Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Dynamische Datenbankorganisation für multimediale Informationssysteme

Diplomarbeit

Vorgelegt von: Torsten Schlieder

Betreuer: Prof. Dr. E. Rahm, Dr. D. Sosna

Leipzig, November 1998

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Wir leben im Informationszeitalter. Systeme, die in der Lage sind, multimediale Daten zu katalogisieren und darzustellen, haben in den letzten Jahren sprunghaft an Bedeutung gewonnen. Einen entscheidenden Anteil an dieser Entwicklung hat das World Wide Web, mit dem eine breite Öffentlichkeit erreicht werden kann. Der Kreis der Anbieter von Informationssystemen, der ursprünglich hauptsächlich dem akademischen Bereich entstammte, erweiterte sich auf Firmen, die ihre Produkte vorstellen wollten, öffentliche Einrichtungen und Privatpersonen.

Die Palette der eingesetzten Lösungen für Informationssysteme ist breit. Sie umfaßt Information-Retrieval-Systeme zur Suche in Texten, multimediale Präsentationen auf der Grundlage statischer Dokumente und datenbankbasierte Systeme, die auf Nutzeranfragen reagieren und alle Dokumente dynamisch erzeugen. Während für einfache Präsentationen der Einsatz von statischen Dokumenten meist ausreichend ist, stößt diese Technik für größere, stark strukturierte Datenbestände an ihre Grenzen. Änderungen des Datenbestandes sind hier nur mit großem Aufwand möglich und bergen stets die Gefahr von Inkonsistenzen in sich.

Datenbank-Management-Systeme (DBMS) sind hingegen in der Lage, große Datenmengen effizient zu verwalten und die Integrität der Daten zu garantieren, so daß sie sich als Grundlage für die dynamische Generierung von Dokumenten anbieten. Allerdings bringt der Einsatz von DBMS wiederum einige Probleme mit sich. So unterstützen die meisten DBMS keine indexbasierte Suche in allen Texten des Datenbestandes. Weiterhin ist es oft nur mit hohem Aufwand möglich, die meist stark strukturierten Katalogdaten auf ein relationales Datenbankschema abzubilden [STS97, S. 42ff]. Objektorientierte DBMS (OODBMS) sind dagegen in der Lage, heterogene Datenstrukturen adäquat zu modellieren. Durch die angestrebte enge Bindung zwischen OODBMS und objektorientierten Programmiersprachen ergibt sich jedoch der Nachteil, daß ein Datenbankschema normalerweise zur Übersetzungszeit der Datenbank-Applikationen feststehen muß. Da

in Informationssystemen nicht nur die Daten häufigen Änderungen unterliegen, sondern oft auch die Struktur der Daten angepaßt werden muß, benötigen diese Systeme ein anpassungsfähiges, dynamisch änderbares Datenbankschema, das weitgehend unabhängig von den aufgesetzten Applikationen ist.

Die vorliegende Arbeit stellt ein Datenbankschema vor, das als Basis für viele Informationssysteme und Präsentationen dienen kann. Es gestattet die dynamische Erzeugung sich selbst beschreibender Datenschemata[1], unterstützt die flexible Modellierung wichtiger objektorientierter Konzepte, speichert Anwendungsdaten und Strukturinformationen in mehreren Sprachen und verwaltet multimediale Daten auf generische Weise. Das Datenbankschema ist in eine Architektur eingebettet, die aus einem Schemalader, einem Datenlader, einer WWW-Anbindung und einem Information-Retrieval-System besteht.

Gliederung der Arbeit

Im folgenden Kapitel wird zunächst ein Katalog von Anforderungen herausgearbeitet, die an ein datenbankbasiertes, multimediales Informationssystem gestellt werden müssen.

Kapitel 3 untersucht, welche Komponenten für die Architektur des Informationssystems benötigt werden, welche Abhängigkeiten zwischen den Teilsystemen bestehen und auf welche Weise die Kommunikation im System erfolgen soll.

Kernpunkt der Arbeit ist die Entwicklung des Datenbankschemas. In Kapitel 4 wird zu Beginn analysiert, welche Nachteile entstehen, wenn die Datenbank und die angekoppelte Programmiersprache kein flexibles und selbstbeschreibendes Datenmodell besitzen. Ausgehend von diesen Nachteilen wird im weiteren Verlauf des Kapitels ein Datenbankschema entwickelt, auf dessen Grundlage dynamisch änderbare, reflexive Datenschemata erstellt werden können.

Das Datenbankschema bildet nur den Rahmen für die Modellierung vieler Anwendungsbereiche. In Kapitel 5 wird eine formale Sprache entworfen, mit der ein weiter Bereich von Modellwelten beschrieben werden kann. Ein Schemalader erstellt aus der Beschreibung ein Datenschema. Ebenfalls in diesem Kapitel wird ein Lader vorgestellt, der die zu einem Datenschema gehörenden Anwendungsdaten in die Datenbank importiert.

[1] In der vorliegenden Arbeit wird zwischen den Begriffen “Datenbankschema” und “Datenschema” unterschieden. Das Datenbankschema wird mit Hilfe der *Data Definition Language* des DBMS erstellt; ein Datenschema ist eine dynamisch generierte Beschreibung der Anwendungsdaten auf der Grundlage des Datenbankschemas (vgl. Kapitel 4 und das Glossar in Anhang B).

Die Anbindung des Datenbankschemas an das World Wide Web ist Gegenstand des 6. Kapitels. Ausgehend von einem allgemeinen Modell der Darstellung von Anwendungsdaten wird ein Klassenschema entwickelt, das eine zunächst noch implementierungsunabhängige Anbindung des Datenbankschemas an das World Wide Web beschreibt. Eine Realisierung mit dem Programmpaket O2Web wird im Anschluß daran vorgestellt.

In Kapitel 7 wird diskutiert, warum es sinnvoll ist, ein Information-Retrieval innerhalb eines Informationssystems einzusetzen. Am Beispiel einer angepaßten Version der Klassenbibliothek lsearch wird gezeigt, wie ein konkretes Information-Retrieval-System an das Datenbankschema angekoppelt werden kann.

Kapitel 8 faßt die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Verbesserungen und Erweiterungen.

Der praktische Teil der Arbeit befaßt sich damit, die Eignung der Architektur für einen konkreten Anwendungsfall zu zeigen. In Zusammenarbeit mit dem Musikinstrumentenmuseum der Universität Leipzig wurde eine Präsentation des Museums im World Wide Web auf der Grundlage wissenschaftlich katalogisierter Daten entwickelt. Diese Anwendung ist unter der folgenden Adresse zu erreichen:

`http://www.uni-leipzig.de/museum/musik`

Danksagung

Ich möchte mich bei meinen Betreuern, Herrn Prof. Dr. E. Rahm und Herrn Dr. D. Sosna, für die interessante Aufgabenstellung und die ausgezeichnete Betreuung bedanken. Auch den anderen Mitarbeitern der Abteilung Datenbanken bin ich sehr dankbar für ihre freundliche Hilfe bei technischen Problemen.

Mein besonderer Dank gilt der Leiterin des Muskinstrumentenmuseums, Frau Dr. E. Fontana, für ihre unermüdliche Unterstützung und fachliche Beratung. Weiterhin möchte ich mich bei allen Mitarbeiterinnen und Mitarbeitern des Musikinstrumentenmuseums bedanken, insbesondere bei Frau C. Weiss und Herrn W. Hecht für die geduldige Beschaffung und Katalogisierung der Daten.

Nicht zuletzt möchte ich meinen Eltern und allen meinen Freunden für ihr Verständnis dafür danken, daß ich in den letzten Monaten nur wenig Zeit für sie hatte. Vor allem aber danke ich meiner Freundin Ella für ihre Unterstützung, ihre Geduld und ihre Wärme..

2 Anforderungen an multimediale Informationssysteme

Die klassische Aufgabe von Informationssystemen ist es, das Informationsbedürfnis eines bestimmten Benutzerkreises möglichst adäquat zu befriedigen. Aus diesem Grund müssen die gespeicherten Daten den zugrundeliegenden Problembereich exakt modellieren. Weiterhin muß die Benutzerschnittstelle des Systems einen schnellen und komfortablen Zugriff auf die Daten gewährleisten sowie insbesondere eine Suchfunktion anbieten. Eine selbstverständliche Anforderung ist, daß die Daten stets auf dem aktuellen Stand sein sollten. Um dieses Ziel zu erreichen, muß die Änderung der Anwendungsdaten und gegebenenfalls auch der Datenstruktur auf einfache Weise möglich sein. Informationssysteme, die einem weltweiten Anwenderkreis angeboten werden, sollten ihre Daten in mehreren Sprachen bereitstellen.

Informationssysteme werden zunehmend auch für die Präsentation von öffentlichen Einrichtungen oder Firmen eingesetzt. Sie sollten daher in der Lage sein, das Interesse eines Benutzers zu wecken und zu erhalten. Nicht zuletzt deshalb müssen neben Texten und numerischen Daten auch Bild-, Video- und Tondaten gespeichert und wiedergegeben werden können. Allerdings ist der Einsatz von multimedialen Daten nicht auf Präsentationen beschränkt. Sie können vielmehr überall dort vorteilhaft eingesetzt werden, wo ein Sachverhalt durch rein textliche Beschreibung nicht oder nur ungenügend veranschaulicht werden kann.

In der folgenden Aufzählung sollen die Anforderungen präzisiert werden, die sowohl an ein Informationssystem in seiner Gesamtheit als auch an jede Komponente, insbesondere das Datenbankschema, zu stellen sind. Jeder Punkt nennt eine grundlegende Anforderung, deren Notwendigkeit im eingerückten Teil begründet wird. Gegebenenfalls wird die Bedeutung einer Anforderung am Beispiel des Musikinstrumentenmuseums zusätzlich verdeutlicht.

Modellierung von heterogenen Problembereichen.
Informationssysteme sollen einen gegebenen Problembereich möglichst genau repräsentieren. Die wesentliche Voraussetzung dafür ist, daß dieser Problembereich exakt auf ein Datenschema abgebildet wird. Besonders wissenschaftliche Kataloge können einen sehr komplexen Aufbau besitzen, was sich oft in einer großen Anzahl von Typen bei einer geringen Anzahl von Ausprägungen je Typ niederschlägt.

Der Datenbestand des Musikinstrumentenmuseums liefert ein gutes Beispiel für diese Feststellung: Die derzeit etwa 5000 Exponate des Museums lassen sich in mehr als 100 Kategorien einteilen. Daneben sind weitere Klassen wie Sammlung, Standort, Besitzer und Erbauer zu modellieren und in Relation zueinander und zu den Exponaten zu setzen.

Leichte Änderbarkeit und Erweiterbarkeit des Datenbestandes.
Zu den selbstverständlichen Fähigkeiten, die ein datenbankbasiertes Informationssystem besitzen muß, gehören der Import neuer Anwendungsdaten und die Aktualisierung des bestehenden Datenbestandes. Der Datenimport muß in einer einfachen und flexiblen Weise vor sich gehen und stets einen konsistenten Datenbestand erzeugen, auch dann, wenn die Daten mehrsprachig vorliegen oder multimedialen Charakter haben.

Leichte Änderbarkeit und Erweiterbarkeit des Datenschemas.
Das Hinzufügen neuer Anwendungsdaten erfordert oft eine Anpassung des Schemas. Der Grund dafür ist, daß eine öffentliche Einrichtung oder eine Firma gerade dann den Datenbestand des Informationssystems anpassen möchte, wenn sich der modellierte Problembereich geändert hat. Der Problembereich ändert sich beispielsweise dann, wenn ein Produkt mit neuen Eigenschaften eingeführt wird oder ein neues Museumsexponat die Erweiterung der Klassifikation erfordert. In solchen Fällen muß es möglich sein, das Datenschema anzupassen, ohne die aufgesetzten Applikationen ändern zu müssen.

Für das Musikinstrumentenmuseum gilt, daß die Klassifikation der Exponate zum Zeitpunkt des Entstehens dieser Arbeit noch nicht abgeschlossen war. Hinzu kommt, daß ein neu erworbenes Exponat oft in eine Kategorie fällt, die bisher nicht berücksichtigt wurde.

Einfache und transparente Kopplung von Datenbank und Anwendungen.
Weil neben den Anwendungsdaten auch das Datenschema eines Informationssy-

stems häufigen Änderungen unterliegt, dürfen die Datenbank-Applikationen nicht von einem bestimmten Datenschema abhängig sein. Weiterhin sollte jede Applikation geändert werden können, ohne daß dies Rückwirkungen auf das Datenbankschema und das jeweilige Datenschema hat. Nicht zuletzt sollte die Anbindung einer Applikation an die Datenbank mit geringem technischen und zeitlichen Aufwand möglich sein. Beispielsweise darf der Wechsel von einer fensterbasierten Oberfläche zu einem Hypertext-System nicht bedeuten, daß in der neuen Applikation das vollständige Wissen über ein bestimmtes Datenschema fest verankert werden muß.

Die Unabhängigkeit zwischen der Struktur der Anwendungsdaten und den Applikationen kann nur erreicht werden, wenn das Datenschema flexibel ist sowie umfangreiches Wissen über den eigenen Aufbau hat. Aus diesem Grund muß das *Datenschema*, das die Struktur der Anwendungsdaten bestimmt, vom *Datenbankschema*, das die generische Funktionalität zur Verwaltung der Anwendungsdaten bereitstellt, entkoppelt werden. Abhängigkeiten bestehen dann nur noch zwischen dem verhältnismäßig einfachen statischen Datenbankschema und den Applikationen. Die häufig sehr komplexe Struktur der Anwendungsdaten kann hingegen zur Laufzeit abgefragt und interpretiert werden.

Speicherung und Retrieval von multimedialen Daten.

Multimediale Daten besitzen in Informationssystemen zwei Aufgaben: Sie *veranschaulichen* die numerisch oder in Textform abgespeicherten Daten und sie *fördern* das *Interesse* des Benutzers. Informationen, die kombiniert in Bild-, Ton-, und Textform angeboten werden, prägen sich dem Anwender weitaus besser ein als reine Textdaten [GM97, S. 11f]. Aus Informationssystemen im World Wide Web sind multimediale Daten nicht mehr wegzudenken; sie bestimmen maßgeblich, ob ein Informationssystem oder eine Präsentation von einem breiten Personenkreis angenommen wird.

Mehrsprachige Speicherung der Daten.

Für ein Informationssystem, insbesondere wenn es an das World Wide Web angebunden wird, ist eine durchgängige Mehrsprachigkeit aller Anwendungsdaten unerläßlich, da nur so ein breiter Interessentenkreis angesprochen werden kann. Neben den Anwendungsdaten müssen auch alle *beschreibenden* Informationen, sofern sie zur Darstellung benutzt werden, mehrsprachig vorliegen. Das betrifft beispielsweise die Namen von Objekten, Attributen und Verknüpfungen. Die Verwaltung

mehrsprachiger Daten muß in effizienter und konsistenter Weise geschehen, so daß ein Wechsel auf eine andere Darstellungssprache jederzeit problemlos möglich ist.

Integration eines Information-Retrieval-Systems.

Daten in einem Informationssystem liegen zum großen Teil in Textform vor. Dabei kann es sich um kurze Texte handeln, die in Attributen von Tabellen oder Objekten gespeichert werden, oder aber um größere, möglicherweise strukturierte Texte, die in *Binary Large Objects (BLOBs)* bzw. außerhalb der Datenbank abgelegt werden. Weiterhin ist die Struktur der Daten oft kompliziert, so daß ein hohes Maß an Wissen über die Datenstruktur für eine gezielte Suche nötig ist. So muß ein Benutzer, der sich für Angaben über eine bestimmte Person interessiert, bereits *vor* der Suche wissen, *wo* er diese Informationen finden kann. Er muß wissen, ob es sich um einen Erbauer bzw. Besitzer eines Exponats handelt, ob die Person der Herausgeber einer Publikation ist oder ob der Name der Person lediglich in einer Bemerkung vorkommt.

Sowohl der hohe Textanteil als auch die Komplexität der Datenschemata in Informationssystemen sprechen für eine *Volltextsuche* im Datenbestand. Für diese Aufgaben sind Information-Retrieval-Systeme gut geeignet, da sie für die Indexierung und das Retrieval von Texten optimiert sind und einige weitere Eigenschaften besitzen, die von Standard-DBMS nicht unterstützt werden. Dazu gehören die Fähigkeit, natürlichsprachliche Anfragen zu verarbeiten, und die Relevanzbewertung der Suchergebnisse.

3 Systementwurf

3.1 Einleitung

Während sich die Analyse einer Problemstellung damit befaßt, *was* zu tun ist, ist es die Aufgabe des Entwurfs festzulegen, *wie* dies geschieht. Der Systementwurf ist ein Teil des Entwurfsprozesses. Hier wird die Gesamtstruktur des Systems festgelegt und entschieden, welcher Architekturstil benutzt wird. Große Systeme besitzen oft eine Vielzahl von Funktionen und sind in ihrer Gesamtheit schwer zu überschauen. Eine Hauptaufgabe des Systementwurfs ist es daher, das System in Komponenten zu zerlegen, die physisch und logisch in sich geschlossene Einheiten bilden und nur über exakt definierte Schnittstellen miteinander kommunizieren. Das Design der einzelnen Komponenten wird an dieser Stelle noch nicht festgelegt; der Systementwurf bildet lediglich ein Rahmenwerk für das Zusammenspiel der Teile. Einzelne Komponenten können unabhängig voneinander entwickelt und variiert werden, solange die Konventionen der Schnittstelle eingehalten werden. Auf diese Weise können mehrere Personen gemeinsam an einem Projekt arbeiten. Ebenso ist ein späterer Austausch einzelner Komponenten durch angepaßte Versionen möglich, ohne daß das System in seiner Gesamtheit geändert werden müßte. Vertiefende Betrachtungen über komponentenbasierte Software bieten [RBP+94], [Str97] und [Wee97].

Nach [RBP+94, S. 242] gehören zu den Aufgaben, die während des Systementwurfs zu lösen sind, unter anderem die folgenden Punkte:

- Aufteilen des Systems in Komponenten
- Anordnen der Komponenten in Schichten und Partitionen
- Festlegen der Kommunikation zwischen den Teilsystemen

Im nächsten Abschnitt werden die Komponenten herausgearbeitet, aus denen die durch

die Anforderungsanalyse bereits grob umrissene Gesamtarchitektur bestehen soll. Anschließend wird die horizontale und vertikale Gliederung festgelegt, indem untersucht wird, welche Teilsysteme voneinander abhängig sind und welche Schnittstellen zwischen den Komponenten vorgesehen werden müssen (Abschnitt 3.3). Falls Teilsysteme voneinander abhängig sind, müssen sie auf eine festgelegte Weise miteinander kommunizieren (Abschnitt 3.4).

3.2 Aufteilung in Komponenten

Die Untergliederung des zu entwerfenden Systems ergibt sich zum großen Teil bereits aus der Analyse der Anforderungen. Neben der Grundaufgabe, die Anwendungsdaten zu speichern, muß eine Möglichkeit vorhanden sein, diese Daten in die Datenbank einzulesen und gegebenenfalls zu ändern. Die für die WWW-Anbindung zuständige Applikation setzt, im Zusammenspiel mit einem Information-Retrieval-System, auf die Datenbank auf.

Datenbank und Datenbankschema. Als persistentes Speichersystem für die Anwendungsdaten kommt ein Datenbank-Management-System (DBMS) zum Einsatz. DBMS haben gegenüber der Speicherung im Dateisystem des Betriebsystems eine Reihe von Vorteilen: Sie sind in der Lage, sehr große Datenbestände auf einheitliche Weise zu verwalten; sie gestatten einen kontrollierten Mehrbenutzerbetrieb, erlauben die Rekonstruktion von Daten im Fehlerfall, bieten ein Transaktionskonzept und verfügen nicht zuletzt über mächtige Anfragesprachen.

Objektorientierte DBMS (OODBMS) besitzen eine Reihe von Zusatzeigenschaften. Für die Verwaltung stark strukturierter Daten sind insbesondere die folgenden Merkmale von großem Nutzen [MW97, S. 5]:

- Verwaltung komplexer Objekte
- Datenkapselung
- Vererbung und Polymorphie
- Modellierungsvollständigkeit
- Erweiterbarkeit

Diese Merkmale ermöglichen es einem OODBMS, einen heterogenen Problembereich auf nn vorher in normalisierte Bestandteile zerlegen zu müssen. OODBMS unterstützen außerdem Referenzen bzw. Zeiger auf persistente Objekte. Sie sind deshalb gut für die Implementierung des Datenbankschemas geeignet, welches stark auf dem Zeigerkonzept aufbaut (Kapitel 4).

Anstelle eines OODBMS kann auch ein objektrelationales DBMS (ORDBMS) eingesetzt werden, da diese DBMS ebenfalls alle wichtigen objektorientierten Konzepte bereitstellen. Voraussetzung ist lediglich, daß das ORDBMS die Anbindung einer (objektorientierten) Programmiersprache erlaubt, um die Kommunikation mit den anderen Komponenten des Systems zu ermöglichen. Weiterhin sollte das ORDBMS das Konzept des Datenbankzeigers unterstützen bzw. effizient emulieren.

Schema- und Datenlader. Weil das Datenbankschema, das in Kapitel 4 vorgestellt wird, die dynamische Schemaerzeugung erlaubt, wird ein Ladeprogramm benötigt, mit dessen Hilfe die Beschreibung eines Datenschemas gelesen und interpretiert werden kann. Aus der Schemadefinition wird ein Datenschema erstellt, das die Struktur der Anwendungsdaten vorschreibt. Erst nachdem ein Datenschema existiert, können die Anwendungsdaten vom Datenlader in die Datenbank importiert werden. Schema- und Datenlader sind die einzigen Komponenten des Gesamtsystems, die *schreibend* auf die Datenbank zugreifen. Sie stellen zugleich Schnittstellen bereit, die von Programmen zur *interaktiven* Schema- und Datenmanipulation genutzt werden können (Kapitel 5).

WWW-Anbindung. Die Aufgabe der WWW-Anbindung ist es, die Anwendungsdaten zu visualisieren. Die Komponente nimmt Nutzeranfragen vom WWW-Server entgegen, analysiert sie, selektiert die gewünschten Daten aus der Datenbank und generiert daraus Hypertext-Dokumente, die an den WWW-Server weitergereicht werden. Die Komponente stellt daneben Funktionen zur strukturierten Suche bereit und bildet die Schnittstelle zum Information-Retrieval-System (Kapitel 6).

Information-Retrieval-System (IRS). Daten in Informationssystemen liegen oft in Textform vor. Die Anbindung eines IRS an die Datenbank gestattet es, auf einfache und einheitliche Weise in allen Texten des Datenbestandes zu suchen.

Durch seine spezielle Art, Textattribute zu verwalten, ist das Datenbankschema nicht auf ein bestimmtes IRS festgelegt. Folgende Varianten für die Anbindung eines IRS sind denkbar:

- Das IRS ist eine eigenständige Applikation, die mit dem Datenbankschema und der WWW-Anbindung kommuniziert.
- Das IRS ist eine Programm-Bibliothek, die in das Gesamtsystem fest eingebunden ist (Kapitel 7).
- Das IRS ist eine proprietäre DBMS-Erweiterung in Form eines *Text Extenders* oder eines *Text Data Blades*.

Benutzerschnittstelle. Als Schnittstelle zu den Anwendern kommen WWW-Browser zum Einsatz. Sie erhalten ihre Daten von einem Standard-WWW-Server, der wiederum mit dem Teilsystem WWW-Anbindung zusammenarbeitet. Daneben sind auch andere Schnittstellen zum Anwender denkbar, so z.B. grafische Oberflächen von Betriebssystemen.

3.3 Gliederung in Schichten und Partitionen

Unter der Bildung von *Schichten* versteht man die vertikale Gliederung eines Systems in Teilsysteme. Objekte innerhalb einer Schicht können unabhängig voneinander sein. Jede Schicht wird durch die darunterliegenden Ebenen bestimmt; höhere Schichten abstrahieren von der Funktionalität tieferer Schichten. Jede Schicht kennt die darunterliegenden Ebenen und kommuniziert mit ihnen. Im Gegensatz dazu sind höherliegende Schichten einer darunterbefindlichen Ebene nicht bekannt.

Unter *Partitionen* werden Teilsysteme verstanden, die nicht oder nur wenig voneinander abhängig sind. Jede Partition erfüllt einen speziellen Aufgabenbereich, durch den sie von anderen Teilsystemen abgegrenzt ist. Partitionen können Wissen übereinander besitzen; größere Entwurfsabhängigkeiten bestehen jedoch nicht.

Abbildung 3.1 zeigt die Einteilung des Gesamtsystems in Schichten und Partitionen. Auf der untersten Ebene der zu entwerfenden Architektur ist die Computer- und Netzwerk-Hardware angesiedelt, auf die möglicherweise verschiedene Betriebssysteme, die durch ein

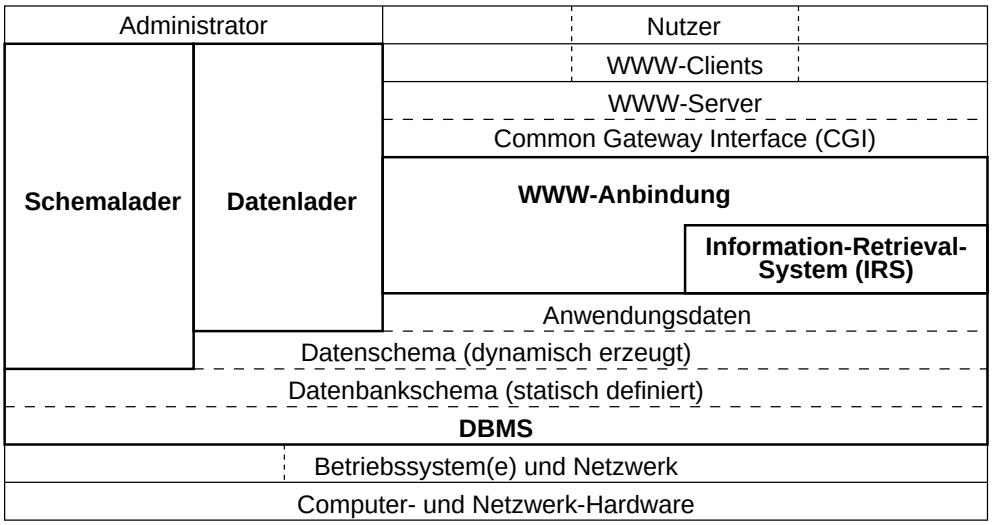

Abbildung 3.1: Einteilung des Systems in Schichten und Partitionen

Netzwerk verbunden sind, aufsetzen. Das DBMS, das diese Schicht nutzt, kann über eine Client/Server-Architektur verfügen. Auf diese Weise müssen Datenbank- und WWW-Server nicht auf demselben Rechner laufen. Weiterhin können die Ladeprogramme auf einem beliebigen Rechner des lokalen Netzwerks gestartet werden.

Die Komponenten Schemalader, Datenlader, IRS und WWW-Anbindung bauen auf der Datenbank auf. Durch gestrichelte Linien ist die innere Schichtung angedeutet: Das Datenbankschema bildet nach dem Datenbank-Kern die unterste, statisch definierte Schicht. Der Schemalader kommuniziert unmittelbar mit dem Datenbankschema; der Datenlader hingegen benötigt ein Datenschema, um die Anwendungsdaten importieren zu können. Das IRS sowie die WWW-Anbindung schließlich setzen Datenschema und Anwendungsdaten voraus. Die WWW-Anbindung ist die Schnittstelle zu einem WWW-Server, der wiederum die Basis für einen oder mehrere WWW-Browser darstellt. Die oberste Schicht symbolisiert die Benutzer des Informationssystems sowie den Datenbank-Administrator, der für den Import des Schemas und der Daten zuständig ist.

In horizontaler Richtung ist das System in Schemalader, Datenlader, IRS und WWW-Anbindung unterteilt. Jedes dieser Systeme ist funktional weitgehend unabhängig von den anderen. Jedoch bestehen die oben beschriebenen Datenabhängigkeiten zwischen Schema- und Datenlader, IRS und WWW-Anbindung.

3.4 Kommunikation zwischen den Komponenten

Die Kommunikation unter den Teilsystemen betrifft in erster Linie den Nachrichtenaustausch zwischen den Schichten. Je nach betrachteter Ebene kommen recht unterschiedliche Kommunikationsformen zum Einsatz. Das DBMS nutzt die Programmierschnitt-

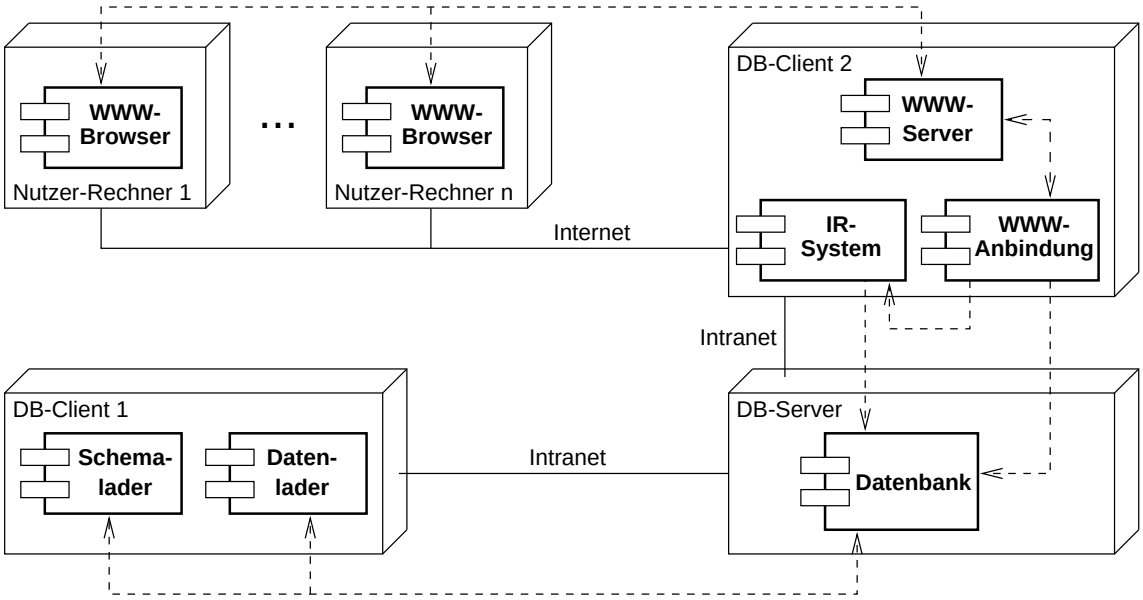

Abbildung 3.2: Verteilungsdiagramm

stelle des Betriebssystems. Im Fall eines Client/Server-DBMS gehören dazu auch die Netzwerkfunktionen, die das DBMS zum Nachrichtenaustausch nutzt. Die Kommunikation zwischen DBMS und der darüberliegenden Schicht erfolgt über eine standardisierte Programmierschnittstelle. Strenggenommen handelt es sich dabei um eine weitere Schicht: Das DBMS bildet auf der einen Seite datenbankinterne Funktionen auf eine Programmiersprache ab und/oder stellt einen Präprozessor bereit, der datenbankspezifische Konstrukte in Aufrufe der Programmiersprache umsetzt [Rah98, S. 8-9]. Auf der anderen Seite kommunizieren die Lader, das IRS und die WWW-Anbindung über die Programmierschnittstelle mit dem DBMS.

Der WWW-Server bindet die WWW-Komponente sowie indirekt das IRS über das *Common Gateway Interface (CGI)* ein und stellt die Schnittstelle zum World-Wide-Web dar. Bei einer Client/Server-Datenbank können sich WWW-Server und Datenbank-Server auf unterschiedlichen Rechnern befinden.

Die Verbindung zwischen den WWW-Browsern, die als Schnittstelle zu den Anwendern fungieren, und dem WWW-Server wird über das *Hypertext Transfer Protokol (HTTP)* hergestellt. Der Nutzer ruft eine Funktion der WWW-Anbindung über einen *Unified Resource Locator (URL)* auf. Das Ergebnis der Anfrage wird in Form eines HTML-Dokuments geliefert, das dynamisch von dieser Komponente erstellt wird.

Der Datenbank-Administrator kommuniziert mit dem Schema- und dem Datenlader über die Kommandozeilen dieser Programme. Mögliche Kommandozeilen-Parameter sind die Namen der Dateien, welche die Schema- und Datendefinitionen enthalten.

Abbildung 3.2 zeigt die physische Aufteilung des Systems und hebt besonders die Kommunikationswege und die Abhängigkeiten zwischen den Komponenten hervor.

3.5 Zusammenfassung

Die Aufgabe des Systementwurfs ist es, eine komplexe Applikation in ihre Bestandteile aufzubrechen, horizontal und vertikal zu gliedern sowie die Kommunikation zwischen den Komponenten festzulegen.

Die Architektur des zu entwickelnden Informationssystems besteht aus den Komponenten DBMS mit Datenbankschema, Schemalader, Datenlader, WWW-Anbindung, IRS und Benutzerschnittstelle.

Ein fundamentaler Bestandteil ist das DBMS, welches das Datenbankschema aufnimmt. Der Schemalader erzeugt dynamisch ein Datenschema. Auf der Grundlage des Datenschemas importiert der Datenlader die Anwendungsdaten. Die WWW-Komponente verbindet das Datenbankschema mit dem WWW-Server, der wiederum mit den WWW-Browsern kommuniziert.

Das Datenbankschema wird über eine programmiersprachliche Schnittstelle an die Komponenten Schemalader, Datenlader, WWW-Anbindung und IRS angebunden. Die Lader erhalten ihre Befehle über die Kommandozeile. Die WWW-Anbindung kommuniziert mit den WWW-Browsern über das HTTP-Protokoll.

4 Entwurf des Datenbankschemas

4.1 Einleitung

Das Datenbankschema bildet den Kern der Architektur des Systems und stellt die Basis für alle anderen Komponenten dar. Der Entwurf des Datenbankschemas muß einer Reihe von Anforderungen genügen: Erstens muß das Datenbankschema es erlauben, reale Problembereiche exakt zu modellieren. Zweitens soll es nicht auf ein bestimmtes Anwendungsgebiet beschränkt sein, sondern in der Lage sein, Datenschemata dynamisch zu erstellen und an veränderte Bedingungen anzupassen. Drittens sollen nur geringe Abhängigkeiten zwischen einem Datenschema und den Datenbank-Applikationen bestehen, so daß das Datenschema und die Anwendungen unabhängig voneinander variiert werden können. Viertens sollten die Komplexität und der Ressourcenverbrauch des Datenbankschemas so gering wie möglich sein.

Am Anfang des Kapitels werden zunächst einige wichtige Begriffe eingeführt und diskutiert. Danach wird dargelegt, warum sich Methoden aus dem Bereich der objektorientierten Softwaretechnik für den Datenbankentwurf eignen und worin ihre Grenzen hinsichtlich der in Kapitel 2 gestellten Anforderungen liegen. Anschließend wird analysiert, welche Nachteile entstehen, wenn die Datenbank und die angekoppelte Programmiersprache kein flexibles und selbstbeschreibendes Datenmodell besitzen. Ausgehend von diesen Nachteilen wird im weiteren Verlauf des Kapitels ein Datenbankschema entwickelt, auf dessen Grundlage dynamisch änderbare, reflexive Datenschemata erstellt werden können. Zum Schluß wird untersucht, welche Vorteile und Grenzen das entwickelte Modell aufweist.

4.2 Vorbetrachtungen

4.2.1 Anmerkungen zu den verwendeten Begriffen

Objekt und Klasse

In der Literatur über objektorientierte Programmiersprachen und Datenbanken existiert eine Vielzahl von abweichenden Definitionen der Begriffe *Objekt*, *Klasse* und *Typ* sowie der damit verbundenen Konzepte *Extension*, *Typ-* und *Klassenhierarchie* [Heu97].

Die Autoren von [RBP+94, S. 27] definieren ein Objekt als "Konzept, Abstraktion oder einen Gegenstand mit klaren Abgrenzungen und einer präzisen Bedeutung für das anstehende Problem". Etwas konkreter ist die Definition von Rao: "Ein Objekt ist die Einkapselung von Daten, die ein reales Ding repräsentiert, mit Operationen, die die Daten manipulieren. Objekte interagieren mit anderen Objekten, indem sie Dienste bereitstellen oder Dienste anderer Objekte anfordern." [zitiert nach STS97, S. 61]

Während sich in den Definitionen des Begriffs Objekt noch einige Gemeinsamkeiten finden lassen, wird der Begriff der Klasse unter zwei völlig gegensätzlichen Gesichtspunkten betrachtet. Im Zusammenhang mit objektorientierten Programmiersprachen überwiegt die *intensionale* Sichtweise: Eine Klasse ist hier ein programmiersprachliches Konstrukt, das einen komplexen Typ kapselt und implementiert. Der Typ legt den Aufbau der (potentiellen) Objekte der Klasse fest. In diesem Modell wird unter *Typhierarchie* die Vererbung des Verhaltens verstanden, während der Begriff *Klassenhierarchie* die Vererbung der Implementierung beschreibt.

Im Bereich der objektorientierten Datenbanken wird das *extensionale* Prinzip bevorzugt. Eine Klasse besitzt hier den Charakter einer *Menge* von (potentiellen) Objekten mit gleicher Struktur und Semantik. Die *Extension* einer Klasse ist die Menge der bislang erzeugten und noch nicht wieder gelöschten Objekte. Der Begriff der *Klassenhierarchie* beschreibt eine Inklusion von Objektmengen. "Eine Klasse $\mathcal{K}$ ist spezieller als eine Klasse $\mathcal{K}'$, wenn die Objektmenge von $\mathcal{K}$ Teilmenge der Objektmenge von $\mathcal{K}'$ ist." [Heu97, S. 318] Auch hier beschreibt der Begriff *Typhierarchie* die Vererbung des Verhaltens.

In ihrem Buch zur *Object Modeling Technique* betonen J. Rumbaugh et al. ebenfalls den extensionalen Aspekt des Klassenbegriffs: "Eine *Objektklasse* beschreibt eine Gruppe von Objekten mit ähnlichen Eigenschaften (Attributen), gemeinsamen Verhalten

(Operationen), gemeinsamen Relationen zu anderen Objekten und einer gemeinsamen Semantik." [RBP+94, S. 28]

Das Datenbankschema, das in diesem Kapitel entworfen wird, stellt lediglich generische Funktionen zum Verwalten von Objekten bereit. Instanzen einer Klasse des Datenbankschemas können sich in ihren Attributen, Verknüpfungen und Komponenten unterscheiden. Es liegt somit nahe, Mengen von Objekten mit gleicher Struktur und Semantik zu Klassen nach extensionaler Definition zusammenzufassen. Der Übergang zur intensionalen Sichtweise wird mit der Einführung der Schemaebene vollzogen, die zum Zeitpunkt des Datenimports die Struktur aller Objekte vorschreibt.

Zur besseren Abgrenzung der Begriffe werden Klassen, die aufgrund gleicher Struktur und Semantik von Objekten entstehen, *extensionale Klassen* genannt. Ihre Namen werden in Sans-Serif-Font gesetzt. Die in einer Programmiersprache definierten Klassen des Datenbankschemas werden mit *Implementierungsklasse* bezeichnet. Namen von Implementierungsklassen werden in `Typewriter`-Schrift dargestellt. Wenn die Bedeutung aus dem Kontext hervorgeht, wird in allen Fällen kurz von *Klasse* gesprochen. Zwischen Typ- und Klassenhierarchie wird nicht unterschieden. Entlang einer Spezialisierungshierarchie wird sowohl das Verhalten vererbt als auch die Bedingung erfüllt, daß die Extension einer Unterklasse Teilmenge der Extension der Oberklasse sein muß.

Klassenobjekte und Metaklassen

In den meisten objektorientierten Programmiersprachen wird eine Klasse vollständig zur Übersetzungszeit spezifiziert. Die zur Laufzeit möglichen Operationen sind Objekterzeugung und -vernichtung sowie Änderung des Objektzustands. Damit wird die *Struktur* eines Objekts zur Übersetzungszeit festgelegt und bleibt konstant. Einige objektorientierte Programmiersprachen und OODBMS heben diese Einschränkung auf und fassen Klassen wiederum als Objekte auf. Zur besseren Unterscheidung von statisch definierten Klassen nennt man solche Klassen auch *Klassenobjekte*. Klassenobjekte besitzen die Eigenschaften von intensionalen Klassen: Sie *definieren* Struktur und Semantik ihrer Instanzen. Darüberhinaus lassen sich Klassenobjekte zur Laufzeit erzeugen, vernichten und in ihrem Zustand ändern. Klassenobjekte sind Instanzen von Klassen höherer Ebene, die *Metaklassen* genannt werden. In einigen Systemen ist die Struktur der Metaebene fest eingestellt, weshalb man von *impliziten Metaklassen* spricht. Andere Systeme erlauben

das Erzeugen, Ändern und Löschen von Metaklassen, die dann *explizite Metaklassen* genannt werden.

Verknüpfung und Assoziation

Nach [RBP+94] ist eine *Verknüpfung* eine physikalische oder konzeptuelle Verbindung zwischen Objekten. Aus mathematischer Sichtweise kann eine Verknüpfung als eine geordnete Liste von Objekten aufgefaßt werden. *Assoziationen* fassen Verknüpfungen in Gruppen zusammen, deren Elemente gleiche Struktur und Semantik besitzen. Verknüpfungen derselben Assoziation verbinden stets Objekte gleicher Klassen.

Assoziationen sind inhärent bidirektional. Allerdings beschreibt der *Name* einer Assoziation meist eine bestimmte Richtung, in der die Assoziation durchlaufen wird. Solche *unidirektionale* Assoziationen werden auf Implementierungsebene normalerweise durch Zeiger oder Referenzen modelliert. Sind beide Richtungen von Bedeutung, wird die *Rückrichtung* ebenfalls auf diese Weise gebildet.

Für Assoziationen definiert man den Begriff der *Kardinalität*. Die Kardinalität spezifiziert, wieviele Instanzen einer Klasse mit einer Instanz einer assoziierten Klasse verknüpft sein können. Die Kardinalität schränkt die Anzahl verknüpfter Objekte ein.

Eine *Aggregation* ist eine Assoziationen mit zusätzlichen Eigenschaften. Sie beschreibt eine Verbindung zwischen einem Ganzen und seinen Teilen. Das Ganze (Aggregat) nimmt die Aufgaben stellvertretend für die Teile wahr. Nachrichten werden vom Aggregat an die Teile weitergeleitet. Der umgekehrte Fall gilt nicht: Ein Teil kennt das Aggregat nicht. Die Lebensdauer eines Teils hängt *nicht* vom Ganzen ab. Es kann vor dem Aggregat erzeugt werden und nach dem Löschen des Ganzen weiterexistieren. Daher kann ein Teil zu mehreren Objekten gehören. Aggregationen sind transitiv: Ein Teil eines Aggregats kann seinerseits wiederum Teile besitzen.

Eine Erweiterung erfährt das Assoziationskonzept durch den Begriff der *Komposition*. Eine Komposition ist eine spezielle Variante der Aggregation, weswegen die meisten Eigenschaften der Aggregation auch für die Komposition zutreffen. Jedoch ist die Bindung an das Aggregat stärker: Die Lebenszeit der Teile ist der des Ganzen untergeordnet. Das Teil entsteht mit dem Ganzen oder im Anschluß daran und wird zerstört, bevor das Aggregat zerstört wird. Die zweite Restriktion betrifft die Kardinalität auf der Seite des

Aggregats. Sie kann höchstens eins sein, was besagt, daß es nicht mehr als einen Besitzer eines Teils geben darf.

Betrachtet man Aggregation und Komposition unter dem Gesichtspunkt der Implementierung, so werden nach [Oes97, S. 205] Aggregationen durch *Referenzierung* modelliert, Kompositionen hingegen durch *Einbettung*. Folglich besitzen Komponentenobjekte keine eigene Identität. Sie können nicht durch andere Objekte referenziert werden. In objektorientierten Datenbanken werden Komponentenobjekte nach [STS97, S. 113] nicht in die Extension ihrer Klasse aufgenommen.

In der vorliegenden Arbeit wird für Assoziationen, ebenso wie für Klassen, eine extensionale und eine intensionale Sichtweise benutzt. Assoziationen, die durch die *Zusammenfassung* von gleichen Verknüpfungen entstehen, werden *extensionale Assoziationen* genannt. Sie werden durch Sans-Serif-Font kenntlich gemacht. Assoziationen, die den Aufbau von Verknüpfungen *definieren*, werden *Assoziationsobjekte* genannt.

Alle in diesem Kapitel verwendeten Begriffe sind im Anhang B auf Seite 111 zusammengestellt.

4.2.2 Objektorientierte Methoden für den Datenbankentwurf

Für den Entwurf eines Datenbankschemas hat sich eine Einteilung in Phasen durchgesetzt [Vos94, S. 48ff]. In der wichtigen *konzeptionellen Phase* wird das *konzeptionelle Datenbankschema* erstellt, das unabhängig von dem später zur Implementierung verwendeten DBMS ist. Die am häufigsten angewendete Entwurfstechnik innerhalb der konzeptionellen Phase ist das *Entity-Relationship-Modell (ER-Modell)*. Obwohl sich das ER-Modell, erweitert um die Konzepte Aggregation und Generalisierung/Spezialisierung, auch für den Entwurf objektorientierter Schemata eignet, werden hier oft Methoden aus der objektorientierten Softwaretechnik eingesetzt. Wichtige Vertreter von objektorientierten Analyse- und Entwurfsmethodiken sind die *Objekt Modeling Technique (OMT)* von Rumbaugh et al. [RBP+94], die Methode von Booch [Boo94] und die *Use Cases* von Jacobson et al. [JCJO94]. Trotz der Unterschiede in vielen Details besitzen alle objektorientierten Entwurfsmethodiken zumindest bei der statischen Modellierung wesentliche Gemeinsamkeiten. Alle unterstützen die Konzepte Objekt, Klasse, Generalisierung/Spezialisierung und Assoziation [FS98, S. 64]. Aus diesem Grund sind prinzipiell alle diese Techniken auch für den Entwurf objektorientierter Datenbankschemata geeignet [STS97, S. 432].

Der Grundsatz der Unabhängigkeit zwischen Entwurf und Implementierung gilt unverändert auch für objektorientierte Entwurfstechniken. Zwar ist der Übergang vom Entwurf zur Implementierung am leichtesten zu vollziehen, wenn das DBMS ebenfalls objektorientierte Konzepte unterstützt. Aber auch andere Datenbankmodelle sind ohne Auswirkungen auf den Entwurf als Zielsysteme einsetzbar. "Die Verwendung eines objektorientierten Entwurfs geht über die Wahl einer Datenbank hinaus. Sie können hierarchische, netzwerkorientierte, relationale und objektorientierte Datenbanken entwerfen." [RBP+94, S. 443] Die Autoren des zitierten Werkes geben eine Heuristik an, mit der objektorientierte Entwürfe in ein relationales Datenbankschema überführt werden können.

Auffällig ist, daß keine der erwähnten Entwurfsmethoden auf das Konzept der *Persistenz* eingeht; auch die Modellierungssprache *Unified Modeling Language (UML)* kennt in der Version 1.1 dafür keine Notation. In der Praxis muß dies nicht unbedingt ein Nachteil sein, wenn als Zielsystem ein objektorientiertes DBMS gewählt wird. Ein wesentlicher Grund für die Entwicklung von OODBMS war, die Konzepte der objektorientierten Softwaretechnik auf die Datenbanktechnologie zu übertragen. Besonderer Wert wurde darauf gelegt, beliebig komplexe Anwendungsobjekte auf *transparente Weise* um persistente Eigenschaften anzureichern. In diesem Zusammenhang erwähnenswert ist der ODMG-Standard, mit dem eine einheitliche Schnittstelle zwischen objektorientierten Programmiersprachen und Datenbanken geschaffen wird. Während des konzeptionellen Entwurfs muß also nur festgelegt werden, *welche* Schemabestandteile persistent sein sollen und nicht, *wie* dies geschieht.

Nachteilig an den objektorientierten Analyse- und Entwurfstechniken ist, daß sie vor allem auf "klassische" Anwendungsgebiete zugeschnitten sind. Diese Anwendungsgebiete sind dadurch gekennzeichnet, daß sich die modellierte Welt, einmal analysiert, nicht mehr ändert und daß weiterhin das Modell keine Informationen über sich selbst benötigt. Schwierigkeiten entstehen dann, wenn der Anwendungsbereich zum Zeitpunkt der Analyse nicht in jedem Detail feststeht, sich oft ändert oder generell nur unscharf umrissen werden kann.

4.2.3 Datenbanken und Datenbank-Programmiersprachen

Flexibilität

Eine Stärke von DBMS ist es, Änderungen am Datenbestand schnell und komfortabel vornehmen zu können. Die aktuelle Ausprägung der Daten muß dabei stets den Vorgaben des Datenbankschemas genügen. Das Datenbankschema bestimmt die Struktur und den Wertebereich jeder Relation bzw. den Aufbau und die Zustandsmenge jedes Objekts. In Informationssystemen ist über die Änderung der Daten hinaus auch eine gelegentlich notwendige Anpassung des Datenbankschemas zu erwarten. Das ist beispielsweise immer dann der Fall, wenn in einen Museumskatalog neue Exponate aufgenommen werden, für die bisher keine Klassifikation existierte, oder eine Firma ein neues Produkt entwickelt hat, dessen Eigenschaften durch das bisherige Datenbankschema nicht adäquat modelliert werden.

Nahezu alle DBMS erlauben die Anpassung oder Erweiterung eines bestehenden Datenbankschemas. In relationalen Datenbanken wird ein Schema durch die Definition von neuen Tabellen erweitert; bestehende Tabellen können durch den SQL-Befehl `ALTER TABLE` geändert werden. Die Möglichkeiten für die Schema-Evolution in objektorientierten DBMS reichen je nach Datenbank-Produkt von der Änderung des Schemas, für das noch keine Instanzen vorliegen, bis hin zur interaktiven Schema-Anpassung mit sofortigem Update der zugrundeliegenden Objektmenge [Heu97, S. 393ff].

In der Praxis sind diese Fähigkeiten oft nur bedingt nutzbar. Der Grund dafür ist, daß es zu jedem Datenbankschema sinnvollerweise Applikationen geben muß, die mit der Datenbank interagieren. Die gegenwärtig am meisten zur Datenbankprogrammierung verwendeten Sprachen sind C/C++ sowie in zunehmenden Maße Java. Zu den wichtigen Gemeinsamkeiten dieser Sprachen gehört das statische Typsystem. Typisierte Programmiersprachen verlangen, daß die Definition der Datenstrukturen zur Übersetzungszeit feststeht. Erstens können auf diese Weise Typfehler frühzeitig erkannt werden. Zweitens sind die so entstehenden Programme effizienter, da symbolische Namen auf Speicheradressen abgebildet werden. In objektorientierten Programmiersprachen werden die Attribute einer Klasse normalerweise vor dem unmittelbaren Zugriff verborgen. Die Manipulation der Daten erfolgt stattdessen über Methoden. Durch geschickten Einsatz von Vererbung und Polymorphie kann ein gewisses Maß an Flexibilität erreicht werden. Wird allerdings ein neues Attribut oder eine neue Assoziation benötigt, so läßt

sich eine Änderung und Neuübersetzung der betreffenden Klasse sowie aller abhängigen Applikationen kaum umgehen.

Reflexivität

Die Reflexivität einer Datenstruktur ist eine Eigenschaft, die sehr eng mit dem Begriff der Flexibilität zusammenhängt. Man versteht darunter die Fähigkeit der Datenstruktur, Informationen über den eigenen Aufbau bereitstellen zu können.

Viele DBMS besitzen reflexive Eigenschaften, wenn auch in sehr unterschiedlicher Ausprägung. In relationalen DBMS werden die Tabellendefinitionen im sogenannten *Data-Dictionary* oder Systemkatalog abgelegt und können teilweise auch vom nichtautorisierten Benutzer abgefragt werden. Einige objektorientierte DBMS besitzen ein Metaschema, das Informationen über die in der Datenbank definierten Datenbankschemata verwaltet. Auch das Metaschema kann prinzipiell von den Anwendungsprogrammen genutzt werden.

Leider sind die Möglichkeiten der Metaebene einer Datenbank in angekoppelten typisierten Programmiersprachen kaum einsetzbar, auch dann nicht, wenn die in der Metaebene abgelegten Informationen von den Applikationen abgefragt werden können. Eine Anwendung erhält die Metainformationen in Form von Datenwerten, beispielsweise numerisch oder als Zeichenketten. Auf die Felder einer Struktur oder die Attribute und Methoden einer Klasse wird hingegen über *symbolische* Namen zugegriffen, die während der Programmübersetzung auf relative Adressen abgebildet werden. Jede explizite Abbildung von Zahlen oder Zeichenketten auf symbolische Namen innerhalb der Anwendungen erfordert externes Wissen über das Datenbankschema.

Falls ein Datenbankschema kein selbstbezügliches Wissen besitzt oder falls das Wissen nicht von der angekoppelten Programmiersprache verwendet werden kann, sind die Abhängigkeiten zwischen dem Datenbankschema und den Applikationen notwendigerweise sehr hoch. Jede Anwendung muß den Aufbau aller von ihr benutzten Tabellen bzw. die Definition der Schnittstellen aller verwendeten Klassen kennen. Daher erfordert eine Schema-Evolution in der Regel die Anpassung der betreffenden Applikationen. Falls das Datenbanksystem die Abfrage der Metaebene zuläßt, benötigen die Applikationen trotzdem eine Funktion, mit der Informationen aus dem Metaschema auf die Symbolbezüge der Programmiersprache abgebildet werden können. Ist der Zugriff auf das Metasche-

ma nicht möglich, so muß das Wissen über jede verwendete Tabelle oder Klasse in der Anwendungslogik verankert werden.

Lösungsmöglichkeiten

Um die beschriebenen Nachteile zu umgehen, können zwei Wege beschritten werden. Erstens kann eine untypisierte oder dynamisch typisierte Programmiersprache mit vollständigem Metamodell eingesetzt werden. Zweitens kann die Typisierung der Programmiersprache durch eine dynamische *Data Manipulation Language (DML)* umgangen werden. Beide Ansätze erfordern, daß das DBMS über ein Metamodell verfügt bzw. das Metamodell der Programmiersprache unterstützt.

Untypisierte Programmiersprachen. Zu den objektorientierten Programmiersprachen, die über ein vollständiges Metamodell verfügen, gehören beispielsweise Smalltalk und CLOS. Diese Sprachen sind gleichzeitig untypisiert bzw. dynamisch typisiert und erlauben es deshalb, aus dem Metamodell die Namen von Attributen und Methoden einer Klasse zu ermitteln und mit Hilfe dieser Angaben auf die Attribute bzw. Methoden zuzugreifen, ohne daß externes Wissen notwendig wäre. In [Str97, S. 98] ist der Quelltext eines "Inspektors" angegeben, mit dem Namen und Werte jedes beliebigen Objekts ausgegeben werden können. Auch Java besitzt seit der Version 1.1 das sogenannte Reflexion-API, mit dem die Struktur von Objekten abgefragt werden kann. Darüber hinaus kann eine Methode über ihren in Stringform vorliegenden Namen aufgerufen werden, was einen harten Einschnitt in die strenge Typisierung dieser Programmiersprache darstellt.

Dynamische DML. Falls das DBMS und die Programmiersprache die Integration einer dynamischen DML erlauben, so kann diese ebenfalls dazu benutzt werden, Strukturinformationen zur Laufzeit zu erhalten und gegebenenfalls auf eine Änderung des Datenbankschemas zu reagieren. Das wohl bekannteste Beispiel für eine integrierte dynamische DML ist *Dynamic SQL*. Dieser Standard entstand aus der Absicht heraus, auch solche Queries innerhalb einer Wirtssprache auszuführen, deren Aufbau zur Übersetzungszeit noch nicht feststeht. Ein typisches Anwendungsgebiet sind Applikationen, die Ad-Hoc-Anfragen über ein Terminal entgegennehmen. Da die Struktur der Ergebnismenge einer dynamisch generierten Query erst zur Laufzeit bekannt ist, muß die Wirtssprache notwendigerweise mit Informationen *über* das Ergebnis versorgt werden.

Dynamic SQL benutzt zu diesem Zweck *Deskriptoren*, die unter anderem den Namen und den Typ jeder Tabellenspalte des Ergebnisses liefern. Mit Hilfe dieser implizit gegebenen Metainformationen kann ein "Inspektor" für beliebige Tabellen implementiert werden (z.B. durch die Erweiterung des Beispiels in [DD93, S. 316]).

Die Benutzung von Dynamic SQL ist dann sinnvoll, wenn das Ergebnis der Query *am Ort* ausgewertet werden kann. Die numerisch kodierten Ergebnistypen können per vollständiger Fallunterscheidung einer geeigneten Variable oder Funktion zugeordnet werden. Schwierigkeiten entstehen immer dann, wenn ein Tupel, dessen Struktur erst zur Laufzeit bekannt ist, in *seiner Gesamtheit* zwischengespeichert oder an eine (statisch definierte) Funktion weitergegeben werden soll. Hier muß die Ergebnismenge wiederum in dynamisch erzeugten Datenstrukturen abgelegt werden.

Dynamic SQL vergrößert den ohnehin vorhandenen *Impedance Mismatch* zwischen relationalen DBMS und typisierten Programmiersprachen. Die Typprüfung des Compilers wird vollständig ausgeschaltet. Es liegt in der Verantwortung des Programmierers, die korrekte Zuordnung der Datentypen zu Variablen der Programmiersprache zu gewährleisten. Die auf diese Weise entstehenden Programme sind kompliziert und fehleranfällig.

Schlußfolgerung

Die vorgestellten Lösungsmöglichkeiten besitzen einen gemeinsamen, sehr wesentlichen Nachteil: Sie machen den Entwurf von der zur Implementierung verwendeten Programmiersprache und/oder dem eingesetzten DBMS abhängig. So sind die Vorteile von untypisierten Programmiersprachen mit vollständigem Metamodell nur dann nutzbar, wenn das angekoppelte DBMS die Metaebene und die reflexiven Fähigkeiten der Programmiersprache komplett unterstützt. Der Lösungsansatz über die DML eines Datenbanksystems kann nur dann verwendet werden, wenn das DBMS über eine *eingebettete dynamische DML* verfügt und eine Metaebene bereitstellt, die innerhalb der Applikationen explizit oder implizit abgefragt werden kann. Dynamic SQL ist eng an das relationale Paradigma gekoppelt und wird darüberhinaus erst ab dem *Intermediate Level* des SQL92-Standards vorgeschrieben. Viele DBMS unterstützen jedoch nur den *Entry Level* (vgl. z.B. [Bay97]).

Entscheidungen für eine bestimmte Programmiersprache oder ein spezielles Datenbankprodukt sollten keinesfalls schon während des *konzeptionellen* Entwurfs gefällt werden [RBP+94, Vos94]. Andererseits müssen solche grundlegenden Eigenschaften des Daten-

modells wie Flexibilität und Reflexivität in jedem Fall bereits während der konzeptionellen Phase herausgearbeitet werden. Eine saubere Lösung muß daher vom Einzelfall abstrahieren und die erforderlichen Konzepte unabhängig von einer Programmiersprache oder einem Datenbankprodukt modellieren.

4.3 Dynamische Zuordnung von Attributen zu Objekten

4.3.1 Motivation

In den meisten objektorientierten Programmiersprachen ist ein Attribut fester Bestandteil einer Klasse. Instanzen der Klasse kapseln Attributwerte in Feldern definierter Größe, die in linearer Folge abgelegt sind. Der *Name* eines Attributs gibt implizit den Offset zum Zeiger auf die Instanz wieder. Durch Addition von Objektzeiger und Offset und anschließender Dereferenzierung kann der *Wert* angesprochen werden. Soll ein neues Attribut zu einer bestehenden Klasse hinzugefügt werden, muß die Klasse angepaßt und neu übersetzt werden, um dem veränderten Speicherbedarf einer Instanz zu bestimmen und um die neuen Zuordnungen von Attributnamen und Zeiger-Offsets zu errechnen. Normalerweise ändert ein neues Attribut auch die Schnittstelle der Klasse, so daß alle Applikationen, welche die Klasse benutzen, ebenfalls angepaßt werden müssen.

Die enge Bindung von Attributen an ein Objekt erschwert daneben jede Operation, die ausschließlich über den Attributen ausgeführt werden soll. So kann eine Suche in allen Texten des Datenbestandes nur über den Umweg des besitzenden Objekts ausgeführt werden, dessen Struktur der Anwendung bekannt sein muß. Auch die bloße Auflistung der Namen und Werte aller Attribute eines Objekts erfordert Kenntnisse über dessen Aufbau. Das Verlagern des Wissens über den Aufbau des Klassenschemas in die Anwendungen erschwert die Wartbarkeit des Schemas und macht Änderungen schwierig.

Eine *mehrsprachige* Abspeicherung von Attributwerten erfordert einen erhöhten Aufwand. Entweder wird ein Attribut mehrfach mit verschiedenen Namen in einem Objekt vorgesehen, oder jedes Objekt wird mehrfach erzeugt und mit den zur jeweiligen Sprache gehörenden Werten initialisiert. Gegen die erste Variante spricht, daß für viele Sprachen eine inflationär hohe Zahl von Attributen implementiert werden muß, wodurch

die Änderbarkeit des Klassenschemas sehr erschwert wird. Die zweite Variante erfordert, daß nicht nur alle Objekte, sondern auch alle Verknüpfungen zwischen den Objekten mehrfach erzeugt werden müssen. Weiterhin müssen alle Speicherzugriffsstrukturen der Datenbank, die dieses Objekt betreffen, mehrfach vorgesehen werden. Es liegt auf der Hand, daß diese Variante die Gefahr von Inkonsistenzen vergrößert. Beiden Varianten gemeinsam ist der Nachteil, daß durch das mehrfache Vorhandensein von Attributen oder Objekten noch mehr Wissen auf Seiten der Anwendungen verlagert werden muß.

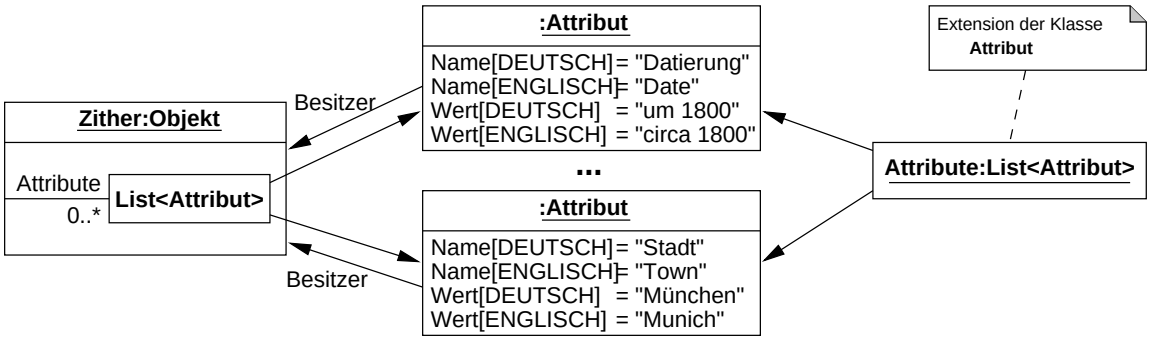

Abbildung 4.1: Beispiel dynamischer Zuordnung von Attributen

4.3.2 Modellierung der dynamischen Attributzuordnung

Die im letzten Unterabschnitt beschriebenen Nachteile der engen Bindung von Objekt und Attribut können behoben werden, indem jedes Attribut, das mehrsprachige Anwendungsdaten speichert, als ein eigenständiges Objekt aufgefaßt wird. Jedes Datenobjekt besitzt eine geordnete Kollektion, in der Referenzen auf alle seine Attribute gespeichert sind. Umgekehrt kennt jedes Attribut sein besitzendes Objekt. Abbildung 4.1 zeigt an einem Beispiel die Zuordnung von zwei Textattributen zu einem Objekt. In Abbildung 4.2 ist die abstrakte Implementierungsklasse `DatenBasis` dargestellt, die sämtliche Funktionalität bereitstellt, die zum Einfügen und Abrufen von Attributen benötigt wird. Jedes Objekt der Klasse `Attribut` speichert seinen Wert entweder mehrsprachig in Textform oder numerisch. Für die meisten Anwendungen dürften diese beiden Datentypen ausreichen. Wenn zusätzliche Datentypen (beispielsweise der Typ Datum) benötigt werden, bietet sich die in Abbildung 4.3 dargestellte alternative Modellierung der Klasse `Attribut` an. Die Klasse ist nun abstrakt und dient als Basisklasse für abgeleitete Attribut-Klassen, die jeweils einen Standard-Datentyp kapseln. Der Nachteil

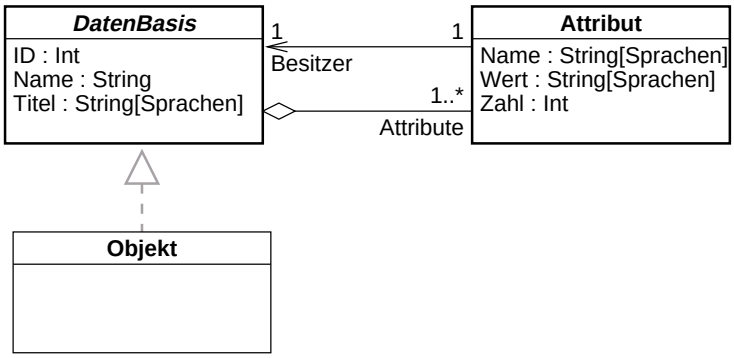

Abbildung 4.2: Auszug aus dem Datenbankschema: Modellierung der Klassen `DatenBasis` und `Attribut`

dieses Modells ist der aufwendigere Zugriff auf ein Attribut. Eine Query muß die genaue Implementierungsklasse jedes Attributs kennen oder abfragen, um *vor* dem Zugriff auf das jeweilige `Wert`-Feld eine geeignete Typumwandlung durchführen zu können. In der zuerst vorgestellten Variante bleibt ein nichtbenutztes `Wert`-Feld einfach leer und deutet damit gleichzeitig auf eine numerische Belegung im Feld `Zahl` hin. Eine Textsuche in einem Leerfeld ändert das Suchergebnis nicht, und eine Typumwandlung ist nicht mehr erforderlich.

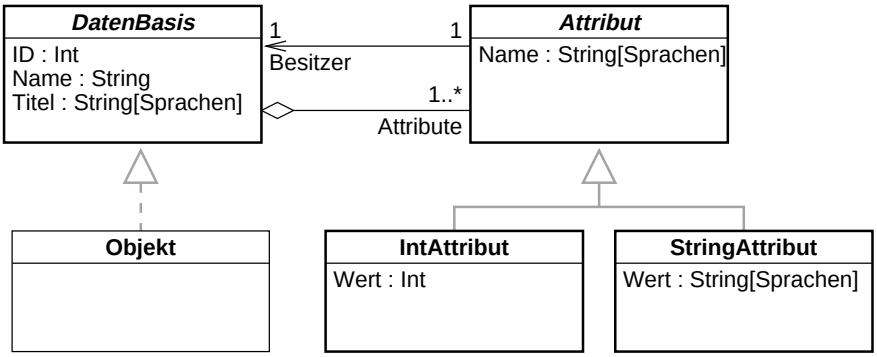

Abbildung 4.3: Alternative Modellierung der Klasse `Attribut`

Da alle Attribute in einer Kollektion des Objekts abgelegt sind, kann die Zuordnung von Attributen zu einem Objekt *zur Laufzeit* erfolgen. Auf ein bestimmtes Attribut eines Objekts kann wahlweise über seine Position in der Kollektion, seinen Namen oder seinen Wert zugegriffen werden. Wenn der Zugriff über die Position nicht benötigt wird, müssen Attribute mit NULL-Werten nicht in die Kollektion aufgenommen werden und benötigen damit auch keinen Speicherplatz. Zur Darstellung aller Attribute eines Ob-

jekts ist kein externes Wissen nötig, da jedes Attribut seinen Namen und seinen Wert in allen erforderlichen Sprachen kennt. Um alle Attribute anzuzeigen, muß lediglich die Kollektion `DatenBasis::Attribute` durchlaufen werden. Damit ist im Hinblick auf die Darstellung der Objektattribute eine vollständige Entkopplung von Daten und Anwendungen erreicht.

Die Speicherung der Namen in *jedem* Attribut ist redundant, da jedes Objekt gleichen Typs Attribute gleichen Namens besitzt. Es würde prinzipiell genügen, diese Namen nur einmal abzuspeichern und in jedem Objekt der Klasse `Attribut` einen Verweis auf die Namen zu verwalten. Im vorliegenden Schema wird auf diese Möglichkeit verzichtet, um einen möglichst einfachen und schnellen Zugriff auf die Attributnamen zu gewährleisten. Die entstehende Redundanz wird bewußt in Kauf genommen, da Attributnamen in der Regel kurz sind und somit der Mehrbedarf an Speicherplatz verhältnismäßig gering ist.

Die dynamische Attributzuordnung ist der erste Schritt auf dem Weg zur Bildung von extensionalen Klassen nach der Definition von J. Rumbaugh et al. Zu einer extensionalen Klasse gehören an dieser Stelle alle Objekte, die den gleichen Klassennamen tragen und die gleichen Attribute besitzen. Eine Erweiterung des Konzepts um generische Assoziationen wird in Abschnitt 4.4 besprochen; die dynamische Einordnung von Objekten in die Klassenhierarchie ist Gegenstand des Abschnitts 4.5.

4.3.3 Attribut-Extension

Wie Abbildung 4.1 zeigt, wird jedes Objekt der Implementierungsklasse `Attribut` sowohl vom Objekt, zu dem es gehört, als auch von der Kollektion `Attribute` referenziert. Diese Kollektion verwaltet die *Extension* der Klasse `Attribut`. Alle in der Datenbank gespeicherten Texte sind somit auf einheitliche Weise ansprechbar. Das bietet die Möglichkeit, ein Information-Retrieval-System an die Datenbank zu koppeln: Ein Textindex kann über der Extension aller Attribute erstellt werden; das besitzende Objekt wird über den Pfad `Attribut::Besitzer` erreicht. In Kapitel 7 wird gezeigt, wie diese Fähigkeit für die Ankopplung eines Information-Retrieval-Systems an die Datenbank genutzt werden kann.

Weiterhin kann die Attribut-Extension als alternative Möglichkeit für Queries über den Objekt-Attributen benutzt werden. Um beispielsweise alle Objekte der extensionalen Klassen Zither und Harfe zu erhalten, deren Saiten aus Messingdraht bestehen, können folgende alternative OQL-Queries formuliert werden:

1. ```
   select O from O in Objekte
     where (O.Name = "Zither" or O.Name = "Harfe")
       and exists A in O.Attribute:
         (A.Name[DEUTSCH] = "Saitenmaterial" and
          A.Wert[DEUTSCH] = "Messingdraht")
   ```

2. ```
   select A.Besitzer from A in Attribute
     where (A.Besitzer.Name = "Zither" or A.Besitzer.Name = "Harfe")
       and A.Name[DEUTSCH] = "Saitenmaterial"
       and A.Wert[DEUTSCH] = "Messingdraht"
   ```

Durch die Speicherung der Attributnamen in Textform sind auch solche Anfragen möglich, bei denen der Benutzer nur weiß, daß z.B. alle Saiteninstrumente eine Eigenschaft "Saitenmaterial" haben, er aber die Klassenhierarchie nicht kennt oder sich nicht mit komplexen Anfragen beschäftigen will. Die Query

```
select A.Besitzer from A in Attribute
  where A.Name[DEUTSCH] = "Saitenmaterial"
```

listet alle Instrumente mit der betreffenden Eigenschaft auf.

4.4 Generische Verknüpfungen und Assoziationen

4.4.1 Motivation

Die meisten objektorientierten Programmiersprachen und Datenbanksysteme sehen kein explizites Assoziationskonzept vor. Normalerweise sind nur die Basiskonstrukte *Zeiger* und/oder *Referenz* vorhanden, auf welche die Assoziationen des Entwurfsdiagramms abgebildet werden. Assoziationen, die durch Attribute qualifiziert sind, werden durch *Assoziationsklassen* implementiert. Der ODMG-Standard erweitert die primitiven Konstrukte durch den Begriff der *inversen Referenz*. Hier muß der Programmierer die Verknüpfung nur in einer Richtung herstellen, während das DBMS die Gegenrichtung automatisch hinzufügt. Auch der ODMG-Standard unterstützt Kardinalitätsangaben nur in eingeschränkter Weise. Durch einen Datenbankzeiger wird die Kardinalität auf der Zielseite

der Verknüpfung auf das Intervall $[0, 1]$ eingeschränkt. Mit Hilfe einer Kollektion von Datenbankzeigern läßt sich die Kardinalität $[0, \infty]$ modellieren. Alle anderen Kardinalitätsangaben des Entwurfs müssen von der Anwendungslogik überprüft und gesichert werden.

Ein Datenbankzeiger ist, genau wie jedes andere Attribut, fester Bestandteil eines Objekts. Damit ist bereits zur Übersetzungzeit festgelegt, zu welchen Klassen die beteiligten Objekte einer Verknüpfung gehören müssen und welche Eigenschaften eine Verknüpfung hat. Lediglich die Verknüpfung selbst wird zur Laufzeit hergestellt. Mit anderen Worten, alle Assoziation werden statisch definiert, während ihre Instanzen, die Verknüpfungen, dynamisch erstellt werden. Damit gilt auch hier, daß jede Anwendung, die das Datenbankschema benutzt, Wissen über die Assoziationen einer Klasse benötigt, um deren Objekte und die Verknüpfungen zwischen den Objekten darstellen zu können. Wird eine neue Assoziation zum Datenbankschema hinzugefügt, so ändert sich in der Regel auch die Schnittstelle der Klasse, so daß alle Applikationen, welche die Klasse benutzen, angepaßt werden müssen.

Um eine Verknüpfung korrekt darzustellen, wird neben den Namen der beteiligten Objekte auch der Name der Assoziation benötigt. In einem statischen Schema müssen die Applikationen die Namen jeder Verknüpfung, eventuell in mehreren Sprachen, kennen. Zusätzliches externes Wissen wird benötigt, wenn die Assoziation durch Attribute, die zu einer Assoziationsklasse gehören, qualifiziert ist. Für Attribute von Assoziationsklassen gelten alle in Unterabschnitt 4.3.1 besprochenen Einschränkungen.

4.4.2 Modellierung von generischen Verknüpfungen und Assoziationen

Um die im letzten Unterabschnitt beschriebenen Nachteile statischer Assoziationen aufzuheben, wird das Klassenschema um die Implementierungsklasse `Verknüpfung` ergänzt, die folgende Eigenschaften besitzt:

- Sie ist von `DatenBasis` abgeleitet und kann daher mehrsprachige Attribute besitzen.
- Sie besitzt einen *Namen*. Dieser Name entspricht der Benennung einer Assoziation in einem statischen Klassenschema.

- Sie speichert den Titel der Assoziation in mehreren Sprachen. Der Titel wird ausschließlich zur Darstellung der Verknüpfung benötigt. Er kann aus mehreren Worten bestehen und muß nicht notwendigerweise mit dem Namen übereinstimmen.
- Sie implementiert einen Datenbankzeiger auf die Zielklasse.
- Sie enthält eine optionale Referenz auf ein weiteres Objekt der Klasse `Verknüpfung`, das die *inverse Verknüpfung* darstellt.
- Sie besitzt einen Assoziationstyp, der die Werte `ASSOZIATION` und `AGGREGATION` annehmen kann.

Als Quell- und Zielklasse für alle Verknüpfungen dient die abstrakte Implementierungsklasse `ObjektBasis`. Diese Klasse verwaltet außerdem alle Komponentenobjekte in der Kollektion `Komponenten`. Abbildung 4.4 zeigt das erweiterte Klassendiagramm.

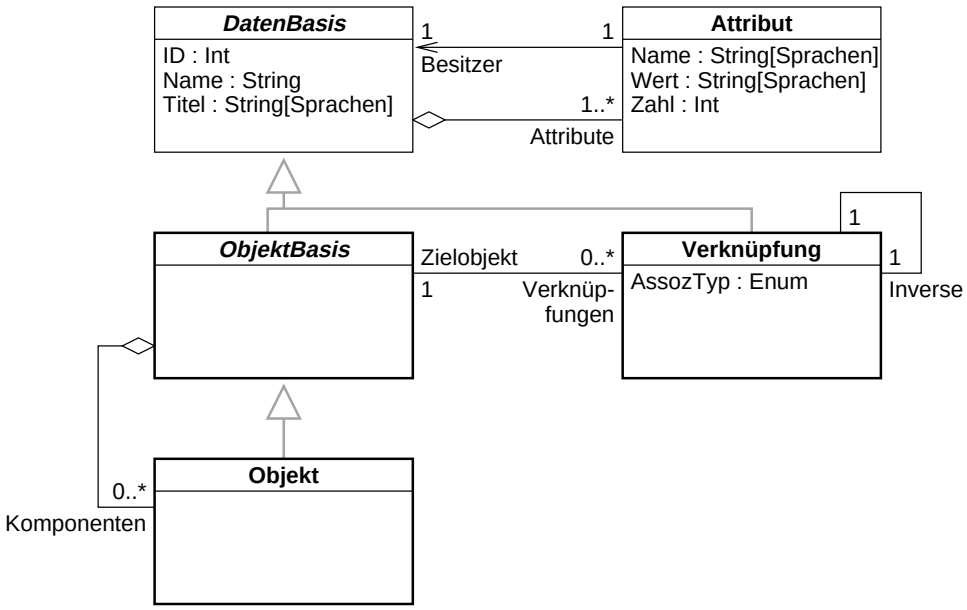

Abbildung 4.4: Auszug aus dem Datenbankschema: Modellierung der Klassen `ObjektBasis` und `Verknüpfung`

Durch die generische Modellierung von Verknüpfungen kann der Begriff der extensionalen Klasse dahingehend erweitert werden, daß Objekte dann zur selben Klasse gehören, wenn sie neben dem gleichen Namen und den gleichen Attributen auch die gleichen Verknüpfungen besitzen. Gleiche Verknüpfungen werden zu *extensionalen Assoziationen* zusammengefaßt. Instanzen der Implementierungsklasse `Verknüpfung` gehören dann zur selben Assoziation, wenn sie

- denselben Assoziationsnamen tragen,
- Attribute gleichen Namens und Typs in gleicher Anzahl besitzen,
- von Objekten ausgehen, die derselben extensionalen Klasse oder Unterklasse angehören,
- auf Objekte zeigen, die zur selben extensionalen Klasse oder Unterklasse gehören.

4.4.3 Realisierung des Assoziationskonzepts an Beispielen

Im letzten Unterabschnitt wurde ein Konzept vorgestellt, mit dem Verknüpfungen und extensionale Assoziationen in generischer Weise modelliert werden können. Nun soll an Beispielen untersucht werden, ob und in welcher Weise sich die in Unterabschnitt 4.2.1 besprochenen Assoziationskonzepte der objektorientierten Modellierung dynamisch nachbilden lassen.

Unidirektionale Verknüpfung

Eine unidirektionale Verknüpfung kann nur in einer Richtung durchlaufen werden. Ein Objekt, welches das Ende einer Verknüpfung darstellt, kennt die Verknüpfung nicht.

Abbildung 4.5 zeigt, wie zwei Objekte der extensionalen Klassen Zither und Meister durch eine Verknüpfung der extensionalen Assoziation ErbautVon verbunden werden.

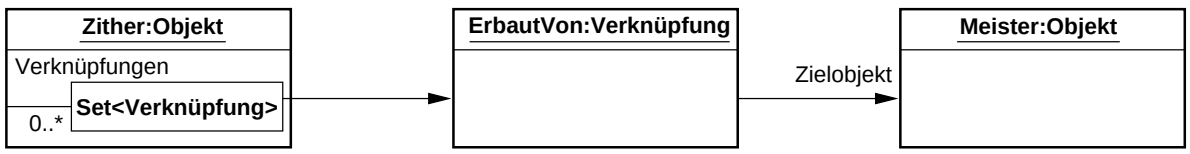

Abbildung 4.5: Objektdiagramm einer unidirektionalen Verknüpfung

Bidirektionale Verknüpfung

Eine bidirektionale Verknüpfung kann als zwei komplementäre unidirektionale Verknüpfungen aufgefaßt werden, die unterschiedliche Namen besitzen *können*. Modelliert wird dies, indem zum bisher eingeführten Verknüpfungsobjekt, das die Beziehung in einer

Richtung herstellt, ein *komplementäres* Verknüpfungsobjekt bereitgestellt wird. Jede Teilverknüpfung speichert einen Verweis auf ihr Komplement.

Abbildung 4.6 erweitert die unidirektionale Verknüpfung aus Abbildung 4.5 um eine inverse Verknüpfung der extensionalen Assoziation Erbaute.

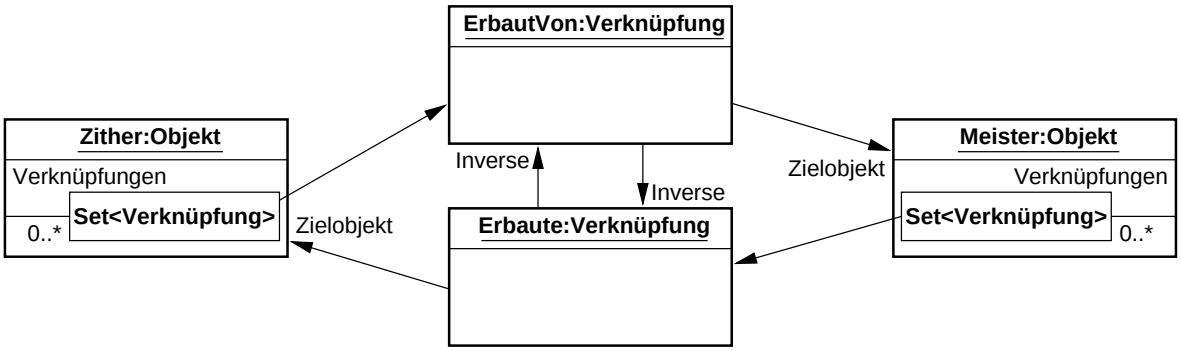

Abbildung 4.6: Objektdiagramm einer bidirektionalen Verknüpfung

Aggregation

Eine Aggregation entspricht einer Assoziation, jedoch mit der zusätzlichen Eigenschaft, daß die Teile vom Ganzen abhängig sind. Dabei werden keine Einschränkungen hinsichtlich der Lebensdauer gemacht. Die Instanz einer Aggregation wird genau wie eine unidirektionale Verknüpfung modelliert. Wichtig ist hier, daß es *keine* inverse Verknüpfung geben darf, da sonst die Ganzes-Teile-Hierarchie durchbrochen wird.

Komposition

Die Komposition entspricht aus Sicht der Implementierung der Einbettung des Komponentenobjekts in das Aggregatobjekt. Eine Komponente kann nur Teil eines Aggregats sein. Im vorgestellten Schema wird die Komposition durch einen Zeiger vom Aggregat- auf das Komponentenobjekt modelliert. Der *Rollenname* einer Komposition wird im Attribut `Objekt::Name` auf der Seite der Komponente gespeichert. Damit ist der Name der extensionalen Klasse des Komponentenobjekts nicht mehr ersichtlich. Allerdings wird der Klassenname nicht mehr benötigt, da Komponentenobjekte keine Identität haben und nicht in die Extension ihrer Klasse aufgenommen werden [STS97, S. 113]. Abbildung 4.7

zeigt ein Beispiel einer Komposition: Ein Objekt der Klasse Orgel besitzt neben einem Attribut drei Komponenten, die jeweils wieder Attribute haben.

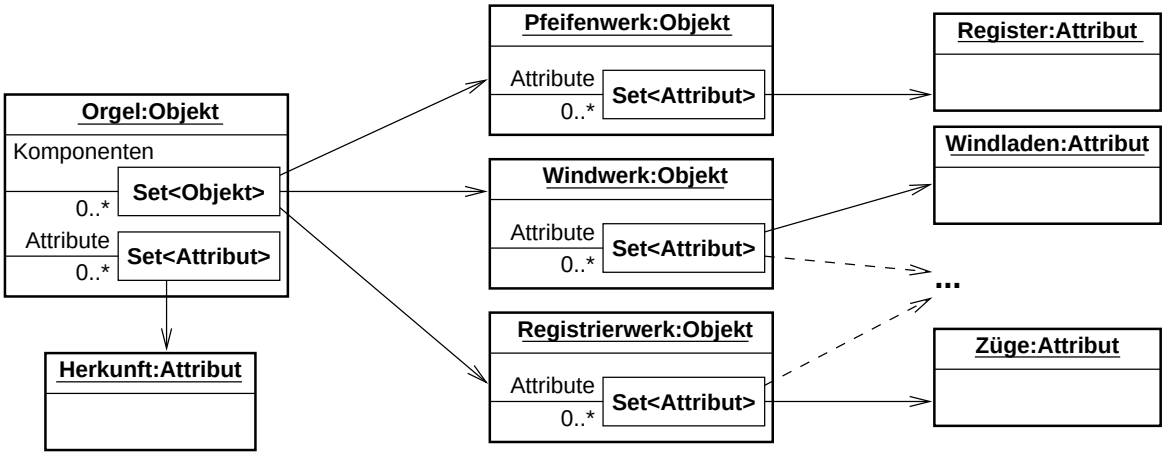

Abbildung 4.7: Objektdiagramm einer Komposition

Kardinalität

Jede Assoziation besitzt eine Kardinalität Q auf der Quellseite sowie eine Kardinalität Z auf der Zielseite. Die Kardinalitäten müssen im Bereich $Q = [q_{min}, q_{max}]$ bzw. $Z = [z_{min}, z_{max}]$ liegen. Durch die Intervalle $Q = [0, 1]$ und $Z = [0, 1]$ wird eine 1:1-Relation beschrieben, während die Intervalle $Q = [0, \infty]$ und $Z = [0, \infty]$ den Spezialfall einer n:m-Relation definieren [Vos94, S. 68]. Die Vorgaben für die Ziel- und die Quellkardinalität einer extensionalen Assoziation $\mathcal{A}$ zwischen den extensionalen Klassen $\mathcal{K}_1$ und $\mathcal{K}_2$ werden auf folgende Weise geprüft:

Zielkardinalität: Jedes Objekt $o_1 \in \mathcal{K}_1$ muß in seiner Kollektion `Verknüpfungen` mindestens z_{min} und darf höchstens z_{max} Instanzen der Klasse `Verknüpfung` speichern, die zur extensionalen Assoziation $\mathcal{A}$ gehören.

Quellkardinalität: Sei $o_2 \in \mathcal{K}_2$ ein Objekt, auf das eine Verknüpfung zeigt, die von einem Objekt $o_1 \in \mathcal{K}_1$ ausgeht und zur Assoziation Typ $\mathcal{A}$ gehört. Dann dürfen auf das Objekt insgesamt nicht weniger als q_{min} und nicht mehr als q_{max} Verknüpfungen der Assoziation $\mathcal{A}$ zeigen, welche von allen Objekten der Klasse $\mathcal{K}_1$ ausgehen.

Inverse Assoziationen können als zwei getrennte unidirektionale Assoziationen aufgefaßt werden, welche die Zusatzbedingung erfüllen müssen, daß die Quellkardinalität jeder Assoziation der Zielkardinalität der jeweils inversen Assoziation entsprechen muß.

Die Kardinalitäten einer extensionalen Assoziation werden durch das Datenschema festgelegt, das im Abschnitt 4.7 besprochen wird. In Abschnitt 5.5 wird erläutert, wie die Überprüfung der Kardinalitäten durch den Datenlader vorgenommen wird.

4.5 Die Rolle der Kategorie

4.5.1 Motivation

Das bisher entwickelte Schema ermöglicht die Selbstdarstellung jedes Objekts dadurch, daß es seine Attributnamen und -werte, seine Verknüpfungen mit anderen Objekten sowie den Namen seiner extensionalen Klasse kennt. Das Objekt besitzt jedoch kein Wissen darüber, an welcher Position in der Klassenhierarchie sich seine extensionale Klasse befindet und welches die über- und untergeordneten Klassen sind. Nützlich ist die Kenntnis der Klassenhierarchie für die Darstellung: Der Benutzer kann ein wesentlich klareres Bild über die Bedeutung des Objekts gewinnen, wenn er weiß, auf welcher Ebene der Hierarchie es sich befindet. Zugleich ist es in vielen Fällen sinnvoll, wenn nicht nur Informationen über ein *bestimmtes* Objekt vorliegen, sondern Angaben über *alle* Objekte einer Klasse abgerufen werden können. Um dieses Ziel zu erreichen, wird das neue Schemaelement *Kategorie* benötigt, das eine extensionale Klasse repräsentiert und das gleichzeitig alle Eigenschaften eines Datenobjekts besitzt.

4.5.2 Modellierung von Kategorien

Eine Kategorie wird wie jedes andere Datenobjekt zur Darstellung benutzt und muß daher Attribute und Verknüpfungen besitzen können. Aus diesem Grund wird die neue Implementierungsklasse `Kategorie` von der abstrakten Klasse `ObjektBasis` abgeleitet. Die Instanzen der Implementierungsklasse `Kategorie` unterliegen einer Kardinalitätseinschränkung: Zu jeder extensionalen Klasse muß es genau ein Kategorieobjekt geben, das *vor* allen Objekten dieser Klasse erzeugt wird. Wie Abbildung 4.8 zeigt,

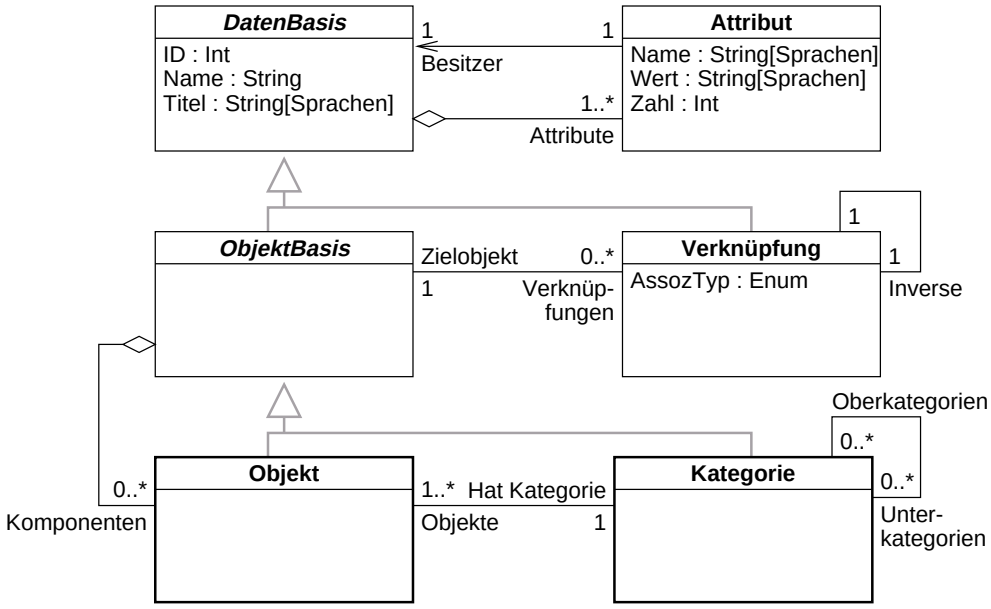

Abbildung 4.8: Auszug aus dem Datenbankschema: Modellierung der Klassen `Objekt` und `Kategorie`

sind die Implementierungsklassen `Objekt` und `Kategorie` durch eine bidirektionale Assoziation verbunden, die jedes Objekt mit einem Verweis auf seine Kategorie versorgt und gleichzeitig jeder Kategorie die Menge aller Objekte der zugehörigen extensionalen Klasse zuordnet. Beide Gesichtspunkte werden in den folgenden Unterabschnitten näher betrachtet.

4.5.3 Darstellung einer Klasse

Kategorieobjekte besitzen alle Eigenschaften von Datenobjekten und abstrahieren gleichzeitig vom Einzelobjekt der zugehörigen extensionalen Klasse. So wie eine Klasse die strukturellen und operationalen Gemeinsamkeiten ihrer Instanzen zusammenfaßt, so verallgemeinert ein Kategorieobjekt die *inhaltlichen* Eigenschaften der Objekte. Ein Kategorieobjekt kann zusammenfassende Angaben über die zugehörigen Objekte speichern und dadurch helfen, die Bedeutung des Einzelobjekts besser einzuordnen. Daneben können Kategorieobjekte Redundanzen vermeiden, indem sie allgemeine Informationen verwalten, die anderenfalls jedem einzelnen Objekt zugeordnet werden müßten. Zwei Beispiele sollen diese Möglichkeiten verdeutlichen:

In einem wissenschaftlichen Katalog für Musikinstrumente werden genaue Angaben über jedes einzelne Instrument verwaltet. Jede Gruppe von Instrumenten hat gemeinsame Konstruktionsmerkmale und physikalische Eigenschaften. So ist bei allen Violinen eine ähnliche Schallausbreitung festzustellen, da der grundlegende Aufbau des Resonanzkörpers gleich ist.

In einem Informationssystem über Datenbankprodukte sind detaillierte Informationen zu jedem einzelnen Produkt gespeichert. Fachlich weniger versierte Benutzer können sich daneben über die grundlegenden Eigenschaften von relationalen und hierarchischen DBMS informieren oder die Merkmale von Objekt- und Pageservern auf der Ebene der objektorientierten DBMS abrufen.

Jedes Kategorieobjekt speichert Verweise auf seine übergeordneten (allgemeineren) und untergeordneten (spezielleren) Kategorien. Eine Applikation kann dadurch leicht den Vererbungsbaum zeichnen: Bei den Kategorien der obersten Ebene beginnend, wird die Menge der `Unterkategorien` rekursiv durchlaufen. Klassen, die mehrere Oberklassen besitzen, erscheinen mehrfach im Baum. Durch die Implementierungsklasse `Kategorie` ist die Klassenhierarchie in der Lage, sich selbst darzustellen, und kann als Ausgangspunkt für die Navigation im Datenbestand oder als Startpunkt für die Suche in einer Teilmenge der Daten dienen.

Jedes Objekt hat über die Assoziation `Objekt::HatKategorie` unmittelbaren Zugriff auf das zu seiner extensionalen Klasse gehörende Kategorieobjekt. Dadurch ist es in der Lage, seine Position in der Klassenhierarchie zu ermitteln.

4.5.4 Extension einer Klasse

Nach [STS97, S. 125] versteht man unter einer *flachen Extension* einer Klasse genau die Menge der Objekte, die dieser Klasse direkt zugeordnet sind, also nicht Instanzen von deren Unterklassen sind. Die *tiefe* Extension einer Klasse umfaßt die Objekte der Klasse und aller Unterklassen und stellt somit eine Obermenge der flachen Extension dar.

Die inverse Richtung der Assoziation `Objekt::HatKategorie` wird durch die Assoziation `Kategorie::Objekte` modelliert. Jedes Objekt der Klasse `Kategorie` speichert die

Extension seiner Klasse. Die Entscheidung darüber, ob eine Kategorie die flache oder die tiefe Extension ihrer Klasse verwalten soll, erfordert ein Abwägen zwischen Speicherbedarf und Performance. Falls nur die flache Extension je Kategorie vorliegt, jedoch die tiefe Extension benötigt wird, muß die Vereinigungsmenge aller Objekte zur Laufzeit ermittelt werden. Das geschieht durch rekursives Traversieren aller Unterkategorien. Der entgegengesetzte Fall liegt vor, wenn jeder Kategorie die tiefe Extension ihrer Klasse zugeordnet wird. Der Zeitbedarf für eine Operation auf der tiefen Extension ist proportional dem Zeitbedarf für den Zugriff auf die Kollektion `Kategorie::Objekte`. Der Preis für den schnellen Zugriff ist eine hohe Redundanz bei der Speicherung der Objektreferenzen. Angenommen, die Objekte des Datenbestandes sind gleichmäßig auf K Kategorien aufgeteilt und jede Kategorie habe M Unterkategorien. Weiterhin gebe es keine Mehrfachvererbung. Dann beträgt die Tiefe des Kategoriebaums $T = \log_M (K(M-1)+1)$. Also befindet sich jedes Objekt durchschnittlich in $R = \log_M (K(M-1)/2+1)$ Kategorien. Eine Erhöhung der Anzahl von Kategorien K bei gleichbleibender Anzahl der Objekte führt zu einem lediglich logarithmischen Anstieg der Redundanz R. Betrachtet man weiterhin die Tatsache, daß Objektreferenzen durch physische oder logische Zeiger implementiert werden, deren Speicherbedarf gering ist, so kann man die Redundanz zugunsten einer guten Performance in den meisten Fällen in Kauf nehmen.

4.6 Multimediale Daten

Die Verwaltung von multimedialen Daten, insbesondere von Tondokumenten, ist sehr aufwendig, wenn die Datenbank den speziellen Anforderungen für die Wiedergabe dieser Daten genügen soll. Dazu gehören der blockweise oder kontinuierliche Abruf der Daten innerhalb fester Zeiteinheiten, synchrone Wiedergabe mehrerer zusammengehöriger Dokumente und der wahlfreie Abruf von Teildokumenten. Diese Funktionen können von einer Anwendung allerdings nur dann genutzt werden, wenn alle Ebenen zwischen Datenbank und Ausgabemedium ebenfalls in der Lage sind, zeitkritische Daten zu handhaben. Sobald eine Schicht oder eine Schnittstelle zwischen den Schichten die synchrone oder kontinuierliche Datenübertragung nicht garantieren kann, sind die diesbezüglichen Datenbankfähigkeiten nutzlos. Diese Einschränkungen treffen auf alle Netzwerke zu, deren Übertragungseinheit Pakete variabler Länge sind. Einer zusätzlichen Einschränkung unterliegt das HTTP-Protokoll, das als schwächstes Glied eines Informationssystems im WWW angesehen werden muß. Es handelt sich um ein zustandsloses Protokoll. Jede

Einzelübertragung erfolgt atomar und hat keinen Bezug zur vorangegangenen Transaktion. Weiterhin ist es nicht möglich, eine maximale Übertragungszeit der Daten zu garantieren. Selbst unmittelbar aufeinanderfolgende Zugriffe können über unterschiedliche Pfade geroutet werden, so daß auch der Zeitversatz zwischen einzelnen Anfragen stark variieren kann. Als Ausweg bietet sich an, zeitkritische Daten stets in ihrer Gesamtheit zu übertragen. Die Funktionalität, die zur Wiedergabe der Daten benötigt wird, muß von einer Anwendung bereitgestellt werden, die sich jenseits der schwächsten Stelle der Übertragungskette befindet. Im Fall einer WWW-Anbindung ist dies der Browser, der die Daten entweder selbst interpretiert oder ein externes Programm zur Wiedergabe heranzieht.

Prinzipiell sind nur drei Informationen nötig, um Daten beliebigen Typs zu speichern und wiederzugeben:

1. Die Binärdaten selbst,
2. die Größe und
3. der Typ der Daten.

Der Browser ermittelt anhand des HTTP-Headers des empfangenen Dokuments den Typ der nachfolgenden Daten und wählt anhand einer Zuordnungstabelle das geeignete Ausgabegerät. Der Dokumententyp wird im MIME[1]-Format im HTTP-Header kodiert. Beispielsweise erkennt der Browser ein JPEG-Bild an folgendem Eintrag:

```
Content-type: image/jpeg
```

Diese Art der Zuordnung durch Tabellen ist nicht nur auf Informationssysteme im WWW beschränkt, sondern ist für beliebige Plattformen realisierbar, sofern dort Programme zur Wiedergabe von Standard-Datenformaten existieren.

Binärdaten können in DBMS entweder in *Binary Large Objects (BLOBs)* gespeichert oder alternativ in externen Dateien abgelegt werden. Im letzteren Fall verwaltet das DBMS lediglich den Namen der Datei, gegebenenfalls ergänzt um weitere Angaben. Das DBMS kann keine Garantie für die Konsistenz der multimedialen Daten geben, da es keine Handhabe gegen das versehentliche oder absichtliche Löschen bzw. Ändern externer Dateien hat. Im Gegensatz dazu stehen multimediale Daten, die in BLOBs

[1] Multipurpose Internet Mail Extensions

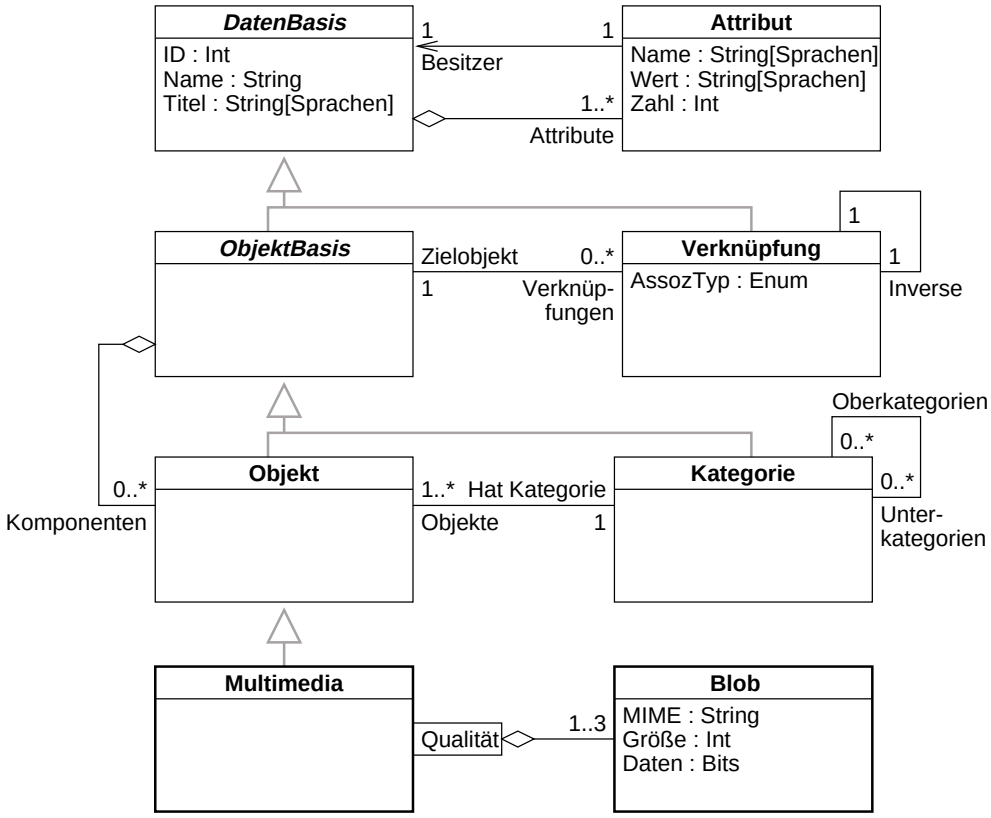

Abbildung 4.9: Auszug aus dem Datenbankschema: Modellierung der Klassen `Multimedia` und `Blob`

abgelegt sind, vollständig unter der Kontrolle des DBMS. Das System garantiert die Existenz und Korrektheit der Daten, gewährleistet einen effizienten Abruf sowie die Recovery im Fehlerfall.

Ein Objekt der Implementierungsklasse `Multimedia` kann maximal drei BLOBs besitzen (Abbildung 4.9). Jedes dieser BLOBs speichert das inhaltlich gleiche Bild- oder Tondokument in jeweils anderer Qualität und/oder einem anderen Datenformat. Daten mit geringerer Qualität können kompakter kodiert werden und eignen sich auch für Übertragungsmedien mit geringer Bandbreite. Durch mehrere angebotene Datenformate erhöht sich die Wahrscheinlichkeit, daß es für mindestens eines der Formate ein geeignetes Wiedergabegerät auf der Anwenderseite gibt. Auf der Grundlage der Klasse `Multimedia` können speziellere extensionale Klassen wie Foto oder Klang modelliert werden, die neben den BLOBs Attribute und Verknüpfungen besitzen können oder ihrerseits Komponenten anderer extensionaler Klassen sein können. Abbildung 4.10 zeigt an

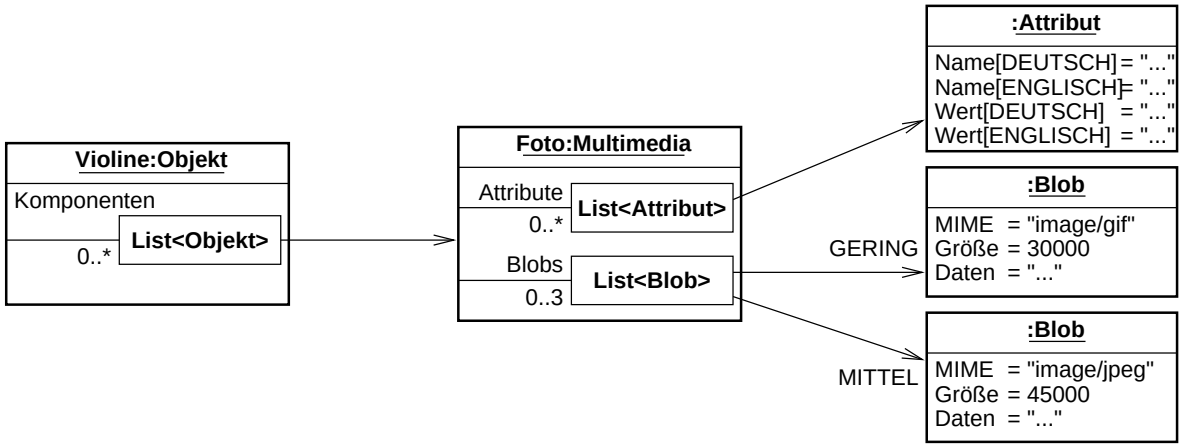

Abbildung 4.10: Beispiel für die Modellierung multimedialer Daten

einem Beispiel die Zuordnung eines Fotos zu einem Objekt durch Komposition. Damit das Foto als Objekt mit eigener Identität in seine Extension aufgenommen wird, bietet sich die alternative Zuordnung zum Objekt über eine Aggregation an.

4.7 Datenschema und Metaschema

4.7.1 Motivation

Bisher wurde bei der Vorstellung der einzelnen Modellierungskonzepte stets von einer *existierenden* Menge von Objekten und Verknüpfungen ausgegangen. Objekte mit gleichen Attributen, Verknüpfungen und Komponenten sowie der gleichen Position in der Klassenhierarchie wurden zu extensionalen Klassen zusammengefaßt. Bereits durch die bisher eingeführten Konzepte besitzt jeder Datenbestand vollständiges Wissen über die eigene Struktur und ist in der Lage, sich selbst darzustellen. Allerdings gibt es noch keine Instanz, die *vorschreibt*, welchen Aufbau die Objekte des Datenbestandes haben dürfen. Das Wissen über die *zulässige* Struktur der Daten wird für den Datenimport benötigt: Die betreffende Applikation muß garantieren können, daß die geladenen Daten gewissen Vorgaben genügen. Diese Vorgaben werden von einem *Datenschema* gemacht. Es legt den strukturellen und semantischen Rahmen fest, innerhalb dessen sich die Anwendungsdaten bewegen müssen. Durch die Einführung eines Datenschemas wird der Übergang von der extensionalen zur intensionalen Sicht auf die Datenstruktur vollzogen.

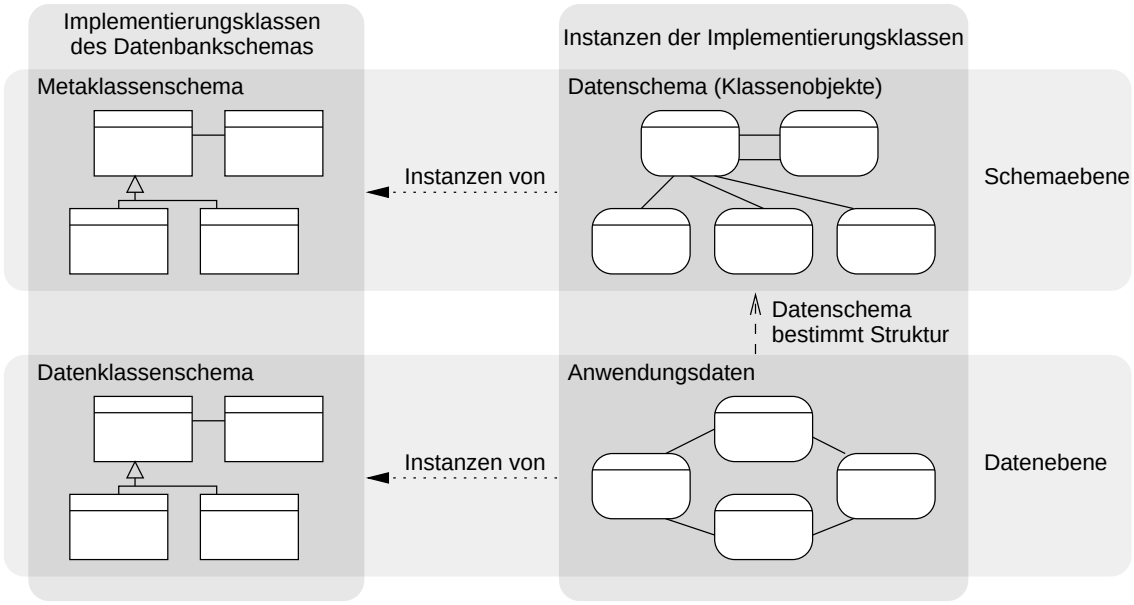

Abbildung 4.11: Zusammenhang zwischen Metaschema, Datenschema und Anwendungsdaten

Das Datenbankschema sollte nicht nur ein einziges, fest vorgeschriebenes Datenschema zulassen. Das Datenschema sollte vielmehr innerhalb bestimmter Grenzen frei definierbar und an veränderte Gegebenheiten anpaßbar sein. Um die gewünschte Flexibilität des Datenschemas zu erreichen, muß dessen Aufbau durch ein *Metaschema* gesteuert werden.

4.7.2 Realisierung von Metaschema und Datenschema

Die in Abbildung 4.9 dargestellten Implementierungsklassen, deren Instanzen die Anwendungsdaten aufnehmen, werden im folgenden unter dem Begriff *Datenklassenschema* zusammengefaßt. Das Datenklassenschema und die Anwendungsdaten bilden die *Datenebene*. Der Datenebene wird nun eine *Schemaebene* hinzugefügt (Abbildung 4.11), deren Aufgabe es ist, den Aufbau der Objekte der extensionalen Klassen und Assoziationen der Datenebene zu steuern. Die Schemaebene läßt sich wiederum in Metaklassenschema und Datenschema untergliedern. Das Metaklassenschema setzt sich aus den *Metaklassen* zusammen, bei denen es sich um Implementierungsklassen handelt. Die Instanzen der Metaklassen werden *Klassenobjekte* genannt. Jedes Klassenobjekt bestimmt den Aufbau genau einer extensionalen Klasse der Datenebene sowie ihrer Assoziationen mit anderen Klassen. Gleichzeitig definiert das Klassenobjekt den Aufbau des zugehörigen Katego-

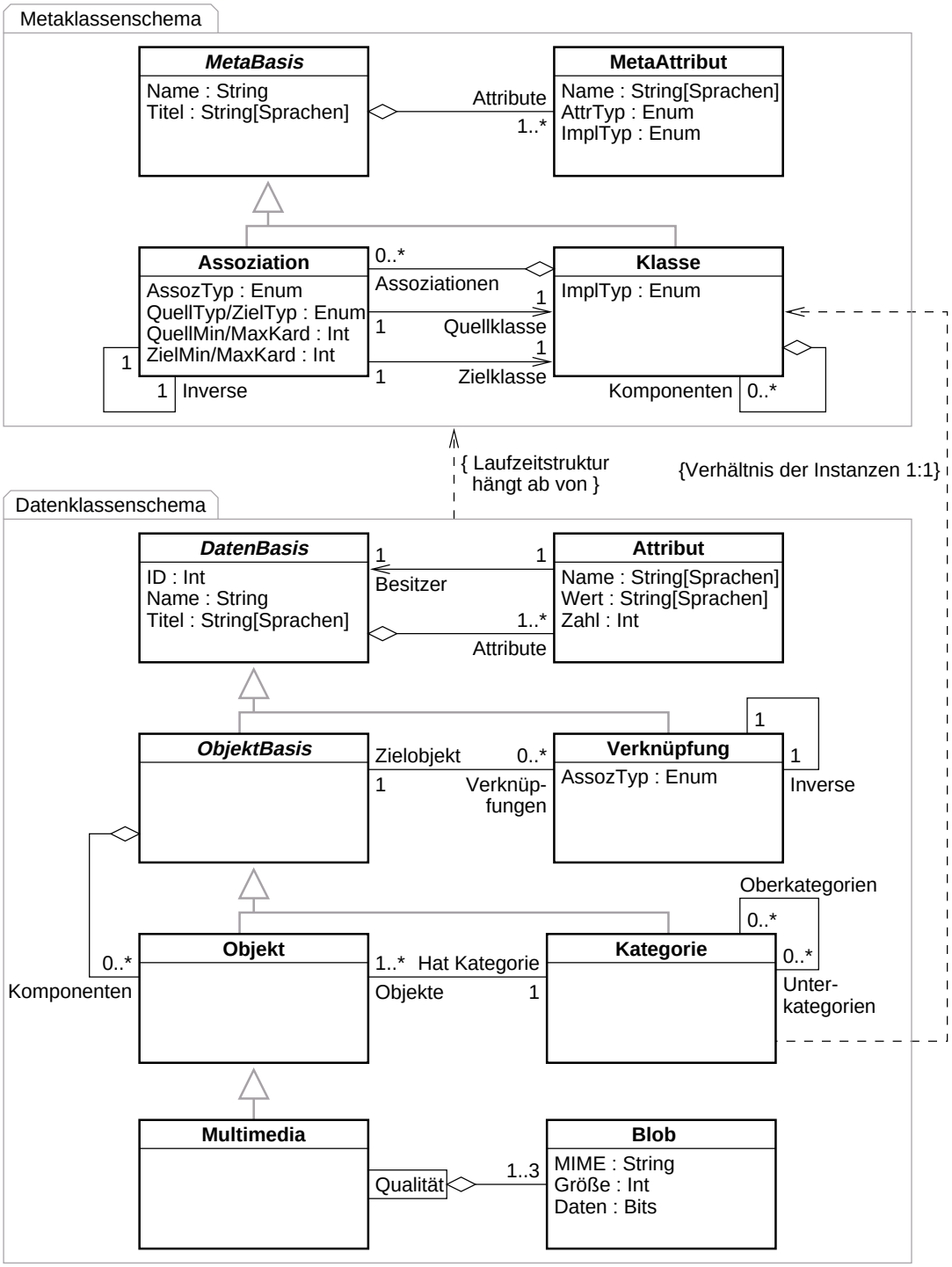

Abbildung 4.12: Vollständiges Datenbankschema mit Datenklassenschema und Metaklassenschema. Nicht dargestellt sind die Methoden der Klassen (vgl. die Anhänge D.3.2 und D.3.3).

rieobjekts (Unterabschnitt 4.7.3). Aus diesem Grund besitzen Datenklassenschema und Metaklassenschema einen ähnlichen Aufbau, wie Abbildung 4.12 zeigt. Jede Instanz der Implementierungsklasse `Klasse` verwaltet drei Kollektionen, deren Aufgaben nachfolgend zusammengestellt sind:

`Attribute`: Jedes Element dieser Kollektion ist eine Instanz der Klasse `MetaAttribut` und definiert Name und Datentyp eines Attributs der zugehörigen extensionalen Klasse. Die Reihenfolge der Kollektionseinträge bestimmt die Reihenfolge der Attribute in den Objekten der extensionalen Klasse. Die korrespondierende Kollektion im Datenklassenschema ist `DatenBasis::Attribute`.

`Komponenten`: Jeder Eintrag in dieser Kollektion bestimmt den Aufbau einer extensionalen Komponentenklasse. Komponentenklassen können wiederum Attribute, Komponenten und Assoziationen besitzen. Hier heißt das Pendant im Datenklassenschema `ObjektBasis::Komponenten`.

`Assoziationen`: Ein Element dieser Kollektion definiert eine extensionale Assoziation, die bestimmt wird durch Name, Semantik (Assoziation oder Aggregation), Quell- und Zielklasse (sowie deren Unterklassen), Richtung (uni- oder bidirektional) sowie minimale und maximale Kardinalität auf Quell- und Zielseite. Die entsprechende Kollektion im Datenklassenschema ist `ObjektBasis::Verknüpfungen`.

Ein Objekt der Datenebene wird also durch zwei Klassen bestimmt. Es ist einerseits eine "reale" Instanz einer Implementierungsklasse des Datenklassenschemas. Diese Klasse besitzt Methoden zum Erzeugen des Objekts, implementiert die Kollektionen, welche die Attribute, Komponenten und Verknüpfungen aufnehmen, und stellt eine Methodenschnittstelle zur Manipulation der Kollektionseinträge bereit. Andererseits kann ein Datenobjekt auch als "virtuelle" Instanz eines Klassenobjekts der Schemaebene aufgefaßt werden. Das Klassenobjekt definiert den Namen, die Art und Anzahl der Attribute des Datenobjekts, legt dessen Einordnung in die Klassenhierarchie fest und bestimmt Struktur und Semantik der Verknüpfungen mit anderen Datenobjekten.

Weil Meta-Attribute und Assoziationen der Schemaebene eigenständige Objekte sind, ähnlich den Attributen und Verknüpfungen der Datenebene, besitzt jedes Datenschema den gleichen Grad an Flexibilität und Reflexivität wie die Anwendungsdaten. Das bedeutet insbesondere, daß ein Datenschema auf der Grundlage des Metaklassenschemas dynamisch erzeugt und verändert werden kann.

4.7.3 Dualität von Klassenobjekten und Kategorien

Die Implementierungsklasse `Kategorie` spielt eine besondere Rolle im entworfenen Schema. Einerseits gehört sie zum Datenklassenschema und dient der Aufnahme von Anwendungsdaten, die den Kategorieobjekten zu einem beliebigen Zeitpunkt *nach* der Schemadefinition zugewiesen werden können. Andererseits können Kategorien dem Datenschema zugeordnet werden. Jede Kategorie wird sofort nach dem zugehörigen Klassenobjekt erzeugt, so daß sichergestellt ist, daß es zu jedem Klassenobjekt genau ein Kategorieobjekt gibt. Anders als eine Kategorie ist ein Klassenobjekt für den Anwender nicht sichtbar. Es dient lediglich dazu, Struktur und Semantik der Datenobjekte seiner extensionalen Klasse festzulegen. Zusätzlich bestimmt es den Aufbau *seiner* Kategorie. Dazu dienen das Attribut `ImplTyp` der Klasse `MetaAttribut` sowie die Attribute `QuellTyp` und `ZielTyp` der Klasse `Assoziation`, die die Werte `OBJEKT` und `KATEGORIE` annehmen können. Wenn ein Meta-Attribut vom Typ `KATEGORIE` ist, darf die zur Klasse gehörende

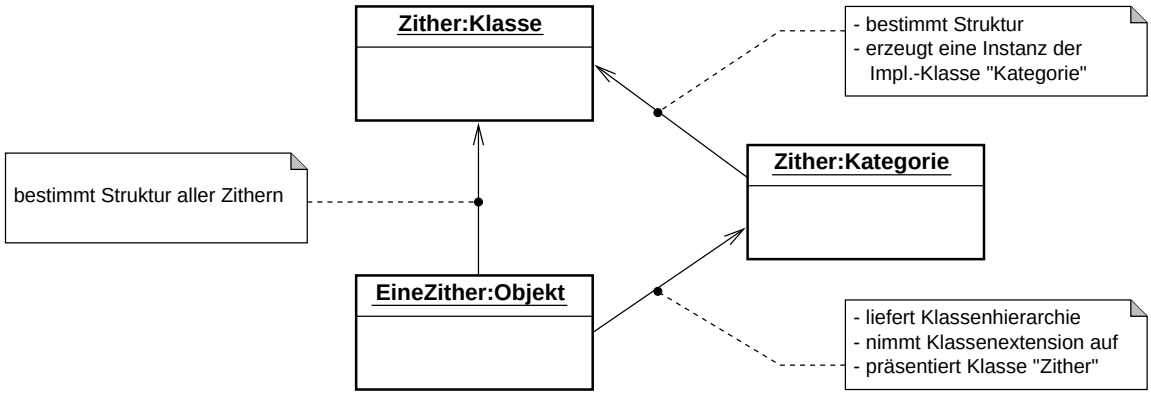

Abbildung 4.13: Dualität der Klasse `Kategorie`

Kategorie ein Attribut mit dem Namen und dem Datentyp dieses Meta-Attributs besitzen. Falls der Quell- bzw. Zieltyp einer Assoziation den Wert `KATEGORIE` hat, dann darf eine Verknüpfung dieser Assoziation nur von dem Kategorieobjekt ausgehen bzw. nur auf dieses zeigen. Abbildung 4.13 illustriert die Dualität der Kategorien.

4.8 Bewertung des entwickelten Datenbankschemas

4.8.1 Vorteile

Das in mehreren Schritten entwickelte Datenbankschema eignet sich zur flexiblen Modellierung komplexer Anwendungsgebiete und kann damit als Grundlage für viele Informationssysteme und Präsentationen benutzt werden. Ermöglicht wird dies durch die folgenden Merkmale des Schemas:

Dynamische Schemaerzeugung und -erweiterung. Das vorgestellte Modell gestattet es, *extensionale Klassen* und *Assoziationen* zu generieren. Extensionale Klassen unterscheiden sich voneinander durch ihre Bezeichnung, ihre Position in der Klassenhierarchie, Art und Anzahl der Attribute sowie Art und Anzahl der Assoziationen zu anderen extensionalen Klassen. Die Strukturinformation für extensionale Klassen und Assoziationen wird vom *Datenschema* geliefert.

Modellierung objektorientierter Konzepte. Das Datenbankschema ermöglicht es, einige objektorientierte Konzepte dynamisch zu modellieren. Dazu gehören *Klasse*, *Assoziation*, *Aggregation*, *Komposition* und *Spezialisierung/Generalisierung* auf Schemaebene sowie *Objekt*, *Attribut*, *Verknüpfung* und *Komponente* auf Datenebene.

Reflexivität auf Datenebene. Die Anwendungsdaten sind in der Lage, sich selbst darzustellen, ohne das zugehörige Datenschema befragen zu müssen. Jedes Objekt kennt den Namen seiner Klasse, seine Position in der Klassenhierarchie sowie seine Attribute und Verknüpfungen. Jedes Attribut kennt seinen Datentyp, seinen Namen sowie das besitzende Objekt. Ebenso kennt jede Verknüpfung den Namen ihrer Assoziation, ihre Attribute, das Zielobjekt sowie die inverse Verknüpfung. *Kategorien* repräsentieren die zugehörigen extensionalen Klassen, speichern Verweise auf alle Objekte der (tiefen) Extension der Klasse und sind in der Lage, die Klassenhierarchie darzustellen.

Reflexivität auf Schemaebene. Neben der aktuellen Objektstruktur ist der Aufbau der extensionalen Klassen und Assoziationen zur Laufzeit abfragbar. Das betrifft Art und Typ der Attribute, Assoziationen und Komponenten. Diese Informationen werden für das Importieren der Daten benötigt (Kapitel 5) und können zur dynamischen Erstellung von Queries herangezogen werden (Unterabschnitt 6.4.5).

Durchgehende Mehrsprachigkeit. Sämtliche Attributwerte und -namen können mehrsprachig angegeben werden. Darüberhinaus liegen alle Namen von Kategorien sowie von extensionalen Klassen und Assoziationen in allen erforderlichen Sprachen vor und können zur Laufzeit abgerufen werden.

Einfache Anbindung eines Information-Retrieval-Systems. Alle im Datenbestand existierenden Textattribute werden von der persistenten Kollektion `Attribute` referenziert. Über dieser Kollektion kann ein Information-Retrieval-System einen Index erstellen (Kapitel 7). Jedes Objekt, das ein Textattribut besitzt, kann über den Pfad `Attribut::Besitzer` erreicht werden.

Generische Speicherung multimedialer Daten. Jede Instanz der Implementierungsklasse `Multimedia` kann bis zu drei Objekte der Klasse `Blob` besitzen, welche jeweils die inhaltlich gleichen Daten in unterschiedlicher Qualität und/oder unterschiedlichem Datenformat speichern.

Objekte der Klasse `Blob` verwalten multimediale Daten auf generische Weise, indem sie neben den eigentlichen Binärdaten die Größe der Daten sowie deren MIME-Typ speichern. Anhand des MIME-Typs kann eine Applikation das geeignete Wiedergabeprogramm für die Binärdaten auswählen.

Jedem Objekt der Klassen `Objekt` und `Kategorie` kann wiederum eine beliebige Anzahl von Objekten der Klasse `Multimedia` per Komposition oder Aggregation zugeordnet werden.

4.8.2 Grenzen

Das vorgestellte Datenbankschema eignet sich als Basis für die Modellierung eher kleiner, aber sehr heterogener Datenbestände. Die hohe Flexibilität wird durch eine gesteigerte Komplexität zur Laufzeit sowie durch Performance-Verluste erkauft. Da das Schema im Kern auf Implementierungsklassen aufsetzt, unterliegt es außerdem den Restriktionen der jeweiligen Programmiersprache. Die folgende Aufzählung beschreibt die Grenzen des Modells ausführlicher und begründet gegebenenfalls, warum diese Grenzen in vielen Fällen hingenommen werden können.

Performance-Verlust durch dynamische Bindung. Die Attribute, Komponenten und Verknüpfungen eines Objekts werden in Kollektionen verwaltet. Der Zugriff auf ein

bestimmtes Kollektionselement erfolgt über den *Namen* des Elements und erfordert eine lokale Suche. Im Gegensatz dazu wird auf ein Attribut eines Objekts, das in einer Programmiersprache definiert ist, meist per Adreßrechnung zugegriffen.

Dieser Nachteil ist nur für strukturierte Queries über dem Datenbestand relevant. Der Zugriff auf die Anwendungsdaten erfolgt aber größtenteils *navigierend* oder über textbasierte Suche. Zur Darstellung eines Objekts werden seine Kollektionen *iteriert* und die Daten der Einträge angezeigt (Unterabschnitt 6.4.4). Die dabei entstehenden Performance-Verluste gegenüber adreßbasiertem Zugriff können in den meisten Fällen vernachlässigt werden.

Hoher Aufwand für die Konsistenzprüfung. Zu den Aufgaben eines DBMS gehört es, die Konsistenz der Daten zu sichern. Durch die dynamische Schemaerzeugung werden die Prüfmechanismen eines DBMS weitgehend ausgeschaltet. Die Ladeprogramme für das Schema und die Daten übernehmen die Integritätssicherung vollständig (Kapitel 5).

Im Gegensatz zu den Prüfmechanismen eines RDBMS (Constraints, Trigger) sind in den meisten OODBMS nur wenige Mechanismen zur Integritätssicherung vorhanden [Heu97, S. 331]. Gerade der ODMG-Standard kennt keine Integritätsbedingungen, die über das Typkonzept der angebundenen Programmiersprachen hinausgehen. Aus diesem Grund muß beispielsweise die Kardinalität von Assoziationen in jedem Fall von einem Ladeprogramm geprüft werden.

Grenzen in der Modellierungsfähigkeit. Das vorgestellte Modell erlaubt es, *einige* objektorientierte Konzepte in *eingeschränkter Form* nachzubilden.

Erstens ist die dynamische Modellierung auf den Strukturteil von Klassen beschränkt. Es gibt keine Möglichkeit, einer *extensionalen* Klasse neue Methoden hinzuzufügen. Eine Methode kann in den bestehenden Implementierungsklassen ergänzt werden, wenn sie von globaler Bedeutung ist. Alternativ besteht die Möglichkeit, die dynamische Klassenhierarchie bis zur Klasse, welche die Methode benötigt, statisch nachzubilden.

Zweitens sind Attributwerte nur in Textform oder numerisch angebbar. Für ein Informationssystem ist diese Einschränkung aber meist akzeptabel: *Flache Datentypen* können stets in Textform konvertiert werden; *Kollektionstypen* werden durch mehrfaches Einfügen eines Attributs mit dem Namen der Kollektion modelliert; *strukturierte Datentypen* lassen sich stets durch *Kompositionen* beschreiben. Falls

doch weitere Datentypen benötigt werden, können Attribute alternativ durch eine Hierarchie von Implementierungsklassen modelliert werden (Unterabschnitt 4.3.2).

Drittens kann nur ein für Informationssysteme wichtiger Teil von objektorientierten Konzepten modelliert werden. Beispielsweise ist es nicht möglich, ternäre Assoziationen adäquat nachzubilden. Weiterhin sind auch die unterstützten Konstrukte mit Einschränkungen behaftet. So kann für eine Assoziation ein Kardinalitäts-*Intervall* angegeben werden, jedoch kein *beliebiger* Ausdruck.

4.9 Zusammenfassung

Das Datenbankschema bildet den Kern der Architektur des Informationssystems. Von seinem Entwurf hängt wesentlich ab, ob die im Kapitel 2 gestellten Anforderungen erfüllt werden können.

Datenbankschemata, die mit klassischen Entwurfstechniken entwickelt und anschließend mit Hilfe von statisch typisierten Datenbank-Programmiersprachen implementiert werden, besitzen meist eine Reihe von Nachteilen. Erstens sind sie nicht reflexiv. Eine Anwendung, die das Datenbankschema nutzt, muß die Schnittstellen aller von ihr benutzten Tabellen oder Klassen kennen. Zweitens sind sie nicht flexibel. Eine Änderung des modellierten Problembereichs erfordert oft einen hohen Aufwand bei der Anpassung von Datenbankschema und Applikationen. Lösungen, die diese Nachteile umgehen, sind oft stark implementierungsabhängig.

Das in diesem Kapitel entwickelte Datenbankschema ist flexibel, da Attribute und Verknüpfungen physisch vom Objekt getrennt und durch Kollektionen von Datenbankzeigern verwaltet werden. Dadurch ist es möglich, Objekte und Verknüpfungen mit vorgegebener Struktur und Semantik zur Laufzeit zu konstruieren. Objekte und Verknüpfungen mit gleichem Aufbau und gleicher Semantik werden unter den Begriffen *extensionale Klasse* bzw. *extensionale Assoziation* zusammengefaßt. Der Aufbau der extensionalen Klassen und Assoziationen wird von einem Datenschema gesteuert, das jeder extensionalen Klasse ein strukturbestimmendes Klassenobjekt zuordnet. Das Datenschema ist ebenfalls dynamisch generierbar und baut auf einem statisch definierten Metaschema auf.

Die Reflexivität der Anwendungsdaten wird dadurch erreicht, daß jedes Objekt, jede Verknüpfung und jedes Attribut nicht nur seine Werte, sondern auch seinen Namen in

mehreren Sprachen speichert. Gleiches gilt für das Datenschema. Auch hier verwaltet jedes Klassenobjekt, jedes Assoziationsobjekt sowie jedes Meta-Attribut alle strukturbestimmenden Angaben in Form von mehrsprachigen Texten. Aufgrund der durchgängigen Reflexivität ist es möglich, die Schnittstelle des Datenbankschemas zu den Applikationen schlank zu halten.

Durch seine Flexibilität und Reflexivität ist das Datenbankschema gut geeignet, eine große Zahl von Problembereichen adäquat zu modellieren. Auf Veränderungen kann reagiert werden, ohne daß die Datenbank-Applikationen angepaßt werden müssen. Der Nachteil der hohen Flexibilität ist eine geringere Performance bei der Suche im Datenbestand. Weiterhin ist durch die Einführung einer eigenen Schemaebene ein relativ hoher Aufwand für die Sicherung der Datenkonsistenz notwendig.

5 Schema- und Datenimport

5.1 Einleitung und Motivation

Das in Kapitel 4 entwickelte Datenbankschema bildet den konzeptionellen Rahmen für eine Vielzahl von Datenschemata. Es legt nicht fest, auf welche Weise ein Datenschema erstellt werden kann, wie die Anwendungsdaten importiert werden können und wie die Integrität von Datenschema und Anwendungsdaten geprüft werden soll.

Zur Definition eines Datenbankschemas wird von vielen DBMS eine Daten-Definitionssprache (*Data Definition Language, DDL*) bereitgestellt [Vos94, S. 26f]. Der Datenimport sowie die Manipulation geladener Daten erfolgt mit Hilfe einer Daten-Manipulationssprache (*Data Manipulation Language, DML*). In objektorientierte DBMS wird eine weitergehende Integration von Datenbank und objektorientierten Programmiersprachen angestrebt. Programmiersprache und Datenbank sollen nicht nur zusammenarbeiten, sondern ein einheitliches Typsystem besitzen. Die Einbettung erfolgt in der Syntax der Programmiersprache. Die Programmiersprache übernimmt somit die Aufgaben der Daten-Definitionssprache, indem die Implementierungsklassen und ihre Relationen untereinander das Datenbankschema definieren. Zumindest für Wirtssprachen mit *statischem* Typsystem gilt, daß dadurch die Schemadefinition zur Übersetzungszeit feststeht und zur Laufzeit nicht mehr geändert werden kann (Unterabschnitt 4.2.3). Die Daten selbst werden als Instanzen der Klassen aufgefaßt und können zur Laufzeit erzeugt, manipuliert und vernichtet werden.

Das im letzten Kapitel entwickelte Datenbankschema trennt den Aufbau der Datenobjekte weitgehend von den zugrundeliegenden Klassen der Programmiersprache. Die Objektstruktur wird nicht mehr von den Implementierungsklassen bestimmt, sondern erst *zur Laufzeit* festgelegt. Aus diesem Grund wird eine Schema-Definitionssprache benötigt, in der Schemaentwürfe formuliert werden können. Ein Schemalader muß die Beschrei-

bung des Datenschemas in der Datenbank bekannt machen, bevor ein Datenbestand importiert werden kann. Daher gliedert sich der Datenimport in zwei Schritte:

- Laden des Datenschemas durch den **Schemalader**,
- Import der Daten durch den **Datenlader**.

Der Schemalader muß es ermöglichen, jedes in Kapitel 4 besprochene Modellierungskonzept aus einer formalen textlichen Beschreibung in ein persistentes Datenschema umzusetzen. Der Datenlader hingegen muß unabhängig von einem konkreten Datenschema sein und jeden Datenbestand, dessen Struktur durch ein (vom Schemalader interpretierbares) Datenschema beschrieben wird, importieren können.

Um die Datenformate des Schemaladers und des Datenladers sowohl für den Menschen als auch für den Computer lesbar zu gestalten, bietet sich eine Beschreibung der Formate durch *kontextfreie Grammatiken* an. Diese kontextfreien Grammatiken bilden die Grundlage für die Erstellung der Ladeprogramme. Daneben dient die Grammatik des Datenladers als Zielformat für die *Vorverarbeitung* der Daten.

In den folgenden Abschnitten wird zunächst besprochen, warum sich kontextfreie Grammatiken gut zur Beschreibung sowohl des Datenschemas als auch der Daten selbst eignen. Anschließend wird erläutert, wie auf der Grundlage dieser Grammatiken Scanner und Parser mit den Werkzeugen Lex und Yacc generiert werden können. Nach den allgemeineren Erläuterungen werden der Schema- und der Datenlader konkret beschrieben: Grammatik der Ladesprachen, Datenbankkopplung und Prüfung der Datenkonsistenz.

5.2 Formalisierung der Syntaxanalyse

5.2.1 Kontextfreie Grammatiken

Kontextfreie Grammatiken und die durch sie beschriebenen *kontextfreien Sprachen* besitzen eine große praktische Bedeutung für die *Definition* und *Analyse* der Syntax von Programmiersprachen. Sie können darüber hinaus überall dort vorteilhaft eingesetzt werden, wo es um die Beschreibung und Analyse von Blockstrukturen, insbesondere um die korrekte Anordnung von Klammern oder anderer korrespondierender Symbole geht.

Eine kontextfreie Grammatik G wird formal durch ein Tupel $G = (V, T, P, S)$ beschrieben, wobei V eine endliche Menge von Variablen, die *Nichtterminale* darstellt. Jede dieser Variablen repräsentiert eine Sprache, die wiederum rekursiv aus anderen Variablen sowie primitiven Symbolen, den *Terminalsymbolen*, zusammengesetzt wird. Die Menge T der Terminalsymbole beinhaltet syntaktisch nicht weiter zerlegbare Einheiten. Die Menge P der Produktionen beschreibt die Regeln, mit denen Zeichenketten und gegebenenfalls Terminalsymbole zu neuen Zeichenketten verknüpft werden können. Alle beteiligten Zeichenketten sind dabei Elemente der durch die zugehörige Variable aus V repräsentierten Sprache. $S \in V$ ist eine spezielle Variable, die als *Startsymbol* bezeichnet wird.

5.2.2 Backus-Naur-Form

Die *Backus-Naur-Form (BNF)* ist eine häufig benutzte Notationsform für kontextfreie Grammatiken. Sie wurde entwickelt, um die Syntax der Programmiersprache Algol 60 zu spezifizieren, und hat seitdem die Entwicklung von Programmiersprachen und Compilern wesentlich vereinfacht. Der Grund dafür ist die intuitive Darstellung der Produktionen einer kontextfreien Grammatik. Die Notation einer Produktion setzt sich aus einem Nichtterminal auf der linken Seite, dem Ableitungssymbol und einer Anzahl von Terminalen und Nichtterminalen auf der rechten Seite zusammen. Wenn ein Nichtterminal auf der linken Seite von mehreren Produktionen vorkommt, läßt sich die Schreibweise dadurch vereinfachen, daß die rechten Seiten der Produktionen durch das Zeichen '|' (oder) separiert werden. Eine einfache Grammatik für die Addition und Subtraktion von Zahlen kann in BNF wie folgt angegeben werden:

```
ausdruck  →  ZAHL
          |  ausdruck '+' ZAHL
          |  ausdruck '-' ZAHL
```

In diesem Beispiel ist `ausdruck` die einzige Variable, während `ZAHL`, + und - Terminalsymbole sind.

Oftmals ist der Zwang zur rekursiven Notation der Ableitungsregeln für das menschliche Verständnis eher hinderlich. Daher wurde die BNF um die Konstrukte Optionalität und Repetition erweitert und mit dem Kürzel *EBNF* bezeichnet [Wir96]. Ein von eckigen Klammern eingefaßter Ausdruck [a] kann einmal oder keinmal angegeben werden, kennzeichnet also die Optionalität von a. Der Ausdruck {a} beschreibt eine beliebig lange

Folge von a. Eine Aneinanderreihung von Ausdrücken in einem Programm kann statt der Form

`programm` $\rightarrow \varepsilon \mid$ `ausdruck programm`

nun einfacher durch

`programm` $\rightarrow \{$ `ausdruck` $\}$

angegeben werden.

5.2.3 Konzeptionelle Phasen der Textanalyse

Der Prozeß der Analyse einer formalen Sprache umfaßt alle Schritte vom Lesen des Quelltextes bis zur Prüfung der inhaltlichen Korrektheit. Man unterscheidet dabei meist drei konzeptionelle Phasen: *lexikalische Analyse*, *syntaktische Analyse* und *semantische Analyse*. Prinzipiell können diese Teilaufgaben streng nacheinander abgearbeitet werden. In realen Programmen werden die Phasen jedoch meist ineinander verschränkt. Die Analyse des Gesamttextes erfolgt dann in mehreren Teilschritten, von denen jeder mehrere konzeptionelle Phasen beinhalten kann.

Lexikalische Analyse

Lexikalische Analyse beschreibt das Zerlegen eines Textes in Zeichenfolgen. Die Zeichenfolgen werden *Symbole* (engl. *tokens*) genannt und sind die kleinsten sinntragenden Einheiten bei der Beschreibung einer formalen Sprache. Sie sind die Terminalsymbole einer kontextfreien Grammatik.

Zur lexikalischen Analyse dienen sogenannte *Scanner*. Scanner untersuchen den Quelltext nach Mustern, die oft durch reguläre Ausdrücke beschrieben sind. Jedem Muster wird ein Symbol zugeordnet. Alle Zeichenfolgen eines Textes, auf die kein Muster paßt, gehören nicht zum Wortschatz der formalen Sprache und führen zu einem Fehler.

Syntaktische Analyse

Die syntaktische Analyse untersucht die Struktur eines Textes, der bereits in einer Folge von Symbolen vorliegt. Die Symbolfolge wird dann als korrekt akzeptiert, wenn sie ein Element der formalen Sprache ist. Fast immer kommen zur Beschreibung der formalen Sprache kontextfreie Grammatiken zum Einsatz. Eine Symbolfolge ω wird dann als Element der formalen Sprache akzeptiert, wenn es eine Folge von Produktionen gibt, durch die sich ω aus dem Startsymbol ableiten läßt. Für die Syntaxanalyse wird ein sogenannter *Parser* eingesetzt. Dieses Programm erhält Symbole von einem Scanner und erstellt während der Analyse einen Parse-Baum. Der Parse-Baum beschreibt die syntaktische Struktur der Eingabe und wird für Optimierungen sowie für die semantische Analyse benutzt.

Die Trennung zwischen lexikalischer und syntaktischer Analyse ist im allgemeinen unscharf. Da reguläre Ausdrücke prinzipiell sowohl vom Scanner als auch vom Parser gelesen werden können, liegt es im Ermessen des Programmierers, wo die Grenze zwischen den Aufgaben der beiden Programme gezogen wird.

Semantische Analyse

In der Phase der semantischen Analyse wird der Quelltext auf inhaltliche Fehler überprüft. Daneben werden Typinformationen gesammelt und in geeignete Datenstrukturen (Symboltabellen) abgelegt. Die semantische Analyse nutzt die hierarchische Struktur, die während der Syntaxanalyse erzeugt wurde. Die Knoten des Syntaxbaums werden, beginnend an den Blättern, mit den Bedeutungen des jeweiligen Teilbaums *attribuiert*. Während der semantischen Analyse kann beispielsweise überprüft werden, ob

- der Wert eines Operanden innerhalb des zulässigen Bereichs liegt,
- die Typen von Operanden zu einem Operator passen,
- eine verwendete Variable deklariert wurde,
- ein Bezeichner innerhalb eines Blockes eindeutig ist.

5.2.4 Der Scannergenerator Lex

Lex ist ein Werkzeug zum Generieren von Scannern. Es entstand als Ergänzung zu Yacc in den siebziger Jahren in den Bell-Laboratorien. Lex erwartet als Eingabe eine Be-

schreibung des zu erstellenden Scanners. Die Beschreibung besteht aus Paaren von regulären Ausdrücken und sogenannten Regeln, die als C-Anweisungen angegeben sind. Lex erzeugt aus der Beschreibung den C-Quelltext eines Scanners, der als Modul in C/C++-Programme eingebunden werden kann oder Teil eines Parser ist. Wichtigster Bestandteil des erzeugten Moduls ist die Funktion `yylex()`. Ein Programm ruft diese Funktion wiederholt auf. Als Ergebnis jedes Aufrufs liefert `yylex()` das nächste Symbol des Eingabestroms oder `EOF`. Andere Informationen werden dem aufrufenden Programm in globalen Variablen bereitgestellt. Dazu gehören die Zeichenfolge, die das Symbol repräsentiert, und die Zeilennummer, an der die Zeichenfolge auftrat.

Das folgende Beispiel zeigt die Spezifikation eines einfachen Lex-Moduls, das Leerzeichen und Zeilenenden ignoriert, Zahlen akzeptiert und für alle weiteren Zeichenfolgen eine Warnung ausgibt:

```
%%
[ \t\n]   ; /* Leerzeichen und Zeilenenden ignorieren */
[0-9]+    { yylval.wert = atoi(yytext); return ZAHL; }
.         { printf("Unerwartetes Zeichen %s\n", yytext); }
```

Auf der linken Seite der Scanner-Spezifikation stehen reguläre Ausdrücke; jeweils rechts daneben befinden sich die Aktionen, die beim Auftreten der regulären Ausdrücke im Text auszuführen sind. Bei den Aktionen handelt es sich um C-Anweisungen oder Lex-Makros, die während der Übersetzung in C-Quelltext überführt werden.

5.2.5 Der Parsergenerator Yacc

Yacc steht für die Abkürzung "Yet Another Compiler Compiler". Das Programm entstand, zeitlich etwas vor Lex, ebenfalls in den Bell-Laboratorien. Der Name deutet an, daß in dieser Zeit Compiler-Generatoren Hochkonjunktur hatten.

Yacc erwartet eine kontextfreie Grammatik in Backus-Naur-Notation als Eingabe und erzeugt den C-Quelltext des zugehörigen Parsers. Die Yacc-Spezifikation des Beispiels aus Unterabschnitt 5.2.2 hat folgendes Aussehen:

```
%{
int Wert;            /* Variablendeklaration           */
%}
%union { int wert; } /* Vereinbarung der Symboltypen    */
%token <wert> ZAHL   /* Deklaration der Terminalsymbole */
%start ausdruck      /* Vereinbarung des Startsymbols   */
%%
ausdruck : ZAHL               { Wert = yylval.wert; }
         | ausdruck '+' ZAHL  { Wert+= yylval.wert; }
         | ausdruck '-' ZAHL; { Wert-= yylval.wert; }
```

Die Terminalsymbole müssen vor ihrer Benutzung deklariert werden; intern werden sie auf natürliche Zahlen abgebildet. Das ist auch der Grund, warum Einzelzeichen (hier `'+'` und `'-'`) direkt im Regelteil benutzt werden dürfen: C behandelt Zeichen genauso wie vorzeichenlose Integer. An beliebigen Stellen des rechten Teils einer Regel dürfen C-Ausdrücke, umschlossen von geschweiften Klammern, eingefügt werden. Diese Ausdrücke werden *Aktionen* genannt. Wenn das vor einer Regel stehende Symbol ein Terminal repräsentiert, so wird der C-Code ausgeführt, wenn das Symbol gelesen wurde. Handelt es sich dagegen um ein Nichtterminal, so wird der Programmteil dann abgearbeitet, wenn die zum Nichtterminal gehörende Regel komplett angewendet wurde. Die Aktionen sind die Schnittstelle des Parsers zu anderen C-Modulen. Das Beispiel macht ebenfalls die Zusammenarbeit mit dem Scanner deutlich, der neben den Symbolen (hier `ZAHL`) auch deren Wert als *Attribut* (hier `yylval.wert`) liefert.

Obwohl Yacc prinzipiell für beliebige kontextfreie Grammatiken einsetzbar ist, gibt es Fälle, die Yacc nicht handhaben kann. Es kann mehrdeutige Grammatiken nicht parsen. Mehrdeutige Grammatiken besitzen für wenigstens eine Satzform mehr als einen möglichen Ableitungsbaum. Weiterhin kann Yacc nur Grammatiken parsen, bei denen stets mit höchstens einem Symbol "Vorausschau" entscheidbar ist, welche Regel angewendet werden muß.

Yacc baut den Syntaxbaum für einen Quelltext nach dem *Bottom-up*-Prinzip auf. Ohne zunächst die Zielstruktur des Baums zu kennen, wird der Eingabetext gelesen, und nach jedem Symbol wird geprüft, ob die gelesene Folge einem Syntaxkonstrukt entspricht. Wenn *keine* Regel anwendbar ist, wird das Symbol auf einen Stack gelegt (*geshifted*). Wird jedoch eine passende Regel gefunden, so wird dieses Konstrukt auf das Nichtterminal auf der rechten Seite der Regel *reduziert*. Man spricht aufgrund dieser Arbeitsweise

auch von *shift/reduce*-Parsern. Beim Bottom-up-Prinzip wird stets die Symbolfolge am *rechten Ende* des Stacks reduziert. Nach D.E. KNUTH nennt man dieses Prinzip LR(1)-Parsing. Der Buchstabe L spiegelt die Tatsache wider, daß der Eingabestrom von *links* nach rechts gelesen wird; die (1) steht für ein Symbol Vorausschau. Auf die Parsing-Technik von Yacc wird ausführlich in [LMB92] und [ASU88] eingegangen.

5.3 Spezifikation der Importformate

5.3.1 Anforderungen an die Importformate

Die Grammatiken der Importformate für den Schema- sowie den Datenlader müssen allen Anforderungen genügen, wie sie generell an Programmiersprachen und andere strukturierte Sprachen gestellt werden, die sowohl für den Menschen verständlich als auch für die Maschine lesbar sein sollen. Sie müssen zunächst *vollständig* in bezug auf ihre Ausdrucksfähigkeit sein. Das bedeutet, daß (mindestens) alle Konstrukte erzeugt werden können, die für die Anwendung benötigt werden. Mit den Grammatiken dürfen jedoch nur *korrekte* Konstrukte generierbar sein. Diese Forderung ist auf der Ebene der Grammatik sicherlich zu streng. Zur Überprüfung der Korrektheit muß daher zusätzlich eine semantische Analyse erfolgen.

Eine weitere wichtige Anforderung an die Importformate ist die *Robustheit*. Die Ladeprogramme müssen auch bei fehlerhafter Syntax und Semantik des Textes ein definiertes Verhalten zeigen. Robustheit ist besonders wichtig, weil die Texte vom Menschen erzeugbar und editierbar sind und deshalb mit relativ hoher Wahrscheinlichkeit unkorrekte Passagen enthalten können.

Gerade um die Fehlerquellen für Menschen gering zu halten und die zu ladenden Texte gut interpretierbar zu gestalten, muß die Syntax einen "intuitiven" Aufbau haben. Dabei spielt es eine eher untergeordnete Rolle, ob Informationen redundant auftreten, vorausgesetzt, sie tragen zum Verständnis der Quelle bei.

Der entgegengesetzte Aspekt ist die Maschinenlesbarkeit. Prinzipiell kann für jede Grammatik ein Parser konstruiert werden. In der Praxis benutzte Grammatiken haben jedoch einige besondere Eigenschaften, um die Komplexität und den Zeitbedarf des Parsers niedrig zu halten. Zu jeder kontextfreien Grammatik existiert ein Parser mit einem

Zeitbedarf der Ordnung $O(n^3)$ für die Analyse eines Wortes aus n Symbolen [ASU88, S. 50]. Allerdings ist kubischer Zeitbedarf in der Regel nicht akzeptabel. Daher versucht man, Grammatiken mit geringerer Komplexität zu konstruieren. Tatsächlich kann man für fast jede praktisch benutzte Sprache Grammatiken finden, die sich mit geringerem Zeitbedarf analysieren lassen. Meist kommt man mit linearen Algorithmen aus, die die Eingabe mit einem Symbol Vorausschau von links nach rechts symbolweise abarbeiten. Die von Yacc akzeptierte Klasse der LALR(1)-Grammatiken, die eine Teilklasse der LR(1)-Grammatiken ist, besitzt diese Eigenschaften.

Zusammenfassend kann gesagt werden, daß die Grammatiken für die Lader einerseits die Eigenschaften Vollständigkeit, Korrektheit und Robustheit haben müssen, und andererseits einen guten Kompromiß zwischen Verständlichkeit für den Menschen und Lesbarkeit für die Maschine bilden sollten.

5.3.2 Zusatzanforderungen an den Schemalader

Das Datenschema beschreibt die Struktur einer Instanz des Metaklassenschemas. Es wird geprägt durch die Anzahl und den Aufbau der Klassenobjekte sowie durch Anzahl und Typ der Assoziationen zwischen den Klassenobjekten.

Eine Grammatik, die als Grundlage für den Schemalader dient, muß jedes auf der Basis des Metaklassenschemas erzeugbare (sinnvolle) Datenschema korrekt und vollständig beschreiben können. Um diesen Anspruch erfüllen zu können, müssen sprachliche Mittel für die folgenden Punkte zur Verfügung stehen:

- Definition von Klassenobjekten als Instanzen von Implementierungsklassen.
- Angabe von Basis-Klassenobjekten, deren Eigenschaften geerbt werden.
- Zuordnung von Attributen bestimmten Typs zu Klassenobjekten.
- Erstellen von Assoziationen zwischen Klassenobjekten.
- Erzeugen von zusammengesetzten Klassenobjekten mittels Aggregation und Komposition.

Insbesondere das Erstellen von Assoziationen, Aggregationen und Kompositionen birgt einige Schwierigkeiten. Es müssen Verweise auf Klassenobjekte möglich sein, die noch nicht gelesen wurden (Vorwärtsreferenzen). Assoziationen können *invers* sein. In diesem Fall muß das Klassenobjekt am Ende der Assoziation ebenfalls eine Assoziation

definieren. Die Zusammengehörigkeit beider Assoziationen muß vermerkt werden. Assoziationen besitzen eine vorgeschriebene Kardinalität sowohl auf der Quell- als auch auf der Zielseite. Für jede Assoziation muß folglich angebbar sein, welche Kardinalität an beiden Enden erlaubt ist.

5.3.3 Zusatzanforderungen an den Datenlader

Der Datenlader benötigt ein Importformat, dessen Syntax *vollständig unabhängig* von einem konkreten Datenschema ist. Das ist eine wichtige Forderung, da ein Datenschema vom Schemalader erzeugt wird und auf der Grundlage der beschriebenen Modellierungskonzepte beliebig variieren kann. Das Importformat des Datenladers muß all diesen möglichen Varianten gerecht werden, ohne daß die Grammatik angepaßt werden müßte. Daneben gibt es noch eine Reihe weiterer Anforderungen an das Format des Datenladers. Es muß

- für Attribute mit beliebigem Namen mehrsprachige oder numerische Werte zulassen,
- es gestatten, Attributwerte aus einer Datei zu laden,
- das Erstellen von Verknüpfungen auf bereits geladene Daten mittels frei wählbarer Kriterien erlauben,
- die Angabe von Komponentenobjekten zulassen und
- es ermöglichen, multimediale Daten zu klassifizieren und in die Datenbank zu laden.

Noch einmal soll unterstrichen werden, daß die Syntax des Datenladers nicht festlegen darf, welche Attribute für ein bestimmtes Objekt angegeben werden dürfen, welcher Art eine Verknüpfung sein darf und welchen Aufbau ein Komponentenobjekt haben kann. Die Überprüfung dieser und anderer schemaspezifischer Angaben muß der Datenlader auf semantischer Ebene durchführen, indem er beispielsweise gelesene Klassen-, Attribut- und Verknüpfungsnamen mit dem Datenschema abgleicht.

5.3.4 Sprachdefinition und Pragmatik

Für den Entwurf einer Sprache gibt es keine allgemeingültige Vorgehensweise. Grundsätzlich muß eine zu erstellende Grammatik die in Unterabschnitt 5.3.1 gestellten An-

forderungen erfüllen. Innerhalb dieses Anforderungskatalogs verbleibt ein relativ großer Spielraum bei der Festlegung einer konkreten Sprache.

Beim Entwurf der Grammatiken für den Schema- und den Datenlader wurden einige Syntaxelemente von objektorientierten Programmiersprachen übernommen. Besonders die Beschreibung eines Klassen- bzw. Datenobjekts ist an C++ angelehnt. Anders verhält es sich bei der Definition der Elemente eines Klassenobjekts. Hier liegt besonderes Augenmerk darauf, die *Funktion* des jeweiligen Elements zu verdeutlichen. Deshalb werden diese Elemente durch Schlüsselworte wie `TEXT` oder `AGGREGATION` eingeleitet. Im Gegensatz zu Programmiersprachen dürfen Bezeichner aus mehreren Worten bestehen. Sie werden durch Strukturzeichen oder Schlüsselworte, die stets aus Großbuchstaben bestehen, separiert.

Das Design der Grammatiken für Schema- und Datenlader ist hochgradig subjektiv. Viele weitere Entwurfsvarianten sind denkbar, die ebenso allen Anforderungen genügen würden.

5.3.5 Importformat des Schemaladers

Der Aufbau einer Definitionsdatei für ein konkretes Datenschema besteht aus der Aufzählung aller zu erstellenden *Klassenobjekte*, die jeweils eine extensionale Klasse definieren. Die folgenden Absätze beschreiben die Bestandteile einer Klassendefinition näher, gehen jedoch nicht auf alle Details ein. Der informellen Beschreibung eines Sprachelements schließt sich jeweils die formale Notation an, die aus Gründen der Übersichtlichkeit etwas vereinfacht wurde und in *erweiterter* BNF angegeben ist. Die vollständige Grammatik des Schemaladers in Standard-BNF befindet sich im Abschnitt C.1 des Anhangs.

Eine Klassendefinition wird durch den Namen der *Implementierungsklasse* eingeleitet. Die gegenwärtig zulässigen Angaben sind `OBJEKT` und `MULTIMEDIA`, welche die Klassen `Objekt` bzw. `Multimedia` des Datenbankschemas in Abbildung 4.12 auf Seite 49 symbolisieren. Fehlt die Angabe der Implementierungsklasse, so wird von der Klasse `Objekt` ausgegangen. In jedem Fall muß der *interne Name* der Klasse angegeben werden. Dabei handelt es sich um einen beliebigen eindeutigen Bezeichner für das Klassenobjekt. Optional kann eine Liste von Basisklassen folgen, deren Eigenschaften, einschließlich der Verweise auf andere Klassen, an das neu erzeugte Klassenobjekt vererbt werden.

```
Klassenobjekt  →  [ ImplKlasse ] KlassenName [ BasisKlassen ] KlassenDef
ImplKlasse     →  OBJEKT | MULTIMEDIA
BasisKlassen   →  ':' BasisKlasse { ',' BasisKlasse }
```

Die eigentliche Klassendefinition wird in geschweifte Klammern eingefaßt. An erster Stelle nach der öffnenden Klammer muß der *Titel* der Klasse in allen erforderlichen Sprachen angegeben sein. Der Titel wird ausschließlich für Darstellungszwecke benötigt. Nach dem Titel schließen sich in beliebiger Reihenfolge Attributdeklarationen, Angaben zur Kategorie der Klasse, Definitionen von Komponentenobjekten sowie Assoziations- und Aggregationsbeziehungen zu anderen Klassenobjekten an.

```
KlassenDef    →  '{' KlassenTitel { KlassenDaten } '}'
KlassenTitel  →  TITEL Titel { '|' Titel }
KlassenDaten  →  Kategorie
              |  Attribut
              |  Assoziation
              |  Aggregation
              |  Komposition
```

Attribute können vom Typ `KURZTEXT`, `TEXT` oder `ZAHL` sein. Die Typen `KURZTEXT` und `TEXT` unterscheiden sich lediglich in ihrer Struktur. Texte dürfen im Gegensatz zu Kurztexten Zeilenend-Zeichen und HTML-Formatanweisungen beinhalten. Nach der Typangabe muß der Name des Attributs in allen geforderten Sprachen folgen.

```
Attribut     →  AttributTyp AttributName { '|' AttributName }
AttributTyp  →  ZAHL | KURZTEXT | TEXT
```

Assoziationen sind durch das Schlüsselwort `ASSOZIATION` gekennzeichnet. Optional kann die Angabe eines Kardinalitätsintervalls folgen. Hieran schließt sich der *interne Assoziationsname* an, gefolgt von dem Namen der Zielklasse der Assoziation. Der Programmierer kann mit dem Schlüsselwort `KATEGORIE` festlegen, daß diese Assoziation nur auf das Kategorieobjekt der Zielklasse zeigen darf. Eine Assoziation kann *invers* sein, was durch das Schlüsselwort `INVERSE`, gefolgt vom Namen der inversen Assoziation, kenntlich gemacht wird. Assoziationen dürfen durch Attribute spezifiziert werden, die genau wie Attribute in der Klassendefinition notiert werden. Attribute inverser Assoziationen werden für

beide Richtungen benutzt. Eine Aggregation wird durch das Schlüsselwort `AGGREGATION` gekennzeichnet. Im Gegensatz zu Assoziationen dürfen Aggregationen keine Inverse besitzen.

```
Assoziation  →  ASSOZIATION AssozKopf [ INVERSE InverseName ] AssozDef
Aggregation  →  AGGREGATION AssozKopf AssozDef
AssozKopf    →  [ Kardinalität ] AssozName ':' [ KATEGORIE ] ZielKlasse
AssozDef     →  '{' AssozTitel { AssozDaten } '}'
AssozTitel   →  TITEL Titel { '|' Titel }
AssozDaten   →  Attribut
```

Eine Komposition entspricht einer Einbettung der Komponente in das Aggregat, wobei die Komponente keine eigene Identität besitzt. Daher kann eine Komposition syntaktisch wie eine eigene Klasse innerhalb der umgebenden Klasse formuliert werden. Alternativ oder gleichzeitig können Strukturinformationen von einer oder mehreren Basisklassen in die Komponentenklasse übernommen werden. In einer Komponentenklasse darf im Gegensatz zu Klassenobjekten keine Kategorie angegeben werden.

```
Komposition       →  KOMPOSITION KompositionKopf KompositionDef
KompositionKopf   →  [ Kardinalität ] KompositionName [ BasisKlassen ]
BasisKlassen      →  ':' BasisKlasse { ',' BasisKlasse }
KompositionDef    →  '{' { KompositionDaten } '}'
KompositionDaten  →  Attribut | Assoziation | Aggregation | Komposition
```

Für jedes Klassenobjekt wird ein Objekt der Implementierungsklasse `Kategorie` mit dem Namen des Klassenobjekts erzeugt. Die Definition des Kategorieobjekts wird innerhalb der Klassendefinition mit dem Schlüsselwort `KATEGORIE` eingeleitet. Ein Kategorieobjekt kann die gleichen Bestandteile wie ein Klassenobjekt besitzen; unzulässig ist lediglich die Definition einer Kategorie innerhalb einer äußeren Kategorie.

Als Beispiel für eine Schemadefinition dient ein Auszug aus dem Datenschema für das Musikinstrumentenmuseum:

```
Exponat {
    TITEL Exponat | Exhibit
    KURZTEXT Datierung | Date
```

```
    TEXT Besonderheiten | Specials
    KATEGORIE {
        TEXT Geschichte | History
    }
    ASSOZIATION [0..1,0..*] Besitzer: Person INVERSE Besitz {
        TITEL Besitzer | Owner
    }
    ASSOZIATION Erbauer: Hersteller INVERSE Werke {
        TITEL Erbauer | Builder
    }
    AGGREGATION Fotos: Foto {
        TITEL Fotos | Photos
    }
}

MULTIMEDIA Foto {
    TITEL    Foto     | Photo
    KURZTEXT Name     | Picture
    KURZTEXT Fotograf | Photographer
}

Person {
    TITEL Person     | Person
    KURZTEXT Name    | Name
    KURZTEXT Wohnort | Residence
    ASSOZIATION [0..*,0..1] Besitz: Exponat INVERSE Besitzer {
        TITEL Hat Exponate | Has exhibits
    }
}

Hersteller {
    TITEL Hersteller | Manufacturer
    KURZTEXT Name | Name
    ASSOZIATION Werke: Exponat INVERSE Erbauer {
        TITEL Werke | Work
```

```
        KURZTEXT Bemerkung | Annotation
    }
}

Meister: Hersteller, Person {
    TITEL Meister | Master
    KURZTEXT Weitere Werke | Other work
}

...

Orgel: Tastenaerophon {
    TITEL: Orgel | Organ
    KOMPOSITION Pfeifenwerk {
        KURZTEXT Register  | Pipes
    }
    KOMPOSITION Windwerk {
        KURZTEXT Windladen | Wind-chests
    }
}
```

5.3.6 Importformat des Datenladers

Die Struktur der Anwendungsdaten ist vollständig vom Datenschema abhängig, welches vom Schemalader erzeugt wird. Die Syntax des Datenladers ist daher sehr allgemein gehalten. Dagegen ist ein relativ hoher Aufwand für die Überprüfung der Semantik nötig: Für eine syntaktisch korrekte Importdatei muß unter anderem geprüft werden, ob die Namen der Klassenobjekte, der Typ und die Kardinalität der Verknüpfungen sowie der Name und der Wertebereich der Attribute mit den Angaben im Datenschema übereinstimmen (Abschnitt 5.5).

Ein Objekt besteht aus dem Namen des Klassenobjekts sowie der Definition von Attributen, Verknüpfungen und Komponentenobjekten.

```
Objekt       →  KlassenName ObjektTitel '{' ObjektDaten'}'
ObjektTitel  →  TITEL Titel { '|' Titel }
ObjektDaten  →  Attribut | Verknüpfung | Blob | Objekt
```

Attribute werden durch ihren im Datenschema festgelegten Namen eingeleitet. Nach einem trennenden Doppelpunkt schließen sich die Attributwerte in mehreren Sprachen an. Falls ein Attributwert nur in einer Sprache angegeben ist, beispielsweise bei einem Eigennamen, kopiert der Datenlader diesen Wert für alle anderen Sprachen. Numerische Attributwerte werden auf der Ebene der Grammatik ebenfalls durch Zeichenketten spezifiziert, die vom Datenlader in Zahlenwerte konvertiert werden. Alternativ können die Attributwerte aus einer Datei geladen werden. In diesem Fall muß der Dateiname in eckigen Klammern angegeben sein.

```
Attribut      →  AttributName ':' AttributWert { '|' AttributWert }
AttributWert  →  Text | '[' Dateiname ']'
```

Verknüpfungen beginnen mit dem Namen der zugehörigen Assoziation. Eingeschlossen in spitze Klammern folgt eine beliebige Anzahl von Attributnamen und -werten, die das Zielobjekt der Verknüpfung eindeutig spezifizieren müssen. Falls das Datenschema für die Assoziation Attribute vorsieht, kann die Verknüpfung eine Teilmenge dieser Attribute mit Werten versehen.

```
Verknüpfung   →  VerknüpfName '<' VerknüpfAttr '>' [ VerknüpfDef ]
VerknüpfAttr  →  Attribut { Attribut }
VerknüpfDef   →  '{' { Attribut } '}'
```

Zu jedem Objekt der Implementierungsklasse `Multimedia` können, eingeleitet durch die Schlüsselworte `GERING`, `MITTEL` und `HOCH`, Namen von Dateien mit multimedialen Daten angegeben werden. Die Endung der Binärdatei bestimmt den zugehörigen MIME-Typ. Die Zuordnung einer Dateiendung zu einem MIME-Typ erfolgt über eine einfach aufgebaute Textdatei, die beliebig erweitert werden kann (vgl. Anhang D.1). Der Datenlader verlangt, daß mindestens die Qualitätsstufe `GERING` angegeben ist.

```
Blob     →  BlobTyp ':' '[' Dateiname ']'
BlobTyp  →  GERING | MITTEL | HOCH
```

Die Syntax für die Notation von Komponenten unterscheidet sich nicht von der Syntax für Objekte der Ebene 0. Die Attribute und Verknüpfungen von Kategorieobjekten werden ebenfalls vom Datenlader importiert. Kategorieobjekte werden zur Unterscheidung von einfachen Datenobjekten mit dem Schlüsselwort `KATEGORIE` eingeleitet.

Das folgende Beispiel zeigt einen Ausschnitt aus dem Datenbestand des Musikinstrumentenmuseums:

```
Meister {
    Name: Johann Matthäus Schmahl
    Wohnort: Ulm
}

Foto {
    Name: Hammerklavier | Fortepiano
    Fotograf: Janos Stekovics
    GERING: [fotos/hammerkl.gif]
    MITTEL: [fotos/hammerkl.jpg]
}

Foto {
    Name: Hammerklavier, Deckelgemälde | Fortepiano, painting on the lid
    GERING: [fotos/hkldetail.gif]
    MITTEL: [fotos/hkldetail.jpg]
}

Saitenklavier {
    Name: Hammerklavier | Fortepiano
    Herkunft: Ulm
    Datierung: um 1780 | circa 1780
    Inventarnummer: 104
    Erbauer <Name: Johann Matthäus Schmahl> {
        Bemerkung: zugeschrieben | attributed to
    }
    Fotos <Name: Hammerklavier
          Fotograf: Janos Stekovics>
    Fotos <Name: Hammerklavier, Deckelgemälde>
```

```
    Beschreibung: [texte/de/inv104.txt] | [texte/en/inv104.txt]
}

Orgel {
    Name: Orgelpositiv | Positive organ
    Herkunft: Tirol | Tyrol
    Datierung: vermutlich 1614 | presumably 1614
    Pfeifenwerk {
        Register: Prinzipal 4', Oktave 2' | Principal 4', Octave 2'
    }
}
```

5.4 Architektur und Funktionsweise des Schemaladers

5.4.1 Die Komponenten und ihre Aufgaben

Der Schemalader gliedert sich in fünf funktionelle Einheiten: Scanner, Parser, Klassenpuffer, Metaklassenschema und Hauptprogramm. Abbildung 5.1 zeigt die Klassenstruktur des Schemaladers in vereinfachter Form.

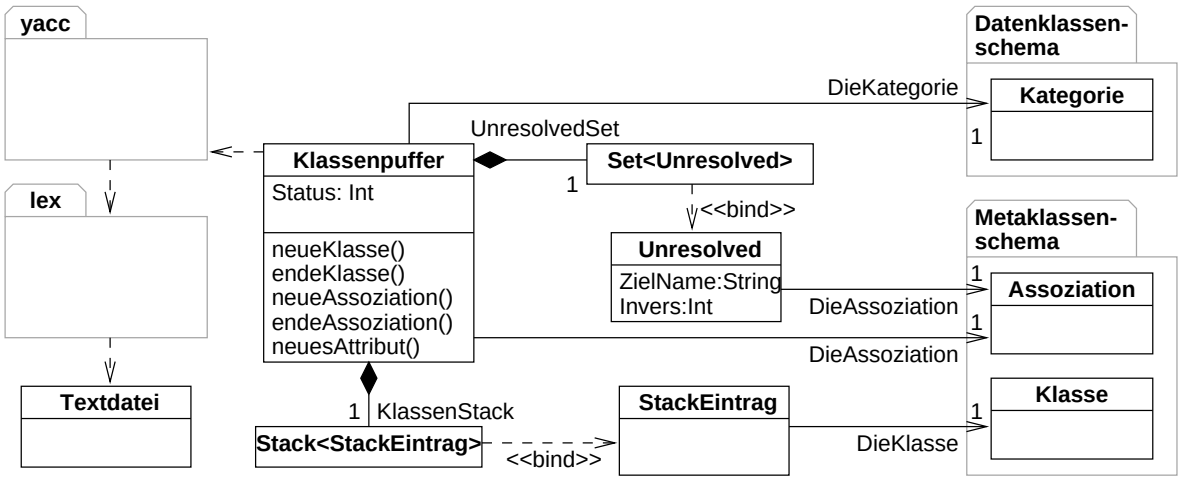

Abbildung 5.1: Klassendiagramm des Schemaladers (vereinfacht)

Scanner. Der Scanner ist ein C-Modul, das vom Scannergenerator Lex erzeugt wird. Es definiert die Funktion `yylex()`, die aus dem Eingabestrom das nächste Symbol extrahiert und an den Aufrufer zurückliefert.

Parser. Der Parser ist eine C-Datei, die von Yacc generiert wird. Innerhalb der Hauptroutine `yyparse()` des Parsers wird iterativ die Funktion `yylex()` des Scanners gerufen. Wie in Unterabschnitt 5.2.5 erläutert wurde, kann innerhalb des Regelteils von Yacc-Quelldateien C-Code eingefügt werden, der als Schnittstelle zu anderen Programmteilen dient. Im Fall des Schemaladers handelt es sich dabei um Methoden der Klasse `Klassenpuffer`, die vom Parser als *Callbacks* an entsprechenden Stellen gerufen werden.

Klassenpuffer. Die Klasse `Klassenpuffer` speichert die vom Parser gelieferten Informationen zwischen. Der Aufruf der Methode `Klassenpuffer::neueKlasse()` durch den Parser signalisiert den Beginn eines neuen Klassenobjekts. Innerhalb der Methode wird eine neue Instanz von `Klasse` erzeugt. Da eine extensionale Klasse Komponenten besitzen darf, die wiederum selbst Komponenten definieren dürfen, genügt eine einfache Pufferung nicht. Die Klassenobjekte müssen auf einem Stack zwischengespeichert werden. Nur das jeweils obenliegende Objekt nimmt neue Attribute oder Assoziationen auf. Die Methode `Klassenpuffer::endeKlasse()`, deren Aufruf von einer schließenden geschweiften Klammer ausgelöst wird, nimmt das oberste Klassenobjekt vom Stack und speichert es in der Datenbank. Analog dazu dienen die Referenzen `DieKategorie` und `DieAssoziation` zum Puffern von Strukturinformation der aktuell geparsten Kategorie bzw. Assoziation. Klassenobjekte dürfen andere Klassenobjekte sowohl rückwärts als auch vorwärts referenzieren. Kann eine Referenz nicht sofort aufgelöst werden, wird sie zunächst in der Kollektion `Klassenpuffer::UnresolvedSet` abgelegt. Am Ende des Ladevorgangs wird erneut versucht, diese Verweise zu erstellen.

Metaklassenschema. Die Verbindung zum Metaklassenschema der Datenbank wird über die Programmierschnittstelle der Datenbank hergestellt. Alle komplett gelesenen Klassenobjekte werden in der persistenten Kollektion "Klassen" abgelegt. Diese Kollektion wird außerdem für das Erstellen von Assoziationen genutzt.

Hauptprogramm. Das Hauptprogramm initialisiert den Parser und den Klassenpuffer und öffnet die an der Kommandozeile des Programms angegebene Quelldatei. Daran

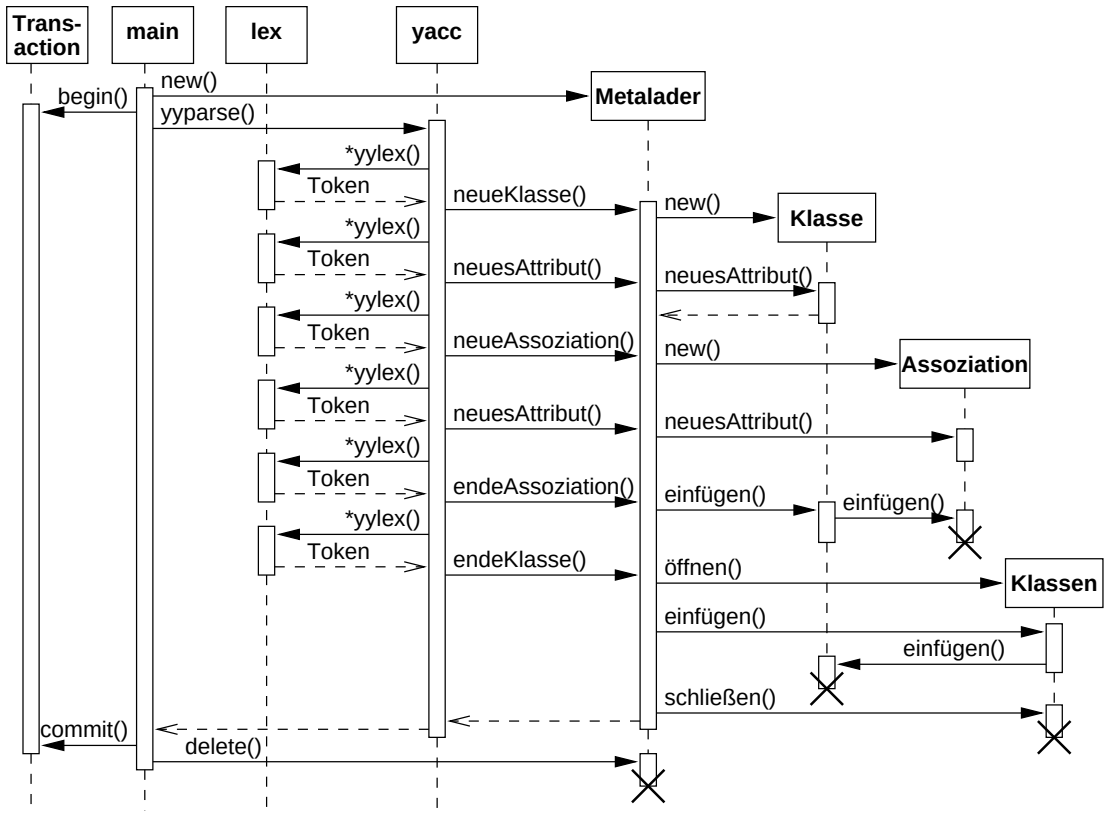

Abbildung 5.2: Typischer Nachrichtenfluß des Schemaladers (vereinfacht)

anschließend stellt das Hauptprogramm die Verbindung zur Datenbank her und startet eine Transaktion, innerhalb derer der gesamte Ladevorgang abläuft. Abbildung 5.2 zeigt den zeitlichen Ablauf des Ladens in vereinfachter Form. Dargestellt ist das Laden eines Klassenobjekts mit einem Attribut und einer Assoziation, die wiederum ein Attribut besitzt.

Zu Beginn des Ladevorgangs wird ein eventuell schon vorhandenes Datenschema vom Lader gelöscht. Dies kann auch dann geschehen, wenn Daten zu diesem Datenschema existieren, da es keine physische Verbindung zwischen Datenschema und Datenbestand gibt. Allerdings liegt es in der Verantwortung des Administrators sicherzustellen, daß das neu zu ladende Datenschema identisch zum bisherigen ist oder dieses *erweitert*. Die Erweiterung umfaßt folgende Möglichkeiten:

- Erzeugung eines neuen *Klassenobjekts*. Die neue Klasse kann an beliebiger Stelle in die bisherige Typhierarchie eingefügt werden oder eine neue Teilhierarchie beginnen.
- Ergänzung neuer Attribute zu einem bestehenden Klassenobjekt.
- Definition neuer Assoziations-, Aggregations- und Kompositionsbeziehungen. Dabei spielt es keine Rolle, ob Quell- und Ziel-Klassenobjekte im bisherigen Datenschema existierten oder neu erzeugt wurden.

Dabei handelt es sich ausschließlich um Erweiterungen des Datenschemas, für die am zugehörigen Datenbestand keine oder ebenfalls nur erweiternde Änderungen vorgenommen werden müssen. Neue Attribute oder Assoziationen zwischen Klassenobjekten haben keine Auswirkungen auf den Datenbestand, da Attribute und Verknüpfungen über eine Kollektion von Zeigern verwaltet werden. Daher müssen für neue Attribute und Verknüpfungen keine leeren Speicherplätze im Datenbestand reserviert werden. Die Erzeugung eines neuen Klassenobjekts muß nur in der Hierarchie der Kategorien vermerkt werden. Bei Ergänzung einer neuen Oberklasse muß das zugehörige Kategorieobjekt insbesondere die Extensionen seiner Unterkategorien in die eigene Extension aufnehmen.

5.4.2 Verarbeitung von Sprachelementen

Vererbung. Ein Klassenobjekt kann von einer oder mehreren Klassen abgeleitet werden, die vollständig *vor* der erbenden Klasse definiert sein müssen. Vererbt werden alle Attribute, Assoziationen, Aggregationen und Kompositionen der Basisklasse, indem die Kollektionen `Attribute`, `Assoziationen` und `Komponenten` kopiert werden. Bei mehreren Basisklassen kann das ohne Schwierigkeiten erfolgen, wenn der Inhalt der Kollektionen disjunkt ist. Besitzen hingegen zwei oder mehr Basisklassen Attribute oder Referenzen gleichen Namens, so kommt es zu einem Konflikt, der dadurch gelöst wird, daß nur das Attribut oder die Referenz des in der Vererbungsliste zuerst angegebenen Klassenobjekts in die Tochterklasse übernommen wird. Der Nutzer erhält eine detaillierte Nachricht über die Art des Konflikts.

Assoziationen. Trifft der Schemalader auf die Schlüsselworte `ASSOZIATION` oder `AGGREGATION`, so wird ein Objekt der Implementierungsklasse `Assoziation` erzeugt. Die Assoziation wird sofort ihrem Klassenobjekt zugeordnet. Anschließend wird unter

den bisher geladenen Klassenobjekten die Zielklasse der Assoziation gesucht. Kann das Ziel nicht gefunden werden, so legt der Schemalader die Assoziation in der Kollektion `UnresolvedSet` der nicht aufgelösten Verweise ab. Nachdem das Datenschema komplett gelesen wurde, versucht der Lader, alle noch nicht erstellten Referenzen aufzulösen. Gelingt das nicht in allen Fällen, bricht der Ladevorgang mit einem Fehler ab.

Eine besondere Behandlung erfordern *inverse Assoziationen*. Hier muß die Zielklasse eine Assoziation mit dem Namen besitzen, der nach dem Schlüsselwort `INVERSE` angegeben wurde. Diese Assoziation muß ebenfalls als invers deklariert sein. Zusätzlich dürfen sich eventuell angegebene Kardinalitäten inverser Assoziationen nicht widersprechen.

Kompositionen. Eine Komponentenklasse wird innerhalb einer umgebenden Klasse definiert und muß deshalb nur einen *lokal* eindeutigen Bezeichner haben. Nach dem Lesen des Schlüsselwortes `KOMPOSITION` wird eine Klasse erzeugt und auf dem `KlassenStack` abgelegt. Alle folgenden Zugriffe beziehen sich stets auf das oberste Stack-Element. Falls eine oder mehrere Basisklassen angegeben sind, wird deren Strukturteil kopiert. Ausgenommen davon sind Angaben zur Kategorie und inverse Assoziationen. Im Körper einer Komponentenklasse angegebene Attribute, Assoziationen, Aggregationen und Kompositionen werden genau wie bei der Klasse der obersten Ebene behandelt, wiederum mit der Einschränkung, daß Angaben zur Kategorie und zu inversen Assoziationen unzulässig sind.

Kategorien. Ein Kategorieobjekt wird erzeugt, nachdem ein neuer Klassenname gelesen wurde und es sich dabei um eine Klasse der obersten Ebene handelt. Das Kategorieobjekt wird im Feld `DieKategorie` des Klassenpuffers abgelegt. Angaben zur Kategorie der Klasse werden durch das Schlüsselwort `KATEGORIE` eingeleitet, das nur innerhalb einer Klassendefinition der Ebene 0 erscheinen darf. Alle Attribut-, Assoziations-, Aggregations- und Kompositionsdefinitionen werden sofort dem erzeugten Kategorieobjekt zugeordnet.

5.5 Architektur und Funktionsweise des Datenladers

5.5.1 Die Komponenten und ihre Aufgaben

Das Klassendiagramm des Datenladers ähnelt stark dem des Schemaladers (Abbildung 5.3). Der Datenlader besteht aus sechs funktionellen Einheiten: Scanner, Parser, Objektpuffer, Metaklassenschema, Datenklassenschema und Hauptprogramm. Der *Scanner*

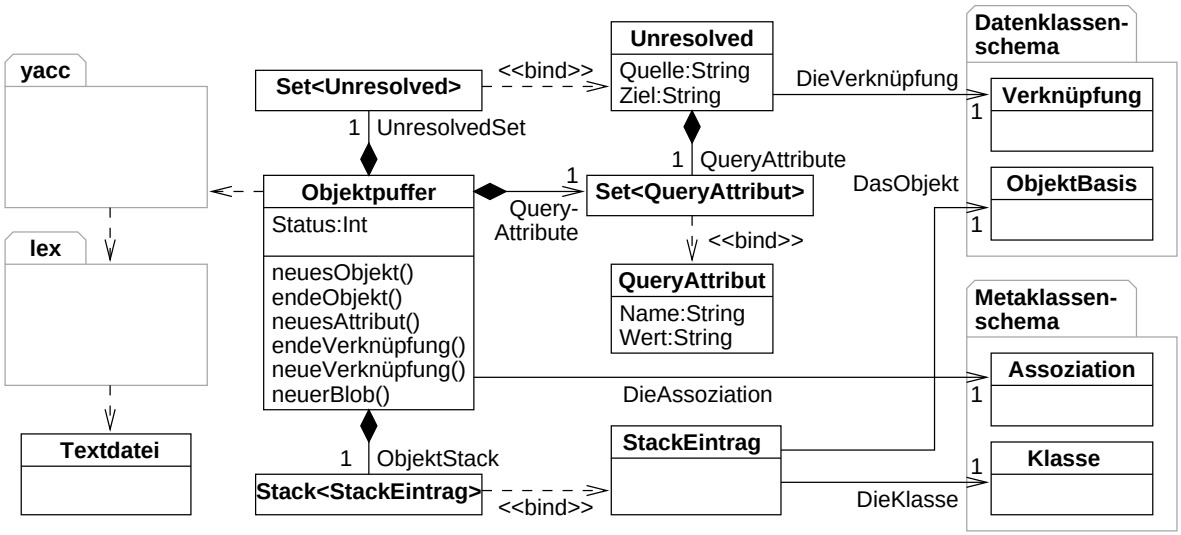

Abbildung 5.3: Klassendiagramm des Datenladers (vereinfacht)

und der *Parser* erfüllen im wesentlichen die gleichen Aufgaben wie die korrespondierenden Komponenten des Schemaladers. Das *Hauptprogramm* initialisiert den Parser und den Objektpuffer, stellt die Verbindung zur Datenbank her, startet eine Transaktion und öffnet nacheinander alle an der Kommandozeile des Programms angegebenen Quelldateien, deren Daten dem Parser übergeben werden. Der *Objektpuffer* speichert die vom Parser gelieferten Angaben zwischen und setzt die Datenobjekte, die Instanzen des *Datenklassenschemas* sind, sukzessive zusammen. Jedes komplett gelesene Objekt wird in die Extension seiner Klasse aufgenommen, die vom zugehörigen Kategorieobjekt verwaltet wird. Wie der Klassenpuffer des Schemaladers besitzt auch der Objektpuffer einen Stack, auf dem das jeweils aktuelle Objekt abgelegt wird. Ebenfalls vorhanden ist eine Kollektion `UnresolvedSet`, in der bisher nicht aufgelöste Verknüpfungen abgelegt werden. Das Set `QueryAttribute` speichert alle Attribute zwischen, die für eine Query benötigt werden, mit deren Hilfe das Zielobjekt einer Verknüpfung ermittelt wird.

Der wesentliche Unterschied zum Schemalader besteht darin, daß der Datenlader weniger syntaktische Überprüfungen vornehmen kann, da die Syntax der Daten-Definitionssprache unabhängig von jedem konkreten Datenschema ist. Aus diesem Grund muß der Datenlader eng mit dem Datenschema, das eine Instanz des *Metaklassenschemas* ist, zusammenarbeiten. Das Datenschema liefert die Antworten auf folgende Fragen:

- Ist die Klasse eines gelesenen Objekts definiert?
- Darf ein Objekt ein Attribut bestimmten Namens besitzen?
- Stimmt der Wert des Attributs mit dem vorgeschriebenen Wertebereich überein?
- Entspricht ein Komponentenobjekt in Name und Aufbau einer der Komponentenklassen des zugehörigen Klassenobjekts?
- Ist die zu einer Verknüpfung gehörende extensionale Assoziation für das zum Objekt gehörende Klassenobjekt definiert?
- Zu welcher extensionalen Klasse dürfen Objekte gehören, auf die eine Verknüpfung erstellt werden soll?
- Liegen die Kardinalitäten beider Enden einer Verknüpfung im zulässigen Bereich?

5.5.2 Verarbeitung von Sprachelementen

Prüfung des Objekttyps. Sobald der Name eines neuen Objekts gelesen ist, wird in der Menge der Klassenobjekte des Datenschemas nach einer Klasse mit diesem Namen gesucht. Existiert ein solches Klassenobjekt, wird es im Objektpuffer für die nachfolgenden semantischen Prüfungen gehalten und nach dem Lesen der Endmarkierung eines Objekts wieder entfernt.

Attribute. Für ein neu gelesenes Attribut wird geprüft, ob es eine Instanz der Klasse `MetaAttribut` mit gleichem Namen in der Menge der Attribute des zugehörigen Klassenobjekts gibt. Trifft dies zu, muß nur noch ermittelt werden, ob der Wert des Attributs innerhalb des Wertebereichs liegt, der vom `MetaAttribut` vorgeschrieben wird. Ist dieser Typ beispielsweise `ZAHL`, so muß der Attributwert numerisch sein.

Verknüpfungen. Verknüpfungen können, analog zu Assoziationen im Schemalader, sowohl auf bereits geladene Objekte als auch auf Objekte gerichtet sein, die später gela-

den werden. Das Zielobjekt einer Verknüpfung wird spezifiziert, indem in spitzen Klammern hinter dem Namen der Verknüpfung eine beliebige Anzahl von Attributnamen und -werten angegeben wird, die das Zielobjekt beschreiben. Der Datenlader erkennt am Typ der Verknüpfung, der Assoziation also, welche Teilbäume der Klassenhierarchie an der Assoziation beteiligt sind. Erstens wird so erkannt, ob ein Objekt eine Verknüpfung bestimmten Typs benutzen darf. Zweitens erfährt der Datenlader, in welcher Klassen-Extension er das Zielobjekt suchen muß. Werden mehrere Zielobjekte gefunden, bricht der Lader mit einem Fehler ab. Liegt kein Ergebnis vor, so wird das Verknüpfungsobjekt in der Kollektion `UnresolvedSet` des Objektpuffers abgelegt. Am Ende der Transaktion wird erneut versucht, die verbleibenden Verknüpfungen aufzulösen.

Kardinalität. Bevor eine Verknüpfung der Assoziation $\mathcal{A}$ erstellt wird, muß ermittelt werden, ob die maximale Kardinalität auf Quell- und Zielseite schon erreicht wurde. Wie die Kardinalitäten prinzipiell ermittelt werden können, wurde in Unterabschnitt 4.4.2 beschrieben. Der Datenlader prüft die Zielkardinalität einer Assoziation $\mathcal{A}$ von einer Klasse $\mathcal{K}_1$ auf eine Klasse $\mathcal{K}_2$, indem er für jedes Objekt $o \in \mathcal{K}_1$ ermittelt, wieviele Einträge in seiner Kollektion `Verknüpfungen` vorliegen, deren Variable `Name` den Wert "A" hat und deren Feld `ZielObjekt` auf ein Objekt der Klasse $\mathcal{K}_2$ zeigt. Um die Quellkardinalität der Assoziation $\mathcal{A}$ zu ermitteln, werden alle Objekte der tiefen Extension der Klasse $\mathcal{K}_1$ durchlaufen. Pro Quellobjekt werden alle Verknüpfungen der Assoziation $\mathcal{A}$ aus der Kollektion `Verknüpfungen` selektiert. Die Menge aller Verknüpfungen der Assoziation $\mathcal{A}$ wird nun nach dem Wert der Variable `ZielObjekt` partitioniert. Die Mächtigkeit jeder Partition muß im Intervall liegen, das durch die Quellkardinalität der Assoziation $\mathcal{A}$ vorgeschrieben wird.

Kompositionen. Ein Komponentenobjekt wird innerhalb des besitzenden Objekts angegeben und kann wiederum innere Objekte besitzen. Als Konsequenz daraus ergibt sich, daß der Lader den Zustand des äußeren Objekts zwischenspeichern muß, um nach dem Lesen des inneren Objekts wieder an der gemerkten Stelle fortfahren zu können. Zu diesem Zweck besitzt der Objektpuffer einen Stack mit dem Namen `ObjektStack`. Sobald der Anfang einer Objektdefinition gelesen ist, wird das neue Objekt auf dem Stack unter dem Referenznamen `DasObjekt` abgelegt. Es verbleibt dort solange, bis der Lader die Endmarkierung des Objekttextes erreicht. Attribute, Verknüpfungen und weitere Komponentenobjekte werden stets zum obersten Stack-Eintrag hinzugefügt.

5.6 Zusammenfassung

Ziel des Kapitels war der Entwurf der Grammatiken für eine Schema-Beschreibungssprache und eine Daten-Beschreibungssprache sowie die Entwicklung der zugehörigen Ladeprogramme.

Mit Hilfe der Schema-Definitionssprache können Datenschemata in leicht verständlicher Weise in Textform notiert werden. Die Sprache schöpft die Modellierungsfähigkeiten des Datenbankschemas voll aus und läßt gleichzeitig nur inhaltlich korrekte Datenschemata zu. Die Aufgabe des Schemaladers ist es, die Beschreibung eines Datenschemas zu interpretieren und die erforderlichen Klassenobjekte und Assoziationsobjekte auf der Basis des Metaklassenschemas zu erstellen. Neben der syntaktischen Überprüfung, die von einem durch Yacc generierten Parser vorgenommen wird, kontrolliert der Lader außerdem die inhaltliche Korrektheit des Datenschemas.

Die Daten-Definitionssprache liefert ein strukturiertes Format für den Import der Anwendungsdaten eines gegebenen Datenschemas. Da der Aufbau eines Datenschemas zur Übersetzungszeit noch nicht feststeht, ist die Syntax der Daten-Beschreibungssprache unabhängig von einem konkreten Datenschema. Daher muß der Datenlader wesentlich mehr semantische Überprüfungen in Zusammenarbeit mit dem bereits geladenen Datenschema durchführen. Beispiele dafür sind die Verifikation jedes Bezeichners im Datenschema oder die Kontrolle der minimalen und maximalen Kardinalität von Verknüpfungen. Hervorzuheben ist weiterhin, daß der Datenlader Binärdaten aufgrund der Dateiendung klassifiziert und als BLOBs in die Datenbank importiert.

6 Anbindung an das World Wide Web

6.1 Einleitung

Der Applikation zwischen Datenbank und Benutzerschnittstelle kommt in der Gesamtarchitektur eine besondere Rolle zu: Sie visualisiert Inhalt und Struktur der Anwendungsdaten und nimmt sämtliche Nutzeranfragen entgegen. Weil durch die Erzeugung von Hypertext-Dokumenten indirekt auch die Schnittstelle zum Anwender gestaltet wird, müssen beim Entwurf dieser Komponente neben technischen Aspekten, die in das Gebiet der Informatik fallen, ebenso gestalterische und ergonomische Gesichtspunkte beachtet werden. In diesem Kapitel wird allerdings ausschließlich auf die technische Seite der WWW-Integration eingegangen.

Zu Beginn wird noch einmal zusammenfassend erläutert, durch welche Eigenschaften das in Kapitel 4 entwickelte Datenbankschema in der Lage ist, sich selbst darzustellen, und wie diese Eigenschaften von einer Applikation genutzt werden können. Im Anschluß daran wird ein einfaches Modell vorgestellt, das auf der Grundlage des Datenbankschemas zur Darstellung der Anwendungsdaten dienen kann. Im darauffolgenden Abschnitt wird dieses Modell im Hinblick auf ein Informationssystem im World Wide Web konkretisiert. Am Ende des Kapitels wird eine Realisierung der WWW-Anbindung mit dem Programmpaket O2Web vorgestellt.

6.2 Selbstdarstellung des Datenbankschemas

Eines der wesentlichen Ziele beim Entwurf des Datenbankschemas war, die Anwendungsdaten mit der Fähigkeit der *Reflexivität* zu versehen. Eine Applikation muß in der Lage sein, den Datenbestand darzustellen, ohne vollständiges Wissen über dessen Struktur

zu besitzen. Dadurch werden die Abhängigkeiten zwischen Datenschema und Anwendung gering gehalten. Beide Komponenten können weitgehend unabhängig voneinander variiert oder ausgetauscht werden. Ermöglicht wird dies dadurch, daß die Implementierungsklassen lediglich generische Behälter für Objekte sind, deren genaue Struktur durch Klassenobjekte vorgeschrieben wird. Jedes Objekt des Datenbestandes speichert vollständige Informationen über seinen Aufbau und seine Relationen zu anderen Objekten. Im Detail sind es die folgenden Merkmale des Datenbankschemas, die zur Selbstdarstellung benutzt werden:

- Jedes Attribut speichert seinen Namen in mehreren Sprachen.
- Klassen- und Assoziationsnamen werden ebenfalls mehrsprachig in den zugehörigen Objekten bzw. Verknüpfungen abgelegt.
- Jedes Objekt kennt seine Komponentenobjekte und seine Verknüpfungen mit anderen Objekten.
- Kategorie-Objekte ermöglichen die Darstellung von Angaben, die alle Objekte einer Klasse gemeinsam haben, sowie die Visualisierung der Klassenhierarchie.
- Jedes Objekt besitzt eine Referenz auf seine Kategorie, so daß seine Einordnung in die Klassenhierarchie festgestellt werden kann.

Attribute, Verknüpfungen und Komponenten werden in *Kollektionen* verwaltet. Um beispielsweise alle Attribute eines Objekts zu erhalten, muß nur die Kollektion `Attribute` mittels eines Iterators durchlaufen und der Name und der Wert jedes Attributs angezeigt werden. Die Applikation benötigt kein externes Wissen über die Namen der Attribute, die auszugeben sind.

Für eine einfache Darstellung des Datenbestandes genügt es, jeder Implementierungsklasse eine Formatierungsfunktion zuzuordnen sowie einige Einstiegspunkte in die Datenbank festzulegen. Variationsmöglichkeiten bestehen bei der Gestaltung der Formatierungsfunktion, die bestimmt, in welcher Weise die Objektdaten angezeigt werden. Anspruchsvollere Anwendungen benötigen einige zusätzliche Module, die beispielsweise eine komfortable Navigation und Suche im Datenbestand gestatten (Abschnitt 6.4). Der folgende Abschnitt untersucht zunächst die Frage, auf welche Weise die Formatierungsfunktionen den Implementierungsklassen des Datenbankschemas zugeordnet werden können.

6.3 Grundmodell der Darstellung von Anwendungsdaten

Grundsätzlich gibt es zwei Möglichkeiten, das Datenbankschema mit Funktionalität zu versehen, die auf ein spezielles Ausgabemedium zugeschnitten sind:

1. Jede Implementierungsklasse wird mit Methoden angereichert, die Objekte, Verknüpfungen und deren Attribute darstellen.
2. Die Funktionalität für spezielle Ausgabemedien wird in einem separaten Klassenschema gekapselt. Objekte dieses Klassenschemas greifen über die Schnittstelle des Datenbankschemas auf die Anwendungsdaten zu.

Die Variante 2 entspricht etwa dem Entwurfsmuster des *Besuchers*, wie sie in [GHJV96, S. 269ff] beschrieben wird. Sie besitzt gegenüber der Variante 1 den Vorteil, daß das Datenbankschema und die Applikationen sauber getrennt sind. Somit ist die Variation der Darstellungsfunktionen möglich, ohne daß das Datenbankschema angepaßt werden muß. Selbst der Wechsel auf ein anderes Darstellungsmedium erfordert keinerlei Änderungen des Datenbankschemas. Schließlich ist in Variante 2 sogar der gleichzeitige Zugriff von Applikationen mit unterschiedlichen Darstellungsfunktionen möglich. In Variante 1 müßten die speziellen Formatierungsfunktionen jeder Applikation im Datenbankschema fest verankert werden.

Der Besucher in Variante 2 steht vor der Schwierigkeit, daß er den Typ der Klasse kennen muß, über deren Instanzen er eine Operation ausführen soll. In [GHJV96] wird das Problem dadurch gelöst, daß das zu besuchende Objekt O der Klasse K den Besucher B entgegennimmt (O.`NimmEntgegen(`B`)`) und die Methode B.`zeige_`K`(this)` aufruft. Durch die Polymorphie der Methode `NimmEntgegen()` wird stets die zur Klasse K passende Methode des Besuchers B ausgewählt. Der Nachteil dieser Variante besteht darin, daß die zu besuchenden Klassen explizit auf den Besucher vorbereitet werden müssen.

Abbildung 6.1 zeigt eine alternative Variante des Besuchers. Die Klasse `Besucher` definiert eine Methode `zeigeObjekt()`, welche die Darstellung aller zugeordneten Objekte übernimmt, wobei sie auf deren Daten über die öffentliche Schnittstelle zugreift. Der Besucher fragt den Typ der zu besuchenden Klasse zur Laufzeit ab und führt gegebenenfalls eine Typumwandlung durch, bevor er auf die Methoden der Klasse zugreift. Der

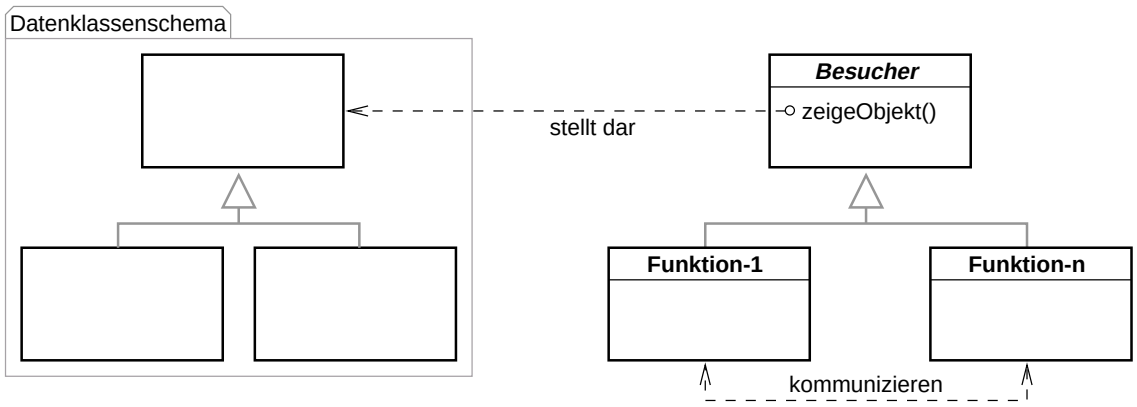

Abbildung 6.1: Grundmodell der Darstellung von Anwendungsdaten

Vorteil dieser Technik besteht darin, daß die zu besuchenden Klassen keinerlei Kenntnisse über einen potentiellen Besucher haben müssen. Dem steht der Nachteil der etwas uneleganten expliziten Typabfrage und -umwandlung entgegen.

Von der Besucherklasse sind Klassen abgeleitet, die jeweils eine spezielle Aufgabe erfüllen und dabei gegebenenfalls untereinander kommunizieren. Je nach Plattform erfolgt diese Kommunikation durch Methodenaufrufe, Botschaften oder, im Fall einer WWW-Anbindung, durch parametrisierte Hyperlinks.

6.4 Architektur der WWW-Anbindung

6.4.1 Überblick über die Komponenten

Beim Entwurf der WWW-Komponente müssen einige Besonderheiten beachtet werden, deren Ursache in den Beschränkungen des HTTP-Protokolls liegen. Wie schon in Abschnitt 4.6 erläutert wurde, ist das HTTP-Protokoll *zustandslos*. Jeder Zugriff auf einen HTTP-Server ist in sich geschlossen und ohne Bezug auf eine vorangegangene Anfrage. Ein serverseitiges Programm kann nie mit Bestimmtheit sagen, ob zwei aufeinanderfolgende Transaktionen tatsächlich zusammengehören. Das hat zwei Konsequenzen für die Architektur der WWW-Anbindung: Erstens müssen Dokumente immer *komplett* erzeugt werden. Jedes Objekt, das durch einen URL referenziert wird, muß folglich alle Bestandteile eines Dokuments, einschließlich des HTTP-Headers, erzeugen können. Da normalerweise jedes Dokument wiederkehrende Komponenten, wie ein Menü oder eine

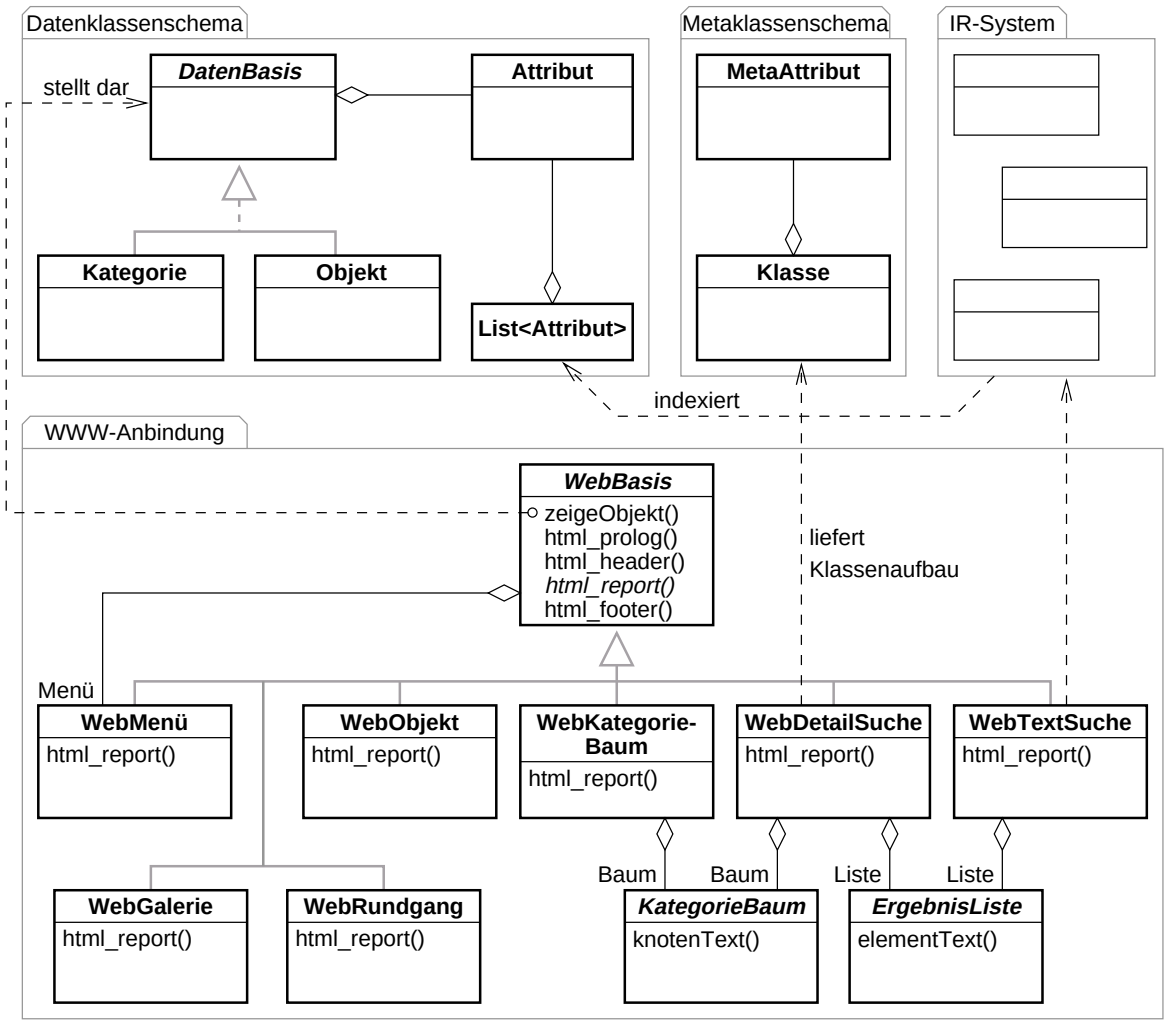

Abbildung 6.2: Komponenten der WWW-Anbindung (vereinfacht)

Kopfzeile hat, muß jedes Objekt auch diese Dokumentenbestandteile generieren können. Zweitens können Statusinformationen nur über URL-Parameter ausgetauscht werden[1]. Das bedeutet, daß *jedes* Objekt in der Lage sein muß, URL-Parameter zu interpretieren und geeignet zu reagieren. Beispielsweise wird die aktuelle Sprache sowie die Qualitätsstufe und das Datenformat der Binärdaten per URL an das Zielobjekt weitergereicht, das diese Parameter für die Darstellung der Anwendungsdaten verwendet.

[1] Eine Alternative dazu besteht im Einsatz von Cookies - einem Mechanismus, der es dem Server erlaubt, Daten auf der Seite des Clients abzulegen und abzurufen. Der Einsatz von Cookies gilt allgemein als unsaubere Technik und als Sicherheitsrisiko. Deshalb verbieten viele Nutzer das Abspeichern von HTTP-Serverdaten auf ihrem Rechner.

Die genannten Gründe sprechen für eine relativ umfangreiche Basisklasse, welche die übergreifend benötigte Funktionalität kapselt. Wie Abbildung 6.2 zeigt, werden von dieser Basisklasse Unterklassen abgeleitet, die jeweils eine bestimmte Aufgabe erfüllen. Dazu gehört die Darstellung des Kategoriebaums sowie die Text- und die Detailsuche. Da fast alle Unterklassen auch Datenobjekte darstellen müssen, implementiert die Basisklasse zusätzlich die Aufgabe des Besuchers und vererbt diese Fähigkeit an ihre Unterklassen.

Jede nicht abstrakte Klasse in Abbildung 6.2, deren Name mit dem Präfix `Web` beginnt, besitzt genau eine Instanz, die durch einen eindeutigen Namen gekennzeichnet ist. Über diesen Namen, der ebenfalls als URL-Parameter angegeben wird, kann ein `Web`-Objekt durch den HTTP-Server angesprochen werden. Jede Unterklasse von `WebBasis` besitzt die speziellen Methoden `html_prolog()`, `html_header()`, `html_report()` und `html_footer()`, die in dieser Reihenfolge aufgerufen werden und HTML-Text erzeugen. Die Namen der Methoden entsprechen den Konventionen des Programmpakets O2Web, auf das im Abschnitt 6.5 eingegangen wird.

Neben den unmittelbar zur Darstellung dienenden Klassen umfaßt die WWW-Komponente die Hilfsklassen `KategorieBaum` und `ErgebnisListe`, deren Funktionen in verschiedenen Kontexten benötigt werden. In den folgenden Unterabschnitten werden die einzelnen Komponenten des Klassendiagramms näher beschrieben.

6.4.2 Die Basisfunktionen

Rahmenerzeugung

Die abstrakte Klasse `WebBasis` implementiert sämtliche Funktionen, die für *jedes* zu generierende Dokument benötigt werden. Dazu gehört vor allem die Erstellung eines Rahmens, der für alle Teilaufgaben weitgehend konstante Gestalt besitzt. Neben dekorativen Elementen gehören zum Rahmen vor allem das Auswahlmenü sowie die Verwaltung der History, die im nächsten Unterabschnitt beschrieben wird.

Jede Instanz einer Unterklasse von `WebBasis` ruft nach der Ausgabe des HTTP-Headers durch die Funktion `html_prolog()` die Methode `html_header()` auf, die den Rahmen öffnet. In Abhängigkeit von der aktuellen Sprache wird dann das Auswahlmenü mit Verweisen auf die Objekte erstellt, welche die Funktionen des Informationssystems kap-

seln. Jedem dieser Objekte müssen alle Parameter weitergereicht werden, die den aktuellen Zustand des Systems charakterisieren. Das Auswahlmenü wird von der Klasse `WebMenü` erstellt, die der Klasse `WebBasis` per Aggregation zugeordnet ist. Gleichzeitig erbt sie die Eigenschaften der Basisklasse und kann daher vollständige HTML-Dokumente generieren. Somit kann das Menü entweder in jedes Dokument eingebettet werden oder alternativ als eigenständiges Objekt ein komplettes Menü-Dokument erzeugen. Das Menü-Dokument kann vom Hauptbereich der Darstellung abgekoppelt werden und in einem Frame oder in einem separaten Browser-Fenster ausgegeben werden. Der Rahmen wird durch die Methode `html_footer()` geschlossen.

History-Funktion

Die Aufgabe einer History-Funktion ist es, den Zustand eines Systems an definierten Zeitpunkten zwischenzuspeichern. Der Zustand wird durch Nutzereingaben verändert, die im Fall eines WWW-Informationssystems aus dem Anklicken von Hyperlinks oder dem Abschicken von Formularen bestehen. Der History-Mechanismus konkurriert mit der History-Funktion des WWW-Browsers. Allerdings hat die serverseitige Aufzeichnung von Zuständen aus folgenden Gründen ihre Berechtigung: Erstens zeichnet die History nur die für die Anwendung *relevanten* Zustände auf, während der Browser *jede* Änderung protokolliert. So führt beispielsweise die Umschaltung des Audio-Datenformats zu keiner inhaltlichen Änderung in der Darstellung und wird vom Browser, nicht aber von der History der WWW-Anbindung protokolliert. Zweitens besitzt ein dynamisch generiertes Dokument meist einen konstanten Titel, z.B. den Namen eines Museums. Das führt dazu, daß unterschiedliche Dokumente in der History des Browsers nicht mehr unterscheidbar sind.

Zu einer History gehört eine Liste gespeicherter Zustände, einschließlich eines Namens für jeden Zustand, und ein Mechanismus zum Wiederherstellen eines gespeicherten Zustands. Wie schon mehrfach erwähnt wurde, können Statusinformationen nur über URL-Parameter an das aufzurufende Objekt weitergereicht werden. Damit das Zielobjekt eine vorangegangene HTML-Seite erneut erzeugen kann, sind die folgenden Angaben nötig:

- der Pfadanteil des URLs, der das betreffende Dokument addressiert
- alle Parameter dieses URLs
- eine Kurzbeschreibung des Dokuments für die Anzeige der History

Wird die Instanz einer von `WebBasis` abgeleiteten Klasse zum Erstellen eines HTML-Dokuments aufgerufen, so ist sie dafür zuständig, an jeden Hyperlink, den sie in das neue Dokument einbettet, alle Parameter der bisherigen History sowie alle Informationen, die zur Beschreibung des aktuellen Dokuments nötig sind, anzuhängen. Überschreitet die Anzahl der History-Einträge einen festen Wert, so werden die ältesten Einträge entfernt. Die Methode `html_footer()` hat die Aufgabe, für jeden History-Eintrag einen Link anzuzeigen, mit dem der betreffende Zustand wiederhergestellt werden kann.

6.4.3 Assistierende Klassen

Hierarchie der Kategorien

In Abschnitt 4.5 wurde erläutert, daß Kategorien zur Visualisierung von extensionalen Klassen dienen und daß sie insbesondere die Klassenhierarchie durch Verknüpfungen zwischen den Kategorieobjekten nachbilden.

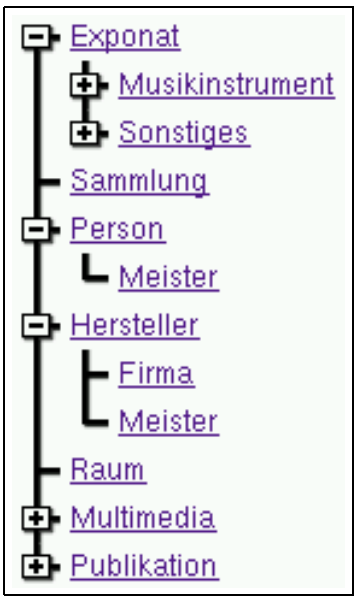

Abbildung 6.3: Kategoriebaum

In einem Informationssystem läßt sich die Visualisierung der Klassenhierarchie für wenigstens zwei Anwendungen nutzen: Erstens kann eine Kategorie als Einstiegspunkt für das Traversieren des Datenbestandes dienen, da sie Verweise auf alle Objekte der zugehörigen extensionalen Klasse sowie deren Unterklassen speichert. Zweitens kann sie als Ausgangspunkt für eine Suche in allen zur Klasse gehörenden Objekten genutzt werden. Auf diesen Anwendungsfall wird in Unterabschnitt 6.4.5 näher eingegangen. Da durch die Instanzen der Klasse `Kategorie` des Datenklassenschemas alle Angaben geliefert werden, die zur Darstellung der Kategoriehierarchie nötig sind, kann diese Hierarchie völlig unabhängig von einem konkreten Datenschema visualisiert werden. Die Hierarchie wird durch einen *Baum* dargestellt. Der Spezialfall der Mehrfachvererbung wird dadurch aufgelöst, daß eine extensionale Klasse, die mehrere Väter hat, *jedem* Vater zugeordnet wird und deshalb mehrfach im Baum erscheint. Bei dem in Abbildung 6.3 dargestellten Beispielbaum trifft das auf die Klasse Meister zu, die sowohl von Person als auch von Hersteller

abgeleitet ist.

Um auch bei sehr tiefen Klassenhierarchien die Übersichtlichkeit der Darstellung zu bewahren, können Teilbäume versteckt werden. Zu diesem Zweck besitzt jeder Knoten, der kein Blatt ist, ein Schaltfeld, mit dem der zugehörige Teilbaum alternierend versteckt oder expandiert werden kann. Der Baum besitzt eine initiale Darstellungstiefe, die sich aus der Anzahl der Kategorien und dem maximalen Verzweigungsgrad errechnet.

Die Funktionalität des Kategoriebaums wird in der abstrakten Klasse `KategorieBaum` gekapselt. Eine abgeleitete Klasse muß die Methode `knotenText()` überschreiben. Durch diese Methode wird festgelegt, wie der Knoten zu beschriften ist und welche Funktion aufgerufen werden soll.

Der Kategoriebaum wird durch eine HTML-Tabelle dargestellt, deren Struktur durch rekursives Durchlaufen der Kategoriehierarchie erzeugt wird.

Ergebnisliste

Ergebnisse von Suchanfragen und andere Zusammenstellungen von Verweisen auf Objekte werden in Listenform ausgegeben. Häufig sind mehr Einträge in einer Liste vorhanden, als sinnvollerweise dargestellt werden können. In diesen Fällen wird nur eine *Partition* der Gesamtmenge angezeigt. Jede momentan nicht dargestellte Partition ist über einen Hyperlink erreichbar. Problematisch ist das Weiterreichen des Zustandes beim Umschalten zu einer anderen Partition. Zum einen kann dies durch Weitergabe der gesamten Ergebnisliste sowie der Partitionsnummer als URL-Parameter erfolgen. Zum anderen kann der URL, dessen Aufruf zur Erzeugung der Ergebnisliste geführt hat, stets mitgeführt werden. Nachteilig an der zweiten Variante ist, daß eine Suchanfrage bei jeder Partitionsumschaltung erneut ausgeführt werden muß. Dieser Nachteil muß jedoch in Kauf genommen werden, da die erste Variante ein nicht akzeptables Risiko birgt: Durch große Ergebnislisten kommt es zu überlangen URLs, die einen Speicherüberlauf des HTTP-Servers herbeiführen können [Apa98].

Die abstrakte Klasse `ErgebnisListe` implementiert die gesamte Logik, die zur Verwaltung von Ergebnislisten benötigt wird. Sie kann durch Überschreiben der Methode `elementText()` von den darstellenden Klassen angepaßt und genutzt werden.

6.4.4 Darstellung von Datenobjekten

Die Methode `zeigeObjekt()` der abstrakten Klasse `WebBasis` übernimmt die Visualisierung der Objekte des Datenklassenschemas. Jede abgeleitete Klasse kann diese Methode benutzen und gegebenenfalls anpassen. Dabei ist besonders die Klasse `WebObjekt` zu erwähnen, deren Funktion lediglich darin besteht, eine Anfrage auszuwerten, das gewünschte Datenobjekt zu selektieren und anschließend darzustellen. Die Instanz dieser Klasse wird hauptsächlich als Zielobjekt für Hyperlinks genutzt, die Verküpfungen zwischen Datenobjekten repräsentieren.

Für die Darstellung eines beliebigen Datenobjekts muß lediglich der Aufbau seiner Implementierungklasse bekannt sein; externe Informationen über die konkrete Objektstruktur werden nicht benötigt. Die Visualisierung eines Datenobjekts erfolgt in maximal sechs Schritten, die in beliebiger Reihenfolge ausgeführt werden können:

Klassenname: Jedes Datenobjekt speichert den Namen seiner Klasse in mehreren Sprachen. Obwohl der Name vor allem zur Formatierung von Ergebnislisten benötigt wird, kann er auch zur Darstellung des Objekts benutzt werden, um dem Benutzer eine leichtere Einordnung der angezeigten Daten zu ermöglichen.

Kategorie: Die Informationen zur Klassenzugehörigkeit eines Objekts können dadurch erweitert werden, daß ein Verweis auf die Kategorie des Objekts angezeigt wird.

Attribute: Attribute werden durch Iteration der Kollektion `Attribute` dargestellt. In Abhängigkeit von der Sprache wird der Attributname und der zugehörige Wert ausgegeben. Jedes Attribut kennt außerdem seinen Datentyp, der gegebenfalls zur Formatierung herangezogen werden kann.

Komponenten: Die Kollektion `Komponenten` referenziert alle Teile des aktuellen Objekts. Komponenten werden *am Ort* expandiert. Dies geschieht durch Aufruf der Formatierungsroutine des Teilobjekts und Einfügen des erzeugten HTML-Textes in den des Aggregats. Der HTML-Standard gestattet es, sowohl Listen als auch Tabellen zu verschachteln. Die maximale Tiefe einer Ganzes-Teile-Hierarchie hängt somit ausschließlich von den Darstellungsmöglichkeiten des Browsers ab.

Verknüpfungen: Über die Kollektion `Verknüpfungen` sind alle Verknüpfungen erreichbar, die vom aktuellen Objekt ausgehen. Wenn die Verknüpfung zu einer Aggregation gehört, wird sie am Ort expandiert. Alle anderen Verknüpfungen werden

als Hyperlink dargestellt. Die `ID` des Zielobjekts wird als Parameter des URLs benutzt, über den das Verknüpfungsziel erreicht werden kann. Der Name der Verknüpfung, der ebenfalls mehrsprachig vorliegt, dient zur Beschriftung des Links. Verknüpfungen können Attribute haben, die genauso wie Objektattribute formatiert werden.

Multimediale Daten: Jede Instanz der Implementierungsklasse `Multimedia` speichert Referenzen auf multimediale Objekte in der Kollektion `Blobs`. Anhand eines URL-Parameters wird das geeignete BLOB ausgewählt. Falls zu der gewünschten Qualitätsstufe keine Binärdaten gespeichert sind, wird automatisch das BLOB der Qualität `GERING` gewählt. Dessen Existenz wird vom Datenlader garantiert. Die Darstellung von Blobs erfolgt weitgehend generisch: Durch das Feld `MIME` der Klasse `Blob` wird dem Browser der Typ der Daten im HTTP-Vorspann mitgeteilt (siehe Abschnitt 4.6). Der Browser ordnet nun die Daten selbst einem geeigneten Ausgabegerät zu. Die Aufgabe der Dokumentenerzeugung ist lediglich zu entscheiden, ob es sich um Bilddateien handelt, die über den `<IMG>`-Tag in den HTML-Text eingesetzt werden, oder um Daten beliebigen anderen Formats, für die ein Hyperlink generiert wird.

Instanzen der Implementierungsklasse `Kategorie` können zusätzlich Verweise auf ihre Ober- und Unterkategorien darstellen. Mit Hilfe einer Instanz der Klasse `ErgebnisListe` kann außerdem die Extension der zu dieser Kategorie gehörenden Klasse aufgelistet werden.

6.4.5 Suche

Die WWW-Anbindung stellt zwei grundlegend verschiedene Formen der Suche bereit: Suche mit Kenntnis der Klassenstruktur sowie Suche in allen Texten des Datenbestandes. In diesem Unterabschnitt wird nur die strukturierte Suche betrachtet; die Textsuche ist Gegenstand des 7. Kapitels.

Die strukturierte Suche nutzt Informationen über den Aufbau der Objekte einer extensionalen Klasse aus, um ein Suchformular zu erstellen, das wiederum als Grundlage für eine Query dient. Insgesamt wird eine strukturierte Suche in vier Schritten ausgeführt:

1. Der Nutzer bekommt die Hierarchie der Kategorien angezeigt. Aus dieser Hierarchie wählt er die gewünschte Kategorie aus, welche die Klasse repräsentiert, in deren Objektmenge gesucht werden soll.
2. Aus dem Datenschema wird das zu der ausgewählten Kategorie gehörende Klassenobjekt selektiert. Das Klassenobjekt speichert den Aufbau aller Objekte der extensionalen Klasse. Aus diesen Strukturinformationen wird ein HTML-Formular erstellt, indem für jedes Attribut ein Eingabefeld angezeigt wird, das mit dem Attributnamen beschriftet ist.
3. Der Nutzer füllt einige Felder aus und schickt das Suchformular ab. Aus den Namen und dem Inhalt der ausgefüllten Felder wird eine Query generiert und über der Extension der Klasse ausgeführt. Falls jede Kategorie die tiefe Extension ihrer Klasse speichert, genügt *eine* Anfrage. Anderenfalls müssen die flachen Extensionen aller Unterkategorien ebenfalls durchsucht und die Ergebnismengen zusammengefaßt werden.
4. Die Objekt-Namen aller Resultate werden dem Nutzer mit Hilfe einer Instanz der Klasse `ErgebnisListe` ausgegeben. Jeder Listeneintrag verweist über die Objekt-ID auf das zugehörige Objekt, das per Mausklick angezeigt werden kann.

6.5 Realisierung mit O2Web

6.5.1 Aufbau und Funktionsweise von O2Web

O2Web ist ein Programmpaket, mit dem ein O2-Datenbank-Server an das WWW angebunden werden kann. Es besteht aus mehreren Bibliotheken, die eine Programmierschnittstelle bereitstellen, sowie drei Programmen, die im Zusammenspiel mit einem Standard-HTTP-Server zum Einsatz kommen:

- O2Web-Gateway
- O2Web-Dispatcher
- O2Web-Server

Das O2Web-Gateway ist ein Programm, das über die CGI-Schnittstelle des HTTP-Servers aufgerufen wird, wenn ein zugehöriger URL angefordert wurde. Das Gateway

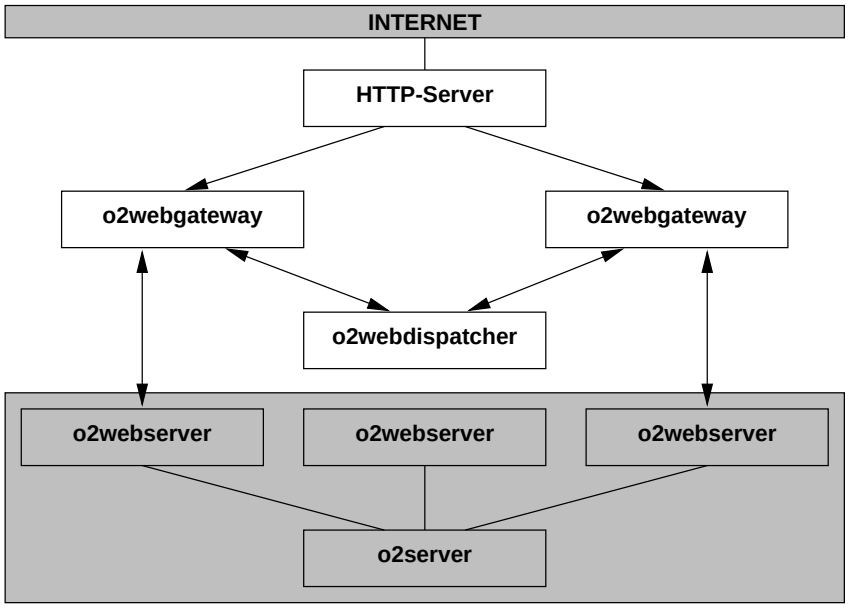

Abbildung 6.4: O2Web-Architektur (aus [O2W96])

nimmt via *Remote Procedure Call (RPC)* Kontakt zum O2Web-Dispatcher auf. Der Dispatcher kennt jeden O2Web-Server im lokalen Netz und teilt dem Dispatcher dessen Addresse mit. Anschließend verbindet sich das Gateway mit dem ermittelten O2Web-Server, stellt ihm den URL zu und leitet die Ausgaben des O2Web-Servers an die CGI-Schnittstelle weiter. Abbildung 6.4 zeigt die O2Web-Architektur.

Ein O2Web-Server läuft als Dämon-Prozeß auf einem beliebigen Rechner im lokalen Netzwerk. Er wird vom Programmierer unter Einbeziehung der O2Web-Bibliotheken erstellt und beherbergt die vollständige Anwendungslogik. Beim Starten des O2Web-Servers wird die Verbindung zu einem O2-Datenbank-Server erstellt, indem eine *Session* gestartet wird. Anschließend tritt der O2Web-Server in eine Endlosschleife. Durch den permanenten Kontakt zum Datenbank-Server wird eine sehr gute Performance bei Transaktionen erreicht.

Zum Zugriff auf eine Datenbank unter O2Web werden OQL-Queries benutzt, die als Parameter an den URL angehängt werden. Der URL kodiert außerdem den Namen des Systems (O2-Server) und den Namen der logischen Datenbank. Ein URL zum Zugriff auf eine O2Web-Datenbank hat somit folgende Struktur:

```
http://host[:port]/cgi-path/o2webgateway/systemname/basename?query
```

Ein persistenter Name ist eine spezielle OQL-Query. So verweist der URL

```
http://.../cgi-bin/o2webgateway/generic/mim?WebBasis
```

auf das persistente Objekt mit dem Namen "WebBasis", das sich in der logischen Datenbank "mim" des O2-Servers "generic" befindet.

Jedes Objekt in der Ergebnismenge einer Query stellt sich selbst dar, indem es auf der Grundlage der Objektstruktur HTML-Text erzeugt. O2Web kennt einen *generischen* Modus, der jedes komplexe Objekt rekursiv zerlegt und den Grundtypen geeignete HTML-Tags zuweist. Listen werden beispielsweise durch `<UL>`- Tags dargestellt; Referenzen auf Objekte durch Hyperlinks. Ein fortgeschrittenerer Modus gestattet *globale* Anpassungen des zu erzeugenden HTML-Textes. So kann z.B. jedes Dokument mit einem konstanten Kopf- und Fußteil versehen werden. Der dritte Modus schließlich erlaubt die *lokale* Anpassung der HTML-Generierung jedes Objekts. Der Programmierer hat nun die volle Kontrolle über den HTML-Text, der für ein Objekt erzeugt wird. Erreicht wird dies durch das Überschreiben der Methoden `html_prolog()`, `html_header()`, `html_report()` und `html_footer()` des Objekts. Der O2Web-Server ruft diese Methoden für das Ergebnisobjekt einer Query in der angegebenen Reihenfolge auf und sendet den generierten HTML-Text an das Gateway. Das Gateway reicht das generierte Dokument über die CGI-Schnittstelle an den HTTP-Server weiter, der es schließlich an den anfordernden Browser sendet.

HTML-Formulare verlangen eine etwas andere Herangehensweise. Der URL, der beim Abschicken des Formulars aufgerufen wird, muß eine *Methode* eines persistenten Objekts aufrufen und darf neben der OQL-Query keine weiteren Parameter besitzen. Die Parameter müssen stattdessen als versteckte Formular-Felder kodiert werden, deren Name und Inhalt zusammen mit den anderen Feldern an die Methode übergeben werden. Die Methode erzeugt den gesamten HTML-Text des Dokuments einschließlich des HTTP-Headers.

6.5.2 Integration des WWW-Klassenschemas in O2Web

Um das in Abbildung 6.2 dargestellte Klassenschema in das Programmpaket O2Web zu integrieren, sind im wesentlichen zwei Anpassungen nötig:

- Von jeder darstellenden Klasse, also jeder Klasse, deren Name mit der Vorsilbe `Web` beginnt, wird genau ein *persistentes* Objekt erzeugt, dessen Name mit dem Namen seiner Klasse übereinstimmt. Mit Hilfe des persistenten Namens kann das gewünschte Objekt durch einen URL angesprochen werden.
- Jeder Hyperlink, der in ein dynamisch erzeugtes Dokument eingefügt wird, muß den Konventionen von O2Web genügen. So muß der URL-Parameter `query` ein einzelnes Objekt oder eine Methode eines persistenten Objekts selektieren. O2Web verlangt weiter, daß alle zusätzlichen Parameter in einer genau definierten Weise an den URL angehängt werden.

Die persistenten Objekte werden in der Startphase des O2Web-Servers erzeugt, der anschließend in eine Endlosschleife tritt und auf die Anfragen des O2Web-Gateways wartet.

6.6 Zusammenfassung

In diesem Kapitel wurde der Aufbau einer Applikation vorgestellt, welche die Fähigkeiten des Datenbankschemas ausnutzt, sich selbst darzustellen. Die WWW-Schnittstelle ist damit vollständig unabhängig von einem konkreten Datenschema. Zwischen dem Datenbankschema sowie der WWW-Komponente existieren nur wenige Abhängigkeiten, da die für die Dokumentenerzeugung zuständigen Klassen mit den Klassen des Datenbankschemas über deren sehr schlanke Methodenschnittstelle kommunizieren. Somit kann die Darstellung ohne Auswirkungen auf das Datenbankschema variiert werden.

Die Methode zur Darstellung von Datenobjekten, die in einer Basisklasse definiert ist, wird von sieben abgeleiteten Klassen genutzt, die jeweils eine Grundfunktion der WWW-Anbindung kapseln. Dazu gehören beispielsweise die Darstellung der Hierarchie der Kategorien sowie die Suche mit Kenntnis der Struktur einer extensionalen Klasse. Die assistierenden Klassen `KategorieBaum` und `ErgebnisListe` stellen die Basisfunktionen zur Darstellung von Bäumen und Listen bereit.

Die Realisierung der WWW-Anbindung erfolgte mit dem Programmpaket O2Web, das den Zugriff auf Objekte der Datenbank über URLs erlaubt sowie Hilfsmittel für den Programmierer zum Entwickeln von datenbankbasierten WWW-Applikationen bereitstellt.

7 Information Retrieval

7.1 Einleitung: Warum ein Information-Retrieval-System?

Nahezu alle modernen DBMS besitzen mächtige und komfortable Anfragesprachen. Bei relationalen Datenbanken hat sich der Sprachstandard SQL durchgesetzt. Für objektorientierte DBMS bemüht sich die *Object Data Management Group (ODMG)* um die Standardisierung der Anfragesprachen. Die aktuelle Revision 2.0 des ODMG-Standards definiert die Anfragesprache *OQL (Object Query Language)*, die von einigen kommerziellen OODBMS, darunter O2, unterstützt wird [Sch97].

Angesichts datenbankeigener Anfragemöglichkeiten stellt sich die Frage nach dem Sinn eines Information-Retrieval-Systems (IRS) als zusätzlicher Suchmöglichkeit in der Datenbank. In Kapitel 2 wurde bereits erwähnt, daß der größte Teil der Daten in einem Informationssystem Textform hat. Über ein IRS können daher nahezu alle Daten erreicht werden. Das IRS soll die datenbankeigene Anfragesprache nicht ersetzen, sondern eine von zwei sich ergänzenden Alternativen darstellen, zwischen denen der Nutzer frei wählen kann.

Im folgenden sollen wichtige Eigenschaften von DBMS und IRS gegenübergestellt werden, um die Vor- und Nachteile der Systeme, bezogen auf den Anwendungsbereich Information Retrieval, zu untersuchen (nach [vR79]).

Übereinstimmung: DBMS liefern nur Ergebnisse, die *exakt* zur Anfrage passen; IRS finden auch Resultate mit *teilweiser* Übereinstimmung bzw. das Ergebnis, das *am besten* zutrifft. Suchergebnisse können sortiert nach ihrer Relevanz ausgegeben werden (Ranking).

Schlußweise: IRS arbeiten *induktiv*: Der Aufbau *jedes einzelnen* Dokuments wird ausgewertet, um daraus einen Index zusammenzusetzen, der für *alle* Dokumente gilt. Queries in Datenbanken werden *deduktiv* abgearbeitet: Das Datenbankschema bestimmt gleichzeitig die Struktur der Daten und den Aufbau der Queries. Der Einzelfall (das Suchergebnis) wird vom Allgemeinen (dem Schema) abgeleitet.

Modell: Anfragen in Datenbanken werden streng *deterministisch* ausgewertet, während in IRS nur *Annahmen* über die Relevanz von Dokumenten im Hinblick auf eine Suchanfrage gemacht werden können.

Klassifikation: In Datenbanken ist meist eine *monothetische* Klassifikation von Dokumenten anzutreffen. Jedes Tupel gehört zu genau einer Relation, jedes Objekt zu genau einer Klasse. In IRS ist meist eine *polythetische* Klassifikation gewünscht, das heißt, ein Dokument kann mehreren Klassen angehören.

Anfragesprache: Viele IRS unterstützen Anfragen in *natürlicher* Form, d.h. als Aneinanderreihung von Suchworten ohne Strukturinformationen. Dies trifft unter anderem auf IRS zu, die auf dem Vector-Space-Model basieren [Sal71]. Anfragesprachen von DBMS sind hingegen stets *strukturiert*.

Spezifizierung der Anfrage: Eine Schlußfolgerung aus den unterschiedlichen Formen der Übereinstimmung zwischen Anfrage und Ergebnis ist, daß eine Query in einem DBMS das Resultat *exakt* spezifizieren muß, während in einem IRS nur eine *ungefähre* Beschreibung genügt.

7.2 Die Information-Retrieval-Bibliothek Isearch

Eines der frühesten populären Information-Retrieval-Systeme des Internets war WAIS [LPJN94]. WAIS hatte einen monolithischen Aufbau. Suchmaschine und Kommunikationsprotokoll waren eng verschmolzen, so daß das System schwer anpaßbar und erweiterbar war. Aus diesem Grund begannen N. Nassar und K. Gamiel im Jahre 1994 mit der Entwicklung des modularen Information-Retrieval-Systems Isite, das aus einer Implementierung des Protokolls Z39.50 sowie der Suchmaschine Isearch besteht [Nas97]. Isearch setzt sich zusammen aus einer flexiblen, erweiterbaren C++-Klassenbibliothek und sogenannten *Shells*, die die Schnittstelle zum Nutzer bilden.

lsearch basiert auf dem Vector-Space-Model. Anfragen lassen sich sowohl natürlichsprachlich als auch in boolescher Notation formulieren. Suchterme können optional mit Gewichten versehen werden, durch die der Nutzer die Bedeutung des Terms in der Anfrage hervorhebt. Als Beispiel soll eine Suche im Index `EXPONATE` dienen:

```
Isearch -d EXPONATE Violine:3 Zither
```

Violinen werden in diesem Beispiel mit dreifach höherem Gewicht gegenüber dem Normalwert belegt.

Das folgende Beispiel zeigt eine Anfrage in boolescher Notation:

```
Isearch -d EXPONATE -infix "(Violine or Zither) andnot Klavier"
```

lsearch unterstützt in eingeschränktem Maße die Trunkierung von Suchworten. Durch Anhängen des Zeichens '*' an eine Zeichenkette ω werden alle Dokumente gefunden, die Worte mit dem Präfix ω enthalten.

Ein mächtiges Werkzeug der Bibliothek lsearch sind Dokumententypen. Mit Hilfe eines Dokumententyps lassen sich Texte in logische Einheiten untergliedern. Die Suche nach einem Term kann auf einen bestimmten Textteil eingegrenzt werden. Im folgenden Beispiel werden alle HTML-Dateien im aktuellen Verzeichnis indexiert. Der Parameter `-d` schreibt den Dokumententyp "SGMLTAG" vor, der die Dokumente vor dem Indexieren anhand ihrer Struktur zerlegt. Der zweite Befehl sucht nach dem Wort "Datenbanken" im Titel der Dokumente:

```
Iindex  -d WEB -t SGMLTAG *.html
Isearch -d WEB title/Datenbanken
```

Die einzelnen Dokumententypen sind in C++-Modulen mit definierter Schnittstelle gekapselt. Dadurch ist es möglich, die Hierarchie bestehender Dokumententypen um neue Typen zu ergänzen.

7.3 Anpassung und Integration von Isearch

7.3.1 Abstraktion vom Zugriff auf Speichermedien

Trotz ihres modularen und flexiblen Aufbaus ist die Bibliothek Isearch auf das Indexieren und Durchsuchen von *Dateien* beschränkt. Jede C++-Klasse, die lesend auf ein Dokument zugreift, ruft eigenständig die dazu nötigen Funktionen des Betriebssystems auf. Aus diesem Grund ist es nicht möglich, andere Speichermedien, wie Datenbanken oder über ein Netzwerk erreichbare Datenträger, anzusprechen. Abbildung 7.1 zeigt schematisch den über mehrere Klassen verteilten Dateizugriff.

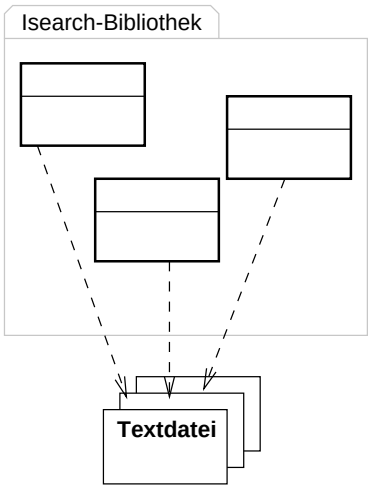

Abbildung 7.1: Isearch - Originalversion

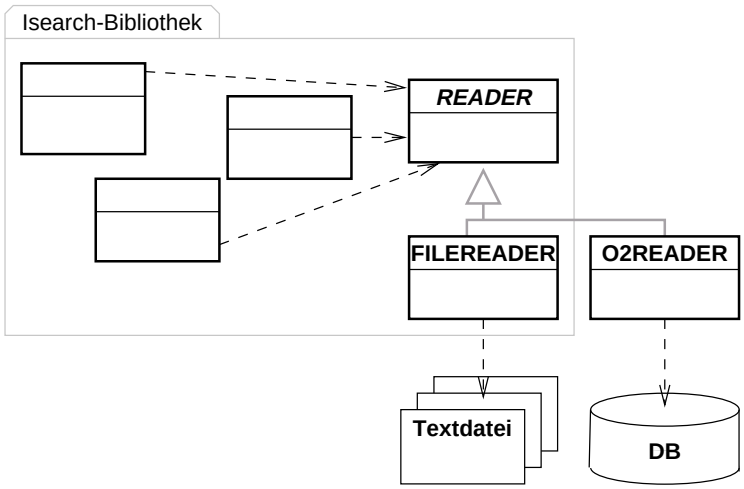

Abbildung 7.2: Isearch - angepaßte Version

Der Nachteil der fehlenden Transparenz beim Zugriff auf einen Datenträger wird durch eine Zwischenschicht aufgehoben, die von einem konkreten Speichermedium abstrahiert (Abbildung 7.2). Die abstrakte Klasse `READER` stellt eine Schnittstelle zur Verfügung, die von allen Klassen der Bibliothek zum Zugriff auf ein Speichermedium genutzt werden kann. Der Quelltext dieser Klasse ist im Anhang D.3.4 zu finden. Dem Konstruktor jeder Klasse, die Zugriff auf einen Datenspeicher benötigt, wird beim Programmstart eine Instanz einer von `READER` abgeleiteten Klasse übergeben. Die Klasse `FILEREADER` implementiert die Funktionalität zum Zugriff auf Dokumente, die im Dateisystem abgelegt sind. Die Klasse `O2READER` dient zum Zugriff auf die O2-Datenbank, welche die

Anwendungsdaten verwaltet. Analog dazu sind weitere Spezialklassen zum Lesen anderer Speichermedien denkbar. Voraussetzung dafür ist, daß sich der wahlfreie Zugriff auf den Datenspeicher auf die Operationen `open()`, `close()`, `read()`, `tell()` und `seek()` abbilden läßt. Zusätzlich muß sich ein Dokument eindeutig über eine Pfadangabe identifizieren lassen.

Im Datenbankschema, das in Kapitel 4 entwickelt wurde, ist jedes Objekt über eine eindeutige Nummer (ID) ansprechbar. Außerdem befindet sich jedes Objekt in mindestens einer Kollektion. Folglich ist es immer möglich, ein Objekt durch einen Pfadausdruck der Gestalt

```
/collection_name/objekt_id
```

zu identifizieren. Zusätzlich ist jedes Objekt, das Textattribute besitzt, indirekt über die Kollektion `Attribute` zu erreichen, die sämtliche Textattribute der Datenbank speichert.

7.3.2 Indexierung einer Datenbank

Die Indexierung erfolgt in zwei Schritten. Zunächst werden die IDs aller Objekte ermittelt, die Textattribute besitzen:

```
select distinct A.Besitzer.ID from A in Attribute
```

Anschließend werden die logischen Pfade für alle selektierten Objekte erstellt. Beispielsweise wird dem Objekt mit der ID 215, das sich über die Kollektion “Attribute” erreichen läßt, der Pfad

```
/Attribute/215
```

zugeordnet. Die logischen Pfade werden nun dem IR-System mitgeteilt. Ab diesem Punkt können alle lsearch-Funktionen völlig transparent genutzt werden. Intern wird jeder Zugriff auf ein Speichermedium auf die Klasse `O2READER` abgebildet. Die Methode `open()` erhält als Argument einen logischen Pfadausdruck, der in die Bestandteile `collection_name` und `objekt_id`, im Beispiel “Attribute” und 215, zerlegt wird. Die OQL-Query

```
select distinct A from A in Attribute where A.Besitzer.ID = 215
```

selektiert alle Textattribute des Objekts mit der ID 215 und faßt die Attributwerte durch Verkettung zu einem *virtuellen Dokument* zusammen. In diesem Dokument werden alle `read()`, `tell()` und `seek()`-Operationen ausgeführt, bis es durch Aufruf von `close()` zerstört wird.

7.3.3 Suche in einer Datenbank

Die Suche im Textbestand erfolgt ebenfalls weitgehend transparent durch Methoden der IR-Bibliothek. Soll die Suche mit dem Kommandozeilen-Programm `Isearch` ausgeführt werden, ist keine Änderung des Programms notwendig, da `Isearch` die gefundenen (virtuellen) Dokumente selbst anzeigt.

Für die Integration in die WWW-Komponente müssen allerdings die IDs der Objekte bekannt sein, für die HTML-Text erzeugt werden soll. Deshalb ist eine Zerlegung der logischen Pfade notwendig, die vom IR-System geliefert werden. Die Menge der relevanten IDs wird zunächst in Listenform ausgegeben, aus welcher der Nutzer die gewünschten Objekte auswählen kann. Das Objekt mit der ID 215 kann durch die Query

```
element(select A.Besitzer from A in Attribute where A.Besitzer.ID = 215)
```

selektiert und anschließend formatiert und angezeigt werden.

7.3.4 Nutzung von Isearch-Dokumententypen

Die Indexierung sowie die Suche benutzen ausschließlich den Standard-Dokumententyp “SIMPLE”. Eine Erweiterung der Retrievalfunktionen könnte dadurch geschehen, daß ein Datenobjekt als ein *strukturiertes* Dokument aufgefaßt wird, dessen Aufbau durch die Attributnamen vorgegeben ist. Durch die Einschränkung auf einen Attributnamen bei der Formulierung der Anfrage würde die Suche nur im zugehörigen Attributwert durchgeführt werden. Das folgende Beispiel sucht in allen Exponaten nach dem Wort “Sachsen”, das im Feld “Herkunft” vorkommen muß:

```
Isearch -d EXPONATE Herkunft/Sachsen
```

Diese Form der strukturierten Suche weist große Ähnlichkeit mit der automatischen Query-Generierung auf, die in Unterabschnitt 6.4.5 vorgestellt wurde. Die dort beschriebene Methode könnte von einem lsearch-Dokumententyp aber nicht völlig ersetzt werden, da mit einem IR-System nur in Texten gesucht werden kann, während die strukturierte Suche auch numerische Datentypen einbezieht.

7.4 Zusammenfassung

Mit IRS kann ein großer Teil der Daten in Informationssystemen auf einheitliche Weise erreicht werden. Sie lassen oft natürlichsprachliche Anfragen zu und sortieren die Suchergebnisse nach ihrer Relevanz. Dadurch bilden IRS einen interessanten Kontrast zu den datenbankeigenen Anfragesprachen.

Die freie C++-Klassenbibliothek lsearch baut auf dem Vector-Space-Model auf. Sie gestattet natürlichsprachliche Anfragen und unterstützt das Ranking der Suchergebnisse. Da diese Bibliothek ausschließlich für die Suche in Dateien vorgesehen ist, wurde eine Zwischenschicht eingeführt, die den Zugriff auf andere Speichermedien, insbesondere Datenbanken, gestattet. Solange sich jedes (virtuelle) Dokument über einen eindeutigen Pfad erreichen läßt, kann das IRS zwischen einem Zugriff auf das Dateisystem und dem Zugriff auf eine Datenbank nicht mehr unterscheiden.

Sowohl während der Indexierung einer Datenbank als auch während der Suche werden die logischen Pfade zerlegt und das betreffende Objekt durch eine Query, die aus den Bestandteilen des Pfades generiert wird, aus der Datenbank selektiert. Alle Textattribute des Objekts werden zu einem virtuellen Dokument verkettet, in dem alle weiteren Zugriffe stattfinden.

8 Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurde ein Informationssystem für heterogene, multimediale Daten konzipiert und realisiert, das aus den Komponenten Datenbankschema, Schemalader, Datenlader, WWW-Anbindung und Information-Retrieval-System besteht.

Zu Beginn der Arbeit wurden die *Anforderungen* analysiert, die an ein datenbankbasiertes Informationssystem gestellt werden müssen. Dabei wurde festgestellt, daß das Datenbankschema in der Lage sein muß, heterogene Anwendungsbereiche adäquat zu modellieren. Weiterhin benötigt es ein hohes Maß an Flexibilität. Das Datenbankschema und die Applikationen müssen weitgehend voneinander entkoppelt werden, um alle Komponenten unabhängig voneinander variieren zu können. Das Informationssystem muß alle Anwendungsdaten mehrsprachig verwalten und die Speicherung und das Retrieval multimedialer Daten unterstützen.

In Kapitel 4 wurde nach der Einführung und Diskussion wichtiger Begriffe untersucht, warum eine starre Kopplung zwischen Datenbankschema und Programmiersprache für Informationssysteme oft ungeeignet ist. Anschließend wurde ein Datenbankschema entwickelt, das die Modellierung wichtiger objektorientierter Konzepte erlaubt, Anwendungsdaten und Strukturinformationen mehrsprachig speichert und multimediale Daten in generischer Weise verwaltet. Auf der Grundlage eines *Metaklassenschemas* können *Datenschemata* dynamisch erstellt werden, deren Aufgabe es ist, die Struktur der zugehörigen *Anwendungsdaten* zu bestimmen.

Ein Datenschema wird durch eine *Schema-Definitionssprache* spezifiziert. Die Aufgabe des *Schemaladers* ist es, die Schemabeschreibung zu interpretieren und daraus ein Datenschema zu erstellen. Der *Datenlader* importiert die in einer *Daten-Definitionssprache* angegebenen Anwendungsdaten. Die inhaltliche Korrektheit der Daten wird mit Hilfe des Datenschemas verifiziert.

Die *WWW-Anbindung* hat die Aufgabe, Nutzeranfragen entgegenzunehmen, die angeforderten Daten aus der Datenbank zu selektieren und daraus Hypertext-Dokumente zu generieren. Dabei wird die Reflexivität des Datenschemas und der Anwendungsdaten ausgenutzt, so daß die WWW-Anbindung und das jeweilige Datenschema vollständig unabhängig voneinander sind. Ausgehend von einem allgemeinen Modell der Objektdarstellung wurde ein Klassenschema vorgestellt, das als Grundlage eines Informationssystems im World Wide Web dienen kann. Anschließend wurde gezeigt, wie sich dieses Schema in das Programmpaket O2Web integrieren läßt.

Kapitel 7 erläuterte am Beispiel einer angepaßten Version der Klassenbibliothek lsearch, auf welche Weise ein *Information-Retrieval-System* an das Datenbankschema angebunden werden kann.

Die vorgestellte Architektur läßt einen breiten Spielraum für Verbesserungen und Erweiterungen. Durch einige konzeptionelle und viele praktische Ergänzungen kann das System über vorrangig *darstellungsorientierte* Anwendungen hinaus zu einer Applikation ausgebaut werden, die zur *Verwaltung* heterogener, multimedialer Daten verwendet werden kann. Mögliche Einsatzgebiete sind überall dort zu finden, wo der zu verwaltende Datenbestand eine komplexe Struktur aufweist, Binärdaten verschiedenen Typs umfaßt, häufigen Schema-Änderungen unterliegt sowie gleichzeitig eine strukturierte und eine textbasierte Dokumentensicht benötigt. Das trifft auf Bibliotheksprogramme und wissenschaftliche Kataloge in Museen zu, aber auch auf weniger offensichtliche Anwendungen wie interaktive Lehrmaterialien und Konstruktionspläne.

Die für ein breiteres Einsatzgebiet notwendigen *konzeptionellen* Erweiterungen umfassen:

- Zuordnung von Methoden zu extensionalen Klassen
- Unterstützung weiterer Datentypen für Attribute
- mehrfache oder dynamisch änderbare Klassenzugehörigkeit von Objekten
- verbesserte Verwaltung von *strukturierten* Texten und multimedialen Daten
- Ergänzung eines Sichtkonzepts und einer Benutzerverwaltung

Die *praktischen* Erweiterungsmöglichkeiten bestehen vor allem im Ausbau der Schnittstelle zu den (autorisierten) Benutzern. Das Datenbankschema sowie die Ladeprogramme besitzen Programmierschnittstellen, an die eine grafische Oberfläche angekoppelt werden kann. Auf diese Weise stehen alle Möglichkeiten moderner Benutzerschnittstellen

für die interaktive Schema- und Datenverwaltung zur Verfügung. Ebenfalls denkbar ist eine CORBA-Anbindung der grafischen Oberfläche an das Datenbankschema und die Ladeprogramme.

Anhang

A Verzeichnis der verwendeten Abkürzungen

API	**A**pplication **P**rogram **I**nterface
BLOB	**B**inary **L**arge **Ob**ject
BNF	**B**ackus **N**aur **F**orm
CGI	**C**ommon **G**ateway **I**nterface
CLOS	**C**ommon **L**isp **O**bject **S**ystem
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
DBMS	**D**ata**b**ase **M**anagement **S**ystem
DDL	**D**ata **D**efinition **L**anguage
DML	**D**ata **M**anipulation **L**anguage
EBNF	**E**rweiterte **B**ackus **N**aur **F**orm
EOF	**E**nd **O**f **F**ile
ER	**E**ntity **R**elationship (-Modell)
HTML	**H**yper **T**ext **M**arkup **L**anguage
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
IDF	**I**nverse **D**ocument **F**requency
IR	**I**nformation **R**etrieval
IRS	**I**nformation **R**etrieval **S**ystem
IS	**I**nformations**s**ystem

JPEG	**J**oint **P**hotographic **E**xperts **G**roup
LALR(1)	**L**ook **A**head **L**eft **R**ecursive, **1** Token lookahead (Parsing-Technik)
LR(1)	**L**eft **R**ecursive, **1** Token lookahead (Parsing-Technik)
MIME	**M**ultipurpose **I**nternet **M**ail **E**xtensions.
ODMG	**O**bject **D**ata **M**anagement **G**roup
OMT	**O**bject **M**odeling **T**echnique
OODBMS	**O**bject **O**riented **D**ata**b**ase **M**anagement **S**ystem
OQL	**O**bject **Q**uery **L**anguage
ORDBMS	**O**bject **R**elational **D**ata**b**ase **M**anagement **S**ystem
RDBMS	**R**elational **D**ata**b**ase **M**anagement **S**ystem
RPC	**R**emote **P**rocedure **C**all
SQL	**S**tructured **Q**uery **L**anguage
UML	**U**nified **M**odeling **L**anguage
URL	**U**nified **R**esource **L**ocator
VSM	**V**ector **S**pace **M**odel
WAIS	**W**ide **A**rea **I**nformation **S**erver
WWW	**W**orld **W**ide **W**eb
YACC	**Y**et **A**nother **C**ompiler **C**ompiler

B Glossar

Im Zusammenhang mit dem in Kapitel 4 entwickelten Datenbankschema wurden eine Reihe von Fachbegriffen benutzt. Viele besitzen eine mehr oder weniger feste Bedeutung im Kontext von objektorientierten Programmiersprachen oder Datenbanken. Die Definition solcher Begriffe ist in normaler Schrift angegeben. Einige Begriffe haben (außerdem) eine spezielle Bedeutung für das entwickelte Datenbankschema. Die schemaspezifische Bedeutung ist in *kursiver* Schrift gesetzt.

Aggregation Eine Aggregation ist eine Sonderform der Assoziation, bei der die beteiligten Klassen keine gleichwertige Beziehung führen, sondern eine Ganzes-Teile-Hierarchie darstellen. [Oes97, S. 266]

Assoziation Eine Relation zwischen Instanzen von zwei oder mehr Klassen. Beschreibt eine Gruppe von Verknüpfungen mit gemeinsamer Struktur und Semantik. [RBP+94, S. 553]

Assoziationsobjekt *Instanz der Implementierungsklasse* `Assoziation` *des Metaklassenschemas. Bestimmt den Aufbau der zugehörigen Verknüpfungen.*

Datenklassenschema *Menge von Implementierungsklassen des Datenbankschemas, deren Instanzen die Anwendungsdaten aufnehmen.*

Datenobjekt *Instanz der Implementierungsklasse* `Objekt` *des Datenklassenschemas; nimmt Anwendungsdaten auf.*

Datenbankschema Unter einem Datenbankschema versteht man eine Datenbankbeschreibung, d.h. die Spezifikation von Datenstrukturen mitsamt ihren zugehörigen Integritätsbedingungen. [Mei98, S. 23]

Das Datenbankschema umfaßt die Implementierungsklassen des Datenklassenschemas und des Metaklassenschemas.

Datenschema *Das Datenschema wird aus Instanzen des Metaklassenschemas (Klassen- und Assoziationsobjekte) gebildet. Es beschreibt die Struktur der Datenobjekte und legt deren Verknüpfungen mit anderen Datenobjekten fest.*

Implementierungsklasse Statisch (zur Übersetzungszeit) in einer Programmiersprache definierte Klasse.

Alle Klassen des Datenbankschemas sind Implementierungsklassen.

Klasse Eine Gruppe von Objekten mit gleichen Merkmalen (Attributen) und gleichem Verhalten (Methoden), gemeinsamen Relationen zu anderen Objekten und einer gemeinsamen Semantik. [RBP+94, S. 28]

Eine **intensionale Klasse** ist ein programmiersprachliches Konstrukt, das einen komplexen Typ kapselt und implementiert. Der Typ *definiert* die zu einer Klasse gehörende Domäne und bestimmt somit die Menge potentieller Objekte.

Eine **extensionale Klasse** ist eine *Menge* von (potentiellen) Objekten, die gleiche Struktur und Semantik besitzen. Einer extensionalen Klasse wird gegebenenfalls ein komplexer Typ zugeordnet.

Datenobjekte mit gleichen Attributen, Verknüpfungen, Komponenten sowie gleicher Position in der Klassenhierarchie werden zu extensionalen Klassen zusammengefaßt. Die Struktur und Semantik einer extensionalen Klasse wird durch ein Klassenobjekt (mit intensionalem Charakter) bestimmt.

Klassenobjekt Instanz einer Metaklasse mit den Aufgaben einer (intensionalen) Klasse und einem (änderbaren) Zustand.

Instanz der Implementierungsklasse `Klasse` *des Metaklassenschemas. Ein Klassenobjekt ist eine intensionale Klasse, da es den Aufbau der Objekte der zugehörigen extensionalen Klasse bestimmt.*

Komposition Eine Komposition ist eine strenge Form der Aggregation, bei der die Teile vom Ganzen existenzabhängig sind. [Oes97, S. 272]

Metaklasse Klasse höherer Ebene, deren Instanzen Klassenobjekte sind. Man unterscheidet *implizite* Metaklassen, deren Aufbau fest eingestellt ist, und *explizite* Metaklassen, die zur Laufzeit erzeugt, verändert und vernichtet werden können.

Eine Metaklasse ist eine Implementierungsklasse des Metaklassenschemas. Sie hat somit impliziten Charakter.

Metaklassenschema *Menge von Implementierungsklassen des Datenbankschemas, deren Instanzen (Klassenobjekte) das Datenschema bilden.*

Objekt Ein Konzept, eine Abstraktion oder ein Gegenstand mit klaren Abgrenzungen und einer präzisen Bedeutung für das anstehende Problem. [RBP+94, S. 27]

Verknüpfung Physikalische oder konzeptuelle Verbindung zwischen Objektinstanzen. [RBP+94, S. 34]

Instanz der Implementierungsklasse `Verknüpfung` *des Datenklassenschemas. Die Struktur und Semantik einer Verknüpfung wird durch das zugehörige Assoziationsobjekt festgelegt.*

C Grammatik der Laderformate

C.1 Schemalader

```
Start          → ε | KlassenObjekt Start
KlassenObjekt  → ImplKlasse KlassenName BasisKlassen KlassenDef Ende
ImplKlasse     → ε | OBJEKT | MULTIMEDIA
KlassenName    → WORT
BasisKlassen   → ε | ':' WORT WeitereBasis
WeitereBasis   → ε | ',' WORT WeitereBasis
KlassenDef     → '{' Ende TITEL KlassenTitel KlassenDaten '}'
KlassenTitel   → Text KlassenTitel2 ENDE
KlassenTitel2  → ε | '|' Text KlassenTitel2
KlassenDaten   → ε
                 | Kategorie   KlassenDaten
                 | Attribut    KlassenDaten
                 | Assoziation KlassenDaten
                 | Aggregation KlassenDaten
                 | Komposition KlassenDaten
Kategorie      → KATEGORIE '{' Ende InnereDaten '}' Ende
InnereDaten    → ε
                 | Attribut    InnereDaten
                 | Assoziation InnereDaten
                 | Aggregation InnereDaten
```

	|	Komposition InnereDaten
Attribut	→	AttributTyp AttributName ENDE
AttributTyp	→	ZAHL | KURZTEXT | TEXT
AttributName	→	Text AttributNamen
AttributNamen	→	ε | '|' Text AttributNamen
Assoziation	→	ASSOZIATION AssozKopf Inverse AssozDef Ende
Aggregation	→	AGGREGATION AssozKopf AssozDef Ende
AssozKopf	→	Kardinalität AssozName ':' ZielTyp AssozZiel
AssozName	→	WORT
ZielTyp	→	ε | KATEGORIE
AssozZiel	→	WORT
Kardinalität	→	ε | '[' ZielBereich QuellBereich ']'
EinfacheKard	→	ε | '[' ZielBereich ']'
ZielBereich	→	BereichsDef
QuellBereich	→	ε | ',' BereichsDef
BereichsDef	→	Mimimal BEREICH Maximal
Mimimal	→	ANZAHL
Maximal	→	ANZAHL
Inverse	→	ε | INVERSE InverseName
InverseName	→	WORT
AssozDef	→	'{' Ende TITEL AssozTitel AssozDaten '}'
AssozTitel	→	TITEL Text AssozTitel2 ENDE
AssozTitel2	→	ε | '|' Text AssozTitel2
AssozDaten	→	ε | Attribut AssozDaten
Komposition	→	KOMPOSITION KompositionKopf KompositionDef Ende
KompositionKopf	→	EinfacheKard ImplKlasse KlassenName BasisKlassen
KompositionDef	→	'{' Ende InnereDaten '}'
Text	→	WORT WeitererText
WeitererText	→	ε | WORT WeitererText

```
Ende              →  ε | ENDE
```

Die in Großbuchstaben angegebenen Symbole werden vom Scanner geliefert und sind durch die folgenden regulären Ausdrücke definiert:

```
ANZAHL       ::=  [0-9]+|\*
AGGREGATION  ::=  AGGREGATION
ASSOZIATION  ::=  ASSOZIATION
BEREICH      ::=  \.\.
ENDE         ::=  \n
INVERSE      ::=  INVERSE
KATEGORIE    ::=  KATEGORIE
KOMPOSITION  ::=  KOMPOSITION
KURZTEXT     ::=  KURZTEXT
MULTIMEDIA   ::=  MULTIMEDIA
TITEL        ::=  TITEL
OBJEKT       ::=  OBJEKT
TEXT         ::=  TEXT
WORT         ::=  [a-zA-ZäöüßÄÖÜ0-9\-]+
ZAHL         ::=  ZAHL
```

C.2 Datenlader

```
Start          → ε
               | Kategorie Start
               | Objekt    Start
Kategorie      → KATEGORIE KategorieTyp KategorieDef Ende
KategorieTyp   → Text
KategorieDef   → '{' Ende ObjektDaten '}'
Objekt         → KlassenName ObjektDef Ende
KlassenName    → Text
ObjektDef      → '{' Ende ObjektTitel ObjektDaten '}'
ObjektTitel    → TITEL ':' TitelName ENDE
TitelName      → Text TitelNamen
TitelNamen     → ε | '|' Text TitelNamen
ObjektDaten    → ε
               | Attribut    ObjektDaten
               | Verknüpfung ObjektDaten
               | Blob        ObjektDaten
               | Objekt      ObjektDaten
Attribut       → AttributName ':' AttributWert ENDE
AttributName   → Text
AttributWert   → Text AttributWerte | Datei AttributWerte
AttributWerte  → ε | Text AttributWerte | Datei AttributWerte
Datei          → '[' DateiName ']'
DateiName      → WORT
Verknüpfung    → VerknüpfName '<' Ende VerknüpfAttr '>' VerknüpfDef
VerknüpfName   → Text
VerknüpfAttr   → Attribut VerknüpfAttr2
VerknüpfAttr2  → ε | Attribut VerknüpfAttr2
```

```
VerknüpfDef    →  ENDE | '{' Ende VerknüpfDaten '}' Ende
VerknüpfDaten  →  ε | Attribut VerknüpfDaten
Blob           →  BlobTyp BlobDef ENDE
BlobTyp        →  GERING | MITTEL | HOCH
BlobDef        →  ':' Datei
Text           →  WORT WeitererText
WeitererText   →  ε | WORT WeitererText
Ende           →  ε | ENDE
```

Die Symbole für die Grammatik des Datenladers sind durch die folgenden regulären Ausdrücke definiert:

```
ENDE       ::=  \n
GERING     ::=  GERING
HOCH       ::=  HOCH
KATEGORIE  ::=  KATEGORIE
MITTEL     ::=  MITTEL
TITEL      ::=  TITEL
WORT       ::=  [^ \t\|\n\>\}\[\]#]+
```

D Dokumentation der implementierten Version

D.1 Bedienung der Programme des Informationssystems

Alle nachfolgend angegebenen Programme befinden sich im Verzeichnis `$HOME/mmis/bin` des Nutzers **mim**. Die meisten Programme werten die Umgebungsvariable `MMIS_BASE` aus, die den Namen der logischen O2-Datenbank angibt. Dieser Name kann durch den Parameter `-b Base` an der Kommandozeile des jeweiligen Programms überschrieben werden. Wenn der Schalter `-v` angegeben ist, protokollieren die Programme ihre Abarbeitung auf `stdout`.

`mmis-schemalader [-b Base] [-v] Datei`
: Importiert das in `Datei` spezifizierte Datenschema. Ein eventuell bereits vorhandenes Datenschema wird gelöscht.

`mmis-datenlader [-a] [-b Base] [-q] [-v] Datei [Datei [...]]`
: Importiert die in einer oder mehreren Dateien angegebenen Anwendungsdaten. Falls der Parameter `-a` *nicht* gesetzt ist, wird mit jedem Start des Datenladers der gesamte Datenbestand gelöscht. Der Schalter `-q` unterdrückt die Ausgabe von Warnungen.

  Der Typ multimedialer Daten wird durch die Dateiendung der Originaldaten bestimmt. Der Datenlader sucht im aktuellen Verzeichnis nach einer Datei mit dem Namen `MIME_TYPES`. In dieser Datei wird jeder Dateiendung ein MIME-Typ zugeordnet. Um einen Typ "Postscript" zu definieren, genügen die folgenden Einträge:

```
ps      application/postscript
eps     application/postscript
```

Der Datenlader führt nach dem Lesen jeder Datei ein `commit` aus. Deshalb dürfen sich *unidirektionale* Verknüpfungen nicht auf Objekte in nachfolgenden Dateien beziehen. Bei *bidirektionalen* Verknüpfungen sollte immer die jeweilige Vorwärtsrichtung angegeben werden. *Innerhalb* einer Datei sind sowohl Vorwärts- als auch Rückwärtsreferenzen erlaubt.

`lademimdaten`

Shell-Skript, das sämtliche Daten des Musikinstrumentenmuseums importiert.

`mmis-textindex [-b Base] [-v]`

Erstellt einen lsearch-Index über alle Texte der Datenbank. Die Indexdateien werden im Verzeichnis `$HOME/mmis/daten/textindex` abgelegt. Der Name einer Indexdatei setzt sich aus dem Namen der logischen Datenbank und einer Ziffer für die Sprache (0 = Deutsch, 1 = Englisch) zusammen. Mit dem Aufruf

```
mmis-textsuche -d $HOME/mmis/daten/textindex/mim1 Query
```

kann der englischsprachige Index über der Datenbank "mim" getestet werden. Der Parameter `Query` muß eine gültige lsearch-Suchanfrage beinhalten. Wird das Programm `mmis-textsuche` ohne Parameter gestartet, listet es eine Reihe von Beispielen für Queries auf.

`mmis-weblader [-b Base] [-v] Datei [Datei [...]]`

Lädt alle Icons, Image-Maps und Texte für den Webserver. Erstellt außerdem alle für die Galerie und die Lagepläne benötigten Referenzen.

`lademimconfig`

Shell-Skript, das den Webserver für das Musikinstrumentenmuseum konfiguriert.

`mmis-webserver [-v] Base [Base2 [...]]`

Startet den Webserver. Das Programm erwartet beim ersten Aufruf die Namen aller logischen Datenbanken, deren Daten es anzeigen soll. Alle logischen Datenbanken müssen zum O2-Schema `mmis` gehören, welches das Datenbankschema von Abbildung 4.12 auf Seite 49 beherbergt. *Vor* dem Start des Webservers sollten die Programme

```
$HOME/httpd/src/httpd -f $HOME/httpd/conf/httpd.conf
$O2HOME/bin/o2open_dispatcher &
```

gestartet werden. Prinzipiell können die Programme `httpd`, `o2open_dispatcher` und `mmis-webserver` auf einem beliebigen Rechner der Abteilung Datenbanken ausgeführt werden, sofern dieser für das O2Web-System konfiguriert wurde [O2W96, S. 5-67]. Für den Rechner **doesen6** wurden diese Vorbereitungen bereits getroffen.

Im Verzeichnis `$HOME/httpd/cgi-bin` befindet sich das CGI-Programm `baumtest`. Das Programm kann als Hilfsmittel bei der Schemaentwicklung benutzt werden. Es interpretiert eine per URL-Parameter angegebene Schema-Definitionsdatei und visualisiert die Klassenhierarchie. Die Anwendung kann über den URL

```
http://host:5050/cgi-bin/baumtest?Datei=/mim/schema.txt&Sprache=0
```

erreicht werden, wobei `host` den Rechner `doesen6.informatik.uni-leipzig.de` bezeichnet. Der Parameter `Sprache` kann gegenwärtig die Werte 0 (= Deutsch) und 1 (= Englisch) annehmen. Der Parameter `Datei` muß den Pfad einer Schema-Definitionsdatei *relativ* zum Verzeichnis `$HOME/mmis/daten` enthalten.

D.2 Erschließung neuer Anwendungsgebiete

Durch die Flexibilität des Datenbankschemas ist die Erstellung weiterer Informationssysteme, die parallel zur Präsentation des Musikinstrumentenmuseums betrieben werden können, relativ leicht möglich. Dabei können zwei Abstufungen im benötigten Aufwand unterschieden werden:

Änderungen der Daten, des Datenschemas und der Grafiken (z.B. Menü).
Hier sind *keine* Änderungen am Quelltext der Applikationen notwendig. Alle Anpassungen beschränken sich auf das Erstellen/Anpassen von Daten-Definitions-, Schema-Definitions- und Konfigurationsdateien sowie gegebenenfalls das Zeichnen von Grafiken.

Änderungen des Layouts und Erweiterung der Funktionalität.
Diese Variante ist erforderlich, wenn das Informationssystem neue Funktionen benötigt, beispielsweise neue Einstiegspunkte in die Datenbank. Hier muß der Quelltext der WWW-Anbindung angepaßt werden. Das ist auch dann nötig, wenn das Layout der erzeugten HTML-Dokumente grundlegend geändert werden soll.

Der nachfolgende Unterabschnitt befaßt sich mit der Erstellung eines neuen Informationssystems *ohne* Quelltext-Änderung. Im Unterabschnitt D.2.2 wird besprochen, wie das Informationssystem mit neuer Funktionalität versehen werden kann bzw. wie die Erzeugung von HTML-Text angepaßt werden kann.

D.2.1 Neue Informationssysteme ohne Quelltext-Anpassung

In den nachfolgenden Ausführungen wird von einer neuen Präsentation für ein Museum ausgegangen, welches "antik" genannt wird.

1. Zu Beginn muß die Umgebungsvariable `MMIS_BASE` gesetzt werden, die von den meisten Applikationen des Informationssystems ausgewertet wird:

   ```
   export MMIS_BASE=antik
   ```

2. Nun muß eine neue logische O2-Datenbank erzeugt werden. Dies geschieht mit dem O2-Programm `o2shell`. Nachdem dieses Programm gestartet wurde, wartet es auf Nutzereingaben. Mit den folgenden Befehlen wird die logische Datenbank "antik" erstellt:

   ```
   set schema mmis;
   create base antik;
   quit;
   ```

   Das letzte Kommando muß durch Enter *und* Ctrl^D abgeschlossen werden.

3. An dieser Stelle sind alle Vorbereitungen für den Import des Datenschemas und der Anwendungsdaten getroffen. Für die Daten- und Schema-Definitionsdateien sollte ein neues Verzeichnis angelegt werden:

   ```
   cd $HOME/mmis/daten
   mkdir antik
   cd antik
   ```

   Sei nun `antikschema.txt` die Schema-Definitionsdatei und `antikdaten.txt` eine Daten-Definitionsdatei. Schema und Daten werden dann mit den Befehlen

```
mmis-schemalader antikschema.txt
mmis-datenlader  antikdaten.txt
```

in die Datenbank "antik" importiert. Mit dem Kommando

```
mmis-textindex
```

wird ein Index über alle Texte der logischen Datenbank "antik" erstellt.

4. Nach dem Schema- und Datenimport muß der Webserver konfiguriert werden. Das dafür zuständige Programm `mmis-weblader` importiert alle für die Darstellung benötigten Texte, Icons und Image-Maps in persistente Kollektionen und erstellt alle für die Galerie benötigten Referenzen. Da der Webserver über den Kollektionsindex auf die Einträge zugreift, müssen Anzahl und Reihenfolge der Einträge in den jeweiligen Kollektionen genau stimmen. Daher bietet es sich an, alle Konfigurationsdateien des Musikinstrumentenmuseums zu kopieren:

   ```
   cd $HOME/mmis/config
   mkdir antik
   cp -r mim/* antik
   cd antik
   ```

   In den kopierten Konfigurationsdateien können nun beispielsweise einzelne Icons oder Image-Maps angepaßt oder ausgetauscht werden[1]. Die Konfigurationsdaten werden mit dem Kommando

   ```
   mmis-weblader webtexte.txt webicons.txt webmaps.txt webgalerie.txt
   ```

   importiert.

5. Zum Schluß muß der Webserver auf dem Rechner **doesen6** neu gestartet werden:

   ```
   stop-mmis-server
   mmis-webserver mim antik &
   ```

[1] Jedes Icon liegt sowohl als GIF- als auch als XCF-Datei vor. XCF-Dateien können mit dem Programm gimp (GNU Image Manipulation Program) bearbeitet und anschließend im GIF-Format exportiert werden.

Das neue Informationssystem kann nun mit dem URL

`doesen6.informatik.uni-leipzig.de:5050/o2web/generic/antik?WebBasis`

aufgerufen werden.

D.2.2 Neue Informationssysteme mit Quelltext-Anpassung

Um neue Funktionen zum Menü des Informationssystems hinzuzunehmen oder um die Erzeugung des HTML-Textes zu ändern, ist eine Anpassung der Klassen des Webservers notwendig. Alle Klassen des Webservers befinden sich im Verzeichnis

`$HOME/mmis/src/webserver`.

Jede Klasse des Klassendiagramms (siehe Abbildung 6.2 auf Seite 87) befindet sich in einer eigenen Datei, die den Namen der Klasse trägt.

Da die vollständige Beschreibung aller Details der Implementierung sehr umfangreich wäre, werden nachfolgend nur einige mögliche Szenarios beschrieben. Grundsätzlich sollte das Kapitel 6 dieser Diplomarbeit für die prinzipielle Arbeitsweise des Webservers zu Rate gezogen werden. Die Programmierschnittstelle des Datenbankschemas ist in den Unterabschnitten D.3.2 und D.3.3 angegeben.

Erweiterung auf drei oder mehr Sprachen

Um das System um weitere Sprachen zu ergänzen, muß zunächst die Konstante `SPRACHEN` in der Datei

`$HOME/mmis/src/klassen/meta.hh`

auf die Anzahl der gewünschten Sprachen gesetzt werden. Anschließend sollte die Enumeration `SprachTyp` in derselben Datei um einen symbolischen Namen für die neue Sprache erweitert werden. Anschließend muß jede Stelle im Quelltext des Webservers, an der mehrsprachige Zeichenketten ausgegeben werden, um einen Eintrag in der neuen Sprache ergänzt werden. Alle betreffenden Stellen werden leicht mit den Befehlen

```
cd $HOME/mmis/src/webserver
grep ENGLISCH *.cc
```

gefunden.

Änderung des Layouts der HTML-Dokumente

Die Grundstruktur jedes generierten HTML-Dokuments besteht aus den Teilen *linke obere Ecke*, *Kopfteil*, *Menüleiste* und *Hauptbereich*. Diese Bereiche werden durch eine HTML-Tabelle festgelegt, die in der Methode `WebBasis::html_header()` geöffnet und in `WebBasis::html_footer()` geschlossen wird. Der Haupbereich des Dokuments wird durch die jeweilige Implementierung der Methode `WebBasis::html_report()` generiert. Die Struktur von HTML-Tabellen verdeutlicht man sich am besten, indem man den Parameter `BORDER` auf einen Wert größer Null setzt.

Alle verwendeten Farben werden durch Konstanten in der Datei `webdefs.hh` definiert. Hier kann auch die Schriftart eingestellt werden.

Erweiterung oder Änderung des Menüs

Das Menü läßt sich logisch in eine grafische und eine funktionelle Komponente unterteilen. Die grafische Komponente besteht aus mehreren Grafiken und zugeordneten HTML-Image-Maps, die vom Weblader interpretiert und in die Datenbank importiert werden. Alle Image-Maps sind über feste Indizes in der persistenten Liste "HtmlMaps" zu erreichen. Der Index einer Image-Map wird durch ihre Position in der Konfigurationsdatei bestimmt. Die Methode `html_report()` der Klasse `WebMenü` zeigt die Grafiken des Menüs in Abhängigkeit von der aktuellen Sprache und Bildqualität sowie dem aktuellen Audioformat an. Zu jeder Grafik existiert eine Menge von *Areas*, die jeweils einen aktiven Bereich der Image-Map festlegen. Die Klasse `WebMenü` ordnet jeder Area einen URL zu, der ein persistentes Objekt (beispielsweise "WebKategorieBaum" oder "WebDetailSuche") selektiert.

Eine Änderung oder Erweiterung des Menüs verläuft in fünf Schritten:

1. Zeichnen der Menü-Grafiken (je eine für jede Spache, Bildqualität und jedes Audioformat) und Bestimmen der aktiven Regionen[2].
2. Erstellen oder Anpassen einer Konfigurationsdatei für den Weblader. In dieser Datei werden die Namen der Bilddateien sowie die Namen und Positionen der aktiven Bereiche für jede Grafik spezifiziert.
3. Import der Daten der Konfigurationsdatei in die Datenbank.
4. Zuordnen der Areas zu URLs in der Methode `WebMenü::html_report()`.
5. Neuübersetzen des Webservers.

Änderung der Darstellung von Anwendungsdaten

Die Darstellung der Anwendungsdaten ist grundsätzlich unabhängig von einem bestimmten Datenschema. Anpassungen sollten daher nur notwendig sein, wenn das *Layout* verändert werden soll.

Die Methode `WebBasis::zeigeObjekt()` ist in der Lage, jedes Objekt des Datenklassenschemas zu formatieren. Sie wird ausschließlich innerhalb der Methode `html_report()` aufgerufen. Für jedes Datenobjekt wird eine (innere) Tabelle geöffnet. Der Aufbau der Tabelle richtet sich danach, ob das Objekt eine Verknüpfung mit einem Objekt der Klasse `Multimedia` besitzt. Ist das der Fall und handelt es sich um eine Grafik, so wird anhand der Maße der Grafik entschieden, ob die Tabelle horizontal oder vertikal gegliedert wird. Für die Darstellung von Attributen und Verknüpfungen werden wiederum innere Tabellen geöffnet. Übersteigt die Anzahl der Verknüpfungen einer Assoziation eine feste Zahl (`PARTITION_SIZE`), so wird die Liste der Verknüpfungen an eine Instanz der Klasse `ErgebnisListe` übergeben, die jeweils nur einen Ausschnitt der Liste anzeigt. Komponenten sowie Verknüpfungen mit der Semantik einer Aggregation werden durch rekursiven Aufruf der Methode `zeigeObjekt()` dargestellt.

Da die Darstellung der Anwendungsdaten im wesentlichen vom Aufbau der inneren Tabellen abhängt, empfielt es sich auch hier, zur Veranschaulichung den Parameter `BORDER` aller Tabellen auf einen Wert größer Null zu setzen.

[2] Beispielsweise mit dem Programm xv, das bei Betätigung der mittleren Maustaste die aktuellen Koordinaten anzeigt.

Neuübersetzen des Webservers

Nach der Anpassung des Quelltextes des Webservers muß die Applikation mit den Kommandos

```
cd $HOME/mmis/src/webserver
make
```

neu übersetzt werden. Gegebenenfalls importiert der O2-Präprozessor die geänderten Klassen erneut in die Datenbank. Anschließend sollte der Webserver mit dem Kommando

```
make install
```

in das Verzeichnis `$HOME/mmis/bin` kopiert werden.

Wenn die Anzahl unterstützter Sprachen geändert wurde, müssen *alle* Applikationen neu übersetzt werden:

```
cd $HOME/mmis/src
perl build-mmis
```

Paralleler Betrieb von Webservern

Leider erschwert die Arbeitsweise des O2Web-Systems den *parallelen* Betrieb von mehreren Webservern mit *unterschiedlicher* Funktionalität[3] über einem O2-Schema. Der Grund dafür ist, daß sich ein Webserver nicht eindeutig zu einer logischen Datenbank zuordnen läßt. Jeder Webserver kennt lediglich das O2-Schema, über dessen Daten er operiert. Der O2Web-Dispatcher nutzt *Heuristiken*, um den geeigneten Webserver für einen URL zu finden [O2W96, S. 5-67]. Ein wichtiges Kriterium ist der Name der logischen Datenbank. Anhand des Datenbank-Namens kann der Dispatcher auf das zugehörige O2-Schema und damit auf den passenden Webserver schließen. Das bedeutet aber auch, daß es zu jedem O2-Schema nur einen Webserver geben darf. Anderenfalls kann der Dispatcher die Zuordnung nicht eindeutig treffen.

[3] Das betrifft *nicht* die in Unterabschnitt D.2.1 besprochenen Informationssysteme ohne Quelltext-Anpassung. Hier bedient *ein* Webserver mehrere logische Datenbanken.

Um mehrere unterschiedliche Webserver parallel betreiben zu können, muß für jeden dieser Server ein eigenes O2-Schema erzeugt werden. Das Perl-Skript `mmis-setup` erstellt ein O2-Schema und eine logische Datenbank, importiert alle Klassen in das Schema und übersetzt alle Anwendungen neu. Vor dem Start des Skriptes sollten alle Dateien des Verzeichnisses `$HOME/mmis/bin` umbenannt oder in ein anderes Verzeichnis kopiert werden. Das folgende Beispiel geht von einer Datenbank "antik" und einem Schema "antikschema" aus:

```
cd $HOME/mmis/src
perl setup-mmis antik antikschema
```

Danach sollten die Umgebungsvariablen `MMIS_BASE` und `MMIS_SCHEMA` auf die korrekten Werte gesetzt werden. Nun können alle neu übersetzten Programme wie gewohnt verwendet werden. Für jedes erzeugte O2-Schema muß der zugehörige Webserver gestartet werden.

D.2.3 Registrierung beim WWW-Server der Universität

Auf dem Rechner **www.uni-leipzig.de** existiert ebenfalls ein Account **mim**. Es wurde bereits ein Alias von der Addresse

```
www.uni-leipzig.de/museum/musik/
```

auf die Datei

```
$HOME/webdir/mim/index.html
```

eingerichtet. Die Datei `index.html` enthält lediglich ein Frameset mit einem leeren oberen Frame und einem unteren Frame mit der Adresse

```
doesen6.informatik.uni-leipzig.de:5050/o2web/generic/mim?WebBasis
```

Um das neue Informationssystem "antik" beim WWW-Server der Universität zu registrieren, muß ein neuer Alias erstellt werden (E-Mail an Dr. Kunze, kunze@rz.uni-leipzig.de), beispielsweise von

`www.uni-leipzig.de/museum/antik/`

auf die Datei

`$HOME/webdir/antik/index.html`

Die Datei `index.html` kann eine Kopie aus dem Unterverzeichnis `mim` sein, in der lediglich die Zeichenkette “mim” durch “antik” ersetzt wird.

D.3 Schnittstellen

D.3.1 Anmerkungen zur Implementierung

Die Implementierung des Informationssystems erfolgte in der Programmiersprache C++. Zur Integration der Datenbank O2 wurde die ODMG-C++-Schnittstelle der Datenbank benutzt. Der ODMG Standard definiert unter anderem die folgenden Datentypen:

- **d_String** für die Verwaltung von Strings variabler Länge
- **d_Bits** für binäre Daten (Strings ohne abschließendes Zeichen '\0')
- **d_Ref** für (logische) Datenbankzeiger
- **d_List** für geordnete Listen
- **d_Iterator** für Iteratoren über Kollektionen (d_Set, d_Bag und d_List)

Eine Assoziation zwischen zwei Klassen kann mit dem Schlüsselwort **inverse** als invers gekennzeichnet werden. Für die Vorübersetzung der Klassendefinitionen ist ein spezieller Präprozessor notwendig, der gleichzeitig den Import der Klassen in die Datenbank übernimmt.

D.3.2 Die Schnittstelle des Metaklassenschemas

```
#define SPRACHEN 2 // Anzahl Sprachen, konstante Zahl aus Performance-Gründen

enum SprachTyp       { DEUTSCH, ENGLISCH };
enum QualitTyp       { GERING, MITTEL, HOCH }; // Zuordnung von Blobs
enum AttributTyp     { ZAHL, KURZTEXT, TEXT };
enum AssoziationsTyp { ASSOZIATION, AGGREGATION, KOMPOSITION };
enum ImplementTyp    { OBJEKT, MULTIMEDIA, KATEGORIE }; // Implementierungsklasse


class MetaAttribut {
protected:
  d_String Name[SPRACHEN];
  AttributTyp AttrTyp;
  ImplementTyp ImplTyp;
```

```
public:
  MetaAttribut(AttributTyp attrTyp, ImplementTyp implTyp);
  void setzeName(const char* name, SprachTyp sprache = DEUTSCH);
  // Zugriff
  const char* name(SprachTyp sprache = DEUTSCH) const;
  AttributTyp attrTyp() const;
  ImplementTyp implTyp() const;
};

class MetaBasis {
protected:
  d_String Name;
  d_String Titel[SPRACHEN];
  d_List<d_Ref<MetaAttribut> > Attribute;
public:
  MetaBasis(const char* name);
  void setzeTitel(const char* titel, SprachTyp sprache = DEUTSCH);
  void neuesAttribut(const d_Ref<MetaAttribut>& attribut);
  // Zugriff
  const char* name() const;
  const char* titel(SprachTyp sprache = DEUTSCH) const;
  d_Iterator<d_Ref<MetaAttribut> > attributeIterator() const;
  unsigned long attributeAnzahl() const;
};

class Assoziation : public MetaBasis {
  friend class Klasse;
protected:
  AssoziationsTyp AssozTyp;
  int QuellMinCard, QuellMaxCard;
  int ZielMinCard, ZielMaxCard;
  ImplementTyp QuellTyp, ZielTyp;
  d_Ref<Klasse> Quellklasse, Zielklasse;
  d_Ref<Assoziation> Inverse;
public:
```

```
  // Konstruktor 1: Zielklasse steht noch nicht fest
  Assoziation(const char* name, // Name der Assoziation
              AssoziationsTyp assozTyp, // Semantik der Assoziation
              ImplementTyp quellTyp, // Implementierungsklasse der Quelle
              const d_Ref<Klasse>& quellKlasse, // Quellklasse
              ImplementTyp zielTyp); // Implementierungsklasse des Ziels
  // Konstruktor 2: Zielklasse ist angegeben
  Assoziation(const char* name,
              AssoziationsTyp assozTyp,
              ImplementTyp quellTyp,
              const d_Ref<Klasse>& quellKlasse,
              ImplementTyp zielTyp,
              const d_Ref<Klasse>& zielKlasse);
  // Initialisierung
  void setzeZielklasse(const d_Ref<Klasse>& zielKlasse);
  void setzeInverse(const d_Ref<Assoziation>& inverse);
  void setzeKardinalitaet(int quellMinCard, int quellMaxCard,
              int zielMinCard, int zielMaxCard);
  // Zugriff
  AssoziationsTyp assozTyp() const;
  const d_Ref<Assoziation> inverse();
  const d_Ref<Klasse> quellKlasse();
  const d_Ref<Klasse> zielKlasse();
  int quellMinCard() const;
  int quellMaxCard() const;
  int zielMinCard() const;
  int zielMaxCard() const;
  ImplementTyp quellTyp() const;
  ImplementTyp zielTyp() const;
};

class Klasse : public MetaBasis {
protected:
  d_List<d_Ref<Assoziation> > Assoziationen;
  d_List<d_Ref<Klasse> > Komponenten;
```

```
  ImplementTyp ImplTyp;
  int MinCard, MaxCard; // Kardinalität für Komponenten
public:
  Klasse(const char* name, ImplementTyp implTyp);
  void neueAssoziation(const d_Ref<Assoziation>& assoziation);
  void neueKomponente(const d_Ref<Klasse>& komponente);
  void setzeKardinalitaet(int minCard, int maxCard);
  // Zugriff
  d_Iterator<d_Ref<Klasse> > komponentenIterator() const;
  d_Iterator<d_Ref<Assoziation> > assoziationenIterator() const;
  ImplementTyp implTyp() const;
  int minCard() const;
  int maxCard() const;
};
```

D.3.3 Die Schnittstelle des Datenklassenschemas

```
class Attribut {
protected:
  d_String Name[SPRACHEN];
  d_String Wert[SPRACHEN];
  int Zahl;
  d_Ref<DatenBasis> Besitzer inverse DatenBasis::Attribute;
public:
  // Konstruktor erzeugt Attribut und kopiert alle
  // Namen vom angegebenen MetaAttribut:
  Attribut(const d_Ref<MetaAttribut>& metaAttribut);
  // Konstruktur kopiert Werte eines existierenden Attributs:
  Attribut(const d_Ref<Attribut>& kopieren);
  void setzeWert(const char* wert, SprachTyp sprache = DEUTSCH);
  void setzeWert(int wert);
  // Zugriff
  const char* name(SprachTyp sprache = DEUTSCH) const;
  const d_String wert(SprachTyp sprache = DEUTSCH) const;
  int numerisch() const;
```

```
};

class DatenBasis {
protected:
  int ID;
  d_String Name;
  d_String Titel[SPRACHEN];
  d_List<d_Ref<Attribut> > Attribute inverse Attribut::Besitzer;
public:
  // Konstruktor erzeugt Objekt und kopiert alle
  // Namen von der angegebenen Metaklasse (Klasse oder Assoziation):
  DatenBasis(const d_Ref<MetaBasis>& metaBasis);
  void neuesAttribut(const d_Ref<Attribut>& attribut);
  // Zugriff
  virtual ImplementTyp implTyp() const;
  const char* name() const;
  int id() const;
  const char* titel(SprachTyp sprache = DEUTSCH) const;
  d_Iterator<d_Ref<Attribut> > attributeIterator() const;
  unsigned long attributeAnzahl() const;
};

class Verknuepfung : public DatenBasis {
protected:
  AssoziationsTyp AssozTyp;
  d_Ref<ObjektBasis> Zielobjekt;
  d_Ref<Verknuepfung> Inverse inverse Verknuepfung::Inverse;
public:
  // Konstruktor erzeugt Verknüpfung und kopiert alle
  // Namen von der angegebenen Assoziation:
  Verknuepfung(const d_Ref<Assoziation>& assoziation);
  // Wie oben, zusätzlich wird das Zielobjekt initialisiert:
  Verknuepfung(const d_Ref<Assoziation>& assoziation,
               const d_Ref<ObjektBasis>& zielobjekt);
  void setzeInverse(d_Ref<Verknuepfung>& inverse);
```

```
  void setzeZielobjekt(const d_Ref<ObjektBasis>& zielobjekt);
  // Zugriff
  AssoziationsTyp assozTyp() const;
  const d_Ref<ObjektBasis> zielObjekt() const;
};

class ObjektBasis : public DatenBasis {
protected:
  d_List<d_Ref<Verknuepfung> > Verknuepfungen;
  d_List<d_Ref<Objekt> > Komponenten;
public:
  ObjektBasis(const d_Ref<Klasse>& klasse);
  void setzeTitel(const char* titel, SprachTyp sprache);
  void neueVerknuepfung(const d_Ref<Verknuepfung>& verknuepfung);
  void neueKomponente(const d_Ref<Objekt>& komponente);
  // Zugriff
  d_Iterator<d_Ref<Verknuepfung> > verknuepfungenIterator() const;
  unsigned long verknuepfungenAnzahl() const;
  d_Iterator<d_Ref<Objekt> > komponentenIterator() const;
  unsigned long komponentenAnzahl() const;
};

class Objekt : public ObjektBasis {
protected:
  d_Ref<Kategorie> HatKategorie;
public:
  Objekt(const d_Ref<Klasse>& klasse);
  void neueKategorie(const d_Ref<Kategorie>& kategorie);
  // Zugriff
  ImplementTyp implTyp() const;
  const d_Ref<Kategorie> hatKategorie() const;
};

class Kategorie : public ObjektBasis {
protected:
  // Keine Inverse, da 'Objekte' die *transitive* Extension speichert:
```

```
  d_List<d_Ref<Objekt> > Objekte;
  d_List<d_Ref<Kategorie> > Oberkategorien inverse Kategorie::Unterkategorien;
  d_List<d_Ref<Kategorie> > Unterkategorien inverse Kategorie::Oberkategorien;
public:
  Kategorie(const d_Ref<Klasse>& klasse);
  // Neues Objekt in Kategorie aufnehmen. Falls transitiv != 0,
  // wird das Objekt auch in alle Oberkategorien aufgenommen:
  void neuesObjekt(const d_Ref<Objekt>& objekt, int transitiv);
  void neueOberkategorie(const d_Ref<Kategorie>& oberkategorie);
  void loescheObjekte();
  void loescheAttribute();
  // Zugriff
  ImplementTyp implTyp() const;
  d_Iterator<d_Ref<Objekt> > objekteIterator() const;
  unsigned long objekteAnzahl() const;
  d_Iterator<d_Ref<Kategorie> > oberKategorienIterator() const;
  unsigned long oberKategorienAnzahl() const;
  d_Iterator<d_Ref<Kategorie> > unterKategorienIterator() const;
  unsigned long unterKategorienAnzahl() const;
};

class Blob {
protected:
  d_String MIME;
  d_Bits Daten;
  long Breite, Hoehe;
public:
  // Konstruktoren
  Blob();
  Blob(const char* dateiname);
  int ladeDatei(const char *dateiname); // Rückgabe: 1 == ok, 0 == Fehler
   // Zugriff
  long breite() const;
  long hoehe() const;
  long groesse() const;
```

```
  d_String mime() const;
  const char *daten() const;
};

class Multimedia : public Objekt {
protected:
  d_Ref<Blob> Blobs[3];
public:
  Multimedia(const d_Ref<Klasse>& klasse);
  int ladeBlob(const char* dateiName, QualitTyp qual);
   // Zugriff
  ImplementTyp implTyp() const;
  d_String blobMime(QualitTyp qual) const;
  long blobGroesse(QualitTyp qual) const;
  long blobBreite(QualitTyp qual) const;
  long blobHoehe(QualitTyp qual) const;
  const char *blobDaten(QualitTyp qual) const;
};
```

D.3.4 Anpassung der Bibliothek lsearch

```
class READER {
public:
  virtual GDT_BOOLEAN Open(const STRING& aName) = 0;
  virtual GDT_BOOLEAN IsOpen() = 0;
  virtual GDT_BOOLEAN Close() = 0;
  virtual GDT_BOOLEAN Seek(long Offset, int Whence) = 0;
  virtual long Tell() = 0;
  virtual size_t Read(char* Buffer, size_t Size) = 0;
};
```

Die abstrakte Klasse `READER` ist die Basisklasse für die Klasse `FILEREADER`, die in die geänderte Version der Bibliothek lsearch aufgenommen wurde, sowie für die Klasse `O2READER`, die zum Zugriff auf die Extension der Klasse `Attribute` des Datenbankschemas dient.

D.4 Verzeichnisstruktur der beigefügten CD

Verzeichnis **bin**

Alle ausführbaren Dateien des Informationssystems (siehe Abschnitt D.1).

Verzeichnis **lib**

Programm-Bibliotheken: Datenbankschema, Weblader und tstools.

Verzeichnis **include**

Header-Dateien aller Bibliotheken. Werden teilweise erst vom Präprozessor erstellt.

Verzeichnis **src**

Unterverzeichnis **klassen**

Quelltext des Datenbankschemas (siehe Unterabschnitte D.3.2 und D.3.3).

Unterverzeichnis **lader**

Quelltext des Schemaladers und des Datenladers.

Unterverzeichnis **textindex**

Quelltext für das Programm `mmis-textindex` *und die Klasse* `O2READER`.

Unterverzeichnis **weblader**

Quelltext des Laders für die Konfiguration des Webservers.

Unterverzeichnis **webserver**

Quelltext des Webservers.

Unterverzeichnis **lsearch-1.14**

Angepaßte Version der Information-Retrieval-Klassenbibliothek lsearch.

Unterverzeichnis **tstools**

Einige Hilfsmittel für die Programmierung.

Verzeichnis **daten**

Unterverzeichnis **mim**

Kompletter Datenbestand des Musikinstrumentenmuseums.

Verzeichnis **config**

Unterverzeichnis **mim**

Konfiguration des Webservers für das Musikinstrumentenmuseum.

Verzeichnis **dok**

Postscript-Datei mir dem Inhalt dieser Arbeit.

Literaturverzeichnis

[Apa98] Apache HTTP-Server Project, 1998. [http://www.apache.de/docs/mod/core.html#limitrequestline].

[ASU88] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilerbau, Teil 1.* Addison-Wesley, Bonn, 1. Auflage, 1988.

[Bay97] B. Bayard. Nagelprobe: SQL-fähige RDBMS unter Linux. *iX Multiuser Multitasking Magazin*, Mai 1997.

[Boo94] G. Booch. *Objekt-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[DD93] C.J. Date and H. Darwen. *A guide to the SQL Standard: a user's guide.* Addison-Wesley, Reading, Massachusetts, 3rd edition, 1993.

[FS98] M. Fowler and K. Scott. *UML konzentriert.* Addison-Wesley Longman, Bonn, 1. Auflage, 1998.

[Fuh96] N. Fuhr. *Information Retrieval.* Vorlesungsskript, Universität Dortmund, 1996. [http://ls-6www.informatik.uni-dortmund.de/ir/teaching/courses/ir].

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, Bonn, 1. Auflage, 1996.

[GM97] M. Grauer and U. Merten. *Multimedia: Entwurf, Entwicklung und Einsatz in betrieblichen Informationssystemen.* Springer, Berlin, 1. Auflage, 1997.

[Heu97] A. Heuer. *Objektorientierte Datenbanken.* Addison-Wesley Longman, Bonn, 2. Auflage, 1997.

[HU94] J.E. Hopcroft and J.D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, Bonn, 3. Auflage, 1994.

[JCJO94] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Objekt-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1994.

[KS95] W. Klas and M. Schrefl. *Metaclasses and Their Application: Data Model Tailoring and Database Integration*, volume 943 of *Lecture notes in computer science.* Springer, Berlin, 1995.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., Sebastopol CA, 2nd edition, 1992.

[LPJN94] C. Liv, J. Peek, R. Jones, and A. Nye. *Managing Internet Information Services.* O'Reilly & Associates, Inc, Sebastopol, CA, 1994.

[Mei98] A. Meier. *Relationale Datenbanken: eine Einführung für die Praxis.* Springer, Berlin, 3. Auflage, 1998.

[MW97] A. Meier and T. Wüst. *Objektorientierte Datenbanken: ein Kompaß für die Praxis.* dpunkt-Verlag, Heidelberg, 1. Auflage, 1997.

[Nas97] N. Nassar. Searching with Isearch: Moving beyond WAIS. *Web Techniques Magazine*, May 1997.
[http://www.webtechniques.com/features/1997/05/nassar/nassar.shtml].

[O2W96] *O2Web User Manual.* O2 Technology, January 1996. Release 4.6.

[Oes97] B. Oestereich. *Objektorientierte Softwareentwicklung mit der UML.* R. Oldenbourg Verlag, München, 3. Auflage, 1997.

[Rah98] E. Rahm. *Implementierung von Datenbanksystemen I.* Vorlesungsskript, Universität Leipzig, 1998.
[http://www.informatik.uni-leipzig.de/ifi/abteilungen/db/skripte/IDBS1/inhalt.html].

[RBP+94] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Objektorientiertes Modellieren und Entwerfen.* Hanser Verlag, München, 1. Auflage, 1994.

[Sal71] G. Salton. *The SMART Retrieval System - Experients in Automatic Document Processing.* Prentice Hall, Englewood Cliffs/NJ, 1971.

[Sch97] M. Schader. *Objektorientierte Datenbanken: die C++-Anbindung des ODMG-Standards.* Springer, Berlin, 1. Auflage, 1997.

[Str97] A. Stritziger. *Komponentenbasierte Softwareentwicklung.* Addison-Wesley Longman, Bonn, 1. Auflage, 1997.

[STS97] G. Saake, C. Türker, and I. Schmitt. *Objektdatenbanken: Konzepte, Sprachen, Architekturen.* International Thomson Publishing, Bonn, 1. Auflage, 1997.

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

[vR79] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, London, 1979. [http://www.dcs.glasgow.ac.uk/Keith/Preface.html].

[Wee97] A. Weede. *Möglichkeiten der Software-Wiederverwendung durch komponentenbasierte Anwendungsentwicklung in einem Versicherungsunternehmen.* Diplomarbeit, Universität Leipzig, 1997.

[Wir96] N. Wirth. *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, Bonn, 1. Auflage, 1996.

[Wol98] C. Wolff. *Information Retrieval - Vorlesungsnotizen.* Vorlesungsskript, Universität Leipzig, 1998.

Erklärung

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

Leipzig, den 24. November 1998 .....

(Torsten Schlieder)