Konzeption und Implementation einer Import- / Exportschnittstelle für Datenstrukturen in objektorientierten Datenbanken

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplomingenieur (FH)
an der Fachhochschule für Technik und Wirtschaft Berlin

Fachbereich Ingenieurwissenschaften I
Studiengang Technische Informatik

Betreuer:	Dipl. Ing. Peter Puschmann
	Dr. Thomas Schmidt
Eingereicht von:	Arne Hildebrand

Berlin, 12. März 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ANSI – American National Standards Institute
BLOB – Binary Large Object
CORBA – Common Object Request Broker Architecture
CRUD – Create, Read, Update, Delete
DBMS – Database Management Systeme
Dob – Datenobjekte
DTD – Dokument-Typ-Definition
EDV – Elektronische Datenverarbeitung
EJB – Java Database Object
FHTW – Fachhochschule für Technik und Wirtschaft
HTML – Hyper Text Markup Language
HyLOs – Hypermedia Learning Objects
ID – Identifikationsschlüssel
IDREF – Referenzen auf Identifikationsschlüssel
IEC – International Electrotechnical Commission
IP – Internetprotokoll
ISO – International Organization for Standardization
JDBC – Java Database Connectivity
JDO – Java Database Object
JNDI – Java Naming and Directory Interface
IIOP – Internet Inter-ORB Protocol
MAF – MIR Application Framework
MIME – Multipurpose Internet Mail Extension
MIR – Media Information Repository
MobIT – Media Objects in Time
MW – Middleware
ODBC – Open Database Connectivity
ODL – Object Definition Language
ODMG – Object Database Management Group
OJB – Object Relational Bridge
OMG – Object Management Group
OQL – Object Query Language
PDF – Portable Document Format
RDF – Resource Description Framework
SGML – Standarized General Markup Language
SQL – Structured Query Language
TCP – Transmission Control Protocol

UML – Unified Modelling Language
W3C – World Wide Web Consortium
XHTML – eXtensible Hyper Text Markup Language
XMI – XML Metadata Interchange
XML – eXtensible Markup Language
XSD – XML Schema Definition

1 Einleitung

EDV-Systeme sammeln, ordnen und verdichten Daten und stellen sie für die Auswertung zur Verfügung. Dabei ist es von entscheidender Bedeutung, dass sinnvolle Informationen eingegeben werden und dass die ausgegebenen Informationen für den jeweiligen Empfänger von Wert sind. Durch einen zielgerichteten, organisierten Informationsfluss, zwischen Menschen und Maschinen und insbesondere zwischen Maschinen und Maschinen, entsteht ein Informationssystem.

Die Grundlagen eines solchen Informationssystems sind die Speicherung und der Zugriff auf diese Informationen. Das Wo und Wie der Informationsspeicherung ist demnach ein wichtiger Aspekt. Dabei hat die Entscheidung für das verwendete Datenbankmodell einen wesentlichen Einfluss auf die Leistungsfähigkeit und die Zugriffszeiten der späteren Anwendung. In den frühen 1970er Jahren fand die dazu entscheidende Entwicklung auf dem Gebiet der Datenbanksysteme statt. Heute sind aus dieser Zeit zwei Datenmodelle erwähnenswert: das Hierarchische Modell und das Relationale Modell.

Das Hierarchische Modell wird durch eine Baumstruktur definiert und kann im einfachsten Fall nur aus einer Referenz, der Eltern-Referenz, aufgebaut werden. Es erlaubt einen schnellen Datenzugriff auf zusammengesetzte Informationsstrukturen und ist leicht überschaubar.

Das Relationale Modell speichert die Daten in Tabellen, die bestimmte semantisch zusammengehörige Informationen gruppieren, und verbinden diese durch Referenzen. Sie sind insbesondere durch die Verwendung von Tabellen für Operationen mit vielen ähnlichen (gleich strukturierten) Datensätzen geeignet. Die unterschiedlichen Relationalen Datenbanksysteme werden durch internationale Standards[1] bestimmt und gestatten den Zugriff mit unterschiedlichen Frontendsystemen.

Mit der Weiterentwicklung von strukturierten, hin zu objektorientierten Programmiersprachen entstand der Wunsch, die dazugehörigen Daten nicht umständlich auf Tabellen zu verteilen, sondern die jetzt sprachentypischen Objekte direkt abzuspeichern. Seit den 1990er Jahren gibt es die Entwicklung von objektorientierten Datenmodellen. Diese enthalten analog zu den Programmiersprachen eine Klassenstruktur und können eigenständige Funktionen (Klassenmethoden) enthalten.

[1] Vgl. Abschnitt 2.1: ISO/ANSI und OMG.

Durch die weite Vernetzung über Internet/Intranet besteht zunehmend der Wunsch auf die unter Umständen sehr unterschiedlich aufgebauten Daten einzelner Informationsquellen zugreifen zu können, um diese auswerten, austauschen und übernehmen zu können.

Für diesen Vorgang werden Programme benutzt, die den Zugriff auf bisher nicht nutzbare Datenquellen ermöglichen. Über diese Software-Schnittstellen erfolgt die Umcodierung (Anpassung) der Informationen. Sie ermöglichen den Zugriff auf verteilte Informationsquellen über Computer- und Softwaresysteme hinweg.

Um also eine Information nutzbar machen zu können, muss sie durch die Schnittstelle in ein lesbares kompatibles bzw. entgegengesetzt in ein fremdes Format umgewandelt werden. Für die Interoperabilität von Systemen muss aber, neben der Überführung der Daten selbst, die Bedeutung der Informationen allgemein verständlich dargestellt werden.

So genannte Datenstruktur-Beschreibungen bilden standardisiert den Aufbau von Daten ab und ermöglichen so die Interpretation von Informationen gleicher Bedeutung. Deshalb soll eine allgemein deutbare Darstellung der objektorientierten Datenstrukturen (also die Informationen über den Aufbau von Daten) gefunden werden.

Um diese Datenstrukturschnittstelle konzipieren zu können, werden in Kapitel 2 die Strukturen der wichtigsten Datenmodelle sowie in Kapitel 3 deren Beschreibungssprachen auf Möglichkeiten und Nachteile, zur Abbildung der objektorientierten Datenstrukturen, untersucht werden.

In Kapitel 4 werden die Konzepte und Datenstrukturen in der objektorientierten Metadatenbank *„Media Information Repository“* betrachtet. Es werden neben den Anforderungen und vorhandenen Strukturierungsmöglichkeiten auch die erforderlichen Erweiterungen des Systems erarbeitet.

Die Konzeption der Strukturschnittstelle erfolgt im Kapitel 5. Nach einer Einordnung in das Schichtenmodell werden, nach Selektion einer geeigneten Persistenzschicht, die Abbildungs- und Interpretationsmöglichkeiten der Anwendung-, Klassen- und Dokumentstrukturen dargestellt.

Die Kapitel 6 und 7 dokumentieren die Implementierung und den Test auf Funktionalität der entwickelten Software auf der Grundlage des dazugehörigen Meta-Datenbankmodells.

Eine Einschätzung der erreichten Ergebnisse und ein Ausblick auf eine mögliche Weiterentwicklung schließen in Kapitel 8 die Arbeit ab.

2 Strukturen in Datenmodellen

Datenbanken benötigen für einen effizienten Datenzugriff ein leistungsfähiges Datenmodell. Der Aufbau der vorwiegend benutzten Modelle orientiert sich zunächst an allgemein gültigen Anforderungen:

Einfacher und effektiver Zugriff auf den Datenbestand Die abgelegten Informationen sollen schnell und einfach abrufbar sein. So werden die Zugriffszeiten der Applikation gering gehalten, was wesentlich die Bedienbarkeit des Systems beeinflusst.

Abbildung von komplexen Such- und Zugriffsoperationen Der Datenbestand soll flexibel ausgewertet und verwaltet werden können. Damit wird die Logik der Anwendung entlastet und ermöglicht so eine effiziente, leistungsfähige Entwicklung.

Vermeidung von redundanten Daten Um Speicherplatz zu sparen und die Flexibilität der Anwendung zu erhöhen, sollen Informationen mehrfach (durch Verweise) genutzt werden.

Alle Modelle werden von der Datenlogik bestimmt. Sie legt fest, wie die Entitäten[1] gruppiert und miteinander verknüpft werden. Damit werden die semantischen Beziehungen als Datenstruktur im Modell abgebildet.

Um die existierenden objektorientierten Strukturen in ein standardisiertes externes Format zu bringen, werden im Folgenden die Eigenschaften der wichtigsten Datenmodelle betrachtet.

2.1 Relationale Datenmodelle

Relationale Datenmodelle sind wegen ihrer Stabilität und Effizienz verhältnismäßig lange im Einsatz[2]. Sie sind durch ihren Aufbau für die Verwaltung großer, gleich

[1] Eine nicht mehr sinnvoll zerlegbare (atomare) Informationseinheit.

[2] Erste relationale Datenbanken: Oracle 2 sowie SQL/DS für System/R (IBM) aus den 1980er Jahren.

strukturierter Datenmengen geeignet[3]. Mit der Structured Query Language (SQL) existiert eine einfache Beschreibung für Datenstrukturen in relationalen Modellen. Sie ist zuerst von IBM in den 1960er und 1970er Jahren entwickelt worden und konnte sich dann als Standard durchgesetzten, der 1992 durch das ISO/IEC-Konsortium[4] als SQL2 (ANSI/ISO 9075-x:1992) und im Jahre 1999 zu SQL3 (ISO/IEC 9075-x:1999) weiter entwickelt wurde.

2.1.1 Datenstrukturierung

In relationalen Datenbanken werden Tabellen zur Definition von Entitätengruppen und Beziehungen für eine semantische Strukturierung benutzt. Die Beziehungen werden durch Festlegung von Primärschlüsseln und deren Aufnahme in Fremdtabellen hergestellt [Heu97, 52+]. Diese Relationen können weitere Integritätsbestimmungen zur Datenkonsitenz-Sicherung besitzen (Dies sind zum Beispiel: die referenzielle Integrität, Löschweitergabe u.a.).

Durch die Anwendung der in [Cod70] definierten Regeln für relationale Modelle werden die durch die Semantik gebildeten Entitätsgruppierungen (Tabellen) für die optimale Benutzung weiter zerlegt und miteinander verknüpft. Dieser Vorgang wird als Normalisierung bezeichnet und dient der Sicherung der Konsistenz der Daten und der Vermeidung von Redundanz. Abbildung 2.1 verdeutlicht die Datenstrukturierung.

Dieses Datenmodell eignet sich, wegen der Verwendung von Tabellen, besonders für die Verwaltung vieler, gleich strukturierter Datensätze. Die Selektions- und Verknüpfungsabfragen zur angepassten Ansicht auf den Datenbestand sind, durch temporär erzeugte Tabellen, effektiv und flexibel.

2.1.2 Datenzugriff

Die meisten Datenbanken unterstützen mittels SQL einfache und komplexe Zugriffsoperationen. In SQL lassen sich neben Basisoperationen, die als CRUD-Operationen bezeichnet (Create, Read, Update, Delete) werden, auch komplexe Abfragen und Operationen definieren.

Von heutigen Datenbanksystemen werden auch programmierbare Schnittstellen unterstützt. Sie werden als *Stored Procedures*[5] bezeichnet und bieten einen gekapselten funktionalen Zugriff auf die Datenbank.

[3] Zum Beispiel: Messwerterfassung, Kataloge und andere häufig wiederkehrende Informationen

[4] International Organisation for Standardization: `http://www.iso.org/`

[5] In der Datenbank enthaltene Funktionen, die über eine Datenbankschnittstelle zur Verfügung gestellt werden. Sie dienen der Kapselung des SQL-Zugriffes.

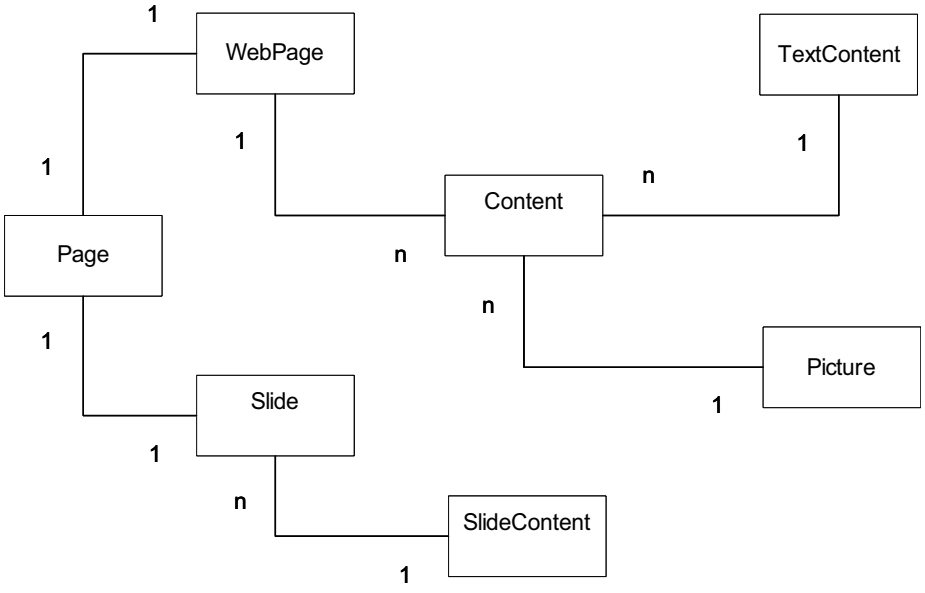

Abbildung 2.1: Schematische Darstellung des relationalen Modells

Der direkte Zugriff mit SQL oder indirekt über Stored Procedures kann durch direkte, herstellerspezifische Software erfolgen. Die Kommunikation mit der Datenbank wird in der Regel über TCP/IP durchgeführt. Einen standardisierten Zugriff ermöglichen entsprechende ODBC-/JDBC-Treiber[6].

2.2 Hierarchische Datenmodelle

Die hierarchischen Datenmodelle wurden seit Beginn der Datenbanksysteme eingesetzt. Sie sind zwar alt, aber nicht altmodisch und aus alltäglichen Anwendungsbereichen, wie zum Beispiel dem Dateisystem, bekannt. In den letzten Jahren sind zusätzliche Verwendungsmöglichkeiten entstanden. So haben sich mit der Verbreitung des Internets textbasierte, hierarchisch aufgebaute „Markup-Sprachen“ als plattformunabhängige Informationsträger[7] etabliert. Die *eXtensible Markup Language* [W3C98] (XML), als Nachfolger der Standarized General Markup Language[8] (SGML), stellt eine flexible, offene, erweiterbare Möglichkeit dar, hierarchische Datenstrukturen abzubilden.

Daraus haben sich die XML-basierten Datenbanken entwickelt, die durch den offenen

[6] ODBC = Open Database Connectivity; JDBC = Java Database Connectivity

[7] Die bekannteste Sprache ist die *Hyper Text Markup Language* [W3C03a] (HTML), als das wichtigste Dokumentformat des World Wide Web.

[8] SGML wurde 1986 von der ISO standardisiert (ISO 8879:1986).

Standard und die hohe Wiederverwendbarkeit die Interoperabilität der Datenbanksysteme verbessern. Die dazugehörige Strukturbeschreibungssprache ist XML-Schema [W3C01a] (XSD) als Nachfolger der Dokument-Typ-Definitionen [W3C98] (DTD).

Alle hierarchischen Modelle stellen Baumstrukturen dar. Durch Verknüpfungen eines Elementes mit einem Eltern- und/oder Kind-Elementen können diese Bäume leichtaufgebaut werden. Sie bieten aber nur eine eingeschränkte Ansicht auf die logischen Zusammenhänge der Daten, da eventuell mögliche semantische Datenstrukturen wie Ring-Beziehungen nicht direkt darstellbar sind.

Die Selektion einer Baumstruktur aus dem semantischen Aufbau der Informationen kann gleich strukturierte Daten im Baum verteilen und erschwert so Abfragen auf diese strukturell gleichen Daten[9].

Bäume bilden exponentiell wachsende Daten in linear wachsende Strukturtiefen ab. Dieser Vorteil macht umfangreiche hierarchische Modelle überschaubar. Abbildung 2.2 verdeutlicht dieses Prinzip.

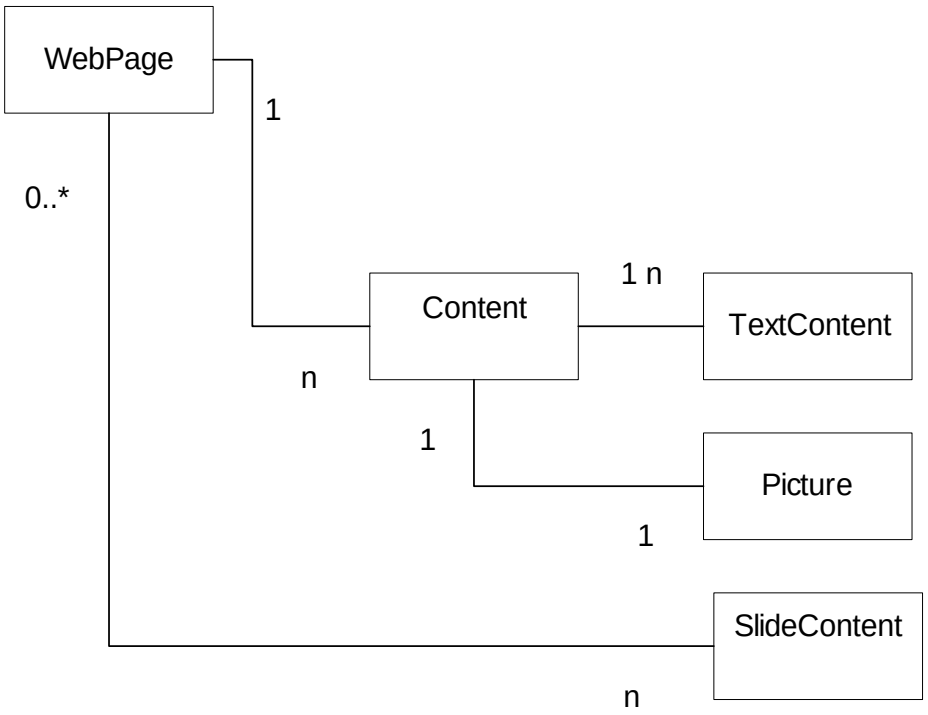

Abbildung 2.2: Schematische Darstellung des hierarchischen Modells

[9]Zum Beispiel die Selektion aller vorhandenen Slides eine Seite aus aus Abbildung 2.2.

2.2.1 Datenstrukturierung

Im hierarchischen Datenmodell werden die Entitäten zu Knoten des Strukturbaumes zusammengefasst. Die Knoten können sowohl eine Gruppierung von Entitäten (Attribute eines Knotens) als auch Container für Substrukturen (Kind-Elemente des Knotens) darstellen. Alle zugehörigen Substrukturen stehen damit direkt als eingebettete, sogenannte komplexe Elemente zur Verfügung.

Ringe und indirekte Verzweigungen innerhalb des Baumes können nur als relative oder absolute Verweise in andere Baumstrukturen dargestellt werden[10]. In Datenbanksystemen, die hierarchische Modelle benutzen, werden, zur besseren Verknüpfung und zur Vermeidung von Redundanz, Attribute mit Identifikationsschlüsseln (ID) versehen. Sie können an geeigneter Stelle als Referenzen (IDREF, IDREFS) eingesetzt werden.

2.2.2 Datenzugriff

Für den Zugriff auf Daten innerhalb des Baumes wird eine relative oder eine absolute Adressierung benutzt. Die Daten können als einzelne Entitäten oder mit gesamten Substrukturen angelegt, abgerufen, ausgetauscht und gelöscht werden.

Hierarchische Modelle ermöglichen neben dem direkten Abrufen von Daten das lokale Navigieren innerhalb der Baumstruktur über die Eltern-/Kind-Elemente und bieten so eine semantisch verknüpfte Ansicht auf die gespeicherten Informationen.

In XML-Dokumenten wird die Adressierung durch XPath[11] und deren Erweiterung XPointer[12] übernommen. XPath wird zur Adressierung ganzer Entitäten und Substrukturen benutzt. Mit XPointer können Teile einer Entität (Fragmente) angesprochen werden.

2.3 Objektorientierte Datenmodelle

Objektorientierte Datenmodelle stellen in objektorientierten Progammiersprachen verwendbare, persistente Objekte zur Verfügung, die die Datenlogik zusammen mit den Daten kapseln und die Vorteile der Vererbung nutzen. Sie unterstützen damit die Anwendungsentwicklung mit den Programmiersprachen dritter Generation und haben für die heutige Softwareentwicklung erhebliche Vorteile.

[10] Eine Verknüpfung im Dateisystem stellt zum Beispiel einen Verweis auf eine Datei dar, die an einem anderen Ort liegt.
[11] XPath-Spezifikation: http://www.w3c.org/TR/xpath
[12] XPointer-Spezifikation: http://www.w3c.org/TR/xpointer

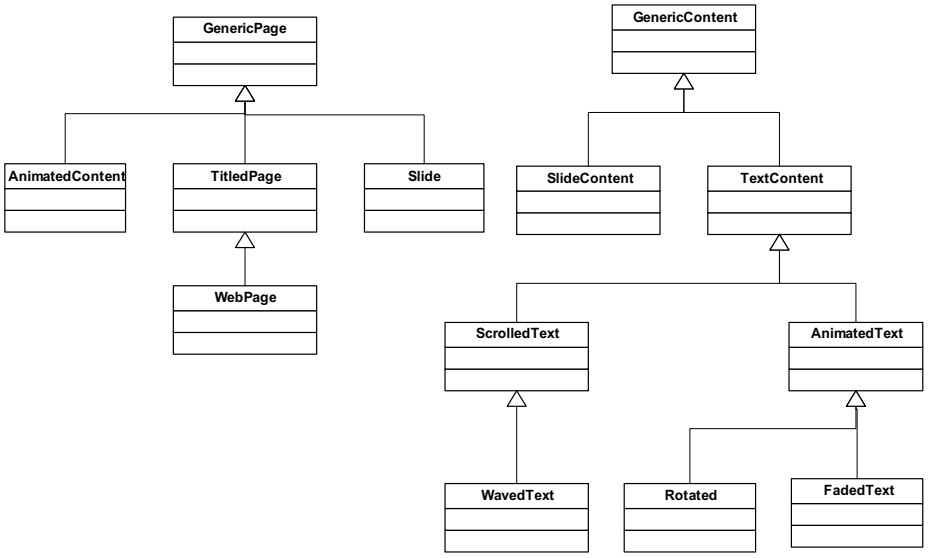

Abbildung 2.3: Vererbungshierarchie im objektorientierten Modell

Für die Beschreibung dieser neuen Modelle scheint sich der Standard ODMG 3.0[13] der *Object Management Group* (OMG) durchzusetzen. Dieser beschreibt abstrakt die in der Datenbank enthaltenen Klassen, die Vererbungshierarchie und die Beziehungen der Objekte untereinander.

2.3.1 Strukturdefinitionen

Analog zu den objektorientierten Programmiersprachen existieren in den objektorientierten Modellen das Vererbungs- und Schnittstellenmuster. Die Vererbungsstruktur gibt die Hauptgruppierung der Objektvarianten vor.

Bei der Vererbung gibt es zwei Ausprägungen. Das Erben der Eigenschaften und Methoden von einer Klasse wird als Einfachvererbung bezeichnet. Das Erben von zwei oder mehreren Basisklassen wird Mehrfachvererbung genannt.

Bei der Einfachvererbung können Zugriffe auf die Eigenschaften und Methoden der Basisklasse ohne Probleme ermittelt und ausgeführt werden. Bei der Mehrfachvererbung gibt es an dieser Stelle verschiedene Hindernisse. Insbesondere bei gleichen Methodendeklarationen und Variablendefinitionen der Basisklassen können Zuordnungsfehler auftreten, die nur umständlich vermieden werden können. Deshalb wird unter anderem im ODMG-Standard auf Mehrfachvererbung verzichtet.

[13] ODMG-Spezifikation : http://www.odmg.org/

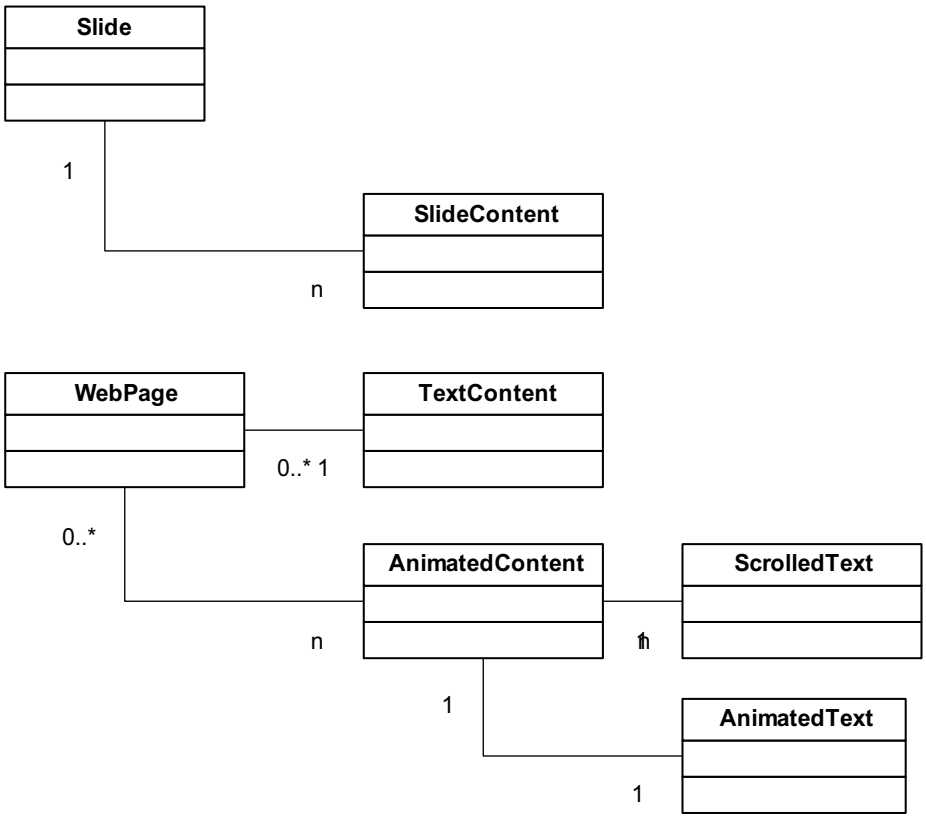

Abbildung 2.4: Anwendungsstruktur in objektorientierten Modellen

Um aber die durch Mehrfachvererbung darstellbaren Eigenschaften der Datenstruktur trotzdem abbilden und nutzen zu können, wurde die Möglichkeit zur Definition von Schnittstellen eingeführt.

In Schnittstellen[14] werden für die applikationsseitige Nutzung Grundeigenschaften bestimmter Objektgruppen zusammengefasst. Damit ist es möglich die meisten Probleme der Mehrfachvererbung zu umgehen. Abbildung 2.3 verdeutlicht die Vererbungsstrukturen.

Die Objekte haben neben der Vererbungsstruktur eine Anwendungsstruktur. Darunter wird die Verbindung von Objekten untereinander[15] verstanden. Sie legt fest in welchen semantischen Relationen sich die Objekte zueinander befinden. Diese Beziehungen werden in Abbildung 2.4 verdeutlicht.

[14]engl. Interface
[15]Zum Beispiel durch typisierte Zeiger oder eindeutige Attibutschlüssel.

```
...

class TitledPage ( keys(Nr)  ):{
  attribute long Nr;
  attribute string Title;
  attribute struct{string Strasse,string Ort, int plz} Addresse;
}

class WebPage extends TitledPage ( keys(Nr) ):{
  relationship TextContent GotText inverse TextContent::TextFromPage;
  int requestCounter() raises( No_Requests);
}

...
```

Tabelle 2.1: Beispiel für Klassendeklarationen in ODL

ODMG unterstützt bidirektionale Verknüpfungen bei der Definition von Beziehungen durch die Deklaration einer in gleicher Relation stehenden Beziehung im verknüpften Klassentyp. Tabelle 2.1 stellt dies beispielhaft in der Object Definition Laguage[16] (ODL) dar. Die damit entstandenen Relationen lassen auch komplexe Anfragen an das Datenmodell zu.

Zusätzlich können Objekte Methoden enthalten, die als aktive Komponenten die Anwendungsentwicklung unterstützen. Sie können spezifische Datenmanipulationen kapseln und so die Anwendungslogik entlasten.

2.3.2 Zugriffs- und Manipulationsoperationen

Datenbanken, die objektorientierte Datenmodelle benutzen, sollen sowohl einfache Zugriffsoperationen als auch Suchoperationen ermöglichen.

Es kann auf kleinster Stufe nur auf ganze Objekte zugegriffen werden. Der Zugriff auf einzelne Entitäten oder zusammengesetzte Ansichten auf den Datenbestand[17], d.h. Zusammenführung von Eigenschaften mehrerer miteinander verknüpfter Objekte, ist nicht möglich. Der Abruf zusammenhängender Objektstrukturen ist ebenfalls problematisch. Bei ungünstiger Anwendungsstrukturierung besteht die Möglichkeit alle Objekte, die in dem Modell enthalten sind, aus der Datenbank abzurufen.

Da die Objekte selbst „aktive Komponenten“ enthalten, sind auch eigenständige komplexe Datenmanipulationen innerhalb der Instanz möglich.

[16] ODL-Spezifikation als Teil des ODMG-Standards: http:/www.odmg.org/
[17] Vgl. View-Konzept in der Structured Query Language.

Für Abfragen und Zugriffe hat die OMG die *Object Query Language* veröffentlicht. Sie definiert, ähnlich dem SQL, Objektfilter für relevante Datenstrukturen.

2.4 Relationale Metamodelle

Metamodelle basieren auf der Simulation eines gewünschten Modells mit Hilfe eines Ersatzmodells. Bei relationalen Metamodellen wird das relationale Modell genutzt, um zum Beispiel ein objektorientiertes oder ein hierarchisches Modell abzubilden.

Der große Vorteil von Metamodellen ist die Möglichkeit von den relationalen Verknüpfungen mit ihren Konsistenzsicherungs- und Abfragemechanismen zu profitieren. Dies ist ein Schritt in Richtung Normalisierung der Information, wodurch zusätzliche Auswertungen effizient umsetzbar werden. Außerdem ist der hohe Verbreitungsgrad entsprechender relationaler Datenbanken, und deren weite Akzeptanz, ein wichtiger Aspekt für den Einsatz.

Die Datenstrukturierung lässt sich in Etappen aufteilen. Zuerst einmal gibt es die Strukturierung des relationalen Metamodells. Sie schafft die Möglichkeit die Datenentitäten aus dem simulierten objektorientierten oder hierarchischen Modellen, die dafür in ihre Bestandteile zerlegt werden, abzuspeichern und bietet somit die Basis für die darzustellenden Modelle.

Die Struktur des simulierten Modells ist innerhalb des relationalen Metamodells mit mehreren sogenannten Metatabellen abgebildet. Dort werden die Attribute und Verknüpfungen in separaten Tabellen verwaltet. Durch die Vertauschung von Spalten und Zeilen wird eine dynamisch erweiterbare Struktur zur Haltung der Entitäten geboten. Dadurch können die Anwendungsstrukturen im Datenmodell flexibel erweitert und angepasst werden.

Auf die abgelegten Objekte wird durch Prozeduren[18] innerhalb der Datenbank zugegriffen, die die entspechenden Operationen im Metamodell ausführen. Applikationsseitig kann auf die Strukturen durch eine Vermittler-Schicht zugegriffen werden. Diese Middleware(MW)-Komponente kann als eigenständige Anwendung konzipiert werden, die einen web-basierten Datenzugriff auf die Datenbank kapselt.

Das hat Vorteile bei der Sicherung des administrativen Datenzugriffes[19], bei der Kontrolle der genutzten internen Ressourcen[20] und bei dem systemunabhängigem Zugang zum DBMS.

[18] Stored Procedures; vgl. Abschnitt 2.1.2

[19] Vermeidung von Angriffen auf den Datenbank-Server.

[20] Durch Mehrfachverwendung von Middleware–Datenbankserver–Verbindungen. Dies wird als *Connection-Pooling* bezeichnet.

2.5 Modellunterschiede

Beim abstrakten Vergleich der Modelle ist festzustellen, dass sowohl das relationale als auch das objektorientierte Modell zwei Datenstrukturierungsmechanismen benutzen.

Der erste Mechanismus zur Strukturierung von Daten ist die Datendefinition. Im relationalen Modell werden atomare Entitäten in Tabellen gruppiert. Im objektorientierten Modell stellen die Klassen und ihre Vererbungsstruktur diesen Teil der Datenstrukturierung dar.

Die Datenverknüpfung ist das zweite Mittel, um die Anwendungsstrukturen abzubilden. Das relationale Modell definiert in den gebildeten Tabellen Primärschlüssel, die zur Verknüpfung mit anderen Datengruppen als Fremdschlüssel eingebunden werden. Das objektorientierte Modell definiert typisierte Zeiger auf andere, in Beziehung stehende Objekte.

Hierarchische Modelle orientieren sich stark an Anwendungsstrukturen. Sie vereinen Datendefinition und Datenstrukturierung miteinander. Die Daten werden als komplexe zusammengesetzte Strukturen modelliert. Direkte Datenverknüpfungen werden als eingebettete Substrukturen dargestellt. Indirekte Verweise, in andere Teile des hierarchischen Modells, sind nur über absolute oder relative Referenzen darstellbar.

Relationale Metamodelle bilden flexibel gestaltbare Entitätsgruppierungen ab[21], die je nach Gestaltung ihrer Anwendungsschnittstelle das abzubildende Modell simulieren. Das Datenmodell sollte transparent die Datendarstellung, mit geeigneten Zugriffsmethoden, dem dargestellten Modell anpassen. Zusätzlich können auch die vorteilhaften Eigenschaften des verwendeten relationalen Modells genutzt werden.

2.6 Zusammenfassung

Datenstrukturen sind für komplexe Verarbeitung von Informationen wichtig. Sie bieten die Möglichkeit Informationen in semantischen Einheiten zu gruppieren und zu verknüpfen. Die logischen Zusammenhänge der Anwendungsdaten sowie deren applikationsseitige Nutzung bestimmen den Aufbau aller Datenmodelle.

Im relationalen Datenmodell werden die Entitäten erst in Tabellen semantisch gruppiert und dann durch Relationen miteinander verknüpft.

Im hierarchischen Modell werden die Entitäten in einer Baumstruktur abgelegt. Die Datenstrukturierung erfolgt hier sowohl durch die Gruppierung innerhalb eines Knotens, als auch durch Verweise zu entfernten Teilen des Modellbaumes. Dies bedeutet eine Vermischung von Datenstrukturierung und Datenverknüpfung.

[21] Durch Verschiebung von Tabellenspalten in Zeilen separater Metatabellen.

Das objektorientierte Modell bildet Klassen, die Entitäten gruppieren. Die Klassen besitzen eine Vererbungshierarchie und können zur Abbildung der Anwendungsstruktur durch Verweise bzw. Zeiger miteinander verknüpft werden. Im Aufbau der Vererbungshierarchie hat sich die Einfachvererbung und die Definition von Interfaces durchgesetzt, um die Probleme der Mehrfachvererbung zu umgehen.

3 Paradigmenübergänge und Strukturpersistenz

Um die objektorientierten Strukturen ex- und importieren zu können, muss eine Beschreibungsmöglichkeit zur persistenten Abbildung gefunden werden. Die zur Beschreibung der Datenstrukturen existierenden Sprachen sollen auf ihre Eigenschaften hin untersucht werden, um ein geeignetes Format für die Darstellung der Strukturen zu finden.

Persistente Strukturbeschreibungen eignen sich zur Weiterverwendung in Applikationen sowie zur Archivierung und Wiederherstellung von Klassenstrukturen[1]. Sie sind standardisiert und werden häufig von bestehenden Applikationen unterstützt.

Die Entwicklung der objektorientierten Programmiersprachen hat mit der Forderung nach einer objektorientierten Persistenzschicht, die Entwicklung verschiedener transienter Systeme zur Überwindung der Modellparadigmen hervorgerufen. Die objektrelationalen Brücken benutzen Mechanismen, die einen objektorientierten Zugriff auf bestehende Datenbestände in robusten, relationalen Datenbanken ermöglichen.

Eine Paradigmenüberführung ermöglicht die Nutzung von anderen, weithin verwendbaren Strukturbeschreibungen. Deshalb werden zunächst die generellen Überführungsmöglichkeiten der einzelnen Modelle ineinander betrachtet. Sie zeigen die Möglichkeiten und Hindernisse bei der Überwindung der in Kapitel 2 erarbeiteten Modellunterschiede.

3.1 Modellübergänge

Um die Anforderung nach Umcodierung von Strukturinformationen[2] zu erfüllen, werden allgemein mögliche Modelltransformationen erarbeitet. Sie unterstützen die Auswahl einer geeigneten Darstellung der Anwendungsstrukturen und zeigen die damit verbundenen Lösungsmöglichkeiten. Insbesondere werden die Transformationen ausgehend vom objektorientierten Modell betrachtet.

[1] Zum Beispiel für Neu-Installationen, Applikationstransport etc.
[2] Vgl. Kapitel 1

Die Methoden der Klassen in objektorientierten Modellen können nicht transferiert werden, da für datengebundene Operationen *Stored Procedures* in das Datenbanksystem eingebracht werden müssen, die ereignisgesteuert (wie das Schreiben oder Abrufen eines Datensatzes) diese Funktionen übernehmen. Die sogenannten *Trigger*, die dann diese Ereignisse generieren, sind ebenfalls ein Bestandteil des Datenbanksystems und müssen auch transferiert werden.

Im Folgenden werden ausgewählte Paradigmenübergänge betrachtet:

Objektorientiert–relational Die Überführung objektorientierter in relationale Strukturen ermöglicht die Nutzung von SQL als Strukturbeschreibungssprache und schafft die Möglichkeit bestehende Daten in ein reines relationales Datenbanksystem zu transferieren. Damit können auch bestehende Anwendungen, die auf relationalen Datenmodellen basieren, ins objektorientierte Modell portiert werden.

Objektorientiert–hierarchisch Hierarchische Modelle sind auf einem einfachen System aufgebaut und haben sich weithin etabliert. Eine Überführung ermöglicht die Wahl aus einer Vielzahl verschiedener Beschreibungssprachen und unterstützt die Anwendungsentwicklung, da hierarchische Modelle sich besonders für dokumentorientierte Anwendungen eignen.

Relational–hierarchisch Dieser Übergang wird hier ebenfalls betrachtet, um die Möglichkeiten einer mehrstufigen Transformation (objektorientiert–relational–hierarchisch) zu erarbeiten.

3.1.1 Objektorientiert-relationaler Übergang

Um ein objektorientiertes Modell in ein relationales zu überführen, werden die objektorientierten Datendefinitionen und Datenverknüpfungen in gleichwertige relationale Strukturen umgeformt.

Aus den in Abschnitt 2.5 dargelegten Äquivalenzen ergeben sich folgende Assoziationen:

- Jede Klasse wird (als Element der Datengruppierung) im relationalen Modell als Tabelle deklariert.
- Die einzelnen Eigenschaften (Attribute) werden als Entitäten in der Klassentabelle, also als Tabellenspalten abgebildet.

- Die durch die Vererbung erzeugten Strukturen werden durch „1:1“-Beziehungen[3] zwischen den Tabellen einer Vererbungsstruktur hergestellt. Dieser Beziehungstyp kann die Normalisierung von relationalen Datenmodellen verletzen. Sie können in einem weiteren Schritt aufgelöst werden. Damit wird wiederum die Rücktransformation der Vererbungsstrukturen erheblich erschwert bzw. ist nicht eindeutig möglich.

- Zeiger (Verweise) sind Datenverknüpfungen und werden als „1:n“-Beziehungen zwischen den Tabellen abgebildet.

- Für Klasseneigenschaften, die als Felder definiert sind, müssen im relationalen Modell als „m:n“-Beziehungen modelliert werden. Dies führt zu weiteren Tabellen, was in der Gegenrichtung nicht mehr wiederherstellbar ist, da separate Tabellen als eigene Klassen definiert werden.

Die Darstellung der Assoziationen zeigt, dass dieser Paradigmenwechsel nur eingeschränkt durchführbar ist.

3.1.2 Objektorientiert-hierarchischer Übergang

Dieser Übergang kann, auf Grund der Vermischung von Datendefinition und Datenverknüpfung im hierarchischen Modell, nur durch eine zusammengeführte Darstellung der Vererbungs- und Anwendungsstruktur abgebildet werden. Es muss aus bestehenden Objektverknüpfungen, die einen Wald darstellen können, ein anwendungsspezifischer, gerichteter Baum selektiert werden, der ein Dokument der Applikation darstellt.

Die folgenden Regeln ermöglichen eine bedingte Überführung der Paradigmen:

- Klassenattribute werden als Subelemente oder Attribute eines Knotens dargestellt.

- Klassen werden als einzelne Knoten (bei Verwendung von Attributen) oder als komplex strukturierte Elemente modelliert.

- Bei der Selektion von Anwendungsdokumenten werden die Vererbungsstrukturen durch die Zusammenführung der geerbten „Elementgruppierungen“ aufgelöst. Damit gehen die Vererbungsstrukturen verloren und können nicht eindeutig wieder hergestellt werden.

[3] Definition durch identische Primärschlüssel in den erzeugten Tabellen.

- Verknüpfungen werden als verschachtelte komplexe Substrukturen dargestellt, wobei auf eine bedingte Verweisverfolgung geachtet werden muss, um Zyklenfreiheit und somit Prozessierbarkeit zu garantieren.
- Jeder Knoten eines Baumes ist von der Wurzel über einen eindeutigen Weg erreichbar. Deshalb können zusätzliche Verweise in der Datenstruktur[4] nicht direkt darstellbare bzw. nicht-referenzielle Verknüpfungen abbilden.

Die Vererbungsstruktur und die Anwendungsstruktur sind nicht gleichzeitig vollständig abbildbar. Auf Grund der Selektion eines gerichteten Baumes muss oft auf Strukturierungsmöglichkeiten (Ringe oder zyklische Beziehungen) des objektorientierten Modells verzichtet werden. Die Rücktransformation hinsichtlich der Vererbungsstruktur ist aus hierarchischen anwendungsorientierten Dokumenten nicht eineindeutig durchführbar.

3.1.3 Relational-hierarchischer Übergang

Bei diesem Übergang muss wieder eine bestimmte Ansicht auf die Daten, die Dokumentenansicht, gewählt werden, da eine vollständige Abbildung des relationalen Modells mit einem hierarchischen Modell nicht möglich ist[5]. Je nach Anwendungsgebiet/Eignung werden, ausgehend von einer Datengruppierung (Tabelle), die relationalen Beziehungen aufgelöst. Das zieht eine Extrahierung eines gerichteten Baumes aus den miteinander verknüpften Tabellen nach sich, bei der auf die Zyklen- und Ringfreiheit zu achten ist.

Folgendes ist beim relational-hierarchischen Übergang zu beachten:

- Jede Tabelle ist eine Datengruppierung, die als Knoten mit Attributen oder als komplexe Elemente im hierarchischen Modell dargestellt wird.
- Wenn „1:1“-Beziehungen modelliert sind, können die einzeln gruppierten Entitäten zusammengeführt oder als Substruktur abgebildet werden.
- Die „1:n“-Beziehungen im relationalen Modell stellen Datenverknüpfungen dar, die als eingebettete Substrukturen deklariert werden.
- Zyklische Beziehungen und Verweise auf im Baum entfernte Strukturen können nur mit Hilfe der unter 3.1.2 gezeigten internen Verweismöglichkeiten dargestellt werden.

Durch die Selektion eines gerichteten Baumes entsteht wieder ein Informationsverlust über semantische Beziehungen der Datenentitäten.

[4] Zum Beispiel durch Referenzen oder über Pfadbeschreibungssprachen wie XPath oder XPointer.
[5] Vgl. Abbildung 2.1 und Abbildung 2.2

3.1.4 Überführung von relationalen Metamodellen

Hier müssen die zwei Ebenen des Datenmodells getrennt betrachtet werden. Da sich das Metamodell durch seine Zugriffsverfahren wie das simulierte Modell verhalten soll, können für den Paradigmenwechsel die, dem dargestellten Datenmodell zugehörigen Überführungsregeln, angenommen werden.

Das relationale Datenmodell, dass die Grundlage bildet, bietet nach einer Überführung wenige nutzbare Möglichkeiten, da durch die Normalisierung im relationalen Modell die originäre Datengruppierung indirekt dargestellt und damit schwer erreichbar ist.

3.2 Strukturbeschreibungen für Datenmodelle

Es haben sich für die betrachteten Datenmodelle persistente Beschreibungssprachen etabliert. Sie dienen dem Transport und zur Verifizierung der Datenstrukturen. Hier wreden für objektorientierte Datenmodelle vor allem die Unified Modelling Language (UML) und die Object Definition Language (ODL als Teil des ODMG-Standards) genannt und vorgestellt.

Die Vererbungsstruktur ähnelt dem (hierarchischen) Aufbau von XML-Dokumenten. Deshalb werden XML-Strukturbeschreibungen (XML Schema oder Dokument-Typ-Definitionen) als weit verbreitete Darstellung hierarchischer Strukturen betrachtet.

Das Ressource Description Framework (RDF) dient zur Beschreibung von Informationen mit maschinen lesbaren, flexibel gestaltbaren Beschreibungsattributen. Durch diese Flexibilität kann RDF die Beschreibung an die Information anpassen und ist damit universell nutzbar.

Die objektrelationalen Brücken finden als aktive Modellübergänge Erwähnung. Es haben sich dort geeignete Modelle für den transienten Strukturwechsel entwickelt. Sie werden kurz vorgestellt, da auch sie Strukturdefinitionen überführen.

3.2.1 Object Definition Language

Die Object Definition Language ist ein Teil des ODMG-Standards. Sie beschreibt Klassen und Beziehungen in einem den ODMG-Standard erfüllenden Format. Diese Strukturspache definiert Schnittstellen und Klassen. Während die Schnittstellendefinition nur die Stellung in der Typhierarchie, Attribute, Beziehungen und Operationen umfasst, enthält die Klassendefinition zusätzlich die Festlegung der Extension[6] und

[6] Anm.: Basisklasse

mehrerer Schlüssel zur Identifikation, sowie die Stellung in der Implementierungshierarchie[7]. In Tabelle 2.1 ist eine Java-ähnliche Syntax erkennbar.

Die ODL beschreibt sehr gut die in objektorientierten Datenmodellen definierten Typstrukturen. Sie kann aber nur zur Übertragung der Datenstrukturen in andere Datenbank-Management-Systeme dienen. Andere Einsatzgebiete, wie die Daten-Modellierung mit externen Werkzeugen, gibt es zur Zeit nicht. Trotz Industrie-Konsortium (OMG) haben die Datenbankhersteller eine sehr unterschiedliche Auffassung über den Aufbau und die Elemente von objektorientierten Datenbanken und implementieren nur eine anteilige Unterstützung des ODMG-Standards.

Da die ODL von Programmiersprachen unabhängig ist, können die Operationen einer Klasse nur als Deklaration abgebildet werden. Eine einheitliche Beschreibung der eigentlichen Funktionalität ist nicht vorhanden und würde eine sehr komplexe Definitionssprache erfordern. Deshalb müssen die Operationen einer Klasse für jedes Zielsystem neu implementiert werden. Eine Erzeugung von ODL-Beschreibungen ist ein spezieller Fall von Code-Generierung.

3.2.2 Unified Modelling Language

UML besteht aus einer Sammlung vorwiegend grafischer Diagramme zur Erstellung von Anforderungs- und Entwurfsmodellen aus verschiedenen Perspektiven. Eine UML-Spezifikation besteht aus einer Menge sich ergänzender und teilweise überlappender Modelle. Im Zentrum steht ein Klassenmodell, das den strukturellen Aufbau eines Systems spezifiziert [Gli03].

Das Klassenmodell beschreibt bei Anforderungsmodellen die Gegenstände der Realität, mit denen das System umgehen muss. Bei der Modellierung von Anforderungen kommt als zentrales Element das Anwendungsfall-Modell hinzu, das die Benutzer-System-Interaktion aus Benutzersicht modelliert.

UML wird in fünf unterschiedliche Diagrammtypen untergliedert:

Statische Sicht Klassen und Objekte, strukturelle Beziehungen

Benutzersicht Anwendungsfälle

Verhaltenssicht Zustandsautomaten

Aktivitätssicht Ablauf von Aktivitäten

Interaktionssicht Interaktion ausgewählter Objekte

[7] Vgl. [Heu97, 445–449]

Gliederungssicht Portionierung der Modelle in Pakete und Subsysteme

Physische Sicht Physische Systemstruktur

Für diese Arbeit ist vor allem diestatische Schicht interessant. Sie kann sowohl Interfaces, Klassen und Vererbungstrukturen beschreiben, als auch Verknüpfungen in Form von typisierten Zeigern abbilden. Es können ebenfalls Restriktionen und Kardinalitäten für die Eigenschaften festgelegt werden.

Mit UML lassen sich so komplexe Anwendungsstrukturen modellieren und durch geeignete Werkzeuge, mittels Code-Generierung, auch in „fast“ fertige Klassen einer objektorientierten Programmiersprache umwandeln.

Für UML-Klassendiagramme existiert eine XML-basierte Beschreibungssprache, die XML Metadata Interchange[8] (XMI). Sie kann als standardisierte, leicht prozessierbare Markupsprache für die Abbildung der objektorientierten Strukturen genutzt werden.

3.2.3 XML-Strukturbeschreibungen

Die eXtensibe Markup Language (XML) ist eine Sprache zur Erzeugung von domainspezifischen Auszeichnungssprachen, die in ihren konkreten Ausprägungen als modernes Datenaustauschformat dienen. Sie ist durchgehend hierarchisch strukturiert und bietet damit sehr gute Voraussetzungen für einen effizienten Zugriff auf die in XML abgelegten Informationen.

Für einen automatisierten, maschinenorientierten Datenaustausch ist es oft von großer Notwendigkeit, Informationen zu den vorliegenden Datenstrukturen zu erhalten. Das Standarisierungskonsortium des World Wide Web[9] (W3C) hat 1998 mit den Document Type Definitions [W3C98] (DTD) eine einfache Strukturbeschreibung für Auszeichnungssprachen annonciert. Ursprünglich für SGML konzipiert, beschreiben DTDs generell den Aufbau von Markup-Sprachen — also auch von XML.

Dokument-Typ-Definitionen definieren nur den Aufbau der XML-Datei und enthalten keine Informationen zu den Entitäten selbst. Außerdem sind Sie in sich nicht XML-konform und deshalb mit gegebenen Zugriffsmitteln nur schwer analysier-/veränderbar. Deshalb lassen sich mit Hilfe von DTDs nur der generelle Aufbau von XML-Dateien und die Vollständigkeit der Informationen analysieren. Sie bieten also die Möglichkeit, die XML-Datei auf Wohlgeformtheit[10] und Gültigkeit[11] zu überprüfen. DTDs können nur beschränkt Dateninhalte beschreiben.

[8] Referenz: `http://www.w3c.org/XMI/`
[9] http://www.w3c.org
[10] Wohlgeformte XML-Dokumente entsprechen den XML Syntax Regeln.
[11] Gültige XML-Dokumente sind nach einer in XSD- oder DTD-definierten Struktur aufgebaut.

```
[...]
<xs:complexType name="WebPage">
<xs:complexContent>
  <xs:extension base="TitledPage">
  <xs:sequence>
   <xs:element name="meta-key" type="String" minOccurs="1" maxOccurs="unbounded"/>
   <xs:choice  minOccurs="1" maxOccurs="unbounded">
    <xs:element name="content" type="TextContent"/>
     [...]
   </xs:choice>
  </xs:sequence>
 </xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="TextContent">
<xs:complexContent>
  <xs:sequence>
    <xs:element name="text" type= "xs:anyType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexContent>
</xs:complexType>
...
```

Tabelle 3.1: Beispiel einer XML Schema Definition

Um diesen Umstand zu beheben, wurde 1998 die XML Schema Definition [W3C01a] (XSD) vom W3C Konsortium verabschiedet. Sie ist selbst in XML definiert und ermöglicht so die interne Verarbeitung mit Hilfe der bestehenden Zugriffsmöglichkeiten auf XML-Daten. XML Schema Definitionen sind leicht in Anwendungsprogrammen auswertbar und veränderbar. Tabelle 3.1 zeigt eine mögliche Definition von Elementen.

Besonders hervorzuheben ist die genaue Möglichkeit zur Spezifikation der einzelnen Entitäten durch das XML Schema. Das bedeutet, dass mit XSD die Möglichkeit besteht, für Entitäten Datentypen, Wertebereiche und Auftretenshäufigkeiten festzulegen.

Datentypen können spezialisiert, eingeschränkt, erweitert oder sogar völlig frei entworfen werden.

3.2.4 Resource Description Framework

Das Resource Description Framework (RDF) ist ein Standard des W3C. Es dient zur Beschreibung von Ressourcen, also Informations- oder Datenquellen in vernetzten Systemen, insbesondere dem World Wide Web. Insbesondere Suchmaschinen erkennen die durch RDF ausgezeichneten semantischen Zusammenhänge, zum Beispiel in HTML-Dateien, und werten diese maschinenorientiert aus.

RDF ist eine Markup-Sprache, die entweder separat abrufbar ist oder in anderen, ähnlichen Sprachen (HTML, XML, SGML, u.a.) hinzugefügt werden kann, um die dort abgelegten Informationen semantisch zu beschreiben. Diese zusätzlichen Informationen über die eigentlichen Daten können zur inhaltlichen Prüfung und Weiterverarbeitung dienen.

RDF unterteilt sich in drei Sektionen zur Beschreibung einer Datenquelle. Das ist zum einen das Datenmodell (RDF Model), eine Syntaxbeschreibung (RDF Syntax), und ein Schema (RDF Schema) zur Definition eines anwendungsspezifischen Vokabulars.

```
<rdf:RDF>
  <rdf:Decription about="meineURI">
    <myNS:Name xmlns:myNS="http://description.org/myNS_Schema"/>
    <myNS:class>
      <myNS:className>WebPage</myNS:className>
        <myNS:classProperty>
          <myNS:PropertyName>content</myNS:PropertyName>
          <myNS:PropertyType>TextContent</myNsPropertyName>
        </myNS:classPropert>>
      </myNS:class>
    </myNS:Name>

    <myNS:class>
      <myNS:className>TextContent</myNS:className>
        <myNS:classProperty>
          <myNS:PropertyName>text</myNS:PropertyName>
          <myNS:PropertyType>
            <myNS:String/>
          </myNsPropertyName>
       </myNS:classProperty>
     </myNS:class>

  </rdf:Decription>
</rdf:RDF>
```

Tabelle 3.2: Beispiel einer RDF Ressourcen Definition

Folgend werden diese drei Säulen, auf denen RDF-Ausdrücke[12] aufbauen, kurz erläutert:

RDF Model Dieses Datenmodell einer syntax-unabhängigen Form von RDF-Ausdrücken spezifiziert drei Objekttypen: Ressourcen, Eigenschaften und Aussagen. Ressourcen stellen die zu beschreibenden Datenquellen dar, die über eine URI mit optionalen Identifier[13] spezifiziert werden. Die Eigenschaften beschreiben ein oder mehrere Merkmale (Attribut, Relation, bestimmter Aspekt, Charakteristikum), die im RDF Schema definiert sind. Aussagen stellen die Zuordnung eines Merkmals zu einem Wert (Literal) dar und bilden somit eine nähere Kennzeichnung der Ressource ab.

RDF Syntax Die RDF Syntax Spezifikation definiert einen XML-basierten Syntax zur Umsetzung des RDF Datenmodells.

RDF Schema Die RDF Schema Spezifikation schreibt vor, wie ein bestimmtes Vokabular zur Beschreibung von Ressourcen gebildet wird. Ein Vokabular ist hierbei eine Menge von wohldefinierten Eigenschaften der zu beschreibenden Ressourcen. Somit ist eine sehr anwendungsbezogene Beschreibung von Ressourcen möglich[14].

Tabelle 3.2 verdeutlicht beispielhaft die Beschreibung einer Ressource in RDF. Es wird über die *Ressource* „meineURI“ eine *Aussage* mit der *Eigenschaft* „class“ aus dem Namensraum „`http://description.org/myNS_Schema`“, das komplex weiter strukturiert ist, getroffen.

Eine detaillierte Beschreibung von RDF ist in [W3C03d, W3C03b, W3C03c] zu finden.

3.2.5 Structured Query Language

Die Structured Query Language (SQL) wird kurz mit ihren Abbildungsmöglichkeiten vorgestellt. SQL kann die relationalen Datenstrukturen hinreichend abbilden. Ein Beispiel für eine SQL-Deklaration einer Tabellenerzeugung mit Definition von Primär- und Fremdschlüsseln ist in Tabelle 3.3 dargestellt.

[12] In RDF formulierte Aussagen über eine Ressourcen.
[13] Zum Beispiel ein Anker auf einer Webseite.
[14] Das ermöglicht eine ständige Anpassung an die speziellen Eigenschaften einer Ressource.

```
CREATE TABLE WebPage_Table
    (w_zNr LONG CONSTRAINT w_primaryKey PRIMARY KEY,
    w_tTitle TEXT)

CREATE TABLE TextContent_Table
    (tc_zNr LONG CONSTRAINT tc_primaryKey PRIMARY KEY,
    tc_w_zNr LONG CONSTRAINT tc_ref_w REFERENCES WebPage_Table(w_zNr))

CREATE TABLE WebPageContainsKey_Table
    (wk_zNr LONG CONSTRAINT wk_primaryKey PRIMARY KEY,
    wk_w_zNr LONG CONSTRAINT wk_ref_w REFERENCES WebPage_Table(w_zNr))
    wk_k_zNr LONG CONSTRAINT wk_ref_k REFERENCES Key_Table(k_zNr))

CREATE TABLE Key_Table
    (k_zNr LONG CONSTRAINT k_primaryKey PRIMARY KEY,
    k_tText STRING)
```

Tabelle 3.3: SQL-Deklaration von Tabellen mit Primär-/Fremdschlüsseln

Sie basiert auf Aussagen, die von der Datenbank-Engine prozessiert werden. Es können neben Abfragen auch das relationale Modell betreffende Festlegungen gemacht werden.

SQL kann so zur automatischen Erzeugung und Konfiguration von relationalen Datenbanken benutzt werden. Eine Standardisierung wurde von der ISO in den Jahren 1992 und 1999 als ISO/IEC 9075-x:199x vorgenommen.

3.2.6 Objekt-relationale Brücken

Objekt-relationale Brücken ermöglichen einen objektorientierten Zugriff auf Daten in relationalen Datenmodellen, indem sie versuchen, die eigentlichen Datenmodellstrukturen (das relationale Datenmodell) als progammiersprachentypische Objekte zur Verfügung zu stellen.

Wegen der Vielfalt der Programmiersprachen werden hier die Möglichkeiten, die im Java-Umfeld existieren, kurz vorgestellt.

Bei der ersten Variante wird in jedem Applikationsobjekt zusätzlich die Erzeugung seiner Persistenz implementiert, die die passenden Datenbankoperationen (Stored Procedures/SQL) mit Hilfe eines Brokers[15] ausführt. Der Broker sorgt mit einer

[15]Klasse zur Verwaltung von Ressourcen.

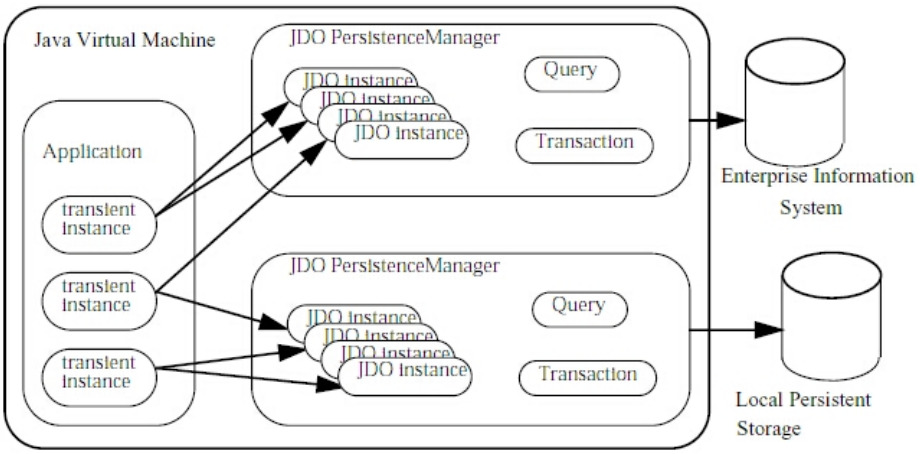

Abbildung 3.1: Java Database Objects - Systemarchitektur

Transaktionsverwaltung für einen gesicherten Transport. Die Methode wird als *Persistance Broker Architekture* bezeichnet und ist in Java durch die Java Database Objects (JDO) repräsentiert. Abbildung 3.1 aus [Rus02] zeigt die Architektur dieser Variante am Beispiel von JDO.

Bei der zweiten Variante wird für jede Tabelle eine Klassen implementiert, in der über die Klassenattribute auf die Entitäten eines Datensatzes zugegriffen werden kann. Dieses sogenannte objekt-relationale Mapping wird in Java durch die Object Relational Bridge[16] (OJB) dargestellt. Die Objekte, die in einem Container laufen, werden als Entity Java Beans (EJB) bezeichnet. Der Container dient als Proxy und kapselt so den Zugriff und die Verwaltung der EJB. Wenn die Zuweisung von Klasse zu Tabelle und Klassen-Eigenschaft zu Tabellen-Entität durch ein XML-Dokument beschrieben ist, können EJB und Container generiert werden.

3.3 Bewertung der Strukturbeschreibungen

Die Object Definition Language und XMI als XML-Repräsentation für die Unified Modelling Language eignen sich beide gut, um objektorientierte Datenstrukturen abzubilden. Leider haben beide eine sehr dediziertes Einsatzgebiet.

Die ODL ist wegen ihrer gezielten Entwicklung zur Abbildung objektorientierte Datenmodelle besonders gut zur Darstellung dieser Strukturen geeignet. ODL kann wegen dieser speziellen Ausrichtung andere Systemteile nur geringfügig unterstützen. Die großen Datenbank-Hersteller (wie Oracle, Sybase, u.a.) versuchen jedoch vermehrt diese Beschreibungssprache, oft mit eingeschränktem Funktionsumfang, bereit

[16] `http://jakarta.apache.org/ojb/`

zu stellen. Für Java gibt es keine speziellen Parser für die ODL-Metasprache, was eine Implementierung erschwert.

UML kann zur Modellierung der Klassenstrukturen außerhalb des Datenbanksystems dienen. Diese Funktionalitäten werden aber oftmals von Datenbankmanagement-Werkzeugen abgebildet. Durch den Import bestehender UML-Diagramme können durch die Software-Entwicklung modellierten Datenstrukturen automatisiert erzeugt werden. XMI stellt eine XML-Repräsentation der Diagramme dar, die auch in XML-Schema überführt werden können [TK03].

RDF kann besonders gut in den markup-basierten Darstellungsschichten als web-orientierte Ressourcenbeschreibung eingesetzt werden. Durch die Flexibilität der RDF Schema kann eine eigene Beschreibungssyntax erzeugt werden. RDF kann nur als zusätzliche externe Information zur semantischen Verarbeitung genutzt werden.

XML Schema ist eine flexible Beschreibungssprache für XML-Dokument-Strukturen. Durch ihre Fähigkeiten zur Abbildung von hierarchischen Strukturen und zur Spezifizikation von Entitätstypen, unterstützt sie die Validierung von internen und externen Datenstrukturen. Die Erweiterungs- und Eingrenzungsmöglichkeiten der vordefinierten Datentypen bieten eine angepasste Beschreibung der Strukturen. Sie können sowohl im systemseitigen, als auch im applikationsseitigen Teil einer Anwendung benutzt werden.

3.4 Zusammenfassung

Alle Datenmodelle lassen sich regelbasiert transformieren. Die Überführungen können jedoch nicht ohne Informationsverlust durchgeführt werden. Eventuell müssen vor und/oder nach der Transformation die Strukturinformationen angepasst werden, um das erzeugte Datenmodell effizient zu nutzen.

Für Beschreibung der Funktionsweise von Methoden aus dem objektorientierten Modell gibt es keine weithin angenommene standardisierte Beschreibungssprache. Nur die Methodendeklarationen können mit objektorientierten Beschreibungssprachen, wie ODL oder UML, modelliert werden.

Für die Abbildung der einzelnen Informationsbausteine können viele Strukturbeschreibungssprachen genutzt werden. XML Schema und das RDF sind als flexible, anpassbare und erweiterbare Definitionssprachen hervorzuheben, die auch außerhalb der reinen Verwendung als Strukturbeschreibung verwendet werden können.
ODL und UML sind ebenfalls gut geeignet, um die objektorientierten Strukturen darzustellen. UML ist hierbei universeller verwendbar, weil sie im Gegensatz zu OQL nicht explizit zur Abbildung von Klassen entwickelt wurde.

4 Aufbau und Strukturierung des Media Information Repository

Das *Media Information Repository* (MIR) ist das Zielsystem für das die Schnittstelle geplant und ausgeführt wird. Dieses System ist im Kern ein relationales Metamodell, das eine objektorientierte Datenbank abbildet. Zum MIR-System gehören weiter Komponenten zur Kapselung und Steuerung des Zugriffes.

Zuerst erfolgt die Beschreibung der Architektur und der Konzepte des MIR-Systems. Sie dienen als Grundlage für die Konzeption der Schnittstelle. Danach werden die vorhandenen Datenstrukturierungsmechanismen sowie deren Defizite und Erweiterungsmöglichkeiten untersucht.

4.1 Das Media Information Repository

Das Media Information Repository ist eine Datenbank für multimediale Inhalte, die am Rechenzentrum der Fachhochschule für Technik und Wirtschaft (FHTW) Berlin seit dem Jahre 2000 entwickelt wird. Es bildet zusammen mit dem *MIR Application Framework* (MAF) eine webfähige Datenbank mit grafischer Benutzeroberfläche zur Administration der Klassenstrukturen und der Inhaltsobjekte.

Das MIR-System bildet gegenwärtig die Grundlage sowohl für experimentelle Prototypen, als auch für Lern- und Publikationsanwendungen im produktiven Einsatz. Beispielhaft finden folgende Projekte Erwähnung:

FHTW.Web FHTW Webauftritt ab Januar 2004. Die Neugestaltung des Webservers der Fachhochschule stellt die neueste Web-Applikation mit Echtzeit-Publizierungsprozess von statischen und dynamischen Inhalten dar. Das FHTW.Web ist ausgerüstet mit einer umfangreichen Java-basierten Autorensuite zur einfachen benutzerfreundlichen Administration der Webserverinhalte.
`http://www.fhtw-berlin.de/`

Virtual Design Virtual Design ist eine interaktive Lern-und Kommunikationsumgebung für Designer im aktiven Lehreinsatz an der FHTW und an Partnerhochschulen.
`http://www.rz.fhtw-berlin.de/virtualDesign/`

HyLOs Das Hypermedia Learning Objects (HyLOs) Projekt ist eine interaktive Lernapplikation, die kontextgesteuert Lerninhalte auf verschiedenen Ansichten vermitteln kann.
`http://www.rz.fhtw-berlin.de/HyLOs/`

MobIT Media Objects in Time (MobIT) ist die Initialanwendung mit der das MIR-System ins Leben gerufen worden ist. Sie präsentiert webbasierte multimediale Lehrinhalte auf zeit- und ereignisgesteuerter Basis.

4.2 Konzepte des MIR

Die Grundideen des Media Information Repository spiegeln sich in der angestrebten Lösung wider. Dafür werden die Konzepte des MIR-Systems kurz erörtert.

Wie in [Sch03] dargestellt, sind im MIR folgende Konzepte umgesetzt:

- Konsequente Trennung von Struktur, Logik, Inhalt und Design in den Anwendungen unterstützt die einfache Anwendungsentwicklung.
- Modulare, frei strukturierbare und wiederverwendbare Inhaltsbausteine bieten die Möglichkeit diese flexibel zu nutzen und weiter zu verwenden.
- Ein Nutzer- und Rollenkonzept hilft bei der Wahrung eines vollständigen Zugiffsrechtemodells. Dies ermöglicht ein flexibles, geschütztes Zusammenarbeiten mehrerer Nutzer.
- Die Bereitstellung eines generischen, verteilten Autorenzugriffs auf die Inhaltsbausteine unterstützt administrative Nutzer des Systems.
- Eine dynamische Bereitstellung einer XML-Repräsentation der Inhaltsbausteine zur flexiblen Erzeugung gewünschter Darstellungsschichten ermöglicht einen anwendungsneutralen Zugriff auf die enthaltenen Strukturen.
- Durch die Bereitstellung einer flexiblen Anwendungsschicht für die einfache Weiterentwicklung der Anwendungen können bestehende Teil-Lösungen wiederverwendet werden.
- Die Verwendung von offenen, weithin akzeptierten Standardtechnologien erweitert die Akzeptanz und Nutzbarkeit des Systems.

4.3 Architektur des Media Information Repository

Abbildung 4.1 zeigt die offene, multimediafähige 3-Tier-Architektur des MIR-Systems. Ein relationales Datenbankmanagementsystem (z.Zt. Sybase) bildet die Basis für das objektorientierte Datenmodell mit Beachtung aller relationalen Verarbeitungsmöglichkeiten [Sch03].

Eine Middlewareschicht stellt Komponenten für Basisoperationen, die Datenzugriffs- und Sessionlogik, bereit. Sie werden nach dem CORBA-Modell zur Verfügung gestellt und sind in Java implementiert. Auf dieser Schicht befindet sich auch die Unterstützung anderer standardisierter Interfaces:

- IIOP als Kommunikationsprotokoll für CORBA
- XML für einen generischen Datenaustausch
- Java Naming and Directory Interface (JNDI) als Verzeichnisdienst für einen standardisierten Zugriff
- Servlets für den webbasierten *http*-Zugriff

Weiterhin werden folgende systemspezifische Funktionalitäten zur Verfügung gestellt:

- Das MIR Applikation Framework für die Entwicklung des Autorensystems
- Ein XML-Generator mit Cache-Funktionalitäten, welcher als Datenquelle für weitere XML-Prozessierung eingesetzt werden kann

Abbildung 4.1 verdeutlicht die im MIR umgesetzte Architektur, die auch in anderen Webanwendungen strukturell ähnlich aufgebaut sein kann - zumindest die Ebenen: Datenbank als Persistenzschicht, Middleware als Webschnittstelle[1] sowie unterschiedlichste darauf aufbauende Clientanwendungen.

Diese sogenannte 3-Tier-Architektur ermöglicht modulare, skalierbare Webserverdienste mit Datenbankunterstützung.

4.4 Datenstrukturdefinitionen

Die bestehende Datenstrukturierung im MIR-System ist entscheidend für die Planung einer Strukturschnittstelle. Das MIR-System enthält zunächst zwei Objektarten, wobei nur Medienobjekte einer Klassenhierarchie auf Basis der Einfachvererbung unterliegen. Außerdem können Verknüpfungen gebildet werden, die den Aufbau von Anwendungsstrukturen ermöglichen.

[1] Es werden diverse Protokolle unterstützt: IIOP, CORBA, XML und diverse, aus XML durch Stylesheet-Prozessierung erzeugbaren Formate, wie HTML, XHTML, PDF, u.a.

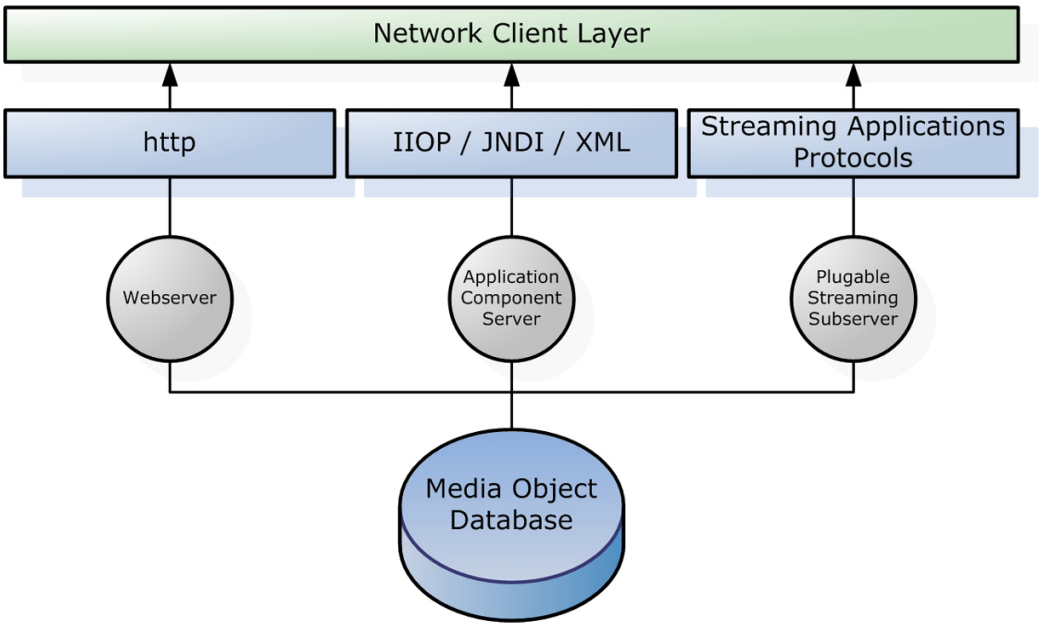

Abbildung 4.1: 3-Tier-Architektur des MIR

4.4.1 Objektarten

Das MIR-System unterscheidet generell zwischen zwei Objektarten, Medien- und Datenobjekte. Sie haben gemeinsame fest zum System gehörende Eigenschaften. Diese Eigenschaften aller MIR-Objekte sind:

- Lokaler Name und Zugriffspfad im globalen Namensraum[2]
- Datumsangaben (Erstellung, letzte Veränderung, letzter Zugriff)
- Zugriffsrechte
- Nutzerzuordnung

Die Datenobjekte (Dob) enthalten neben den allgemeinen, weitere feste Eigenschaften und Biniärdaten. Diese zusätzlichen festen Eigenschaften beschreiben die enthaltenen Daten und können direkt aus diesen ermittelt werden.

[2]Vgl. Abschnitt 4.5.1

Die festen Eigenschaften bei den Datenobjekten beschreiben:

- Dateninhalt
- Datengröße
- Datentyp als MIME-Typ[3] [Moo93]

Als zweite Objektart werden die Medienobjekte (Mob) angeführt. Diese Objekte besitzen neben den allgemeinen festen Eigenschaften, flexibel gestaltbare Attribute (Properties).

Die Medienobjekte sind innerhalb der Datenbank frei strukturierbar. Sie enthalten, zusätzlich zu den frei festlegbaren Eigenschaften, Targets und Events. Diese sind applikationsseitig eigenständige Objekte, die jedoch im lokalen Objektkontext abgelegt werden und nicht über andere Namensräume abrufbar sind.

Targets bilden die Verweise auf andere Objekte im System ab. Über einen lokalen Namen werden sie einem oder mehreren Attributen des Medienobjektes zugeordnet[4].

Events sind frei strukturierbare Objekte und können vielseitig genutzt werden. Durch ihre lokale Verfügbarkeit eignen sie sich besonders zur Abbildung mengenwertiger Attribute, ohne diese als separate Klassen zu modellieren. Ihr originäres Einsatzgebiet war die frei strukturierbare Konfiguration der zeit- und ereignisgesteuerten Medienpräsentation der MobIT-Anwendung. Sie werden in [TCS03] auch für die Abbildung von Ankern im Link-Konzept des MIR verwendet.

4.4.2 Klassenstrukturen

Der Konzeptansatz der MIR-Klassenstrukturen bildet eine weitgehend allgemein gültige Basis für die Modellierung multimedialer Informationsstrukturen: Unter Wahrung einer (Daten-)Typen-und Relationslogik spiegeln sich Anwendungslogiken in vererbbaren Objektklassen wider, welche applikationsspezifisch mit Hilfe eines graphischen Modellierungswerkzeuges, dem *Class Manager* des MAF, erstellt werden können[5].

Dabei ist hervorzuheben, dass das objektorientierte Informationsmodell mit relationalen Leistungsmerkmalen, insbesondere den Prinzipien der referentiellen Integrität, verbunden ist. Die Integritätsbeziehungen lassen sich ebenfalls mit Hilfe des Klassenmanagers deklarieren [Sch03].

[3] Zu Beginn war die „Multipurpose Internet Mail Extension“ für die Inhaltsspezifikation von per Mail versendeten Dokumenten gedacht.

[4] Sie bilden damit den lokalen Namensraum im Datenbanksystem.

[5] Vgl. Abbildung 4.2

Das Prinzip der Einfachvererbung[6] wird nur auf Medienobjekt-Klassen angewendet. Datenobjekte enthalten reine Biniärdaten und sind deshalb strukturell gleich aufgebaut, was die Vererbung obsolet macht.

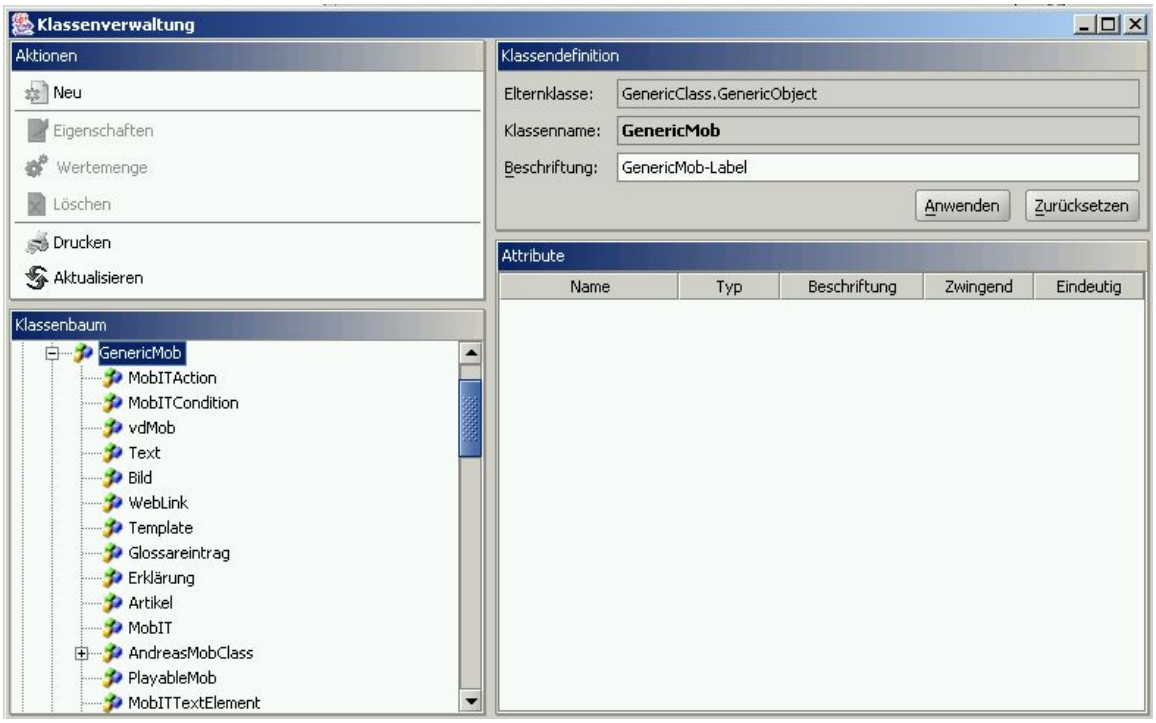

Abbildung 4.2: MIR-Klassenverwaltung

4.4.3 Verknüpfung von Objekten

Die Verknüpfung von Objekten erfolgt durch sogenannte Identifier-Attribute einer Klasse. Der Verweis auf ein anderes Objekt wird durch ein *Target* hergestellt. Ein Target-Objekt hat eine Referenz auf das zu verknüpfende Objekt und einen lokalen Bezeichner (Identifier), der dem entsprechenden Attribut zugeordnet wird.

Da einzelne Attribute einer Medienklasse atomar sind, werden durch Hilfsobjekte (Targets) Tupel von festen Attributen abgebildet, die einen Verweis spezifizieren.

Die Eigenschaften eines Targets sind:

- Lokaler Name zur Verknüpfung mit dem Attribut
- Spezifikation des Zielobjektes im globalen Namensraum

[6] Vgl. Abschnitt 2.3

- Typ des Objektes

Das Metamodell bedingt die Angabe eines Target-Typs, da Medienobjekte und Datenobjekte in getrennten Metatabellen gehalten werden[7]. Für Datenobjekte müssen große Biniärdaten (BLOBs[8]) in geeigneten Tabellen eingetragen werden, die Medienobjekte werden dagegen noch weiter in ihre Attribute aufgesplittet und auf Metatabellen verteilt. Der Typ spezifiziert so die zu erzeugende Referenz im relationalen Metamodell.

4.5 Erweiterung der Strukturierung

Bei der Untersuchung der im MIR-System vorhandenen Datenstrukturen fällt auf, dass die Verweise in Medienobjekten untypisiert, d.h. ohne Angabe einer zugehörigen MIR-Klassenspezifizierung, verwaltet werden.[9]

Im Folgenden werden zuerst die Nachteile der vorhandenen bestehenden Strukturbeschreibungen erarbeitet. Danach werden zur Behebung der Defizite bei der Strukturierung der MIR-Klassen mögliche Erweiterungen des MIR hinsichtlich der Spezifikation des Verweisentitäten zur Abbildung semantischer, dokumentorientierter Datenstrukturen diskutiert.

Nach Betrachtung des Aufbaus des Metamodells werden folgende Varianten zur Abbildung der fehlenden Informationen untersucht:

- Typisierung von Zeigern
- Strukturbeschreibungen durch Pfadausdrücke
- Objektstrukturen als Vorlagen

Des weiteren fehlt den Medienobjekten klassenspezifische, frei gestaltbare Methoden, was erheblich den applikationsseitigen Programmieraufwand erhöht. Für diese Definition von Methoden wird ein Vorschlag zur funktionellen Erweiterung der Klassen erörtert.

[7] Vgl. [Kár02]
[8] Binary Large Objects
[9] Hier ist die Datenlogik vom Datenmodell in den Applikationsteil ausgelagert worden.

4.5.1 Defizite der vorhandenen Datenstrukturierung

Wie schon in den Abschnitten 4.4 und 4.4.3 erläutert, besitzen die MIR-Klassen nur untypisierte Verweisattribute. Sie werden intern durch Targets dargestellt, die so einen lokalen Namensraum bilden.

Der Vorteil untypisierter Verweise ist, dass die Verwendung der Klassenattribute unabhängig von verschiedenen Applikationskontexten möglich ist.

Die fehlende (weil nicht mögliche) Validierung der Verweisattribute auf Gültigkeit der Verweisklasse ist ein großer Nachteil. Es lassen sich Datenstrukturen anlegen, die applikativ nicht mehr prozessierbar sein können. Ohne die Angabe der Zielklasse können keine semantischen Datenverknüpfungen in der MIR-Datenbank abgebildet und demnach auch keine Anwendungsstrukturen definiert werden.

Aus der Sicht der Applikationsprogrammierung sind besonders komplexe Zugriffsverfahren auf ganze, zusammengehörige Datenstrukturen wünschenswert. Die Methoden sollten applikations-(java-)spezifische Objektbäume beim Abruf von Anwendungsstrukturen liefern und Prüfungsmechanismen für deren Speicherung zur Verfügung stellen. Für die Abbildung solcher Strukturen werden weitere Festlegungen hinsichtlich des Aufbaus dieser Dokumente erforderlich.

Des weiteren fehlt den Medienobjekten objekteigene, frei definierbare Methoden. Das erhöht erheblich den applikationsseitigen Programmieraufwand.

4.5.2 Typisierung von Zeigern

Zunächst wird die Typisierung von Zeigern betrachtet. Hier ist eine Erweiterung der Basis-Datentypen, um die Angabe eines MIR-Klassen-Types angedacht. Die Möglichkeit weiterhin kontextunabhängige Verweise innerhalb des Medienobjektes abzulegen, bleibt bestehen, da die verweis-darstellenden Objekte nicht verändert werden. Bestehender Datenbestand kann dabei weiter verwendet werden, denn erst die Zuordnung zu einem Klassenattribut würde eine Typ-Verifizierung auslösen.

Wie in [Kár02] dargelegt, würde das eine Erweiterung der `types`-Tabelle des Metamodells bedeuten. Weiterhin sollte zur Unterstützung von Sammlungen gültiger Klassen-Typen, ähnlich dem `interface`-Begriff des objektorientierten Programmierumfeldes, eine zusätzliche Metatabelle des Datenmodells Kollektionen von Klassen, als Abbildung von logischen Gruppierungen, bilden, die dann als Target-Typ referenziert werden können. Die Zuordnung der Klassen zu diesen Gruppen könnte durch „... ein weiteres Klassenattribut - im Sinne von ‘belongs to’ anstelle von ‘implements’ [erfolgen].“ [Kár02]

Mit der Typisierung von Zeigern können Anwendungsstrukturen aufgebaut werden, die eine Konsistenzsicherung ermöglichen. Sie bieten jedoch keine Möglichkeit Teile einer Anwendungsstruktur für komplexe Zugriffsverfahren zu selektieren.

Vorteile:

- Prüfung der Target-MIR-Klassen
- Einfache Implementierung mit Hilfe des vorhandenen Modells
- Sicherung der Konsistenz von Klassenverknüpfungen bei Zuweisung des Targets
- Aufbau von Anwendungsstrukturen (logische Zusammenhänge)

Nachteile:

- Keine Abbildung von Dokumentstrukturen (semantische Zusammenhänge)
- Eine späte Zuordnung von Target zum Attribut entscheidet über die Gültigkeit des Verweises

4.5.3 Strukturbeschreibungen durch Pfadausdrücke

Eine weitere Möglichkeit stellt eine Strukturierungsebene dar, die auf XPath ähnlichen Pfadbeschreibungen aufbaut.

Mit der Betrachtung der Dokumentstrukturen kann festgestellt werden, dass diese hierarchische Teile der Anwendungsstruktur darstellen. Diese Selektion eines gerichteten Baumes aus der, als Wald interpretierbaren, Anwendungsstruktur führt zu einer bedingten Verweisverfolgung der zur Struktur gehörenden Klassen, in Abhängigkeit von deren Stellung innerhalb der Dokumentenstruktur.

Abbildung 4.3 stellt diese bedingte Auswahl eines gerichteten Dokumentbaumes dar. Die Verfolgung von Verweisen ist von der Stellung einer Klasse innerhalb der Dokumentenstruktur abhängig. Um diese hierarchischen Dokumentstrukturen beschreiben zu können, müssen, beginnend von einer Wurzelklasse, alle gültigen Traversierungen abgebildet werden.

XPath-Ausdrücke beschreiben die Selektion von Knoten einer hierarchischen Struktur sowie die zu überlaufenden Knoten, und zwar ausgehend von einer Wurzelklasse von links nach rechts. Das in XPath reservierte Trennzeichen für die einzelnen Knoten ist der Schrägstrich (/). Für jeden der Knoten können weitere Bedingungen für die Selektion in eckigen Klammern ([]) definiert werden.

Durch Erzeugung mehrerer XPath-ähnlicher Pfadbeschreibungen können angepasste Darstellungen der Anwendungsstruktur abgebildet werden.

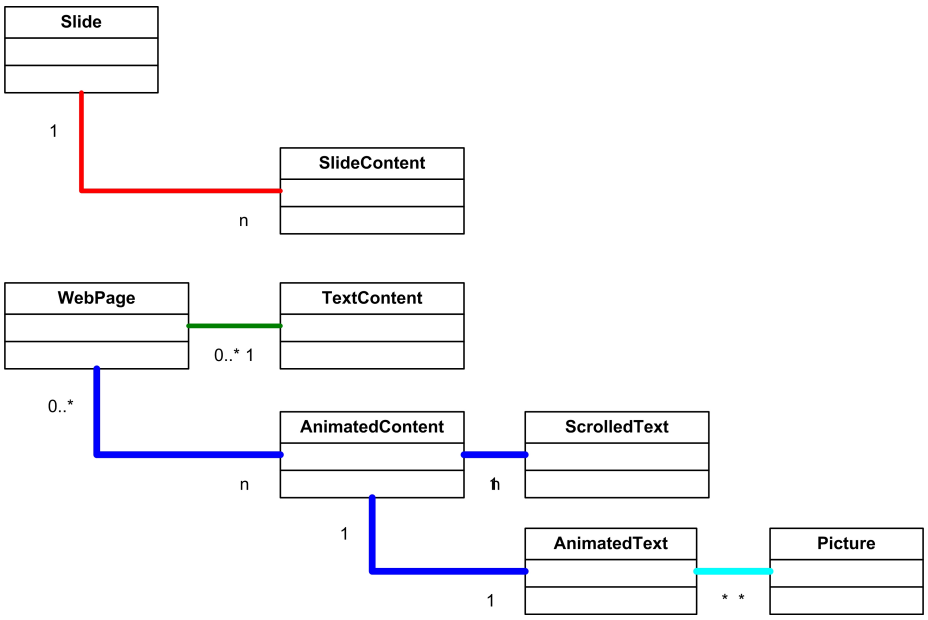

Abbildung 4.3: Selektion von Dokumenten aus Anwendungsstrukturen

Beispiel 4.1 zeigt die passend zur Abbildung 4.3 gebildeten Pfadbeschreibungen. Wie zu erkennen ist, sind die Klassen die Knoten des XPath-Ausdruckes. Als Bedingungen für die Verweisverfolgung werden die zum Dokument gehörenden Verweisattribute (gekennzeichnet durch vorangestelltes @-Zeichen) mit gültigen (Folge-) Klassenbezeichnern angegeben.

So können, wie Beispiel 4.1 zeigt, gleiche Klassen, je nach Stellung in der Dokumentstruktur, unterschiedlich prozessiert werden. Damit werden die Interobjekt-Strukturen hinreichend abgebildet.

Eine Sammlung von Pfadausdrücken bildet eine Dokumentenbeschreibung, die einer Wurzelklasse zugeordnet wird, wobei für jede Klasse beliebig viele dieser Kollektionen möglich sind.

Selbst wenn sie im Metamodell durch eine eigene Tabelle den Wurzelklassen zugeordnet werden, sind sie nicht direkt mit den Klassen und Attributen verbunden. Sie müssen als nebenstehende Eigenschaften der Klasse applikativ administriert werden.

Alle in den Pfaden vorkommenden Bezeichner, ob für Klassen oder Attribute, sind bereits im Klassenmodell der Datenbank. Deshalb ist die weiterführende Abbildung der hierarchischen Pfadausdrücke durch Referenzen auf die Klassen- und Attributdefinitionen des Metamodells denkbar.

Dokument 1:
/WebPage[@content=(AnimatedPage]/AnimatedPage[@aimatedText=(AnimatedText)]/AnimatedText/
/WebPage[@content=(AnimatedPage]/AnimatedPage[@srolledText=(ScrolledText)]/ScrolledText/
/WebPage[@content=(TextContent]/TextContent
/WebPage[@content=(Picture)]/Picture/

Dokument 2:
/Slide[@content=(SlideContent)]/SlideContent/
/Slide[@content=(Picture)]/Picture/
/Slide[@next=(Slide)]/Slide[@content=(Picture)/Picture/
unterschiedliche Darstellungsformen einer Klasse in Abhängigkeit der Stellung in der AW-Struktur:
/AnimatedPage[@next=(Page),prev=(Page)]/WebPage/
/AnimatedPage[@description=(TextContent)]/TextContent[@text=(DOB),@images=(DOB)]/
/AnimatedPage[@content=(TextContent)]/TextContent/

Tabelle 4.1: XPath-ähnliche Ausdrücke zur Dokumentselektion und stellungsbedingte Behandlung gleicher Klassen

Da im MIR-System die Klassenbezeichner eindeutig sind, können keine infiniten Strukturen, wie zum Beispiel zyklische Beziehungen, bidirektionale Verweise oder Ringe, abgebildet werden.

Vorteile:

- Unterstützung der Strukturierung von Anwendungen und ihrer Dokumente
- Von der Vererbungsstruktur unabhängige semantische Beschreibung zusammengehörender Klassen

Nachteile:

- Durch die nebenstehende Zuordnung von Pfadausdrücken besteht keine Konsistenzsicherung[10] der Dokumentbeschreibung
- Darstellung von infiniten Strukturen (Zyklen, Ring, Bidirektionale Verweise) ist nicht möglich

4.5.4 Objektstrukturen als Vorlagen

Die letzte hier dargelegte Möglichkeit nutzt Strukturbäume als Vorlagen. Sie stellt eine einfache Möglichkeit zur Erweiterung der bestehenden Strukturen, durch die

[10]Damit werden die Vorteile des Metamodells nicht genutzt.

Verwendung einer beispielhaften Objektstruktur, dar. Sie kann mit Hilfe von Instanzen, der zur Anwendungsstruktur gehörenden MIR-Klassen, die möglichen, gültigen Objektverknüpfungen abbilden.

Diese können applikationsseitig ausgelesen und verarbeitet werden. Damit ist eine dynamische Änderung der Anwendungsstrukturen möglich, ohne die Klassendefinitionen umzuformen.

Vorteile:

- Modellerweiterungen sind nicht nötig
- Unabhängige, flexible Modellierung der Dokumentstrukturen
- Infinite Beziehungen sind darstellbar
- Bildung der Template-Objekte ermöglicht kontextsensitive Beschreibung der Verweise[11]

Nachteile:

- Kardinalitäten lassen sich nicht abbilden
- Klassenkonsistenzregeln müssen in den Template-Objekten separat gepflegt werden.

4.5.5 Funktionelle Erweiterung

Objektspezifische Methoden sind funktioneller Bestandteil eines objektorientierten Modells, die dem MIR-System zur Zeit fehlt. Solche Methoden müssen innerhalb eines Objektklasse frei strukturierbar und möglichst dynamisch erweiterbar sein, um ein lebendiges System zu gewährleisten.

Für eine flexible Code-Ausführung sollte auf ein Interpreter-Sprache, wie zum Beispiel die Bean-Shell[12] zurückgegriffen werden. Damit kann der Quellcode über Objektinstanzen hinweg weiter gepflegt oder ausgetauscht werden.

Wenn die Ausführung der Methoden clientseitig durchgeführt wird, kann ein hoher Datentransfer zwischen Server und Client vermieden werden. Dabei entstehen wiederum Probleme, da die Verfügbarkeit und Kompatibilität des Code-Interpreters sichergestellt werden muss.

[11] Die Template-Strukturen bilden den applikativen Kontext ab.
[12] Shell-ähnlicher Java-Interpreter. `http://www.beanshell.org`

Die Abbildung der Methoden, als *Large Data Objects*, kann durch eine Erweiterung des relationalen Metamodells um eine Zuordnung von Datenobjekten zur entsprechenden MIR-Klasse erfolgen.

Durch eine Erweiterung der medienobjekt-repräsentierenden Java-Klasse `GenericMob` kann der Quellcode ausgeführt werden. Da sich die `GenericMob`-Klasse im Rahmen des *MAF* befindet, könnte die Verfügbarkeit des Code-Interpreters sicher gestellt werden. Eine neue generische Methode, die den Namen und die Parameter der Funktion übernimmt, führt den Quellcode (in der Interpreter-Sprache) aus.

Abbildung 4.2 zeigt die denkbare Deklaration dieser Methode in `GenericMob`. Die Ausnahmebehandlung wird an den Applikation-Quellcode weitergereicht.

```
public class GenericMob {
...
public Object[] runMethod(String functionName,
                          Object[] param) throws Exception;
...
}
```

Tabelle 4.2: Funktion zur Ausführung von Objektmethoden

Da der Client zur Laufzeit keine Informationen über die im MIR-System abgelegten MIR-Klassenstrukturen und deren Methoden besitzt, muss eine serverseitige Objektinstanziierung[13] erfolgen, was eine Erweiterung der Middlewarekomponente *„Object Manager“*[14] erfordern würde.

Das bedeutet (abstrahiert) die Definition von Initialwerten für MIR-Klassen, die bei der Objekterstellung durch die Middleware gesetzt werden.

Eine Erweiterung zur serverseitigen Ausführung von Quellcode ist ebenfalls denkbar, wenn eine erhöhte Kommunikation zwischen Client und Server akzeptiert werden kann. Dazu sollten die Funktionsaufrufe im Medienobjekt, ähnlich einem asynchronen Batchmodus, zuerst temporär abgelegt und dann, getriggert durch den Client, durch einen Middleware-Funktionsaufruf auf dem Server ausführt werden. Damit kann eine clientunabhängige Funktionsausführung ermöglicht werden. Dies würde auch eine serverseitige Objektinstanziierung nicht unmittelbar erfordern.

Grundvoraussetzung für diese Erweiterung ist ein konsistenter Code-Interpreter, um die stabile Funktionalität abzusichern.

[13] Hier eignet sich das Fabrikmuster (*Factory pattern*).
[14] Vgl. [Kár02]

5 Konzeption der Schnittstelle

In diesem Kapitel wird auf Basis der Erkenntnisse aus Kapitel 2 und 3 eine Strukturschnittstelle zu der in Kapitel 4 vorgestellten objektorientierten Datenbank mit relationalem Metamodell, dem *Media Information Repository* (MIR), konzipiert.

Zunächst wird eine Einordnung der Schnittstelle in eine der drei Stufen der MIR-Systemarchitektur vorgenommen. Danach wird die Abbildung der vorhandenen Strukturen mit XSD dargestellt. Besonders die mit XML einhergehenden Schema- und Namensraum-Spezifikationen erfordern weitere Festlegungen bezüglich der Unterscheidbarkeit und Zugehörigkeit zu bestimmten Applikationen[1].

Die Datentypen im MIR-System können Restriktionen unterliegen. Diese Informationen[2] müssen ebenfalls exportiert werden, um eine gesicherte Übernahme der Daten zu gewährleisten.

Des weiteren gibt es objektorientierte Ansätze, die in einem hierarchischen Modell nicht abbildbar sind. Welche Klassen- und Attributinformationen nicht exportiert werden können, wird ebenfalls untersucht.

5.1 Selektion der Persitenzschicht

Das MIR-System stellt neben den allgemeinen Forderungen an eine Schnittstelle[3] weitere wünschenswerte Eigenschaften an die zu implementierende Schnittstelle. Diese grenzen auch die Umsetzungsmöglichkeiten ein. Die folgenden Anforderungen resultieren aus den spezifischen Grundkonzepten[4] des MIR-Systems:

- Nutzung einer standardisierten, offenen und akzeptierten Sprache
- Inkrementeller Import und Export von Anwendungs- und Klassenstrukturen
- Dynamischer Zugriff auf die Strukturschicht

[1] Vgl. Abschitt 5.3
[2] Zum Beispiel: Zeichenkettenlängen, Wertebereiche für Zahlen etc.
[3] Vgl. Kapitel 1
[4] Vgl. Abschnitt 4.2

- Unterstützung der Darstellungsschichten und der applikationsseitigen Verarbeitung der Inhaltsbausteine

Daraus folgen die Entscheidungen zu den einzusetzenden Technologien: Eine Abbildung, ähnlich der Zuordnungen innerhalb objekt-relationalen Brücken, scheidet durch das Fehlen eines standardisierten Formates aus. Deshalb wird für das MIR-System eine externe persistente Strukturbeschreibungssprache Verwendung finden[5].

Basierend auf den in Abschnitt 3.3 gezeigten Möglichkeiten, stellt XML-Schema eine sinnvolle Grundlage für die Planung der Strukturschnittstelle dar und wird für die folgende Implementierung genutzt.

5.2 Einordnung ins Schichtenmodell

Die Anordnung der Strukturschnittstelle kann sowohl in der Client-, als auch in der Middleware-Schicht erfolgen.

Für die Konzeption als Komponente der Middleware spricht die damit verbundene Bereitstellung von Basistechnologien. Auch die damit einhergehende Unabhängigkeit vom Java-Clientsystem ist ein Vorteil für künftige Entwicklungen und Zugriffe außerhalb der Java-basierten Entwicklungsumgebung MAF.

Eine Einordnung als Teil der Clientsoftware ist ebenfalls möglich, da die Softwareschnittstelle auf den Funktionalitäten der Middleware-Komponente *Class Manager* aufsetzen wird, um die Funktionen zum Zugriff auf die Klassenhierarchie des selbigen zu nutzen. Damit wäre auch die Sicherung der Zugriffsrechte geregelt.

Für die Demonstration der Funktionalität wird eine prototypische, clientseitige Implementierung vorgenommen, die später unkompliziert in die Middleware transferiert werden kann.

5.3 Beschreibung von Dokumentstrukturen

Um den (hierarchischen) Dokumentenansatz abzubilden, werden bis zu der angedachten strukturellen Erweiterung[6], neben den zu exportierenden Klassen, weitere Festlegungen getroffen. Das betrifft im Besonderen die Verknüpfungen der gebildeten Klassenrepräsentationen.

[5] Vgl. Abschnitt 3.2
[6] Vgl. Abschnitt 4.5.3

Nützliche Informationen für die Abbildung eines Dokumenten-Ansatzes:

Dokumentstruktur Bei verknüpften Objekten muss, für eine Selektion eines (endlichen) Dokumentes, die zum Dokument gehörenden MIR-Klassen und deren Verweisverfolgung beschrieben werden. Damit werden, neben der Prozessierbarkeit auch akzeptable Dokumentgrößen und Antwortzeiten erreicht. Außerdem wird die semantische Gültigkeit der erzeugbaren Dokumente gesichert.

Kontext Hier werden Abgaben zur Unterscheidung der einzelnen Dokumente einer Applikation getroffen. Sie können zur lokalen Definition des Ziel-Namensraumes genutzt werden.

Einstiegsobjekt Hierarchische Dokumente benötigen für die maschinelle Verarbeitung die Festlegung eines Wurzelelementes, das den Typ der abgebildeten Datenstruktur definiert. Die Definition eines Einstiegsobjektes ist demnach notwendig.

Namensräume Die Definition von Namensräumen dient der Unterscheidung von Strukturen mit gleichen Bezeichnern. Durch Angabe des Herkunftsortes können die Strukturinformationen eindeutig identifiziert[7] werden.

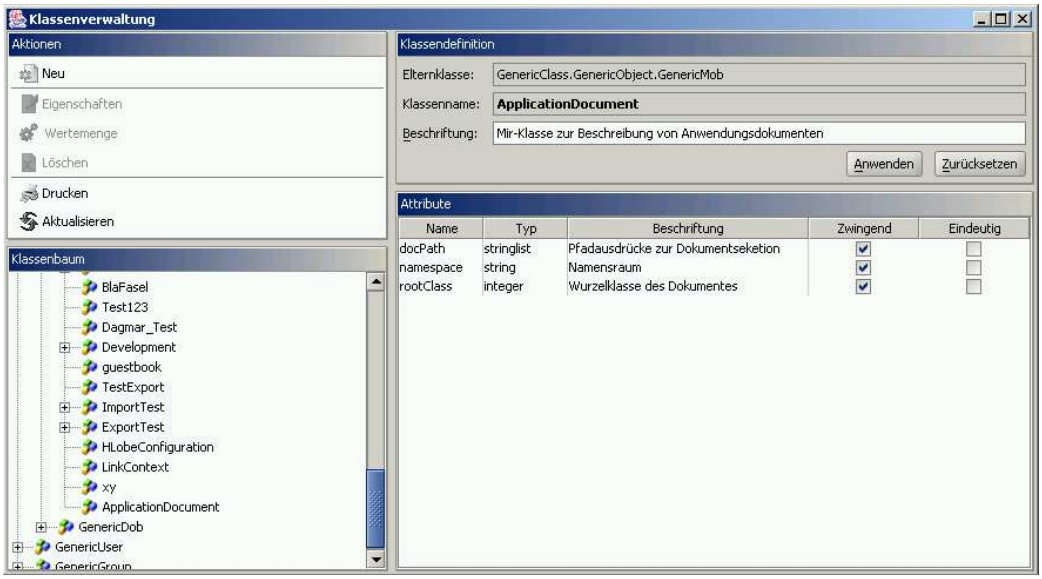

Abbildung 5.1: MIR-Klasse zur Beschreibung eines Applikationsdokumentes

Diese Informationen werden durch fest strukturierte Medienobjekte abgebildet, die ebenfalls in der Datenbank abgelegt werden. Abbildung 5.1 gibt eine Übersicht über eine mögliche Abbildung als MIR-Klasse(n).

[7] Unterscheidbarkeit von anderen ähnlichen Strukturen anderer Quellen.

Die `ApplicationDocument`-Klasse besitzt 3 Attribute:

rootClass Angabe der MIR-Klasse, die das Wurzelelement des Datenbaumes bildet.

namespace Angabe des Namensraumes, in dem die definierten Klassen eindeutig definiert sind (zum Beispiel `http://www.rz.fhtw-berlin.de`).

docPaths XPath-ähnliche Ausdrücke (wie in 4.5 beschrieben) ermöglichen eine nebenstehende, und damit von den Klassen unabhängige, Beschreibung der Dokumentstruktur.

Die XPath-Ausdrücke werden als Zeichenketten dargestellt, die alle gültigen Traversierungen innerhalb der Objektstruktur abbilden. Die gebildeten Regeln werden durch das `docPaths`-Listenattribut zusammengefasst.

5.4 Darstellung und Erkennung von Klassenstrukturen

Als Grundlage für die Modellüberführung dienen die Vorbetrachtungen aus Abschnitt 3.1.2. Hier müssen jedoch, neben den allgemeinen Forderungen, auch auf die Besonderheiten der XML Schema Syntax geachtet werden.

XML ist aus einfachen und komplex strukturierten Elementen aufgebaut. Einfache Elemente stellen atomare Informationsentitäten dar. Komplexe Elemente strukturieren eingebettete, weiter aufgeschlüsselte Datenentitäten. Es ist zu erkennen, dass XML Schema grundsätzlich zwischen diesen zwei Elementtypen, den *simpleType*s und den *complexType*s, unterscheidet.

Die Typendeklaration in XSD bietet die Möglichkeit, definierte, in sich abgeschlossene Datengruppierungen[8] auszulagern. Damit ist die mehrfache Verwendung gleicher Strukturen, ohne Wiederholung der Deklaration möglich.

Objektorientierte Klassen bilden, wie in Abschnitt 2.3 erarbeitet, einen semantischen Container für zusammengehörige, atomare Informationen. Sie müssen demnach als komplexe Typen der XML-Strukturen abgebildet werden. Damit ist die Verwendung einer Klasse an unterschiedlichen Stellen in der Anwendungsstruktur möglich. Die einzelnen Entitäten werden in XML-Schema durch einfache Elemente dargestellt.

Die Verweise auf andere Klassen werden durch Zuordnung des verknüpfenden Elementes (Entität) zu dem komplexen Typ, der durch die Abbildung der möglichen Klasse(n) entstanden ist, hergestellt.

[8] Vgl. [W3C01a, W3C01b, W3C01c]

Die Fähigkeit von XSD zur Beschränkung einfacher Datentypen[9] ist hervorzuheben, weil dadurch diese Randbedingungen des Datenmodells erhalten bleiben und Fehler beim Datenaustausch vermieden werden.

XSD bietet die Möglichkeit Elemente mit Annotationen (Kommentare/Anmerkungen) zu versehen. Diese Eigenschaft wird genutzt, um freie Bezeichner (Labels) der Klassen und Attribute abzubilden. Sie ergänzen die Strukturbeschreibung um von Menschen lesbaren Informationen.

Verknüpfungen werden in XSD-Schema als regelbasierte Einbettungen der als komplexe Typen gebildeten Klassendefinitionen dargestellt.

Tabelle 5.1 gibt einen Überblick auf die Zuordnungsregeln von Klasseneigenschaften zu XSD-Entsprechungen.

MIR-Strukturteil	XML Schema Repräsentation
Eigenschaft mit Basis-Datentyp (DT)	Darstellung als einfaches Element.
Eigenschaft mit restriktivem DT	Aufsplittung in ein einfaches Element mit einfacher Typ-Deklaration (simpleType), der die Restriktionen abbildet.
Eigenschaft als Verweis (Identifier/ Identifier-List)	Einfaches Element mit Angabe des adäquaten komplexen Typs. Eventuell müssen zusätzliche Klassen, die nur auf der Verknüpfungsebene Verwendung finden, ergänzend als komplexe Typen exportiert werden.
Eigenschaften als Listentyp	Die Darstellung der Eigenschaft wird um die Angabe der Auftretenshäufigkeiten eines Elementes erweitert. Dabei ist die maximale Häufigkeit des Elementes unbegrenzt.
Klasse	Erzeugung eines *complexType*s, in dem Klassenattribute als Subelemente dargestellt werden.
Abgeleitete Klasse	Wie „Klasse“, aber mit Kapselung durch XSD Schema Element `<xs:extension>` mit Angabe des komplexen Elterntyps, als `base`-Attribut, von dem geerbt worden ist.

[9] Zum Beispiel als alle ganzen Zahlen von 1 bis 1000 als Beschränkung des `integer`-Datentypes

Kommentare von Klassen und Attributen	Die in Labels abgelegten Kommentare werden durch das XSD-Element `<xs:annotation>` abgebildet, dass als direktes Kindelement des zu kommentierenden Elementes eingefügt werden muss.

Tabelle 5.1: Zuordnungregeln von Klasseneigenschaften zu XSD-Entsprechungen

Die Abbildung der Vererbungshierarchie wird durch Erweiterung eines *complexType*s dargestellt. Beispiel 5.2 zeigt eine erzeugte Vererbungsstufe. Dies wird insbesondere durch die Einfachvererbung ermöglicht, da in XSD für komplexe Typen nur ein einziger *complexType* als Basis der Erweiterung dienen kann.

```
<xs:complexType name="Content">
  <xs:complexContent>
      <xs:sequence>
        <xs:element name="Art" type="xs:string"/>
      </xs:sequence>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="TitledContent">
  <xs:complexContent>
    <xs:extension base="Content">
      <xs:sequence>
       <xs:element name="title" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Tabelle 5.2: XSD-Darstellung von Vererbungstrukturen

Es ist auch die entgegengesetzte Definitionsrichtung (Bottom-Up), durch Restriktion eines existierenden *complexType*s, möglich. Dabei werden, ausgehend von einer Spezialisierung (die in einem complexType definiert wurde), die verbleibenden (gemeinsamen) Attribute definiert.

Für die Implementierung wird jedoch der allgemein übliche Top-Down-Ansatz Verwendung finden, da bei eingeschränkten Typdefinitionen statt der Angabe der wegfallenden Attribute, die verbleibenden Attribute spezifiziert werden müssen. Diese

Darstellungsform verringert nur die Übersichtlichkeit des Schemas, ohne einen weiteren/besseren Informationsgewinn zu erreichen.

Die Erzeugung einer flachen Liste von globalen *complexTypes* stellt eine Normalisierungsschritt dar. Da jetzt ein mehrfaches Auftreten von gleichen (lokalen) Klassenbezeichnern vorstellbar ist, wird ein zusätzliches Namensschema für Klassen benötigt. Durch Bildung eines zusammengesetzten Klassennamens, der aus dem Namen der Elternklasse und dem Namen der Kindklasse zusammengesetzt wird, kann ein eindeutiger Bezeichner gebildet werden. Hier wird die Eindeutigkeit der Elternklasse-Kindklasse-Beziehung, aufgrund der Vererbungshierarchie, genutzt. Die so gebildeten neuen Klassenbezeichner können nur durch Festlegung von reservierten Zeichen[10] eindeutig wieder hergestellt werden.

Dieser Vorgang der Namensbildung muss jedoch solange wiederholt werden, bis keine gleichen Klassenbezeichner gefunden werden, um weiterführende Dopplungen der gebildeten Bezeichner, die sich über mehrere Vererbungsebenen fortsetzen, ebenfalls aufzulösen.

Das dargelegte Namensschema kann durch die Verwendung von voll qualifizierten Typbezeichnern umgangen werden, was aber die Übersichtlichkeit des erzeugten Schemas negativ beeinflusst. Aufgrund der anzunehmenden Eindeutigkeit von lokalen Klassennamen innerhalb einer Anwendungsstruktur, wird bei Bedarf auf das beschriebene Namensschema zurückgegriffen.

5.5 Systemspezifische Datenformate

Wie schon angedeutet, werden von den Standard-Datentypen abweichende Formate als *simpleType*-Elemente dargestellt.

Die im MIR-System verwendbaren Datenformate sind fast alle Standard-Datentypen und können ohne Deklaration von *simpleType*s verwendet werden. Nur bei Zeichenketten gibt es eine datenbankseitige Begrenzung auf eine maximale Länge von 255 Zeichen. Deshalb muss für String-Datentypen eine Restriktion auf Basis der simpleTypes fast immer (bei Verwendung eines String-Attributes) zusätzlich exportiert werden, um die Interoperabilität von Daten, die auf Grundlage des exportierte Schemas erstellt worden sind, zu gewährleisten.

Datentypen, die Aufzählungen (Enumerations) als gültige Datenwerte vorschreiben, können ebenfalls durch XSD als *simpleType*s abgebildet werden. Tabelle 5.3 zeigt *simpleType*s für den Zeichenketten-Datentyp und einem Aufzählungsdatentyp (Enumeration).

[10] Zum Beispiel: Bindestrich, Punkt, Doppelpunkt etc.

```
<xs:simpleType name="String">
  <xs:restriction base="xs:string">
     <xs:length>255</xs:length>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="scrollDirection">
  <xs:restriction base="xs:string">
    <xs:enumeration value="horizontal"/>
    <xs:enumeration value="vertical"/>
  </xs:restriction>
</xs:simpleType>
```

Tabelle 5.3: XSD-Darstellung von einfachen begrenzten Datentypen

Die Datenkonsistenz-Regeln, die durch das relationale Metamodell zur Verfügung stehen, werden so weit wie möglich mit abgebildet.

Die folgende Übersicht zeigt die Abbildungsmöglichkeiten der einzelnen Constraints[11]:

Eindeutigkeit Sie ist nicht exportierbar, weil dazu ein Vergleich mit den bestehenden Daten nötig wäre.

Obligatheit Die zwingende Angabe von Attributen kann über die Auftretenshäufigkeit der Attribute in XSD codiert werden. Das bedeutet: Obligatorische Attribute können kein oder einmal (oder bei Listentypen mehrfach) auftreten. Die Codierung erfolgt, wie bei Listenattributen, durch Spezifikation der minimalen Auftretenshäufigkeit (obligatorische Attribute = 0, sonst 1).

Dimensionierung Die im MIR-System laut [Kár02] bestehende Möglichkeit Datenwerte mit einer Dimension zu versehen, führt in XML-Schema zu begrenzten *simpleTypes*. XSD kann neben typspezifischen Vorgaben auch reguläre Ausdrücke[12] als Formatierungsanweisungen abbilden.

5.6 Nicht exportierbare Datenstrukturen

Es existieren Datenstrukturen, die in einem Schema nicht abbildbar sind. Dazu gehören die Methoden von Objekten. Sie sind wegen den in Abschnitt 3.1 erörterten Gründen nicht exportierbar.

[11] Vgl. [Kár02]
[12] *Regular Expressions*

Außerdem können nicht-referenzielle Verknüpfungen wie Anker, Links, Pfadangaben nicht dargestellt werden, da sie abhängig von der Objektinstanz und ohne direkten Zugriff auf den Datenbestand außerhalb des MIR-Systems ohne Wert sind.

Dieses Defizit kann nur bei Link-Verknüpfungen durch Verweise auf andere, extern abrufbare Dokumente umgangen werden. Deren Aufbau und Gültigkeit lässt sich aber in der Strukturbeschreibung nicht codieren. Da das Referenziel nicht immer das Ausgangsobjekt auf die zu verweisende Anwendungsstruktur sein muss, wird eine anwendungsspezifische Implementierung der Verweisbildung erforderlich.

Auch die Eigentümer- und Zugriffsrechte sind nicht exportierbar, da eine eindeutige asynchrone Zuordnung beim Import nicht immer möglich ist.

Die Eindeutigkeit ist als einzige Integritätsbestimmung des relationalen Metamodells nicht in XSD darstellbar, weil hierfür alle Instanzen einer Klasse betrachtet werden müssen.

5.7 Definitionsvielfalt von XML Schema Definitionen

XML Schema Definitionen bieten die Möglichkeit gleiche Strukturen unterschiedlich abzubilden. Das erhöht zwar deren Flexibilität, erschwert jedoch einen generalisierten Import der abgebildeten Strukturen. Im Folgenden werden die Definitionsvielfalt und die Äquvalenzen gezeigt.

- In XSD können Elemente mit Attributen versehen werden. Sie stellen eine Datengruppierung dar, die auch durch einzelne Subelemente abgebildet werden können.
- Oft sind Datengruppierungen als verschachtelte (also eingebettete), komplexe Typdefinitionen deklariert. Dies entspricht mehr der allgemeinen Vorstellung vom Aufbau strukturierter Dokumente, weshalb diese Ausprägung weit häufiger Verwendung findet als die globale Typ-Definitionen.
- Durch ID-Referenzen kann auf andere, durch IDs markierte Teile einer Dokumentenstruktur verwiesen werden. Dies verringert bzw. verhindert Redundanz (mehrfaches Auftreten) von Informationen. Die Interpretation von Zeigern wird häufig verwendet, wenn XML als Datenbasis genutzt wird. Wird XML zur internen Prozessierung eingesetzt, kann durch gewollte Erzeugung von Redundanz die Effektivität der Weiterverarbeitung erhöht werden, da ein Auffinden der Informationen umgangen wird.
- Eine weitere Möglichkeit von XSD ist die Einbindung von extern spezifizierten Teilen des Schemas. Hier wird Redundanz zwischen XML Schema Definitionen untereinander vermieden und die Administration verbessert.

6 Implementierung

In diesem Kapitel wird die Implementierung der in Kapitel 5 konzipierten Strukturschnittstelle dargestellt. Sie wird in Java, passend zum MIR-System, umgesetzt. Dabei müssen die besonderen Eigenschaften von XML Schema Definitionen und des MIR-Systems beachtet bzw. abgebildet werden.

Zunächst wird der Export von Datenstrukturen erläutert. Dabei ergeben sich Randbedingungen und Darstellungsformen der MIR-Strukturen in XML-Schema.

Die Erkenntnisse aus den Möglichkeiten und Einschränkungen des Exportes von Datenstrukturen werden im weiteren Verlauf für die Anpassung verschiedener möglicher Schemaausprägungen und deren anschließenden Import genutzt.

6.1 Export

Wie in Abbildung 6.1 zu sehen ist, sind für den den Export mehrere Klassen implementiert worden. Die `MirClassExporter`-Klasse bildet die Steuerung der Exportfunktionalität. Ihr kann eine Sammlung der zu exportierenden Klassen übergeben werden. Dies ist insbesondere für den Export von Dokumentstrukturen wichtig, da die zugehörenden Klassen innerhalb der Vererbungshierarchie nicht von einander abhängig sein können.

Die Erstellung der Klassenliste ist je nach Anwendungsfall unterschiedlich. Durch die Definition des `ClassResolverInterface`-Interfaces wird der Exporter unabhängig von der Erstellung der Klassenauswahl und somit modularer.

Zur eigentlichen Erzeugung der `ExportabeMirClass`-Instanzen wurden zwei Class-Resolver implementiert.

Die `InheritanceResolver`-Klasse erstellt die Sammlung von Klassen, die sich in einer Vererbungshierarchie befinden. Durch Zuweisung einer Basisklasse mit Hilfe der `setBaseClass`-Funktion werden diese und ihre Spezialisierungen für den Export selektiert. Dieser Weg ist für die Archivierung und Wiederherstellung von Vererbungshierarchie-Bäumen nützlich.

Die `DocumentResolver`-Klasse analysiert die in Abschnitt 4.5.3 vorgestellten Pfade. Es werden alle vorkommenden Klassen, mit den jetzt typisierten Properties, erfasst.

Im Folgenden werden ggf. die Elternklassen der Dokumentklassen zu der Exportliste hinzugefügt und danach die internen Vererbungstrukturen hergestellt.

Die Funktion `performExport` übernimmt die Ausführung des Exportes. Sie erzeugt einen Ausgabe-Stream, in den die erzeugten Zeichenketten geschrieben werden. Außerdem werden hier die Header und Footer des XML-Schemas erzeugt. Dieser allgemeine Rahmen beinhaltet die Angabe von verwendeten Zeichensätzen und Namensräumen, in denen das Schema definiert ist.

Im weiteren Verlauf wird die Funktionsweise der `MirClassExporter`-Klasse in Bezug auf Klassen, Datentypen und Anwendungsstruktur erläutert.

6.1.1 Klassen

Für die Abbildung der internen Vererbungsstruktur wird die `ExportableMirClass`-Hilfsklasse benötigt, da die klassenrepräsentierenden `MirClass`-Objekte nicht direkt miteinander verknüpft geladen werden können. Der Aufbau des Namensschemas wird dadurch ebenfalls ermöglicht.

Nach der Erstellung der Klassenliste durch den ClassResolver, wird das bereits dargelegte Namensschema für die MIR-Klassen angewendet.

Danach können alle Klassen als globale *complexType*s in das Schema platziert werden. Für diese Prozessierung der `ExportableMirClass`-Objekte ist die `processClass`-Funktion implementiert worden. Sie erzeugt, sich rekursiv an Vererbungsstruktur orientierend, für jede zu exportierende Klasse einen globalen *complexType*[1].

Alle Klassen, die von keiner anderen Klasse der Liste erben, werden mit den bis zu ihrer Vererbungsstufe ererbten Attributen exportiert, damit die Darstellung der Attribute vollständig ist.

Abbildung 6.1 zeigt, wie die einzelnen Funktionen miteinander interagieren.

6.1.2 Klassenattribute, Datentypen und Anwendungsstruktur.

Die Attribute einer Klasse werden, wie schon erwähnt, als Elemente eines *complexTypes* dargestellt. Hier werden auch die Eigenschaften des Attributes, wie Datentyp und Auftretenshäufigkeiten, analog zu den in Tabelle 5.1 gezeigten Zuordnungsregeln abgelegt. Die generelle Erzeugung eines Elementes übernimmt die Funktion `createProperty`, die je nach Datentyp des darzustellenden Klassenattributes die geeignete Darstellungsart wählt.

[1] Falls für die Klassen ein freier Bezeichner (Label) spezifiziert ist, wird dieser als Annotation zum *complexType* hinzugefügt.

Die Erzeugung der Elemente ist in drei unterschiedlich zu behandelnde Datentyp-Gruppen unterteilt:

Einfacher Datentyp In der Funktion `createSimpleProperty` werden alle Klassenattribute als einfache Elemente abgebildet, die mit ihrem Datentyp direkt einem XSD-Datentyp entsprechen. Die Nutzung der Standardtypen ist von Vorteil, um das Schema später möglichst flexibel einsetzen zu können.

Aufzählung-Datentyp Dieser Datentyp erfordert, wie in der Konzeption erarbeitet, eine Erzeugung eines *simpleType*, der alle gültigen Werte des Klassenattributes abbildet. Dieser Sonderfall wird an die Funktion `createEnumProperty` deligiert, die zuerst den globalen *simpleType* erzeugt und dann mit dem XSD-Element verknüpft.

Verweis-Datentyp Bei den Identifier-Attributen zeigt der Typ des angelegten Elementes auf den complexType, der die an dieser Stelle zu verwendende Klasse darstellt. Diese Information wird aus den in Abschnitt 4.5.3 erläuterten Pfadausdrücken entnommen. Falls hier mehrere Klassen möglich sind, muss ein weiterer lokaler *complexType* angelegt werden, der die Auswahl der möglichen Klassentypen kapselt. Die Funktion `createIndentifierProperty` übernimmt die Analyse der Pfadausdrücke. Wenn diese (Zum Beispiel beim Export einer Klassenhierarchie) nicht zur Verfügung stehen, wird an dieser Stelle mit dem XSD-spezifischen Datentyp `xs:anyType` die Zuweisung offen gehalten. Dann ist eine freie Kombination der *complexType*s möglich.

Im MIR-System können Attribute ebenfalls mit einem freien Bezeichner (Label) versehen werden. Sie werden, wenn vorhanden, als Annotationen zu den Elementen in das Schema übertragen.Weiterhin ist das Anlegen eines restriktiven *simpleType*s für den String-Datentyp erforderlich, da seitens des MIR-Systems dieser Datentyp auf eine Länge von 255 Zeichen begrenzt ist.

6.2 Import

Beim Import der in XSD codierten Strukturinformationen werden die durch den Export erstellten Strukturen wieder als Klassen interpretiert. Da in XSD verschiedene Darstellungsformen gleichwertiger Informationen möglich sind, ist eine Anpassung der vorhandenen Schema an die durch den Export erzeugten Strukturen notwendig.

Für den Import ist ein weiteres Klassenmodell entworfen worden. Abbildung 6.2 zeigt eine Übersicht über die implementierten Java-Klassen.

Die Durchführung des Importes wird durch die Klasse `MirClassImporter` übernommen. Einen allgemeinen Rahmen bietet dazu die `MirClassImport`-Klasse. Sie sichert

den Zugriff auf die Datenbank, stellt die zu importierende XSD-Quelle zur Verfügung und legt ggf. eine Basisklasse für die importierten Klassen fest. Damit kann der erzeugte Klassenbaum an beliebiger Stelle einer vorhandenen Vererbungshierarchie eingehangen werden.

Um das vorliegende XML-Schema in eine als MIR-Klassen interpretierbare Form, wie sie der Export erzeugt hat, zu überführen, wird zuerst die Transformation an die `SchemaTransformer`-Klasse delegiert. Sie erzeugt die in Abschnitt 6.1.1 erstellten globalen *complexType*s. Eine Interface-Deklaration (`SchemaTransformerInterface`) ermöglicht, dass sich später andere Darstellungsformen von Datenstrukturen ebenfalls durch den Import prozessieren lassen.

Danach wird in der `MirClassImporter`-Klasse, durch die `buildClassImportObjects`-Funktion, eine Liste mit den zu importierenden Klassen mit allen Attributen erstellt. Die entstandenen Klassen werden durch die Funktion `linkImportableMirClasses` untereinander hierarchisch verknüpft und durch die `updateParentClassSpec`-Funktion mit der passenden Elternklassen-Spezifikation versehen. Dies ist notwendig, um eine gültige Vererbungshierarchie herzustellen und anschließend die Klassen, rekursiv der gebildeten Vererbungshierarchie folgend, mit der Funktion `importClassObject` in die Datenbank schreiben zu können.

Im Folgenden werden die Besonderheiten und Anpassung von XSD-Strukturen an die Export-Darstellungsform untersucht. Danach wird der Import der Klassen, ihrer Attribute und der Dokumentstruktur dargestellt.

6.2.1 Angleichung der Schema-Ausprägungen

Die in Abschnitt 5.7 erläuterten Darstellungsmöglichkeiten werden durch die so genannte `SchemaTransformer`-Klasse an das Exportformat angepasst. Sie implementiert das `SchemaTransformerInterface` und kann so durch die `MirClassImport`-Klasse an den Importer zugewiesen werden. Die Funktionsweise der Transformation wird in Abbildung 6.3 dargestellt.Sie erzeugt folgende importierbare Strukturen:

- Globale *complexType*-Definitionen entstehen aus allen eingebetteten *complexTypes*. Da jetzt eine implizite Zuordnung des Elementes zum Typ aufgelöst wird, ist eine Namensgebung für den globalen Typ notwendig. Die Funktion `transformNested2GlobalTypes` untersucht rekursiv alle Elemente des Schemas mit Hilfe der `hasNestedGlobalType`-Funktion auf eingebettete *complexTypes* und deligiert ggf. die Extraktion an die Funktion `extractComplexType`, die auch die Erzeugung eines eindeutigen Typbezeichners übernimmt.
- Der Austausch von Attributen eines Elementes durch Subelemente führt zur Erweiterung vorhandener bzw. zur Erzeugung neuer *complexType*s. Ein weiterer

complexType entsteht, wenn für das prozessierte Element keine Subelemente existieren. Bei vorhandenen Unterstrukturen werden die Attribute in Elemente umgewandelt und zur bestehenden Sammlung der Subelemente hinzugefügt. Die Funktion `transformAttributes` traversiert wieder rekursiv die Schema-Definition und wandelt mit Hilfe der `attribute2ComplexType`-Funktion die Attribute in Subelemente um.

6.2.2 Klassen, Attribute und Anwendungsstruktur

Nachdem die XSD-Ausprägungen an das Exportformat angepasst worden sind, können, wie in Abschnitt 5.4 beschrieben, die Informationen über die Klassen, entgegengesetzt zum Export, wieder ausgelesen werden.

Die Funktion `importClassObject` interpretiert die globalen *complexTypes* als die zu importierenden Klassen. Die Methode erkennt die Elternklasse sowie Name, Beschriftung und Attribute der Klasse und legt eine Instanz der Hilfsklasse `ImportableMirClass` an, die alle zugehörenden Informationen aufnimmt.

Zur Erkennung der Klassenattribute wird die Funktion `findProperties` benutzt. Diese erstellt die anzulegenden Attribute der MIR-Klasse aus allen Elementdefinitionen unterhalb eines *complexType*s mit Hilfe der Funktionen `resolveElementName`, `resolveElementType`, `resolveElementDescription` und `resolveOccurence`. Die gefundenen Properties werden, für das spätere Schreiben in die Datenbank, einem `ImportableMirClass`-Objekt zugeordnet.

Wie bereits erwähnt, wird die Vererbungshierarchie aus den internen Verknüpfungen des XML-Schemas und der Basisklasse aufgebaut, wobei die Funktion `linkClassObjects` die erzeugten Klassen untereinander verknüpft. Die Funktion `updateParentClassSpec` ergänzt die Vererbungstrukturen um die Basisklasse.

Die Erzeugung einer Dokumentbeschreibung ist nur möglich, wenn ein Wurzelelement spezifiziert ist. Ausgehend vom *complexType* dieser Wurzel wird die gesamte Dokumentstruktur ausgelesen. Die Funktion `buildDocumentPaths` erstellt rekursiv die Pfadausdrücke aus den gebildeten Klassen und typisierten Verweisen. Ein typisierter Verweis ist daran zu erkennen, dass ein Element auf keinen Standard-Datentyp, sondern auf einen im Schema definierten Typ zeigt. Durch Mitführung einer Liste der bereits traversierten Klassen, werden Ringbezüge erkannt und der Aufbau redundanter Pfadstufen verhindert.

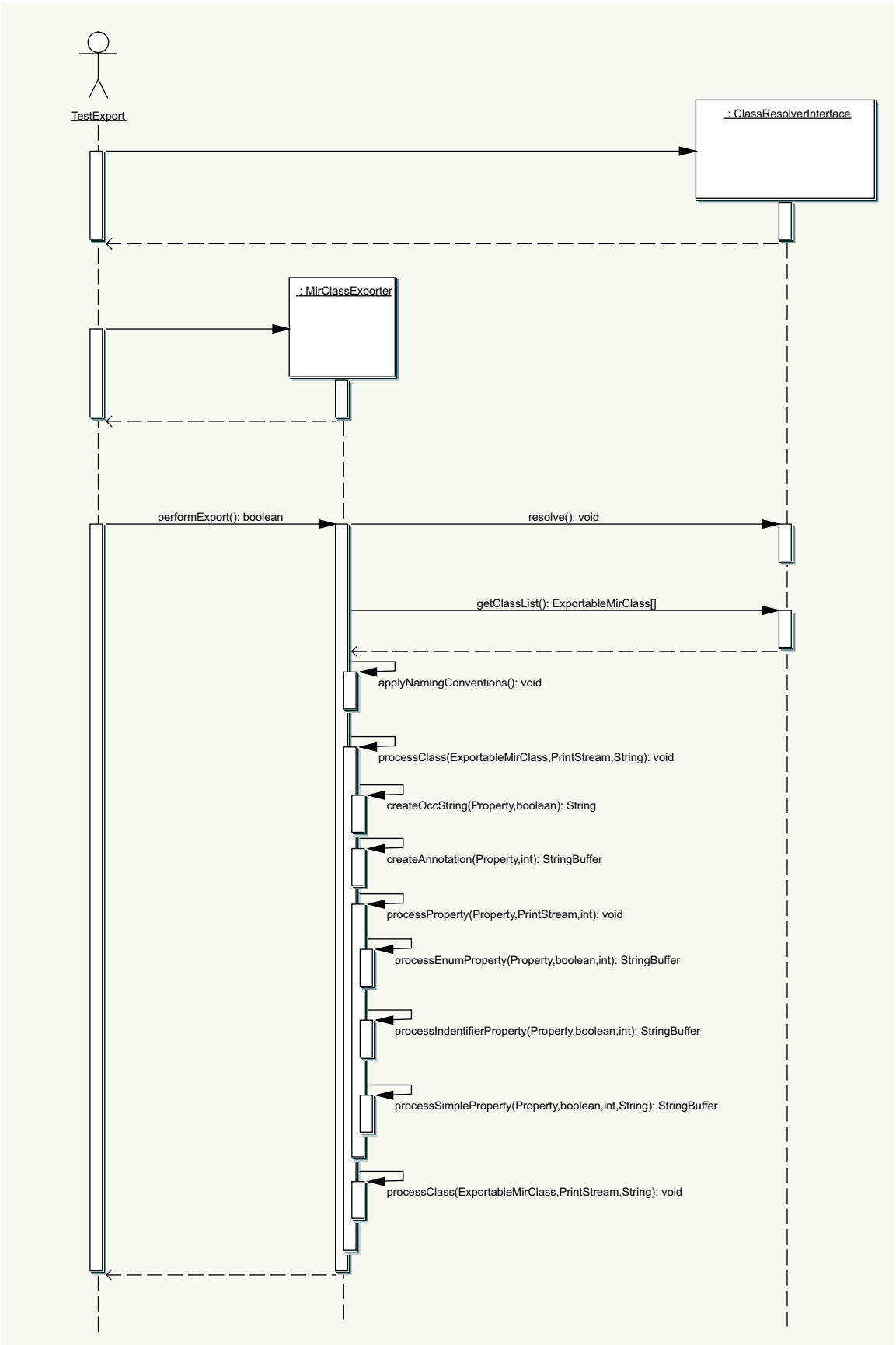

Abbildung 6.1: Funktionsweise des Exporters

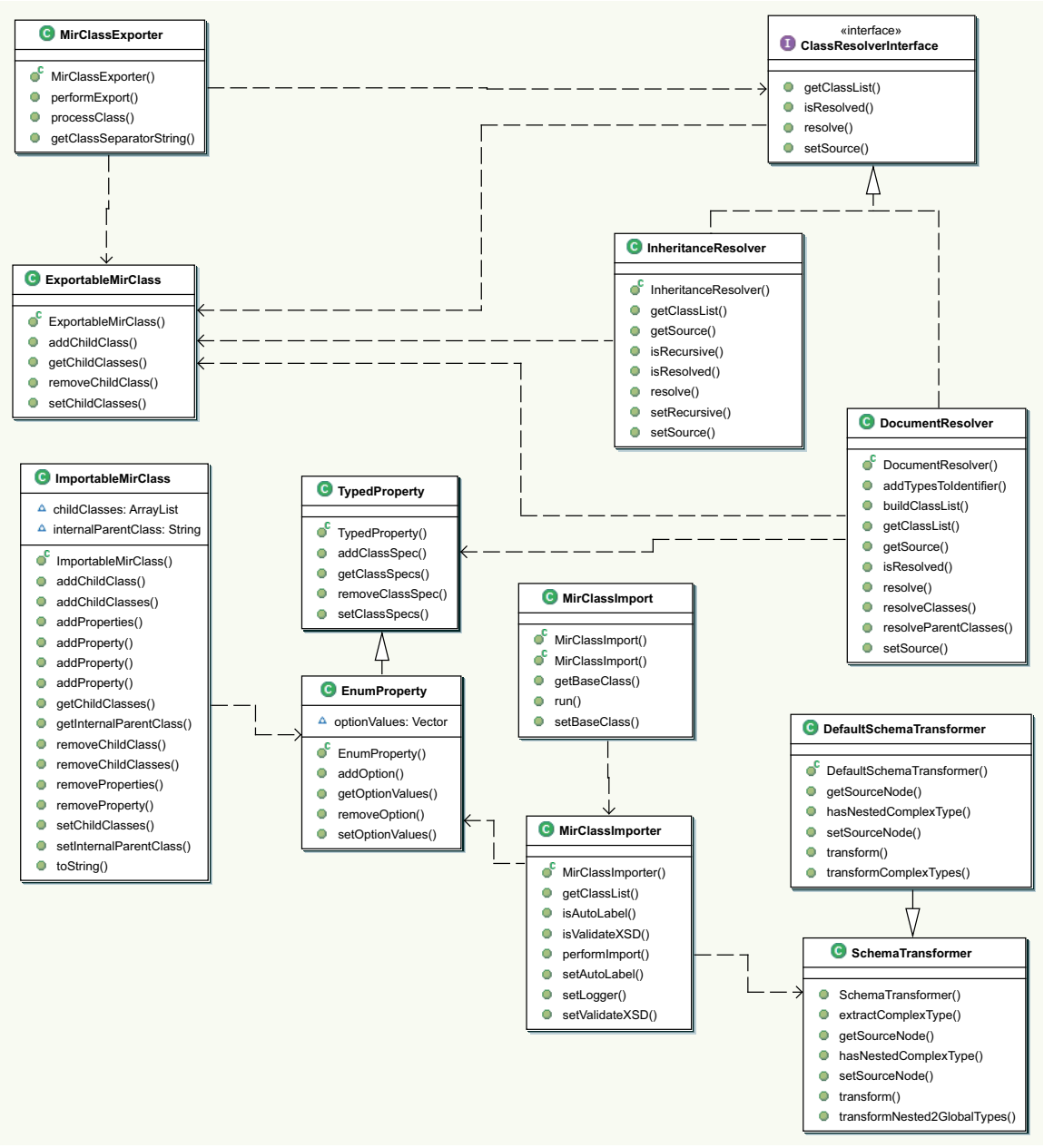

Abbildung 6.2: Klassenmodell der Strukturschnittstelle

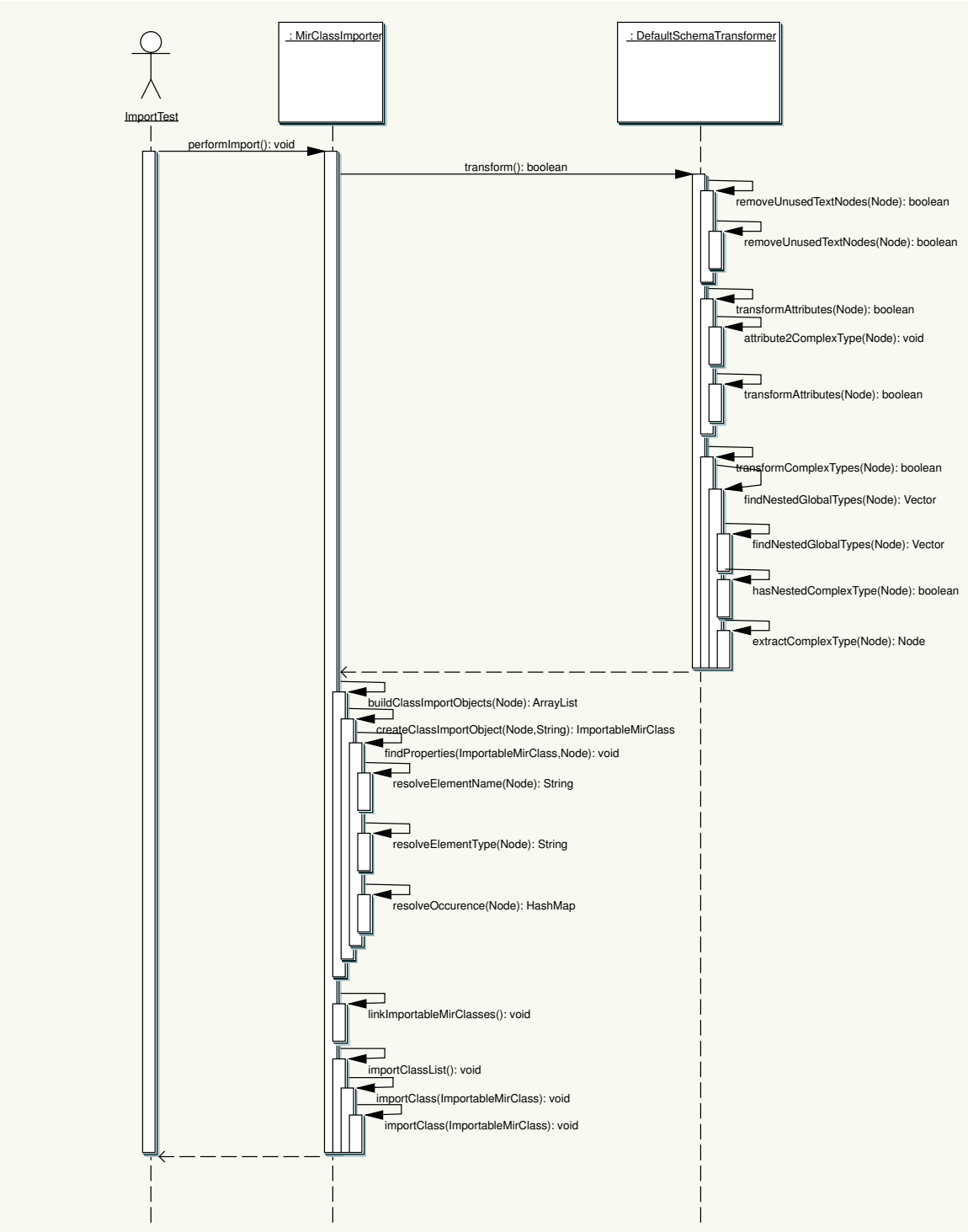

Abbildung 6.3: Funktionsweise des Strukturtransformators

7 Test der Schnittstelle

Ein Test der Implementierung ist für einen sicheren Einsatz im laufenden Betrieb unumgänglich. Er weist die gewünschte Funktionalität der Implementierung nach und zeigt eventuell entstandene Defizite auf.

Um diese Funktionsüberprüfung durchführen zu können, werden typische und besondere Beispielfälle für den Export und den Import entwickelt. da für den Import und Export zum Teil unterschiedliche Zielstellungen verfolgt und demnach andere Erwartungen gestellt werden, wird der Test für jede Transformationsrichtung getrennt durchgeführt.

Der Export bildet die darzustellenden Klassen in ihrer Vererbungsstruktur ab, und baut ggf. durch zusätzliche Strukturinformationen eine Dokumenthierarchie zur Bildung gültiger XML-Dokumente auf. Hier wird insbesondere auf die vollständige Darstellung der Klassen und deren Vererbungsstruktur geachtet.

Beim Import sind zwei Varianten wichtige Testfälle: Zum einen müssen bereits exportierte Informationen wieder eingelesen werden können. Zum anderen werden auch externe XML Schema Definitionen importiert, die unter Umständen andere innere Strukturen aufweisen. Deshalb werden dafür auch getrennte Beispiele entwickelt, um den Test überschaubar darzustellen.

Im folgenden sollen die, aus den einzelnen Varianten entstehenden, Erwartungen entwickelt werden und überprüft werden.

Hinweis:

Bei der Darstellung von XML Schema Definitionen ist zur Erhöhung der Übersichtlichkeit auf die Angabe von Kopf und Fuß des XML-Schemas sowie des freien Bezeichnungstextes (Label) jeder Property verzichtet worden.[1]

7.1 Export von Klassen- und Anwendungsstrukturen

Für den Export wurde mit dem Klassenmanager (Abbildung 7.1) die im Beispiel 2.3 abgebildete Vererbungsstruktur erzeugt. Von den Basisklassen `GenericPage` und

[1] Ein vollständiges Listing befindet sich im Angang

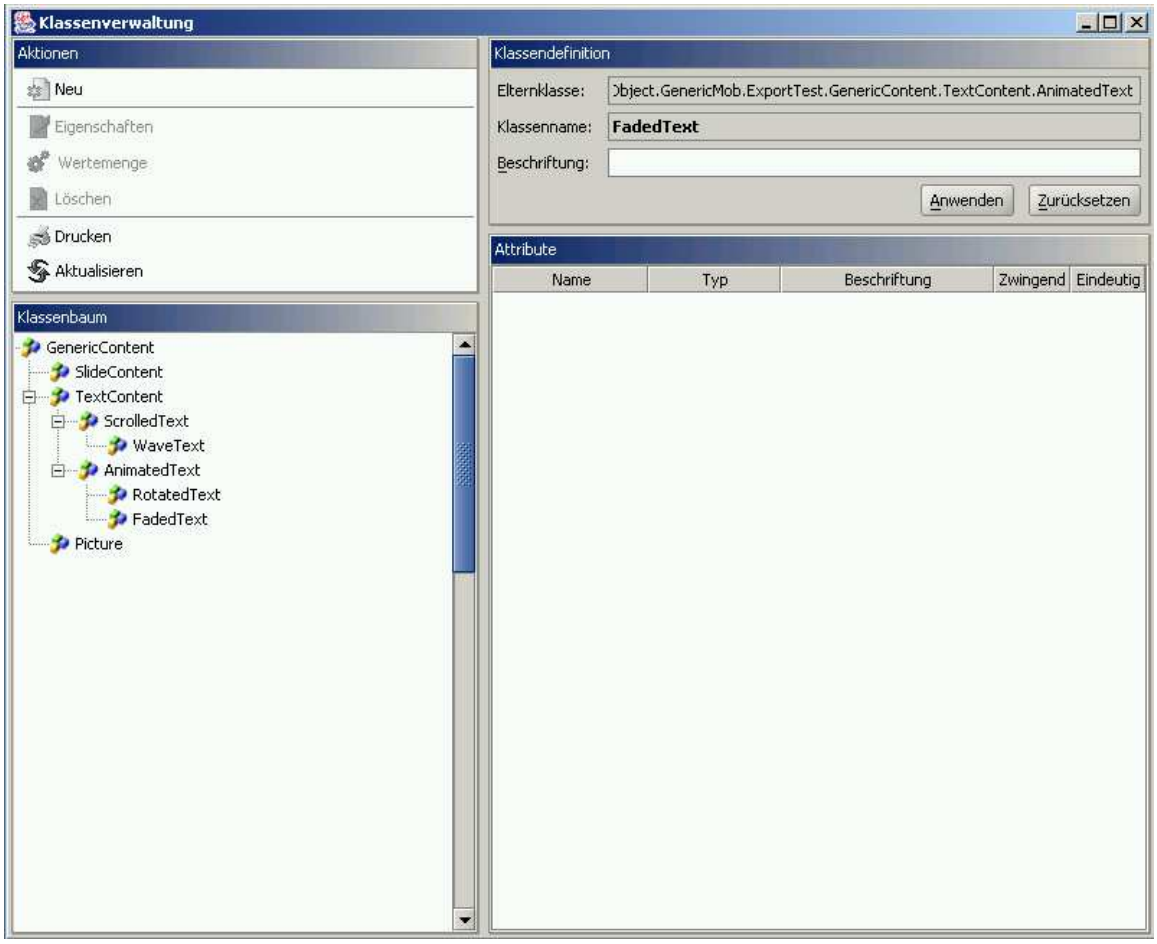

Abbildung 7.1: Testbeispiel für den Export im Klassenmanager

`GenericContent` ausgehend, wurden die dargestellten Klassenstrukturen kreiert. Auf die gebildeten Klassen verteilt, sind alle im MIR-System vorhandenen Datentypen enthalten. Neben Verweisen sind Klassen mit gleichen Namen modelliert, um die Namensbildung beim Normalisierung zu testen.

Für den Export einer Vererbungshierarchie wird, ausgehend von der `GenericContent`-Basisklasse, ein kompletter Klassenbaum in XML-Schema dargestellt. Dafür wird der `MirClassExporter` mit einer Instanz der `InheritanceResolver`-Klasse initialisiert. Abbildung 7.1 zeigt das Ergebnis des Exportes.

```
...
<xs:complexType name=\"GenericPage\"\/>
<xs:complexType name="AnimatedPage">
<xs:complexContent>
```

```
  <xs:extension base="GenericPage">
    <xs:sequence>
      <xs:element name="animatedText" type= "xs:anyType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="scrolledText" type= "xs:anyType" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="AnimatedPage-Print">
<xs:complexContent>
  <xs:extension base="AnimatedPage">
    <xs:sequence>
      <xs:element name="date" type="xs:dateTime" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="TitledPage">
<xs:complexContent>
  <xs:extension base="GenericPage">
    <xs:sequence>
      <xs:element name="title" type="String" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="WebPage">
<xs:complexContent>
  <xs:extension base="TitledPage">
    <xs:sequence>
      <xs:element name="meta-key" type="String" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="content" type= "xs:anyType" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="Slide">
<xs:complexContent>
  <xs:extension base="GenericPage">
    <xs:sequence>
      <xs:element name="slides" type= "xs:anyType" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
```

```
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="Slide-Print">
<xs:complexContent>
  <xs:extension base="Slide">
    <xs:sequence>
      <xs:element name="printable" type="xs:boolean" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
...
```

Tabelle 7.1: Exportierte Vererbungsstruktur

Die Abbildung der Klassenstrukturen ist vollständig, weil alle globalen *complexType*'s zu finden sind, die die einzelnen Klassen abbilden. Da alle Verweisattribute als `"xs:anyType"` dargestellt sind, fehlen jedoch die Dokumentstruktur-Informationen. Damit ist diese Variante fast ausschließlich für den Transport innerhalb von MIR-Datenbankinstanzen, also zur Archivierung und Wiederherstellung, geeignet.

Zur Überprüfung des Exportes einer Dokumentenstruktur werden die in Tabelle 4.1 deklarierten Pfadausdrücke (Dokument1) verwendet. Sie selektieren einen Teil der Klassenstrukturen unterhalb der beiden Basisklassen und stellen die Informationen für typisierte Verweise zur Verfügung.

Für die Auflösung der Dokumentstruktur-Informationen und zur Ermittlung der zu exportierenden Klassen wird jetzt die `DocumentResolver`-Klasse verwendet. Der folgende XSD-Auszug zeigt das neue Ergebnis des Exportes.

```
[...]
<xs:complexType name="GenericContent"/>
<xs:complexType name="TextContent">
<xs:complexContent>
  <xs:extension base="GenericContent">
    <xs:sequence>
      <xs:element name="text" type= "xs:anyType" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
```

```
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="ScrolledText">
<xs:complexContent>
  <xs:extension base="TextContent">
    <xs:sequence>
      <xs:element name="direction" minOccurs="1" maxOccurs="1">
        <xs:annotation>
          <xs:documentation>Scrollrichtung</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="horizontal"/>
            <xs:enumeration value="vertical"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="speed" type="xs:integer" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="AnimatedText">
<xs:complexContent>
  <xs:extension base="TextContent">
    <xs:sequence>
      <xs:element name="speed" type="xs:integer" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="Picture">
<xs:complexContent>
  <xs:extension base="GenericContent">
    <xs:sequence>\scriptsize
      <xs:element name="pic" type= "xs:anyType" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="GenericPage"/>
```

```
<xs:complexType name="AnimatedPage">
<xs:complexContent>
 <xs:extension base="GenericPage">
  <xs:sequence>
   <xs:element name="animatedText" minOccurs="1" maxOccurs="unbounded" type="AnimatedText"/>
   <xs:element name="scrolledText" minOccurs="1" maxOccurs="1" type="ScrolledText"/>
  </xs:sequence>
 </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="TitledPage">
<xs:complexContent>
  <xs:extension base="GenericPage">
    <xs:sequence>
      <xs:element name="title" type="String" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="WebPage">
<xs:complexContent>
  <xs:extension base="TitledPage">
    <xs:sequence>
      <xs:element name="meta-key" type="String" minOccurs="1" maxOccurs="unbounded"/>
      <xs:choice  minOccurs="1" maxOccurs="unbounded">
        <xs:element name="content" type="AnimatedPage"/>
        <xs:element name="content" type="TextContent"/>
        <xs:element name="content" type="Picture"/>
      </xs:choice>
    </xs:sequence>
  </xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:element name="WebPage" type="WebPage"/>
...
```

Tabelle 7.2: Exportierte Dokumentstruktur

Alle Klassen der Dokumentstruktur sind wieder auffindbar, und die Verweise liegen in typisierter Form vor. Auch das Wurzelelement `<xs:element name="WebPage" .../>` ist, für die Validierung und Erzeugung von XML-Dokumenten, definiert. Des weiteren fällt auf, dass auch nicht zur Dokumentstruktur gehörende Klassen exportiert worden sind. So ist die Vererbungshierarchie, über die Verknüpfung durch das `<xs:extension>`-Element, erhalten geblieben.

Die aufgebaute Struktur kann durch XMLSpy[2] übersichtlich dargestellt werden. Hier ist noch einmal die Verbindung von Vererbungs- und Dokumentenstruktur in XSD-Beschreibungen nachvollziehbar.

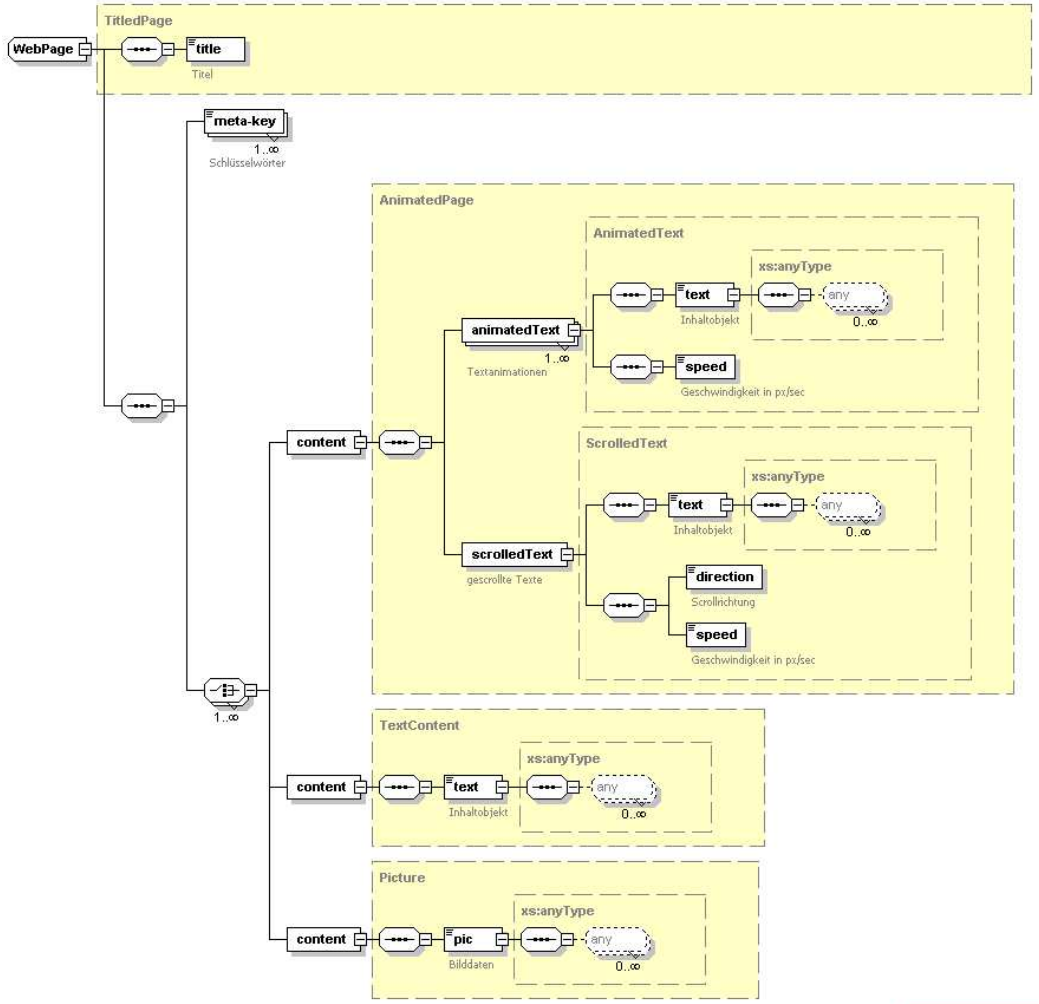

Abbildung 7.2: Grafische Darstellung des exportierten Dokumentes

[2]Grafisches Werkzeug für XML und andere textbasierte Websprachen: `http://www.xmlspy.com/`.

7.2 Import von internen und externen XML-Schema

Beim Import werden wieder zwei charakteristische Varianten getrennt getestet. Zum einen wird eine exportierte Dokumentenstruktur wieder hergestellt werden. Dazu wird das in Abschnitt 7.1 erstellte Schema der Dokumentstruktur wieder eingelesen. Für diesen Vorgang wird zunächst der Klassenbaum gelöscht, um den erfolgten Import der Klassenstrukturen nachvollziehen zu können. Die Abbildung 7.3 zeigt die entstandene Klassenstruktur.

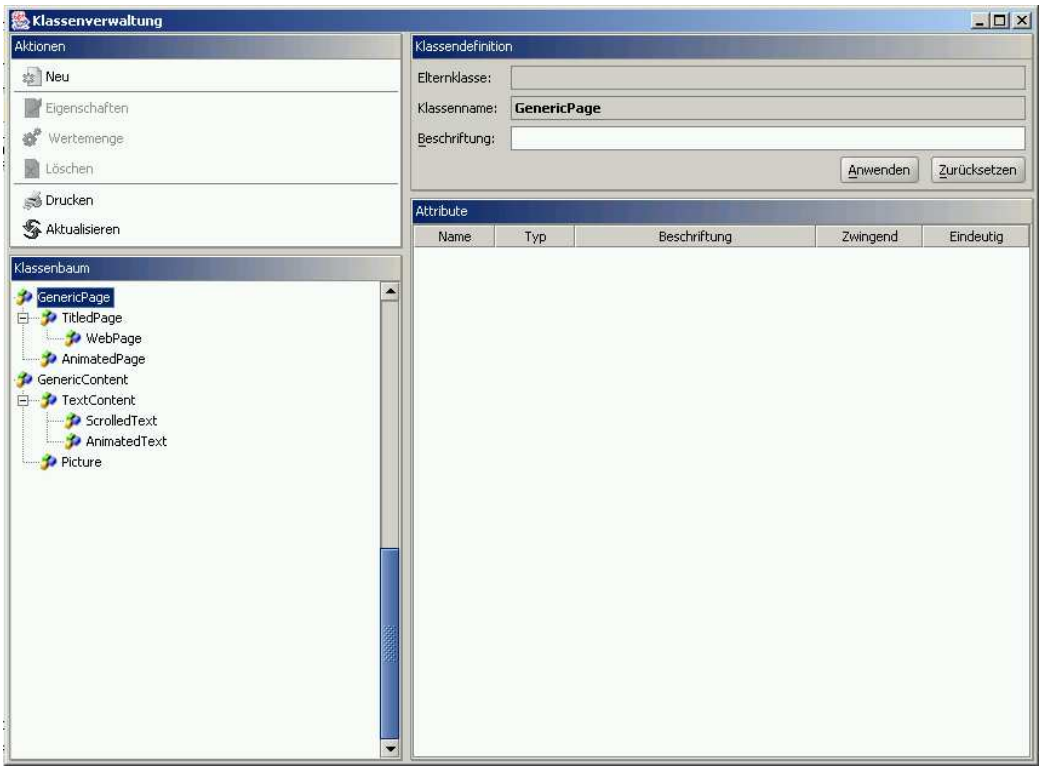

Abbildung 7.3: Klassenstruktur nach internen Schema-Import

Es ist zu erkennen, dass die für die Dokumentenstruktur benötigten Klassen wieder vollständig hergestellt werden konnten. Die gebildeten Pfade sowie die weiteren Informationen (Wurzelelement und Namensraum) aus dem Header des XML-Schema wurden in der `ApplicationDocument`-Klasse abgelegt.

Die Vielfalt von XSD wurden schon in Kapitel 5.7 dargestellt und wird hier durch ein externes Schema mit komplexen eingebetteten Strukturen und Attributen repräsentiert. Die Tabelle 7.3 zeigt das Testbeispiel.

```
...

<xs:element name="CUSTOMER">
  <xs:complexType>
    <xs:attribute name="ID" type="xs:int"/>
    <xs:sequence>
      <xs:element name="FIRSTNAME" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="LASTNAME" type="xs:string"/>
      <xs:element name="ADDRESS">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="LINE1" type="xs:string"/>
            <xs:element name="LINE2" type="xs:string"/>
            <xs:element name="CITY"/>
            <xs:element name="ZIP" type="xs:int"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="BIRTHDATE" type="xs:date"/>
      <xs:element name="ORDER" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="PRODUCT">
             <xs:attribute="ID" type="xs:int"/>
             <xs:attribute="NAME" type="xs:string"/>
            </xs:element>
            <xs:element name="PRIZE" type="xs:double"/>
            <xs:element name="QUANTITY" type="xs:int"/>
            <xs:element name="DATE" type="xs:date"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

...
```

Tabelle 7.3: Externes XML Schema für Import

Abbildung 7.4 zeigt den erzeugten Klassenbaum. Es ist die Benennung der *complexTypes* und die sich daraus ergebenen neuen Klassenbezeichner erkennbar. Dies wird zum Beispiel am unbenannten *complexType* unter dem `ADDRESS`-Element, der zur `ADDRESSType`-Klasse importiert wurde, deutlich. Die Attribute und Elemente sind in Klassenattribute umgewandelt worden, wobei sich neue Klassentypen ergeben haben. Zum Beispiel wird das `PRODUCT`-Element als separate `PRODUCTType`-Klasse importiert.

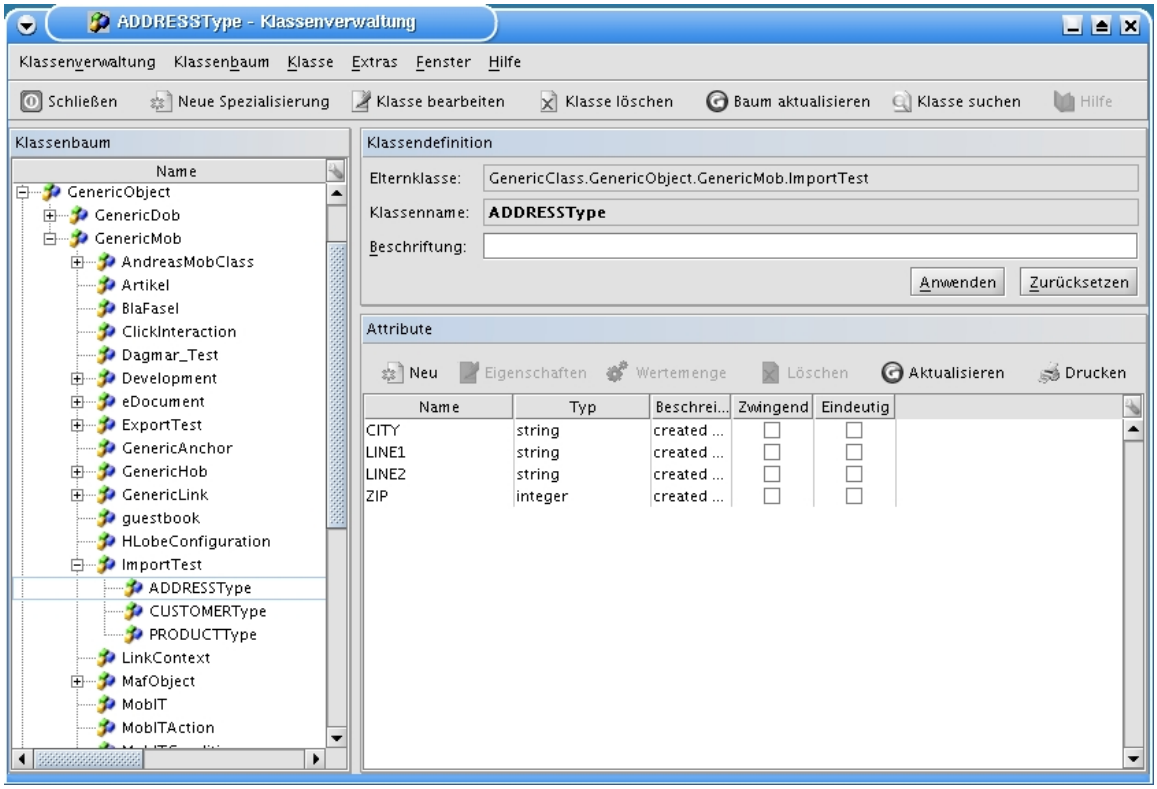

Abbildung 7.4: Klassenstruktur nach externem Schema-Import

Eine erzeugte `ApplicationDocument`-Klasse, die in Abbildung 7.4 mit dargestellt ist, bildet wieder die gewonnen Dokument-Informationen ab.

8 Ausblick

Die objektorientierten Datenmodelle werden sich in der datenbankgestützten Softwareentwicklung durchsetzen. Eine Weiterentwicklung der objektorientierten Datenbanken in Sachen Schnelligkeit, Robustheit und Akzeptanz wird zu einem gesteigerten Bedürfnis hinsichtlich des Austausches von komplexen Informationsstrukturen führen.

Die implementierte Schnittstelle dient dem Aufbau von strukturierten Informationen aus bzw. in objektorientierte Klassen. Durch XML Schema Definitionen wird der Austausch von hierarchisch strukturierten, dokumentorientierten Informationsstrukturen im XML-Format unterstützt, welches im MIR-System bereits Verwendung findet.

Die Erweiterung des MIR-Systems um die Möglichkeiten der internen Daten- und Anwendungsstrukturierung kann die applikationsseitige Nutzung multimedialer Inhalte verbessern.

Literaturverzeichnis

[Cod70] Edgar F. (Ted) Codd. A relational model of data for large shared data banks. In *Communications of ACM*, volume 13, pages 377–387, 1970.

[Gli03] Martin Glinz. 4 - Einführung in UML. In *Spezifikation von Software*. Institut für Informatik der Univewrsität Zürich, 2001, 2003. http://www.ifi.unizh.ch/req/ftp/ses/kapitel_04.pdf letzter Besuch: 02/03/2004.

[Heu97] Andreas Heuer. *Objektorientierte Datenbanken - Konzepte, Modelle, Standards und Systeme*. Addison Wesley, Bonn, 1997.

[Kár02] Andreas Kárpáti. *MIR Datenbank-Konzeption*. FHTW Berlin, 2002. `http://www.rz.fhtw-berlin.de/mirdoc/mir-db/intro.xml`.

[Moo93] K. Moore. *MIME (multipurpose internet mail extensions) part two: Message header extensions for non-ascii text*, 1993. `ftp://ftp.internic.net/rfc/rfc1522.txt`.

[Rus02] Craig Russell. *Java Database Objects („Specification")*. Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A., 2002.

[Sch03] T.C. Schmidt. *MIR – Das Media Information Repository. White Paper*. FHTW – Berlin, 2003. `http://www.rz.fhtw-berlin.de/MIR/papers/mir-wp.pdf`, letzter Besuch : 02/04/2004.

[TCS03] Michael Engelhardt T. C. Schmidt. Semantic linking - a context-based approach to interactivity in hypermedia. In *Berliner XML Tage 2003*. Humboldt Universität Berlin, Eckstein, Tolksdorf, 2003.

[TK03] Thomas Kudrass Tobias Krumbein. Rule-based generation of xml schemas from uml class diagrams. In *Berliner XML Tage 2003*. Humboldt Universität Berlin, Eckstein, Tolksdorf, 2003.

[W3C98] W3C Konsortium des World Wide Web. *Extensibe Markup Language*, 1998. `http://www.w3c.org/TR/REC-xml`.

[W3C01a] W3C Konsortium des World Wide Web. *XSD Schema - Primer*, 2001.
`http://www.w3c.org/TR/xmlschema-0`.

[W3C01b] W3C Konsortium des World Wide Web. *XSD Schema Part 1: Structures*, 2001.
`http://www.w3c.org/TR/xmlschema-1`.

[W3C01c] W3C Konsortium des World Wide Web. *XSD Schema Part 3: Datatypes*, 2001.
`http://www.w3c.org/TR/xmlschema-2`.

[W3C03a] W3C Konsortium des World Wide Web. *HTML*, 2003.
`http://www.w3c.org/??`

[W3C03b] W3C Konsortium des World Wide Web. *RDF- Concepts and Abstract Syntax*, 2003.
`http://www.w3c.org/TR/rdf-concepts`.

[W3C03c] W3C Konsortium des World Wide Web. *RDF- Model and Syntax Specification*, 2003.
`http://www.w3c.org/TR/REWC-rdf-syntax`.

[W3C03d] W3C Konsortium des World Wide Web. *RDF Primer*, 2003.
`http://www.w3c.org/TR/rdf-primer`.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter der Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Datum Arne Hildebrand