daten in einen konsistenten Zustand im Data Warehouse/Data Mart überführt. Neben der Selektion der gewünschten Datenbereiche sollen ausreichende Transformationsmöglichkeiten gegeben werden. Hierzu gehören beispiels-

weise die Definition von Transformationsregeln auf Datenfeldebene durch komplexe String-, Datums- oder Berechnungsfunktionen wie z.B. Aggregationen oder Einschränkungen der Datenmenge durch Anwendung von Filtern. Man kann die Abbildungen zwischen den Quell- und Zielsystemen in zwei Ebenen unterteilen: die Record-Ebene und die Element-Ebene. Um diese Abbildungen inklusive aller relevanten Filter- und Datenkonvertierungs-Regeln darstellen zu können, werden noch weitere Klassen benötigt.

Die erste relevante Klasse ist die Klasse **Mapping**. Sie präsentiert die Abbildungen bzw. Konvertierungen auf der Record-Ebene. Die an einem Mapping beteiligten Records können die Rolle einer Quelle (Source-Record) oder eines Ziels (Target-Record) spielen, abhängig davon, ob sie die Quelldaten enthalten oder durch das Mapping erzeugte neue Daten speichern. Ein Mapping wird nur dann definiert, wenn daran mindestens ein Source- und ein Target-Record teilnehmen.

Im ETL-Prozeß sollen im allgemeinen zahlreiche Mappings durchgeführt werden. Um eine Übersicht von ihnen zu schaffen und sie besser anzuordnen, können mehrere Mapping-Instanzen in einem Folder gehalten werden. Ein Mapping-Folder ist eine Sammlung aller Abbildungen bzw. Konvertierungen, deren Zieldaten zu einer Einheit zusammengefaßt werden können, z.B. einem Data Mart, oder auch nur etwas Gemeinsames haben. Beide Klassen **Mapping_Folder** und **Mapping** haben auch die Attribute *name* (eindeutig) und *aliasname*.

Zwischen den Klassen **Mapping** und **User** (siehe Abbildung 3-2) existiert auch eine Assoziation, die besagt, welcher Benutzer welches Mapping durchführen kann.

Während die Klasse **Mapping** die Abbildung auf der Record-Ebene repräsentiert, müssen zusätzliche Klassen in das Metamodell hinzugefügt werden, um verschiedene Transformationsmöglichkeiten auf der Element-Ebene darstellen zu können. Im allgemeinen kann zwischen zwei Arten von Datentransformation unterschieden werden: Transformation ohne und mit Änderung der Anzahl der Datensätze. Entsprechend dieser Unterscheidung werden die Abbildungen auf der Element-Ebene in zwei unterschiedliche Klassen geordnet: die Klasse **Transformation** für die erste Art und die Klasse **Filter** für die zweite Art. Diese Klassen und weitere mit ihnen zusammenhängende Klassen sollen im weiteren ausführlich behandelt werden.

3.2.2 Das Transformations-Schema

In diesem Abschnitt werden die Klasse **Transformation** und andere Klassen zur Beschreibung einer Transformation diskutiert. Eine ***Transformation*** ist eine Abbildung der Quelldaten in den Zieldaten auf der Element-Ebene, die durch einen mathematischen Ausdruck dargestellt werden kann. Das kann ein arithmetischer Ausdruck aus numerischen Zahlen und/oder den Werten der an der Transformation beteiligten Elemente sein. Häufig wird im System eine Bibliothek von zahlreichen String-, Datum- und Berechnungsfunktionen vordefiniert. Mehrere solche Funktionen können in einem Ausdruck vorkommen.

An einer Transformation können mehrere Quell-Elemente teilnehmen, die im Transformationsausdruck als seine Glieder oder Funktionsparameter vorkommen. Das Ziel-Element nimmt den Wert dieses Ausdruckes. Die Beziehungen zwischen den beiden Klassen **Transformation** und **Element** lassen sich durch zwei Assoziationen darstellen: eine mit den Rollen *„hat Quell-Element“* und *„ist Quell-Element“* und eine mit den Rollen *„hat Ziel-Element“* und *„ist Ziel-Element“*. Die entsprechenden Kardinalitäten sind 1 : 1..* und 1 : 1, das bedeutet: eine Transformation hat nur ein Ziel-Element, aber kann mehrere Quell-Elemente haben. Es ist zu beachten, daß die Anzahl der Datensätze vor und nach einer Transformation unverändert bleibt.

Eine Transformation kann also mittels eines arithmetischen Ausdruckes einschließlich vordefinierter Funktionen dargestellt werden. Dabei entsteht noch ein Problem: wie wird dieser Ausdruck interpretiert? Das Repository soll in der Lage sein, aufgrund der Interpretation eines Transformationsausdruckes diesen zu implementieren. Normalerweise wird ein Ausdruck von links nach rechts und von innen nach außen (bei Klammern) unter Berücksichtigung der Vorrangreihenfolge der Operatoren interpretiert. Der Anfangsausdruck wird infolgedessen in zwei Teilausdrücke geteilt, die durch einen arithmetischen Operator verbunden sind. Dieser Vorgang wird für jeden Teilausdruck fortgeführt, bis er nur noch ein elementarer Ausdruck (eine numerische Zahl, ein Element-Wert oder eine Funktion) bleibt.

Die **Transformations-Grammatik** kann in Backus-Naur Form (BNF) wie folgt beschrieben werden:

- `Transformation ::= Expression`
- `Expression ::= Expression bin_operator Expression`
- `Expression ::= Function | Element | num_literal`
- `bin_operator ::= + | - | * | /`

- `Function ::= name ( Parameterlist )`
- `Parameterlist ::= Parameter [, Parameter]`
- `Parameter ::= Element | num_literal | str_literal`

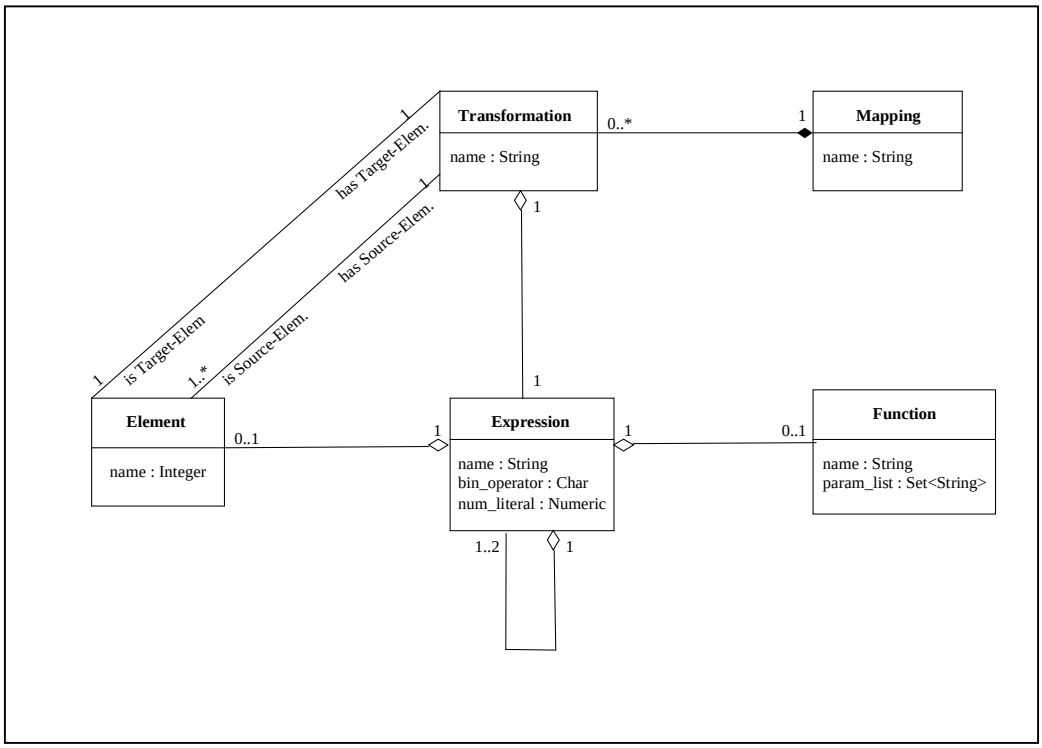


Abbildung 3-3: Das Transformations-Schema

Diese Grammatik kann als ein UML-Schema (Abbildung 3-3) beschrieben werden, das alle mit einer Transformation im Zusammenhang stehenden Klassen enthält: Klasse **Mapping**, Klasse **Transformation**, Klasse **Expression**, Klasse **Function** und Klasse **Element**. Anhand der BNF-Grammatik und des Transformation-Schemas kann der Zusammenhang zwischen ihnen wie folgt interpretiert werden:

- Ein *Mapping* kann keine oder viele *Transformationen* (als Instanzen der Klasse Transformation) enthalten. Dies wird durch eine Kompositionsbeziehung mit der Kardinalität 1 : 0..* dargestellt, weil wenn z.B. eine Mapping-Instanz gelöscht wird, existieren die zugehörigen Transformations-Instanzen auch nicht mehr.

- Eine *Transformation* wird als eine *Expression* dargestellt, d.h. eine Transformations-Instanz enthält genau eine Instanz der Klasse *Expression.* Zwischen den beiden Klassen besteht eine Aggregationsbeziehung mit der Kardinalität 1 : 1.

Bemerkung: In diesem Fall ist die Beziehung nicht eine Komposition, sondern eine Aggregation, weil man davon ausgeht, daß wenn die Transformation nicht mehr da ist, die Expression noch bleiben und z.B. für andere Transformationen genutzt werden kann. Das gilt auch für die Beziehungen zwischen anderen Klassen in diesem Schema.

- Eine *Expression* besteht normalerweise aus zwei anderen *Expressions*, die durch einen binären Operator (+, -, *, /) miteinander verbunden sind. Es existiert eine Aggregationsbeziehung der Klasse Expression zu sich selber mit der Kardinalität 1 : 1..2, wobei 1 dem Fall entspricht, wenn die Expression nur eine elementare Expression enthält. Eine elementare Expression kann eine numerische Zahl (*num_literal*), ein Elementwert (*Element*) oder eine Funktion (*Function*) sein.

  Demzufolge hat die Klasse **Expression** zwei relevante Attribute: *bin_operator* und *num_literal*. Außerdem hat sie jeweils eine Aggregationsbeziehung (Kardinalität 1 : 0..1) zur Klasse **Element** und Klasse **Function**, die beschreibt, ob die (elementare) Expression ein Element oder eine Function ist.

- Eine *Function* wird durch einen *name* und eine aus *Parametern* bestehenden *Parameterlist* charakterisiert.

- Ein *Parameter* ist entweder ein Elementwert oder eine Zeichenkette, oder eine numerische Zahl. Um zu verallgemeinern, vereinbaren wir den Datentyp des Attributs *param_list* der Klasse **Function** als eine Menge von Zeichenketten (Set<String>), die die Rolle der Parameter spielen.

- Rangordnung:
  - Klammern haben den höchsten Rang, dann * und /, schließlich + und -;
  - Bei den Operatoren mit gleichem Rang: links assoziativ.

Beispiele:

(1) Der Umsatz eines Produkts wird wie folgt berechnet:

```
Umsatz = (VerkaufsMenge – ZurückMenge) * Preis
```

Dieses Mapping wird durch einen Transformations-Ausdruck beschrieben, der aus zwei, durch den Multiplikations-Operator verbundenen Teil-Ausdrücken besteht:

```
Expression1 = VerkaufsMenge – ZurückMenge
Expression2 = Preis                    (elementarer Ausdruck)
```

Der Ausdruck *Expression1* enthält wiederum zwei elementare Ausdrücke, die durch die Subtraktion verbunden sind.

(2) Der vollständige Name der Kunden setzt sich aus dem Vornamen und Nachnamen zusammen. Dieses Mapping wird durch die vordefinierte Funktion CONCAT durchgeführt, die zwei Strings zu einem neuen String verknüpft:

```
Name = CONCAT (Vorname, Nachname)
```

Der Transformations-Ausdruck ist also eine *Function*, deren Parameter die Elemente Vorname und Nachname sind.

3.2.3 Das Filter-Schema

Während die Anzahl der Datensätze vor und nach einer Transformation unverändert bleibt, werden in vielen Fällen nicht alle Datensätze eines Quell-Elements, sondern nur welche, die bestimmte Bedingungen erfüllen, zum Zielsystem transportiert. Die Filterung der Quelldaten innerhalb eines Extraktionsprozesses wird durch die Klasse Filter repräsentiert. Ein ***Filter*** ist eine Abbildung der Quelldaten in den Zieldaten auf der Element-Ebene, die durch einen SQL-Ausdruck dargestellt werden kann. Ein Mapping kann mehrere Filter enthalten.

Die Struktur eines SQL-Ausdruckes besteht normalerweise aus drei bzw. vier Klauseln: der Select-, From-/Into-, und Where-Klausel. Jede dieser Klauseln entspricht einer Teilaufgabe eines Filters:

- *Select-Klausel* beschreibt (einfache) Abbildungen zwischen Quell- und Ziel-Elementen und/oder die Berechnungen der Aggregatfunktionen.
- Die Tabellen in *From-* bzw. *Into-Klausel* entsprechen den an dem Mapping beteiligten Quell- bzw. Ziel-Records.
- *Where-Klausel* enthält Filter-Bedingungen, die durch logische Operatoren AND, OR, NOT miteinander verbunden sind.

Zur Beschreibung der Struktur eines Filters werden in dieser Arbeit zwei Ansätze vorgestellt: der Normalform-Ansatz und der sukzessive Ansatz. Anschließend werden diese Ansätze anhand ihrer Vor- und Nachteile gegenübergestellt und ein für die Implementierung ausgewählt.

Der Normalform-Ansatz

Die Struktur eines Filters kann mittels UML modelliert werden. Jede einfache Abbildung zwischen einem Quell- und einem Ziel-Element in der *Select-Klausel* wird als *einfache Transformation* bezeichnet und ist eine Instanz der Klasse **Simple_Transf**. Analog ist jede Aggregat-Funktion in der *Select-Klausel* eine *Aggregat-Transformation* und wird durch die Klasse **Aggr_Transf** repräsentiert. Die Filter-Bedingungen in der *Where-Klausel* (Filter-Ausdrücke) sind durch logische Operatoren AND, OR oder NOT miteinander verbunden und bilden damit einen logischen Ausdruck. Unter der Anwendung von logischen Regeln kann jeder logische Ausdruck in eine konjunktive Normalform (KNF) oder disjunktive Normalform (DNF) umgewandelt werden. Ein Normalform-Ausdruck kann aus mehreren Klauseln bestehen. Jede Klausel enthält wiederum mehrere Literale (A, B, C, etc.). Beispiele für Normalform-Ausdrücke:

$$(\neg A \vee B \vee C) \wedge (D \vee E) \wedge (F \vee G \vee H) \qquad \text{(KNF)}$$

$$(A \wedge B) \vee (C \wedge \neg D \wedge E) \vee (F \wedge G) \vee \neg H \qquad \text{(DNF)}$$

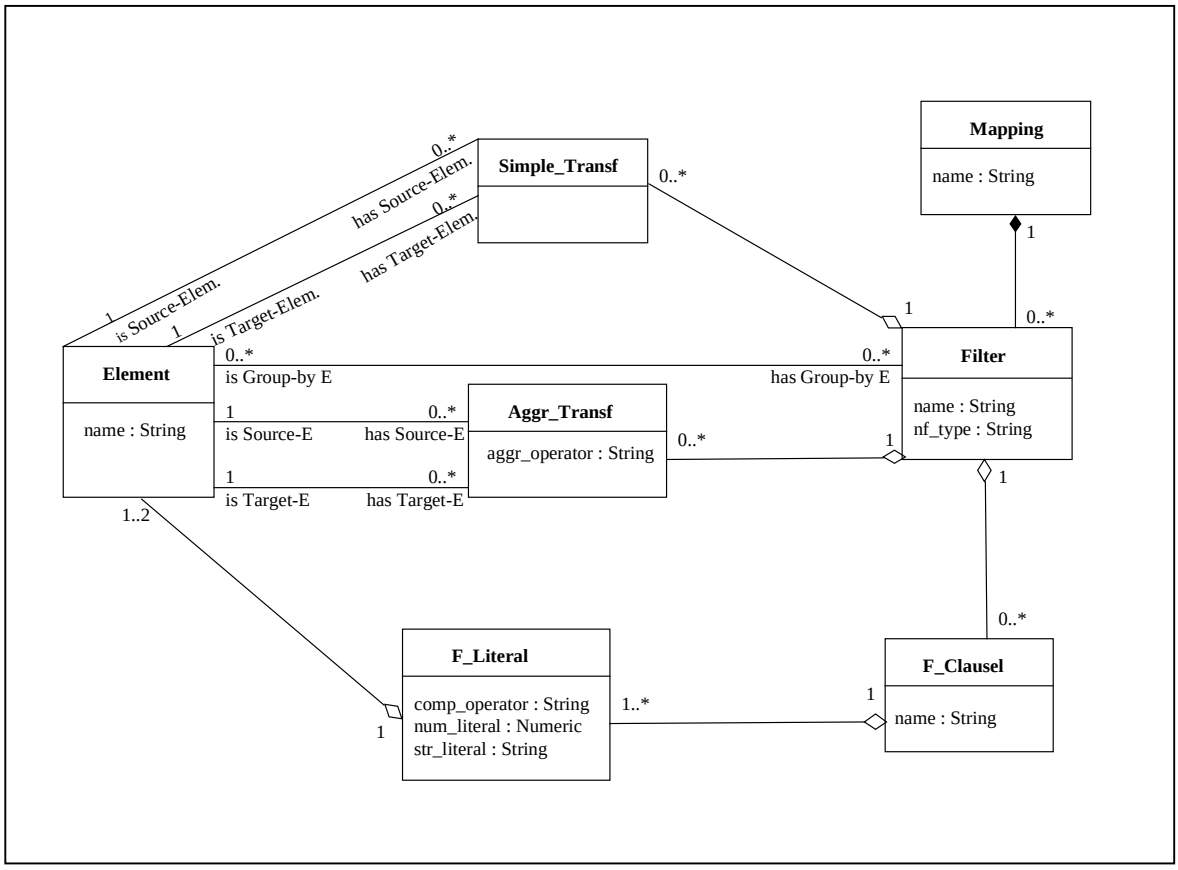


Abbildung 3-4: Das Filter-Schema (Normalform-Ansatz)

Aufgrund dieser Interpretation wird ein Filter-Schema (Normalform-Ansatz) aufgebaut. Das Schema (Abbildung 3-4) kann wie folgt erläutert werden:

- Zwischen den Klassen **Mapping** und **Filter** besteht eine Kompositionsbeziehung (da Filter existenzabhängig vom Mapping ist). Die Kardinalitätsrestriktion beträgt 1 : 0..*, d.h., ein Mapping enthält keinen oder mehere Filter.

- Eine Filter-Instanz kann mehrere Instanzen der Klasse **Simple_Transf** und/oder mehrere Instanzen der Klasse **Aggr_Transf** enthalten. Zu jeder dieser Klassen hat die Klasse **Filter** eine Aggregationsbeziehung mit der Kardinalität 1 : 0..*.

- Falls bestimmte Filter-Bedingungen zu erfüllen sind (*Where-Klausel* vorhanden), werden sie entweder als ein disjunktiver oder ein konjunktiver Ausdruck dargestellt (charakterisiert durch das Attribut *nf_type* der Klasse **Filter**). Dieser logische Ausdruck setzt sich daher aus mehreren Filter-Klauseln (Instanzen der Klasse **F_Clausel**) zusammen, d.h. zwischen den Klassen **Filter** und **F_Clausel** existiert eine Aggregationsbeziehung mit der Kardinalität 1 : 0..*. (Gibt es keine *Where-Klausel*, enthält der Filter keine Filter-Klausel.)

- Eine Filter-Klausel besteht aus einem oder mehreren Filter-Literalen (Instanzen der Klasse **F_Literal**), die durch AND (im Fall einer DNF) oder OR (KNF) miteinander verknüpft werden. Es ist zu beachten, daß die Literale keinen NOT-Operator enthalten sollen, weil der NOT-Operator durch die Umwandlung des Vergleichsoperators im jeweiligen Literal beseitigt werden kann (z.B.: NOT (a < b) ist identisch mit (a >= b)).

- Die linke Seite eines Filter-Literals ist ein Element (*element*) und die rechte Seite ist entweder auch ein Element oder eine numerische Zahl (*num_literal*) oder eine Zeichenkette (*str_literal*). Ein Vergleichsoperator (*comp_operator*) verbindet diese beiden Seiten miteinander. Zur Attributenmenge der Klasse **F_Literal** gehören *com_operator*, *num_literal* und *str_literal*. Die Beziehung zur Klasse **Element** wird durch eine Aggregation mit der Kardinalität 1 : 1..2 beschrieben.

- Die an einem Filter beteiligten Quell- und Ziel-Elemente können durch die Assoziationen zwischen den Klassen **Element** und **Simple_Transf** bzw. **Aggr_Transf** bestimmt werden. Jede dieser Assoziationen hat die Rollen *„hat Quell-Element“* und *„ist Quell-Element“* bzw. *„hat Ziel-Element“* und *„ist Ziel-Element“*. Die jeweilige zugehörige Kardinalität ist 0..* : 1, denn jede (einfache oder Aggregat-) Transformation hat nur ein

Quell- und ein Ziel-Element, aber ein Element kann an mehreren solchen Transformationen teilnehmen.

Hat die *Where-Klausel* eine Group by-Klausel, wird dies durch die Assoziation zwischen den Klassen **Filter** und **Element** dargestellt. Die zugehörigen Rollen und Kardinalität sind: *„hat Group-by Element“, „ist Group-by Element“* und 0..* : 0..*.

Beispiel: Berechne die im April verkaufte Menge der Produkte, deren Verkaufspreis größer als DM 100,00 ist.

Dieses Mapping kann durch den folgenden SQL-Ausdruck beschrieben werden:

```
SELECT p.ProduktID as a.ProduktId, SUM(p.VerkaufsMenge)as a.Menge
INTO  VERKAUF_APRIL a
FROM  PRODUKT p
WHERE p.VerkaufsMonat = „April“AND
      p.Preis >= 100
GROUP BY p.ProduktId
```

Interpretation:

- Quell-Record: Tabelle *PRODUKT*; Ziel-Record: Tabelle *VERKAUF_APRIL*
- Eine einfache Transformation: *p.ProduktId -> a.ProduktId*
- Eine Aggregat-Transformation: *SUM(p.VerkaufsMenge) -> a.Menge*
- Eine Filter-Klausel, deren Filter-Literale durch AND-Operator verbunden sind (*nf_type* des Filters ist DNF)
- Zwei Filter-Literale: *p.VerkaufsMonat = „April“* und *p.Preis >= 100*
- Ein Group-by Element: *p.ProduktId*

Der sukzessive Ansatz

Der Filter-Ausdruck (*Where-Klausel*) kann ähnlich wie beim **Transformations-Schema** von links nach rechts und bei Klammern von innen nach außen unter Berücksichtigung der Vorrangsreihenfolge interpretiert werden. Auch hier ergibt sich eine entsprechende **Filter-Grammatik** in Backus-Naur Form:

- `Filter ::= F_Expression`
- `F_Expression ::= F_Expression log_operator F_Expression | not_operator F_Expression | F_Literal`
- `log_operator ::= AND | OR`
- `not_operator ::= NOT`

- `F_Literal ::= Element comp_operator Element | Element comp_operator num_literal | Element comp_operator str_literal`
- `comp_operator ::= IS | LIKE | = | != | <> | < | <= | > | >= | NOT LIKE`

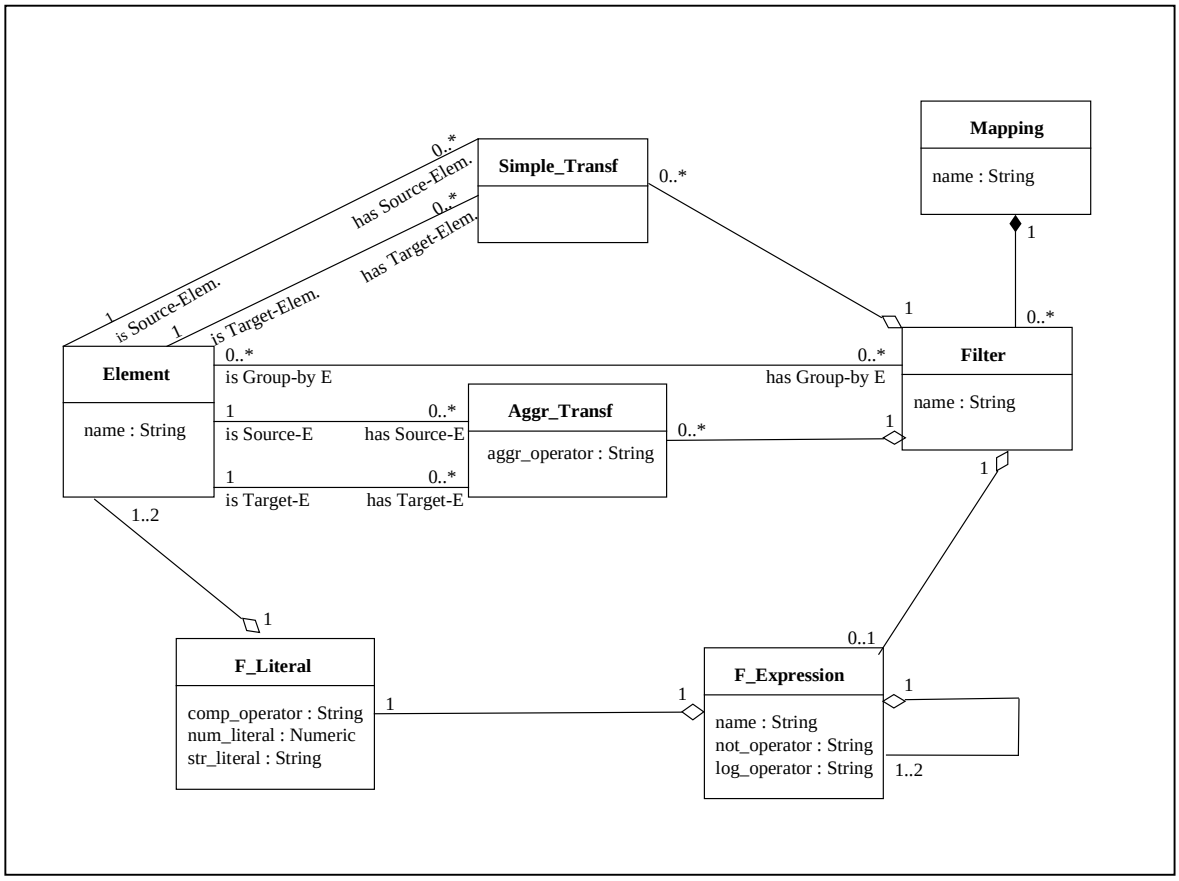


Abbildung 3-5: Das Filter-Schema (sukzessiver Ansatz)

Aus dieser Grammatik entsteht das Filter-Schema des sukzessiven Ansatzes (Abbildung 3-5), das die gleichen Klassen wie das Schema des Normalform-Ansatzes bis auf die Klasse **F_Clausel** enthält. Der Filter-Ausdruck wird durch die Klasse **F_Expression** repräsentiert, die eine Aggregationsbeziehung zur Klasse Filter mit der Kardinalität 1 : 0..1 hat.

Eine *F_Expression* besteht entweder aus zwei anderen *F_Expressions*, die durch einen logischen Operator (*log_operator*: AND, OR) miteinander verbunden sind, oder ist eine negierte *F_Expression.* Eine elementare *F_Expression* ist ein Filter-Literal (*F_Literal*). Die Klasse **F_Expression** hat daher eine rekursive Aggregationsbeziehung zu sich selbst (Kardinalität 1 : 1..2) und eine zur Klasse **F_Literal** (Kardinalität 1 : 1). Außerdem hat sie noch zwei relevante Attribute *not_operator* und *log_operator*.

Vergleich zwischen dem sukzessiven und dem Normalform-Ansatz

Jeder der beiden Ansätze hat seinen Vorteil und Nachteil. Beim sukzessiven Ansatz muß man z.B. nicht darauf achten, ob sich der Filter-Ausdruck in der Select-Klausel schon in der disjunktiven oder konjunktiven Normalform befindet. Die Interpretation dieses Ausdruckes ist dagegen sehr aufwendig, insbesondere bei komplexen Ausdrücken mit vielen NOT-Operatoren und Klammern. Außerdem fordert die rekursive Aggregationsbeziehung der Klasse **F_Expression** eine entsprechende, sehr komplexe Speicherungsstruktur des Ausdruckes in Form einer Relation. Demgegenüber hat die Klasse **F_Clausel** eine viel einfachere Struktur und die gesamte Struktur des aus einer bestimmten Anzahl solcher Klauseln bestehenden Filter-Ausdruckes wird auch übersichtlicher. Übrigens wird der Filter-Ausdruck einer SQL-Anfrage gewöhnlich in Normalformen (DNF oder KNF) formuliert. Die Transformation in die Normalformen ist außerdem auch problemlos. Deshalb wird der Normalform-Ansatz im Vergleich zum sukzessiven Ansatz bevorzugt und für die Implementierung ausgewählt.

3.3 Das objektrelationale Modell für technische Metadaten

Die Entwicklung der objektrelationalen Datenbanksysteme (ORDBS) steht noch am Anfang, aber ihre Bedeutung nimmt stark zu. Wie der Name schon besagt, entsteht das objektrelationale Datenmodell aus einer Erweiterung des relationalen Datenmodells um Objekt-Orientierung. Während es in den relationalen DBS nur einfache Datentypen und Attributtypen gibt, erlauben die ORDBS benutzerdefinierte Datentypen (u.a. Multimedia-Datentypen) und komplexe, nicht-atomare Attributtypen (z.B. relationenwertige Attribute). Dennoch werden die Grundlagen relationaler DBS, insbesondere deklarativer Datenzugriff, Sichtkonzept etc., bewahrt. Wie die relationalen Datenbanken stellen die ORDBS zur Speicherung der Daten Tabellen oder Relationen zur Verfügung. Jede Relation besteht aus einer bestimmten Anzahl von Spalten oder Attributen, die nicht-atomar sein und komplexe Datentypen haben können, und einer Menge von Zeilen oder Tupeln. Integritätsbedingungen wie Primärschlüssel- und Fremdschlüsselkandidaten sind auch anzugeben (siehe [Rah99], [Dus98], [IUS97]).

Die Transformation eines UML-Modells in das objektrelationale Modell (ORM) erfolgt in ähnlicher Weise wie die Transformation eines Entity-Relationship-Modells in das Relationenmodell mit Ausnahme von einigen Besonderheiten bezüglich objektorientierter Merk-

male (vgl. [Rah94], [Vos94] und [Wer95]). Folgende wichtige Regeln müssen dabei beachtet werden:

(1) Jeder Klasse des UML-Modells kann eindeutig eine Relation/Tabelle mit gleichem Namen, demselben Schlüssel- und beschreibenden Attributen innerhalb des ORM zugeordnet werden.

(2) Manche Klassen können durch benutzerdefinierte Datentypen (z.B. row type) beschrieben werden.

(3) Bei einer 1 : 1 - oder 1 : n – Beziehung kann auf die Abbildung durch eine gesonderte Relation/Tabelle verzichtet werden. Hier wird entweder der Primärschlüssel einer der beiden Klassen als Fremdschlüssel in die Attributenmenge der zur anderen Klasse gehörenden Relation übernommen, oder die Klasse der „n-Seite“ wird als Kollektions-Typen (z.B. SET, LIST) auch in die Attributenmenge der Klasse der „1-Seite“ übernommen.

(4) Eine n : m – Beziehung wird in der Regel durch eine eigene Relation/Tabelle dargestellt, deren Attributenmenge die Primärschlüssel der beteiligten Klassen enthält.

(5) Generalisierungs- bzw. Vererbungsbeziehung wird durch das ORM unterstützt. Normalerweise werden die Klassen in der Vererbungshierarchie durch eine Typ-Hierarchie repräsentiert. Darauf wird anschließend eine Tabellen-Hierarchie gebildet.

Nach diesen Regeln wird im weiteren die jeweilige Transformation aus dem Gesamt-Schema, dem Schema der Transformation und dem Schema des Filters in ein entsprechendes objektrelationales Datenmodell durchgeführt. Jede Tabelle/Relation dieser Datenmodelle wird durch die jeweiligen Attribute, Wertebereiche, Integritätsbedingungen (NULL-Wert) und zusätzliche Bemerkungen charakterisiert.

3.3.1 Objektrelational-Modell – Gesamt-Schema

Im Gesamt-Schema werden insgesamt folgende Klassen und ihre Beziehungen zueinander beschrieben: **Data_Store**, **Database**, **File**, **Schema**, **File_Structure**, **Record**, **Element**, **Mapping_Folder**, **Mapping**, **Filter**, **Transformation**, **User** und **Privileg**. Alle diesen Klassen, außer der beiden Klassen **User** und **Privileg**, sollen im Rahmen der Arbeit in geeignete objektrelationale Relationen transformiert werden.

1. Klasse Data_Store => Tabelle *data_store*

Die zuerst untersuchte Klasse ist die Klasse **Data_Store**, aus der eine neue Tabelle mit demselben Name (jedoch alles klein geschrieben) gebildet wird. Die Tabelle *data_store* hat zwei Attribute, wobei das Attribut *name* die Rolle des Primärschlüssels spielt.

Tabelle: data_store

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel
aliasname	VARCHAR(30)		Zur Beschreibung

2. Klasse Database => Tabelle *database*

In ähnlicher Weise entsteht die Tabelle *database* aus der Klasse **Database** mit allen zugehörigen Attributen. Da ein Data Store immer nur eine Database-Instanz repräsentiert, können beide Objekte den gleichen Namen haben. Daher spielt das Attribut *name* der Tabelle *database* die Rolle sowohl des Primärschlüssels als auch des Fremdschlüssels.

Tabelle: database

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Name des Data Store
type	VARCHAR(20)	NOT NULL	Value: ‚RDB', ‚OODB', ‚HDB', ...
system	VARCHAR(20)		
servername	VARCHAR(20)		
portnumber	INTEGER		
hostname	VARCHAR(30)		
connection	VARCHAR(30)		

3. Klasse File => Tabelle *file*

Zwischen der Klasse **Data_Store** und der Klasse **File** existiert eine 1 : n – Beziehung. Demzufolge soll noch ein Attribut, *ds_name*, das als Fremdschlüssel den Primärschlüssel *name* der Tabelle *data_store* referenziert, in die Attributenmenge der Tabelle *file* eingefügt

werden (nach der Regel 4). Nicht nur der Primär- und Fremdschlüssel darf keinen Null-Wert haben, sondern die Lokation der Datei muß auch angegeben werden.

Tabelle: file

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel
ds_name	VARCHAR(20)	NOT NULL	Fremdschlüssel; Name des Data Store
location	VARCHAR(40)	NOT NULL	

4. Klasse Schema => Tabelle *schema* und Klasse File_Structure => Tabelle *file_structure*

Die beiden Klassen **Schema** und **File_Structure** stehen in einer Vererbungshierarchie, wobei **Schema** die Oberklasse und **File_Structure** die Unterklasse ist. Nach der Regel 5 sollen diese Klassen zuerst als Tupel-Typ (row type) in einer Typ-Hierarchie definiert werden. Die aus diesen vordefinierten Datentypen entstandenen Klassen bilden dann eine Tabelle-Hierarchie, wobei die Unterklassen alle Eigenschaften der Oberklasse vererben können.

Tabelle: schema von Tupel-Typ schema_t[3]

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel
ds_name	VARCHAR(20)		Fremdschlüssel; Name des Data Store
aliasname	VARCHAR(30)		Zur Beschreibung

Eine 1 : n – Beziehung zwischen den Klassen **Data_Store** und **Schema** führt dazu, daß der Primärschlüssel der Klasse **Data_Store** als Fremdschlüssel in der Klasse **Schema** unter dem Attribut *ds_name* auftaucht (nach der Regel 3). Dieses Attribut kann einen Null-Wert haben, wenn das Schema nur eine logische Struktur beschreibt und von keinem Data Store referenziert wird.

[3] Das bedeutet: Klasse **schema** wird auf der Grundlage von „row type“ *schema_t* gebildet.

Tabelle: file_structure von Tupel-Typ filestructure_t

Attribut	Wertebereich	Null-Wert	Bemerkung
format	VARCHAR(20)		
desc_format	VARCHAR(20)		
col_length	INTEGER		
col_separator	CHAR(2)		

5. Klasse Record => Tabelle *record*

Zwischen der Klasse **Schema** und der Klasse **Record** besteht eine Kompositionsbeziehung, die einer 1 : n – Beziehung ähnlich ist. Neben den übrigen Attributen name und aliasname muß daher der Tabelle *record* ein neues Attribut, *s_name*, das als Fremdschlüssel auf das zugehörige Schema referenziert, hinzugefügt werden (Regel 3). Außerdem ist der Record-Name nur innerhalb eines Schemas eindeutig. Deshalb kann es passieren, daß im gesamten System manche Tabellen oder Rekords die gleichen Namen haben. Um eine Record-Instanz zu identifizieren, muß der Fremdschlüssel *s_name* zusammen mit dem Attribut *name* zu den Primärschlüsselkandidaten gehören.

Tabelle: record

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel
s_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Schema-Name
aliasname	VARCHAR(30)		Zur Beschreibung

6. Klasse Element => Tabelle *element*

Analog wird der Primärschlüssel der Tabelle *record* als Fremdschlüssel in die Attributenmenge der Tabelle *element* übernommen. Dies sind die Attributen *r_name* (Record-Name) und *s_name* (Schema-Name). Darüber hinaus werden diese Attribute zur Identifizierung der Tabelle *element* als ihre Primärschlüsselkandidaten verwendet.

Tabelle: element

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel
r_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Record-Name
s_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Schema-Name
aliasname	VARCHAR(30)		Zur Beschreibung
datatype	VARCHAR(15)		
length	INTEGER		
scale	INTEGER		
nullable	INTEGER		1: kann Null-Wert haben 0: kein Null-Wert

7. Element-Record-Beziehung => Tabelle *reference*

Es bleiben bei der Transformation der Klassen **Record** und **Element** noch zwei relevante Beziehungen: zum einen die Assoziation zwischen der Klasse **Record** und der Klasse **Element** mit den zugehörigen Assoziationsrollen *„hat Primärschlüssel“* und *„ist Primärschlüssel“;* zum anderen die Assoziation der Klasse **Record** zu sich selbst mit den Rollen *„referenziert“* und *„wird referenziert von“* (n : m – Beziehung). Diese beiden Assoziationen repräsentieren die Entity- und/oder die referenziellen Integritätsbedingungen, d.h. die Primärschlüssel- und/oder die Fremdschlüsselbedingungen. Aus ihnen wird eine spezielle Tabelle namens *reference* gebildet, durch deren Attributenmenge die obigen Integritäts-Bedingungen beschrieben werden können.

Tabelle: reference

Attribut	Wertebereich	Null-Wert	Bemerkung
schema	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Name des Schemas
pktable	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Name der (referenzierten) Tabelle
pkcolumn	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel;

			Name d. Primärschlüssels
pkseq	INTEGER		Reihe des Primärschlüssels
fktable	VARCHAR(20)		Name der referenzierenden Tabelle
fkcolumn	VARCHAR(20)		Name des Fremdschlüssels

Die Tabelle *reference* enthält Information über den Primärschlüssel oder die Primärschlüsselkandidaten einer bestimmten Tabelle in einem Schema (durch die Attribute *schema, pktable, pkcolumn, pkseq*). Die Attribute *fktable* und *fkcolumn* stellen den Namen der Tabelle, die die Primärschlüssel-Tabelle *pktable* referenziert, und deren Fremdschlüssel dar. Weil nicht alle Tabellen von anderen referenziert werden, d.h. in diesem Fall gibt es keine referenzierende Tabelle und ihren Fremdschlüssel, können die Attribute *fktable* und *fkcolumn* einen Nullwert haben. Die Attribute *schema, pktable* und *pkcolumn* bilden die Primärschlüsselkandidaten der Tabelle *reference*. Zugleich spielen sie die Rolle der Fremdschlüssel, die den Primärschlüssel der Tabelle *element* referenzieren.

8. Klasse Mapping_Folder => Tabelle *mapping_folder*

Ähnlich wie die Tabelle *data_store* hat die Tabelle *mapping_folder* nur zwei Attribute, *name* als Primärschlüssel und *aliasname* zur genaueren Beschreibung der Tabelle, falls das nötig ist.

Tabelle: mapping_folder

Attribut	**Wertebereich**	**Null-Wert**	**Bemerkung**
name	VARCHAR(20)	NOT NULL	Primärschlüssel
aliasname	VARCHAR(30)		Zur Beschreibung

9. Klasse Mapping => Tabelle *mapping*

Ein Mapping ist immer in einem bestimmten Mapping-Folder enthalten. Deshalb soll das Attribut *f_name* (Name des Folders) als Fremdschlüssel in der Tabelle *mapping* dienen.

Tabelle: mapping

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel
f_name	VARCHAR(20)	NOT NULL	Fremdschlüssel Name des Folders
aliasname	VARCHAR(30)		Zur Beschreibung

10. Record-Mapping-Beziehung => Tabelle *record_mapping*

Die n : m – Beziehungen zwischen den Klassen **Record** und **Mapping** sollen nach der Regel 5 durch eine eigene Tabelle dargestellt werden. Dabei bilden die Primärschlüssel der entsprechenden Tabellen die Primärschlüsselkandidaten (*r_name*, *s_name*, *m_name*) der neuen Tabelle *record_mapping*. Es wird zusätzlich das Attribut *role* hinzugefügt, das angibt, ob der jeweilige Record an dem gegebenen Mapping in der Rolle als Quelle oder Ziel teilnimmt.

Tabelle: record_mapping

Attribut	Wertebereich	Null-Wert	Bemerkung
r_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Record-Name
s_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel Schema-Name
m_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel Mapping-Name
role	CHAR(6)		Value: ‚source', ‚target'

Die Transformation der beiden Klassen **Filter** und **Transformation** werden in den nächsten Abschnitten ausführlich behandelt, da sie nur zusammen mit zusätzlichen Klassen alle Filter- und Datenkonvertierungsregeln der Abbildung zwischen Quelle und Ziel weitestgehend beschreiben können.

3.3.2 Objektrelational-Modell – Transformations-Schema

1. Klasse Transformation => Tabelle transformation

Die Klasse **Transformation** hat eine 1 : n - Kompositionsbeziehung zur Klasse **Mapping**, deshalb wird nach der Regel 3 der Primärschlüssel der Tabelle *mapping* als Fremdschlüssel in die Attributenmenge der Tabelle *transformation* übernommen (Attribut *m_name* bedeutet Name des zugehörigen Mappings). Ähnlich ist es mit den Beziehungen zur Klasse **Element**: die Tabelle *transformation* hat noch zwei zusätzliche Attribute, nämlich *target_elem* (Ziel-Element) und *source_elem* (Quell-Element). Diese Attribute müssen Primärschlüsselkandidaten der Tabelle *element* sein. Um eine vollständige Übernahme aller drei Attribute (*name, r_name, s_name*) in die Tabelle *transformation* zu vermeiden, werden sie zu einem einzigen Attribut zusammengefaßt. Jedoch muß dieses Attribut zwecks der Eindeutigkeit z.B. wie folgt angegeben werden:

```
source_elem = s_schema.s_record.s_element
target_elem = t_schema.t_record.t_element
```

Der Wertebereich von *source_elem* und *target_elem* wird entsprechend auch vergrößert (VARCHAR(40)). Da an einer Transformation mehrere Quell-Elemente teilnehmen können (1 : n – Beziehung), wird nach der Regel 3 das Attribut *source_elem* als ein Kollektions-Typ, SET (VARCHAR(40)), beschrieben.

Tabelle: transformation

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel
m_name	VARCHAR(20)	NOT NULL	Fremdschlüssel; Mapping-Name
target_elem	VARCHAR(40)	NOT NULL	Name des Ziel-Elements
source_elem	SET (VARCHAR(40))	NOT NULL	Menge der Namen der Quell-Elemente

2. Klasse Function => Tupel-Typ *function_t*

Ein Vorteil der ORDBS gegenüber den RDBS besteht darin, daß sie nicht nur einfache, atomare, sondern auch komplexe Datentypen unterstützen. Oben wurden zwei Arten von komplexen Datentypen vorgestellt: Kollektions-Typen und Tupel-Typen. Die Tupel-Typen

(row types) können zur Erzeugung einer Typ-Tabelle (obige Beispiele: die Tabellen *schema* und *file_structure*) oder oftmals zur Erzeugung einer Spalte verwendet.

Bei der Transformation in das Relationalmodell wird in der Regel jedes Entity oder jede Klasse in einer gesonderten Relation/Tabelle dargestellt. In einem objektrelationalen Modell kann anstelle dieser Tabelle nur eine Spalte als ein Tupel-Typ gebildet werden. Für die Klasse **Function** wird dies (Regel 2) angewendet, weil die Funktionen tatsächlich nur ein Bestandteil eines relevanteren Objekts (Expression) sind und nicht unbedingt durch eine eigene Tabelle dargestellt werden.

Tupel-Typ: function_t

Attribut	**Wertebereich**	**Null-Wert**	**Bemerkung**
name	VARCHAR(20)	NOT NULL	Name der Funktion
param_list	LIST (VARCHAR(20))	NOT NULL	Parameterliste

Weil die Reihenfolge der Parameter einer Funktion relevant ist, muß das Attribut *param_list* den Kollektions-Typ LIST haben, der eine geordnete Menge von Elementen auch mit Duplikatwerten repräsentiert.

3. Klasse Expression => Tabelle *expression*

Neben dem Attribut *name* gehört der Fremdschlüssel *t_name* auch zu den Primärschlüsselkandidaten der Tabelle *expression*. Die rekursive Aggregationsbeziehung wird durch zwei weitere Attribute, *left_expr* und *right_expr*, dargestellt, die zwei Teil-Expressionen der aus ihnen zusammengesetzten Expression repräsentieren. Diese Attribute haben den Datentyp „VARCHAR“ und haben als Werte entweder Namen der (Teil-) Expressionen, falls diese weiter geteilt werden können, oder einen Element-Namen, oder eine numerische Zahl, die auch als Zeichenkette (String) gespeichert wird. Deshalb wird nur das Attribut *bin_operator* und nicht das Attribut *num_literal* der Klasse **Expression** in der Tabelle *expression* übernommen. Außerdem hat diese Tabelle noch ein spezielles Attribut, nämlich *function*, dessen Datentyp der vordefinierter Tupel-Typ *function_t* ist.

Tabelle: expression

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel; Name der Expression
t_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Name der Transformation
left_expr	VARCHAR(20)		Linke Expression
right_expr	VARCHAR(20)		Rechte Expression
bin_operator	CHAR(1)		Value: ‚+', ‚-', ‚*', ‚/'
function	function_t		

3.3.3 Objektrelational-Modell – Filter-Schema

Das Filter-Schema (Normalform-Ansatz) wird für die Implementierung verwendet werden. Deshalb wird im weiteren eine Transformation dieses Schemas in ein objektrelationales Modell vorgenommen. Ähnlich wie mit dem Schema der Transformation brauchen nicht alle Klassen des Filter-Schemas in einer gesonderten Tabelle dargestellt zu werden. Zur Vereinfachung können die Klassen **Simple_Transf**, **Aggr_Transf** und **F_Literal** als Tupel-Typen definiert werden.

1. Klasse Simple_Transf => Tupel-Typ *simpletransf_t*

Eine einfache Transformation enthält nur ein Quell-Element und ein Ziel-Element. Daher benötigt der Tupel-Typ *simpletransf_t* zwei Attribute: *source_elem* und *target_elem*, die folgende Form haben:

source_elem = s_schema.s_record.s_element

target_elem = t_schema.t_record.t_element

Tupel-Typ: simpletransf_t

Attribut	Wertebereich	Null-Wert	Bemerkung
source_elem	VARCHAR(40)	NOT NULL	Name des Quell-Elements
target_elem	VARCHAR(40)	NOT NULL	Name des Ziel-Elements

2. Klasse Aggr_Transf => Tupel-Typ *aggrtransf_t*

Neben den zwei Attributen *source_elem* und *target_elem* wie beim Tupel-Typ *simple-transf_t* hat der Tupel-Typ *aggrtransf_t* noch das Attribut *aggr_operator* zur Bestimmung des Namens des Aggregat-Operators.

Tupel-Typ: aggrtransf_t

Attribut	Wertebereich	Null-Wert	Bemerkung
source_elem	VARCHAR(40)	NOT NULL	Name des Quell-Elements
target_elem	VARCHAR(40)	NOT NULL	Name des Ziel-Elements
aggr_operator	VARCHAR(5)	NOT NULL	Name des Aggregat-Operators

3. Klasse F_Literal => Tupel-Typ *filterliteral_t*

Jedes Filter-Literal besteht immer aus drei Teilen: ein Element in der linken Seite, ein Element oder eine Zeichenkette oder eine numerische Zahl in der rechten Seite und ein die beiden Seiten verbindender Vergleichsoperator. Zur Verallgemeinerung läßt sich diese Struktur aus der linken Seite, dem Vergleichsoperator und der rechten Seite zusammensetzen. Diese Teile bilden auch die Attributenmenge des Tupel-Types *filterliteral_t.*

Tupel-Typ: filterliteral_t

Attribut	Wertebereich	Null-Wert	Bemerkung
lit_left	VARCHAR(40)	NOT NULL	Linke Seite des Filter-Literals
com_operator	VARCHAR(8)	NOT NULL	Vergleichsoperators
lit_right	VARCHAR(40)	NOT NULL	Rechte Seite des Filter-Literals

4. Klasse Filter => Tabelle *filter*

Aufgrund der Kompositionsbeziehung zur Klasse **Mapping** wird der Primärschlüssel der Tabelle *mapping* als Fremdschlüssel in die Tabelle *filter* übernommen. Die Aggregations-beziehungen zu den Klassen **Simple_Transf** und **Aggr_Transf** werden durch die Attribute *simple_transfs* und *aggr_transfs* dargestellt. Das sind Kollektions-Typen von den vordefi-

nierten Tupel-Typen *simpletransf_t* und *aggrtransf_t*. Sie repräsentieren die Menge der in einem Filter enthaltenen einfachen und/oder Aggregat-Transformationen. Weitere relevante Attribute der Tabelle *filter* sind *nf_type* und *grby_elem*, wobei das Attribut *nf_type* folgende Werte haben kann: ‚DNF' (disjunktive Normalform), ‚KNF' (konjunktive Normalform) oder Leer-String ‚' (keine Filter-Bedingung).

Tabelle: filter

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel; Name des Filters
m_name	VARCHAR(20)	NOT NULL	Fremdschlüssel; Mapping-Name
simple_transfs	SET (simpletransf_t)	NOT NULL	Menge der einfachen Transformation
aggr_transfs	SET (aggrtransf_t)	NOT NULL	Menge der Aggregat-Transformation
nf_type	CHAR(10)		Value: ‚DNF', ‚KNF', ‚'
grby_elem	VARCHAR(40)		Name des Group-by-Elements

5. Klasse F_Clausel => Tabelle *filter_clausel*

Eine Filter-Klausel wird durch ihren Namen (Attribut *name*) und den Namen des sie enthaltenden Filters (Attribut *f_name* als Fremdschlüssel)) identifiziert. Außerdem charakterisiert das Attribut *f_literals* eine Menge von Filter-Literalen, d.h. eine Kollektion (SET) vom Tupel-Typ *filterliteral_t*.

Tabelle: filter_clausel

Attribut	Wertebereich	Null-Wert	Bemerkung
name	VARCHAR(20)	NOT NULL	Primärschlüssel; Name der Filter-Klausel
f_name	VARCHAR(20)	NOT NULL	Primär-/ Fremdschlüssel; Filter-Name
f_literals	SET (filterliteral_t)	NOT NULL	Menge der Filter-Literale

4 Implementierung

Im letzten Kapitel haben wir uns mit der Modellierung von technischen Metadaten eines Repository beschäftigt. Die prototypische Implementierung dieses konzeptuellen Metamodells in einem DBMS ist der Gegenstand dieses Kapitels. Als DBMS für das Repository wurde Informix Universal Server (Version 9.1), ein objektrelationales DBMS, ausgewählt. Zunächst muß ein geeignetes Datenbankschema mit Hilfe von DDL-Anweisungen erstellt werden. Diese Datenbank muß in der Lage sein, die technischen Metadaten einer Data Warehouse-Umgebung abzuspeichern. In dieser Umgebung stehen für den Test zwei Datenbanken zur Verfügung: die Film-Datenbank auf dem DB2-DBS (***MOVIEDB2***) als Quellsystem; die Ziel-Datenbank (***ardnt_movie_target***) befindet sich auf dem Informix-DBS. Die Metadaten bezüglich Struktur und Inhalt dieser Systeme und die Definitionen und Regeln einiger Test-Mappings zwischen ihnen sollen in der Repository-Datenbank gespeichert werden.

Um das erstellte Datenbankschema mit den wirklichen (Meta-)Daten auszufüllen, muß vor allem ein Zugriff auf die DBS in der Umgebung möglich sein. Für einen einheitlichen Zugriff wurde die ODBC-Programmierung ausgewählt. Bevor die eigentliche Implementierung behandelt wird, wird zunächst in die Grundlagen des Informix-DBS und der ODBC-Programmierung eingeführt.

4.1 Informix-Universal Server als Repository-Datenbank

Informix Universal Server (IUS) Version 9.1 ist ein objektrelationaler Datenbankserver, der objektorientierte und relationale Fähigkeiten kombiniert (siehe [Dus98] und [IUS97]). Er stellt eine SQL-Version zur Verfügung, die hochkompatibel mit der Standardsprache SQL92 ist. Darüber hinaus werden viele Eigenschaften von SQL3 realisiert, wie z.B. Definition und Verwendung von abstrakten Datentypen, Typhierarchien und Vererbung, komplexe Objekte und Kollektions-Typen etc.

Folgende objektorientierte Fähigkeiten werden vom IUS unterstützt:

- ***Erweiterbarkeit***

  Die Fähigkeit des Datenbankservers kann durch benutzerdefinierte Datentypen, benutzerdefinierte Routinen und benutzerdefinierte Zugriffsmethoden erweitert werden. In-

formix verpackt einige Datentypen und ihre Zugriffsmethoden zu Data Blade Modules oder gemeinsamen Klassenbibliotheken, die in den Datenbankserver hinzugefügt werden können. Data Blade Modules ermöglichen die Speicherung nicht-traditioneller Datentypen wie zwei-dimensionaler Raumobjekte, Image, Large-Textdokumente, TimeSeries, Web etc. und ihre Zugriffspfade durch Indexstrukturen.

- ***Komplexe Typen***

  Ein komplexer Datentyp ist ein benutzerdefinierter Datentyp, der einen oder mehrere existierende Datentypen kombiniert. Ein bedeutendes Charakteristikum eines komplexen Datentyps besteht darin, daß auf jeden seiner Komponent-Datentypen zugegriffen werden kann. IUS unterstützt zwei Arten von komplexen Typen:

  - *Kollektions-Typen (collection types)* ermöglichen, Kollektionen von Daten innerhalb eines einzelnen Tupels einer Tabelle zu speichern und zu manipulieren. Ein Kollektions-Typ hat zwei Komponenten: einen Typkonstruktor (SET, MULTISET oder LIST) und einen Element-Typ, der den Typ der in der Kollektion enthaltenen Daten festlegt. Der Element-Typ einer Kollektion kann einen einzelnen Datentyp (Spalte) oder vielfache Datentypen (Tupel) repräsentieren. Es ist zu beachten, daß eine Kollektion kein Null-Element enthalten kann.

  - *Tupel-Typen (row types)* können einer Spalte oder einer Tabelle zugewiesen werden. Es wird zwischen „named row type“ und „unnamed row type“ unterschieden. Der erste ist eine Gruppe von Feldern, die unter einem einzelnen Name definiert werden. Der zweite ist eine Gruppe von „typed“ Feldern, die mit dem ROW-Konstruktor erzeugt werden.

  Komplexe Typen können als Spalten-Typen, Argument- und Rückgabe-Typen einer Routine oder Feld-Typen in anderen komplexen Typen verwendet werden.

- ***Vererbung***

  Durch den Vererbungsprozeß können Objekte (Typen oder Tabellen) definiert werden, die die Eigenschaften anderer Objekte erwerben und neue spezifische Eigenschaften aufnehmen. Das Objekt, das die Eigenschaften erbt, wird Untertyp (subtype) oder Untertabelle (subtable) genannt. Das Objekt, dessen Eigenschaften geerbt werden, ist der Obertyp (supertype) oder die Obertabelle (supertable).

IUS unterstützt die Vererbung lediglich für „named row types“ und „typed“-Tabellen und auch nur einfache Vererbung, wobei jeder Untertyp oder jede Untertabelle nur einen Obertyp oder eine Obertabelle hat.

IUS stellt objektorientierte Fähigkeiten über das relationale Modell hinaus bereit, repräsentiert aber alle Daten in der Form von Tabellen mit Tupeln und Spalten. Die Beziehungen des objektrelationalen Modells zur realen Welt können wie folgendes beschrieben werden:

- *Tabelle = Entity* (Eine Tabelle repräsentiert alle Informationen über ein Subjekt oder eine Art von Sache.)
- *Spalte = Attribut/e* (Eine Spalte repräsentiert eine oder mehrere Eigenschaften, Charakteristiken oder Fakten des Tabelle-Subjekts.)
- *Tupel = Instanz* (Ein Tupel repräsentiert eine individuelle Instanz des Tabelle-Subjekts.)

4.2 Definition der Repository-Datenbank

Die Repository-Daten sollen auf das Informix-DBMS Version 9.1 gebracht werden. Informix SQL kann zur Datendefinition, Datenmanipulation und zur Datenkontrolle eingesetzt werden. Die Daten-Definitionssprache von Informix wurde verwendet, um ein Datenbankschema zu erstellen, das die einzelnen Typen und Relationen des objektrelationalen Modells aus Kapitel 3 enthält. Insbesondere wurden die von Informix unterstützten objektorientierten Eigenschaften wie Kollektions-Typen, named Tupel-Typen und Vererbung für die Definition der Repository-Datenbank genutzt.

Die Datendefinition zum Erstellen des Datenbankschemas **Test-Repository** befindet sich im Anhang D. Neben den gewöhnlichen Definitionen aus dem Standard-SQL werden hier spezielle Definitionen von Tupel-Typen (named row types), „typed“ Tabellen, Typ- und Tabelle-Hierarchien und Kollektions-Typen zum Einsatz gebracht. Die Syntax der Definition von named Tupel-Typen ist ähnlich wie die von Tabellen:

```
CREATE ROW TYPE schema_t
  (
    name           VARCHAR(20)      NOT NULL,
    ds_name        VARCHAR(20),
    aliasname      VARCHAR(30)
  );


CREATE ROW TYPE filestructure_t
  (
```

```
    format         VARCHAR(20),
    descr_format   VARCHAR(20),
    col_length     INTEGER,
    col_separator  CHAR(2)
  )
UNDER schema_t;
```

Die Tupel-Typen *schema_t* und *filestructure_t* bilden eine Typhierarchie, wobei der Untertyp *filestructure_t* unter („under“) dem Obertyp *schema_t* definiert wird. Aufgrund dieser Tupel-Typen werden in ähnlicher Weise die „typed“-Tabellen *schema* und *file_structure* erzeugt:

```
CREATE TABLE schema OF TYPE schema_t
  (
    PRIMARY KEY (name),
    FOREIGN KEY (ds_name) REFERENCES data_store (name) ON DELETE
    CASCADE
  );

CREATE TABLE file_structure OF TYPE filestructure_t UNDER schema;
```

Komplexe Datentypen wie Kollektions-Typen (SET, LIST) und Tupel-Typen werden als Spalten z.B. in den Tabellen *transformation*, *expression*, *filter* und *filter_clausel* definiert. Da die Kollektions-Typen keinen Null-Wert erlauben, muß die NOT NULL-Bedingung bei deren Definition angegeben werden.

```
CREATE ROW TYPE simpletransf_t
  (
    source_elem    VARCHAR(40)      NOT NULL,
    target_elem    VARCHAR(40)      NOT NULL
  );

CREATE ROW TYPE aggrtransf_t
  (
    source_elem    VARCHAR(40)      NOT NULL,
    target_elem    VARCHAR(40)      NOT NULL,
    aggr_operator  VARCHAR(5)       NOT NULL
  );

CREATE TABLE filter
  (
    name           VARCHAR(20)      NOT NULL,
    m_name         VARCHAR(20)      NOT NULL ,
    simple_transfs SET(simpletransf_t    NOT NULL),
    aggr_transfs   SET(aggrtransf_t      NOT NULL),
    nf_type  CHAR(10),
    grby_elem      VARCHAR(40),

    CHECK (nf_type IN (’’, ’DNF’ ,’KNF’)),
    PRIMARY KEY (name),
    FOREIGN KEY (m_name) REFERENCES mapping (name) ON DELETE
    CASCADE
  );
```

Abgesehen von solchen objektrelationalen Eigenschaften wird die Datenbank wie andere relationale Datenbanken definiert. Die Datenbankkonsistenz wird durch Integritätsbedingungen gewährleistet: atomare Integrität durch die Verwendung von Primärschlüssel und

referentielle Integrität durch Fremdschlüssel. Solche Bedingungen werden durch den Einsatz der PRIMARY- bzw. FOREIGN KEY-Klausel realisiert. Bei den meisten Tabellen wird das Schlüsselwort „ON DELETE CASCADE“ in der FOREIGN KEY-Klausel genutzt, um das Entfernen eines Datensatzes in der referenzierten Tabelle zu veranlassen. Das betrifft insbesondere z.B. die Beziehung zwischen der Tabelle *mapping* und den sie referenzierenden Tabellen wie *transformation* und *filter*: zusammen mit einer Mapping-Instanz werden alle zugehörigen Transformations- und/oder Filter-Instanzen gelöscht. Ähnliche Beziehungen existieren auch zwischen den Tabellen *data_store* und *database/file/schema*, *schema* und *record*, *record* und *element*. Weitere Integritätsbedingungen können durch die Anwendung der CHECK-Klausel formuliert werden.

4.3 ODBC als Mechanismus für Datenbank-Zugriff

4.3.1 Vorteile des ODBC-Konzepts

ODBC (Open Database Connectivity) ist ein Standard-API von Microsoft für einen einheitlichen Zugriff auf SQL-Datenbanken. Dieses API ist unabhängig von irgendeinem DBMS oder Betriebssystem und auch (programmier)sprachunabhängig. Es gibt verschiedene Programmiermodelle für den Datenbankzugriff, die mögliche Alternativen zu ODBC darstellen (siehe [Gei95]). Darunter sind die zwei folgenden Modelle die wichtigsten:

- **Eingebettetes SQL (Embedded SQL)**

  Das sind SQL-Anweisungen, die in andere Programmiersprachen (Hostsprachen) wie z.B. C oder COBOL eingebettet sind. Diese SQL-Anweisungen werden mit normaler Programmsyntax kombiniert und durch einen Precompiler in Funktionsaufrufe für die DBMS-Laufzeitbibliothek übersetzt. Es gibt zwei Hauptformen des eingebetteten SQL: statisch und dynamisch. Beim statischen SQL werden alle SQL-Anweisungen beim Schreiben des Anwendungsprogramms definiert, ihre Struktur verändert sich also nicht während der Laufzeit. Beim dynamischen SQL dagegen können die SQL-Anweisungen zur Laufzeit der Anwendung „dynamisch“ konstruiert werden.

- **Call Level Interfaces (CLI)**

  Ein CLI besteht aus Funktionsaufrufen in einer Programmiersprache wie etwa C, COBOL oder FORTRAN. CLI wird eingesetzt, um eine Schnittstelle (auch als API) zu

Datenbanken zu beschreiben. Die Arbeitsweise ist ähnlich dem Modell mit dem dynamischen eingebetteten SQL, aber ohne den Einsatz des Precompilers.

Der Hauptnachteil der obigen Modelle besteht darin, daß eine einzige Anwendung lediglich auf ein spezielles DBMS zugreifen kann und der Zugriff auf andere DBMS nur möglich ist, wenn diese Anwendung mit Programmieraufwand zur Anpassung an die Eigenschaften anderer Datenbanken gebracht wird. Z.B. bei der ESQL-Programmierung müssen die Funktions-Bibliotheken von DBMS-Herstellern eingebunden werden. Der Programmierer muß die Tabellen des Data Dictionary des entsprechenden DBMS kennen, um auf die dort abgelegten Metadaten zugreifen zu können. Für den Zugriff auf andere DBMS muß der Quellcode geändert und rekompilliert werden, um Runtime-Bibliothek des entsprechenden Herstellers einzubinden. Auch CLI-Funktionen sind herstellerspezifisch und abhängig vom jeweiligen DBMS, das sie unterstützt.

Um den Zugriff einer Anwendung auf unterschiedliche DBMS mit demselben Quellcode zu ermöglichen, definiert ODBC ein Standard-CLI, das auf CLI-Spezifikationen der X/Open[4] und ISO[5]/IEC[6] basiert. Die ODBC-Architektur basiert auf dem Client/Server-Modell und hat vier Komponenten (siehe Abbildung 4-1):

- *Anwendungen*: sind verantwortlich für die Interaktion mit dem Benutzer und für den Aufruf von ODBC-Funktionen, die die SQL-Anweisungen absetzen und Ergebnisse entgegennehmen.

- *Treiber-Manager*: lädt die von den Anwendungen angeforderten Treiber; verarbeitet ODBC-Funktionsaufrufe oder gibt sie einem Treiber weiter.

- *Treiber*: verarbeiten ODBC-Funktionsaufrufe, setzen SQL-Anfragen für bestimmte Datenquellen ab und geben Ergebnisse an die Anwendungen zurück.Wenn notwendig, modifizieren die Treiber die Anfragen einer Anwendung, so daß die Anfragen an der Syntax des SQL-Servers anpassen.

- *Datenquellen*: bestehen aus Datenmengen und den entsprechenden, für den Zugriff auf die SQL-Server verwendeten Betriebssystemen, DBMSs und Netzwerken.

[4] X/Open : ein Zusammenschluß von Firmen für die Standardisierung der Computer-Produkte hinsichtlich insbesondere der Portabilität.
[5] ISO : International Organization for Standardization.
[6] IEC : International Electrotechnical Commission.

Durch das ODBC-Konzept wird es Client-Anwendungen ermöglicht, in einheitlicher Weise mit demselben Quellcode ohne Rekompilieren oder Relinken auf verschiedene DBMSs zuzugreifen. Der Zugriff auf verschiedene DBMSs kann sich simultan vollziehen.

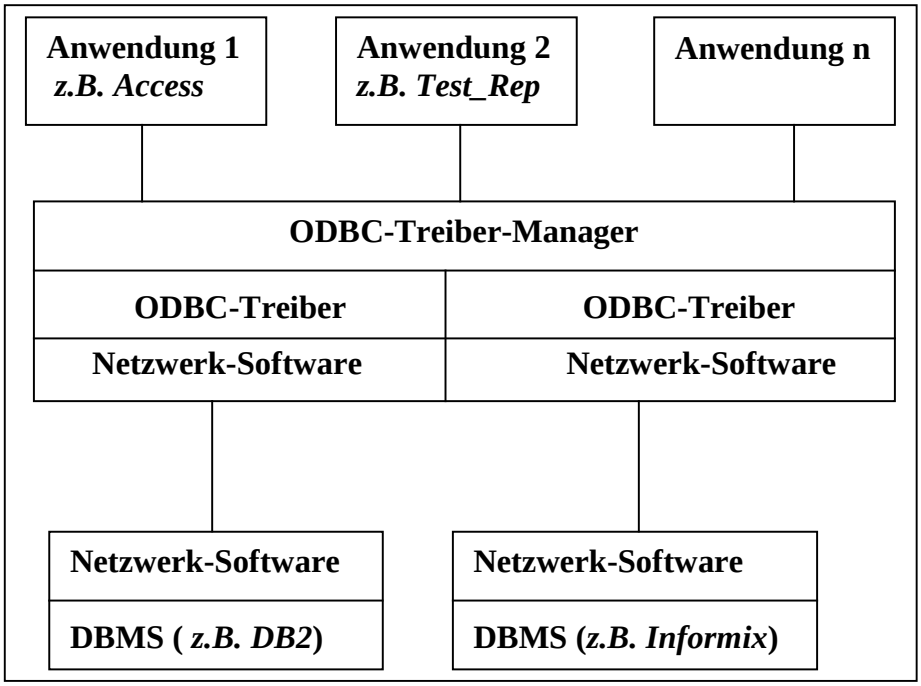


Abbildung 4-1: ODBC-Architektur (Anlehnung an [Rah94])

Insbesondere ermöglicht ODBC auch einen einheitlichen Zugriff auf Metadaten. Die Anwendungen können auf Metadaten der DBMS sowohl durch den Aufruf von Katalogfunktionen als auch durch das Nutzen von sog. Schema-Sichten (INFORMATION_SCHEMA views) zugreifen. Die Schema-Sichten werden durch den ANSI SQL-92 Standard definiert. Für das Retrieval von Metadaten aus diesen Schema-Sichten kann eine Anwendung z.B. eigene SELECT-Anweisungen ausführen, mehrere Sichten verknüpfen oder eine Union von Sichten durchführen. In der vorliegenden Arbeit wurde aber die Verwendung von Katalogfunktionen für das Importieren von Metadaten ausgewählt. Relevante ODBC-Katalogfunktionen sind u.a.:

- *SQLTables*, welches eine Liste von Katalogen, Schemas, Tabellen oder Tabellentypen in der Datenquelle zurückgibt.
- *SQLColumns*, welches eine Liste von Spalten in einer oder mehreren Tabellen zurückgibt.

- *SQLPrimaryKeys*, welches eine Liste von Spalten zurückgibt, die den Primärschlüssel einer einzelnen Tabelle zusammensetzen.

- *SQLForeignKeys*, eine Liste von Fremdschlüsseln in einer spezifischen Tabelle oder eine Liste von Fremdschlüsseln in anderen Tabellen zurückgibt, die den Primärschlüssel in der spezifischen Tabelle referenzieren.

- Andere Katalogfunktionen sind z.B.: *SQLSpecialColumns, SQLTablePrivileges, SQLColumnPrivileges, SQLGetTypeInfo,* etc.

Für genauere Beschreibung der obigen Katalogfunktionen wird auf [ODBC] und [Gei95] verwiesen.

4.3.2 ODBC-Programmierung

ODBC-Handles

Der Einsatz von ODBC durch die Anwendungen erfolgt über Handles. Ein Handle ist eine Anwendungsvariable, in der das System Kontextinformationen über eine Anwendung und ein von der Anwendung verwendetes Objekt ablegen kann. ODBC verwendet drei Arten Handles: Umgebungs-Handles, Verbindungs-Handles und Anweisungs-Handles.

- *Der Umgebungs-Handle* (*henv* – Handle to Environment): ist der globale Kontext-Handle in ODBC. Alle anderen Handles (Verbindungs- und Anweisungs-Handles) werden innerhalb des Kontexts des Umgebungs-Handles ausgeführt.

- *Der Verbindungs-Handle* (*hdbc* – Handle to Database Connection): verwaltet alle Informationen über eine Verbindung. Nach der Allozierung kann er für einen Verbindungsaufbau mit einem DBMS verwendet werden.

- *Der Anweisungs-Handle* (*hstmt* – Handle to Statement): wird für die Verarbeitung von SQL-Anweisungen und Katalogfunktionen verwendet. Wenn der Anweisungs-Handle alloziert ist, kann er für eine beliebige Anzahl von SQL-Anweisungen verwendet werden.

ODBC verwendet das Handle-Konzept auch, um eine robuste Fehlerbehandlung sicherzustellen und die Anwendungsentwickler beim Schreiben von Multithread-Anwendungen zu

unterstützen. Wenn z.B. ein Fehler auftritt, soll die Anwendung den Handle, der in der fehlerhaften Funktion verwendet wurde, der ODBC-Fehlerfunktion übergeben: *SQLError*.

Grundlegende Anwendungsschritte

Im folgenden werden die grundlegenden Schritte von ODBC-Anwendungen beschrieben. Die Anwendungen rufen alle diesen Funktionen jedoch nicht immer genau in der unteren Reihenfolge auf, sondern können auch Abweichungen von diesen Schritten verwenden.

Schritt 1: Verbindungsaufbau

- Allozierung des Umgebungs- und Verbindungs-Handles: *SQLAllocEnv*, *SQLAlloc-Connect*.
- Verbindung zur Datenquelle: *SQLConnect*.
- Setzen der Umgebungs- und Verbindungs-Attribute: *SQLSetEnvAttr*, *SQLSetCon-nectAttr*.

Schritt 2: Initialisierung

- Allozierung des Anweisungs-Handles: SQLAllocStmt.
- Setzen von Anweisungs-Attributen: SQLSetStmtAtt.

Schritt 3: Ausführung der Anweisungen

- Katalogfunktionen: SQLTables, SQLColumns, SQLPrimaryKeys, SQLForeign-Keys...
- Direkte Ausführung: SQLExecDirect.
- Prepared-Ausführung: SQLPrepare, SQLBindParameter, SQLExecute.

Schritt 4: Ermitteln der Ergebnisse

- Select-Anweisungen oder Katalogfunktionen: *SQLBindCol*, *SQLFetch*, *SQLGet-Data*.
- Update-/Delete-/Insert-Anweisungen: *SQLRowCount*.

Schritt 5: Transaktion

In ODBC gibt es zwei Transaktionsarten: Auto-Commit und Manuell-Commit.

- Im Auto-Commit-Modus ist jede Datenbankoperation eine Transaktion, die wenn ausgeführt, dann mit Commit beendet wird.

- Im Manuell-Commit-Modus müssen Anwendungen explizit die Funktion *SQLEndTran* aufrufen, um Transaktionen mit Commit oder Rollback zu beenden. Das gilt für alle Anweisungen innerhalb einer Verbindung oder für alle Verbindungen innerhalb einer Umgebung.

Schritt 6: Verbindungsabbau

- Freigabe des Anweisungs-Handles: *SQLFreeStmt*.
- Freigabe des Verbindungs-Handles: *SQLDisconnect*, *SQLFreeConnect*.
- Freigabe des Umgebungs-Handles: *SQLFreeEnv*.

Konstruktion von SQL-Anweisungen

SQL-Anweisungen, die in den Anwendungen ausgeführt werden, können in drei folgenden Weisen konstruiert werden:

- **Hard-Coded SQL-Anweisungen**

  Anwendungen, die einen festen Task durchführen, enthalten normalerweise hard-coded SQL-Anweisungen. Die Vorteile dabei sind u.a.: die Anweisungen können getestet werden, wenn die Anwendung geschrieben wird; sie sind einfacher zu implementieren als zur Laufzeit konstruierte Anweisungen, und sie vereinfachen die Anwendung.

  Beispiel für das Bilden einer Insert-Anweisung:

```
sprintf(Statement, „INSERT INTO data_store (name, aliasname) „
                   „VALUES (‚%s‘, ‚%s‘) „, Name, Aliasname) ;
SQLExecDirect(hstmt, Statement, SQL_NTS) ;
```

- **Zur Laufzeit konstruierte SQL-Anweisungen**

  Anwendungen, die Adhoc-Analyse durchführen, bilden im allgemeinen SQL-Anweisungen während der Laufzeit. Die Anweisung wird dann mittels *SQLExecDirect* direkt ausgeführt, ohne daß sie vorbereitet werden muß.

  Beispiel:

```
strcpy (Statement, „SELECT „) ;
...
strcat (Statement, „ FROM „) ;
strcat (Statement, TableName) ;
```

- **Vom Benutzer eingegebene SQL-Anweisungen**

  Bei den Anwendungen, die Adhoc-Analyse durchführen, kann der Benutzer SQL-Anweisungen auch direkt eingeben. Das fordert jedoch, daß der Benutzer nicht nur SQL, sondern auch das Schema der gefragten Datenquelle kennt.

  Beispiel:

```
GetSQLStatement(Statement) ;    // Prompt user for SQL   statement
SQLExecDirect(hstmt, Statement, SQL_NTS) ;
```

4.4 Funktionen der Repository-Engine

Nachdem die Repository-Datenbank durch die Definitionssprache von IUS erzeugt wurde, ist der nächste Schritt das Ausfüllen der Datenbank mit den Daten. Diese Daten bestehen aus den Informationen über operative und dispositive Systeme einerseits, und über die Abbildungen zwischen diesen Systemen andererseits. Daraus ergeben sich bei der Implementierung der Datenbank folgende Funktionen:

1. Importieren der Metadaten aus operativen bzw. dispositiven Systemen
2. Definieren der auszuführenden Mappings

Diese zwei Funktionen können jeweils um weitere Funktionalitäten wie Anzeigen und Löschen erweitert werden. Neben dem Import von benötigten Quell- bzw. Ziel-Metadaten in die Repository-Datenbank soll das Repository noch in der Lage sein, die importierten Metadaten anzuzeigen und die nicht mehr gebrauchten Metadaten aus dem Repository zu entfernen. Ähnlicherweise können neue Mappings definiert, vorhandene Mappings angezeigt oder gelöscht werden. Für diese Zwecke wird ein komplexes Implementierungsprogramm erstellt, das sich aus drei Haupt-Units zusammenfaßt :

- ***implement.c***: enthält das Hauptprogramm, das die Funktionen anderer Units aufruft.
- ***metadata.c***: enthält alle Funktionen für das Importieren, Anzeigen, Löschen u.a. von Metadaten der Datenquellen.
- ***mapping.c***: enthält alle Funktionen, die für die Definition, das Anzeigen, das Löschen u.a. von Mappings benötigt werden.

Das Hauptprogramm ist in der Unit ***implement.c*** enthalten, wobei sich die Reihenfolge der Schritte nach den im Abschnitt 4.3 behandelten grundlegenden Anwendungsschritten

richtet. Vor allem muß die Verbindung mit der Repository-Datenbank aufgebaut werden. Dazu gehören die Allozierung des Umgebungs-, des Verbindungs- und des Anweisungs-Handles (*henv*, *hdbc* und *hstmt*) sowie die Verbindung zur Datenquelle. Von dem Benutzer wird aber gefordert, die Angabe des Repository-Namens, der Benutzer-ID und des Paßworts zu machen (Abbildung 4-2).

Die eigentliche Anwendung beginnt mit einem Hauptmenü, wobei eine der o.g. Repository-Funktionen ausgewählt werden kann (Abbildung 4-2). Innerhalb einer Verbindung (Repository-Session) kann man diese Operationen mehrmals und in beliebiger Reihenfolge aufrufen.

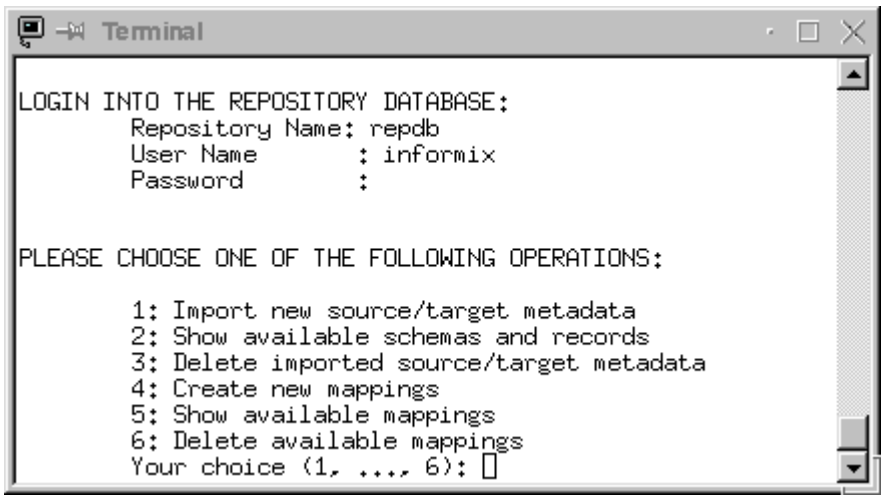


Abbildung 4-2: Login und Operations-Auswahl

In dieser Anwendung ist ein Manuell-Commit-Modus sinnvoller als ein Auto-Commit-Modus, weil dadurch die Transaktionsausführung kontrolliert werden kann. In ODBC kann eine Anwendung durch den Aufruf von *SQLEndTrans* Transaktionen explizit mit COMMIT oder ROLLBACK beenden. Im Normalfall werden Transaktionen durch Ausführung der COMMIT-Operation erfolgreich (d.h. normal) beendet:

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
```

Im Fehlerfall, z.B. aufgrund von erkannten Eingabefehlern oder sonstigen in der Anwendung erkannten Ausnahmesituationen wird eine explizite ROLLBACK-Operation ausgeführt und die Transaktion abgebrochen (abnormales Ende):

```
SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
```

Nach der Transaktions-Behandlung wird die Verbindung zur Repository-Datenbank durch die Freigabe aller Handles getrennt und das Programm dann beendet.

Im weiteren wird auf jede der obigen Repository-Funktionen ausführlicher eingegangen.

4.4.1 Importieren neuer Quell-/Ziel-Metadaten

Der Import von neuen Metadaten erfolgt durch eine Reihe von Insert-Funktionen wie *insertDatastore*, *insertDatabase*, *insertSchema*, *insertRecord*, *insertElement* und *insertReference*. In diesen Funktionen werden die Insert-Anweisungen als hard-coded, parameterisierte Anweisungen definiert. Beispiel für eine solche Anweisung ist:

```
insert into data_store (name, aliasname) values (?, ?)
```

Die Parameter werden mittels *SQLBindParameter* gebunden, wobei jedem Parameter eine Spalte der Tabelle zugeordnet wird und der Datentyp sowie die Länge des Attributs festgelegt werden. Beispiel für das Binden des 1. Parameters der obigen Anweisung:

```
SQLBindParameter ( hstmt, 1, SQL_PARA_INPUT, SQL_C_CHAR, SQL_CHAR,
                   20, 0, szName, 0, &cbName )
```

Für diese Insert-Anweisungen ist eine Prepared-Ausführung durch zwei Funktionsaufrufe *SQLPrepare* und *SQLExecute* vorteilhafter als eine direkte Ausführung, weil sie mehrmals ausgeführt werden können. Das gilt insbesondere für Anweisungen, die Daten in die Tabellen *record*, *element* oder *reference* einfügen.

Ausfüllen der Tabelle *data_store*

Das Einfügen in die Tabelle *data_store* wird durch die Funktion *insertDatastore* durchgeführt:

```
insertDatastore (HENV henv, HDBC hdbc, HSTMT hstmt, char *SqlState,
                 char *datastore )
```

Die Variablen *henv*, *hdbc*, *hstmt* und *SqlState* stellen jeweils den Umgebungs-Handle, den Verbindungs-Handle sowie den Anweisungs-Handle für die Repository-Datenbank und den Code für die Fehlerbehandlung dar. Daneben steht *datastore* für den Namen des Data Store, der in weiteren Funktionen als Parameter verwendet werden kann.

Nachdem der Name und der Aliasname des Data Store (manuell) eingegeben werden, wird die Anweisung ausgeführt und ein neuer Datensatz in die Tabelle *data_store* eingefügt (Abbildung 4-3).

Ausfüllen der Tabelle *database*

Das übernimmt die Funktion:

```
insertDatabase (HENV henv, HDBC hdbc, HSTMT hstmt, char *SqlState,
                char *datastore )
```

wobei die ersten vier Parameter ähnlich wie bei der Funktion *insertDatastore* interpretiert werden und der Parameter *datastore* für den Namen des Data Store steht, das die Datenbank (*database*) repräsentiert. (Nach unserer Vereinbarung haben das Data Store und die Datenbank den gleichen Namen.)

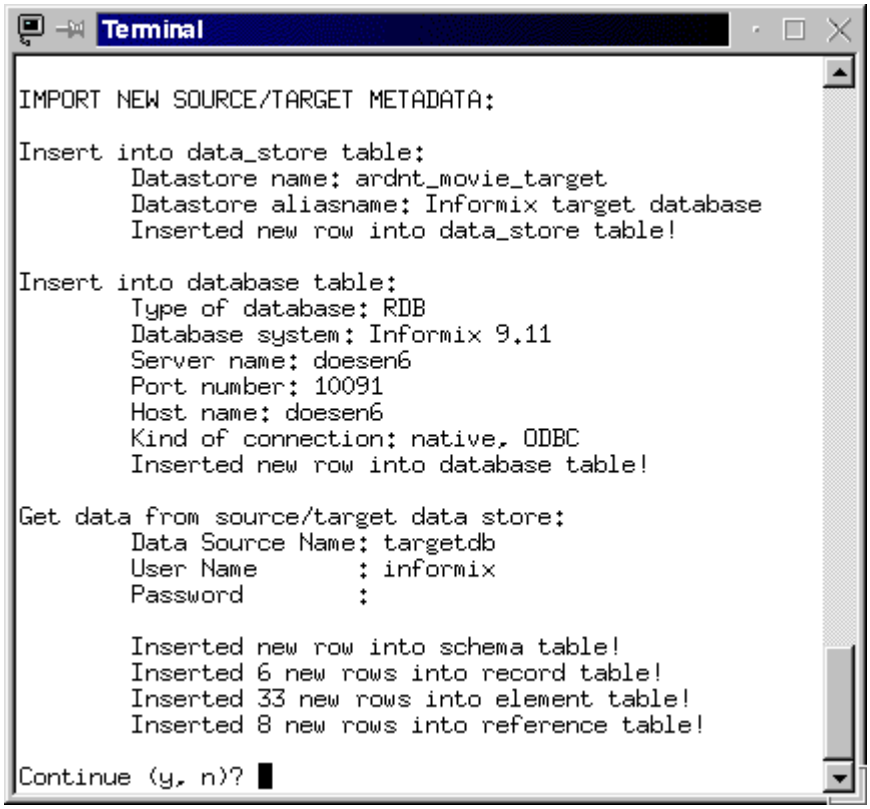


Abbildung 4-3: Metadaten-Import

Für das Einfügen in die Tabelle *database* werden folgende Daten auch manuell eingegeben (Abbildung 4-3):

Typ der Datenbank	Port-Nummer
Datenbanksystem	Host-Name
Server-Name	Verbindungsart

Importieren von Metadaten

Informationen über Schemas, Tabellen, Spalten und referentielle Integrität können aus dem Data Dictionary des jeweiligen DBMS importiert werden. Dafür stellt ODBC zahlreiche Katalogfunktionen zur Verfügung. Um die Tabellen *schema*, *record*, *element* und *reference* auszufüllen, werden folgende Katalogfunktionen verwendet: *SQLTables*, *SQLColumns*, *SQLPrimaryKeys* und *SQLForeignKeys*.

Der Import von solchen Metadaten wird durch die Funktion *getMetadata* der Unit ***metadata.c*** vorgenommen, die für eine bestimmte Datenquelle eine Tabellenliste sowie eine Spaltenliste, eine Liste der Primärschlüssel und eine Liste der Fremdschlüssel der zugehörigen Tabellen als ihre Parameter zurückgibt.

```
getMetadata (struct tableList *restab, struct pkList *respk,
             struct fkList *resfk, struct colList *rescol)
```

Der Zugriff auf eine Datenquelle erfolgt in ähnlicher Weise wie der oben beschriebene Zugriff auf die Repository-Datenbank. Für die jeweilige Datenbank-Verbindung werden der Name der Datenquelle, die Benutzer-ID und das Paßwort verlangt (Abbildung 4-3). Nach der erfolgreichen Verbindung können Anweisungen, in diesem Fall Katalogfunktionen, ausgeführt werden.

In der Funktion *getMetadata* werden die oben genannten Katalogfunktionen wie folgt verwendet:

- ***Import von Tabelleninformation:***

```
SQLTables (hstmt,                   /* Anweisungs-Handle */
           NULL, 0,                 /* Alle Kataloge */
           NULL, 0,                 /* Alle Schemas */
           NULL, 0,                 /* Alle Tabellen */
           „TABLE“, SQL_NTS);       /* Tabellentyp „TABLE“ */
```

Die Ergebnismenge der obigen Funktion enthält eine Liste aller Tabellen mit dem Tabellentyp „TABLE“ in allen Katalogen sowie in allen Schemas. Wird der Tabellentyp nicht extra angegeben (anstelle von *„TABLE“, SQL_NTS* steht *NULL, 0*), werden Tabellen aller Typen, z.B. Systemstabellen, Synonymtabellen, Sichten, etc. zurückgegeben.

- ***Import von Spalteninformation:***

```
SQLColumns (hstmt,                  /* Anweisungs-Handle */
            NULL, 0,                /* Alle Kataloge */
```

```
                NULL, 0,                /* Alle Schemas */
                (char*)table, SQL_NTS,  /* Tabelle table */
                NULL, 0 );              /* Alle Spalten */
```

Diese Katalogfunktion liefert als Ergebnismenge eine Liste aller Spalten in der Tabelle namens *table*.

- ***Import von Information zu Primärschlüsseln:***

```
SQLPrimaryKeys (hstmt,                  /* Anweisungs-Handle */
                NULL, 0,                /* Alle Kataloge */
                NULL, 0,                /* Alle Schemas */
                (char*)table, SQL_NTS) ;  /* Tabelle table */
```

In der Ergebnismenge ist der Primärschlüssel oder die Liste der Schlüsselkandidaten der Tabelle *table* und dessen Folge in der Liste (key sequence) enthalten.

- ***Import von Information zu Fremdschlüsseln:***

```
SQLForeignKeys (hstmt,                  /* Anweisungs-Handle */
                NULL, 0,                /* Alle PS-Kataloge */
                NULL, 0,                /* Alle PS-Schemas */
                (char*)table, SQL_NTS,  /* PS-Tabelle table */
                NULL, 0,                /* Alle FS-Kataloge */
                NULL, 0,                /* Alle FS-Schemas */
                NULL, 0) ;              /* Alle FS-Tabllen */
```

Dadurch wird eine Liste der Fremdschlüssel in allen Tabellen zurückgegeben, die auf den Primärschlüssel der Tabelle *table* referenzieren.

Ausfüllen der Tabelle *schema*

Die durch die Funktion *getMetadata* zurückgegebene Tabellenliste ist eine Struktur, die drei Komponenten enthält: *schema*, *tablename* und *remarks*.

```
struct tableList{
    char     **schema;       /* Name des zugehörigen Schemas */
    char     **tablename;    /* Name der Tabelle */
    char     **remarks;      /* Bemerkung zur Tabelle */
    long     tablecount;     /* Tabellenanzahl in der Liste */
};

struct tableList tlist;      /* Definition der Variable tlist */
```

Der Wert der ersten Komponente (*schema*) kann als Parameter für die Funktion *insert-Schema* verwendet werden:

```
insertSchema (HENV henv, HDBC hdbc, HSTMT hstmt, char *SqlState,
              char *schema, char *datastore )
```

Auch wie bei der Funktion *insertDatabase* steht der Parameter *datastore* für den Namen des Data-Store, zu dem das Schema gehört.

Ausfüllen der Tabelle *record*

Die Struktur der Tabellenliste wird auch als Parameter in der Funktion:

```
insertRecord (HENV henv, HDBC hdbc, HSTMT hstmt, char *SqlState,
              struct tableList tlist )
```

verwendet, um Informationen über die Tabellen der Datenquellen in die Tabelle *record* einzufügen.

Die Zuordnung zwischen den Komponenten der Tabellenliste und den Spalten der Tabelle *record* ist wie folgt:

Struktur *tableList*		**Tabelle *record***
schema	-	*s_name* (Name des Schemas)
tablename	-	*name* (Name des Records)
remarks	-	*aliasname* (Aliasname des Records)

Ausfüllen der Tabelle *element*

Durch die Funktion *getMetadata* wird neben der Tabellenliste noch eine Spaltenliste dieser Tabellen mit folgender Struktur zurüchgegeben:

```
struct colList{
    char     **schema;       /* Name des zugehörigen Schemas */
    char     **tablename;    /* Name der zugehörigen Tabelle */
    char     **colname;      /* Name der Spalte */
    char     **remarks;      /* Bemerkung zur Spalten */
    char     **datatype;     /* Datentyp der Spalte */
    int      colsize;        /* Länge der Spalte */
    int      *decdigits;     /* Skalar der (numerischen) Spalte */
    int      nullable;       /* Nullwert der Tabelle */
    long     colcount;       /* Spaltenanzahl in der Liste */
};
struct colList clist;        /* Definition der Variable clist */
```

Weil die Komponenten dieser Spalten-Struktur jeweils den Spalten der Tabelle *element* entsprechen, kann sie als Parameter in der Funktion *insertElement* für das Einfügen in die Tabelle *element* verwendet werden.:

```
insertElement (HENV henv, HDBC hdbc, HSTMT hstmt, char *SqlState,
               struct colList clist )
```

Ausfüllen der Tabelle *reference*

Um die Tabelle *reference* auszufüllen, werden die Listen der Primärschlüssel und Fremdschlüssel, die bei der Funktion *getMetadata* zurückgegeben werden, als Parameter in der folgenden Funktion verwendet.:

```
insertReference (HENV henv, HDBC hdbc, HSTMT hstmt, char *SqlState,
                 struct pkList pklist, struct fkList fklist )
```

Die Struktur dieser Listen sieht wie folgt aus:

```
struct pkList{
    char    **schema;         /* Name des Schemas */
    char    **table;          /* Name der PS-Tabelle */
    char    **pkcolumn;       /* Name des Primärschlüssels */
    int     keyseq;           /* Folge des PS in der Liste */
    long    count;            /* Anzahl */
};

struct fkList{
    char     **schema;        /* Name des Schemas */
    char     **pktable;       /* Name der PS-Tabelle */
    char     **pkcolumn;      /* Name des Primärschlüssels */
    char     **fktable;       /* Name der FS-Tabelle */
    char     **fkcolumn;      /* Name des Fremdschlüssels */
    long     count;           /* Anzahl */
};

struct pkList pklist;         /* Definition der Variable pklist */
struct fkList fklist;         /* Definition der Variable fklist */
```

Die Spalten *schema*, *pktable*, *pkcolumn* und *pkseq* der Tabelle *reference* werden durch die Komponenten der Primärschlüssel-Liste ausgefüllt. Das ist Information über Primärschlüssel bzw. Schlüsselkandidaten einer bestimmten Tabelle. Wird diese Tabelle von keiner Tabelle referenziert, dann haben die beiden Spalten *fktable* und *fkcolumn* der Tabelle *reference* Nullwerte. Andernfalls dient deren Primärschlüssel als Fremdschlüssel in den referenzierenden Tabellen und die Spalten *fktable* und *fkcolumn* der Tabelle *reference* besitzen entsprechende Werte aus der Fremdschlüssel-Liste. Das Einfügen dieser Werte in die Ta-

belle *reference* erfolgt unter der Bedingung, daß die Objekte Schema, Primärschlüssel-Tabelle und Primärschlüssel in beiden Listen identisch sind.

In der Abbildung 4-3 wird das Einfügen der Metadaten bezüglich des Schemas der Ziel-Datenbank ***ardnt_movie_target*** demonstriert. Bei einer erfolgreichen Durchführung der Anweisungen wird mitgeteilt, wieviele Datensätze in die jeweilige Tabelle eingefügt wurden.

4.4.2 Anzeigen der verfügbaren Schemas und Records

Die Metadaten der Quell- bzw. Ziel-Datenbanken werden in den Tabellen *data_store, database, schema, record* und *element* gespeichert. Zur Demonstration wird in dieser Arbeit auf das Anzeigen der Information über Schemas und Records beschränkt. Außerdem ergibt sich aus den Schemas auch Information über die Data Stores, die von diesen Schemas beschrieben werden.

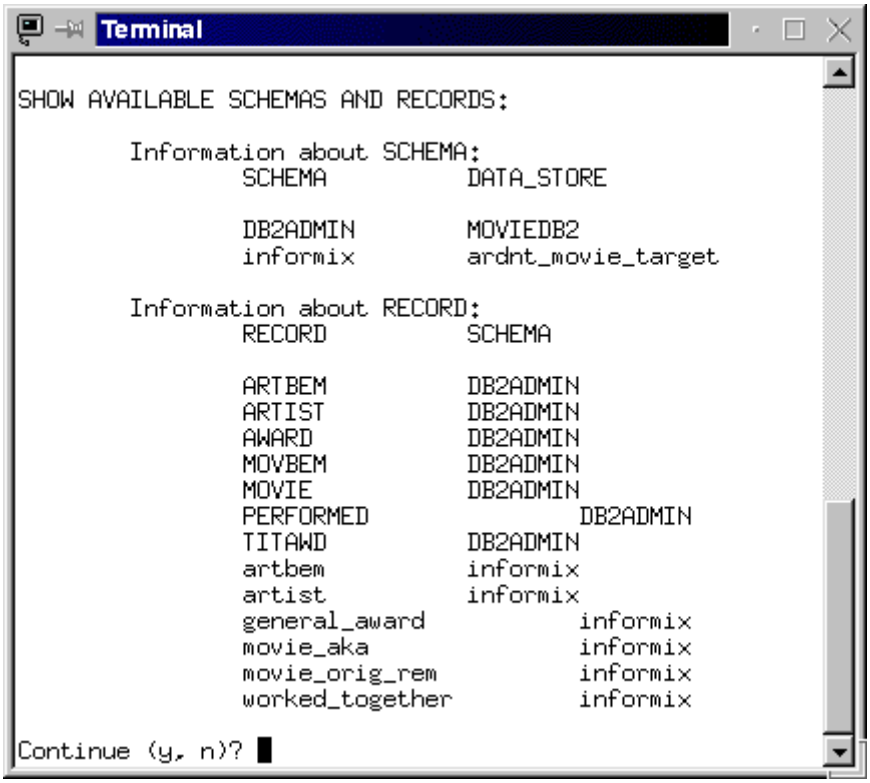


Abbildung 4-4: Metadaten-Anzeigen

Das Anzeigen der verfügbaren Schemas und Records wird durch folgende Funktion realisiert:

```
showMetadata (HENV henv, HDBC hdbc, HSTMT hstmt, char *SqlState )
```

Dabei werden zwei SQL-Select-Anweisungen auf die Tabellen *schema* und *record* angewendet, um Informationen über Schemas und Records zu holen. Aus der Abbildung 4-4 ist entnehmbar, daß das Schema *DB2ADMIN* dem Data Store *MOVIEDB2* und das Schema *informix* dem Data Store *ardnt_movie_target* gehört. Die Tabellen des jeweiligen Schemas wurden auch angezeigt.

4.4.3 Löschen importierter Quell-/Ziel-Metadaten

Werden manche Datenhaltungssysteme bei der Mapping-Definition nicht mehr benötigt, können die Metadaten über sie aus dem Repository entfernt werden. Auch wenn der Inhalt oder die Struktur eines Datenhaltungssystems verändert ist, können seine Metadaten gelöscht und danach wieder erneut importiert werden. Die Löschen-Funktion muß nicht auf alle Tabellen, die die Metadaten enthalten, angewendet werden. Bei der Definition der Tabellen *data_store*, *database*, *schema*, *record*, *element* und *reference* wurde ein kaskadierendes Löschen in der Fremdschlüssel-Klausel vereinbart, so daß das Löschen der Datensätze einer Tabelle auch das Löschen der entsprechenden Datensätze in den referenzierenden Tabellen impliziert. Um die Metadaten über ein bestimmtes Datenhaltungssystem zu löschen, muß demzufolge nur der Datensatz des zugehörigen Data Store in der Tabelle *data_store* gelöscht werden (Abbildung 4-5). Dann werden automatisch alle Datensätze der entsprechenden Databases, Schemas, Records, Elements und References mitgelöscht.

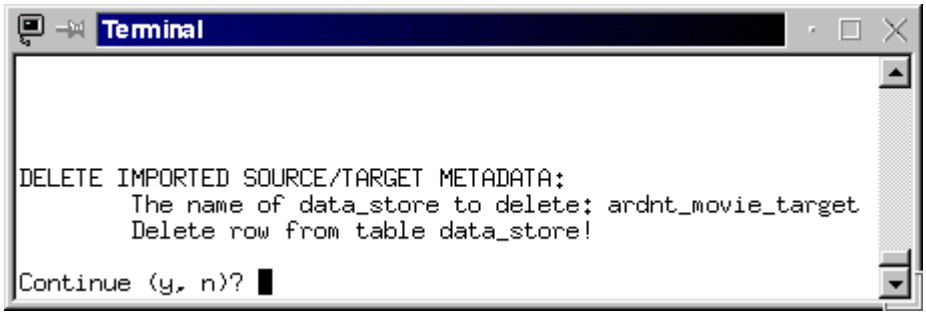


Abbildung 4-5: Metadaten-Löschen

Nachdem z.B. das Data Store *ardnt_movie_target* gelöscht wird, werden alle zugehörigen Metadaten wie Schema, Records, ... auch aus dem Repository entfernt (Abbildung 4-6).

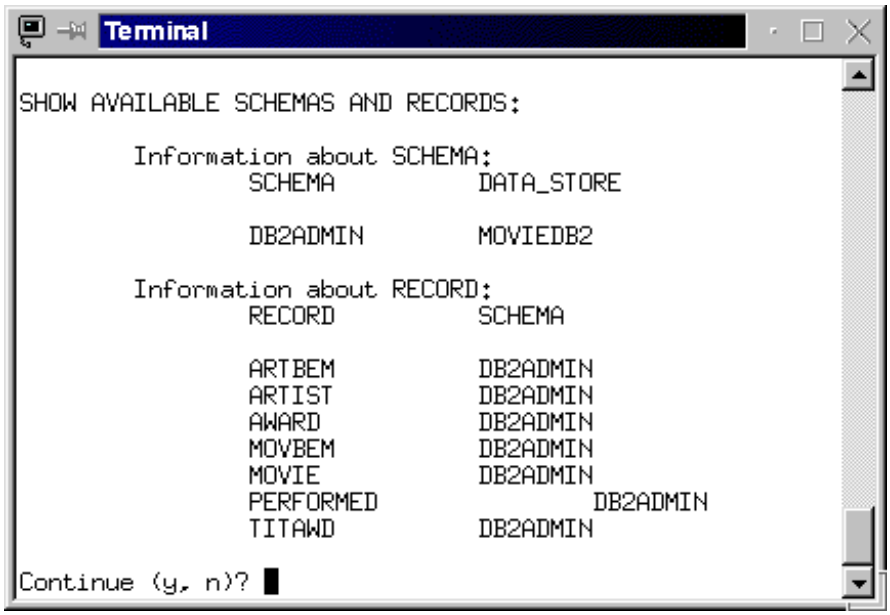


Abbildung 4-6: Metadaten-Anzeigen (2)

4.4.4 Erzeugen neuer Mappings

Neben dem Import von Metadaten aus Quell- und Zielsystemen hat die Repository-Engine noch die wichtige Funktion, Definition von Abbildungen bzw. Mappings zwischen diesen Systemen zu ermöglichen. Das Mapping-Erzeugen wird also auf das Definieren und Speichern von Mappings im Repository beschränkt. Das entspricht auch dem Einfüge-Vorgang in die übrigen Tabellen der Repository-Datenbank (*mapping_folder*, *mapping*, *record_mapping*, *transformation*, *expression*, *filter* und *filter_clausel*), die dann alle relevanten Informationen über ein Mapping enthalten. Im Anhang C steht die Beschreibung der vier Test-Mappings, die in diesem Abschnitt für die Demonstration zum Mapping-Erzeugen verwendet werden.

Ausfüllen der Tabelle *mapping_folder*

Mehrere Mappings können in einem Folder zusammen gespeichert werden. Daher muß beim Erzeugen eines neuen Mappings zugleich festgelegt werden, zu welchem Folder es gehört. Entweder ist der Folder bereits vorhanden, oder eine neuer Folder muß zunächst erzeugt werden. Von diesen beiden Möglichkeiten kann eine ausgewählt werden: bei der 1. Möglichkeit wird der vorhandene Folder durch die Angabe seines Namens bestimmt; bei der 2. Möglichkeit soll die Funktion *insertMFolder* aufgerufen werden, in der nach dem Namen sowie Aliasnamen des neuen Folders gefragt wird. Abbildung 4-7 zeigt ein Bei-

spiel, in dem ein Mapping-Folder namens *folder1* erzeugt wird, dessen Aliasname *DB2-Informix* bedeutet, daß dieser Folder alle Mappings zwischen den Datenbanken DB2 und Informix enthält.

Ausfüllen der Tabelle *mapping*

Nach der Bestimmung des zugehörigen Folders kann nun die Erzeugung des einzelnen Mappings begonnen werden. Zuerst werden Name und Aliasname des jeweiligen Mappings angegeben, die durch die Funktion *insertMapping* in die Tabelle *mapping* eingefügt werden.

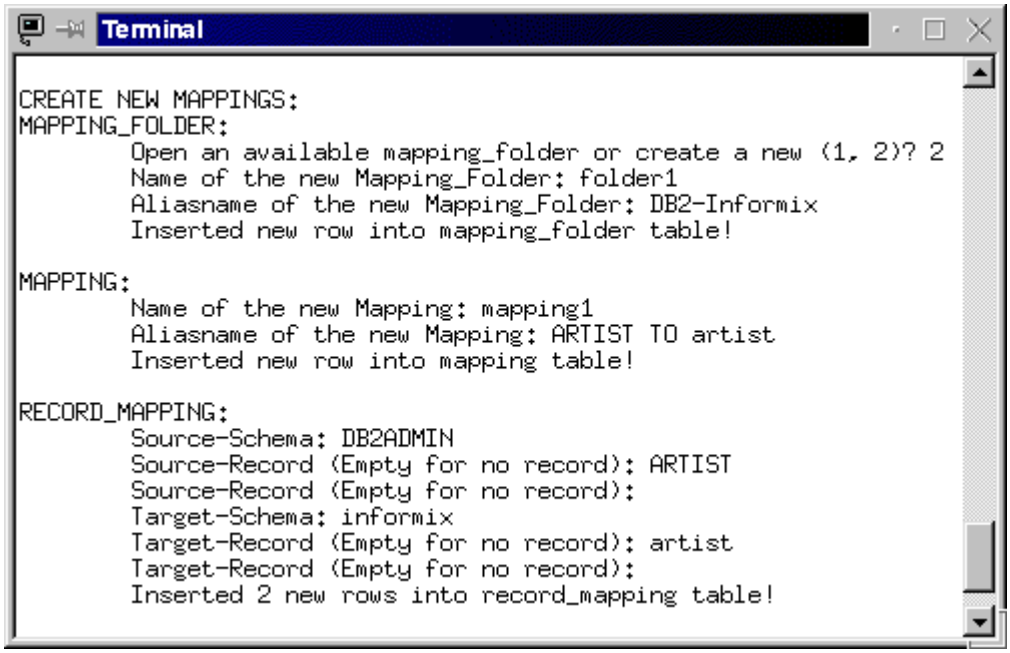


Abbildung 4-7: Mapping-Erzeugen (Mapping 1)

Als erstes Beispiel wurde das **Mapping 1** (siehe Anhang C) genommen, das z.B. *mapping1* genannt wird und eine Abbildung aus der Quell-Tabelle *ARTIST* in die Ziel-Tabelle *artist* ist (Abbildung 4-7).

Ausfüllen der Tabelle *record_mapping*

Nachdem das Mapping durch seinen Namen identifiziert wurde, ist der nächste Schritt die Angabe von näheren Informationen darüber, welche Quell- bzw. Ziel-Records am Mapping beteiligt sind. Diese Informationen werden dann in die Tabelle *record_mapping* durch die Funktion *insertRecordMapping* eingefügt. Die Insert-Anweisung wird als eine hard-

coded und Prepared-Anweisung definiert und kann daher mehrmals mittels *SQLExecute* ausgeführt werden.

Die Records sind nur innerhalb eines bestimmten Schemas eindeutig. Deshalb muß bei der Angabe von Records auch über das zugehörige Schema informiert werden. In dieser Arbeit wird die Implementierung auf zwei relationale DBS beschränkt, wobei jeder dieser Datenbanken nur ein Schema zugeordnet wird. Deshalb müssen zuerst das Quell- bzw. Ziel-Schema und dann die entsprechenden Records eingegeben werden.

Am Mapping *mapping1* sind zwei Records beteiligt: der Quell-Records *ARTIST* aus dem Schema *DB2ADMIN* und der Ziel-Record *artist* aus dem Schema *informix*. Entsprechend werden zwei Datensätze in die Tabelle *record_mapping* eingefügt (Abbildung 4-7).

In der Funktion *insertRecordMapping* wird der Vorteil einer Prepared-Anweisung genutzt, indem sie nur einmal kompiliert und nach jeder Angabe der einzelnen Records wieder ausgeführt wird.

Ausfüllen der Tabelle *transformation*

Ein Mapping repräsentiert eine Abbildung zwischen Quell- und Zielsystemen auf der Record-Ebene. Auf der Element/Attribut-Ebene wird eine Abbildung durch Transformationen und/oder Filter dargestellt, aus denen das Mapping besteht und die bei der Datenbewegung tatsächlich durchgeführt werden. Nach der Angabe der am Mapping beteiligten Records wird gefragt, ob das Mapping Transformation(en) und/oder Filter enthält. Enthält es Transformation(en), kann der Vorgang der Transformations-Definition mit dem Aufruf der Funktion *insertTransformation* begonnen werden.

Im Unterschied zu allen bisherigen Insert-Anweisungen wird in dieser Funktion keine hard-coded, sondern eine zur Laufzeit konstruierte Anweisung ausgeführt. Der Grund dafür liegt darin, daß die Spalte *source_elem* der Tabelle *transformation* ein Kollektions-Typ (SET of VARCHAR) ist und demzufolge nicht als ein Parameter einer hard-coded Anweisung durch den ODBC-Aufruf *SQLBindParameter* gebunden werden kann.

Für das Einfügen in die Tabelle *transformation* werden u.a. eingegeben:

- Name der Transformation (eindeutig),
- Name des Ziel-Elements,
- Namen der Quell-Elemente.

Nachdem die Anweisung aufgrund der Angabe aller nötigen Werte konstruiert wurde, wird sie durch den Funktionsaufruf *SQLExecDirect* (direkt) ausgeführt. Ist die Ausführung erfolgreich beendet, wird die Tabelle *transformation* mit einem neuen Datensatz eingefügt.

Abbildung 4-8: Transformations-Definition (Mapping 1)

Als Demonstration der Transformations-Definition wird das obige **Mapping 1** (*mapping1*) weitergeführt (Abbildung 4-8). Im folgenden wird die Lösung dieses Mappings kurz geschildert:

Zu a) Attributsplit:

Source.artist.name -> Target.artist.lastname, Target.artist.firstname

Angenommen, daß es in der Systembibilothek eine vordefinierte Funktion SUBSTR gibt, die einen Teil einer gegebenen Zeichenkette liefert, wobei deren Teile durch bestimmte Trennzeichen wie „ , “, „ ; “ etc. getrennt werden. Weil die Spalte *NAME* der Tabelle *ARTIST* die Form *firstname, lastname* hat, kann sie unter Anwendung der Funktion SUBSTR in zwei Teile aufgeteilt werden:

```
artist.firstname = SUBSTR (ARTIST.NAME, 1)
                           (der 1. Teil der Zeichenkette ARTIST.NAME)
artist.lastname  = SUBSTR (ARTIST.NAME, 2)
                           (der 2. Teil der Zeichenkette ARTIST.NAME)
```

Diese Attribut-Aufspaltung kann also als zwei Transformationen (namens z.B. *transf1a1* und *transf1a2*) betrachtet werden, deren Ausdrücke jeweils eine Funktion sind. An diesen Transformationen nehmen *ARTIST.NAME* als Quell-Element und *artist.firstname* bzw. *artist.lastname* als Ziel-Element teil (siehe Abbildung 4-8).

Zu b) Konversion inklusive Berechnung:

Source.artist.birth_date -> Target.artist.age

Hat das Attribut *ARTIST.DEATH_DATE* einen Wert (NOT NULL), kann dem Ziel-Element *artist.age* ein Nullwert zugewiesen werden (*age = NULL*). Andernfalls kann *artist.age* durch folgenden Transformationsausdruck (*transf1b* aus der Abbildung 4-8) berechnet werden:

```
artist.age = GET_DATE_PART(SYSDATE,YY) – ARTSIT.BIRTH_DATE/10000
         (artist.age = aktuelles Jahr – Geburtsjahr)
```

Dieser Ausdruck besteht aus zwei Teilausdrücken und dem binären Operator „-“. Der erste Teilausdruck ist eine Funktion (*GET_DATE_PART*), die einen speziellen Teil (Jahr, Monat, ...) eines Datums als einen Integer-Wert zurückgibt. Diese Funktion fordert zwei Parameter, den einen als Datum-/Zeit-Datentyp und den anderen als Datum-/Zeit-Format (z.B.: „YY“=Jahr, „MM“=Monat, „DD“=Tag, ...). Der zweite Teilausdruck enthält wie-

derum zwei (elementare,) durch den ‚/'-Operator verbundene Ausdrücke: das Quell-Element *ARTIST.BIRTH_DATE* und die numerische Zahl 10000. (Angenommen, daß *BIRTH_DATE* in der Form YYYYMMDD, z.B. 19240815, gespeichert wird, kann das Geburtsjahr durch die Division von *BIRTH_DATE* durch 10000 berechnet werden.)

Zu c) Typkonversion: *Source.artist.sex -> Target.artist.gender*

Dazu können zwei Transformationen (*transf1c1* und *transf1c2*) vorgenommen werden, jeweils mit der vordefinierten Funktion IFF(*condition*, *value*):

```
artist.gender = IFF (ARTIST.SEX = m, 0)
artist.gender = IFF (ARTIST.SEX = f, 1)
```

Wenn die Bedingung *ARTIST.SEX = m* erfüllt, nimmt das Ziel-Element *artist.gender* den Wert 0 an. Wenn *ARTIST.SEX = f*, dann *artist.gender* = 1.

Zu d) 1:1 – Abbildung:

Die 1:1 – Abbildung der übrigen Attribute kann auch als Transformationen beschrieben werden, wobei jede von ihnen ein elementarer Ausdruck (Quell-Element) ist:

```
artist.aid = ARTIST.AID;
artist.birth_place = ARTIST.BIRTH_PLACE;
artist.birth_cntry = ARTIST.BIRTH_CNTRY;
artist.death_date = ARTIST.DEATH_DATE;
artist.realname = ARTIST.REALNAME
```

Es gibt jedoch die Möglichkeit, diese Abbildungen durch einen Filter darzustellen. Deshalb werden die obigen Transformationen nicht demonstriert, anstelle davon wird die Definition des Filters im nächsten Abschnitt (siehe Abbildung 4-9) behandelt.

Ausfüllen der Tabelle *expression*

In der Tabelle *transformation* werden nur die Namen der Transformation und der daran beteiligten Elemente gespeichert. Der Transformations-String wird aber in einem oder mehreren Datensätzen der Tabelle *expression* abgelegt. Deshalb wird gleich nach dem Beenden der Funktion *insertTransformation* die Funktion *insertExpression* aufgerufen. In dieser Funktion wird die Eingabe des Transformations-Ausdruckes gefordert, der anschließend als Parameter in der Funktion *getExpression(char *expr_str, struct expr *expr)* verwendet wird. Die Funktion *getExpression* interpretiert den eingegebenen Ausdruck und gibt Fehlermeldungen zurück, wenn einer der folgenden Syntaxfehler auftritt:

- *Operatoren (+, -, *, /), Klammer-Zu oder Komma steht am Anfang des Ausdruckes;*
- *Operatoren (+, -, *, /), Klammer-Auf oder Komma steht am Ende des Ausdruckes;*
- *Zwei Zeichen wie z.B. +), ( ),) (... stehen nebeneinander;*
- *Die Anzahl der Klammer-Auf und Klammer-Zu ist nicht gleich;*
- *Die Anzahl der Parameter einer Funktion ist größer als zwei.*

Solange der Ausdruck syntaktisch noch nicht korrekt ist, wird gefordert, ihn nochmals einzugeben (Beispiel: der Ausdruck von *transf1a1* in der Abbildung 4-8). Ist der Ausdruck fehlerfrei, wird eine Expression-Struktur zurückgegeben:

```
struct expr {
   char        **name;          /* Name des Ausdruckes */
   char        **left;          /* Der linke Teilausdruck oder
                                   1. Parameter der Funktion */
   char        **right;         /* Der rechte Teilausdruck oder
                                   2. Parameter der Funktion */
   char        **operator;      /* Der Operator */
   char        **fctname;       /* Name der Funktion */
   long        count;           /* Anzahl der Listeelemente */
};
```

Eine Insert-Anweisung wird aufgrund der zurückgegebenen Expressions-Struktur konstruiert und dann direkt ausgeführt, so daß neue Datensätze in die Tabelle *expression* eingefügt werden.

Ausfüllen der Tabelle *filter*

Enthält ein Mapping einen Filter, wird die Funktion *insertFilter* aufgerufen, um nähere Informationen über das gegebene Mapping bezüglich des Filterungsprozesses zu erfahren und sie dann in die Tabelle *filter* einzufügen.

Ähnlich wie beim Transformations-Einfügen wird hier die Insert-Anweisung auch zur Laufzeit konstruiert und direkt ausgeführt, weil die Spalten *simple_transfs* und *aggr_transfs* der Tabelle *filter* Kollektions-Typen sind. Außer dem Namen des Mappings, zu dem der Filter gehört, müssen alle übrigen Spalten der Tabelle *filter* (*name*, *simple_transfs*, *aggr_transfs*, *nf_type* und *grby_elem*) noch mit den entsprechenden Werten belegt werden.

Weil ein Filter-Ausdruck (SQL-Ausdruck) ziemlich komplex ist, wird er nicht wie bei der Transformation als ein gesamter Ausdruck auf einmal eingegeben. Aus Einfachheitsgründen der Demonstration werden Informationen über einzelne Teile des Filter-Ausdruckes abgefragt und aufgrund dieser Informationen wird die Anweisung entsprechend der Syntax des objektrelationalen Informix gebildet.

Für das Einfügen in die Tabelle *filter* werden u.a. eingegeben:

- Name des Filters (eindeutig),
- Anzahl der Simple-Transformationen (>= 0) und die zugehörigen Quell- bzw. Ziel-Elemente,
- Anzahl der Aggregat-Transformationen (>= 0) und die zugehörigen Quell-, Ziel-Elemente und Aggregat-Operatoren,
- Normalform-Typ: „DNF“, „KNF“ oder „“ falls keine Where-Klausel,
- Group by- Element („“ für kein).

Wie oben schon erwähnt wird, können die 1:1 – Abbildungen des Mappings *mapping1* (siehe auch Anhang C) durch einen Filter-Ausdruck dargestellt werden:

```
SELECT   ARTIST.AID as artist.aid, ARTIST.BIRTH_PLACE as
         artist.birth_place, ARTIST.BIRTH_CNTRY as artist.birth_cntry,
         ARTIST.DEATH_DATE as artist.death_date, ARTIST.REALNAME as
         artist.realname
INTO     artist
FROM     ARTIST
```

Interpretation (siehe auch Abbildung 4-9):

- Das Mapping *mapping1* enthält einen Filter, z.B. namens *filter1*;
- Ziel-Record: *artist*; Quell-Record: *ARTIST*;
- Der Filter besteht aus (keiner Aggregat-Transformation und) fünf Simple-Transformationen:

```
ARTIST.AID -> artist.aid
ARTIST.BIRTH_PLACE -> artist.birth_place
ARTIST.BIRTH_CNTRY -> artist.birth_cntry
ARTIST.DEATH_DATE -> artist.death_date
ARTIST.REALNAME -> artist.realname
```

- Keine Where-Klausel => Normalform-Typ = „“.

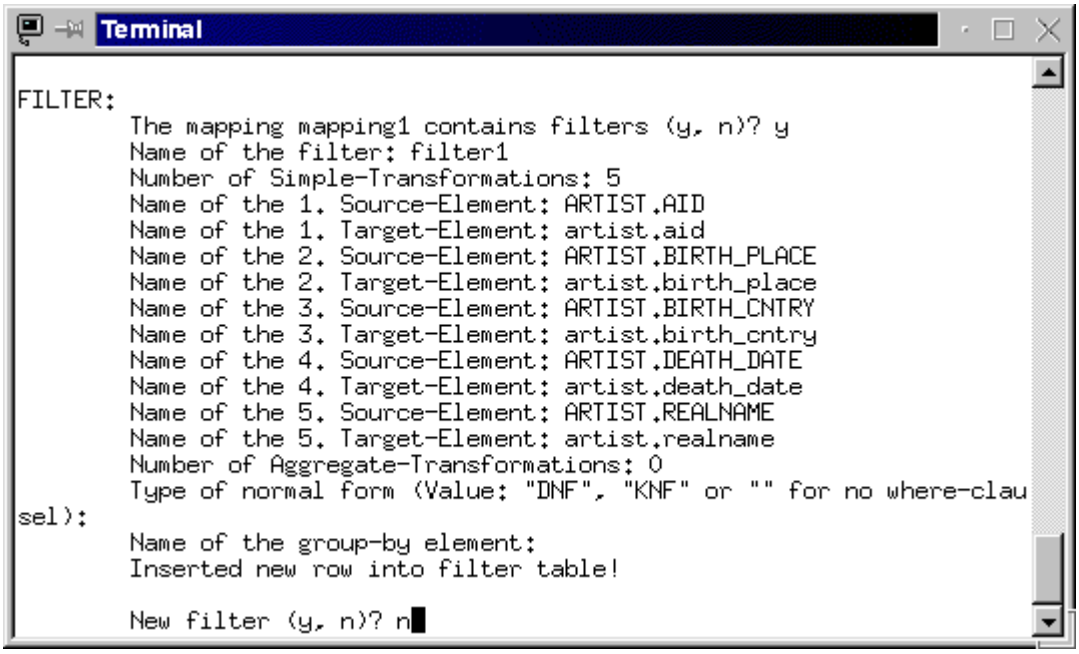


Abbildung 4-9: Filter-Definition (Mapping 1)

Ausfüllen der Tabelle *filter_clausel*

Wenn der Normalform-Typ eines Filters „DNF“ oder „KNF“ ist, wird die Funktion *insertFilterClausel* zum Abspeichern der Filter-Bedingungen in der Tabelle *filter_clausel* aufgerufen. In dieser Tabelle werden Filter-Klauseln abgelegt, die durch ihre eigenen Namen und den Namen des Filters, zu dem sie gehören, voneinander unterschieden werden. Weil der Filter-Name schon eindeutig ist, müssen nur die Filter-Klauseln eines einzelnen Filters unterschiedliche Namen haben. Bei einem gegebenen Filter können infolgedessen die Namen der zugehörigen Filter-Klauseln automatisch durch die verwendete Funktion (*insertFilterClausel*) gebildet werden, z.B.: fc1, fc2, fc3, etc.

Angaben von Filter-Klauseln:

- Anzahl der Filter-Klauseln,
- Anzahl der Literale jeder Klausel,
- Linke, rechte Seite und der Vergleichs-Operator jedes einzelnen Literals.

Für jede Filter-Klausel wird eine Anweisung konstruiert, die dann direkt (mittels *SQLExecDirect*) ausgeführt wird. Die Anzahl der Filter-Klauseln eines Filters bestimmt deshalb, wie viele Anweisungen bezüglich des Einfügens dieser Filter-Klauseln konstruiert und ausgeführt werden.

Abbildung 4-10: Filter-Definition (Mapping 3)

Zur Demonstration der Definition eines Filters mit Filter-Klauseln wurde das **Mapping 3** aus Anhang C genommen (siehe Abbildung 4-10), das durch folgenden SQL-Ausdruck beschrieben wird:

```
SELECT   A1.NAME AS director_name, A2.NAME AS actor_name, MOVIE.TITLE
INTO     worked_together
FROM     ARTIST A1, ARTIST A2, MOVIE, PERFORMED P1, PERFORMED P2
WHERE          P1.MID = MOVIE.MID
      AND      P2.MID = P1.MID
      AND      P1.VROLE = "D"
```

```
AND      P2.VROLE like "A: %"
AND      P1.AID = A1.AID
AND      P2.AID = A2.AID
```

Mehrere Mappings eines Folders können nacheinander definiert werden. Der obige Vorgang des Mapping-Erzeugens erfolgt in änlicher Weise für die übrigen Mappings aus Anhang C. Im folgenden wird nur eine demonstrierte Lösung dieser Mappings kurz geschildert.

Mapping 2: *Source.award, Source.titawd -> Target.general_award*

Dieses Mapping kann durch zwei SQL-Anweisungen beschrieben werden., die alle Datensätze aus jeweils der beiden Quelltabellen *Source.award* und *Source.titawd* in die Zieltabelle *Target.general_award* übertragen. Das **Mapping 2** besteht also aus zwei folgenden Filtern:

filter2a: *Source.award -> Target.general_award*

```
SELECT      AWARD.MID as general_award.mid, AWARD.AID as
            general_award.aid, AWARD.VROLE as general_award.vrole,
            AWARD.ATYPE as general_award.atype
INTO        general_award
FROM        AWARD
```

filter2b: *Source.titawd -> Target.general_award*

```
SELECT      TITAWD.MID as general_award.mid, TITAWD.YEAR as
            general_award.year, TITAWD.ATYPE as general_award.atype
INTO        general_award
FROM        TITAWD
```

Mapping 4: *Source.movie, Source.movbem -> Target.movie_orig_rem, Target.movie_aka*

Zu a) *Source.movie -> Target.movie_aka*

Die Zieltabelle *Target.movie_aka*, die Alternativtitel eines Films enthält, kann durch einen Filter gebildet werden.

filter4a:

```
SELECT      MOVIE.MID as movie_aka.mid, MOVIE.TITLE as movie_aka.
            title, MOVIE.YEAR as movie_aka.year, MOVIE.MV_TYPE as
            movie_aka.mv_type, MOVIE.ORIGINAL as movie_aka.original
INTO        movie_aka
```

```
FROM        MOVIE
WHERE       MOVIE.ORIGINAL is NOT NULL
```

Zu b) *Source.movie, Source.movbem -> Target.movie_orig_rem*

Die alle Originaltitel enthaltende Zieltabelle *Target.movie_orig_rem* kann durch zwei folgende Filter gebildet werden.

filter4b: *Target.movie_orig_rem* enthält alle Originaltitel mit einem Bemerkungstext.

```
SELECT      MOVIE.MID as movie_orig_rem.mid, MOVIE.TITLE as
      movie_orig_rem.title, MOVIE.YEAR as movie_orig_rem.year,
      MOVIE.MV_TYPE as movie_orig_rem.mv_type, MOVBEM.TEXT as
      movie_orig_rem.remark
INTO        movie_orig_rem
FROM        MOVIE, MOVBEM
WHERE       MOVIE.ORIGINAL is NULL
      AND   MOVIE.MID = MOVBEM.MID
      AND   MOVBEM.BEMTYPE = ,Q‘
      AND   MOVBEM.BEM = 1
```

filter4c: *Target.movie_orig_rem* enthält alle Originaltitel ohne Bemerkungstext.

```
SELECT      MOVIE.MID as movie_orig_rem.mid, MOVIE.TITLE as
      movie_orig_rem.title, MOVIE.YEAR as movie_orig_rem.year,
      MOVIE.MV_TYPE as movie_orig_rem.mv_type
INTO        movie_orig_rem
FROM        MOVIE, MOVBEM
WHERE       ( MOVIE.ORIGINAL is NULL AND MOVIE.MID != MOVBEM.MID )
      OR    ( MOVIE.ORIGINAL is NULL AND MOVIE.MID = MOVBEM.MID
              AND MOVBEM.BEMTYPE != ,Q‘ )
      OR    ( MOVIE.ORIGINAL is NULL AND MOVIE.MID = MOVBEM.MID
              AND MOVBEM.BEM != 1 )
```

Anhand der obigen Mapping-Beispiele wurden verschiedene Möglichkeiten des Mapping-Erzeugens behandelt. Manche Mappings bestehen nur aus Transformationen oder Filtern. Andere Mappings können sowohl Transformationen als auch Filter enthalten. Etwas komplizierter ist es, wenn die Abbildung zwischen Quelle und Ziel in zwei Schritten erfolgt: zuerst müssen z.B. die Attribute der Quelltabelle A transformiert, anschließend wieder in die Zieltabelle B selektiert werden. Diese Abbildung kann durch zwei einzelne Mappings realisiert werden. Das 1. Mapping übernimmt die Transformation(en) zwischen der Quelltabelle A und der Zieltabelle A′. Beim 2. Mapping wird die Zieltabelle B durch die Filterung der Datensätze von A′ gebildet, die diesmal die Rolle einer Quelltabelle spielt. Umgekehrt ist es ähnlich, wenn es zuersrt gefiltert und dann transformiert wird.

Tritt beim Maping-Erzeugen ein Fehler auf, z.B. Verletzung der Eindeutigkeit, Fehlen der referenzierten Tabelle oder erfolglose Ausführung eines beliebigen ODBC-Funktionsaufrufs etc., besteht es die Möglichkeit, die Insert-Funktionen nochmals durchzuführen. Wird der nochmalige Versuch verneint, wird zum Ende des Programms gesprungen und die Transaktion mit ROLLBACK beendet (Abbildung 4-11). Alle Anweisungen innerhalb der Verbindung werden automatisch rückgängig gemacht.

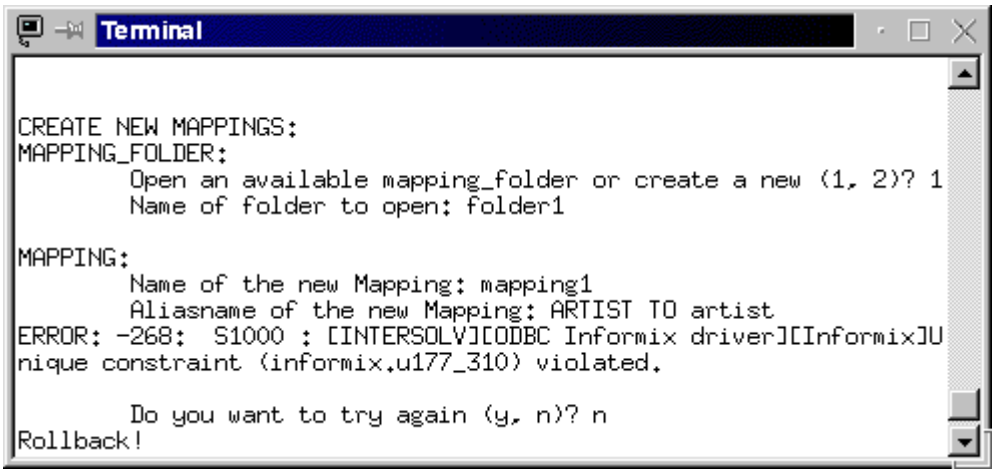


Abbildung 4-11: Rollback-Transaktion

4.4.5 Anzeigen vorhandener Mappings

Neben dem Mapping-Erzeugen stellt das Repository eine weitere Funktion zum Anzeigen von vorhandenen Mappings bereit. Relevante Informationen über ein gegebenes Mapping sind u.a.: Name des Mappings und des zugehörigen Folders, am Mapping teilnehmende Quell- bzw. Ziel-Records, Quell- bzw. Ziel-Elemente, zugehörige Transformationen und/oder Filter. Weil ODBC vorzugsweise lediglich den Zugriff auf relationale Datenbanken ermöglicht, die Repository-Datenbank aber mittels einer objektrelationalen Definitionssprache angelegt wurde, entstehen beim Recherchieren der Datenbank manche Beschränkungen der ODBC-Programmierung. Einige objektorientierte Eigenschaften wie z.B. Kollektions-Typen und Tupel-Typen werden nicht von ODBC unterstützt.

Detaillierte Informationen über Transformationen und Filter (daran teilnehmende Elemente, Transformations- bzw. Filter-Ausdrücke) sind in den Tabellen *transformation*, *expression*, *filter* und *filter_clausel* abgelegt und werden größtenteils als Attribute von Kollektions-Typen und Tupel-Typen definiert. Demzufolge können sie durch ODBC nicht ermittelt werden. Abbildung 4-12 zeigt alle Datensätze aus der Tabelle *transformation* der Reposito-

ry-Datenbank, die durch eine interaktive Select-Anweisung geholt wurden. Da das Attribut *source_elem* dieser Tabelle ein Kollektions-Typ ist, kann Information über die Quell-Elemente einer Transformation durch die Repository-Engine nicht angezeigt werden.

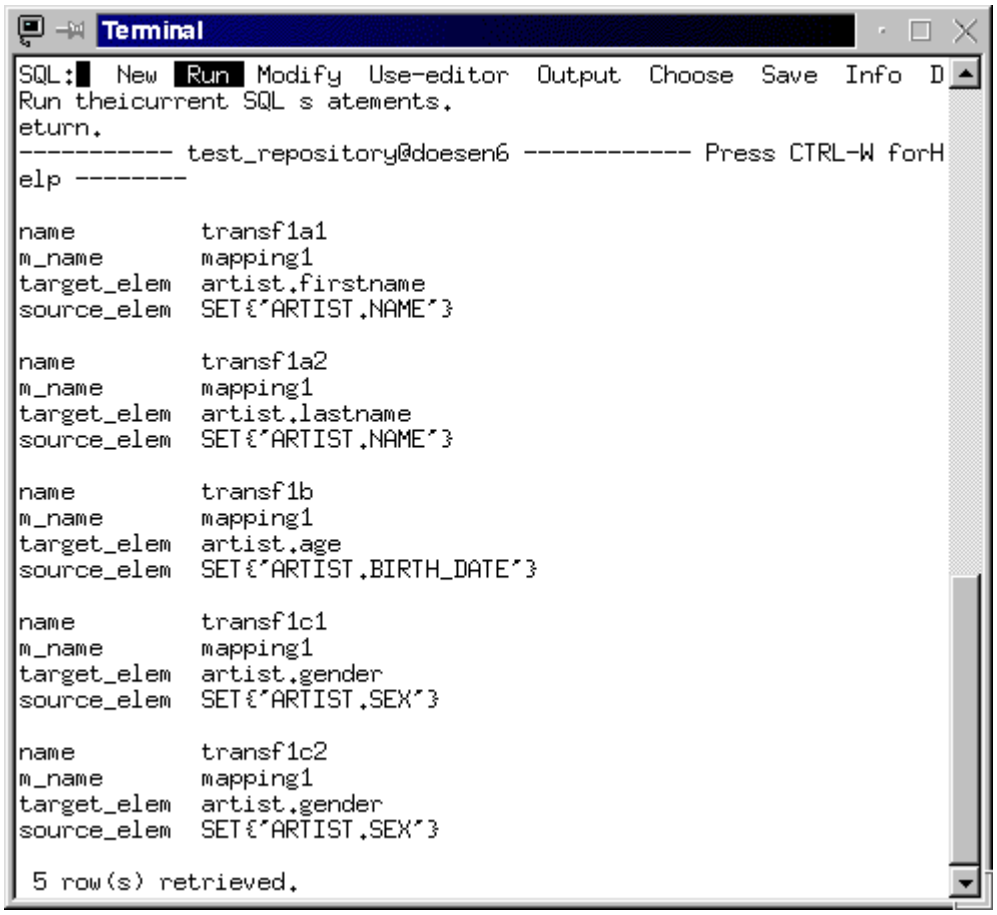


Abbildung 4-12: Interaktiver Zugriff auf Repository-Datenbank

Informationen über jeweilige Folder, Mappings, Quell- und Ziel-Records sowie Namen der Transformationen/Filter können problemlos über ODBC-Funktionsaufrufe aus den Tabellen *mapping_folder*, *mapping*, *record_mapping*, *transformation* und *filter* entnommen werden. Dazu werden Select-Anweisungen verwendet, um spezielle Datensätze aus obigen Tabellen auszuwählen. Bevor die Anweisung ausgeführt wird, müsssen die Spalten der Ergebnismenge mittels *SQLBindCol* gebunden werden, um danach mittels *SQLFetch* geholt zu werden.

Davon ergeben sich drei Ergebnismengen (Abbildung 4-13):

- Vorhandene Mappings und zugehörige Folder,
- Quell- bzw. Ziel-Records und
- Namen der Transformationen bzw. Filter des jeweiligen Mappings.

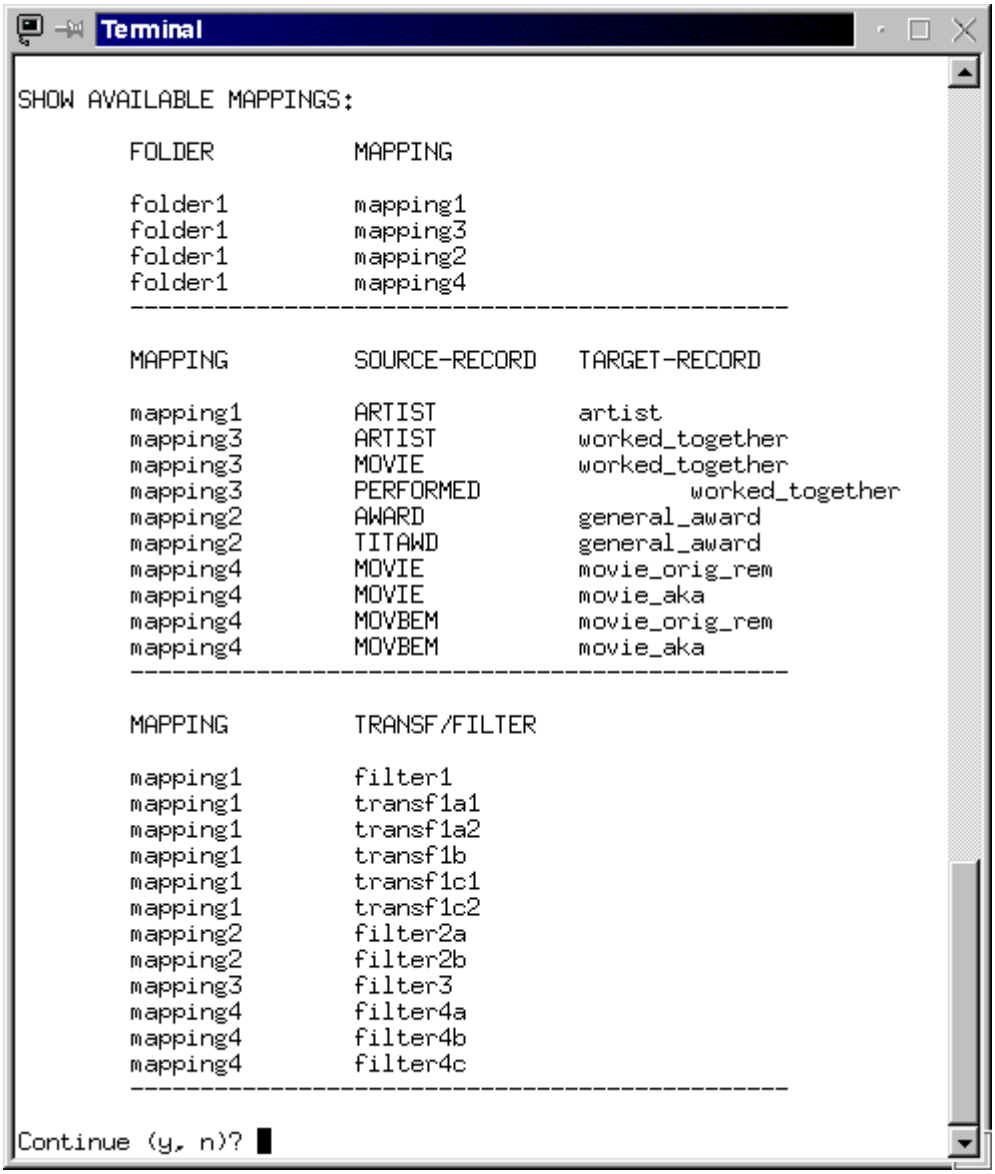


Abbildung 4-13: Mapping-Anzeigen

4.4.6 Löschen vorhandener Mappings

Werden manche Mappings nicht mehr gebraucht, können sie aus dem Repository entfernt werden. Oder muß ein Mapping verändert werden, kann es zunächst gelöscht und danach neu definiert werden. Beim Mapping-Löschen gibt es zwei Möglichkeiten: erstens kann ein gesamter Mapping-Folder, zweitens ein bestimmtes Mapping eines Folders gelöscht werden. Zuerst wird die erste Möglichkeit geboten und man kann den Namen des zu löschenden Folders eingeben (Abbildung 4-14). Gibt es keinen Folder (mehr) zu löschen, braucht man nur die Eingabetaste zu drücken. (Leer-String bedeutet, daß kein Folder zu löschen ist.)

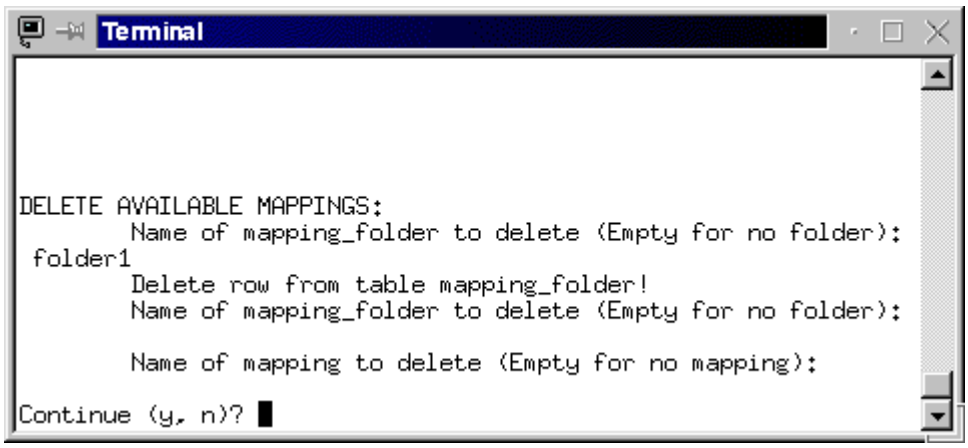


Abbildung 4-14: Mapping-Löschen

Anschließend können einzelne Mappings nacheinander gelöscht werden. Unter der Annahme von der Eindeutigkeit der Mapping-Namen wird nur der Name des Mappings, und nicht zusätzlich des zugehörigen Folders verlangt. Ähnlich wie bei den Foldern können beliebig viele Mappings entfernt werden, bis man Leer-String eingibt.

5 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Repository-Ansatz für die Verwaltung von technischen Metadaten in Data Warehouse-Umgebungen entwickelt. Der Ansatz basiert auf einem Metamodell, das entsprechend den Charakteristika von technischen Metadaten mittels UML entworfen wurde. Dieses UML-Metamodell enthält mehrere Klassen und ihre Beziehungen zueinander, durch die sich die Struktur und der Inhalt verschiedener Datenhaltungssysteme darstellen lassen. Darüber hinaus können dadurch auch die Abhängigkeiten und Abbildungen zwischen diesen Systemen beschrieben werden.

Das Repository entstand durch die Implementierung des entworfenen Metamodells auf dem DBMS des Informix Universal Server Version 9.1 (IUS). Als ein objektrelationales DBMS kann IUS die Vorteile der relationalen Datenbanken (wie z.B. Robustheit, Stabilität) mit den Vorzügen der Objektorientierung kombinieren. Folgende objektorientierte Eigenschaften von IUS wurden bei der Implementierung der Repository-Datenbank genutzt: komplexe Datentypen (Kollektions-Typ, Tupel-Typ) und Vererbung.

Das Repository ist nunmehr in der Lage, alle relevanten, im Metamodell beschriebenen technischen Metadaten abzuspeichern. Für die Nutzung des Repository wurden folgende Funktionen der Repository-Engine implementiert:

- Importieren neuer Quell-/Ziel-Metadaten.
  Metadaten bezüglich Struktur und Inhalt verschiedener Datenhaltungssysteme werden in die entsprechenden Tabellen eingefügt. Der größte Teil dieser Metadaten kann mittels ODBC-Katalogfunktionen direkt aus den Data Dictionaries verschiedener DBMS importiert und dann im Repository gespeichert werden.

- Anzeigen von Quell-/Ziel-Metadaten.
  Durch diese Funktion werden den Benutzern Informationen über Quell- bzw. Zielsysteme bereitgestellt, z.B. über verfügbare Datenbanken und zugehörige Tabellen bzw. Attribute.

- Löschen von Quell-/Ziel-Metadaten.
  Diese Funktion ermöglicht den Benutzern, nicht mehr gebrauchte Quell-/Ziel-Metadaten aus dem Repository zu entfernen.

- Erzeugen/Definieren von Mappings.
  Eine Hauptfunktion der Repository-Engine ist das Erzeugen (in dieser Arbeit das Definieren) von neuen Mappings. Mappings werden nicht nur auf der Tabelle-/Rekord-Ebene definiert. Transfomations- und Filterungsregeln der einzelnen Mappings können auf der Spalte-/Attribut-Ebene dargestellt und in implementierbarer Weise gespeichert werden.

- Anzeigen von Mappings.
  Zur Verfügung stehen den Benutzern allgemeine Informationen über die definierten Mappings, wie z.B. Name des Mapping und der zugehörigen Transformationen und/oder Filter, am Mapping teilnehmende Tabellen/Rekords. Da die Repository-Datenbank viele objektrelationale Eigenschaften besitzt, diese aber durch die ODBC-Programmierung nicht unterstützt werden, können im Repository gespeicherte, detaillierte Abbildungsregeln nicht angezeigt werden.

- Löschen von Mappings.
  Einzelne Mappings oder gesamte Mapping-Folder können gelöscht werden.

Das Hauptergebnis dieser Arbeit liegt darin, daß ein Metamodell konzipiert wurde, das relevante Objekte bezüglich Funktionalitäten von (vor allem technischen) Metadaten in Data Warehouse-Umgebungen beschreiben kann. Durch dieses Metamodell ist eine detaillierte Beschreibung von Abbildungen zwischen Quell- und Zielsystemen möglich. Die Implementierbarkeit des entworfenen Metamodells wurde anhand wichtiger Repository-Funktionen getestet.

Darüber hinaus kann das UML-Metamodell erweitert werden, nicht nur durch die Festlegung von Attributen, sondern wichtiger dadurch, daß neue Klassen und Beziehungen in das Modell hinzugefügt werden können. Semantische Metadaten, die im Rahmen dieser Arbeit nicht behandelt werden, können grundsätzlich durch Erweiterung dieses Metamodells dargestellt werden. Damit können eine einheitliche Repräsentation und insbesondere die Integration von technischen und semantischen Metadaten als Grundlagen für ein leistungsfähiges Metadaten-Management in Data Warehouse-Umgebungen ermöglicht werden.

6 Literaturverzeichnis

[AM97] Anahory, S.; Murray, D.*: Data Warehouse – Planung, Implementierung und Administration*. Addsion-Wesley, 1997.

[BB+99] Bernstein, P.A.; Bergstraesser, T.; Carlson, J.; Pal, S:; Sanders, P.; Shutt, D.: *Microsoft Repository Version 2 and the Open Information Model*. In Information System 24(2), 1999.

[Ber97] Bernstein, P.A.: *Repositories and object oriented databases*. In Proceedings of BTW97: 34-46.

[BHS97] Bernstein, P.A.; Harry, B.; Sanders, P.: *The Microsoft Repository*. In Proceedings of 23. International Conference on Very Large Data Bases, 1997. http://www.mkp.com.

[Bra96] Brackett, M. H.: The Data Warehouse Challenge. Wiley, 1996.

[BRJ99] Booch, G.; Rumbaugh, J.; Jacobson, I.: *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Bur97] Burkhardt, R.: *UML – Unified Modeling Language: Objektorientierte Modellierung für die Praxis*. Addison-Wesley-Longman, Bonn, 1997.

[CD97] Chaudhuri, S.; Dayal, U.: *An overview of Data Warehousing and OLAP-Technology*. In ACM SIGMOD Record 26 (1), 3/1997.

[Clu93] Clure, Mc; Carma, L.: *Software-Automatisierung*. München, Hanser 1993: pp. 161-223.

[Dus98] Dusan, P.: *Informix Universal Server: das objekt-relationale Datenbanksystem, mit Online-XPS und ODS*. Addison-Wesley, 1998.

[ETI96] ETI: *Handbook for the Metadata Exchange Library*. 5/1996.

[FS98] Fowler, M.; Scott, K.: *UML konzentriert: die neue Standardobjekt-modellierungssprache anwenden*. Addison-Wesley-Longman, Bonn, 1998.

[Gei95] Geiger, K.: *Inside ODBC*. Microsoft Press, 1995.

[Inf98] Informatica: *Repository Metadata Exchange API*. 9/1998.

[Inm93] Inmon, W. H.: *Building the data warehouse*. John Wiley & Sons, 1993.

[Inm98] Inmon, B.: *Enterprise meta data*. In DM Review, 11/1998. http://www.dmreview.com/issues/1998/nov/articles/nov98_.htm.

[IUS97] Informix Universal Server: *Informix Guide to SQL: Tutorial/ Syntax/ Reference Version 9.1*. 3/1997.

[JJQ99] Jarke, M.; Jeusfeld, M.A.; Quix, C.; Vassiliadis, P.: *Architecture and Quality in Data Warehouses: an Extended Repository Approach*. In Information Systems, 24(3), 1999.

[JV97] Jarke, M.; Vassiliou, Y.: *Data warehouse quality design: A review of the DWQ project*. In Proc. 2nd Conference on Information Quality, Massachusetts Institute of Technology. Cambrige, MA, 1997.

[Kah98] Kahlbrandt, B.: *Software Engineering. Objektorientierte Software-Entwicklung mit der Unified Modeling Language*. Springer, Berlin, Heidelberg, New York, 1998.

[KE97] Kemper, A.; Eickler, A.: *Datenbanksysteme*. Oldenbourg, 1997.

[Kir98] Kirchner, J.: *Transformationsprogramme und Extraktionsprozesse entscheidungsrelevanter Basisdaten*. In Das Data Warehouse-Konzept, Gabler, Wiesbaden, 1998, pp. 245-273.

[MB98] Mucksch, H.; Behme, W. (Hrsg.): *Das Data Warehouse Konzept*. Gabler, Wiesbaden, 1998.

[MD96] *First IEEE Metadata Conference*, Proceedings. Silver Spring, Maryland, 4/1996. http://www.computer.org/conferen/meta96/paper_list.html.

[MD97] *Second IEEE Metadata Conference*. Silver Spring, Maryland, 9/1997. http://computer.org/conferen/proceed/meta97/list_papers.html.

[MDC97] The Metadata Coalition: *Meta Data Interchange Specification (MDIS Version 1.1)*. 8/1997.
http://www.mdcinfo.com/MDIS/MDIS11.html.

[MDC99] The Metadata Coalition: *Open Information Model (OIM Version 1.0).* 4/1999.
http://www.mdcinfo.com/OIM/OIM10.html.

[MiXIF] Microsoft: XML Interchange Format (XIF).
http://msdn.microsoft.com/repository/.

[MSR99] Müller, R.; Stöhr, T.; Rahm, E.: *An integrative and uniform model for metadata management in data warehousing environments.* In Proceedings Workshop on Design and Management of DWs (DMD'99). Heidelberg, Juni 1999: 12/1-12/16.
http://dol.uni-leipzig.de/pub/1999-22.

[Neu98] Neumann, H.A.: *Objektorientierte Entwicklung mit der Unified Modeling Language (UML).* Hanser, München, 1998.

[ODBC] *ODBC 3.0 Programmer's Reference.*

[Oes98] Oestereich, B.: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language*. Oldenburg, München, 1998.

[OMG97] Object Management Group: *Meta Object Facility (MOF) Specification*. OMG Document ad/97-08-14 and ad/97-08-15.

[OMG99] Object Management Group: *Common Warehouse Model (CWM) Specification.* Initial Submission to OMG RFP: Common Warehouse Metadata Interchange (CWMI), 9/1999.
http://www.omg.org/techprocess/meetings/schedule/CWMI_RFP.html.

[O98/9] Object Management Group*: Common Warehouse Metadata Interchange – Request For Proposal*. OMG Document ad/98-09-02.

[O98/10] Object Management Group: *XML Metadata Interchange (XMI)*. OMG Document ad/98-10-05.

[Rah94] Rahm, E.: *Mehrrechner-Datenbanksysteme*. Addison-Wesley, Bonn, 1994.

[Rah99] Rahm, E.: *Datenbanksysteme 1/ Datenbanksysteme 2. Vorlesungsskript.* http://www.informatik.uni-leipzig.de/ifi/abteilungen/db/datenbanken.html

[SVV99] Staudt, M.; Vaduva, A.; Vetterli, T.*: Metadata management and data warehousing*. Technical Report 21, Swsslife, Information System Research, 7/1999. http://www.ifi.unizh_ch/techreports/TR_1999.html.

[Vos94] Vossen, G.: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*. Addison-Wesley, Bonn, 1994.

[WB97] Wu, M.C.; Buchman, A.P.: *Research Issues in Data Warehousing*. In Proceedings of BTW'97, 1997 .

[Wer95] Werner, D.: *Taschenbuch der Informatik*. Fachbuchverlag Leipzig, 1995, pp.440-490.

[Whi99] White, C.: *Managing distributed data warehouse meta data*. In DM Review, 2/1999. http://www.dmreview.com/issues/1999/feb/articles/feb99_46.htm.

[Wie98] Wieken, J.-H.: *Meta-Daten für Data Marts und Data Warehouses*. In Das Data Warehouse-Konzept, Gabler, Wiesbaden, 1998, pp. 275-315.

[Wie99] Wieken, J.-H.: *Der Weg zum Data Warehouse*. Addison Wesley, 1999.

7 Anhang

7.1 Anhang A: Schema der Quell-Datenbank (MOVIEDB2)

Tabelle: movie

Attribut	Wertebereich	Null-Wert	Bemerkung
mid	INTEGER	NOT NULL	Primärschlüssel
title	VARCHAR(80)	NOT NULL	
year	INTEGER		Value: year > 1880
mv_type	VARCHAR(6)		
original	INTEGER		

Tabelle: artist

Attribut	Wertebereich	Null-Wert	Bemerkung
aid	INTEGER	NOT NULL	Primärschlüssel
name	VARCHAR(30)	NOT NULL	
sex	CHAR(1)	NOT NULL	Value: ‚f', ‚m', ‚?'
birth_date	DATETIME		
birth_place	VARCHAR(40)		
birth_cntry	VARCHAR(20)		
death_date	DATETIME		
realname	VARCHAR(40)		

Tabelle: performed

Attribut	Wertebereich	Null-Wert	Bemerkung
mid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
aid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
vrole	VARCHAR(85)	NOT NULL	Primärschlüssel; Value:substring (vrole,1,1) in(‚A', ‚C', ‚D', ‚M', ‚W')

Tabelle: award

Attribut	Wertebereich	Null-Wert	Bemerkung
mid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
aid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
vrole	CHAR(1)	NOT NULL	Primärschlüssel
atype	VARCHAR(6)	NOT NULL	Primärschlüssel

Tabelle: titawd

Attribut	Wertebereich	Null-Wert	Bemerkung
mid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
year	SMALLINTEGER		
atype	VARCHAR(4)	NOT NULL	Primärschlüssel

Tabelle: movbem

Attribut	Wertebereich	Null-Wert	Bemerkung
mid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
bemtype	CHAR(1)	NOT NULL	Primärschlüssel; Value: ‚C‘, ‚G‘, ‚Q‘, ‚I‘
text	VARCHAR(85)	NOT NULL	Primärschlüssel
bem	SMALLINTEGER	NOT NULL	Primärschlüssel; Value: bem > 0

Tabelle: artbem

Attribut	Wertebereich	Null-Wert	Bemerkung
aid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
bemtype	VARCHAR(2)	NOT NULL	Primärschlüssel
text	VARCHAR(76)	NOT NULL	Primärschlüssel
bem	SMALLINTEGER	NOT NULL	Primärschlüssel; Value: bem > 0

7.2 Anhang B: Schema der Ziel-Datenbank (ardnt_movie_target)

Tabelle: movie_orig_rem

Attribut	Wertebereich	Null-Wert	Bemerkung
mid	INTEGER	NOT NULL	Primärschlüssel
title	VARCHAR(80)	NOT NULL	
year	INTEGER		
mv_type	VARCHAR(6)		
remark	VARCHAR(85)		

Tabelle: artist

Attribut	Wertebereich	Null-Wert	Bemerkung
aid	INTEGER	NOT NULL	Primärschlüssel
lastname	VARCHAR(30)	NOT NULL	
firstname	VARCHAR(20)		
gender	INTEGER	NOT NULL	Value: 0, 1
age	INTEGER		
birth_place	VARCHAR(40)		
birth_cntry	VARCHAR(20)		
death_date	DATETIME		
realname	VARCHAR(40)		

Tabelle: movie_aka

Attribut	Wertebereich	Null-Wert	Bemerkung
mid	INTEGER	NOT NULL	Primärschlüssel
title	VARCHAR(80)	NOT NULL	
year	INTEGER		
mv_type	VARCHAR(6)		
original	INTEGER	NOT NULL	Fremdschlüssel

Tabelle: artbem

Attribut	Wertebereich	Null-Wert	Bemerkung
aid	INTEGER	NOT NULL	Primär-/ Fremdschlüssel
bemtype	VARCHAR(2)	NOT NULL	Primärschlüssel
text	VARCHAR(76)	NOT NULL	Primärschlüssel
bem	SMALLINTEGER	NOT NULL	Primärschlüssel; Value: bem > 0

Tabelle: general_award

Attribut	Wertebereich	Null-Wert	Bemerkung
gaid	SERIAL	NOT NULL	Primärschlüssel
mid	INTEGER	NOT NULL	Fremdschlüssel
aid	INTEGER		Fremdschlüssel
vrole	CHAR(1)		
atype	VARCHAR(8)	NOT NULL	
year	SMALLINTEGER		

Tabelle: worked_together

Attribut	Wertebereich	Null-Wert	Bemerkung
wtid	SERIAL	NOT NULL	Primärschlüssel
director_name	VARCHAR(30)	NOT NULL	
actor_name	VARCHAR(30)	NOT NULL	
title	VARCHAR(80)	NOT NULL	

7.3 Anhang C: Test-Mappings

1. Ein Quellobjekt -> Ein Zielobjekt

Mapping 1: *Tabelle Source.artist -> Tabelle Target.artist*

a) Attributsplit

Source.artist.name -> Target.artist.lastname, Target.artist.firstname

Source.artist.name = "lastname, firstname" wird aufgeteilt in Target.artist.lastname = "lastname" und Target.artist.firstname = "firstname".

b) Konversion incl. Berechnung

Source.artist.birth_date -> Target.artist.age (wenn ‘artist’ noch lebt),

Target.artist.age = NULL (wenn ‘artist’ schon gestorben ist),

c) Typkonversion

Source.artist.sex -> Target.artist.gender

Umwandlung des Geschlechts-Attributes:

'm' -> 0 'f' -> 1 '?' -> Tupel ablehnen

d) 1:1 – Abbildung der übrigen Spalten der Tabelle Source.artist nach Target.artist ohne Transformation.

Mapping 1a: *Source.artbem -> Target.artbem* (1:1 – Abbildung)

2. Zwei Quellobjekte -> Ein Zielobjekt

Mapping 2: *Source.award, Source.titawd -> Target.general_award*

Die Zieltabelle enthält alle Attribute der beiden Quelltabellen zuzüglich eines neuen Primärschlüsselattributes.

Jedes Tupel einer Quelltabelle erhält genau ein Tupel in der Zieltabelle zugeordnet, das bedeutet die Anzahl der Tupel der Zieltabelle entspricht der Summe der Tupel der Quelltabellen.

Die Attribute der Zieltupel, die kein korrespondierendes Attribut im Quelltupel haben, werden NULL gesetzt. Weitere Transformationen finden nicht statt.

3. Mehrere Quellobjekte => Ein Zielobjekt

Mapping 3: *Source.artist, Source.movie -> Target.worked_together*

Source.artist.name -> Target.worked_together.director_name (vrole = “D”)
Source.artist.name -> Target.worked_together.actor_name (vrole = “A: %”)
Source.movie.title -> Target.worked_together.title

Bedeutung: gibt diejenigen Schauspieler (‘*actor’*) und Regisseure (‘*director’*) zusammen mit den Filmtiteln aus, in welchen diese gemeinsam gearbeitet haben.

Erläuterung für *vrole*: Anfangsbuchstabe "D" bedeutet *director*, "A" bedeutet *actor* gefolgt von einem Doppelpunkt und dem Namen der gespielten Rolle.

4. Mehrere Quellobjekte => Mehrere Zielobjekte

Mapping 4: *Source.movie, Source.movbem -> Target.movie_orig_rem, Target.movie_aka*

Target.movie_orig_rem enthält alle originalen Filmtitel während movie_aka die synchronisierten Alternativtitel (z.B. französisch, deutsch, ...) des gleichen Filmes enthält.

Falls das Attribut 'original' NULL enthält, so bedeutet dies, daß dieser Satz keinen Verweis auf einen Originaltitel enthält und daher selbst einen Originaltitel darstellt. Andernfalls enthält das Attribut einen Verweis auf den Originaldatensatz (und repräsentiert daher einen Alternativtitel).

a) Target.movie_aka

Alle Datensätze aus Source.movie, deren Attribut Source.movie.original ungleich NULL ist, werden in Target.movie_aka eingefügt.

b) Target.movie_orig_rem

Kopiert alle diejenigen Source.movie-Tupel nach Target.movie_orig_rem, die einen Originalfilm repräsentieren (Bedingung: Source.movie.original = NULL).

Im Falle, daß es einen korrespondierenden Datensatz in Source.movbem gibt (d.h. Source.movbem.mid = Source.movie.mid), der die Bedingungen ‚Source.movbem.bemtype='Q' AND Source.movbem.bem=1‘ erfüllt, dann wird Target.movie_orig_rem.remark auf den Inhalt von Source.movbem.text gesetzt, andernfalls NULL.

7.4 Anhang D: Definition der Repository-Datenbank

```
CREATE ROW TYPE schema_t
  (
    name           VARCHAR(20)      NOT NULL,
    ds_name        VARCHAR(20),
    aliasname      VARCHAR(30)
  );

CREATE ROW TYPE filestructure_t
  (
    format         VARCHAR(20),
    descr_format   VARCHAR(20),
    col_length     INTEGER,
    col_separator  CHAR(2)
  )
UNDER schema_t;

CREATE ROW TYPE function_t
  (
    name           VARCHAR(20)      NOT NULL,
    param_list     LIST(VARCHAR(20) NOT NULL)
  );

CREATE ROW TYPE simpletransf_t
  (
    source_elem    VARCHAR(40)      NOT NULL,
    target_elem    VARCHAR(40)      NOT NULL
  );


CREATE ROW TYPE aggrtransf_t
  (
    source_elem    VARCHAR(40)      NOT NULL,
    target_elem    VARCHAR(40)      NOT NULL,
    aggr_operator  VARCHAR(5)       NOT NULL
  );

CREATE ROW TYPE filterliteral_t
  (
    lit_left       VARCHAR(40)      NOT NULL,
    comp_operator  VARCHAR(8)       NOT NULL,
    lit_right      VARCHAR(40)      NOT NULL
        );


CREATE TABLE data_store
  (
    name           VARCHAR(20)      NOT NULL ,
    aliasname      VARCHAR(30),
    PRIMARY KEY (name)
  );
revoke all on data_store from "public";

CREATE TABLE database
  (
    name           VARCHAR(20)      NOT NULL ,
    type           VARCHAR(20)      NOT NULL ,
    system         VARCHAR(20),
    servername     VARCHAR(20),
    portnumber     INTEGER,
    hostname       VARCHAR(30),
    connection     VARCHAR(30),

    PRIMARY KEY (name),
    FOREIGN KEY (name) REFERENCES data_store (name) ON DELETE CASCADE
  );
revoke all on database from "public";
```

```
CREATE TABLE file
  (
    name           VARCHAR(20)      NOT NULL,
    ds_name        VARCHAR(20)      NOT NULL,
    location       VARCHAR(40)      NOT NULL ,

    PRIMARY KEY (name),
    FOREIGN KEY (ds_name) REFERENCES data_store (name) ON DELETE CASCADE
  );
revoke all on file from "public";

CREATE TABLE schema OF TYPE schema_t
  (
    PRIMARY KEY (name),
    FOREIGN KEY (ds_name) REFERENCES data_store (name) ON DELETE CASCADE
  );
revoke all on schema from "public";

CREATE TABLE file_structure OF TYPE filestructure_t UNDER schema;
revoke all on file_structure from "public";

CREATE TABLE record
  (
    name           VARCHAR(20)      NOT NULL ,
    s_name         VARCHAR(20)      NOT NULL,
    aliasname      VARCHAR(30),

    PRIMARY KEY (name, s_name),
    FOREIGN KEY (s_name) REFERENCES schema (name) ON DELETE CASCADE
  );
revoke all on record from "public";

CREATE TABLE element
  (
    name           VARCHAR(20)      NOT NULL ,
    r_name         VARCHAR(20)      NOT NULL ,
    s_name         VARCHAR(20)      NOT NULL ,
    aliasname      VARCHAR(30),
    datatype       VARCHAR(15),
    length         INTEGER,
    scale          INTEGER,
    nullable       INTEGER,

    CHECK (nullable IN (1, 0)),
    PRIMARY KEY (name, r_name, s_name),
    FOREIGN KEY (r_name, s_name) REFERENCES record (name, s_name) ON DE-
LETE CASCADE
  );
revoke all on element from "public";

CREATE TABLE reference
  (
    schema         VARCHAR(20)      NOT NULL,
    pktable        VARCHAR(20)      NOT NULL,
    pkcolumn       VARCHAR(20)      NOT NULL,
    pkseq          INTEGER,
    fktable        VARCHAR(20),
    fkcolumn       VARCHAR(20),

    PRIMARY KEY (schema, pktable, pkcolumn),
    FOREIGN KEY (pkcolumn, pktable, schema) REFERENCES element (name,
r_name, s_name)
    ON DELETE CASCADE
 );
revoke all on reference from "public";


CREATE TABLE mapping_folder
  (
```

```
    name           VARCHAR(20)       NOT NULL ,
    aliasname      VARCHAR(30),
    PRIMARY KEY (name)
  );
revoke all on mapping_folder from "public";

CREATE TABLE mapping
  (
    name           VARCHAR(20)       NOT NULL ,
    f_name         VARCHAR(20)       NOT NULL ,
    aliasname      VARCHAR(30),
    PRIMARY KEY (name),
    FOREIGN KEY (f_name) REFERENCES mapping_folder (name) ON DELETE CAS-
CADE
  );
revoke all on mapping from "public";

CREATE TABLE record_mapping
  (
    r_name         VARCHAR(20)      NOT NULL,
    s_name         VARCHAR(20)      NOT NULL,
    m_name         VARCHAR(20)      NOT NULL,
    role           CHAR(6)          NOT NULL,

    CHECK (role IN ('source', 'target')),
    PRIMARY KEY (r_name, s_name, m_name),
    FOREIGN KEY (r_name, s_name) REFERENCES record (name, s_name) ON DE-
LETE CASCADE,
    FOREIGN KEY (m_name) REFERENCES mapping (name) ON DELETE CASCADE
  );
revoke all on record_mapping from "public";

CREATE TABLE transformation
  (
    name           VARCHAR(20)      NOT NULL,
    m_name         VARCHAR(20)      NOT NULL,
    target_elem    VARCHAR(40)      NOT NULL,
    source_elem    SET(VARCHAR(40)  NOT NULL),
    PRIMARY KEY (name),
    FOREIGN KEY (m_name) REFERENCES mapping (name) ON DELETE CASCADE
  );
revoke all on transformation from "public";

CREATE TABLE expression
  (
    name           VARCHAR(20)      NOT NULL,
    t_name         VARCHAR(20)      NOT NULL,
    left_expr      VARCHAR(20),
    right_expr     VARCHAR(20),
    bin_operator   CHAR(1),
    function       function_t,

    CHECK (bin_operator IN ('+', '-', '*', '/')),
    PRIMARY KEY (name, t_name),
    FOREIGN KEY (t_name) REFERENCES transformation (name) ON DELETE CAS-
CADE
  );
revoke all on expression from "public";

CREATE TABLE filter
  (
    name           VARCHAR(20)      NOT NULL,
    m_name         VARCHAR(20)      NOT NULL ,
    simple_transfs SET(simpletransf_t    NOT NULL),
    aggr_transfs   SET(aggrtransf_t      NOT NULL),
    nf_type  CHAR(10),
    grby_elem      VARCHAR(40),

    CHECK (nf_type IN ('', 'DNF' ,'KNF')),
    PRIMARY KEY (name),
```

```
    FOREIGN KEY (m_name) REFERENCES mapping (name) ON DELETE CASCADE
  );
revoke all on filter from "public";

CREATE TABLE filter_clausel
  (
    name           VARCHAR(20)      NOT NULL,
    f_name         VARCHAR(20)      NOT NULL ,
    f_literals     SET(filterliteral_t   NOT NULL),
    PRIMARY KEY (name, f_name),
    FOREIGN KEY (f_name) REFERENCES filter (name) ON DELETE CASCADE
  );
revoke all on filter_clausel from "public";
```

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, Februar 2000 .....

Unterschrift