Johannes Kepler Universität Linz

Institut für Bioinformatik
**Abteilung für Informationssysteme**

---

# Entwurf eines Metaschemas zur Integration von relationalen Datenbanken und XML Schema

**Diplomarbeit zur Erlangung des akademischen Grades:**
**Magister rerum socialium oeconomicarumque (Mag. rer. soc. oec.)**

in der Studienrichtung Wirtschaftsinformatik
eingereicht bei a. Univ.-Prof. Mag. Dr. Werner Retschitzegger

**Thomas Kraushofer**
**9957384 / 175**

Linz, 24. März 2005

## Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, 24.03.2005

# Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während meines Studiums begleitet und mir geholfen haben, mein Studium letztendlich erfolgreich abzuschließen.

Zu Beginn möchte ich mich recht herzlich bei Elisabeth Kapsammer und Werner Retschitzegger bedanken, die mich tatkräftig unterstützt und mir stets über die eine oder andere Hürde geholfen haben.

Vielen Dank auch an meine Studienkollegen und meine Freunde.

Ganz besonders bedanken möchte ich mich bei meiner Familie – meiner Mutter Gudrun Kraushofer und meiner Schwester Sabine, die immer für mich da waren und mir es dadurch ermöglichten, diesen wichtigen Abschnitt meines Lebens erfolgreich zu meistern.

# Kurzfassung

Eine Vielzahl von heute gängigen Applikationen verwenden relationale Datenbanksysteme (RDBS), um Daten persistent speichern, verarbeiten und verwalten zu können. Immer häufiger kommt beim Datenaustausch, sowohl zwischen Applikationen als auch zwischen Benutzern und Applikationen, XML (eXtensible Markup Language) zum Einsatz. XML ist eine erweiterbare Auszeichnungssprache, die unter anderem ein Quasi-Standard für den elektronischen Datenaustausch ist. Mit dem zunehmenden Volumen an ausgetauschten Daten in Form von XML wird der Wunsch nach einer an die veränderten Bedürfnisse angepassten Speichermöglichkeit immer deutlicher. Um die Vorteile beider Technologien nutzen zu können, ist eine Integration von XML und RDBS nötig. Eine Möglichkeit diese beiden Technologien zu integrieren besteht darin, Schemata von XML-Dokumenten auf Schemata von relationalen Datenbanken (RDB) abzubilden. Auf dieser Möglichkeit basiert auch X-Ray, ein generischer Ansatz zur Integration von XML und relationalen Datenbanken auf Basis eines Metaschemas, das selbst in einem RDBS gespeichert ist.

Ziel dieser Diplomarbeit war es, ein Metaschema zu entwickeln, mit welchem relationale Schemata auf Schemata abgebildet werden können, die mit der Sprache XML Schema definiert werden.

Auf Basis dieses Metaschemas, welches das notwendige Abbildungswissen enthält, können Daten aus XML-Dokumenten in ein oder mehrere RDBS eingefügt bzw. Daten aus RDBS in XML-Dokmente ausgelesen werden.

## Abstract

Nowadays many applications use relational database systems (RDBS) to process, manage and persistently store data. At the same time, XML (eXtensible Markup Language) is used more and more often for datatransfer between applications as well as for datatransfer between users and applications. XML is an extensible markup language that is rapidly emerging as the de-facto standard for digital datatransfer. In order to benefit from both technologies, an integration of XML and RDBS is necessary. One approach to achieve an integration is to map the schemata of XML documents to schemata of relational databases. This approach is the base of X-Ray, a generic approach to map XML to relational databases based on a metaschema, stored in a RDBS.

The main objective of this diploma thesis was to develop a metschema, so that relational schemata can be mapped to schemata, defined in *XML Schema*.

Based on this metaschema, which contains the mapping knowledge, data can be imported from XML documents into one or more RDBS as well as exported to XML documents from one or more RDBS.

# Inhaltsverzeichnis

# 1. Einleitung

## *1.1. Motivation und Zielsetzung*

Ein Großteil der Daten weltweit wird heutzutage in relationalen Datenbanksystemen (RDBS) gespeichert. Daher greifen auch viele Applikationen auf RDBS als „Datenquellen“ bzw. „Datensenken“ zu. In der Industrie hat sich SGML (Standard Generalized Markup Language; ISO 8879:1985) [SGML05], seit nahezu 20 Jahren bewährt, um Datenstrukturen verschiedener elektronischer Dokumente zu beschreiben. XML (eXtensible Markup Language [W3C04A]), eine abgespeckte Version von SGML, hat sich in den letzten Jahren sehr stark verbreitet und wird vor allem als Format zum Austausch von Daten zwischen Applikationen verwendet.

Es liegt also nahe, diese beiden Technologien zu integrieren, um XML-Dokumente in RDBS zu speichern bzw. XML-Dokumente aus RDBS zu generieren.
Es gibt zwar einige kommerzielle Anbieter von XML-Datenbanken und RDBS, jedoch existieren derzeit kaum Applikationen, die es, unter Gewährleistung der Autonomien der Schemata auf Seite von XML sowie auf Seite des RDBS, ermöglichen, XML-Daten in ein oder mehrere RDBS unter Verwendung eines von Benutzern erstellten und erweiterbaren Abbildungswissens zu speichern.

Genau dies war jedoch das Ziel eines Projektes an der Johannes-Kepler-Universität Linz, in Rahmen dessen das X-Ray-Konzept entwickelt wurde [Kapp04]. X-Ray ist ein generischer Ansatz zur Integration von XML und relationalen Datenbanken auf Basis eines Metaschemas, in welchem das Abbildungswissen gespeichert wird. Das Metaschema selbst wird ebenfalls in einem RDBS gespeichert. Bei X-Ray ist es jedoch nur möglich, XML-Dokumente, die mittels DTDs (Data Type Definition [W3C04A]) beschrieben werden, auf ein RDBS abzubilden. Ziel dieser Diplomarbeit war es, X-Ray dahingehend zu erweitern, dass auch XML-Dokumente, die mit XML

Schema [W3C04B] beschrieben werden, abgebildet werden können. Der Grund dafür ist, dass XML Schema, im Gegensatz zu DTDs, immer häufiger verwendet wird, um Strukturen von XML-Dokumenten zu beschreiben. Diese erweiterte Variante von X-Ray trägt bezeichnenderweise den Namen X-Rayxs.
Der Terminus *XML Schema* wird im Zuge dieser Arbeit für die gleichnamige Schemasprache des W3C [W3C04B] verwendet. *XML-Schema* hingegen, bezeichnet das Schema eines XML-Dokumentes, egal ob auf DTDs oder XML Schema basierend.
Schwerpunkt dieser Arbeit war der Entwurf des Metaschemas von X-Rayxs, welches den Kern des Ansatzes bildet. Dabei wird das Metaschema von X-Rayxs anhand von Beispielen näher beschrieben.
Um die Funktionalität des Metaschemas von X-Rayxs nachzuweisen, wurde im Zuge von [ORT05] ein Prototyp entwickelt, mit welchem, basierend auf dem Metaschema, XML-Dokumente in RDBS gespeichert bzw. XML-Dokumente aus RDBS generiert werden können.

## *1.2. Aufbau der Diplomarbeit*

Im zweiten Kapitel der Diplomarbeit werden die beiden am meisten verbreiteten Möglichkeiten, die Struktur eines XML-Dokumentes zu beschreiben, gegenüber gestellt. Dabei wird zuerst DTD und im Anschluss XML Schema vorgestellt. Am Schluss des Kapitels werden die Vorteile von XML Schema gegenüber DTDs dezidiert herausgehoben.

Im anschließenden Kapitel wird X-Ray, das die konzeptionelle Basis für X-Rayxs bildet, vorgestellt. Nach den Designzielen und den beiden Arbeitsphasen von X-Ray wird die Architektur dieses Konzeptes beleuchtet. Zum besseren Verständnis werden die Abbildungsstrategien erläutert, die beschreiben, wie Elemente von XML-Dokumenten, abhängig vom Kontext, in dem sie verwendet werden, auf ein RDBS abgebildet werden können. Das Ende des Kapitels widmet sich dem Metaschema, in welchem das Abbildungswissen gespeichert wird.

Das vierte Kapitel stellt den Kern der Diplomarbeit dar, da in diesem Kapitel X-Rayxs präsentiert wird. Es handelt sich bei X-Rayxs um eine Adaption von X-Ray, um XML

Schema in das Konzept zu integrieren. Es werden deshalb im ersten Abschnitt des Kapitels die Konzepte von XML Schema vorgestellt. Dieser Abschnitt wird in die von X-Rayxs unterstützten, sowie die explizit nicht unterstützten Konzepte von XML Schema unterteilt. Nach der eingehenden Beschreibung des Metaschemas, das den Kern von X-Rayxs bildet, wird dessen Arbeitsweise anhand kleinerer Beispiele demonstriert.

Im letzten Kapitel wird eine Übersicht über den zur Validierung des Metaschemas entwickelten Prototyp gegeben. Die vollständige Beschreibung des Prototyps lässt sich in [ORT05] nachlesen.

# 2. Vergleich zwischen DTD und XML Schema

Die eXtensible Markup Language (XML) wurde 1998 in erster Version vom World Wide Web Consortium (W3C) als Empfehlung verabschiedet [W3C05]. XML ist eine Metasprache, die es ermöglicht, anhand von Auszeichnungen, so genannten Tags, neue Markup-Sprachen zu definieren. Auf diese Weise definierte Sprachen werden XML-Anwendungen genannt [SCHN04].

## *2.1. Schemasprachen*

„Der Zweck eines Schemas ist es, eine Klasse von XML-Dokumenten zu definieren. Deshalb wird häufig der Begriff “Instanzdokument” oder kurz “Instanz” verwendet, um ein Dokument zu beschreiben, das einem bestimmten Schema entspricht.“ [W3C04C]

Die ursprüngliche XML 1.0 Empfehlung definierte nur eine Art der Beschreibung von Inhaltsmodellen eines XML Instanzdokumentes: die Document Type Definition (DTD) [W3C04A]. Eine DTD wird verwendet, um in einem XML-Dokument zulässige Elemente und Attribute zu beschreiben und ihre Beziehungen untereinander auszudrücken. DTDs waren somit die erste Möglichkeit, um XML-Anwendungen zu beschreiben.

Bald merkte man allerdings, dass man zum Beschreiben von XML-Anwendungen ein flexibleres Modell benötigte, das u. a. Datentypen berücksichtigt. 1999 begann das [W3C05] an der neuen XML Schema Definitions-Sprache (XSD) zu arbeiten, die heute als endgültige Empfehlung vom 2. Mai 2001 (2. Auflage 28. Okt. 2004) erhältlich ist [W3C04B]. XML Schema baut auf den Erfahrungen mit DTDs und älteren Sprachen auf, und ist die derzeit flexibelste und leistungsfähigste Möglichkeit, um XML-Anwendungen zu beschreiben.

Im Folgenden werden zunächst sowohl DTDs als auch XML Schema kurz beschreiben. Anschließend daran werden die wichtigsten Vorteile von XML Schema im Vergleich zu DTDs diskutiert.

### *2.1.1. Dokument Typ Definition (DTD)*

DTDs dienen zur Definition der erlaubten Elemente, Attribute und ihrer Zusammensetzung für Instanzen, den XML-Dokumenten. Die DTD legt damit die in der Instanz zu verwendenden Tags fest und bestimmt, in welcher Reihenfolge sie auftreten und wie sie verschachtelt werden dürfen. Außerdem werden die bei einem Element erlaubten Attribute und die Art der Werte festgelegt [GALI04].

Liegt eine DTD vor, kann ein konkretes XML-Dokument an Hand der zugeordneten DTD auf Gültigkeit geprüft werden. Die Verwendung von DTDs ist aber nicht vorgeschrieben. Fehlt eine DTD, kann ein XML-Dokument nur auf Wohlgeformtheit geprüft werden. Im Folgenden werden die Stärken und Schwächen von DTDs aufgelistet und es wird ein Beispiel für eine DTD zur Verwaltung von Unterkünften gezeigt (vgl. Abbildung 1).

- *Stärken von DTDs:*
  - Kompakte Syntax
  - Schnelle Verarbeitung möglich
  - Handliche Größe
- *Schwächen von DTDs:*
  - DTDs verwenden keine XML-Syntax
  - DTDs unterstützen keine Namensräume
  - Datentypen sind begrenzt
  - Keine Vererbung möglich
  - Eingeschränktes Identifizierungs- und Referenzierungskonzept

```
      …
<!ELEMENT accommodations (accommodation*)>
<!ELEMENT accommodation (name, address, email*, phone+,
acceptsCreditCard?, facilities, sauna, pool*, description?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (street, village, country)>
 …

<!ATTLIST accommodation id      CDATA      #REQUIRED
state                CDATA      #FIXED "Austria"
kind                 (hotel | motel) "hotel">
…
```

**Abbildung 1 - Beispiel für DTD**

### *2.1.2. XML Schema (XSD)*

Ein XML Schema beschreibt selbstdefinierte Datentypen und die Struktur von XML Dokumenten. Als Basis-Datentypen stehen die in "XML Schema Part 2: Datatypes Second Edition" [W3C04D] definierten Typen zur Verfügung. Hierzu zählen z. B. Integer (Zahlen ohne Nachkommastellen), Float (Zahlen mit Nachkommastellen) und Strings (Zeichenketten mit fester oder variabler Länge) etc. (siehe Abbildung 21). XML Schema erlaubt darüber hinaus auch die Definition eigener Datentypen. In der Strukturdefinition wird festgelegt in welcher Art und Weise (Reihenfolge, Anzahl, etc.) die Elemente und Attribute in den Instanzen auftreten dürfen. Somit ist es möglich z. B. einen Datentyp „Hotel" zu definieren, der wiederum aus mehreren Zimmern besteht, die selbst wieder durch einen eigenen Datentyp definiert werden, und jeweils eine bestimmte Anzahl an Betten usw. aufweisen. Im Gegensatz zu DTDs entsprechen die Möglichkeiten von XML Schema den heutigen Anforderungen von Datenbanken und Programmiersprachen. Die Bestandteile des Schemas werden wiederum in XML-Syntax beschrieben. [GALI04]

Liegt ein Schema vor, kann ein konkretes XML-Dokument mit Hilfe des zugeordneten XML Schemas auf Gültigkeit geprüft werden. Die Verwendung von XML Schemata ist aber, genauso wie die Verwendung von DTDs, nicht vorgeschrieben. Fehlt die Zuordnung zu einem XML Schema, kann ein XML-Dokument nur auf Wohlgeformtheit geprüft werden.

Im Anschluss werden die Stärken und Schwächen von XML Schema aufgelistet und es wird ein Beispiel für ein XML Schema zur Verwaltung von Unterkünften gezeigt (vgl. Abbildung 2).

- *Stärken von XML Schema*
  - Schemadefinitionen sind XML-Dokumente
  - Unterstützung von Namensräumen
  - Viele vordefinierte Datentypen (Datum, Währung, Integer, etc.)
  - Definition eigener Datentypen
  - Vererbung von Typdefinitionen
  - Flexibles Identifizierungs- und Referenzierungskonzept (vgl. Abschnitt **Identifizierung und Referenzierung**)

- *Schwächen von XML Schema*
  - Syntax von XML Schema sehr umfangreich
  - XML Schema Dokumente sehr umfangreich

```
<schema targetNamespace="AccommodationSchema"
elementFormDefault="qualified" xmlns:ac="AccommodationSchema"
xmlns:sp="xray_supervision.xsd"
xmlns="http://www.w3.org/2001/XMLSchema">
...
<complexType name="accommodationType">
   <sequence>
      <element ref="ac:name"/>
      <element name="address" type="ac:addressType"/>
      <element name="manager" type="ac:managerType"/>
      <element name="supervisor" type="su:supervisionType"/>
      <element name="test" type="ac:mangerType"/>
      <element name="email" type="string" minOccurs="0"
           maxOccurs="unbounded"/>
      <element name="phone" type="ac:phoneType"
           maxOccurs="unbounded"/>
      <element name="acceptsCreditCards" minOccurs="0">
           <complexType/>
      </element>
      <element name="facilities">
           <complexType/>
      </element>
      <element name="sauna" type="ac:saunaType"/>
      <element name="pool" minOccurs="0" maxOccurs="unbounded">
           <complexType/>
      </element>
      <element name="description" type="ac:descriptionType"
           minOccurs="0"/>
      <element name="room" type="ac:roomType"
           maxOccurs="unbounded"/>
   </sequence>
   <attribute name="id" type="positiveInteger" use="required"/>
   <attribute name="state" type="string" fixed="Austria"/>
</complexType>
...
```

**Abbildung 2 - Beispiel für XML Schema (XSD)**

## *2.2. Die wesentlichen Vorteile von XML Schema*

***Unterstützung von Namensräumen***

Namensräume dienen zur eindeutigen Identifizierung von Typdefinitionen und Deklarationen von Elementen in einem Schema. Jeder Namensraum ist durch einen URI eindeutig beschrieben. Alle Namen in einem Schema gehören zu einem bestimmten Namensraum. Sie erlauben uns also, zwischen verschieden Vokabularen zu unterscheiden [W3C04C]. Dies geschieht mit Hilfe von Definitionen

verschiedener Namensräume und deren Zuweisung zu jeweils einem Präfix im <*schema*>-Element eines XML Schemas (siehe Abbildung 2).
Beispielsweise kann nun zwischen einem <*Titel*>-Element, das zum Typ „Buch“ gehört, und einem gleichnamigen Element, das allerdings dem Typ „Person“ zugehörig ist, unterschieden werden.

***Definition eigener Datentypen***

XML Schema verfügt bereits über eine Vielzahl von vordefinierten Typen wie z.B. String, Integer, Float, Boolean, etc. Auf diesen Basistypen aufbauend können weitere, eigene Typen definiert werden, wobei zwischen einfachen und komplexen Typen unterschieden wird.
Ein einfacher Typ beinhaltet weder Attribute noch Kindelemente. Ein Element dieses Typs wird als atomares Element bezeichnet. Mit Hilfe des <*restriction*>-, <*list*>- und <*union*>-Elements und den so genannten Facetten ist es jedoch möglich, einfache Typen auf die eigenen Bedürfnisse abzustimmen.
Wesentlich mehr Möglichkeiten eigene Datentypen zu definieren, bieten die komplexen Datentypen. Mit ihnen ist es möglich Elemente und Attribute in nahezu jeder beliebigen Struktur zu kombinieren.

***Vererbung***

Ähnlich wie in objektorientierten Programmiersprachen ist es bei XML Schema möglich, Typen von anderen Typen abzuleiten. XML Schema verfügt dabei über einen Ur-Typ, oder auch Basistyp genannt (siehe Abbildung 21), von dem alle einfachen und komplexen Typen abgeleitet werden. Eine Ausnahme ergibt sich durch die Angabe eines anderen Basistypen mittels des Eigenattributes *source* von Elementen (<*element … source=…>)*.
Dabei ist es möglich bestehende Typdefinitionen zu erweitern oder einzuschränken. XML Schema unterstützt darüber hinaus das Konzept der abstrakten Typen, welche in Instanzen nicht direkt genutzt werden können. Abstrakte Typen stehen nur anderen Typdefinitionen zur Verfügung, um sie zu erweitern oder einzuschränken. Umgekehrt ist es möglich, Typdefinition als „final“ zu deklarieren, um zu verhindern, dass weitere Typen von diesem Typ abgeleitet werden.

# 3. X-Ray

In diesem Abschnitt der Diplomarbeit soll das Konzept und die Architektur von X-Ray erklärt werden. Sämtliche Informationen und Daten in diesem Kapitel der Diplomarbeit wurden [KAPP04] entnommen.

Es gibt mehrere Ansätze und Realisierungen zum Speichern und Verwalten von Daten aus XML-Dokumenten in relationale Datenbanken. Jedoch haben diese Ansätze allesamt unterschiedliche Schwerpunkte. Ein Überblick hierzu wird in [Kapp04] gegeben.

X-Ray wurde entwickelt, um Daten aus XML-Dokumenten mit Hilfe eines Metaschemas in Relationen einer oder mehrerer relationalen Datenbanken zu speichern und auch wieder auslesen zu können, um die gespeicherten Daten in wohlgeformte und gültige XML-Dokumente zu exportieren.

Ein wichtiges Ziel von X-Ray ist die Flexibilität bezüglich der abzubildenden Schemata. Hierfür wurde der sogenannte „benutzerdefinierte“ Ansatz gewählt, welcher die Autonomie der Schemata der XML-Dateien (in Form von DTDs) sowie die Schemata des RDBS (Relationales Datenbanksystem) gewährleistet. Die Schemainformationen der XML-Dokumente sowie jene der Relationen des RDBS werden unabhängig voneinander gespeichert und verwaltet. Die Daten, welche die Informationen zur Abbildung des Schemas der XML-Dateien auf das relationale Schema enthalten, werden im sogenannten „Mappingschema“ gespeichert.

Das Schema eines XML-Dokumentes wird im Folgenden als XML-Schema bezeichnet, unabhängig davon, ob es sich bei der Schemasprache um DTDs oder um XML Schema [W3C04B] handelt.

## 3.1. Designziele von X-Ray

***Art der Schemata***

Bei X-Ray wird wie erwähnt vom „benutzerdefinierten“ Ansatz ausgegangen. Das Gegenteil dazu ist der „abgeleitete“ Ansatz, bei welchem das Datenbankschema vom XML-Schema oder vice versa nach vordefinierten Regeln abgeleitet wird. Beim „benutzerdefinierten“ Ansatz können Datenbankschema als auch Schema eines XML-Dokumentes (in Form einer DTD) unabhängig voneinander entwickelt werden. Dies erlaubt eine höhere Flexibilität. Das Abbilden zwischen den beiden Schemata kann dann jedoch nicht automatisiert nach vordefinierten Regeln ablaufen, sondern wird vom Benutzer von X-Ray durchgeführt.

***Darstellung der Abbildungsinformationen***

Um Abbildungstransparenz zu erreichen, müssen die Abbildungsinformationen (*mapping knowledge*) in irgendeiner Form gespeichert und verwaltet werden. Das Einbinden der Abbildungsinformationen in den Applikationscode ist nachteilig, da bei Änderungen immer der Programmcode der Applikation geändert werden müsste. Da ein wichtiges Designziel von X-Ray neben einem transparenten Mapping (Abbilden eines Schemas auf ein anderes) eine einfache Wartbarkeit ist, werden sämtliche Metadaten, das sind Informationen über die Schemata und über die Abbildung zwischen diesen, in einem Metaschema verwaltet, welches in einem RDBS gespeichert wird.

***Bindung an Schemata***

Bei X-Ray wird die „lose“ Bindung der Abbildungsinformationen an das XML-Schema bzw. DB-Schema realisiert. Dies bedeutet, dass die Abbildungsinformationen unabhängig von den Schemata gespeichert werden können, was wesentlich zur Flexibilität beiträgt.

***Abbildungskardinalität***

Ein weiteres wichtiges Designziel von X-Ray ist die Unterstützung von „multiplen Schemata“. Eine Abbildung (*Mapping*) ist genau für ein XML-Schema und ein DB-Schema gültig. Jedoch kann es mehrere Abbildungen zwischen einem XML-Schema und beliebig vielen DB-Schemata sowie vice versa geben, so auch z.B. mehrere Abbildungen eines XML-Schemas auf verschiedene DB-Schemata eines RDBS.

***Zugriffsmöglichkeit***

X-Ray unterstützt den „*unified approach*“. Dies bedeutet, dass man mit X-Ray nicht nur Daten aus XML-Dokumenten speichern, sondern auch Daten aus einer DB in XML-Format ausgeben kann.

***Zugriffssprache***

Für den Zugriff auf in der DB gespeicherte Daten gibt es den XML-zentrierten sowie den DB-zentrierten Ansatz für Abfragesprachen. X-Ray unterstützt den XML-zentrierten Ansatz und verwendet daher eine XML-basierte Abfragesprache. (z.B. X-RAYQL [KIMM02] oder XQuery [W3C04E])

***Zugriffsziel***

Beim Zugriff auf X-Ray wird eine virtuelle XML-Sicht (*View*) der in einer DB gespeicherten Daten zur Verfügung gestellt. Der Benutzer benötigt daher z.B. beim Erstellen von Abfragen kein Wissen über das DB-Schema.

## *3.2. Arbeitsphasen von X-Ray*

X-Ray lässt sich bzgl. seiner Verwendung in zwei Hauptphasen unterteilen:

- Initialisierungsphase
- Laufzeitphase

Diese beiden Phasen sind von einander abhängig, d.h. zuerst ist die Initialisierungsphase zu durchlaufen und danach ist die Laufzeitphase von X-Ray verfügbar.

### *3.2.1. Initialisierungsphase*

Mit X-Ray können Daten nur verwaltet werden, wenn die entsprechenden Metadaten der XML-Dokumente, der RDBS sowie die Abbildungsdaten, gespeichert in einem Metaschema, vorhanden sind. In der Initialisierungsphase können Schemainformationen von XML-Dokumenten ebenso wie von RDBS importiert werden. Mittels des im Abschnitt ***Architektur von X-Ray*** beschriebenen *„Mapping Knowledge Editors“* kann die Abbildung eines XML-Schemas auf das Schema einer RDBS (deren Relationen) vorgenommen werden. Die dabei verwendeten Abbildungsinformationen werden als Abbildungsdaten in dem Metaschema gespeichert.

Ein Metaschema speichert nach der Initialisierungsphase die Schemadaten von XML-Dokumenten, die Schemadaten der verwendeten Relationen eines RDBS und letztendlich die zugehörigen, vom Benutzer erstellten, Abbildungsdaten (*Mappingdaten*). Diese Phase sollte nur dem X-Ray-Administrator zugängig sein.

### *3.2.2. Laufzeitphase*

Das Importieren von Daten aus XML-Dokumenten, das Exportieren von Daten in ein XML-Dokument, sowie das Ausführen von Abfragen auf gespeicherte Daten erfolgt in der Laufzeitphase von X-Ray.
In dieser Phase kann ein beliebiges Metaschema aus dem im Abschnitt ***Architektur von X-Ray*** beschriebenen „*Metaschema-Repository*“ geladen und anschließend verwendet werden.
Des Weiteren ist es in dieser Phase möglich, Abfragen auf die gespeicherten Daten zu erstellen und bei Bedarf das Ergebnis zu exportieren. Diese Phase ist dem (der) X-Ray-Benutzer(in) zugänglich und stellt den Standardbetrieb von X-Ray dar.

## *3.3. Architektur von X-Ray*

Die statische X-Ray-Architektur, welche in Abbildung 3 dargestellt ist, setzt sich aus folgenden Komponenten, die nachfolgend kurz beschrieben werden, zusammen:

*Metaschema:*
Den Kern von X-Ray bildet das zuvor erwähnte Metaschema. In einem Metaschema werden die Schemainformationen von XML-Dokumenten, der Relationen des (der) RDBS, sowie die entsprechenden Abbildungsdaten verwaltet.

*Decomposer:*
Um Daten aus einem XML-Dokument in die Datenbank zu importieren, wird das XML-Dokument mittels des Decomposers „zerlegt“ und die Daten mittels der XML-Schemainformationen und der gespeicherten Abbildungsdaten in die entsprechenden Relationen des RDBS gespeichert.

*Composer:*
Zu exportierende Daten werden mit Hilfe der gespeicherten Abbildungsdaten aus den Relationen ausgelesen und entsprechend den Daten des gespeicherten XML-Schemas im „Composer“ zu einem XML-Dokument zusammengestellt.

*Mapping Knowledge Editor:*
Diese Komponente von X-Ray dient zum Erstellen von Abbildungsdaten. Mit Hilfe dieser Komponente kann die Abbildung eines XML-Schemas auf ein Schema eines RDBS vorgenommen werden. Diese Komponente kommt in der Initialisierungsphase zum Einsatz.

*Domain DB:*
Die Daten aus XML-Dokumenten werden persistent in einem RDBS gespeichert. X-Ray ermöglicht es, ein XML-Schema auf verschiedene Schemata eines RDBS abzubilden. Die *DomainDB* repräsentiert hierbei ein solches RDBS. In solch einer *DomainDB* sind die Daten eines oder mehrerer XML-Dokumente gespeichert.

*Metaschema-Repository:*
Im *Repository* werden sämtliche Daten des Metaschemas persistent gespeichert. Beim Start von X-Ray wird das Metaschema aus dem „*Metaschema-Repository*“ geladen und X-Ray damit initialisiert. Somit steht das Metaschema während der Laufzeit von X-Ray zur Verfügung. Sämtliche Metadaten werden bei X-Ray persistent in einem RDBS gespeichert.

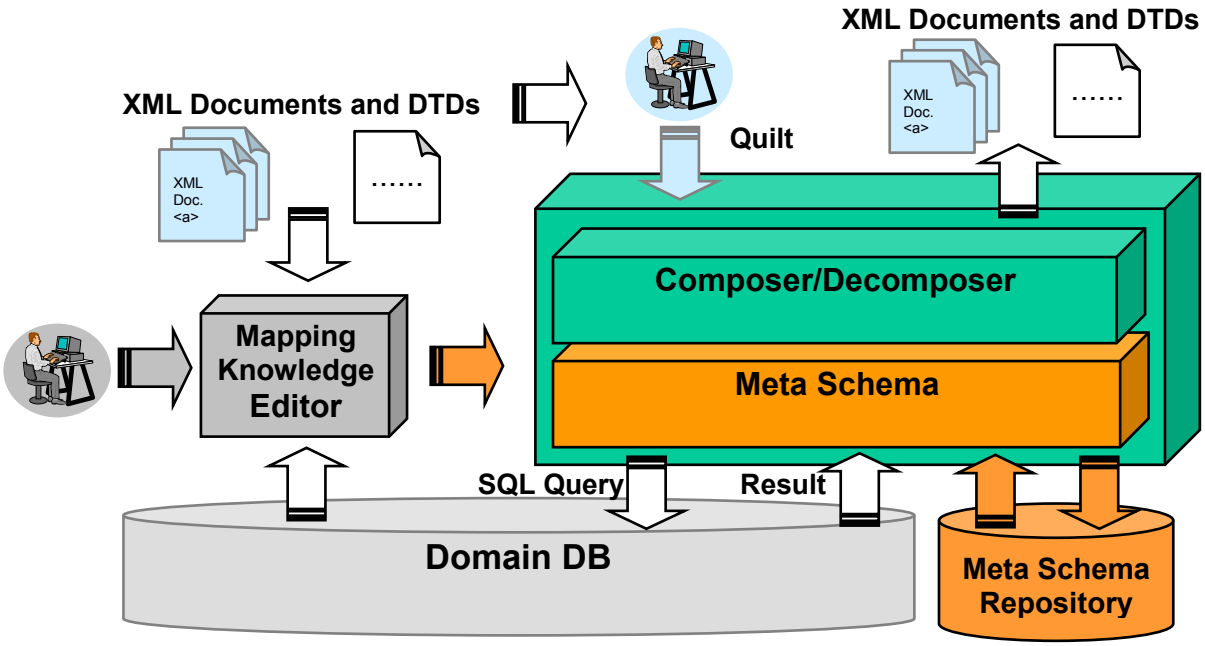



**Abbildung 3 - Architektur von X-Ray**

## 3.4. Abbildungsmuster zwischen XML-Schema und DB-Schema

Um ein XML-Schema auf ein DB-Schema abzubilden, mussten allgemeine Abbildungsmuster bzw. grundlegende Abbildungsregeln entwickelt werden.

In [KAPP04] werden die Unterschiede zwischen den Konzepten von XML sowie RDBS genauer analysiert. Auf Basis dieser Analyse der Konzepte konnten sinnvolle Abbildungsmuster (*mapping patterns*), die wiederum die Basis für das Metaschema darstellen, entwickelt werden.

### 3.4.1. Grundlegende Abbildungsarten

Der einfachste Weg wäre, die XML-Elemente direkt auf Tupel von Relationen der RDBS abzubilden. Dabei werden Elementtypen auf Relationen und die Attribute der Elemente auf die Attribute der entsprechenden Relation abgebildet. Dies ist jedoch aufgrund der Heterogenität zwischen einem XML-Schema und einem DB-Schema nicht immer möglich bzw. sinnvoll. Zusätzlich ergeben sich erhebliche Nachteile bezüglich Performanz und Fragmentierung des RDBS bei tief geschachtelten XML-Elementen.

Unter Berücksichtigung der Strukturen der beiden Schemata wurden für X-Ray drei grundlegende Abbildungsarten entwickelt.

Diese drei Abbildungsarten sind:

- ET_R: Ein Elementtyp (ET) wird auf eine Relation der DB abgebildet. Mehrere Elementtypen können auf eine so genannte Basisrelation (*base relation*) abgebildet werden.
- ET_A: Ein Elementtyp wird auf ein Attribut einer Relation der DB abgebildet, wobei die Relation des Attributes die Basisrelation des Elementtyps darstellt. Mehrere Elementtypen können auf die Attribute einer Basisrelation abgebildet werden.
- A_A: Das Attribut eines Elementtyps wird auf ein Attribut jener Relation abgebildet, welche die Basisrelation des Elementtyps, zu dem das XML-Attribut gehört, repräsentiert.

In Abbildung 4 werden diese drei Abbildungsarten beispielhaft dargestellt.

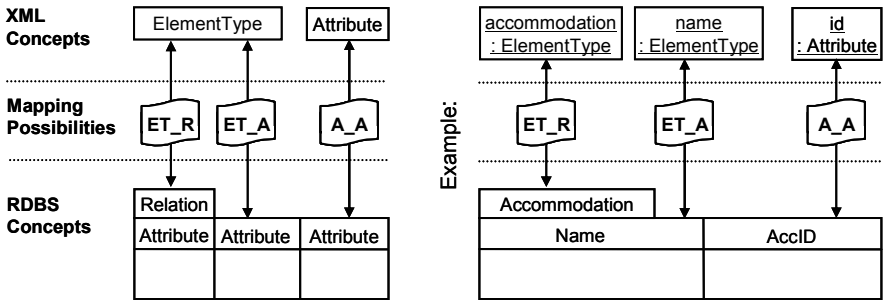



**Abbildung 4 - Grundlegende Abbildungsmuster zwischen XML und DB**

Es ist sinnvoll, dass ET_R in Verbindung mit ET_A und/oder A_A verwendet wird. Des Weiteren ist es nicht zwingend notwendig, Abbildungsmuster für alle Elementtypen bzw. deren Attribute, sowie für alle Relationen und deren Attribute zu verwenden. Dies ist z.B. bei Surrogatschlüsseln von Relationen der DB, welche lediglich Fremdschlüssel für Beziehungen zwischen einzelnen Relationen darstellen, der Fall. Ein Beispiel auf XML Seite ist ein leerer Elementtyp, der exakt nur einmal vorkommt.

Das Konzept der Basisrelation kann noch verfeinert werden. Man betrachtet das erste der Vorgänger-Elementtypen des abzubildenden Elementtyps, das auf eine Relation oder ein Attribut abgebildet wird. Dessen Basisrelation bildet nun die Eltern-Basisrelation (*parent base relation*) des abzubildenden Elementtyps und ist zugleich ein Kandidat als dessen Basisrelation. Ist keiner der Vorgänger-Elementtypen abgebildet worden und besitzt daher keine Basisrelation, kann eine beliebige Relation als Basisrelation verwendet werden. Dies wird in Abbildung 5 verdeutlicht.

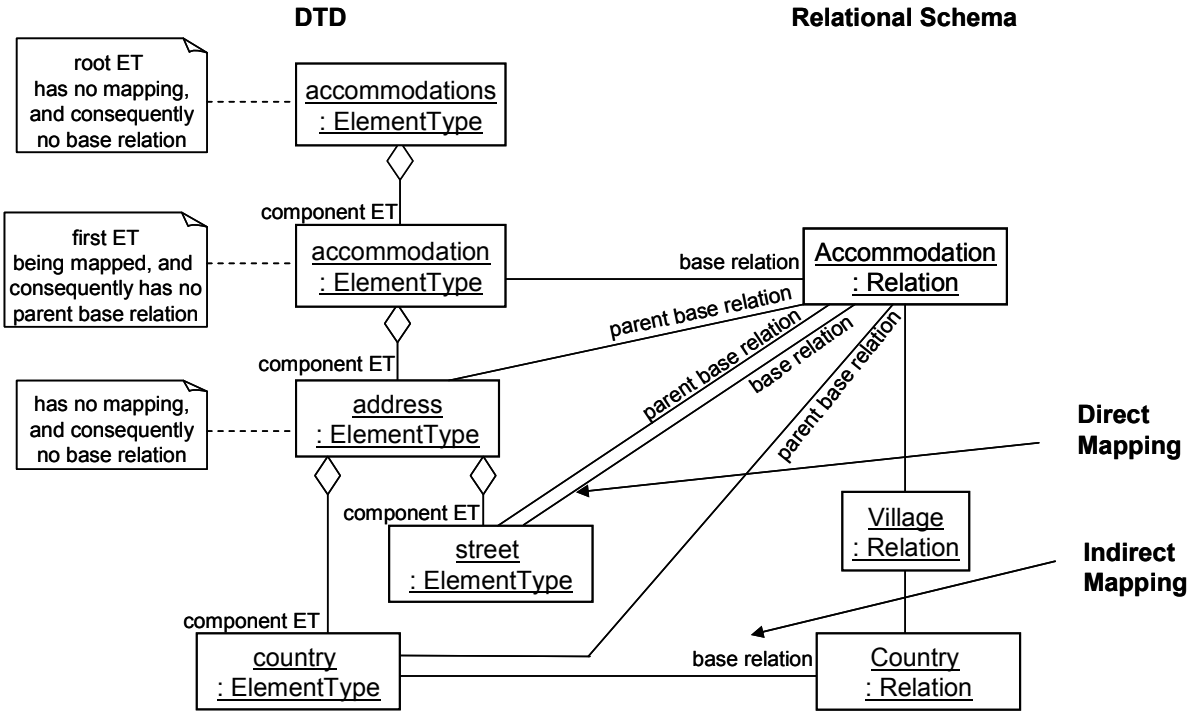



**Abbildung 5 - Verfeinerte Abbildungskonzepte**

Aus den Konzepten der Basisrelation sowie Eltern-Basisrelation wird das Konzept des direkten und indirekten Abbildens (*direct/indirect mapping*) abgeleitet.

Soll ein XML-Attribut abgebildet werden, so wird zuerst dessen Elementtyp betrachtet. Hat dieser jedoch keine Basisrelation, so wird nach dem zuvor erwähnten Ablauf vorgegangen, um eine Basisrelation und/oder Eltern-Basisrelation zu eruieren. Die Eltern-Basisrelation stellt dann zugleich die Basisrelation dar, wenn ein Elementtyp bzw. ein Attribut des Elementtyps auf diese Relation oder eine ihrer Attribute abgebildet werden kann. In diesem Fall spricht man vom direkten Abbilden (*direct mapping*). In Abbildung 5 wird der Elementtyp „*street*“ direkt auf ein Attribut seiner Eltern-Basisrelation „*Accommodation*“ abgebildet.

Wird hingegen eine geeignete Basisrelation gefunden, die über Fremdschlüsselbeziehungen von der Eltern-Basisrelation aus erreicht werden kann, so spricht man von indirektem Abbilden (*indirect mapping*). In Abbildung 5 wird dies durch den Elementtyp „*country*“, welcher indirekt auf die Basisrelation „*Country*“ (mit Eltern-Basisrelation „*Accommodation*“) abgebildet wird, dargestellt. Indirektes Abbilden ist sinnvoll, wenn ein Elementtyp oder Attribut aufgrund von Normalisierung der Relationen in eine eigene Relation ausgelagert werden soll.

Somit ergeben sich letztendlich die sechs Abbildungsarten $ET\_R_{direkt/indirekt}$, $ET\_A_{direkt/indirekt}$ und $A\_A_{direkt/indirekt}$.

### *3.4.2. Sinnvolle Abbildungsstrategien*

Bisher wurden bei den Abbildungsstrategien die verschiedenen Arten von XML-Elementtypen sowie XML-Attributtypen noch nicht berücksichtigt. Für das Entwickeln sinnvoller Abbildungsstrategien ist dies jedoch unerlässlich. Die resultierenden Abbildungsmuster sollen einerseits den Abbildungsprozess auf der syntaktischen Ebene vereinfachen und andererseits Abbildungen vermeiden, bei denen es zu syntaktischen Konflikten kommt. Dieser Ansatz unterscheidet sich wesentlich von [9, 32, 54, 59 – KAPP04].

Die bestimmenden Faktoren können wie folgt unterteilt werden:

- Eigenschaften von XML-Elementtypen
- Eigenschaften von XML-Attributtypen

Diese Eigenschaften werden nun näher beleuchtet.

#### *3.4.2.1. Eigenschaften von XML-Elementtypen*

Die Eigenschaften von XML-Elementtypen lassen sich in drei Dimensionen unterteilen:

1. Art des Elementtyps
2. Existenz von Attributen
3. Kardinalität

In Abbildung 6 werden diese drei entscheidenden Eigenschaften mitsamt ihren Ausprägungen dargestellt.

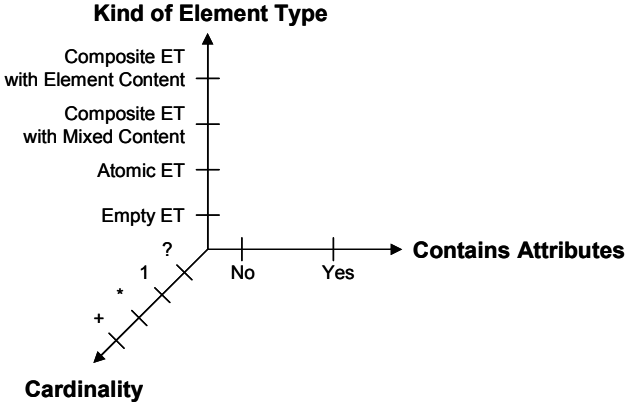



**Abbildung 6 - Charakteristische Eigenschaften von XML-Elementtypen**

Abhängig von der Kombination der Ausprägung dieser drei Eigenschaften lassen sich sinnvolle Abbildungsstrategien ableiten. Weist jedoch ein Elementtyp das Inhaltskonzept *ANY* auf, ist keine Abbildungsstrategie im Vorhinein bestimmbar. Eine Übersicht über die Abbildungsstrategien wird in Abbildung 7 gegeben.

| Kind of element type | Contains attributes | Cardinality | Reasonable mapping |
|---|---|---|---|
| Composite ET with element content | No influence | No influence | ET_$R_{direct/indirect}$; No mapping |
| Atomic ET | No influence | ?, 1 | ET_$A_{direct/indirect}$ |
| Atomic ET | No influence | +, * | ET_$A_{indirect}$ |
| Empty ET | No | 1 | No mapping |
| Empty ET | Yes | 1 | ET_$R_{direct/indirect}$; No mapping |
| Empty ET | No influence | ? | ET_$A_{direct}$ |
| Empty ET | No influence | *, + | ET_$A_{indirect}$ |
| Composite ET with mixed content | No influence | No influence | ET_$A_{indirect}$ |

**Abbildung 7 - Sinnvolle Abbildungsstrategien für XML-Elememttypen**

Im Folgenden werden die einzelnen XML-Elementtypen näher betrachtet, um die Abbildungsstrategien verständlicher zu machen.

*Zusammengesetzter ET mit Element-Inhalt (Composite ET with element content):*
Mit Elementen dieses Typs sind keine zu speichernden Daten verbunden, woraufhin die einzig sinnvolle Abbildungsart ET_R ist. Direkt oder indirekt wird abgebildet, abhängig davon, ob dieses Element auf die Eltern-Basisrelation abgebildet werden kann oder nicht (siehe Abschnitt ***Grundlegende Abbildungsarten***). Es käme keinem Informationsverlust gleich, wenn es kein Abbildungsmuster für dieses Element gäbe, da es keine Daten enthält, die in der DB gespeichert werden.

*Atomarer ET (atomic ET):*
Bei Elementen dieses Typs ist ausschließlich die Kardinalität für das sinnvolle Abbilden ausschlaggebend. Da diese Elemente immer einen Wert enthalten, muss dieser Wert in einem Attribut einer Relation gespeichert werden, was logischerweise zur ET_A – Abbildungsart führt. Die Unterscheidung zwischen direktem und indirektem Abbilden ist abhängig von der Kardinalität. Ist diese als „1“ oder „?“ angeben, ist ET_$A_{direkt}$ sinnvoll. Wird das Element jedoch nicht auf ein Attribut der Eltern-Basisrelation abgebildet, bleibt nur ET_$A_{indirekt}$ als Möglichkeit.
ET_$A_{indirekt}$ ist auch bei Kardinalität „*“ und „+“ die einzig sinnvolle Variante.

*Leerer Elementtyp (empty ET):*

Hat ein Element dieses Typs die Kardinalität „1“, egal ob es Attribute enthält oder nicht, ist ein Abbilden unnötig, da es ja nur exakt einmal vorkommt und keinen Wert enthält. Enthält ein leerer Elementtyp jedoch Attribute, ist ET_$R_{direkt/indirekt}$ sinnvoll, da die Basisrelation auch als Basisrelation für die XML-Attribute dienen kann.

Hat ein leerer Elementtyp eine andere Kardinalität als „1“, ist es egal, ob er Attribute enthält, weshalb ET_A die einzig sinnvolle Abbildungsart ist. Ob direkt oder indirekt abgebildet wird, ist nur von der jeweiligen Kardinalität abhängig.

*Zusammengesetzter ET mit gemischtem Inhalt (Composite ET with mixed content):*

Die Abbildungsart für diesen Elementtyp ist gänzlich unabhängig von dessen Kardinalität und ob er Attribute enthält oder nicht.

Da auf der Instanzebene mehrere Daten in einem Element vorkommen können, ist jedoch nur ET_$A_{indirekt}$ sinnvoll.

#### *3.4.2.2. Eigenschaften von XML-Attributtypen*

Die Eigenschaften von XML-Attributen lassen sich in zwei Dimensionen unterteilen:

1. Multiplizität des XML-Attributes
2. Standarddeklaration des XML-Attributes

In Abbildung 8 werden diese zwei entscheidenden Eigenschaften mitsamt ihren Ausprägungen dargestellt.

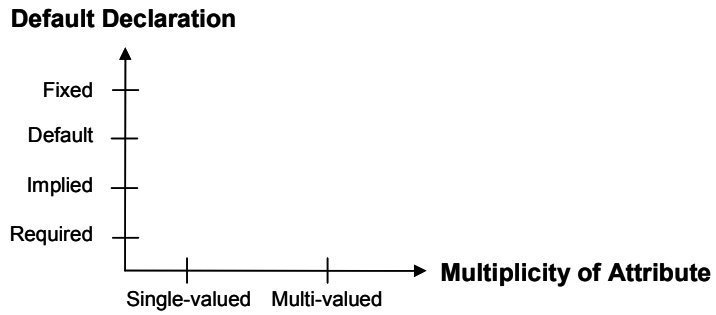



**Abbildung 8 - Charakteristische Eigenschaften von XML-Attributen**

Abhängig von der Kombination der Ausprägung dieser beiden Eigenschaften lassen sich sinnvolle Abbildungsstrategien ableiten. Eine Übersicht über die Abbildungsstrategien wird in Abbildung 9 gegeben.

| Multiplicity of XML attribute | Default declaration | Reasonable mapping |
|---|---|---|
| No influence | #FIXED | No mapping |
| Single-valued | #REQUIRED, #IMPLIED, Default Value | $A\_A_{direct/indirect}$ |
| Multi-valued | #REQUIRED, #IMPLIED, Default Value | $A\_A_{indirect}$ |

**Abbildung 9 - Sinnvolle Abbildungsstrategien für XML-Attribute**

Für Attribute mit der Standarddeklaration „FIXED“ ist kein Mapping nötig, da der Fixwert im Abbildungswissen gespeichert werden kann.
Bei den Attributen, die eine andere Standarddeklaration als „FIXED“ aufweisen, ist nur die Multiplizität (*multiplicity*) des XML-Attributes ausschlaggebend, ob ein direktes oder indirektes Abbilden auf ein Attribut einer Relation der DB verwendet werden soll. Bei einmal vorkommenden Attributwerten kann entweder $A\_A_{direkt}$ verwendet werden, um den Wert in einem Attribut einer Relation der DB zu speichern oder auch $A\_A_{indirekt}$ auf Grund der Normalisierung einer Relation. Bei mehrfachem Vorkommen von Attributwerten kann nur $A\_A_{indirekt}$ verwendet werden.

## *3.5. Metaschema im Detail*

Das Kernstück von X-Ray, dessen Architektur im Abschnitt ***Architektur von X-Ray*** beleuchtet wurde, stellt das Metaschema dar. Es ist der Schlüsselmechanismus für die Generizität von X-Ray um DTDs auf relationale Schemata und vice versa abzubilden. Die Heterogenitäten zwischen den Datenmodellen und denen der Schemata, welche in [KAPP04 S.345ff] genauer erläutert werden, sowie die zuvor beschriebenen Abbildungsstrategien, bilden die Basis für das Metaschema.
Das Metaschema besteht, wie in Abbildung 10 ersichtlich, aus drei Komponenten:

- *DBSchema*
- *XMLDTD*
- *XMLDBSchemaMapping*

Die *DBSchema*-Komponente ist zuständig für das Speichern der Informationen über die Schemata der Relationen, welche auf DTDs abgebildet werden sollen, um in ihnen gespeicherte Daten in XML-Dokumente exportieren zu können.
Die *XMLDTD*-Komponente speichert analog dazu die Schemainformationen der XML-Dokumente, nämlich die Metadaten der entsprechenden DTDs.

Die *XMLDBSchemaMapping*-Komponente überbrückt die Heterogenität der Datenmodelle sowie der Schemata und unterstützt dadurch ein korrektes Abbilden.

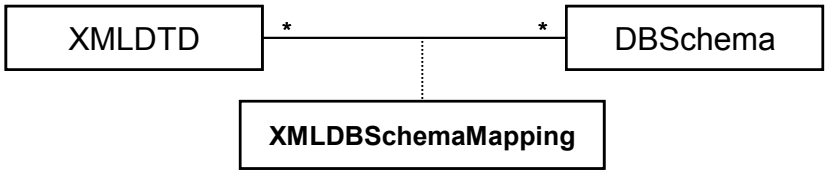



**Abbildung 10 - Komponenten des X-Ray Metaschemas**

Bei X-Ray wird das Abbildungswissen (*mapping knowledge*) auf zwei Arten verwaltet. Gespeichert wird das Abbildungswissen in einer relationalen Datenbank und es ist in der Laufzeitphase als objektorientierte Repräsentation verfügbar.
Das Abbildungswissen wird zum Start einer X-Ray-Session aus dem RDBS in eine objektorientierte Repräsentation geladen und steht dann für das effiziente Zusammenstellen sowie Zerlegen von XML-Dokumenten zur Laufzeit der X-Ray-Session zur Verfügung.
Im Folgenden werden die drei Komponenten näher betrachtet.

### *3.5.1. DB-Schema Komponente*

Es muss nicht das gesamte relationale Schema gespeichert werden, sondern nur jene Relationen und Attribute, die relevant für das Abbilden auf DTDs sind. Dazu gehören auch jene Relationen, die z.B. lediglich als Verbindungsrelationen zwischen zwei Basisrelationen dienen. Wie in Abbildung 11 ersichtlich, besteht die *DBSchema-Komponente* aus mindestens einer *DBRelation* mit zumindest einem *DBAttribute*, die beide zum *DBConcept* generalisiert werden.

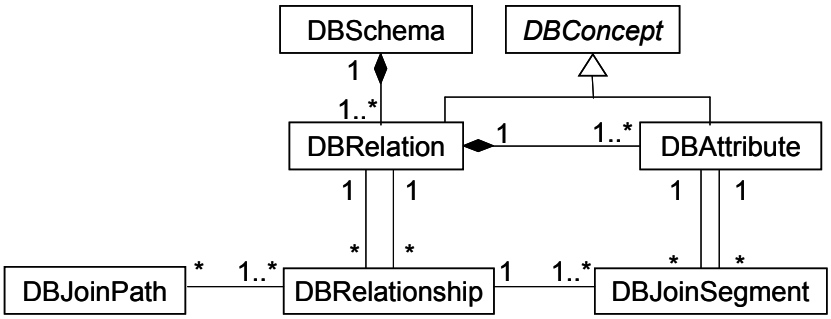



**Abbildung 11 - DB-Schema Komponente des Metaschemas**

Ein *DBAttribute* speichert einen Wert und zusätzlich, ob es der Primärschlüssel oder auch nur Teil des Primärschlüssels der Relation ist. Mittels Beziehungen (*DBRelationship*) werden zwei Relationen verbunden, wobei diese Beziehungen Teil

einer oder mehrerer Verbindungsstücke (*DBJoinSegment*) darstellen können. Die Attribute dieser Verbindungsstücke sind Primär- und Fremdschlüssel zweier verbundener Relationen. Setzt sich der Primärschlüssel aus mehreren Attributen zusammen, dann besteht eine Beziehung aus mehreren Verbindungsstücken. Sind Teile eines XML-Dokumentes auf verschiedene weiter verzweigte Relationen verteilt, muss die Information über entsprechende Verbindungspfade (*DBJoinPath*) gespeichert werden. Ein Verbindungspfad setzt sich aus mindestens einem oder mehreren Beziehungen (*DBRelationship*) zusammen.

### *3.5.2. XMLDTD-Komponente*

Analog zur *DBSchema-Komponente* muss in der *XMLSchema-Komponente* nicht die gesamte DTD gespeichert werden, sondern nur jene Informationen, die zur Abbildung notwendig sind. Entsprechend dem Metawissen muss eine DTD einen Elementtyp (*XMLElemType*) haben, der die Wurzel (*root*) repräsentiert.

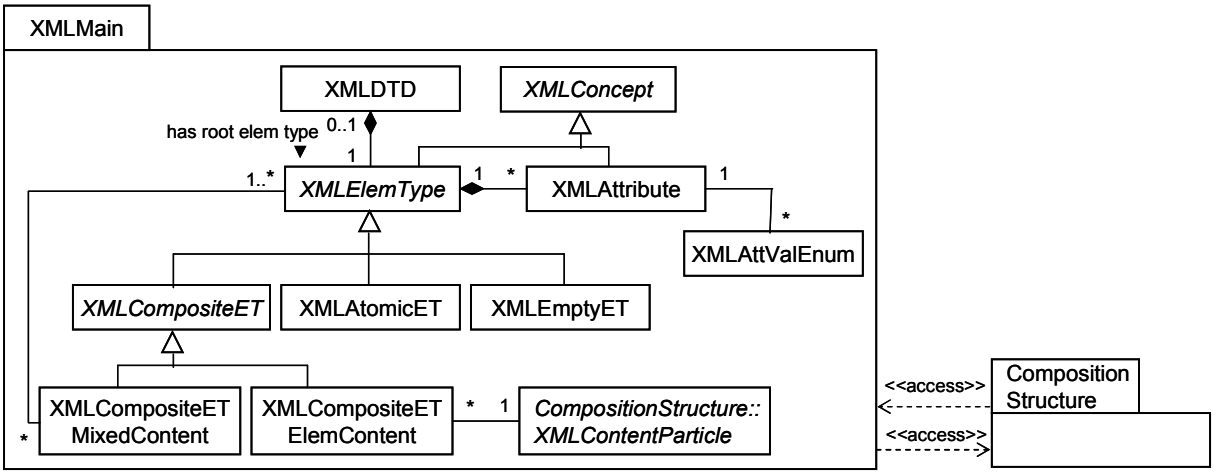



**Abbildung 12 - XMLDTD-Komponente des Metaschemas**

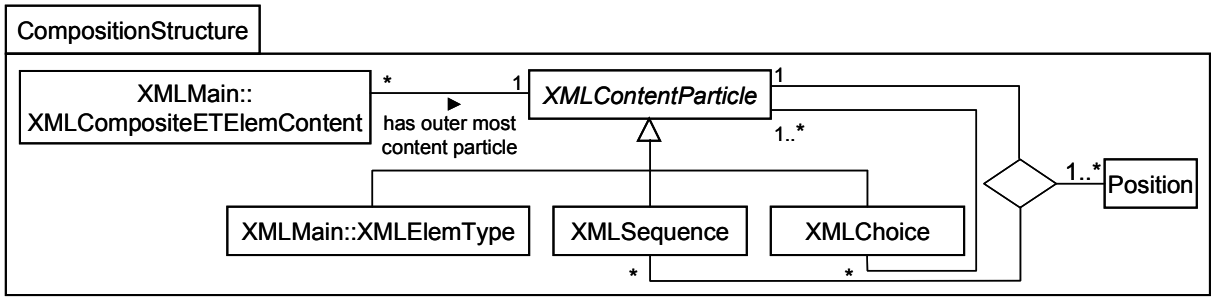



**Abbildung 13 - Verfeinerung des zusammengesetzten ET**

In Abbildung 13 ist die Verfeinerung der in Abbildung 12 dargestellten Komponente *CompositeStructure* dargestellt.

Der Wert sowie die Standarddeklaration von XML-Attributen, die einem Elementtypen angehören, werden in *XMLAttribute* gespeichert. *XMLElemType* und *XMLAttribute* werden zu *XMLConcept* generalisiert.

Die möglichen Werte für Enumerations-Attribute werden in *XMLAttVAlEnum* gespeichert. *XMLElemType* wird weiter unterteilt in die verschiedenen Ausprägungsarten, die im Abschnitt ***Eigenschaften von XML-Elementen*** näher ausgeführt sind. Die verschachtelte Struktur eines Elementtyps wird mittels des Paketes *CompositeStructure* beschrieben. Für Elementtypen, die im Zuge einer *XMLChoice* oder *XMLSequence* vorkommen, wird die Kardinalität berücksichtigt und bei Elementtypen einer *XMLSequence* zusätzlich die Position des Elementtyps in der *Sequence*. Zusätzlich können mittels *XMLContentParticle* beliebige Kombinationen von *Sequence* und *Choice* beschrieben werden.

### *3.5.3. XMLDBSchemaMapping-Komponente*

Das Abbildungswissen wird durch die Assoziationen zwischen Objektklassen der *XMLDTD*-Komponente und jenen der *DBSchema-Komponente* repräsentiert. In Abbildung 14 werden diese Assoziationen durch fett gedruckte Linien dargestellt.

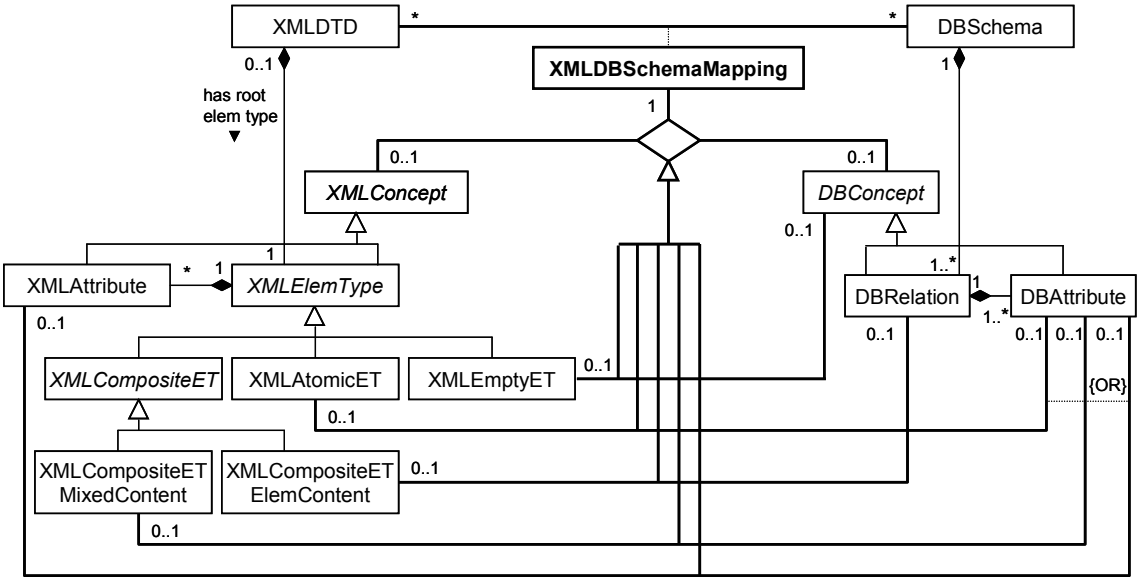



**Abbildung 14 - Ausschnitt aus dem Metaschema (XMLDBSchemaMapping-Komponente )**

Durch die ternäre Beziehung zwischen *XMLConcept*, *DBConcept* und *XMLDBSchemaMapping* wird ein Hauptziel von X-Ray, nämlich die Abbildungen zwischen mehreren DTDs und Datenbankschemata zu speichern, realisiert. Wie ein Elementtyp tatsächlich abgebildet wird, hängt, wie im Abschnitt ***Eigenschaften von XML-Elementen*** diskutiert, von dessen Art und Aufbau ab. Daher werden bis auf

zwei Ausnahmen alle Blätter der *XMLElemType*-Hierarchie auf *DBAttribute* abgebildet. Dies sind der *XMLEmptyET* und der *XMLCompositeETElemContent*. Im Besonderen ist das Abbilden von *XMLEmptyET* nicht genau vordefinierbar, da es abhängig davon ist, ob diese Art von Elementtyp Attribute enthält oder nicht, sowie von dessen Kardinalität. Das Abbilden der XML-Attribute erfolgt gemäß der im Abschnitt ***Eigenschaften von XML-Attributen*** vorgestellten Abbildungsarten.

# 4. Erweiterung von X-Ray zur Integration von XML Schema und RDBS

Im vorhergehenden Kapitel wurde der Aufbau, die Abbildungskonzepte und das Metaschema von X-Ray vorgestellt. Wie in dem Kapitel beschrieben, werden DTDs als Basis zum Abbilden von XML-Dokumenten auf ein oder mehrere DB-Schemata verwendet. Im Abschnitt **Future work** von [KAPP04] wird vorgeschlagen, X-Ray dahingehend zu erweitern, um auch das Abbilden von XML-Dokumenten auf Basis von XML Schema zu realisieren. Diese erweiterte Variante von X-Ray trägt die Bezeichnung X-Rayxs.

XML Schema wird zunehmend häufiger zum Beschreiben der Struktur von XML-Dokumenten verwendet. Die Vorteile (und auch Nachteile) von XML Schema gegenüber DTD wurden bereits im Kapitel **Vergleich zwischen DTD und XML Schema** näher beleuchtet.

Da XML Schema im Vergleich zu DTDs ungleich umfangreicher ist, konnten nicht alle Konzepte von XML Schema [W3C04B] im Zuge der Diplomarbeit berücksichtigt werden. Ein Überblick über die berücksichtigten und explizit nicht berücksichtigten Konzepte von XML Schema sowie eine genauere Beschreibung der Konzepte wird in dem Abschnitt **Konzepte von XML Schema** gegeben. Um XML Schema zu unterstützen mussten die drei Komponenten des Metaschemas überarbeitet bzw. teilweise neu entwickelt werden. Die Beschreibung des Metaschemas mitsamt seiner Komponenten wird im Abschnitt **Metaschema** dieses Kapitels gegeben.

Am Ende dieses Abschnittes werden mittels Auszügen aus einem repräsentativen XML-Dokument und dem zugehörigen Schema Beispiele für das Zusammenspiel der einzelnen Komponenten des Metaschemas gegeben.

Im anschließenden Kapitel **Prototyp** wird ein im Zuge der Diplomarbeit [ORT05] entwickelter Prototyp vorgestellt, mit welchem die Laufzeitphase unter Verwendung des Metaschemas für XML Schema, getestet und demonstriert werden kann.

Im Zuge dieses Abschnitts werden auch die Algorithmen für die Funktionen von X-Rayxs in der Laufzeitphase (Import, Export, Abfragen) beschrieben.

## *4.1. Konzepte von XML-Schema*

### *4.1.1. Unterstützte XML Schema Konzepte*

In diesem Abschnitt werden die in X-Rayxs berücksichtigten Konzepte von XML Schema erläutert. Der Aufbau orientiert sich dabei an der „XML Schema - Structures Quick Reference Card“ [DVIN03A], in der sich auch die genaue Syntax zur Umsetzung der Konzepte befindet. Da die Berücksichtigung aller Konzepte von XML Schema, den Rahmen der Diplomarbeit sprengen würde, mussten Konzepte selektiert werden, die in X-Rayxs umgesetzt werden sollten. Die Konzepte wurden dahingehend ausgewählt, um einen größtmöglichen Teil der in XML Schema vorhandenen Konzepte umzusetzen, der nötig ist, um Standard-XML-Anwendungen behandeln zu können. Darüber hinaus wurde bei der Auswahl darauf geachtet, dass die im Kapitel **Vergleich zwischen DTD und XML Schema** vorgestellten Vorteile von XML Schema ebenso enthalten sind.
Die Paragraphen neben den Konzeptbezeichnungen beziehen sich auf das nicht normative Dokument „XML Schema Teil 0: Einführung“ [W3C04C]. Diese Einführung beschreibt die Sprachkonstrukte anhand zahlreicher Beispiele, ergänzt durch etliche Verweise auf die normativen Texte.

**1. Atomare Elemente** ***§2.3***

Ein Element auf Basis eines vordefinierten Datentyps (*simple predefined data type*) wird als atomares Element bezeichnet. Es besteht somit aus einem unteilbaren Wert. Beispielsweise besteht ein String zwar aus Zeichen, diese werden jedoch als Einheit betrachtet.

```
<element name="email" type="string"/>
```

**Abbildung 15 - Beispiel atomares Element**

## *2.* Komplexe Datentypen *§2.2*

Eine komplexe Typdefinition wird verwendet, um ein Inhaltsmodell, das Elemente und/oder Attribute enthält, zu spezifizieren. Wenn komplexe Typen einen Namen besitzen, können sie mehrfach zur Deklaration von Elementen verwendet werden, auch jene aus externen Schemadokumenten. Ansonsten können anonyme Typdefinitionen innerhalb einer Elementdeklaration erfolgen.

Mit Hilfe des komplexen Datentyps können folgende Inhalte definiert werden:

- Einfacher Inhalt (simple content)
- Komplexer Inhalt (complex content) in den Varianten:
  - Nur-Element Inhalt
  - Gemischter Inhalt (in X-Rayxs nicht unterstützt; siehe dazu Abschnitt **Komplexe Datentypen mit gemischtem Inhalt**)
  - Leerer Inhalt (siehe dazu Abschnitt **Leere Elemente**)

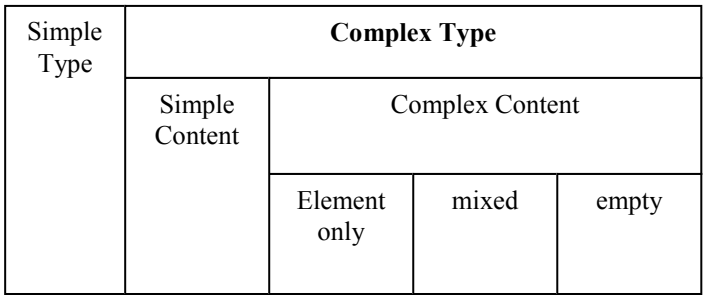



**Abbildung 16 - Übersicht Datentypen**

Abbildung 16 soll den Zusammenhang der möglichen Datentypen verdeutlichen. Bei den komplexen Datentypen mit Nur-Element-Inhalt dürfen Elemente, die mit diesem Datentyp deklariert werden, keinen einfachen eigenen Text beinhalten. Im Gegensatz dazu ist dies beim gemischten Inhalt möglich. Komplexe Datentypen mit einfachem Inhalt werden dazu verwendet, um einfache vordefinierte Datentypen mit Attributen zu versehen bzw. um diese erweitern oder einschränken zu können. Das Inhaltsmodell eines komplexen Datentyps beschreibt die Ordnung und Struktur der im Typ enthaltenen Elemente. Es setzt sich aus Modellgruppen (*sequence*, *choice*, *all*), Elementen und *wildcards* (in X-Rayxs nicht unterstützt) zusammen.

Die Inhaltsmodelle von benannten komplexen Typen können außerdem mit Hilfe der Vererbungsmechanismen erweitert (*extension*) oder eingeschränkt (*restriction*) werden.
Im folgenden Beispiel (vgl. Abbildung 17) wird ein benannter komplexer Typ „*address*“ definiert. Die Definition enthält in diesem Fall drei weitere Elementdefinitionen und eine Attributbeschreibung, wobei „village“ wiederum aus einem global definierten komplexen Datentyp besteht.
Die Reihenfolge der Elemente ist durch das <*sequence*>-Element festgelegt.

```
<complexType name="address">
      <sequence>
            <element name="street" type="string"/>
            <element name="village"
                  type="villageType"/>
            <element name="country" type="string"/>
      </sequence>
      <attribute name="postalCode" type="string"
                  use="required"/>
</complexType>
```

**Abbildung 17 - Beispiel komplexer Datentyp**

### *3.* Leere Elemente *§2.5.3*

Ein leeres Element speichert keinen Wert zwischen den Element-Tags, die Information liegt im Namen des Tags bzw. in den möglichen Attributen. Durch das Vorhandensein bzw. Nichtvorhandensein eines leeren Elements in der XML-Instanz entsteht ein weiterer Informationsgehalt des Elements. Ein leeres Element darf keine untergeordneten Elemente beinhalten. Das leere Element ist also ein komplexer Datentyp, dessen leerer Inhalt die Einschränkung von „anyType“ auf die leere Menge ist.

```
<element name="pool" minOccurs="0" maxOccurs="unbounded">
      <complexType/>
</element>
```

**Abbildung 18 - Beispiel leeres Element**

## *4.* Vererbung *§4*

Ähnlich wie in objektorientierten Programmiersprachen ist es auch in XML Schema möglich, Typen von anderen Typen abzuleiten.
Zur Unterstützung von Wiederverwendung und Erhöhung der Strukturierung des Entwurfs definiert XML Schema ein Vererbungskonstrukt zur Bildung neuer komplexer Typen auf der Basis bereits bestehender. Diese müssen allerdings so genannte „benannte“, globale komplexe Typen sein, um auf sie verweisen zu können.
Zwei verschiedene Ableitungsarten werden angeboten:

- Ableitung durch Einschränkung (*restriction*) - der erbende Subtyp beschreibt eine engere Definition des Supertypen
- Ableitung durch Erweiterung (*extension*) - der erbende Subtyp erweitert die Definition des Supertypen

XML Schema unterstützt darüber hinaus das Konzept der abstrakten Typen. Abstrakte Typen können in keinem XML-Instanzdokument vorkommen. Sie dienen nur anderen Typdefinitionen, um den abstrakten Typ einzuschränken oder zu erweitern. Umgekehrt kann eine Typdefinition als endgültig (*final*) deklariert werden, um zu verhindern, dass weitere Typen von diesem Typ abgeleitet werden.
Des Weiteren lassen sich auch vordefinierte Datentypen zu neuen Datentypen ableiten. Um zum Beispiel den Wertebereich des „decimal“ Typs auf Zahlen mit zwei Nachkommastellen für einen neuen Typ „Währung“ einzuschränken [KENN00] (Benutzerdefinierte einfache Typen (*simple data type*) werden in XRayxs nicht unterstützt).
Im folgenden Beispiel erbt „*standardRoomType*“ von „*roomType*“ und erweitert diesen um die Elemente „*shower*“ und „*tv*“.

```
<complexType name="standardRoomType">
   <complexContent>
      <extension base="act:roomType">
           <sequence>
                <element name="tv" type="int"
                     minOccurs="0" maxOccurs="2"/>
                <element name="shower" minOccurs="0">
                     <complexType/>
                </element>
           </sequence>
      </extension>
   </complexContent>
</complexType>
```

**Abbildung 19 - Beispiel Vererbung**

## *5.* Identifizierung und Referenzierung §5

Die XML Schema-Konzepte für Identifizierungen und Referenzierungen werden verwendet, um eindeutige Werte, Schlüssel oder Verweise auf Schlüssel festzulegen. Dies wird durch die Elemente *unique*, *key* und *keyref* gewährleistet.

Der Unterschied zwischen *key* und *unique* besteht darin, dass das mit *key* spezifizierte Element oder Attribut bzw. Kombinationen von Elementen und Attributen vorkommen und eindeutig sein müssen. U*nique* besagt, dass das Element, falls es vorkommt, eindeutig sein muss. *Keyref* entspricht im Wesentlichen dem Fremdschlüsselkonzept in RDBS (referenzielle Integrität).

Der Wert eines <*keyref*>-Elements muss also mit einem Wert (Schlüssel) an anderer Stelle im Dokument, welcher durch das <*key*>-Element beschrieben wird, übereinstimmen.

Das Schlüsselkonzept von XML Schema kann auf alle Elemente und Attribute angewandt werden, unabhängig von deren Typ. Der Bereich, in dem Eindeutigkeit gelten soll, kann festgelegt werden. Eindeutigkeit kann auch für Kombinationen aus mehreren Elementen und Attributen festgelegt werden. Die Identifizierungen und Referenzierungen werden durch ein <*selector*>-Element und einem oder mehreren <*field*>-Elementen beschrieben. Das <*selector*>-Element enthält einen Xpath-Ausdruck [W3C04F], der eine Menge von Elementen referenziert. Diese stellen den Bereich dar, für den Eindeutigkeit gewährleistet wird. Das <*field*>-Element enthält einen XPath

Ausdruck, der die Attribute bzw. Elemente identifiziert, die einen eindeutigen Wert besitzen sollen [ECKS04].

Im folgenden Beispiel ist der Schlüssel das Attribut „*id*“ vom Element „*accommodation*“. Des Weiteren wird durch das Element *keyref* festgelegt, dass der Wert des Elements „*winner*“ bereits als Schlüssel („*id*“ von „*accommodation*“) vorhanden sein muss.

```
<key name="AccKey">
     <selector xpath="accommodation"/>
     <field xpath="@id"/>
</key>
<keyref name="AwardWinnerRef" refer="AccKey">
     <selector xpath="accAwards/award"/>
     <field xpath="winner"/>
</keyref>
```

**Abbildung 20 - Beispiel Identifizierungen und Referenzierungen**

## *6.* Vordefinierte einfache Datentypen *§2.3*

XML Schema verfügt insgesamt über 44 vordefinierte einfache Datentypen *(simple predefined datatypes)*. Abbildung 21 zeigt eine hierarchische Gliederung dieser Typen. Eine weitere Übersicht der Datentypen und deren Wertebereiche befindet sich in „XML Schema - Data Types Quick Reference“ [DVIN03B]. Neue einfache Typen können durch Ableiten erzeugt werden. Dabei wird durch Einschränken des Wertebereichs eines bestehenden einfachen vordefinierten Datentyps ein neuer Datentyp erzeugt (Einfache Typen (*simple data type*) werden in X-Rayxs nicht unterstützt, somit auch nicht das Ableiten von vordefinierten einfachen Datentypen).

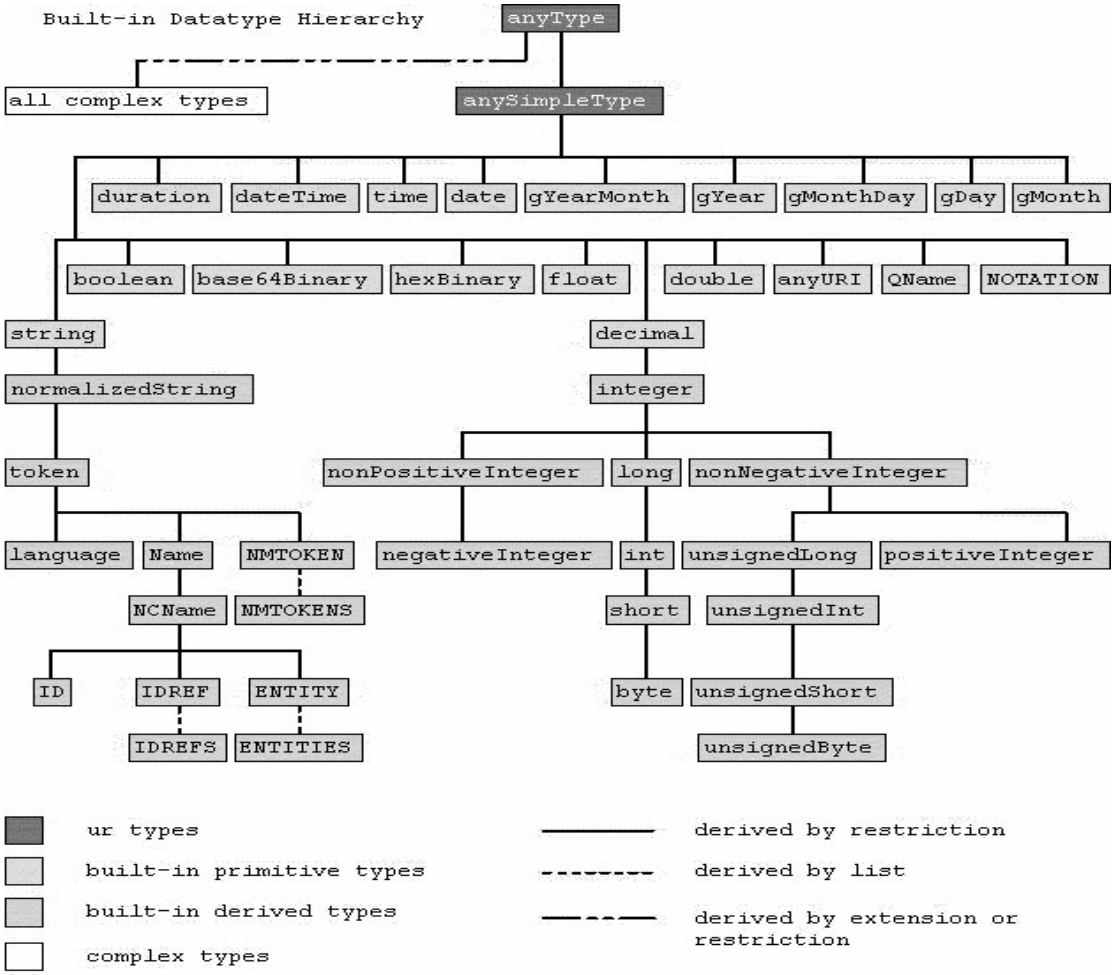



**Abbildung 21 - XML Schema Datentyp Hierarchie [W3C04D]**

## 7. Lokale bzw. globale Elemente, Attribute und Datentypen *§2.2.2 / §3*

In XML Schema besteht die Möglichkeit, Datentypen, Elemente und Attribute global oder lokal zu definieren.

Global bedeutet, dass sie direkt unterhalb des <*schema*>-Elements spezifiziert werden. Diese Elemente, Attribute und Datentypen sind im gesamten Schema sichtbar und lassen sich von beliebigen Stellen aus über ihren Namen referenzieren. Für Elemente, Attribute und Datentypen lokaler Definitionen ist dies nicht gültig. Sie lassen sich ausschließlich an der Position verwendet, an der sie definiert werden. Im Gegensatz zu Datentypen müssen Elemente und Attribute immer Namen gegeben werden, unabhängig davon, ob sie global oder lokal definiert sind. Diese Namen werden im XML-Dokument für die Auszeichnung benutzt. Durch die Namensgebung können Typdefinitionen bzw. Element- und Attribut-Deklarationen mehrfach verwendet werden.

In folgendem Beispiel ist das Element "*name*" global deklariert und kann somit referenziert werden. Das Element „*accommodation*" hingegen wird in einem lokal definierten Datentyp deklariert und referenziert selbst wieder einen global definierten Datentyp „*accommodationType*".

```
<schema targetNamespace="http://www.ifs.jku.at/
      XRay/AccommodationSchema" …>

<element name="name" type="string"/>
<element name="accommodations">
      <complexType>
            <sequence>
                  <element name="accommodation"
                        type="accommodationType" minOccurs="0"
                        maxOccurs="unbounded"/>
                  <element name="accAwards"
                        type="accAwardsType"/>
            </sequence>
      </complexType>
      …
<complexType name="accommodationType">
      <sequence>
            <element ref="name"/>
            <element name="address" type="addressType"/>
      …
```

**Abbildung 22 - Beispiel lokale bzw. globale Elemente, Attribute und Datentypen**

## *8.* Namensräume *§3*

Um die Wiederverwendbarkeit und die Kombinationsmöglichkeiten von Elementen, Attributen und Typdefinitionen von XML-Schemata zu erhöhen, müssen diese über eindeutige Bezeichnungen verfügen, um Verwechslungen zu verhindern. Um dies zu erreichen, gibt es Namensräume (*namespaces*). Nur durch die Vergabe von Namensräumen können die verwendeten Elemente, Attribute und Typen überhaupt identifiziert werden.

Um nicht immer den gesamten Namensraum vor ein Element oder Attribut zu schreiben, lassen sich beliebige Präfixe vergeben, die als abkürzende Schreibweise gedacht sind. Derartige Abkürzungen lassen sich durch den Präfix-Zusatz am <*xmlns*>-Element realisieren.

Mit *„xmlns:xs="http://www.w3.org/2001/XMLSchema"* lässt sich beispielsweise der Namensraum für die XML Schema Schemadefinition durch *xs:* abkürzen.

Der aktuelle Namensraum, für den die Elemente eines Schemas geschrieben werden, kann durch die Verwendung des *<targetNamespace>*-Elements im *<schema>*-Element für alle Elemente gesetzt werden.
Des Weiteren gibt es einen Standardnamensraum, der überall gültig ist, wo sonst keine besondere Kennzeichnung durch ein Namensraumpräfix mit zugeordnetem Namensraum-URI existiert. Der URI (*Unified Ressource Identifier*) für den Standardnamensraum wird durch das Attribut *xmlns="Standard-Namensraum-URI"* gesetzt.
Das Attribut *elementFormDefault* hat ebenso Auswirkungen auf die Namensräume. Es kann einen der beiden Werte *qualified* oder *unqualified* annehmen. Der Wert *unqualified* bewirkt, dass in Instanzen nur globale Elemente durch ein Namensraumpräfix qualifiziert werden dürfen. Lokale Elemente werden implizit über das globale Element qualifiziert, in das sie eingebettet sind. Ist der Wert von *elementFormDefault* dagegen auf *qualified* gesetzt, so müssen alle Elemente in XML-Instanzen qualifiziert werden. Die explizite Angabe des Namensraumpräfixes in jedem einzelnen Element kann aber durch die Deklaration eines Standardnamensraumes umgangen werden [HOLZ04]. X-Rayxs unterstützt für das Element *<elementFormDefault>* nur den Wert qualified. Eine mögliche Namensraumdefinition zeigt das Beispiel in Abbildung 23 .

```
<schema targetNamespace="http://www.ifs.uni-
linz.ac.at/XRay/AccommodationSchema"
elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:ac="http://www.ifs.uni-
lnz.ac.at/XRay/AccommodationSchema"
xmlns:su="http://www.ifs.uni-linz.ac.at/XRay/Supervision">
```

**Abbildung 23 – Beispiel fürNamensräume**

## *9.* Schema Management *§4.1 / §4.5 / §5.1*

Wenn Schemata größer werden, ist es wünschenswert, ihren Inhalt auf mehrere Dateien aufzuteilen. Dadurch wird die Wartung, die Zugriffsregelung und auch die Lesbarkeit des Schemas verbessert.

Um ein XML-Schema in ein anderes, bzw. in dessen Namensraum einzubinden, wird das Element *<include>* verwendet. Dieses Element holt die Deklarationen und Definitionen aus einem externen XML-Schema und macht sie im Namensraum verfügbar. Wenn das Element *<include>* verwendet wird, müssen jedoch beide Schemata denselben Zielnamensraum (*targetNamespace*) haben. Das Element *<redefine>* erfüllt den selben Zweck, jedoch mit dem Unterschied, dass Modifikationen an den eingebundenen Definitionen und Deklarationen vorgenommen werden können (vgl. Abbildung 24). Sollen nun Definitionen oder Deklarationen aus einem XML Schema eingebunden werden, das nicht denselben Zielnamensraum hat, kann das *<import>*-Element verwendet werden (vgl. Abbildung 24).

```
<schema targetNamespace="http://www.ifs.uni-
      linz.ac.at/XRay/AccommodationSchema"
      elementFormDefault="qualified"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:ac="http://www.ifs.uni-
      linz.ac.at/XRay/AccommodationSchema"
      xmlns:su="http://www.ifs.uni-
      linz.ac.at/XRay/Supervision">
<import namespace="http://www.ifs.uni-
      linz.ac.at/XRay/Supervision"
      schemaLocation="xray_supervision.xsd"/>
<redefine schemaLocation="xray_management.xsd">
      <complexType name="managerType">
            <complexContent>
                  <extension base="ac:managerType">
                        <sequence>
                              <element name="salary"
                                    type="positiveInteger"/>
                        </sequence>
                        <attribute name="mgrId"
                              type="positiveInteger"/>
                  </extension>
            </complexContent>
      </complexType>
</redefine>
…
```

**Abbildung 24 - Beispiel Schema Management**

### *4.1.2. Nicht unterstützte XML Schema Konzepte*

**1. Komplexe Datentypen mit gemischtem Inhalt** ***§2.5.2***

Bei komplexen Datentypen mit gemischtem Inhalt ist der Wert des <*mixed*>-Attributes „true“. Somit können Elemente, die mit einem komplexen Datentyp mit gemischtem Inhalt definiert werden, aus weiteren Elementen und einem eigenen einfachen Text bestehen.

**2. Einfache Datentypen** ***§2.3***

Das <*simpleType*>-Element wird verwendet um Ableitungen von einfachen Typen durch Einschränkung (*restriction*), sowie Listentypen und Vereinigungstypen zu definieren.

**3. Schema Dokumentation** ***§2.6***

XML Schema definiert drei Elemente, um Schemata mit Anmerkungen zu versehen, die sowohl für menschliche Leser als auch für Anwendungen gedacht sind. Für Dokumentation, die für Personen gedacht sind, ist das Element <*documentation*> vorgesehen. Für Anwendungen hingegen das Element <*appInfo*>. <*dokumentation*> und <*appInfo*> sind Subelemente vom Element <*annotation*>, welches wiederum am Anfang der meisten Schema-Konstrukte erscheinen darf.

**4. Notation**

Notationen dienen zur Beschreibung des Formats von Nicht-XML-Daten in einem XML-Dokument. Notationen werden in X-Rayxs nicht unterstützt.

**5. Any Konzept** ***§5.5***

Um flexible Dokumente zu erzeugen, gibt es die vordefinierten Typen *anySimpleType* und *anyType*. Sie erlauben für Elemente, die mit diesen Typen deklariert werden, alle einfachen bzw. jeden beliebigen Typ im Instanzdokument anzunehmen.

Des Weiteren stellt XML Schema die Platzhalter *any* und *anyAttribute* zur Verfügung, die verwendet werden können, damit Elemente und Attribute aus dem angegebenen Namensraum in einem Inhaltsmodell auftreten können.

### *6. Element/Attribut Gruppe §2.7 / §2.8*

Gruppen verbessern die Strukturierung und Wiederverwendbarkeit von Elementen. Mit ihnen lassen sich verschiedene Elemente zusammenfassen. Dabei sind die gleichen Mechanismen zu verwenden wie bei komplexen Datentypen. Diese Gruppen können nur global definiert werden und sind somit an beliebiger Stelle im Schema referenzierbar.
Gruppierung kann ebenso für Attribute verwendet werden. Da Attribute keine Strukturen beinhalten, werden in Attributgruppen die Attribute in Form einer Liste angeführt.

## *4.2. Metaschema*

Die Architektur von X-Rayxs ist großteils analog zu jener von X-Ray. Eine Übersicht über die Architektur von X-Ray und die Beschreibung selbiger wurde bereits im Abschnitt **Architektur von X-Ray** des Kapitels **X-Ray** gegeben.

Jedoch verfügt X-Rayxs „nur“ über zwei Hauptkomponenten, nämlich den *Composer/Decomposer* und dem *Metaschema*. Die Adaption des „*Mapping Knowledge Editors*“ ist im Umfang der Diplomarbeit nicht enthalten.

Das Metaschema besteht, wie in Abbildung 25 ersichtlich, ebenso wie jenes von X-Ray aus drei Komponenten:

- *RDB Schema-Komponente* (analog zu *DBSchema* von X-Ray)
- *XML Schema-Komponente* (analog zu *XMLDTD* von X-Ray)
- *Mapping-Komponente* (analog zu *XMLDBSchemaMapping* von X-Ray)

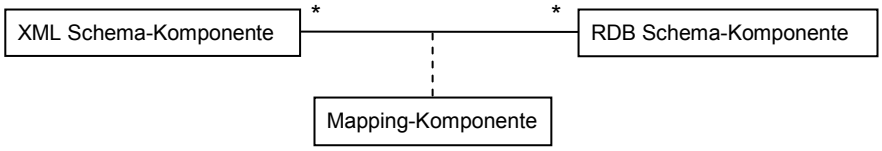



**Abbildung 25 - Komponenten des X-Rayxs Metaschemas**

Diese drei Komponenten werden im Anschluss einzeln genauer beschrieben. Hierfür werden die Relationen und objektorientierten Repräsentationen der Metaschemakomponenten beschrieben, da diese in der Laufzeitphase von X-Rayxs für das Exportieren, Importieren und Abfragen von Daten verwendet werden. Da die Aufgaben des Metaschemas von X-Rayxs analog jener von X-Ray sind, können diese im Detail im Abschnitt **Metaschema im Detail** im Kapitel **X-Ray** nachgelesen werden. Die Zugehörigkeit der Relationen zur jeweiligen Komponente, wird mittels eines Präfixes am Anfang des Namens der Relation ausgedrückt. So gehört ***DB****Schema* zur *RDB Schema-Komponente* und ***XSD****SchemaDeclaration* ist Teil der *XML Schema-Komponente*. Die Relationen der *Mapping-Komponente* hingegen besitzen kein Präfix.

Die Relationen aller drei Komponenten befinden sich in einem RBDS. Am Beginn der Laufzeitphase von X-Rayxs werden alle Daten der drei Metaschema-Komponenten aus dem RDBS in objektorientierte Klassen geladen, die das objektorientierte Metaschema repräsentieren. Dies hat den großen Vorteil, dass nur einmal, nämlich beim Laden des Metaschemas, auf das RDBS für die Akquisition der Metadaten der

Relationen zugegriffen werden muss. Dadurch wird die Performanz einer X-Rayxs-Applikation gesteigert, da sämtliche Metadaten im Arbeitsspeicher vorhanden sind. In den Tabellen, welche die Relationen bzw. Klassen darstellen, sind nur die zur Erklärung notwendigen Attribute enthalten. Die kompletten Relationen bzw. Klassen befinden sich im Anhang.

### *4.2.1. RDB Schema-Komponente*

Diese Komponente hat die Aufgabe, die Metadaten der Relationen des RDBS, in denen Daten von XML-Dokumenten gespeichert sind bzw. gespeichert werden sollen, für die Laufzeitphase im Arbeitsspeicher des Computers, auf dem die X-Rayxs-Applikation läuft, zur Verfügung zu stellen.
Neben den Metadaten der XML Schemata sind jene der Relationen unabdingbar für das Abbildungswissen. Die Metadaten der Relationen stellen einerseits für den Import die Zielinformationen und andererseits für den Export die Quellinformationen für die zu importierenden bzw. zu exportierenden Daten dar.
Die objektorientierten Klassen haben die gleichen Namen und Attribute wie die Relationen, welche die Metadaten zwecks persistenter Speicherung in einem RDBS enthalten. Für die Metadaten werden die folgenden drei Relationen bzw. objektorientierten Klassen benötigt:

- *DBSchema*
- *DBAttribute*
- *DBJoinSegment*

Beim Start einer X-Rayxs-Session (Laufzeitphase) werden die Daten aus diesen drei Relationen tupelweise ausgelesen und dienen als Daten für die Instanzen der jeweiligen Klassen. Eine Instanz einer Klasse speichert genau einem Tupel der jeweiligen Relation. Somit entspricht eine Instanz der Klasse *DBAttribut* genau einem Tupel der Relation *DBAttribute* usw.
Sämtliche Instanzen dieser Klassen werden in drei entsprechenden Containern verwaltet und stehen somit in der Laufzeitphase von X-Rayxs zur Verfügung.
Die Relation *DBRelation*, die bei X-Ray noch Verwendung fand, ist bei X-Rayxs nicht mehr notwendig, da man mittels der Daten in *DBJoinSegment* von einer Basisrelation aus jede weitere Relation, die mit dieser direkt oder indirekt verbunden ist, erreichen kann.

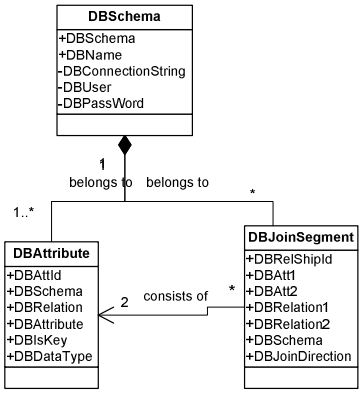



**Abbildung 26 - UML-Diagramm der RDB Schema-Komponente**

In Abbildung 26 wird ein Überblick über die Relationen bzw. Klassen der *RDB Schema-Komponente* und über ihre Attribute gegeben.

Im Folgenden werden die Klassen bzw. Relationen dieser Komponente vorgestellt

#### *4.2.1.1. DBSchema*

Die Instanzen der Klasse *DBSchema* speichern Metadaten über das Datenbank-Schema und die Datenbank, in der die Daten dieses Schemas gespeichert werden. Die Attribute dieser Klasse (bzw. Attribute der Relation *DBSchema*) sind:

- *DBSchema*: Name des Datenbankschemas, der eindeutig eine Instanz dieser Klasse identifiziert
- *DBName*: Name der Datenbank, in welcher sich die mittels *DBAttribute* beschriebenen Relationen befinden
- *DBConnectionString*: Gibt die URL und den Port der Datenbank an
- *DBUser:* Benutzername, der in Verbindung mit dem Passwort den Zugriff auf die Datenbank ermöglicht
- *DBPassword:* Passwort, das in Verbindung mit dem Benutzernamen den Zugriff auf die Datenbank ermöglicht

Die Attribute *DBName, DBConnectionString, DBUser* und *DBPassword* speichern die zum Aufbau einer Datenbankverbindung notwendigen Daten.

In Tabelle 1 wird ein Tupel der Relation *DBSchema*, welches zugleich einer Instanz der Klasse *DBSchema* entspricht, beispielhaft dargestellt. Schlüsselattribute sind, wie bei allen weiteren Relationen auch, unterstrichen dargestellt. Fremdschlüssel sind kursiv gekennzeichnet.

| DBSchema | | | | |
|---|---|---|---|---|
| DBSchema | DBName | DBConnectString | DBUser | DBPassword |
| accomDBSchema | lehre92 | jdbc:oracle:thin:@140.78.90.210:7778: | xray | xray |

**Tabelle 1 - Beispiel eines DBSchema-Tupels**

#### *4.2.1.2. DBAttribute*

Die Instanzen dieser Klasse enthalten die Metadaten der Relationen und ihrer Attribute, in denen Daten aus bzw. von XML-Dokumenten gespeichert sind bzw. gespeichert werden sollen. Es werden alle Attribute der Relationen gespeichert, auf die XML-Elemente und/oder XML-Attribute abgebildet werden können. Sie können als „Datenquellen“ für das Importieren bzw. „Datensenken“ für das Exportieren von XML Daten in Frage kommen.

Die Klasse *DBAttribute* hat die folgenden Attribute:

- *DBAttId*: Eindeutige Nummer, die eine Instanz dieser Klasse eindeutig identifiziert (Primärschlüssel der Relation *DBAttribute*)
- *DBSchema*: Name des DB-Schemas, zu dem dieses Attribut gehört
- *DBRelation*: Name der Relation, zu dem dieses Attribut gehört
- *DBAttribute*: Name des Attributs der Relation
- *DBIsKey*: Speichert, ob es sich bei diesem Attribut um einen Schlüssel (bzw. Teilschlüssel) der *DBRelation* handelt
- *DBDataType*: Speichert den Datentyp dieses Attributs (VARCHAR, etc.)

In Tabelle 2 ist ein Tupel der Relation *DBAttribute*, welches zugleich einer Instanz der Klasse *DBAttribute* entspricht, beispielhaft dargestellt.

| DBAttribute | | | | | |
|---|---|---|---|---|---|
| DBAttId | *DBSchema* | DBRelation | DBAttribute | DBIsKey | DBDataType |
| 1 | accomDBSchema | Accommodation | AccId | true | INT |

**Tabelle 2 - Beispiel eines DBAttribute-Tupels**

### 4.2.1.3. DBJoinSegment

Um die Fremdschlüsselbeziehungen zwischen Relationen zu speichern, werden diese Beziehungsdaten in den Instanzen der Klasse *DBJoinSegment* verwaltet.
Die Attribute dieser Klasse sind:

- *DBRelShipId*: Identifiziert eindeutig eine Instanz der Klasse *DBJoinSegment*
- *DBAtt1*: Name des Attributes von *DBRelation1*
- *DBAtt2*: Name des Attributes von *DBRelation2*
- *DBRelation1*: Name der Relation, zu der die Beziehung von *DBRelation2* geht
- *DBRelation2*: Name der Relation, von der die Beziehung zu *DBRelation1* geht
- *DBSchema*: Name des DB-Schemas, zu dem die Instanz der Klasse *DBJoinSegment* gehört
- *DBJoinDirection*: Gibt die Richtung der Fremdschlüsselbeziehung (abhängig von der Kardinalität) aus Sicht der Basisrelation bzw. der Eltern-Basisrelation an

In Tabelle 3 sind zwei Tupel der Relation *DBAttribute*, welches zugleich einer Instanz der Klasse *DBAttribute* entspricht, beispielhaft dargestellt.

| DBJoinSegment | | | | | | |
|---|---|---|---|---|---|---|
| DBRelShipId | *DBAtt1* | *DBAtt2* | *DBRelation1* | *DBRelation2* | *DBSchema* | DBJoinDirection |
| 1 | Name | VillageName | Village | Accommodation | accomDBSchema | 12 |
| 5 | AccId | AccId | Pool | Accommodation | accomDBSchema | 21 |

**Tabelle 3 - Beispiele für DBJoinSegment-Tupel**

Eine Beziehung zwischen zwei Relationen wird immer aus Richtung der Basisrelation gelesen, wobei *DBRelation2* näher bei der Basisrelation liegt als *DBRelation1*. Abbildung 27 soll dies für das zweite Tupel aus Tabelle 3 veranschaulichen.

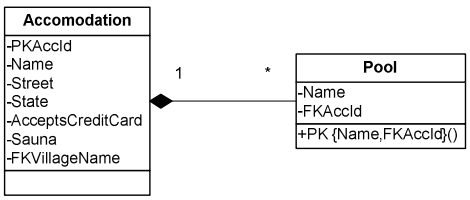



**Abbildung 27 - UML-Beispiel *DBJoinSegment* mit *DBJoinDirection*=21**

In Abbildung 27 stellt *Accommodation* die Basisrelation und zugleich *DBRelation2* dar. Die Relation *Pool* ist die *DBRelation1*.

Die bei X-Ray verwendete Relation *DBJoinPath* wird nicht mehr benötigt, da man mit Hilfe der Informationen in *DBJoinSegment* ohnehin von Relation zu Relation weiternavigieren kann. Das Attribut *DBJoinDirection* gibt die Richtung der Fremdschlüsselbeziehung an.
Im Fall von Abbildung 27 handelt es sich um eine 1 zu * - Beziehung (von der Basisrelation *Accommodation* aus gesehen), weshalb *DBJoinDirection* den Wert 21 enthält. In Abbildung 28 hingegen handelt es sich um eine * zu 1 - Beziehung, weshalb *DBJoinDirection* beim ersten Tupel in Tabelle 3 den Wert 12 hat.

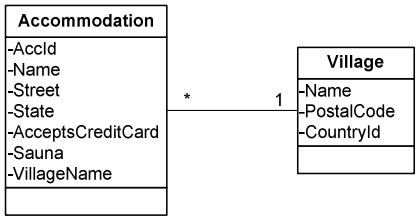



**Abbildung 28 - UML-Beispiel DBJoinSegment mit DBJoinDirection=12**

Im Anhang sind die drei Relationen dieser Komponente mit den Daten des Beispiels *Accommodation.xml*, welches in einem späteren Abschnitt auszugsweise vorgestellt wird, zu finden. Des Weiteren ist ein UML-Diagramm, welches die Relationen zum Speichern der Werte aus dem XML-Dokument *Accommodation.xml* darstellt, im Anhang zu finden.

### *4.2.2. XML Schema-Komponente*

Die Relationen dieser Komponente werden zum Speichern der Schemainformationen der von X-Rayxs verwalteten XML-Schemata verwendet. Dazu werden in der Initalisierungsphase von X-Rayxs die Schemainformationen einer oder mehrerer XML-Schema-Datei(en) in die Relationen dieser Komponente des Metaschemas gespeichert.
Am Beginn der Laufzeitphase werden die Daten aus diesen Relationen in die entsprechenden Klassen für die objektorientierte Repräsentation der Komponente geladen. Dies hat, wie bereits erwähnt den großen Vorteil, dass nur einmal, nämlich beim Laden des Metaschemas, auf das RDBS für das Auslesen der XML Schema-Daten zugegriffen werden muss.

Ausgangspunkte zur Identifizierung der notwendigen objektorientierten Klassen und Relationen waren die umgesetzten Konzepte von XML Schema (siehe Abschnitt **Konzepte von XML Schema** in diesem Kapitel) sowie der XML Schema-Standard [W3C04B].

Für jedes Konzept von XML Schema gibt es eine eigene Klasse sowie eine eigene Relation, um die XML Schema-Informationen zu verwalten und auch persistent zu speichern. Eine Übersicht aller in der *XML Schema-Komponente* verwendeten Relationen und Klassen ist im Anhang ersichtlich. Dort befindet sich auch ein UML-Diagramm, in dem sich alle umgesetzten Konzepte wieder finden lassen. Es stellt die objektorientierten Klassen bzw. Relationen mitsamt den Beziehungen, die zwischen den Klassen bzw. Relationen bestehen, dar.

Alle Relationen dieser Komponente haben das Präfix *XSD* im Namen, z.B. *XSDSchemaDeclaration*. Die Namen der Klassen sind gleich jenen der entsprechenden Relationen, jedoch ohne das Präfix *XSD*. (Klasse *Element* ↔ Relation *XSDElement*). Der Grund hierfür ist, dass die Klassen dieser Komponente ohnehin in einem Paket mit einem spezifischen Namen zusammengefasst sind.

Die objektorientierten Klassen enthalten neben den Attributen der korrespondierenden Relationen zusätzliche Attribute, die Informationen über die Beziehungen der einzelnen Relationen untereinander, die zum Speichern von XML Schema-Elementen dienen, speichern. Mittels Zeigern werden die Beziehungen zwischen den Instanzen der Klassen in der Laufzeitphase von X-Rayxs realisiert. Diese Zeiger werden unter Verwendung der Beziehungsinformationen zu Beginn der Laufzeitphase von X-Rayxs erstellt. Dies ermöglicht ein einfaches und schnelles Navigieren durch ein XML Schema.

Somit beinhaltet jede Klasse bzw. Relation drei Attributarten:

1. Attribute gemäß XML Schema-Standard [W3C04B]; Standardattribute
2. Beziehungsattribute zum Beschreiben der Beziehungen zwischen XML Schema-Elementen
3. Zeiger, basierend auf den Beziehungsattributen

Jede Klasse besitzt zumindest die Standardattribute. Die Beziehungsattribute und Zeiger besitzen nur jene Klassen, die diese auch benötigen.

Viele Klassen bzw. Relationen haben ein Attribut, welches sich aus dem Klassennamen und dem Suffix *Id* zusammensetzt. Dies sind Surrogatschlüssel zum

eindeutigen Identifizieren eines XML Schema-Bestandteiles (*ElementId*, *SchemaDeclarationId*, *SequenceId*…)
Nicht alle Klassen benötigen diesen Surrogatschlüssel, z.B. wenn sich der Schlüssel aus Fremdschlüsseln zusammensetzt (beispielsweise braucht *Import* keinen).

Es werden nun drei Bereiche aus dem UML-Diagramm herausgegriffen, um den Aufbau der *XML Schema-Komponente* zu verdeutlichen.

### *Element*

Anhand von Abbildung 29 wird der Teil der *XML Schema-Komponente*, welcher für das Speichern und Verwalten eines *XSDElements* [W3C04B; §3.3.2 pt1] verwendet wird, eingehender beleuchtet.

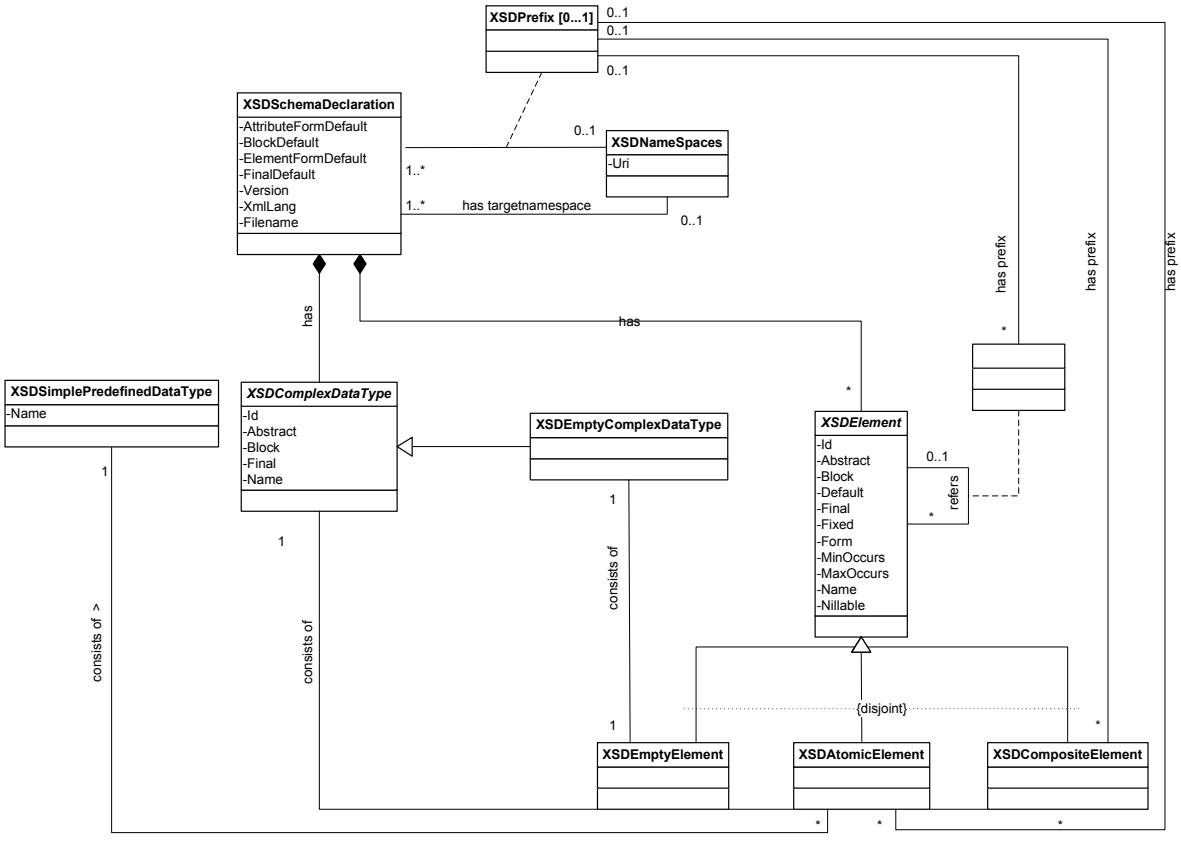



**Abbildung 29 - Ausschnitt XML Schema-Komponente: Element**

In diesem UML-Diagramm sind neben dem *Element* auch dessen wichtigste Beziehungen zu anderen Bestandteilen des XML Schemas dargestellt. Aus Gründen der Übersichtlichkeit ist unter anderem das Schlüsselkonzept (*keyref* etc.) in dieser Abbildung nicht enthalten.

Jedes Tupel der Relation *XSDElement* entspricht genau einer Element-Deklaration (<*element*>) in einem XML-Schema und somit auch genau einer Instanz der Klasse *Element*.
Der Aufbau der objektorientierte Klasse für das Element ist, bezüglich der Attribute, analog der Relation, welche zum persistenten Speichern der Daten des Schema-Elementes dient.

| XSDElement | | | | | | |
|---|---|---|---|---|---|---|
| ElementId | Name | *SchemaDeclarationId* | PrefixId | KindOfElement | KindOfElementId | IsRoot |
| 1 | name | 1 | 1 | SP | 1 | 0 |

**Tabelle 4 - Relation zum Speichern der XSDElement-Daten**

Neben den Standardattributen des Elementes entsprechend [W3C04B] enthält die Klasse bzw. Relation folgende zusätzlich Attribute:

- *ElementId:* Identifiziert eindeutig ein Element (Primärschlüssel)
- *TargetElementId:* Entspricht dem XML Schema - Attribut *ref*
- *SchemaDeclarationId: Id* der Schemadeklaration, der das Element angehört
- *PrefixId:* Schlüssel des Präfixes bei der Typeangabe des Elements
- *KindOfElement:* Art des Elements (ob komplex oder einfacher Datentyp)
- *KindOfElementId:* Identifiziert die Art des Elementes
- *IsRoot:* Kennzeichnet, ob dieses Element ein Wurzelelement ist

Diese Attribute speichern u.a. Informationen über die Beziehungen im UML-Diagramm aus Abbildung 29. Die Zeiger zur Realisierung dieser Beziehungen in der Klasse *Element* sind:

- *SimplePredefinedDT simple*
- *ComplexDataType complex*
- *Element ref*

Das Attribut *SchemaDeclarationId* speichert die Zugehörigkeit eines *XSDElement*s zu einer *XSDSchemaDeclaration*. Dies ist nötig, um jedes Element seinem XML-Schema zuzuordnen bzw. alle Elemente zu identifizieren, die zu einem XML-Schema gehören.

Das Attribut *PrefixId* speichert ein etwaiges Präfix bei der Typangabe des Elementes bzw. bei der Referenzierung eines anderen Elementes (vgl. **Konzept der Namensräume**). Dieses Attribut speichert somit die Information über die Beziehung zwischen *XSDPrefix* und *XSDElement* in Abbildung 30.

```
<element ref="ac:name"/>
<element name="address" type="ac:addressType"/>
```

**Abbildung 30 - Beispiel für *PrefixId***

In Abbildung 30 referenziert das erste Element ein globales Element mit dem Präfix *ac*. Werden mehrere Schemata verwendet, wird durch das Präfix ein globales Element oder ein globaler Datentyp eindeutig identifiziert.

Mit dem Attribut *KindOfElement* wird die Art des Elementes gespeichert. Das Attribut kann nur eine der beiden folgenden Ausprägungen besitzen:

- *SP SimplePredefinedDatatype*
- *CT ComplexDataType*

Dieses Attribut hat den Wert *null*, wenn sich das Element einer Referenz auf ein anderes Element bedient (siehe Attribut *TargetElementId*).
Wie in Abbildung 29 ersichtlich, kann ein *Element* drei Ausprägungen annehmen. Atomare Elemente (*atomic element*) bestehen aus einfachen vordefinierten Datentypen (*SimplePredefinedDatatype*).
Zusammengesetzte Elemente (*composite element*) bestehen aus komplexen Datentypen (*ComplexDataType*). Das Gleiche gilt auch für die leeren Elemente (*empty element*), da sich der *EmptyComplexDataType* vom *ComplexDataType* ableitet.
Für die Referenzierung eines anderen Elementes wird im Attribut *TargetElementId* die *XSDElementId* des referenzierten Elementes gespeichert. Im UML-Diagramm entspricht dieses der Referenz des *XSDElements* auf sich selbst.

Auf diese Art wurden sukzessive die weiteren Klassen und Relationen der *XML Schema-Komponente* gebildet. Es wurden aber auch zusätzliche Klassen und Relationen entwickelt, die nicht direkt aus dem XML Schema-Standard [W3C04B] abgeleitet werden konnten.

Ein Beispiel hierfür ist das *XSDContentParticle*, welches ein Hilfskonstrukt ist, um eine Kombination von XML Schema-Elementen zu realisieren. Mit dieser Klasse kann der verschachtelte Aufbau eines <*sequence*>- oder <*choice*>-Elementes gespeichert werden. Dies trifft allerdings auf <*all*>-Elemente nicht zu, da sich diese nur aus Elementen und nicht aus *ContentParticles* zusammensetzten.

### *ContentParticle, Sequence, SequencePosition*

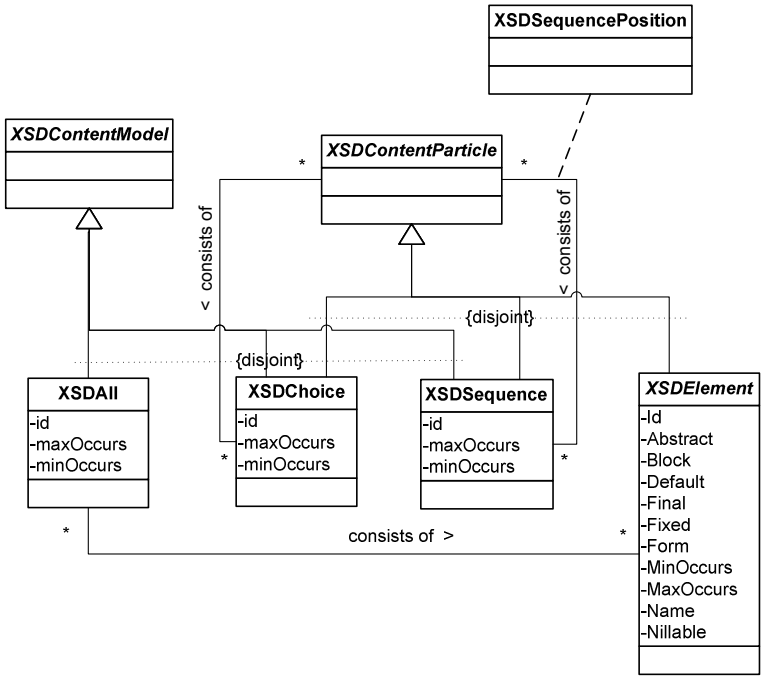



**Abbildung 31 - Ausschnitt XML Schema-Komponente: ContentParticle**

Entsprechend dem XML Schema-Standard kann ein <*sequence*>-Element aus einem oder mehreren Partikeln bestehen, wobei ein Partikel ein <*choice*>-, <*element*>- oder <*sequence*>- Element darstellen kann. Dabei wird für jedes Partikel einer <*sequence*> dessen Position innerhalb der <*sequence*> mitgespeichert. Die Reihenfolge und Beziehungen der Partikel in einer <*sequence*> werden in der Klasse *XSDSequencePosition* gespeichert.

Das Gleiche gilt für <*choice*>-Elemente, wobei hier die Reihenfolge der Partikel nicht vorgegeben ist und daher nicht verfügbar sein muss. Dies ist in Abbildung 31 ersichtlich.

Die Klasse bzw. Relation des *XSDContentParticle,* sowie die der *XSDSequence* und *XSDSequencePosition* sind in Tabelle 5, Tabelle 6 und Tabelle 7 dargestellt:

| XSDSequence | | | |
|---|---|---|---|
| SequenceId | Id | MinOccurs | MaxOccurs |
| *1* | | | |

**Tabelle 5 - Relation und Klasse zum Speichern von XSDSequence-Daten**

In dieser Klasse und Relation werden die Standardattribute einer <*sequence*> gespeichert. Haben die Attribute die Werte *null*, so werden beim Auslesen die Standardwerte für diese Attribute verwendet. In diesem Fall ist entsprechend dem Standard [W3C04B] *MinOccurs*=1und *MaxOccurs*=1.
Die Referenzen auf die Partikel (*ContentParticles*), aus der eine <*sequence*> besteht, werden in der Klasse *SequencePosition* vermerkt. Daher enthält die Klasse *Sequence* einen Container zum Speichern der Zeiger auf die entsprechenden *SequencePositions*.

| XSDSequencePosition | | |
|---|---|---|
| *SequenceId* | *ContentParticleId* | Position |
| 1 | 3 | 1 |
| 1 | 4 | 2 |
| 1 | 5 | 3 |

**Tabelle 6 - Relation und Klasse zum Speichern von XSDSequencePosition-Daten**

Die Klasse *SequencePosition* hat folgende Attributte:

- *SequenceId*: Speichert die *Id* der <*Sequence*>, zu der das *ContentParticle* an dieser Position gehört
- *ContentParticleId*: Speichert die *ContentParticleId* des *ContentParticle*, das an dieser Position der <*sequence*> steht
- *Position*: Speichert die Position, an der sich das *ContentParticle* in der <*sequence*> befindet

Zusätzlich enthält diese Klasse einen Zeiger, der die Beziehung zum entsprechenden *ContentParticle* speichert.

Die Relation *XSDContentParticle* (vgl. Tabelle 7) speichert die Partikel, aus denen eine <*choice*> und/oder <*sequence*> besteht.

| XSDContentParticle | | |
|---|---|---|
| ContentParticleId | KindOfContentParticle | KindOfContentParticleId |
| 3 | E | 3 |
| 4 | E | 4 |
| 5 | E | 5 |

**Tabelle 7 - Relation und Klasse zum Speichern von XSDContentParticle-Daten**

Diese Klasse umfasst folgende Attribute:

- *ContentParticleId*: Identifiziert eindeutig ein *XSDContentParticle*
- *KindOfContentParticle*: Art des Partikels (E=*Element;* S=*Sequence;* C=*Choice)*
- *KindOfContentParticleId*: Identifiziert das Strukturobjekt des *XSDContentParticles*

Im Fall von Tabelle 5 besteht das erste Partikel aus einem *XSDElement* mit der *XSDElementId*=1.

In der objektorientierten Klasse werden zusätzlich folgende Zeiger verwendet:

- *private Element element*
- *private Sequence sequence*
- *private Choice choice*

Auf Basis der Beziehungsdaten wird der entsprechende Zeiger gesetzt.

Gemäß den drei oben angeführten Beispieltabellen (Tabelle 5, Tabelle 6 und Tabelle 7) besteht die *Sequence* mit der *SequenceId*=1 aus den drei Elementen mit den *ElementIds* ={3, 4, 5} an den *Positions* = {1, 2, 3}.

### *ComplexDataType*

Ist ein Element komplexer aufgebaut, kann man zwei Arten komplexer Datentypen verwenden. Globale Datentypen, durch Angabe des Basistyps, sowie lokal definierte komplexe Datentypen. So verwenden z.B. leere Elemente ausschließlich komplexe Datentypen.

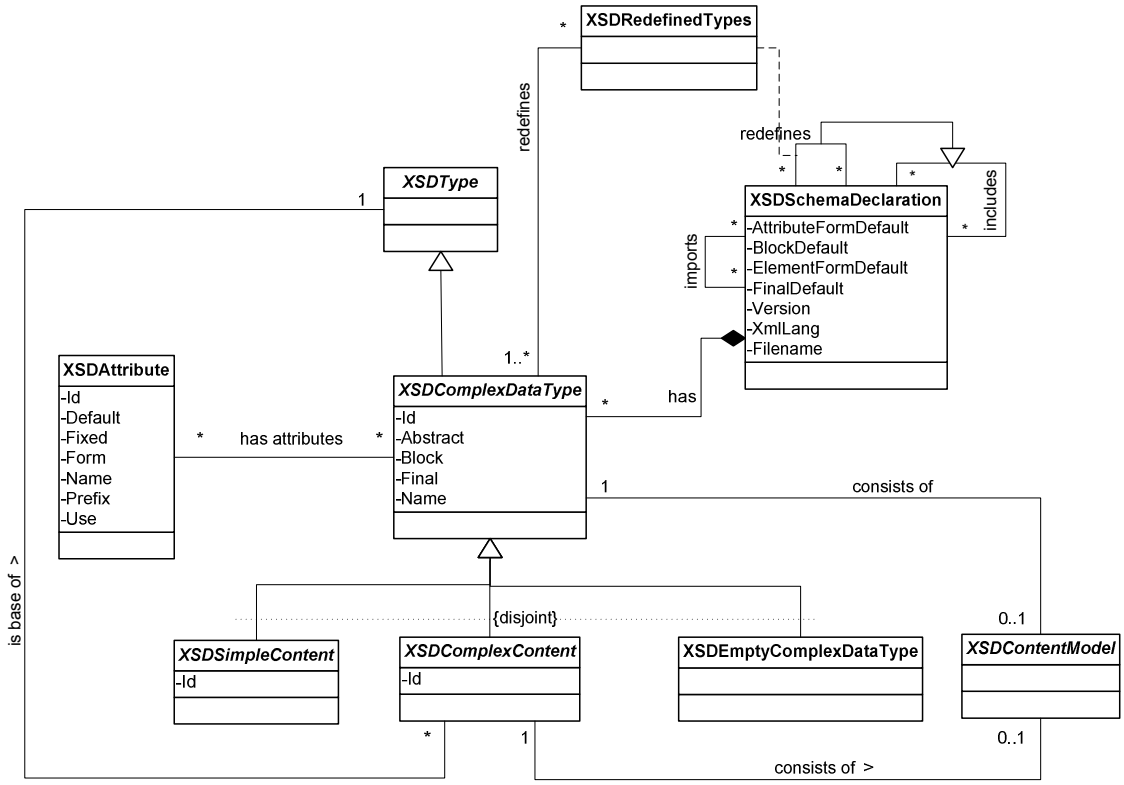



**Abbildung 32 - Ausschnitt XML Schema-Komponente: ComplexDataType**

Abbildung 32 zeigt den Aufbau eines komplexen Datentyps sowie seine Beziehungen zu anderen Schema-Elementen. Der Afbau der objektorientierten Klasse für den komplexen Datentyp ist analog der Relation, welche zum persistenten Speichern der Daten dieses Schema-Elements dient.

| XSDComplexDataType | | | | | | |
|---|---|---|---|---|---|---|
| ComplexDataTypeId | Id | Abstract | Name | *SchemaDeclarationId* | KindOfType | KindOfTypeId |
| 1 | | | villageType | 1 | SC | 1 |

**Tabelle 8 - Relation und Klasse zum Speichern von XSDComplexDataType-Daten**

Neben den Standardattributen entsprechend [W3C04B] enthält die Klasse bzw. Relation folgende zusätzliche Attribute:

- *ComplexDataTypeId:* Eindeutige Nummer eines komplexen Datentyps
- *SchemaDeclarationId: Id* der Schemadeklaration, welcher der Datentyp angehört
- *KindOfType:* Art des komplexen Datentyps
- *KindOfTypeId:* Identifiziert das Strukturobjekt des Datentyps

Diese Attribute speichern u.a. Informationen über die Beziehungen im UML-Diagramm aus Abbildung 32.

Wie aus Abbildung 32 ersichtlich und entsprechen dem XML Schema-Standard [W3C04B], kann ein komplexer Datentyp einer von vier Ausprägungen annehmen. Daher wird im Attribut *KindOfType* gespeichert, um welche Art von Ausprägung es sich bei dem jeweiligen komplexen Datentyp handelt. Diese vier potentiellen Ausprägungen sind:

- *CC ComplexContent*
- *CM ContentModel*
- *EC EmptyComplexDataType*
- *SC SimpleContent*

Die Zeiger zur Realisierung der Beziehungen in dieser Klasse sind folglich:

- *ContentModel content*
- *SimpleContent simplec*
- *ComplexContent complexc*
- *EmptyComplexDataType empty*

Auf Basis der Beziehungsattribute kann der entsprechende Zeiger gesetzt werden. Die anderen Zeiger dieser Klasse enthalten dann den Wert *null*. Zusätzlich können komplexe Datentypen XML Schema-Attribute (<*attribute*>) enthalten. Die Zeiger auf die dem komplexen Datentyp zugehörigen XML Schema-Attribute werden in einem Container (z.B. *LinkedList attributes* in Java) verwaltet. Eine weitere Beziehung besteht zwischen den Klassen *ComplexDataType* und <*redefine*>. Diese Beziehung geht allerdings von Richtung <*redefine*> aus und ist nur existent, wenn ein komplexer Datentyp eines anderen XML-Schemas in dem aktuellen XML-Schema abgeändert bzw. erweitert wird.

Komplexe Datentypen können auf zwei Arten deklariert werden:

1. Lokal innerhalb eines Elementes; Das Attribut *Name* ist leer
2. Global; Das Attribut *Name* enthält den im XML-Schema eindeutigen Namen

Beispiele für diese beiden Deklarationsarten, Attribute und <*redefine*> sind im Abschnitt **Konzepte von XML Schema** dieses Kapitels angeführt.

### *4.2.3. Mapping-Komponente*

Diese dritte Komponente des Metaschemas dient als Brücke zwischen der *XML Schema-Komponente* und der *RDB Schema-Komponente*.
In der Initalisierungsphase werden die Relationen dieser Komponente mit jenen Daten gefüllt, die notwendig sind, um das Schema eines XML-Dokumentes auf das einer Datenbank und vice versa abzubilden. Erst dadurch ist es möglich, in der Laufzeitphase Daten aus einem XML-Dokument in die Relationen einer oder mehrerer Datenbank(en) zu importieren und zu exportieren. Des Weiteren ist es auch möglich, Abfragen mittels XQuery auf gespeicherte XML Schemata abzusetzen.
Wird ein XML-Schema auf Relationen einer Datenbank abgebildet, so spricht man von einer Abbildungsvariante. Es ist jedoch möglich, für ein XML-Schema beliebig viele Abbildungsvarianten in dieser Komponente zu speichern.
Einflussfaktoren für verschiedene Abbildungsvarianten:

- Mehrere Wurzelelemente: Von jedem Wurzelelement eines XML Schemas ausgehend, kann es eine eigene Abbildungsvariante geben
- Mehrere Datenbanken: Ein XML-Schema kann auf die Schemata verschiedener Datenbanken abgebildet werden
- Mehrere Datenbankschemata: Eine Datenbank kann mehrere Schemata enthalten. Ein XML Schema könnte auf jedes dieser Schemata abgebildet werden

Die verschiedenen Abbildungsvarianten sind aber nur möglich, solange bestimmte Grundregeln und Einschränkungen beim Abbilden beachtet und eingehalten werden. Diese werden im folgenden Abschnitt betrachtet.

#### 4.2.3.1. Sinnvolle Abbildungsstrategien

Grundsätzlich sind diese analog zu den Abbildungsstrategien, die im Abschnitt **Sinnvolle Abbildungsstrategien** des Kapitels **X-Ray** dargestellt wurden.
Bei X-Rayxs finden daher diese sechs, schon bekannten, Abbildungsarten die analoge Anwendung. Es wurden jedoch zwei Abbildungsarten hinzugefügt, die es aber auch bei X-Ray grundsätzlich schon gab. Allerdings nur in einer impliziten Form. Wurde ein Element bei X-Ray nicht abgebildet, z.B. weil es sich um das Wurzelelement handelt, wurde dies im Abbildungsschema mittels des Wertes ***null*** dargestellt.
Bei X-Rayxs hingegen wird dies mit ET_0 für nicht abzubildende Elemente und analog A_0 bei nicht abzubildenden Attributen symbolisiert. Es gibt somit bei X-Rayxs insgesamt acht Abbildungsarten:

- ET_0
- $ET\_R_{direkt/indirekt}$
- $ET\_A_{direkt/indirekt}$
- $A\_A_{direkt/indirekt}$
- A_0

Es müssen, wie bei X-Ray, die Eigenschaften der Elemente sowie jene der Attribute bei der Wahl der Abbildungsart berücksichtigt werden.

Alle Informationen, die zum Abbilden von Elementen und Attributen notwendig sind, werden in fünf Relationen persistent gespeichert. Wie bei den beiden vorhergehenden Komponenten werden am Beginn der Laufzeitphase von X-Rayxs die Daten aus den Relationen geladen. Bei dieser Komponente werden die Daten in die Instanzen der folgenden fünf objektorientierten Klassen geladen:

- *Mappings*
- *ElementMapping*
- *AttributeMapping*
- *ElementBaseRelation*
- *AbstractTypeKeys*

Diese Klassen bzw. Relationen werden nun im Detail näher erklärt.

### *Mappings*

Es kann, wie schon erwähnt, für ein XML Schema mehrere Abbildungsvarianten geben. Mittels dieser Relation werden die Metadaten der einzelnen Abbildungsvarianten gespeichert. Die korrespondierende objektorientierte Klasse ist analog zur Relation aufgebaut. Es wird gespeichert, welches XML-Schema mit welchem Wurzelelement auf welches Datenbankschema abgebildet wird. Tabelle 9 stellt ein Tupel zum Speichern einer Abbildungsvariante dar.

| Mappings | | | | |
|---|---|---|---|---|
| MappingId | SchemaDeclarationId | RootElementId | MappingVariant | DBSchema |
| 1 | 1 | 33 | Accommodation1 | accomDBSchema |

**Tabelle 9 - Relation und Klasse zum Speichern der Abbildungsvariante**

- *MappingId*: Identifiziert eindeutig eine Abbildungsvariante
- *SchemaDeclarationId*: Identifiziert das abzubildende XML Schema
- *RootElementId*: Identifiziert das Wurzelelement des abzubildenden XML Schemas
- *MappingVariant*: Frei wählbare Bezeichnung für eine Abbildungsvariante
- *DBSchema*: Identifiziert das Datenbankschema, auf welches das XML Schema abgebildet wird

### *ElementMapping*

Diese Relation ist eines der Kernstücke der Mapping-Komponente. Mit Hilfe dieser Relation werden die Abbildungsstrategien für jedes Element eines XML-Dokumentes einzeln und detailliert gespeichert. Erst die Verwendung dieser Relation bzw. der Instanzen der entsprechenden Klasse ermöglicht es, Werte aus einem XML-Dokument in die entsprechenden Attribute von Relationen eines RDBS zu speichern und auch wieder in ein XML-Dokument zu exportieren. Es wird gespeichert, welches Element mittels welcher Abbildungsstrategie auf welches Attribut bzw. welche Relation abgebildet wird.

Tabelle 10 stellt mehrere Tupel der Relation bzw. mehrere Instanzen dieser Klasse dar, in welchen alle diese Informationen enthalten sind.

| ElementMapping | | | | |
|---|---|---|---|---|
| XSDElementId | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 33 | ET_0 | NULL | NULL | 1 |
| 28 | ET_R | NULL | NULL | 1 |
| 23 | ET_R | NULL | 6 | 1 |
| 3 | ET_A1 | 3 | NULL | 1 |
| 4 | ET_A2 | 28 | 1 | 1 |

**Tabelle 10 - Relation und Klasse zum Speichern der Abbildungsdaten eines Elementes**

Die Klasse *ElementMapping* hat folgende Attribute:

- *XSDElementId*: Identifiziert eindeutig ein Element (vgl. Tabelle 4)
- *KindOfMapping*: Enthält die Abbildungsstrategie
- *DBAttr*: Identifiziert eindeutig ein Attribut einer Relation des DB Schemas (vgl. Tabelle 2)
- *DBRelShipId*: Identifiziert eindeutig eine Instanz der Klasse *DBJoinSegment* bzw. ein Tupel der Relation (vgl. Tabelle 3)
- *MappingId*: Identifiziert eindeutig eine Abbildungsvariante (vgl. Tabelle 9)

*DBAttr* und *XSDElementId* stellen die Quelle bzw. das Ziel der Daten beim Abbilden dar. *XSDElementId* bildet zusammen mit der *MappingId* den Schlüssel dieser Relation. D.h. ein XML Schema-Element wird für genau eine Abbildungsvariante genau einmal auf ein RDBS abgebildet. Für jede Abbildungsvariante muss für jedes abzubildende Element ein neues Tupel bzw. eine neue Instanz der Klasse angelegt werden.

Dem Attribute *DBRelShipId* fällt zusätzlich eine spezielle Funktion bei der Abbildungsart ET_R zu. Es dient zur Unterscheidung, ob es sich beim Abbilden um $ET\_R_{direkt}$ oder $ET\_R_{indirekt}$ handelt. Enthält bei ET_R das Attribut *DBRelShipId* den Wert *null*, so wird das Element auf die Basisrelation des DB Schemas abgebildet ($ET\_R_{direkt}$). Das zweite Tupel in Tabelle 10 stellt solch einen Fall dar. Ansonsten findet ein indirektes Abbilden statt ($ET\_R_{indirekt}$).
Das Attribut *DBRelShipId* speichert für die indirekten Abbildungsarten ($ET\_R_{indirekt}$, $ET\_A_{indirekt}$) den entsprechenden Verweis auf die Verbindungsdaten (vgl. Abschnitt **Relationale Komponente – DBJoinSegment**). Das dritte Tupel in Tabelle 10 ist ein

Beispiel dafür. Die beiden letzten Tupel von Tabelle 10 zeigen jeweils ein Beispiel für ET_$A_{direkt}$ und ET_$A_{indirekt}$. Daher ist im Attribut *DBAttr* jeweils ein Wert eingetragen.

### *AttributeMapping*

Diese Relation bzw. Klasse ist großteils analog zu *ElementMapping* aufgebaut. In XML Schema sind <*attribute*>-Deklarationen (auch wenn teilweise global definiert), letztendlich immer einem komplexen Datentyp zugeordnet. Verschiedene Elemente können jedoch den gleichen komplexen Datentyp als Basis haben (um diesen z.B. zu erweitern oder zu beschränken). Es werden allerdings nur Elemente und nicht die komplexen Datentypen auf ein DB-Schema abgebildet. Es ist daher notwendig, zu speichern, welches <*attribute*> welchem <*element*> in einem XML-Dokument zugehörig ist. Um diese Information zu speichern, wurde das Attribut *XSDElementId* der Relation bzw. Klasse *AttributeMapping*, wie in Tabelle 11 dargestellt, hinzugefügt. Wie bereits erwähnt, kommt auch hier eine implizite Abbildungsart von X-Ray explizit hinzu. Wird ein <*attribute*> nicht abgebildet, ist im Attribut *KindOfMapping* A_0 eingetragen.

| AttributeMapping | | | | | |
|---|---|---|---|---|---|
| XSDElementId | XSDAttributeId | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 28 | 8 | A_0 | NULL | NULL | 1 |
| 28 | 7 | A_A1 | 1 | NULL | 1 |
| 6 | 2 | A_A2 | 29 | 1 | 1 |

**Tabelle 11 - Relation und Klasse zum Speichern der Abbildungsdaten eines Attributes**

Die Klasse *AttributeMapping* enthält folgende Attribute:

- *XSDElementId*: Identifiziert eindeutig ein Element (vgl. Tabelle 4)
- *XSDAttributetId*: Identifiziert eindeutig ein Attribut im XML-Schema
- *KindOfMapping*: Enthält die Abbildungsstrategie
- *DBAttr*: Identifiziert eindeutig ein Attribut einer Relation des DB-Schemas (vgl. Tabelle 2)
- *DBRelShipId*: Identifiziert eindeutig eine Instanz der Klasse *DBJoinSegment* bzw. ein Tupel der Relation (vgl. Tabelle 3)
- *MappingId*: Identifiziert eindeutig eine Abbildungsvariante (vgl. Tabelle 9)

In der Relation bzw. der Klasse *AttributeMapping* sind die Attribute *XSDElementID*, *XSDAttributeID* und *MappingId* gemeinsam Schlüssel zum Identifizieren eines Tupels bzw. einer Instanz der Klasse.
Die letzten beiden Tupel in Tabelle 11 sind Beispiele für die Abbildungsarten A_$A_{direkt}$ und A_$A_{indirekt}$.

***ElementBaseRelation***

Diese Relation bzw. Klasse hat zur Aufgabe, die notwendigen Daten zu speichern, um vom ersten Element unterhalb des Wurzelelementes, welches mit ET_R abgebildet wird, die erste Basisrelation eines DB-Schemas zu erreichen. Kommt nach dem Wurzelelement ein Element, das mittels ET_A1 abgebildet wird, ist die Information bzgl. der Basisrelation implizit durch den Wert in *DBAttr* verfügbar.
Ein Element, bestehend aus einem komplexen Datentyp, kann jedoch nur mit ET_R abgebildet werden. *DBAttr* ist in diesem Fall immer null, woraufhin nun die Information über die Basisrelation in die Relation bzw. Klasse *ElementBaseRelation* ausgelagert wird.
Das zweite Tupel in Tabelle 10 stellt diesen Fall dar. Es sind deshalb die Daten, wie in Tabelle 12 aufgelistet, notwendig, um die erste Basisrelation des DB-Schemas zu erreichen.

| ElementBaseRelation | | |
|---|---|---|
| MappingId | ElementId | DBRelation |
| 1 | 28 | Accommodation |

**Tabelle 12 - Relation und Klasse zum Erreichen der Basisrelation**

Die Klasse *ElementBaseRelation* hat die folgenden Attribute:

- *MappingId*: Identifiziert eindeutig eine Abbildungsvariante (vgl. Tabelle 9)
- *ElementId*: Identifiziert eindeutig ein Element (vgl. *XSDElementId* in Tabelle 4)
- *DBRelation*: Name der Basisrelation (vgl. Tabelle 2)

*MappingId* und *ElementId* bilden zusammen den Schlüssel für diese Relation, da mehrere Elemente einer Abbildungsvariante verschiedene Basisrelationen haben können. Diese Relation bzw. Klasse findet keine Verwendung, wenn sämtliche Elemente eines XML Schemas mittels ET_$A_{direkt}$ (ET_A1) abgebildet werden.

### *AbstractTypeKeys*

Mit XML Schema ist es möglich, durch Verwendung abstrakter, komplexer Datentypen, Vererbungen zu realisieren (siehe Abschnitt **Konzepte von XML Schema**). Verwendet ein Element bei seiner Deklaration als Basis einen abstrakten komplexen Datentyp, wird erst in der Instanz des Schemas - also im XML-Dokument – festgelegt, welcher Vererbungstyp bei dem Element Anwendung findet. Dies wird mittels des XML Schema – spezifischen Attributes *xsi:type*, wie in Abbildung 33 dargestellt, realisiert.

```
<room xsi:type='standardRoomType'>
      <roomNumber>101</roomNumber>
      <size>25</size>
      <nrOfBeds>2</nrOfBeds>
      <tv>Panasonic TX-32PS11D</tv>
      <shower/>
</room>

<room xsi:type='luxuryRoomType' hasButler='true'>
      <roomNumber>305</roomNumber>
      <size>95</size>
      <nrOfBeds>3</nrOfBeds>
      <tv>Sony KV-32FQ85</tv>
</room>
```

**Abbildung 33 - Beispiele für Vererbungen in XML Schema**

Man kann daher erst beim Importieren eines XML-Dokumentes während der Laufzeitphase feststellen, welchen Vererbungstyp ein Element als Basis hat. In der Relation bzw. den Instanzen der Klasse *AbstractTypeKeys* wird für jedes Element eines XML-Dokumentes gespeichert, um welchen Vererbungstyp es sich bei der Instanz des Elementes handelt.

| AbstractTypeKeys | | | |
|---|---|---|---|
| DBRelation | DBKeyAttr | DBKeyValue | HeritageType |
| Room | RoomId | 1 | standardRoomType |
| Room | RoomId | 2 | luxuryRoomType |
| Room | RoomId | 3 | standardRoomType |

**Tabelle 13 - Relation und Klasse zum Speichern von Vererbungsdaten**

Die Klasse *AbstractTypeKeys* besteht aus folgenden Attributen:

- *DBRelation*: Name der Basisrelation (vgl. Tabelle 2)
- *DBKeyAttr*: Attribut der *DBRelation* (vgl. *DBAttribute* in Tabelle 9)
- *DBKeyValue*: Schlüsselwert des *DBKeyAttr*
- *HeritageType:* Speichert den Vererbungstyp als Wert

Ein kleines Beispiel soll dies verdeutlichen:
Der erste Raum (*roomNumber*=101) in Tabelle 13 hat beim Import in die Relation *Room* die *RoomId*=1 (Surrogat) zugewiesen bekommen. Danach wird in der Relation *AbstractTypeKeys* gespeichert, dass die *RoomId*=1 dem *xsi:type* - Attribut entsprechend den Wert *standardRoomType* hat.

Dies muss beim Import eines XML-Dokumentes für jedes Element durchgeführt werden, welches das XML Schema-spezifische Attribut *xsi:type* besitzt. Deshalb muss nach jedem Import diese Relation erneut in Instanzen der korrespondierenden objektorientierten Klasse ausgelesen werden.

## *4.3. Metaschema Beispiele*

Bisher wurden die drei Komponenten des Metaschemas im Einzelnen betrachtet. Das Metaschema erhält seine Funktionalität jedoch durch das Zusammenspiel der Komponenten miteinander. Daher soll in diesem Abschnitt mit Hilfe von kleinen Beispielen das Zusammenwirken der einzelnen Komponenten des Metaschemas hervorgehoben werden.

Die Daten für die Beispiele liefert das im Anhang befindliche XML-Dokument samt dem zugehörigen Schema (*accommodation.xml* und *xray_accommodation.xsd*). In diesem XML-Dokument werden mehrere Unterkünfte verwaltet. Jedes der gespeicherten Hotels hat eine Vielzahl von Daten. Dazu gehört u.a. der Name des Hotels, Telefonnummer, Anzahl der Pools, Ausstattung der einzelnen Zimmer, Bewertung des Hotels etc. Der exakte Aufbau sowie die genauen Daten sind im Anhang nachlesbar.

Der Aufbau des XML-Dokumentes ist so gewählt, dass seine zugehörige Schemadatei fast sämtliche im Abschnitt **Konzepte von XML Schema – Unterstütze Konzepte** aufgelisteten Konzepte enthält. Es ist somit in der Schemadatei von einfachen vordefinierten Datentypen über komplexe Datentypen, dem Schlüsselkonzept und der Verwendung von Namensräumen bis hin zur Vererbung komplexer Datentypen beinahe jedes im Abschnitt **Konzepte von XML Schema – Unterstützte Konzepte** vorgestellte Konzept vorhanden.

In der Initialisierungsphase werden die Daten aus der Schemadatei in die entsprechenden Relationen der *XML Schema – Komponente* eingetragen. Ebenso wird mit den Metadaten der Relationen eines korrespondierenden RDBS verfahren. Diese Metadaten werden in die entsprechenden Relationen der *RDB Schema-Komponente* eingefügt. Die Abbildungsdaten für die Mapping-Komponente werden unter Berücksichtigung der Abbildungsregeln aus den Abschnitten **Mapping-Komponente - Sinnvolle Abbildungsstrategien** und **X-Ray - Sinnvolle Abbildungsstrategien** aus dem Schema des XML-Dokumentes abgeleitet. Eine Übersicht der befüllten und für das XML-Dokument verwendeten Relationen aller drei Metaschemakomponenten kann im Anhang gefunden werden.

Da es den Rahmen dieses Abschnittes sprengen würde, das komplette Metaschema für das im Anhang befindliche XML-Dokument detailliert zu beschrieben, werden vier Beispiele herausgegriffen. Ein Beispiel für Vererbung komplexer Datentypen wurde schon bei der Präsentation der Relation *AbstractTypeKeys* der *Mapping-Komponente* gegeben.

### *4.3.1. Beispiel 1: ET_R$_{direkt}$, ET_A$_{direkt}$ und ET_A$_{indirekt}$*

Der nachfolgende Ausschnitt des XML-Dokumentes und des zugehörigen Teils der Schemadatei stellt die Konstellation der Elemente <*accommodation*> und <*name*>, sowie des Attributs <*id*> im Kontext dar. Mit Hilfe dieser Elemente sollen die Abbildungsarten ET_R und ET_A demonstriert werden.

```
<accommodations ...>
  <accommodation id="45" state="Austria">
    <name>Steigenberger MAXX Hotel Linz</name>
    ....
  </accommodation>
  <accommodation id="46" state="Austria">
    <name>Hotel Wolf-Dietrich</name>
    ...
  </accommodation>
  <accAwards>
    ...
  </accAwards>
</accommodations>
```

**Abbildung 34 - Ausschnitt *Accommodation.xml* Beispiel 1**

```
<schema...>
  <element name="name" type="string"/>
  <element name="accommodations">
       <complexType>
         <sequence>
           <element name="accommodation" type="ac:accommodationType"
             minOccurs="0" maxOccurs="unbounded"/>
           <element name="accAwards" type="ac:accAwardsType"/>
         </sequence>
       </complexType>
       ...
  </element>
  <!-- Start of complex types-->
  <complexType name="accommodationType">
    <sequence>
      <element ref="ac:name"/>
      ...
    </sequence>
    <attribute name="id" type="positiveInteger" use="required"/>
    <attribute name="state" type="string" fixed="Austria"/>
  </complexType>
</schema>
```

**Abbildung 35 - Ausschnitt xray_accommodation.xsd Beispiel 1**

Aus dem Ausschnitt des XML-Dokumentes in Abbildung 34 ergeben sich für die beiden Elemente und das Attribut folgende Abbildungsstrategien:

- <*accommodation*>: ET_$R_{direkt}$, da es das erste komplexe Element nach dem nicht abzubildenden Wurzelelement <*accommodations*> ist.
- <*name*>: ET_$A_{direkt}$, da es Teil von <*accommodation*> ist und Kardinalität „1“ hat, da *MinOccurs* und *MaxOccurs* nicht angegeben sind
- <*id*>: A_$A_{direkt}$, da es nur einmal im Element <*accommodation*> vorkommen kann und auch keine Normalisierung nötig ist

In den nachfolgenden Tabellen wurden zwecks Übersichtlichkeit einige für Demonstrationszwecke irrelevante Attribute weggelassen. Die Tabellen sollen veranschaulichen, wie die Schemastruktur gespeichert wird. Das Element mit der Referenz *(<element ref="ac:name"/>)* auf das Element <*name*> ist Teil der <*sequence*>*,* aus welcher ein <*complexType*> besteht. Das Element <*accommodation*> wiederum hat als Basis diesen <*complexType*> (*accommodationType*).

| XSDElement | | | | | | | |
|---|---|---|---|---|---|---|---|
| ElementId | MinOccurs | MaxOccurs | Name | *Target ElementId* | *Schema DeclarationId* | KindOf Element | KindOf ElementId |
| 1 | | | name | | 1 | SP | 1 |
| 2 | | | | 1 | 1 | | |
| 28 | 0 | unbounded | accommodation | | 1 | CT | 13 |
| 33 | | | accommodations | | 1 | CT | 16 |

**Tabelle 14 - Relation für Element Beispiel 1**

| XSDComplexDataType | | | | |
|---|---|---|---|---|
| ComplexDataTypeId | Name | *SchemaDeclarationId* | KindOfType | KindOfTypeId |
| 13 | accommodationType | 1 | CM | 7 |
| 16 | | 1 | CM | 10 |

**Tabelle 15 - Relation für ComplexType Beispiel 1**

| XSDContentModel | | |
|---|---|---|
| ContentModelId | KindOfContentModel | KindOfContentModelId |
| 7 | S | 6 |
| 10 | S | 9 |

**Tabelle 16 - Relation für ContentModel Beispiel 1**

| XSDSequence | | | |
|---|---|---|---|
| SequenceId | Id | MinOccurs | MaxOccurs |
| *6* | | | |
| *9* | | | |

**Tabelle 17 - Relation für Sequence Beispiel 1**

| XSDSequencePosition | | |
|---|---|---|
| *SequenceId* | *ContentParticleId* | Position |
| 6 | 2 | 1 |
| 9 | 34 | 1 |

**Tabelle 18 - Relation für SequencePosition Beispiel 1**

| XSDContentParticle | | |
|---|---|---|
| ContentParticleId | KindOfContentParticle | KindOfContentParticleId |
| 2 | E | 2 |
| 34 | E | 28 |

**Tabelle 19 - Relation für ContentParticle Beispiel 1**

Die Datenbank enthält die Relation *Accommodation* zum Speichern <*accommodation*>-spezifischer Daten.

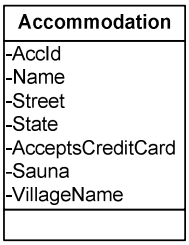



**Abbildung 36 - UML-Diagramm der Relation Accommodation in RDBS**

Die Metadaten zum Beschreiben der Relation *Accommodation* sind in der *RDB Schema-Komponente* entsprechend gespeichert.

| DBSchema | | | | |
|---|---|---|---|---|
| DBSchema | DBName | DBConnectString | DBUser | DBPassword |
| accomDBSchema | lehre92 | jdbc:oracle:thin:@140.78.90.210:7778: | xray | xray |

**Tabelle 20 - Relation DBSchema von Beispiel 1**

| DBAttribute | | | | | |
|---|---|---|---|---|---|
| DBAttId | *DBSchema* | DBRelation | DBAttribute | DBIsKey | DBDataType |
| 1 | accomDBSchema | Accommodation | AccId | true | INT |
| 2 | accomDBSchema | Accommodation | Name | false | VARCHAR(50) |
| 3 | accomDBSchema | Accommodation | Street | false | VARCHAR(50) |
| 4 | accomDBSchema | Accommodation | State | false | VARCHAR(50) |
| 5 | accomDBSchema | Accommodation | VillageName | false | VARCHAR(50) |
| 6 | accomDBSchema | Accommodation | AcceptsCreditCards | false | BOOLEAN |
| 7 | accomDBSchema | Accommodation | Sauna | false | BOOLEAN |

**Tabelle 21 - Relation DBAttribute von Beispiel 1**

Die für das Demonstrationsbeispiel relevanten Werte aus Abbildung 34 sind in der Relation *Accommodation* gespeichert.

| Accommodation | | | | | | |
|---|---|---|---|---|---|---|
| AccId | Name | Street | State | *VillageName* | Accepts CreditCards | Sauna |
| 45 | Steigenberger MAXX Hotel Linz | Am Winterhafen 13 | Austria | Linz | true | false |
| 46 | Hotel Wolf-Dietrich | Wolf-Dietrich-Straße 7 | Austria | Salzburg | true | true |

**Tabelle 22 - Relation Accommodation von Beispiel 1**

Nachdem feststeht, welches XML-Element bzw. XML-Attribut auf welches Attribut welcher Relation in der Datenbank unter Verwendung welcher Abbildungsstrategien abgebildet werden soll, können nun auch die Relationen der *Mapping-Komponente* gefüllt werden.

| Mappings | | | | |
|---|---|---|---|---|
| MappingId | SchemaDeclarationId | RootElementId | MappingVariant | DBSchema |
| 1 | 1 | 33 | Accommodation1 | accomDBSchema |

**Tabelle 23 - Relation Mappings von Beispiel 1**

| ElementMapping | | | | |
|---|---|---|---|---|
| XSDElementID | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 33 | ET_0 | NULL | NULL | 1 |
| 28 | ET_R | NULL | NULL | 1 |
| 2 | ET_A1 | 2 | NULL | 1 |

**Tabelle 24 - Relation ElementMapping von Beispiel 1**

| AttributeMapping | | | | | |
|---|---|---|---|---|---|
| XSDElementID | XSDAttributeID | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 28 | 7 | A_A1 | 1 | NULL | 1 |

**Tabelle 25 - Relation AttributeMapping Beispiel 1**

Da das Element <*accommodation*> mittels ET_$R_{direkt}$ abgebildet wird, ist auch ein Eintrag in der Relation *ElementBaseRelation* notwendig.

| ElementBaseRelation | | |
|---|---|---|
| MappingId | ElementId | DBRelation |
| 1 | 28 | Accommodation |

**Tabelle 26 - Relation ElementBaseRelation Beispiel 1**

Beim Durchlaufen der in der *XML Schema – Komponente* gespeicherten XML-Schemastruktur erreicht man, ausgehend vom Wurzelelement, jedes sich in dessen Hierarchie befindliche Element und Attribut.

Wie in Tabelle 24 und Tabelle 26 ersichtlich, wird das XML-Element <*accommodation*> auf die Relation *Accommodation* und das Subelement <*name*>, auf deren Attribut *Name* derselben Relation abgebildet, weil ET_$A_{direkt}$ bzw. ET_A1 vorliegt. Der Wert des Attributs <*id*>, welches Bestandteil von <*accommodation*> ist, , wird, da A_$A_{direkt}$ bzw. A_A1 Anwendung findet, im Attribut *AccId* der Relation *Accommodation* gespeichert.

Um die korrekten Werte für die aktuelle Instanz eines XML-Elementes beim Durchlauf der Schemastruktur zu erhalten, muss das richtige Tupel selektiert werden. Aus der Relation *DBAttribute* ist bekannt, welches Attribut einer Relation den Primärschlüssel darstellt. Im Fall von Beispiel 1 ist es das Attribut *AccId* der Relation *Accommodation*. Man muss sich für das Auslesen weiterer Werte aus Attributen der Relation *Accommodation* den jeweils aktuellen Wert des Schlüssels merken. Dies wird z.B. beim Prototyp der X-Rayxs-Applikation [ORT05] im Programmcode realisiert.

### *4.3.2. Beispiel 2: ET_A$_{indirekt}$*

In diesem Beispiel soll das indirekte Abbilden auf Relationen und Attribute näher erläutert werden. Abbildung 37 und Abbildung 38 zeigen das XML-Element <*email*>, welches als Subelement von <*accommodation*> eine oder mehrere Email - Adressen für jeweils eine Unterkunft speichert.

```
<accommodations ...>
  <accommodation id="45" state="Austria">
       …
       <email>service.linz@maxxhotel.at</email>
       …
  </accommodation>
  <accommodation id="46" state="Austria">
       …
       <email>office@salzburg-hotel.at</email>
       …
  </accommodation>
</accommodations>
```

**Abbildung 37 - Ausschnitt Accommodation.xml Beispiel 2**

```
<complexType name="accommodationType">
     <sequence>
       <element ref="ac:name"/>
       …
       <element name="email" type="string"
minOccurs="0" maxOccurs="unbounded"/>
       …
     </sequence>
     …
</complexType>
```

**Abbildung 38 - Ausschnitt *xray_accommodation.xsd* Beispiel 2**

Die entsprechenden Relationen zum Speichern der Struktur des XML-Schemas werden aus Gründen der Übersichtlichkeit bei diesem Beispiel weggelassen. Die Relationen sind analog jener aus Beispiel 1 gefüllt.

| XSDElement | | | | | | |
|---|---|---|---|---|---|---|
| ElementId | MinOccurs | MaxOccurs | Name | *SchemaDeclarationId* | KindOfElement | KindOfElementId |
| 15 | 0 | unbounded | email | 1 | SP | 1 |

**Tabelle 27 - Relation für Element Beispiel 2**

Zum Speichern der Werte von *<email>* steht die Relation *EMailAddress* zur Verfügung (vgl. Abbildung 39 – PK = Primärschlüssel, FK = Fremdschlüssel).

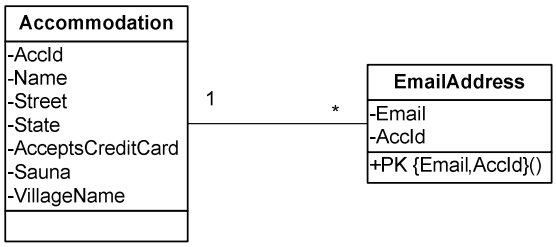



**Abbildung 39 - UML-Diagramm der Relation Accommodation und EMailAddress**

Die Relation *DBSchema* bleibt unverändert (siehe Tabelle 20). Neben *DBAttribute* hat nun auch die Relation *DBJoinSegment* einen entsprechenden Eintrag.

| DBAttribute | | | | | |
|---|---|---|---|---|---|
| DBAttId | *DBSchema* | DBRelation | DBAttribute | DBIsKey | DBDataType |
| 39 | accomDBSchema | EmailAddress | AccId | true | INT |
| 40 | accomDBSchema | EmailAddress | Email | true | VARCHAR(50) |

**Tabelle 28 - Relation DBAttribute von Beispiel 2**

| DBJoinSegment | | | | | | |
|---|---|---|---|---|---|---|
| DBRelShipId | DBAtt1 | DBAtt2 | DBRelation1 | DBRelation2 | *DBSchema* | DBJoinDirection |
| 4 | AccId | AccId | EmailAddress | Accommodation | accomDBSchema | 21 |

**Tabelle 29 - Relation DBJoinSegment von Beispiel 2**

Die für das Demonstrationsbeispiel relevanten Werte aus Abbildung 39 sind in der Relation *EMailAddress* gespeichert. Die Relation *Accommodation* bleibt wie in Tabelle 22 unverändert.

| EmailAddress | |
|---|---|
| *AccId* | Email |
| 45 | service.linz@maxxhotel.at |
| 46 | office@salzburg-hotel.at |

**Tabelle 30 - Relation EMailAddress von Beispiel 2**

In den Relationen der *Mapping-Komponente* sind folgende Einträge zu finden, wobei Tabelle 23 unverändert bleibt.

| ElementMapping | | | | |
|---|---|---|---|---|
| XSDElementID | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 15 | ET_A2 | 40 | 4 | 1 |

**Tabelle 31 - Relation ElementMapping von Beispiel 2**

Kommt man beim Durchlauf des Schemas zum Element <*email*>, erhält man aus Tabelle 31 die Information, dass ein *Join* zwischen den Relationen *Accommodation* und *EMailAddress* nötig ist, um das Attribut *Email* zu erreichen. Die Selektion der konkreten Werte für die jeweilige Instanz des Elementes <*email*> erfolgt unter Verwendung des Schlüssels beider Relationen, in diesem Fall des Attributes *AccId*.
Um alle Werte von <*email*> für eine Instanz von <*accommodation*> zu erhalten, müssen die entsprechenden Tupel, abhängig vom aktuellen Wert von *AccId* der Relation *Accommodation* selektiert werden.
Entsprechend Tabelle 30 enthält jedes <*accommodation*> - Element nur ein <*email*> - Element mit einem konkreten Wert.

### *4.3.3. Beispiel 3: ET_A$_{indirekt}$ für ein leeres Element*

Dieser Fall unterscheidet sich grundsätzlich nicht allzu sehr von Beispiel 2. Es soll nun gezeigt werden wie das Abbilden eines leeren Elementes (*empty element*) mit der Kardinalität * oder + von Statten geht. Das Element <*pool*> ist ein Repräsentant einer solchen Konstellation (vgl. Abbildung 40 und Abbildung 41).

```
<complexType name="accommodationType">
     <sequence>
       <element ref="ac:name"/>
       …
       <element name="pool" minOccurs="0" maxOccurs="unbounded">
         <complexType/>
       </element>
       …
     </sequence>
```

**Abbildung 40 - Ausschnitt accommodation.xml Beispiel 3**

```
<accommodations ...>
  <accommodation id="45" state="Austria">
       …
       <pool/>
       …
  </accommodation>
  <accommodation id="46" state="Austria">
       …
       <pool/>
       <pool/>
       …
  </accommodation>
</accommodations>
```

**Abbildung 41 - Ausschnitt *xray_accommodation.xsd* Beispiel 3**

Wie aus Abbildung 41 ersichtlich, verfügt *<accommodation id=45>* über einen Pool, *<accommodation id=46>* hingegen über zwei Pools.
Aufgrund der Normalisierung der Relationen werden die Elemente *<pool>* in die Relation *Pool* ausgelagert (vgl. Abbildung 42 PK = Primärschlüssel, FK = Fremdschlüssel).

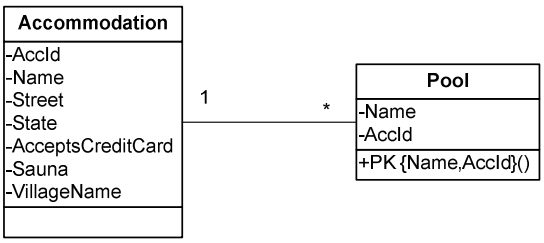



**Abbildung 42 - UML-Diagramm der Relation Accommodation und Pool**

Da es sich bei *<pool>* um leere Elemente handelt, werden keine realen Werte in die Relation *Pool* geschrieben. Bei leeren Elementen mit Kardinalität * oder +, wenn sie zusätzlich keine Attribute enthalten, ist letztendlich nur die Anzahl der Instanzen des leeren Elementes im XML-Dokument interessant.
Da jedoch in der Relation *Pool* der Fremdschlüssel allein zugleich Primärschlüssel wäre, musste in diesem Fall das Attribut *Name* der Relation *Pool* hinzugefügt werden. Anders wäre es unmöglich, mehrere Pools für eine *<accommodation>*-Instanz zu speichern. Das Attribut *Name* enthält daher nur, wie in Tabelle 32 ersichtlich, rein fiktive Surrogatnamen, die beim Auslesen der Relation unberücksichtigt bleiben.

| Pool | |
|---|---|
| *AccId* | Name |
| 45 | Pool1 |
| 46 | Pool1 |
| 46 | Pool2 |

**Tabelle 32 - Relation Pool von Beispiel 3**

Die Relationen der *RDB Schema-Komponente* enthalten somit folgende Tupel.

| DBAttribute | | | | | |
|---|---|---|---|---|---|
| DBAttId | *DBSchema* | DBRelation | DBAttribute | DBIsKey | DBDataType |
| 35 | accomDBSchema | Pool | AccId | true | INT |
| 36 | accomDBSchema | Pool | Name | true | VARCHAR(50) |

**Tabelle 33 - Relation DBAttribute von Beispiel 3**

| DBJoinSegment | | | | | | |
|---|---|---|---|---|---|---|
| DBRelShipId | DBAtt1 | DBAtt2 | DBRelation1 | DBRelation2 | *DBSchema* | DBJoinDirection |
| 5 | AccId | AccId | Pool | Accommodation | accomDBSchema | 21 |

**Tabelle 34 - Relation DBJoinSegment von Beispiel 3**

Die Relation *ElementMapping* enthält folgendes Tupel

| ElementMapping | | | | |
|---|---|---|---|---|
| XSDElementID | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 20 | ET_A2 | 36 | 5 | 1 |

**Tabelle 35 - Relation ElementMapping von Beispiel 3**

### *4.3.4. Beispiel 4: A_A$_{direkt/indirekt}$ für Attribute*

Handelt es sich um direktes Abbilden eines XML-Attributes enthält das Attribut *KindOfMapping* der Relation *AttributeMapping* den Wert A_A1.

Hat ein XML-Attribut einen fixen Wert (z.B. *state* in Element <*accommodation*>), ist der Wert im Metaschema gespeichert und *KindOfMapping* hat den Wert A_0. Ist das Element, zu dem ein Attribut gehört, ET_A2 (ET_A$_{indirekt}$) abgebildet, so sind auch dessen Attribute A_A2 (A_A$_{indirekt}$) abgebildet.

Die Werte der Attribute, egal ob A_A1 oder A_A2, werden unter Verwendung des in [ORT05] beschriebenen Schlüsselkonzepts aus den jeweiligen Datenrelationen gelesen und die Attribute entsprechend bei ihrem Element ausgegeben.
Zusätzlich muss jedoch beachtet werden, zu welchem Element ein Attribut gehört. Der Grund dafür ist, dass Attribute bei ihrer Deklaration einem komplexen Datentyp zugewiesen sind. Unerheblich ist, ob das Attribut lokal oder global definiert ist. Es ist nicht unüblich, dass zwei Elemente unterschiedlichen Namens den gleichen komplexen Datentyp (mit Attributen) als Basis haben. Es werden aber nur die Elemente und nicht die Datentypen abgebildet. Somit ist auch das Abbilden der Attribute abhängig von dem Element (*XSDElementID* in Tabelle 36), zu dem sie letztendlich gehören.

| AttributeMapping | | | | | |
|---|---|---|---|---|---|
| XSDElementID | XSDAttributeID | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 28 | 7 | A_A1 | 1 | NULL | 1 |
| 28 | 8 | A_0 | NULL | NULL | 1 |
| 6 | 2 | A_A2 | 29 | 1 | 1 |

**Tabelle 36 - Beispiel für AttributeMapping**

### *4.3.5. Weitere Hinweise bzgl. Metaschema und Abbilden*

Das Abbilden und Füllen der Relationen der drei Komponenten des Metaschemas für alle weiteren Elemente von *Accommodation.xml* und *xray_accommodation.xsd* verläuft großteils wie in den drei eben vorgestellten Beispielen.
Selbstverständlich sind die Besonderheiten der elementspezifischen bzw. attributspezifischen Eigenschaften in der XML-Schema-Instanz zu berücksichtigen. Diese können, wie erwähnt, anhand des im Anhang befindlichen kompletten Beispiels genau verfolgt werden.

Die Schlüsselkonzepte (*key*, *keyref* etc.) brauchen im Metaschema nicht berücksichtigt werden, da sie sich nur auf die Werte der XML-Elementinstanzen beziehen.
Es werden in den Relationen der Datenbank somit ohnehin nur Werte gespeichert, die für die Instanz eines von X-Rayxs verwalteten XML-Schemas gültig sind.

Dies liegt darin begründet, dass nur Daten aus wohlgeformten und gültigen XML-Dokumenten gespeichert werden. Diese Daten entsprechend folglich den in der Schemadatei verwendeten Schlüsselbedingungen.

# 5. Prototyp

In diesem Kapitel wird in gekürzter Form der Prototyp beschrieben, der implementiert wurde, um die Validität von X-Rayxs in der Laufzeitphase nachzuweisen. Die vollständige Beschreibung des Prototyps ist in [ORT05] nachzulesen.

Nach dem Vorstellen der Anwendungsfälle der Laufzeitphase und der Präsentation der Architektur, werden die Algorithmen zur Realisierung der Anwendungsfälle grob umrissen.

## *5.1. Anwendungsfälle*

Es werden nun die Anwendungsfälle (*use cases*) für die Laufzeitphase des X-Rayxs Prototyps vorgestellt. Die Anwendungsfälle beschreiben den typischen Ablauf der drei Hauptfunktionen des Prototys sowie das Login an den Prototyp.

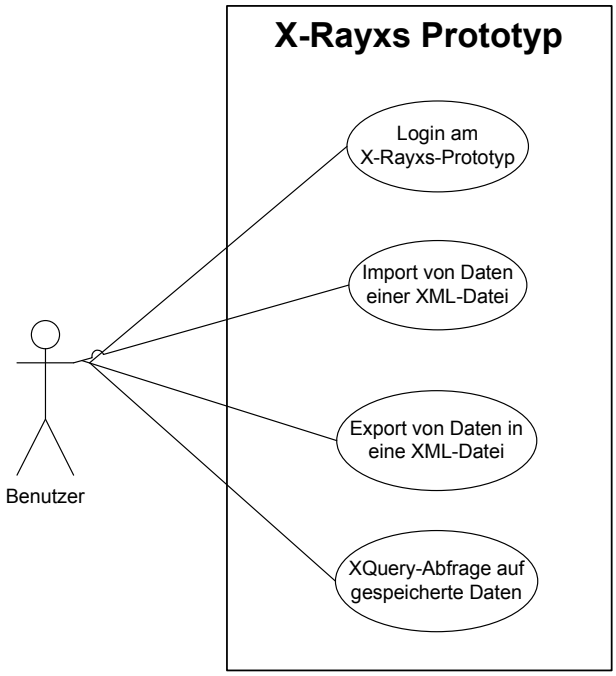



**Abbildung 43 - Use case Diagramm**

Es gibt demnach vier Anwendungsfälle (vgl. Abbildung 43):

### *5.1.1. Login am X-Rayxs-Prototyp*

Um Zugriff auf die Funktionen des Prototyps zu erhalten, muss sich der Benutzer mit einem gültigen Benutzernamen und Passwort am Prototyp anmelden. Erst durch Eingabe dieser Daten ist der indirekte Zugriff auf das Metaschema und dadurch auf die Funktionalität des Prototyps möglich.

### *5.1.2. Import von Daten einer XML-Datei*

Es werden die in einer XML-Datei gespeicherten Daten unter Verwendung des Metaschemas in den entsprechenden Relationen des RDBS gespeichert.

### *5.1.3. Export von Daten in eine XML-Datei*

Es werden die in einem RDBS gespeicherten Daten, die einer Abbildungsvariante und somit einem XML-Schema zugehörig sind, in eine XML-Datei exportiert.

### *5.1.4. XQuery-Abfrage auf gespeicherte Daten*

Es können beliebige XQuery-Abfragen erstellt werden. Diese werden auf eine temporäre XML-Datei abgesetzt, die mittels einer zuvor selektierten Abbildungsvariante erzeugt wurde.

## 5.2. Architektur

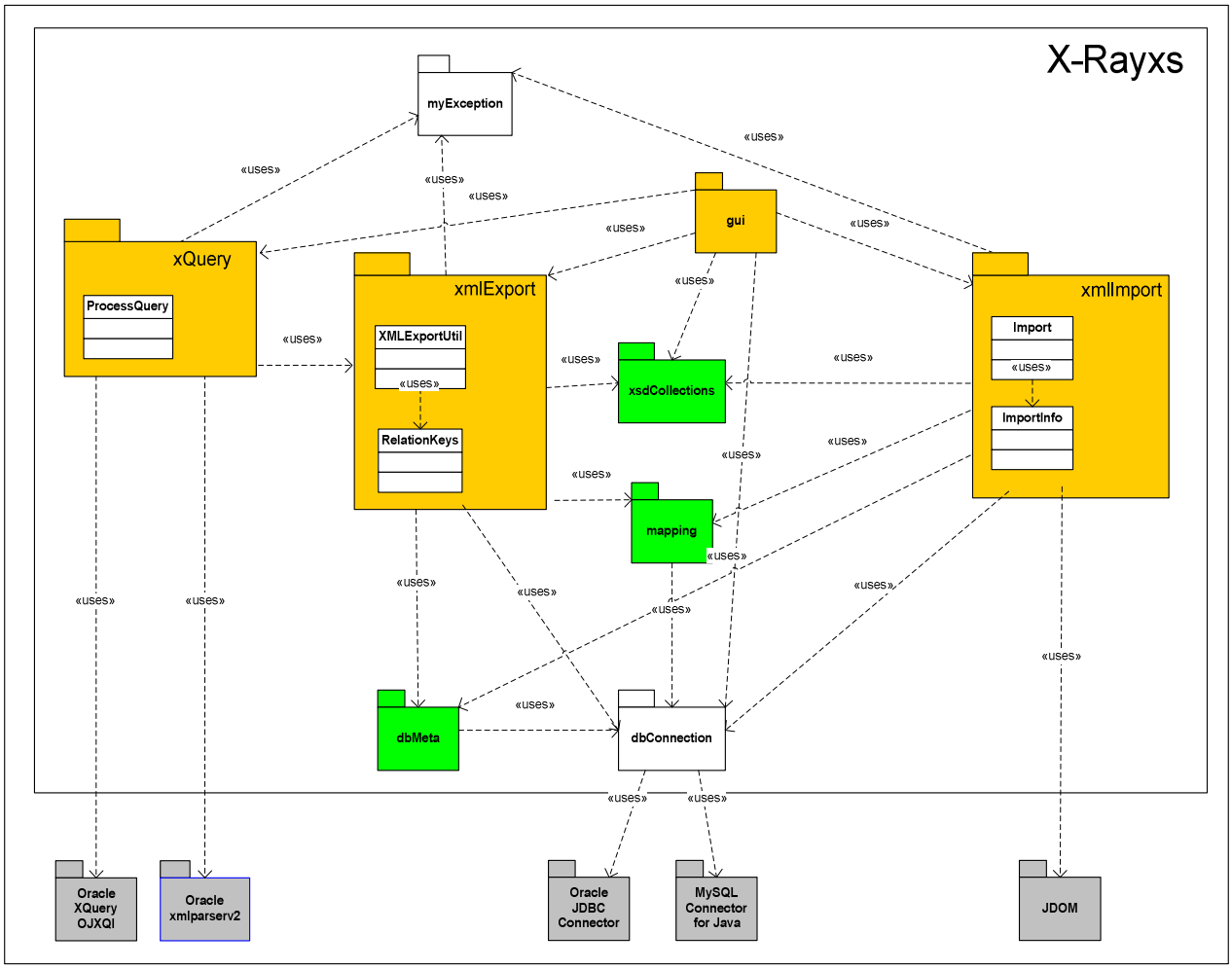



**Abbildung 44 - Architektur Prototyp X-Rayxs**

Die Java-Klassen des Prototyps wurden gemäß ihrer Funktionalitäten und Aufgaben in *Pakete* zusammengefasst. In Abbildung 44 sind die Pakete farblich gruppiert. Dadurch soll die Zusammengehörigkeit der Pakete verdeutlicht werden. So bilden die Pakete *xQuery*, *gui*, *xmlExport* und *xmlImport* dahingehend eine Gruppe, da diese die Verwendung des Metaschemas durch den Benutzer ermöglichen. Die Pakete *xsdCollections*, *dbmeta* und *mapping* entsprechen den drei Komponenten des Metaschmemas. Die Pakete *myException* und *dbConnection* stellen reine Hilfsklassen dar. Durch die Verwendung der externen Pakete (*OJXQI, xmlparserv2, JDBC connector, MySQL connector, JDOM*) verringert sich der Implementierungsaufwand erheblich.

Die genaue Beschreibung der Pakete kann in [ORT05] nachgelesen werden.

## *5.3. Laufzeitphase des Prototypen X-RAYxs*

Wie bereits in Kapitel **X-Ray** erläutert, lässt sich die Verwendung von X-Rayxs in zwei Hauptphasen unterteilen: die Initialisierungsphase und die Laufzeitphase.
Der Prototyp verfügt ausschließlich über die Funktionalitäten der Laufzeitphase (Import, Export, Abfragen mit XQuery). Abbildung 45 zeigt einen Überblick über den Prototyp und die Funktionen der Laufzeitphase. Jede Funktion in dieser Abbildung wird in einem der folgenden Unterkapitel verfeinert und näher erläutert.

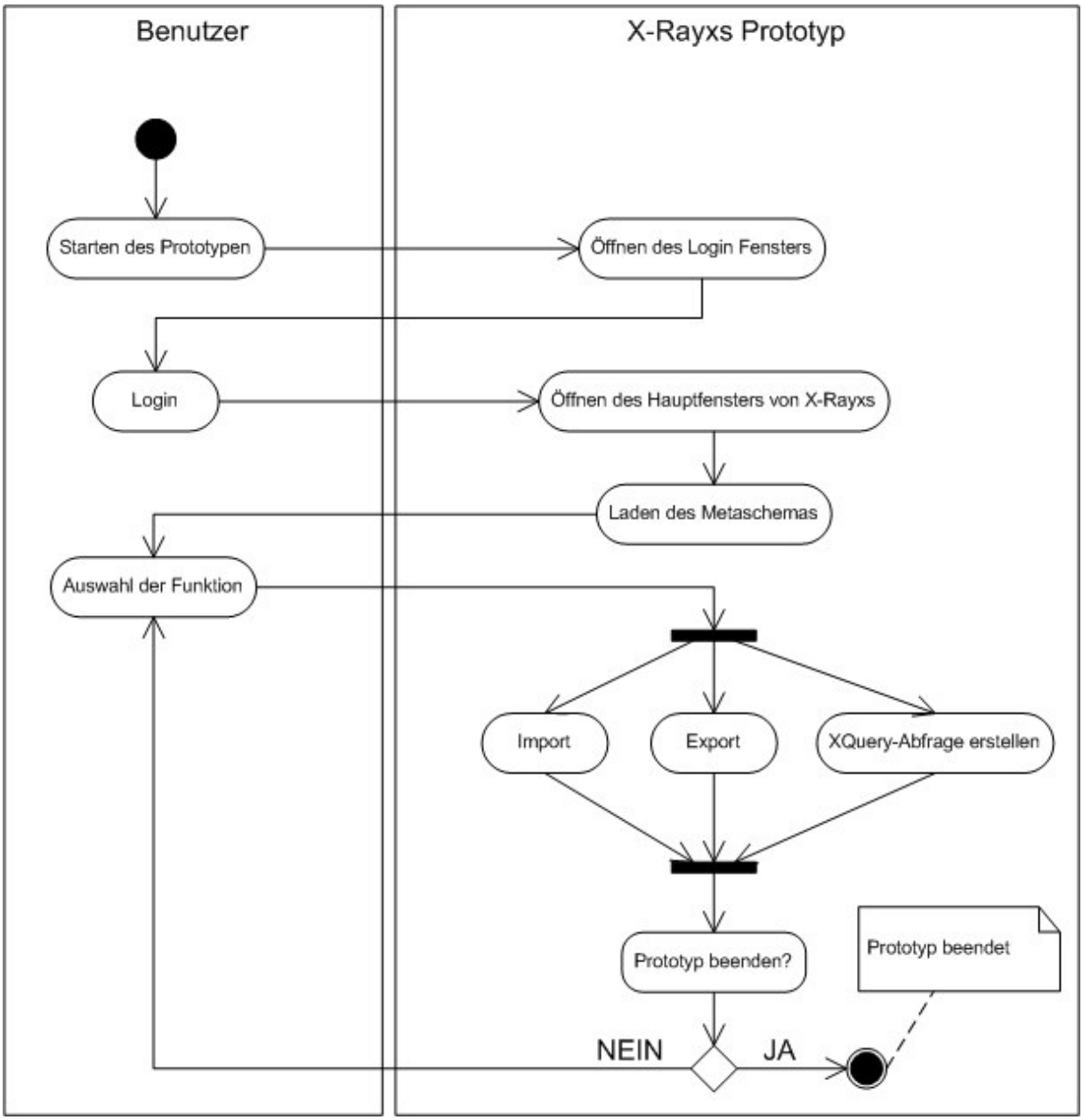



**Abbildung 45 - Übersicht X-Rayxs Prototyp**

### *5.3.1. Login*

Um auf die Funktionen des X-Rayxs-Prototyps zugreifen zu können, muss sich der Benutzer mittels eines Logins am Prototyp anmelden.
Dieses Login hat zwei wesentliche Aufgaben:

1. Schutz vor unbefugtem Zugriff auf das Metaschema
2. Auslösen des Vorganges zum Laden des Metaschemas

Sämtliche Daten des Metaschemas sind in einer Datenbank gespeichert. Diese Datenbank ist vor unbefugtem Zugriff gesichert und der Benutzer erhält nur durch Eingabe eines korrekten Benutzernamens und Passwortes Zugriff auf das Metaschema. Zusätzlich wird durch das erfolgreiche Anmelden das Laden des Metaschemas aus der Datenbank in die objektorientierte Repräsentation gestartet. Danach wird das Hauptfenster von X-Rayxs geöffnet, woraufhin der Benutzer auf die Funktionen des Prototyps, wie Import und Export von XML-Dokumenten, sowie das Erstellen von XQuery-Abfragen auf gespeicherte Daten, zugreifen kann.
Der Ablauf des Login-Vorganges ist in Abbildung 46 schematisch dargestellt.

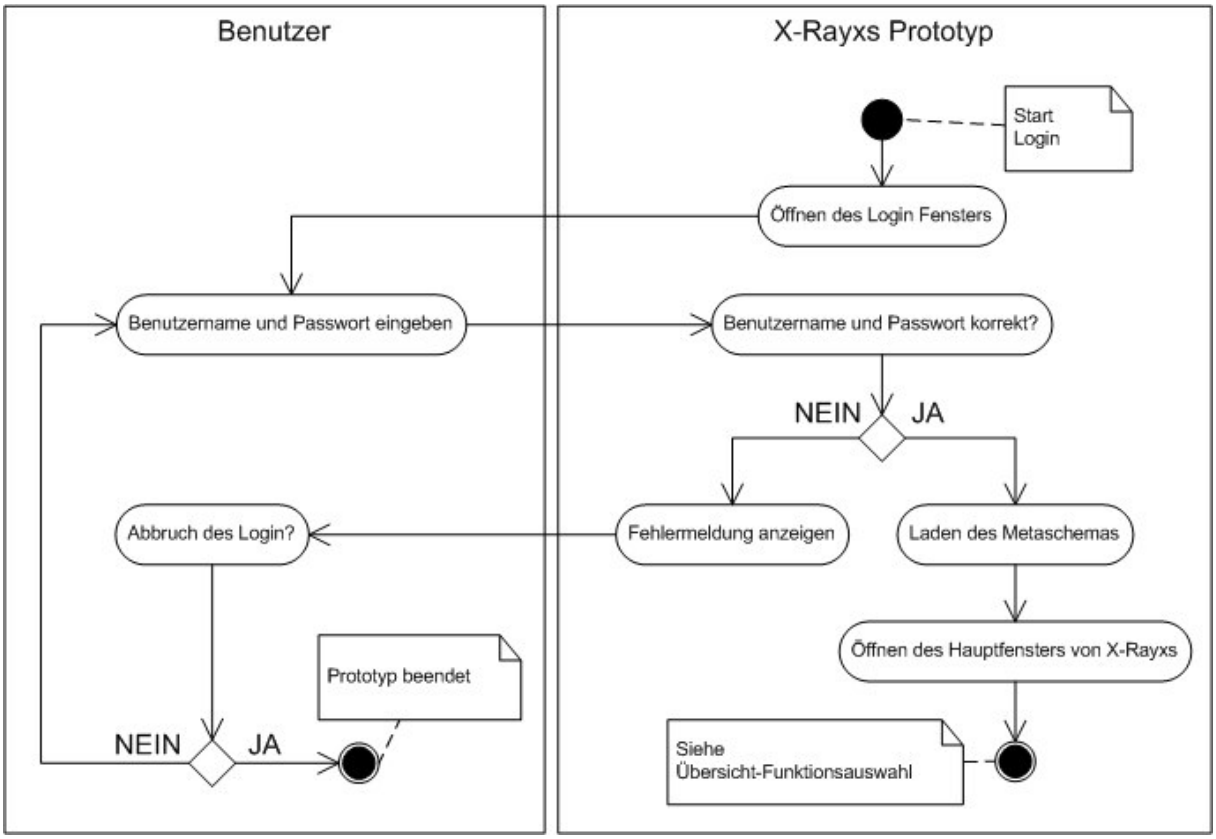



**Abbildung 46 - Login X-Rayxs Prototyp**

### *5.3.2. Laden des Metaschemas*

Das gesamte Metaschema von X-Rayxs, wie es in Kapitel **X-Rayxs** beschrieben wurde, ist in einem RDBS persistent gespeichert. Da bei den Funktionen der Laufzeitphase (Export, XQuery-Abfragen, Import) die Daten des Metaschemas benötigt werden, und somit eine Vielzahl von Datenbankzugriffen nötig wäre, wird eine objektorientierte Repräsentation des Metaschemas erstellt. Dieser nicht unbedingt nötige Aufwand begründet sich durch ein wesentlich verbessertes Laufzeitverhalten beim Ausführen der Funktionen des Prototyps. Diese objektorientierte Repräsentation wird, wie zuvor erläutert, direkt nach dem Login erstellt. Nachteilig dabei ist jedoch die etwas verlängerte Ladephase nach dem Login. Je nach Rechnerleistung und Verbindung zur Datenbank entsteht eine Wartezeit für den Benutzer zwischen dem Login und dem Öffnen des Hauptfensters.

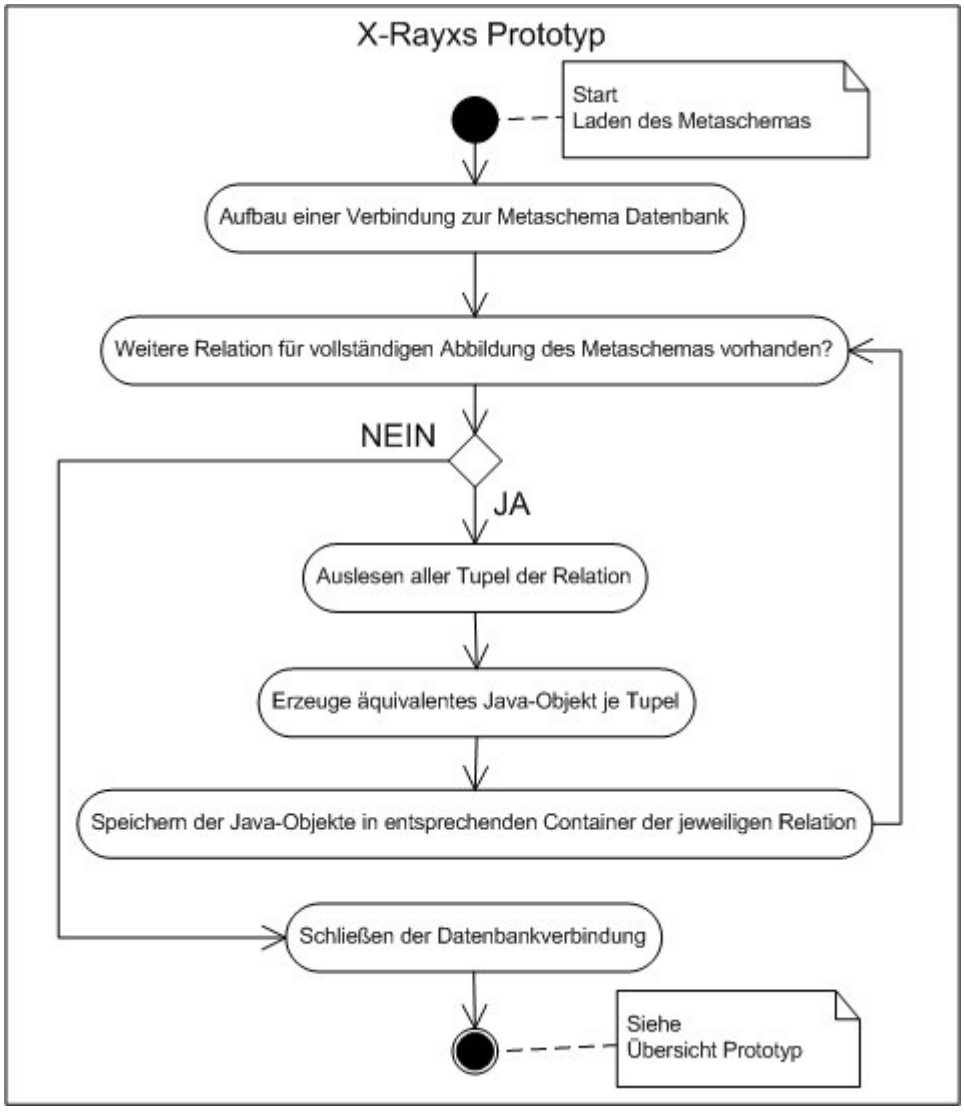



**Abbildung 47 - Ladeprozess**

Der Ladeprozess bezieht sich auf die *RDB Schema*-, *XML Schema*- und *Mapping-Komponente* des Metaschemas, das im Kapitel **Erweiterung von X-Ray zur Integration von XML Schema und RDBS** vorgestellt wurde. Abbildung 47 zeigt den Ablauf des Ladeprozesses. Im Prototyp wird dazu je Relation des Metaschemas ein Container erzeugt, der sämtliche Daten der Relation beinhaltet. Nachdem die Verbindung zum Datenbankserver, der das Metaschema beinhaltet, hergestellt wurde, wird für jede Relation ein eigenes „*ResultSet*“ ausgelesen. Jedes Tupel des jeweiligen „*ResultSet*“ wird anschließend in ein äquivalentes Java-Objekt übergeführt und im dafür erzeugten Container abgelegt. Wenn dies für jede Relation des Metaschemas ausgeführt worden ist, kann die Datenbankverbindung geschlossen werden. Für die *RDB Schema*- und *Mapping-Komponente* ist nach Abschluss des Ladevorganges das objektorientierte Abbild fertiggestellt.

Für die *XML Schema-Komponente* wurden nur die Daten und Beziehungsdaten geladen, die Beziehungen der Schemabestandteile untereinander jedoch noch nicht erstellt. Dies erfolgt erst nach Auswahl der Abbildungsvariante beim Durchführen einer der drei Hauptfunktionen des Prototyps. Diese geladenen Schemata bleiben dann für die Dauer der gesamten X-Rayxs-Session verfügbar.

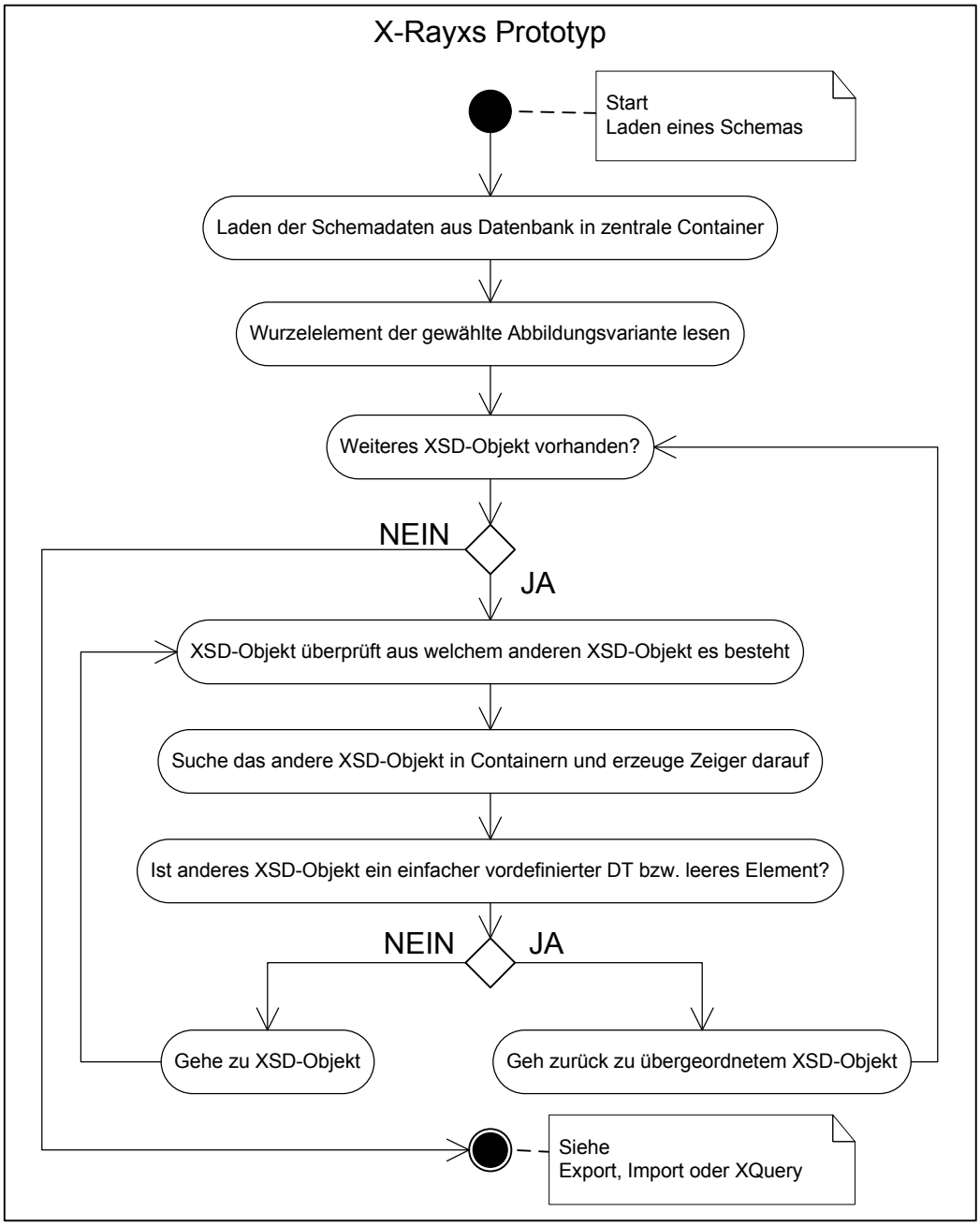



**Abbildung 48 - Aufbau des XML Schema**

Abbildung 48 zeigt den Prozess zur Verknüpfung sämtlicher Bestandteile eines gespeicherten XML-Schemas. Diese Bestandteile eines XML-Schemas werden in Folge XSD-Objekte genannt und beziehen sich in diesem Kontext auf alle möglichen Elemente/Tags die in X-Rayxs auf Grund der in Abschnitt **Unterstützte Konzepte von XML Schema** gemachten Einschränkungen auftreten dürfen.

Demnach kann ein XSD-Objekt u.a. Folgendes sein:

*<element>*, *<sequence>, <complexType>, <attribute>*, etc.

Sowohl die zu verbindenden XSD-Objekte, als auch die möglichen und notwendigen Verknüpfungen selbst, können aus dem UML-Diagramm (siehe Anhang) für XML

Schema abgeleitet werden. Somit können die XSD-Objekte und deren Beziehungen zueinander, wie sie in der Datenbank bereits bestanden haben, wiederhergestellt werden. Beispielsweise wird das XSD-Objekt <*element name="accommodations"*> mit dem im XML-Schema lokal deklarierten Datentyp <*complexType*>, der wiederum aus einer <*sequence*> besteht usw., verknüpft (siehe dazu Anhang).
Dieser Prozess wird ausgeführt, nachdem die Daten des Metaschemas geladen wurden (siehe Abbildung 47) und der Benutzer ausgewählt hat, mit welcher Abbildungsvariante ein Import oder Export durchgeführt werden soll. Dies hat den Vorteil, dass nur jene XSD-Objekte miteinander verknüpft werden müssen, die für das Ausführen der jeweiligen Funktion benötigt werden. Dieser Verknüpfungsprozess vereinfacht die Navigation zwischen den XSD-Objekten beim Ausführen der vom Benutzer gewählten Funktion erheblich.
Die Beziehungen zwischen den Java-Objekten, die durch den Ladeprozess erzeugt wurden und XML Schema-Objekte (XSD-Objekte) repräsentieren, werden mit Hilfe von Zeigern realisiert. Um dies in Java umzusetzen, besitzt jedes dieser Java-Objekte, das zur *XML Schema-Komponente* gehört, die Methode *build()*. Wie aus Abbildung 48 ersichtlich, wird das Wurzelelement des XML-Schemas ermittelt, und durch den Aufruf der *build()*-Methode dieses Elements der Verknüpfungsprozess gestartet. Das Wurzelelement ist deshalb der Ausgangspunkt, weil von hier aus alle weiteren, zum XML-Schema gehörenden XSD-Objekte erreicht werden können. Solange das durch die *build()*-Methode erreichte XSD-Objekt kein einfacher vordefinierter Datentyp oder kein leeres Element ist, wird vom Vorgänger-XSD-Objekt die *build()*-Methode des jeweiligen Nachfolgeobjekts aufgerufen. Es sucht sich somit jedes XSD-Objekt sein unmittelbares Kindobjekt selbstständig durch die *build()*-Methode und richtet einen Zeiger auf dieses. Dieser Vorgang setzt sich so lange fort, bis alle XSD-Objekte behandelt wurden und ein XML-Schema fertig geladen wurde.

### *5.3.3. Export*

Dieser Abschnitt beschreibt, wie ein XML-Dokument mit Hilfe von Schemadaten und Werten aus einer Datenbank erstellt wird. Abbildung 49 stellt den groben Ablauf des Exports dar.

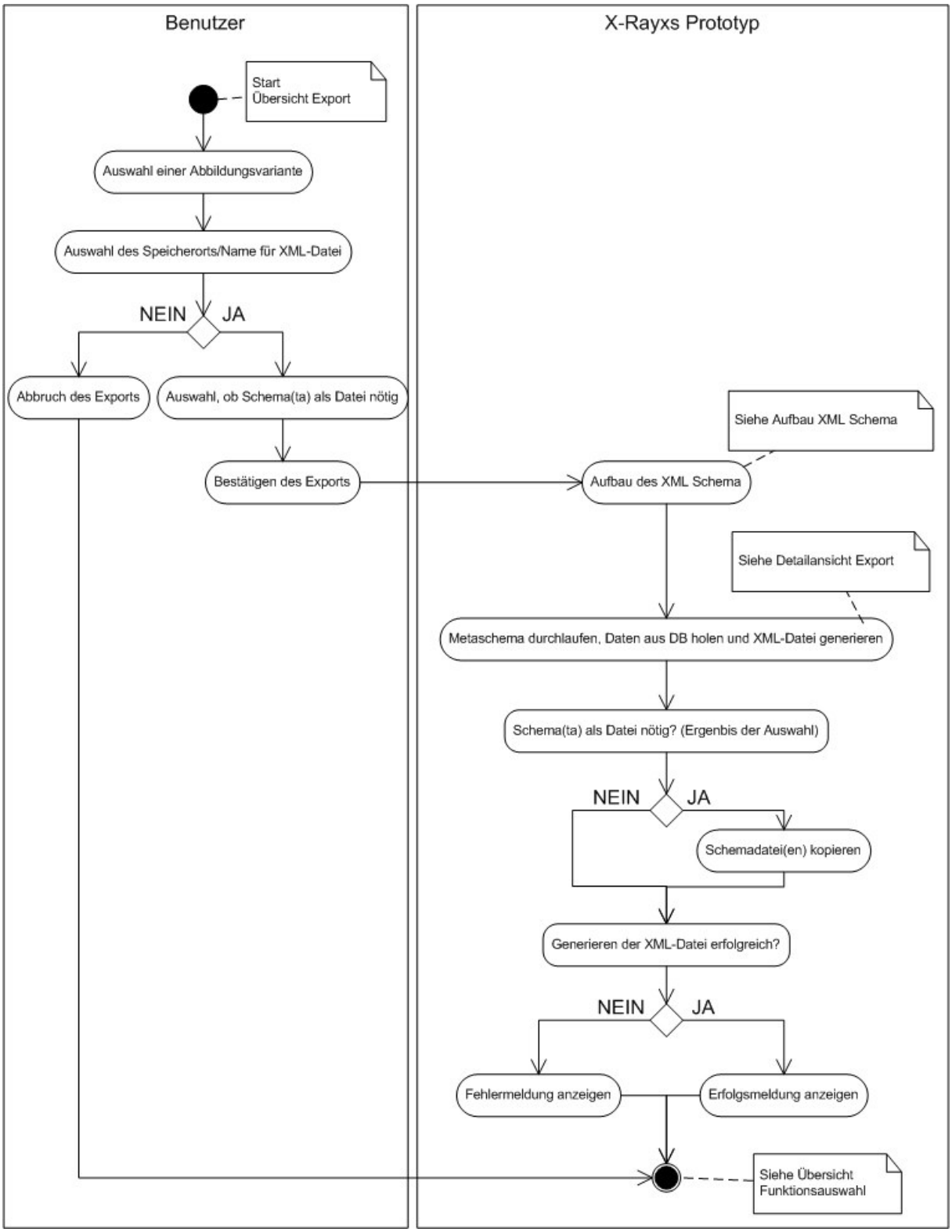



**Abbildung 49 - Übersicht Export**

Nach dem Login sind die Relationen des Metaschemas in eine objektorientierte Repräsentation übergeführt. Sind die Metaschemadaten geladen, werden die Abbildungsvarianten aus der Relation *Mappings* (siehe **X-Rayxs Metaschema Mapping-Komponente**) im Hauptfenster des Prototyps dargestellt. Nach Selektion einer Abbildungsvariante sowie der Auswahl des Speicherortes und Dateinamens für die zu exportierende XML Schema Instanz (XML-Dokument), wird der Aufbau der objektorientierten Schemastruktur mittels Instanzen entsprechender Klassen und Zeigern zwischen den Instanzen gestartet („Aufbau des XML Schemas“ in Abbildung 49).

Die Auswahl der zu exportierenden Abbildungsvariante stellt sich wie in Abbildung 50 dar. Der genaue Ablauf des Exportes aus Sicht des Benutzers kann im Benutzerhandbuch, welches sich im Anhang von [ORT05] befindet, nachgelesen werden.

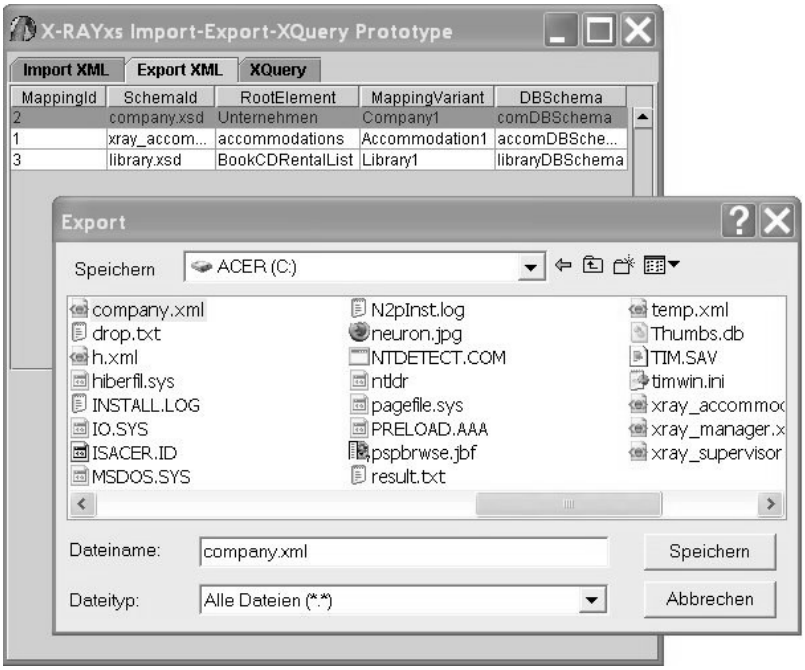



**Abbildung 50 - Export GUI des X-Rayxs Prototyps**

### *5.3.4. Abfragen mittels XQuery*

Diese Funktion des Prototyps ermöglicht es, Abfragen in XQuery [W3C04E] auf gespeicherte Daten abzusetzen. Abbildung 51 stellt den Ablauf der Funktion dar. Als unterstützte Abfragesprache für mit X-Rayxs gespeicherte Daten wurde XQuery gewählt, welche als Basis XPath2.0 [W3C04F], Quilt [QUI01] etc. hat und laut W3C als Abfragesprache für XML Daten zum Standard erhoben werden soll.

XQuery hat u.a. den Vorteil, die Datentypen von XML Schema zu unterstützen. Da die Spezifikation von XQuery noch nicht gänzlich abgeschlossen ist, gibt es kaum Unterstützung (Bibliotheken) für Java. Für den X-Rayxs Prototypen fiel die Wahl auf OJXQI [ORA05]. Oracle stellt dieses Java API für XQuery zur Verfügung. Ein großer Vorteil ist die einfache Handhabung von OJXQI sowie die volle Unterstützung von XQuery. Einer Methode der Klasse des API übergibt man ein XML-Dokument und eine gültige XQuery (*PreparedXQuery*), eingelesen von einem Textfeld des Prototyps, und erhält als Resultat ein *XQueryResultSet*, das man durchlaufen und z.B. wiederum in einem Textfeld oder einer Datei ausgeben kann. Abbildung 52 zeigt die Verwendung von OJXQI im Rahmen des Prototyps. Ein Nachteil ist, dass als Datenquelle nur eine Datenbank oder eine XML-Datei verwendet werden kann. Erstellt der Anwender eine XQuery, muss eine XML-Datei gemäß der selektierten Abbildungsvariante (siehe **Export**) temporär erstellt werden (bei Windows z.B. im Ordner *TEMP*), die dann als Ziel für die XQuery verwendet wird. Da die XML-Dateien in einem Systemordner für temporäre Dateien abgelegt werden, können sie vom System nach Beendigung der X-Rayxs-Session gelöscht werden.

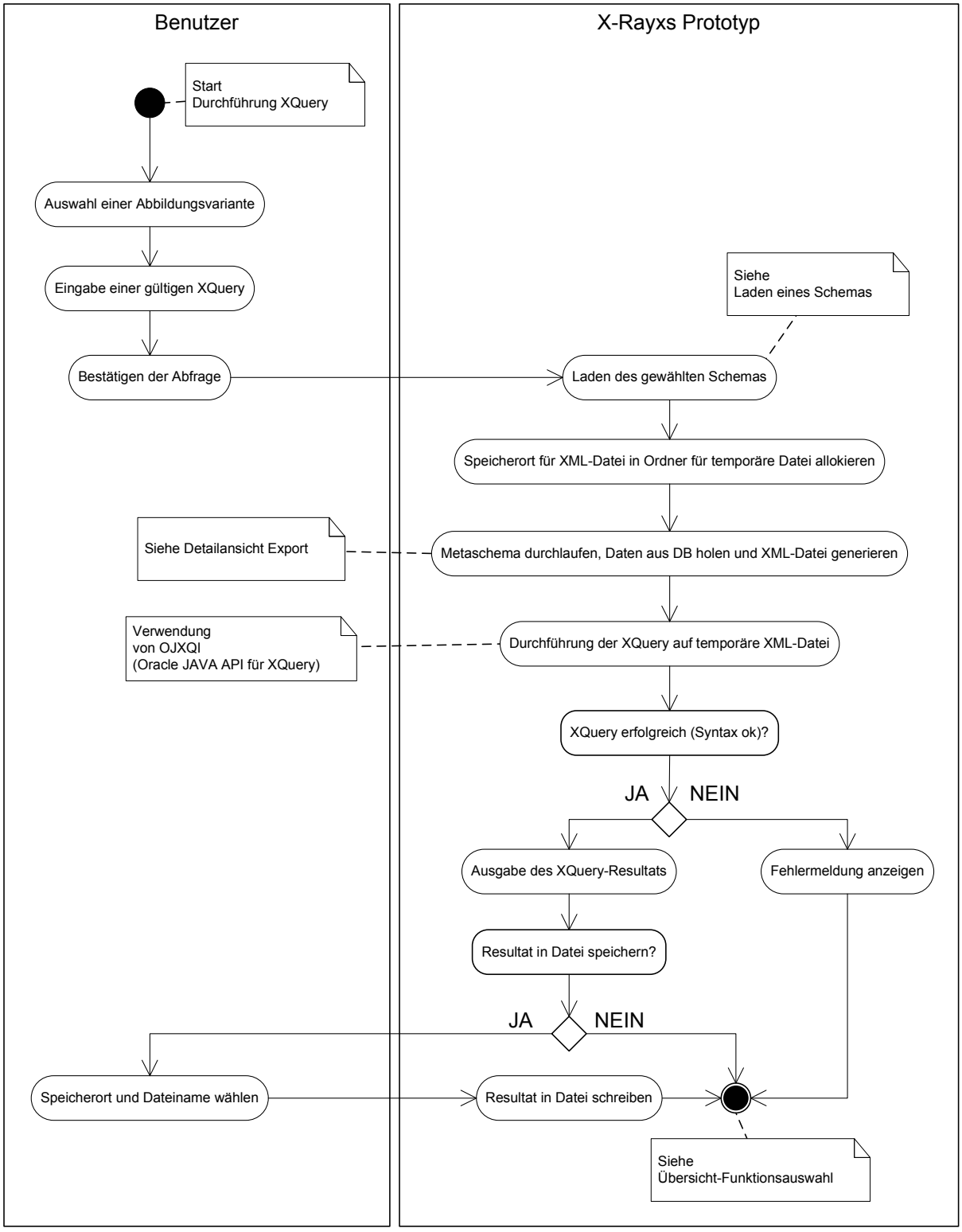



**Abbildung 51 - Abfragen mit XQuery**

```
public class ProcessXQuery{
      public void doTheQuery(String filename,String query,
JTextArea result){
           try{
                XQueryContext ctx = new XQueryContext();
                PreparedXQuery xq = ctx.prepareXQuery(query);
           XQueryResultSet rset = xq.executeQuery(true);

           PrintStream exportStream = new PrintStream(new
PipedOutputStream());
           //Durchlaufen des Resultats der XQuery
           while (rset.next()){
                     XMLNode node = rset.getNode();
                     node.print(exportStream);
           }
           }
           catch(Exception e){}
      }
}
```

**Abbildung 52 - Anwendung von OJXQI [ORA05]**

Das Resultat der XQuery wird in einem Textfeld des Prototyps angezeigt und kann vom Anwender bei Bedarf in eine beliebige Datei exportiert werden. Siehe dazu Abbildung 53.

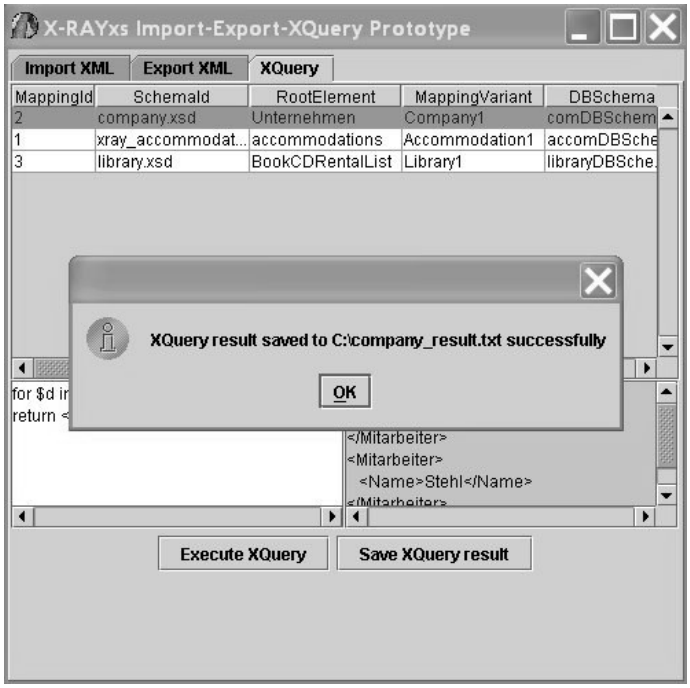



**Abbildung 53 - GUI für XQuery**

### *5.3.5. Import*

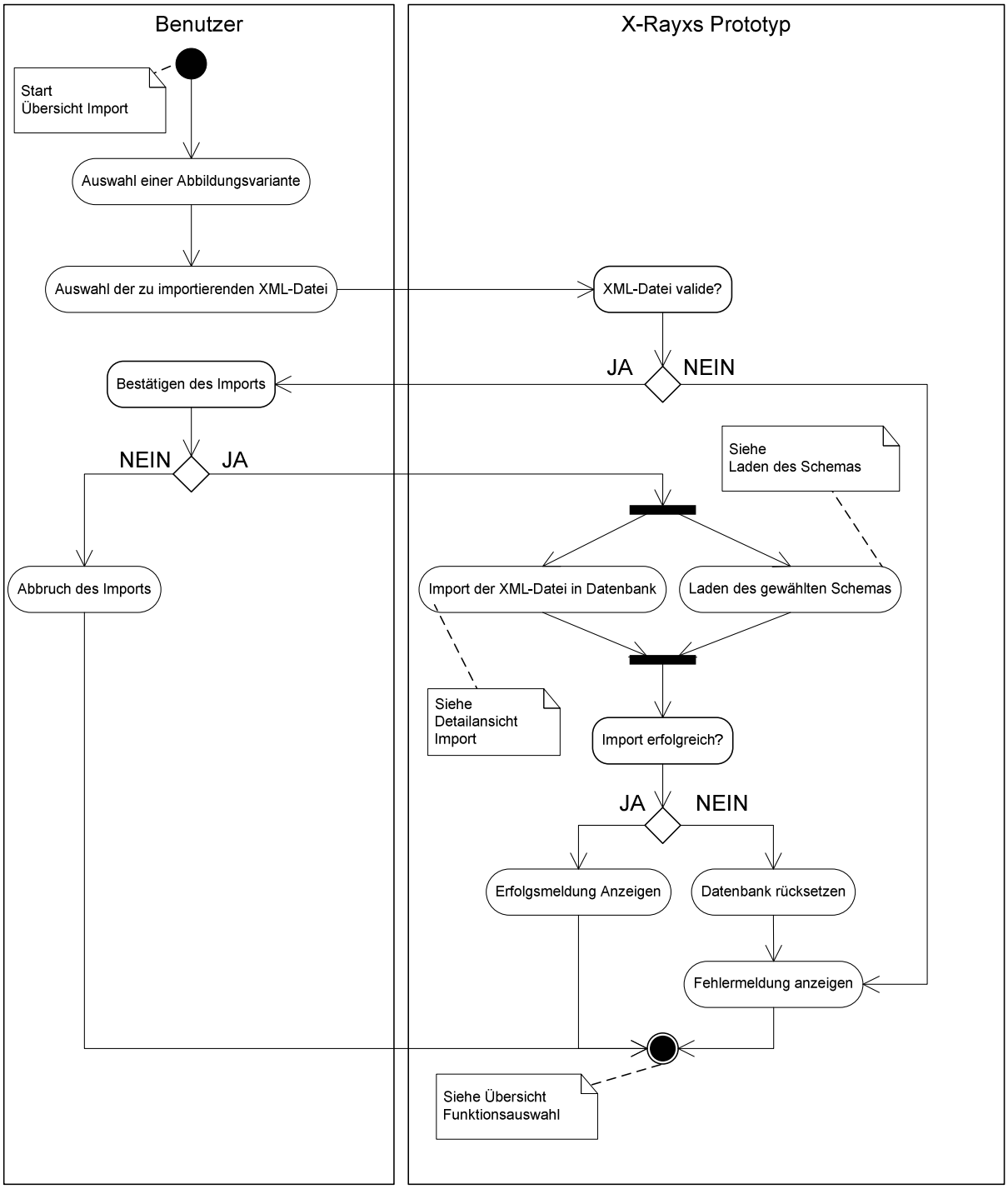



**Abbildung 54 - Übersicht Import**

Abbildung 54 zeigt den grundsätzlichen Ablauf des Imports einer XML-Datei in eine relationale Datenbank. Die Initialisierungsphase ist hier bereits vorweggenommen. Der Prozess wird vom Benutzer durch das Starten des Prototyps X-Rayxs und der Auswahl des Import-Tabs angestoßen (siehe Abbildung 55 / Abbildung 45). Wie

bereits erwähnt, wird beim Starten des Prototyps ein lokales Speicherabbild der gesamten Metadatenbank, in Form von äquivalenten Instanzen von Java Klassen, erstellt (siehe Abbildung 47 / Abbildung 48). Dem Benutzer werden nun die in der Initialisierungsphase eingerichteten Abbildungsvarianten angezeigt, aus denen er die für die zu importierende XML-Datei Passende auswählt. Anschließend werden die zuvor erzeugten Instanzen der Java-Klassen, die der *XML Schema-Komponente* zugehörig sind, für die gewählte Abbildungsvariante in Beziehung gesetzt. Das heißt, mit Hilfe von Zeigern und Containern wird die Struktur und Ordnung, die durch ein XML Schema vorgegeben ist, in das objektorientierte Speicherbild übergeführt. Wählt der Benutzer jetzt eine wohlgeformte und valide XML-Datei aus, ist dies der Start für den eigentlichen Import-Prozess. Dieser wird im folgenden Abschnitt genauer behandelt. Abgeschlossen wird der Import durch eine Mitteilung über den erfolgreichen Import bzw. durch eine entsprechende Fehlermeldung. Im letzteren Fall wird die Datenbank, in der die Werte der XML-Datei gespeichert werden sollen, in den ursprünglichen Zustand zurückgesetzt.

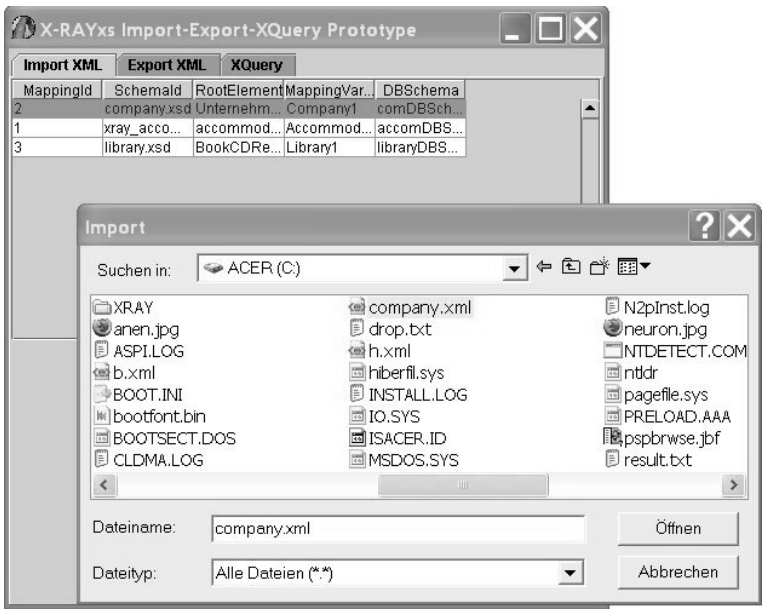



**Abbildung 55 - Prototyp X-Rayxs, Import**

Für den Importalgorithmus wird JDOM [JDOM05] verwendet. Eine detaillierte Beschreibung des Algorithmus ist in [ORT05] zu finden.

# 6. Ausblick und kritische Würdigung

Aufgrund der durch X-Ray vorgegebenen Designziele ist der Variationsraum für andere Lösungen sehr eng. Aufgrund der Vorgaben muss das Metaschema persistent in einer relationalen Datenbank gespeichert und für die Laufzeit in eine objektorientierte Repräsentation übergeführt werden. Es stehen somit letztlich nur kleinere Variationsmöglichkeiten bzgl. des Designs von X-Rayxs zur Verfügung. Einige Variationsmöglichkeiten werden anschließend aufgezählt.

- Bei der Entwicklung des Metaschemas und der korrespondierenden objektorientierten Klassen von X-Rayxs wurde die Integrationsmöglichkeit von X-Ray-QL, der deklarativen XML Datenmanipulationssprache für X-Ray [KIMM02], nicht berücksichtigt.

- Unterschiede zwischen den relationalen Datenbanken verschiedener Anbieter wurden nicht berücksichtigt. Das Metaschema wurde für eine Oracle-Datenbank entwickelt, in der es z.B. im Gegensatz zu einer MySQL-Datenbank keine Attribute vom Typ *Boolean* gibt.

- Durch einen anderen Aufbau des Metaschemas könnte man eventuell die Hilfskonstrukte *XSDContentParticle*, *XSDSequencePosition* etc. einsparen. Ob sich eine dahingehende Veränderung als zielführend erweist, kann derzeit nicht gesagt werden.

- Einige Relationen sind nur der Vollständigkeit halber im Metaschema enthalten. Ein Grund dafür ist, dass diese Relationen eventuell bei einer Erweiterung des Metaschemas doch Verwendung finden. Dies ist z.B. bei der Relation *XSDEmptyComplexDataType* der Fall. Diese Relationen werden derzeit auch nicht ins objektorientierte Abbild übernommen, da sie keinen zusätzlichen Informationsgehalt aufweisen.

Wie erwähnt, sind die Variationsmöglichkeiten aufgrund der Vorgaben sehr gering und das Metaschema müsste anderes konzipiert werden, um eventuelle Verbesserungen zu erreichen.

Im Hinblick auf die Zukunft gibt es jedoch noch einige offene Punkte bzgl. des Metaschemas. Wie im vorhergehenden Kapitel erwähnt, konnten nicht alle Konzepte von XML Schema bei der Entwicklung des Metaschemas berücksichtigt werden. (vgl. **Nicht unterstützte XML Schema Konzepte**). Das Metaschema müsste dahingehend erweitert werden, um auch diese bisher unberücksichtigten Konzepte abbilden zu können. Es müssen Relationen entwickelt werden, um die Daten der Konzepte zu speichern. Dabei könnte es auch zu Veränderungen bestehender Relationen kommen.

# Literaturverzeichnis

**[W3C04A]**
http://www.w3.org/TR/REC-xml
Extensible Markup Language (XML) 1.0 (Third Edition)
W3C05 Recommendation 04 February 2004
Editors: Tim Bray, Textuality and Netscape
Jean Paoli, Microsoft
C. M. Sperberg-McQueen, W3C05
Eve Maler, Sun Microsystems, Inc. - Second Edition
François Yergeau - Third Edition
zuletzt besucht: 15.03.2005

**[W3C04B]**
http://www.w3.org/TR/xmlschema-1/
XML Schema Part 1: Structures Second Edition
W3C05 Recommendation 28 October 2004
Editors: Henry S. Thompson, University of Edinburgh
David Beech, Oracle Corporation
Murray Maloney, for Commerce One
Noah Mendelsohn, Lotus Development Corporation
zuletzt besucht: 15.03.2005

**[SCHN04]**
http://www-lehre.inf.uos.de/~tschnied/HTML/
XML Schema & XSL-Transformationen
Tanja Schniederberend
Universität Osnabrück 11.06.2003
zuletzt besucht: 15.03.2005

**[GALI04]**
http://www.galileocomputing.de/glossar/
Glossar von Galileo Press

**[W3C04C]**
http://www.w3.org/TR/xmlschema-0/
XML Schema Part 0: Primer Second Edition
W3C05 Recommendation 28 October 2004
Editors: David C. Fallside, IBM
Priscilla Walmsley - Second Edition
zuletzt besucht: 15.03.2005

**[W3C04D]**
http://www.w3.org/TR/xmlschema-2/
XML Schema Part 2: Datatypes Second Edition
W3C05 Recommendation 28 October 2004
Editors: Paul V. Biron, Kaiser Permanente, for Health Level Seven
Ashok Malhotra, Microsoft (formerly of IBM)
zuletzt besucht: 15.03.2005

**[W3C05]**
http://www.w3.org
The World Wide Web Consortium (W3C)
zuletzt besucht: 15.03.2005

**[DVIN03A]**
http://www.xml.dvint.com/docs/SchemaStructuresQR-2.pdf
XML Schema – Structures Quick Reference
2003 D Vint Productions
http://www.xml.dvint.com
ver 1/03
zuletzt besucht: 15.03.2005

**[KENN00]**
http://www.mut.de/media_remote/katalog/bsp/3827259444bsp.htm
SOAP developer's guide
ISBN 3-8272-5944-4
Kennard Scribner / Marc C. Stive
Dezember 2000
zuletzt besucht: 15.03.2005

**[ECKS04]**
http://www.dbis.informatik.hu-berlin.de/lehre/WS0405/XMLSW/VL/xml_VLKap4.4.pdf
Script zur Vorlesung XML und Semantic Web 2004
Dr. Rainer Eckstein
zuletzt besucht: 15.03.2005

**[DVIN03B]**
http://www.xml.dvint.com/docs/SchemaDataTypesQR-1.pdf
XML Schema - Data Types Quick Reference
2001 D VInt Productions
http://www.xml.dvint.com
ver 9/01
zuletzt besucht: 15.03.2005

**[HOLZ04]**
http://www.xml-schnittstelle.de/xml-schemas.html
Studie: Entwicklung einer XML-Schnittstelle für den deutschen Lebensversicherungsmarkt am Beispiel der Antragstellung
Dipl.Math.oec. Martin Holzwarth
Kapitel 6: XML-Schemas
zuletzt besucht: 15.03.2005

**[KAPP04]**
ftp://ftp.ifs.uni-linz.ac.at/pub/publications/2004/0304.pdf
Integrating XML and Relational Database Systems
Gerti Kappel, Vienna University
Elisabeth Kapsammer und Werner Retschitzegger,University of Linz
in: World Wide Web Journal(WWWJ), Kluwe Academic Publishers, Vol. 7(4), December 2004, pp. 343-384
zuletzt besucht: 17.03.2005

**[SGML05]**
http://www.w3.org/MarkUp/SGML/
Overview of SGML Resources
zuletzt besucht: 15.03.2005

**[ORT05]**
Christian Ortner; Entwicklung eines Prototypen zur Integration von relationalen Datenbanken und XML Schema; Diplomarbeit; Johannes Kepler Universität Linz; 2005

**[W3C04E]**
http://www.w3.org/XML/Query
W3C XML XQuery
zuletzt besucht: 15.03.2005

**[W3C04F]**
http://www.w3.org/TR/2005/WD-xpath20-20050211/
XML Path Language (XPath) 2.0
zuletzt besucht: 15.03.2005

**[QUI01]**
http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html
Quilt: An XML Query Language
zuletzt besucht: 15.03.2005

**[ORA05]**
http://www.oracle.com/technology/sample_code/tech/xml/xmldb/jxqi.html
JXQI: Java XQuery API (developed by Oracle)
zuletzt besucht: 15.03.2005

**[JDOM05]**
http://www.jdom.org/
JDOM 2005
Jason Hunter
Brett McLaughlin
zuletzt besucht: 15.03.2005

**[KIMM02]**
Eugen Kimmersdorfer; Entwurf einer deklarativen XML Datenmanipulationssprache für X-Ray; Diplomarbeit; Johannes Kepler Universität Linz; 2002

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Anhang A: Beispiel XML-Dokument mit Schemadateien

### xray_accommodation.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.ifs.uni-linz.ac.at/XRay/AccommodationSchema"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:ac="http://www.ifs.uni-
linz.ac.at/XRay/AccommodationSchema" xmlns:su="http://www.ifs.uni-linz.ac.at/XRay/Supervision"
elementFormDefault="qualified">
    <import namespace="http://www.ifs.uni-linz.ac.at/XRay/Supervision"
schemaLocation="xray_supervision.xsd"/>
    <redefine schemaLocation="xray_management.xsd">
        <complexType name="managerType">
            <complexContent>
                <extension base="ac:managerType">
                    <sequence>
                        <element name="salary" type="positiveInteger"/>
                    </sequence>
                    <attribute name="mgrId" type="positiveInteger"/>
                    <attribute name="since" type="gYear"/>
                </extension>
            </complexContent>
        </complexType>
    </redefine>
    <!-- Start of elements-->
    <element name="name" type="string"/>
    <element name="accommodations">
        <complexType>
            <sequence>
                <element name="accommodation" type="ac:accommodationType" minOccurs="0"
                    maxOccurs="unbounded"/>
                <element name="accAwards" type="ac:accAwardsType"/>
            </sequence>
        </complexType>
        <key name="AccKey">
            <selector xpath="ac:accommodation"/>
            <field xpath="@id"/>
```

```
    </key>
    <keyref name="AwardWinnerRef" refer="AccKey">
        <selector xpath="ac:accAwards/ac:award"/>
        <field xpath="ac:winner"/>
    </keyref>
</element>
<!-- Start of complex types-->
<complexType name="accommodationType">
    <sequence>
        <element ref="ac:name"/>
        <element name="address" type="ac:addressType"/>
        <element name="manager" type="ac:managerType"/>
        <element name="supervisor" type="su:supervisionType"/>
        <element name="email" type="string" minOccurs="0" maxOccurs="unbounded"/>
        <element name="phone" type="ac:phoneType" maxOccurs="unbounded"/>
        <element name="acceptsCreditCards" minOccurs="0">
            <complexType/>
        </element>
        <element name="facilities">
            <complexType/>
        </element>
        <element name="sauna" type="ac:saunaType"/>
        <element name="pool" minOccurs="0" maxOccurs="unbounded">
            <complexType/>
        </element>
        <element name="description" type="ac:descriptionType" minOccurs="0"/>
        <element name="room" type="ac:roomType" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="id" type="positiveInteger" use="required"/>
    <attribute name="state" type="string" fixed="Austria"/>
</complexType>
<complexType name="roomType" abstract="true">
    <sequence>
        <element name="roomNumber" type="positiveInteger"/>
        <element name="size" type="positiveInteger"/>
        <element name="nrOfBeds" type="positiveInteger"/>
    </sequence>
</complexType>
<complexType name="standardRoomType">
    <complexContent>
        <extension base="ac:roomType">
```

```
            <sequence>
                <element name="tv" type="string" minOccurs="0" maxOccurs="2"/>
                <element name="shower" minOccurs="0">
                    <complexType/>
                </element>
            </sequence>
        </extension>
    </complexContent>
</complexType>
<complexType name="luxuryRoomType">
    <complexContent>
        <extension base="ac:standardRoomType">
            <attribute name="hasButler" type="boolean" use="required"/>
        </extension>
    </complexContent>
</complexType>
<complexType name="villageType">
    <simpleContent>
        <extension base="string">
            <attribute name="yearOfFoundation" type="gYear"/>
        </extension>
    </simpleContent>
</complexType>
<complexType name="addressType">
    <sequence>
        <element name="street" type="string"/>
        <element name="village" type="ac:villageType"/>
        <element name="country" type="string"/>
    </sequence>
    <attribute name="postalCode" type="string" use="required"/>
</complexType>
<complexType name="descriptionType">
    <all minOccurs="0" maxOccurs="unbounded">
        <element name="rating" type="string"/>
        <element name="comment" type="string"/>
    </all>
</complexType>
<complexType name="phoneType">
    <attribute name="number" type="string" use="required"/>
</complexType>
<complexType name="saunaType">
```

```
        <attribute name="available" type="boolean" use="required"/>
    </complexType>
    <complexType name="accAwardsType">
        <sequence>
            <element name="award" minOccurs="0" maxOccurs="unbounded">
                <complexType>
                    <sequence>
                        <element name="year" type="gYear"/>
                        <element name="winner" type="positiveInteger"/>
                    </sequence>
                </complexType>
            </element>
        </sequence>
    </complexType>
</schema>
```

## xray_supervision.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.ifs.uni-linz.ac.at/XRay/Supervision"
xmlns:su="http://www.ifs.uni-linz.ac.at/XRay/Supervision"
xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <complexType name="supervisionType">
        <sequence>
            <element name="supId" type="positiveInteger"/>
            <element name="firstName" type="string"/>
            <element name="lastName" type="string"/>
        </sequence>
    </complexType>
</schema>
```

## xray_management.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.ifs.uni-linz.ac.at/XRay/AccommodationSchema"
xmlns:ac="http://www.ifs.uni-linz.ac.at/XRay/AccommodationSchema"
xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <complexType name="managerType">
        <sequence>
            <element name="firstName" type="string"/>
            <element name="lastName" type="string"/>
        </sequence>
    </complexType>
</schema>
```

## accommodation.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<accommodations xmlns="http://www.ifs.uni-linz.ac.at/XRay/AccommodationSchema"
xmlns:su="http://www.ifs.uni-linz.ac.at/XRay/Supervision"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.ifs.uni-
linz.ac.at/XRay/AccommodationSchema xray_accommodation.xsd">
    <accommodation id="45" state="Austria">
        <name>Steigenberger MAXX Hotel Linz</name>
        <address postalCode="4020">
            <street>Am Winterhafen 13</street>
            <village yearOfFoundation="1600">Linz</village>
            <country>Oberoesterreich</country>
        </address>
        <manager mgrId="1" since="1995">
            <firstName>Karin</firstName>
            <lastName>Maier</lastName>
            <salary>20000</salary>
        </manager>
        <supervisor>
            <su:supId>2</su:supId>
            <su:firstName>Klaus</su:firstName>
            <su:lastName>Klammer</su:lastName>
        </supervisor>
        <email>service.linz@maxxhotel.at</email>
        <phone number="+4373278990"/>
```

```
    <acceptsCreditCards/>
    <facilities/>
    <sauna available="false"/>
    <pool/>
    <description>
        <comment>noisy</comment>
        <rating>3</rating>
        <comment>nice rooms</comment>
        <rating>2</rating>
    </description>
    <room xsi:type="standardRoomType">
        <roomNumber>101</roomNumber>
        <size>25</size>
        <nrOfBeds>2</nrOfBeds>
        <tv>Panasonic TX-32PS11D</tv>
        <shower/>
    </room>
    <room xsi:type="standardRoomType">
        <roomNumber>105</roomNumber>
        <size>33</size>
        <nrOfBeds>2</nrOfBeds>
    </room>
</accommodation>
<accommodation id="46" state="Austria">
    <name>Hotel Wolf-Dietrich</name>
    <address postalCode="5020">
        <street>Wolf-Dietrich-Strasse 7</street>
        <village yearOfFoundation="1500">Salzburg</village>
        <country>Salzburg</country>
    </address>
    <manager mgrId="2" since="1985">
        <firstName>Klaus</firstName>
        <lastName>Klammer</lastName>
        <salary>30000</salary>
    </manager>
    <supervisor>
        <su:supId>1</su:supId>
        <su:firstName>Herman</su:firstName>
        <su:lastName>Maier</su:lastName>
    </supervisor>
    <email>office@salzburg-hotel.at</email>
```

```
            <phone number="0043 662 871275"/>
            <facilities/>
            <sauna available="true"/>
            <pool/>
            <pool/>
            <description>
                <comment>beautiful</comment>
                <rating>1</rating>
                <comment>very nice landscape</comment>
                <rating>1</rating>
            </description>
            <room xsi:type="luxuryRoomType" hasButler="true">
                <roomNumber>305</roomNumber>
                <size>95</size>
                <nrOfBeds>3</nrOfBeds>
                <tv>Sony KV-32FQ85</tv>
            </room>
        </accommodation>
        <accAwards>
            <award>
                <year>2003</year>
                <winner>46</winner>
            </award>
            <award>
                <year>2004</year>
                <winner>45</winner>
            </award>
        </accAwards>
    </accommodations>
```

# Anhang B: Relationen des Metaschemas mit Beispieldaten gefüllt

### Relationen der XML Schema - Komponente

| XSDImport | | | |
|---|---|---|---|
| Id | *SchemaDeclarationIdBase* | *SchemaDeclarationIdImport* | *NameSpaceId* |
| | 1 | 2 | 3 |

| XSDRedefine | | | |
|---|---|---|---|
| Id | *SchemaDeclarationIdBase* | *SchemaDeclarationIdRedefine* | *ComplexDataTypeId* |
| | 1 | 3 | 4 |

| XSDInclude | | |
|---|---|---|
| Id | *SchemaDeclarationIdBase* | *SchemaDeclarationIdInclude* |

| XSDNameSpaces | |
|---|---|
| NameSpaceId | Uri |
| 1 | http://www.ifs.uni-linz.ac.at/XRay/AccommodationSchema |
| 2 | http://www.w3.org/2001/XMLSchema |
| 3 | http://www.ifs.uni-linz.ac.at/XRay/Supervision |

| XSDPrefix | | | |
|---|---|---|---|
| PrefixId | *NameSpaceId* | *SchemaDeclarationId* | Prefix |
| 1 | 1 | 1 | ac |
| 2 | 2 | 1 | |
| 3 | 3 | 1 | su |
| 4 | 2 | 2 | |
| 5 | 3 | 2 | su |
| 6 | 2 | 3 | |
| 7 | 1 | 3 | ac |

| XSDComplexDataType | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ComplexDataTypeId | Id | Abstract | Block | Final | Name | *Schema DeclarationId* | KindOf Type | KindOf TypeId |
| 1 | | | | | villageType | 1 | SC | 1 |
| 2 | | | | | addressType | 1 | CM | 1 |
| 3 | | | | | managerType | 3 | CM | 2 |
| 4 | | | | | managerType | 1 | CC | 1 |
| 5 | | | | | supervisorType | 2 | CM | 4 |
| 6 | | | | | phoneType | 1 | EC | 1 |
| 7 | | | | | | 1 | EC | 2 |
| 8 | | | | | | 1 | EC | 3 |
| 9 | | | | | saunaType | 1 | EC | 4 |
| 10 | | | | | | 1 | EC | 5 |
| 11 | | | | | descripionType | 1 | CM | 5 |
| 12 | | true | | | roomType | 1 | CM | 6 |
| 13 | | | | | accommodationType | 1 | CM | 7 |
| 14 | | | | | | 1 | CM | 8 |
| 15 | | | | | accAwardsType | 1 | CM | 9 |
| 16 | | | | | | 1 | CM | 10 |
| 17 | | | | | | 1 | EC | 6 |
| 18 | | | | | standardRoomType | 1 | CC | 2 |
| 19 | | | | | luxuryRoom | 1 | CC | 3 |

ComplexDataType – KindOfType:

CM ContentModel
SC SimpleContent
CC ComplexContent
EC EmptyComplexDataType

| XSDSimplePredefinedDataType | |
|---|---|
| SimplePredefinedDataTypId | Name |
| 1 | string |
| 2 | normalizedstring |
| 3 | token |
| 4 | language |
| 5 | name |
| 6 | ncname |
| 7 | nmtoken |
| 8 | nmtokens |
| 9 | id |
| 10 | idref |
| 11 | idrefs |
| 12 | entities |
| 13 | base64binary |
| 14 | hexbinary |
| 15 | boolean |
| 16 | float |
| 17 | double |
| 18 | decimal |
| 19 | integer |
| 20 | nonpoitiveinteger |
| 21 | negativeinteger |
| 22 | long |
| 23 | int |
| 24 | short |
| 25 | byte |
| 26 | nonnegativeinteger |
| 27 | unsignedlong |
| 28 | unsignedint |
| 29 | unsignedschort |
| 30 | unsignedbyte |
| 31 | positiveinteger |
| 32 | anyuri |
| 33 | qname |
| 34 | notaion |
| 35 | duration |
| 36 | datetime |
| 37 | time |
| 38 | date |
| 39 | gyearmonth |
| 40 | gyear |
| 41 | gmonthday |
| 42 | gday |
| 43 | gmonth |

| XSDSimpleContent | | |
|---|---|---|
| SimpleContentId | Id | *PrefixId* |
| 1 | | |

| XSDSimpleContentRestriction | | |
|---|---|---|
| *SimpleContentRestricionId* | Id | *RestrictsSimpleContentId* |

| XSDSimpleContentExtension | | | |
|---|---|---|---|
| *SimpleContentExtentionId* | Id | KindOfExtension | KindOfExtensionId |
| 1 | | SP | 1 |

CT ComplexType
SP SimplePredefinedDataType

| XSDComplexContent | | | | | |
|---|---|---|---|---|---|
| ComplexContentId | Id | ContentModelId | KindOfBase | KindOfBaseId | *PrefixId* |
| 1 | | 3 | CT | 3 | 1 |
| 2 | | 11 | CT | 12 | 1 |
| 3 | | | CT | 18 | 1 |

CT ComplexDataType
SP SimplePredefinedDataTypeId

| XSDComplexContent Restriction | |
|---|---|
| *ComplexContentRestricionId* | Id |

| XSDComplexContent Extension | |
|---|---|
| *ComplexContentExtentionId* | Id |
| 1 | |
| 2 | |
| 3 | |

| XSDContentModel | | |
|---|---|---|
| ContentModelId | KindOfContentModel | KindOfContentModelId |
| 1 | S | 1 |
| 2 | S | 2 |
| 3 | S | 3 |
| 4 | S | 4 |
| 5 | A | 1 |
| 6 | S | 5 |
| 7 | S | 6 |
| 8 | S | 7 |
| 9 | S | 8 |
| 10 | S | 9 |
| 11 | S | 10 |

A All
C Choice
S Sequence

| XSDAll | | | |
|---|---|---|---|
| AllId | Id | MinOccurs | MaxOccurs |
| 1 | | 0 | 1 |

| XSDChoice | | | |
|---|---|---|---|
| ChoiceId | Id | MinOccurs | MaxOccurs |

| XSDSequence | | | |
|---|---|---|---|
| SequenceId | Id | MinOccurs | MaxOccurs |
| *1* | | | |
| 2 | | | |

| | | | |
|---|---|---|---|
| *3* | | | |
| *4* | | | |
| *5* | | | |
| *6* | | | |
| *7* | | | |
| *8* | | | |
| *9* | | | |
| *10* | | | |

| XSDSequencePosition | | |
|---|---|---|
| *SequenceId* | *ContentParticleId* | Position |
| 1 | 3 | 1 |
| 1 | 4 | 2 |
| 1 | 5 | 3 |
| 2 | 8 | 1 |
| 2 | 9 | 2 |
| 3 | 11 | 1 |
| 4 | 14 | 1 |
| 4 | 15 | 2 |
| 4 | 16 | 3 |
| 5 | 28 | 1 |
| 5 | 29 | 2 |
| 5 | 30 | 3 |
| 6 | 2 | 1 |
| 6 | 7 | 2 |
| 6 | 13 | 3 |
| 6 | 18 | 4 |
| 6 | 19 | 5 |
| 6 | 20 | 6 |
| 6 | 21 | 7 |
| 6 | 22 | 8 |
| 6 | 23 | 9 |
| 6 | 24 | 10 |
| 6 | 27 | 11 |
| 6 | 33 | 12 |
| 7 | 35 | 1 |
| 7 | 36 | 2 |
| 8 | 37 | 1 |
| 9 | 34 | 1 |
| 9 | 38 | 2 |
| 10 | 40 | 1 |
| 10 | 41 | 2 |

| XSDAllElements | |
|---|---|
| *AllId* | *ElementId* |
| 1 | 21 |
| 1 | 22 |

| XSDContentParticle | | |
|---|---|---|
| ContentParticleId | KindOfContentParticle | KindOfContentParticleId |
| 1 | E | 1 |
| 2 | E | 2 |
| 3 | E | 3 |
| 4 | E | 4 |
| 5 | E | 5 |
| 6 | S | 1 |
| 7 | E | 6 |
| 8 | E | 7 |
| 9 | E | 8 |
| 10 | S | 2 |
| 11 | E | 9 |
| 12 | S | 3 |

| | | |
|---|---|---|
| 13 | E | 10 |
| 14 | E | 11 |
| 15 | E | 12 |
| 16 | E | 13 |
| 17 | S | 4 |
| 18 | E | 14 |
| 19 | E | 15 |
| 20 | E | 16 |
| 21 | E | 17 |
| 22 | E | 18 |
| 23 | E | 19 |
| 24 | E | 20 |
| 25 | E | 21 |
| 26 | E | 22 |
| 27 | E | 23 |
| 28 | E | 24 |
| 29 | E | 25 |
| 30 | E | 26 |
| 31 | S | 5 |
| 32 | S | 6 |
| 33 | E | 27 |
| 34 | E | 28 |
| 35 | E | 29 |
| 36 | E | 30 |
| 37 | E | 31 |
| 38 | E | 32 |
| 39 | E | 33 |
| 40 | E | 34 |
| 41 | E | 35 |

C Choice
S Sequence
E Element

| XSDChoiceContentParticle | |
|---|---|
| *ChoiceId* | *ContentParticleId* |

| XSDEmptyComplexDataType |
|---|
| EmptyComplexDataTypeId |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

SP SimplePredefinedDataType
CT ComplexDataType

| XSDIdentityConstraints | |
|---|---|
| IdentityConstraintsId | *ElementId* |
| 1 | 34 |

| XSDUnique | | |
|---|---|---|
| *UniqueId* | Id | Name |

| XSDKey | | |
|---|---|---|
| *KeyId* | Id | Name |
| 1 | | AccKey |

| XSDKeyRef | | | |
|---|---|---|---|
| *KeyRefId* | Id | Name | *KeyId (refer)* |
| 1 | | AwardWinnerRef | 1 |

| XSDSelector | | | | |
|---|---|---|---|---|
| SelectorId | Id | XPath | *KindOfMembership* | *KindOfMembershipId* |
| 1 | | ac:accommodation | K | 1 |
| 2 | | ac:awards/ac:award | R | 1 |

| XSDField | | | | |
|---|---|---|---|---|
| *FieldId* | Id | XPath | *KindOfMembership* | *KindOfMembershipId* |
| 1 | | @id | K | 1 |
| 2 | | ac:winner | R | 1 |

U Unique
K Key
R KeRef

| XSDElement | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ElementId | Id | Abstract | Block | DefaultValue | Final | Fixed | Form | MinOccurs | MaxOccurs | Name | Nillable | *TargetElementId(Ref)* | *SchemaDeclarationId* | Prefixld | KindOfElement | KindOfElementId | isRoot |
| 1 | | | | | | | | | | name | | | 1 | 1 | SP | 1 | 0 |
| 2 | | | | | | | | | | | | 1 | 1 | | | | 0 |
| 3 | | | | | | | | | | street | | | 1 | | SP | 1 | 0 |
| 4 | | | | | | | | | | village | | | 1 | 1 | CT | 1 | 0 |
| 5 | | | | | | | | | | country | | | 1 | | SP | 1 | 0 |
| 6 | | | | | | | | | | address | | | 1 | 1 | CT | 2 | 0 |
| 7 | | | | | | | | | | firstName | | | 3 | | SP | 1 | 0 |
| 8 | | | | | | | | | | lastName | | | 3 | | SP | 1 | 0 |
| 9 | | | | | | | | | | salary | | | 1 | | SP | 31 | 0 |
| 10 | | | | | | | | | | manager | | | 1 | 1 | CT | 4 | 0 |
| 11 | | | | | | | | | | supId | | | 2 | | SP | 31 | 0 |
| 12 | | | | | | | | | | firstName | | | 2 | | SP | 1 | 0 |
| 13 | | | | | | | | | | lastName | | | 2 | | SP | 1 | 0 |
| 14 | | | | | | | | | | supervisor | | | 1 | 5 | CT | 5 | 0 |
| 15 | | | | | | | | 0 | unbounded | email | | | 1 | | SP | 1 | 0 |
| 16 | | | | | | | | | unbounded | phone | | | 1 | 1 | CT | 6 | 0 |
| 17 | | | | | | | | 0 | | acceptsCreditCards | | | 1 | 1 | CT | 7 | 0 |
| 18 | | | | | | | | | | facilities | | | 1 | | CT | 8 | 0 |
| 19 | | | | | | | | | | sauna | | | 1 | 1 | CT | 9 | 0 |
| 20 | | | | | | | | 0 | unbounded | pool | | | 1 | | CT | 10 | 0 |
| 21 | | | | | | | | | | rating | | | 1 | | SP | 1 | 0 |
| 22 | | | | | | | | | | comment | | | 1 | | SP | 1 | 0 |
| 23 | | | | | | | | 0 | | description | | | 1 | 1 | CT | 11 | 0 |
| 24 | | | | | | | | | | roomNumber | | | 1 | | SP | 31 | 0 |
| 25 | | | | | | | | | | size | | | 1 | | SP | 31 | 0 |
| 26 | | | | | | | | | | nrOfBeds | | | 1 | | SP | 31 | 0 |
| 27 | | | | | | | | | unbounded | room | | | 1 | 1 | CT | 12 | 0 |
| 28 | | | | | | | | 0 | unbounded | accommodation | | | 1 | 1 | CT | 13 | 0 |
| 29 | | | | | | | | | | year | | | 1 | | SP | 40 | 0 |
| 30 | | | | | | | | | | winner | | | 1 | | SP | 1 | 0 |
| 31 | | | | | | | | 0 | unbounded | award | | | 1 | | CT | 14 | 0 |
| 32 | | | | | | | | | | accAwards | | | 1 | 1 | CT | 15 | 0 |
| 33 | | | | | | | | | | accommodations | | | 1 | | CT | 16 | 1 |
| 34 | | | | | | | | 0 | 2 | tv | | | 1 | | SP | 1 | 0 |
| 35 | | | | | | | | 0 | | shower | | | 1 | | CT | 17 | 0 |

| XSDAttribute | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AttributeId | Id | Default Value | Fixed | Form | Name | UseValue | *PrefixId* | *TargetAttributeId* | *SchemaDeclarationId* | KindOf Membership | KindOf MembershipId | *SimplePredefinedDataTypeId* |
| 1 | | | | | yearofFoundation | | | | 1 | SC | 1 | 40 |
| 2 | | | | | postalCode | required | | | 1 | CM | 1 | 1 |
| 3 | | | | | mgrId | | | | 1 | CC | 1 | 31 |
| 4 | | | | | since | | | | 1 | CC | 1 | 40 |
| 5 | | | | | number | required | | | 1 | EC | 1 | 1 |
| 6 | | | | | available | required | | | 1 | EC | 4 | 15 |
| 7 | | | | | id | required | | | 1 | CM | 7 | 31 |
| 8 | | | Austria | | state | | | | 1 | CM | 7 | 1 |
| 9 | | | | | hasBulter | required | | | 1 | CC | 3 | 15 |

ATTRIBUTE - *KindOfMembership*

CM ContentModel
EC EmptyComplexDataType
SP SimplePredefidedDataType
SC SimpleContend
CC ComplexContent

| XSDSchemaDeclaration | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Schema DeclarationId | Id | Attribute Form Default | Block Default | Element Form Default | Final Default | Version | XMLlang | Filename | *TargetNameSpaceId* |
| 1 | | | | qualified | | | | xray_accommodation.xsd | 1 |
| 2 | | | | qualified | | | | xray_supervision.xsd | 3 |
| 3 | | | | qualified | | | | xray_management.xsd | 1 |

## Relationen der RDB Schema - Komponente

| DBSchema | | | | |
|---|---|---|---|---|
| DBSchema | DBName | DBConnectString | DBUserName | DBPassword |
| accomDBSchema | lehre92 | 140.78.90.210:1521: | xray | xray |

| DBAttribute | | | | | |
|---|---|---|---|---|---|
| DBAttId | *DBSchema* | DBRelation | DBAttribute | DBIsKey | DBDataType |
| 1 | accomDBSchema | Accommodation | AccId | 1 | INT |
| 2 | accomDBSchema | Accommodation | Name | 0 | VARCHAR |
| 3 | accomDBSchema | Accommodation | Street | 0 | VARCHAR |
| 4 | accomDBSchema | Accommodation | State | 0 | VARCHAR |
| 5 | accomDBSchema | Accommodation | VillageName | 0 | VARCHAR |
| 6 | accomDBSchema | Accommodation | AcceptsCreditCards | 0 | BOOLEAN |
| 7 | accomDBSchema | Accommodation | Sauna | 0 | BOOLEAN |
| 8 | accomDBSchema | Manager | MgrId | 1 | VARCHAR |
| 9 | accomDBSchema | Manager | FirstName | 0 | VARCHAR |
| 10 | accomDBSchema | Manager | LastName | 0 | VARCHAR |
| 11 | accomDBSchema | Manager | Salary | 0 | INT |
| 12 | accomDBSchema | Manager | Since | 0 | INT |
| 13 | accomDBSchema | Manager | AccId | 0 | INT |
| 14 | accomDBSchema | Supervisor | SupId | 1 | VARCHAR |
| 15 | accomDBSchema | Supervisor | FirstName | 0 | VARCHAR |
| 16 | accomDBSchema | Supervisor | LastName | 0 | VARCHAR |
| 17 | accomDBSchema | Supervisor | AccId | 0 | INT |
| 18 | accomDBSchema | Room | RoomNr | 0 | INT |
| 19 | accomDBSchema | Room | RoomSize | 0 | INT |
| 20 | accomDBSchema | Room | NrOfBeds | 0 | INT |
| 21 | accomDBSchema | Room | HasShower | 0 | BOOLEAN |
| 22 | accomDBSchema | Room | AccId | 0 | INT |
| 23 | accomDBSchema | Room | RoomId | 1 | INT |
| 24 | accomDBSchema | LuxuryRoom | HasButler | 0 | BOOLEAN |
| 25 | accomDBSchema | LuxuryRoom | RoomId | 1 | INT |
| 26 | accomDBSchema | TV | Description | 0 | VARCHAR |
| 27 | accomDBSchema | TV | RoomId | 1 | INT |
| 28 | accomDBSchema | Village | Name | 1 | VARCHAR |
| 29 | accomDBSchema | Village | PostalCode | 0 | INT |
| 30 | accomDBSchema | Village | CountryId | 0 | INT |
| 31 | accomDBSchema | Country | CountryId | 1 | INT |
| 32 | accomDBSchema | Country | Name | 0 | VARCHAR |
| 33 | accomDBSchema | History | VillageName | 1 | VARCHAR |
| 34 | accomDBSchema | History | YearFound | 0 | INT |
| 35 | accomDBSchema | Pool | AccId | 1 | INT |
| 36 | accomDBSchema | Pool | Name | 1 | VARCHAR |
| 37 | accomDBSchema | Phone | AccId | 1 | INT |
| 38 | accomDBSchema | Phone | Number | 1 | VARCHAR |
| 39 | accomDBSchema | EmailAddress | AccId | 1 | INT |
| 40 | accomDBSchema | EmailAddress | Email | 1 | VARCHAR |
| 41 | accomDBSchema | Description | DescriptionId | 1 | INT |

| | | | | | |
|---|---|---|---|---|---|
| 42 | accomDBSchema | Description | AccId | 0 | INT |
| 43 | accomDBSchema | Description | Rating | 0 | INT |
| 44 | accomDBSchema | Description | Comment | 0 | VARCHAR |
| 45 | accomDBSchema | AccAwards | Winner | 1 | INT |
| 46 | accomDBSchema | AccAwards | Year | 1 | INT |

| DBJoinSegment | | | | | | |
|---|---|---|---|---|---|---|
| DBRelShipId | *DBAtt1* | *DBAtt2* | *DBRelation1* | *DBRelation2* | *DBSchema* | DBJoinDirection |
| 1 | Name | VillageName | Village | Accommodation | accomDBSchema | 12 |
| 2 | AccId | AccId | Phone | Accommodation | accomDBSchema | 21 |
| 3 | Winner | AccId | AccAwards | Accommodation | accomDBSchema | 21 |
| 4 | AccId | AccId | EmailAddress | Accommodation | accomDBSchema | 21 |
| 5 | AccId | AccId | Pool | Accommodation | accomDBSchema | 21 |
| 6 | AccId | AccId | Description | Accommodation | accomDBSchema | 21 |
| 7 | VillageName | Name | History | Village | accomDBSchema | 21 |
| 8 | CountryId | CountryId | Country | Village | accomDBSchema | 12 |
| 9 | AccId | AccId | Manager | Accommodation | accomDBSchema | 21 |
| 10 | AccId | AccId | Supervisor | Accommodation | accomDBSchema | 21 |
| 11 | AccId | AccId | Room | Accommodation | accomDBSchema | 21 |
| 12 | RoomId | RoomId | TV | Room | accomDBSchema | 21 |
| 13 | RoomId | RoomId | LuxuryRoom | Room | accomDBSchema | 21 |

## Relationen der Mapping – Komponente

| ElementMapping | | | | |
|---|---|---|---|---|
| XSDElementID | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 33 | ET_0 | NULL | NULL | 1 |
| 28 | ET_R | NULL | NULL | 1 |
| 2 | ET_A1 | 2 | NULL | 1 |
| 6 | ET_0 | NULL | NULL | 1 |
| 3 | ET_A1 | 3 | NULL | 1 |
| 4 | ET_A2 | 28 | 1 | 1 |
| 5 | ET_A2 | 32 | 8 | 1 |
| 15 | ET_A2 | 40 | 4 | 1 |
| 16 | ET_R | NULL | 2 | 1 |
| 17 | ET_A1 | 6 | NULL | 1 |
| 18 | ET_0 | NULL | NULL | 1 |
| 19 | ET_0 | NULL | NULL | 1 |
| 20 | ET_A2 | 36 | 5 | 1 |
| 23 | ET_R | NULL | 6 | 1 |
| 21 | ET_A2 | 43 | 6 | 1 |
| 22 | ET_A2 | 44 | 6 | 1 |
| 10 | ET_R | NULL | 9 | 1 |
| 7 | ET_A2 | 9 | 9 | 1 |
| 8 | ET_A2 | 10 | 9 | 1 |
| 9 | ET_A2 | 11 | 9 | 1 |
| 14 | ET_R | NULL | 10 | 1 |
| 11 | ET_A2 | 14 | 10 | 1 |
| 12 | ET_A2 | 15 | 10 | 1 |
| 13 | ET_A2 | 16 | 10 | 1 |
| 32 | ET_0 | NULL | NULL | 1 |
| 31 | ET_R | NULL | 3 | 1 |
| 29 | ET_A2 | 46 | 3 | 1 |
| 30 | ET_A2 | 45 | 3 | 1 |
| 27 | ET_R | NULL | 11 | 1 |
| 24 | ET_A2 | 18 | 11 | 1 |
| 25 | ET_A2 | 19 | 11 | 1 |
| 26 | ET_A2 | 20 | 11 | 1 |
| 34 | ET_A2 | 26 | 12 | 1 |
| 35 | ET_A2 | 21 | 11 | 1 |

| AttributeMapping | | | | | |
|---|---|---|---|---|---|
| XSDElementID | XSDAttributeID | KindOfMapping | DBAttr | DBRelShipId | MappingId |
| 28 | 7 | A_A1 | 1 | NULL | 1 |
| 28 | 8 | A_0 | NULL | NULL | 1 |
| 6 | 2 | A_A2 | 29 | 1 | 1 |
| 4 | 1 | A_A2 | 34 | 7 | 1 |
| 16 | 5 | A_A2 | 38 | 2 | 1 |
| 19 | 6 | A_A1 | 7 | NULL | 1 |
| 10 | 3 | A_A2 | 8 | 9 | 1 |
| 10 | 4 | A_A2 | 12 | 9 | 1 |
| 27 | 9 | A_A2 | 24 | 13 | 1 |

| Mappings | | | | |
|---|---|---|---|---|
| MappingId | SchemaDeclarationId | RootElementId | MappingVariant | DBSchema |
| 1 | 1 | 33 | Accommodation1 | accomDBSchema |

| AbstractTypeKeys | | | |
|---|---|---|---|
| DBRelation | DBKeyAttr | DBKeyValue | HeritageType |
| Room | RoomId | 1 | standardRoomType |
| Room | RoomId | 2 | luxuryRoomType |
| Room | RoomId | 3 | standardRoomType |

| ElementBaseRelation | | |
|---|---|---|
| MappingId | ElementId | DBRelation |
| 1 | 28 | Accommodation |

## Relationen zum Speichern der Daten von accommodation.xml

| Accommodation | | | | | | |
|---|---|---|---|---|---|---|
| AccId | Name | Street | State | *VillageName* | Accepts CreditCards | Sauna |
| 45 | Steigenberger MAXX Hotel Linz | Am Winterhafen 13 | Austria | Linz | true | false |
| 46 | Hotel Wolf-Dietrich | Wolf-Dietrich-Straße 7 | Austria | Salzburg | true | true |

| Manager | | | | | |
|---|---|---|---|---|---|
| MgrId | FirstName | LastName | Salary | Since | *AccId* |
| 1 | Karin | Maier | 20000 | 1995 | 45 |
| 2 | Klaus | Klammer | 30000 | 1985 | 46 |

| Supervisor | | | |
|---|---|---|---|
| SupId | FirstName | LastName | *AccId* |
| 1 | Herman | Maier | 46 |
| 2 | Hans | Klammer | 45 |

| Room | | | | | |
|---|---|---|---|---|---|
| RoomId | RoomSize | NrOfBeds | HasShower | *AccId* | RoomNr |
| 1 | 25 | 2 | true | 45 | 101 |
| 2 | 95 | 3 | false | 46 | 301 |
| 3 | 33 | 2 | false | 45 | 105 |

| LuxuryRoom | |
|---|---|
| *RoomId* | HasButler |
| 2 | true |

| TV | |
|---|---|
| *RoomId* | Description |
| 1 | Panasonic TX-32PS11D |
| 2 | Sony KV-32FQ85 |

| Village | | |
|---|---|---|
| Name | PostalCode | *CountryId* |
| Linz | 4020 | 1 |
| Salzburg | 5020 | 2 |

| Country | |
|---|---|
| CountryId | Name |
| 1 | Oeberoesterreich |
| 2 | Salzburg |

| History | |
|---|---|
| *VillageName* | YearFound |
| Linz | 1600 |
| Salzburg | 1500 |

| Pool | |
|---|---|
| *AccId* | Name |
| 45 | Pool1 |
| 46 | Pool1 |
| 46 | Pool2 |

| Phone | |
|---|---|
| *AccId* | Number |
| 45 | +4373278990 |
| 46 | 0043 662 /87 12 75 |

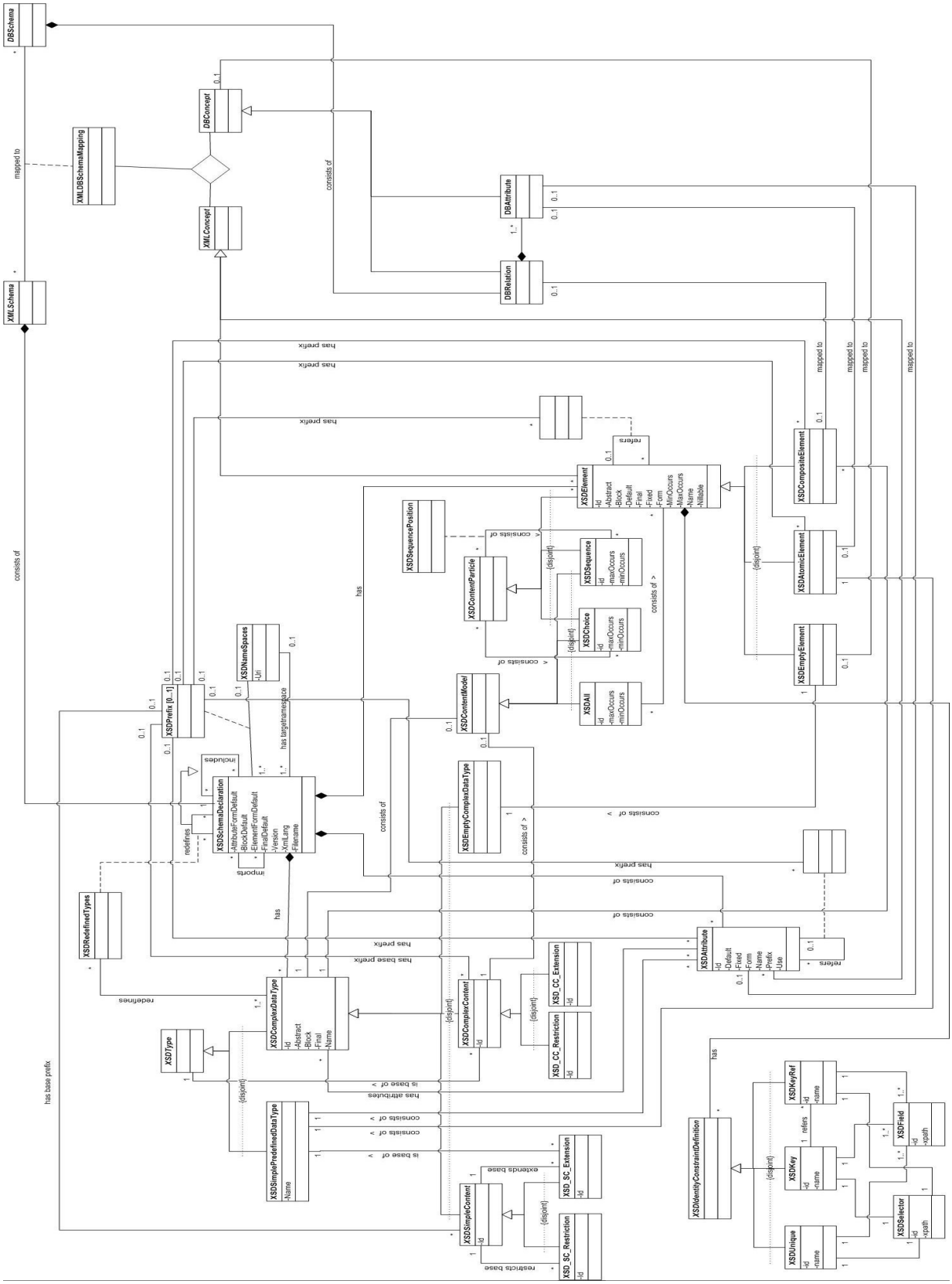
| EmailAddress | |
|---|---|
| *AccId* | Email |
| 45 | service.linz@maxxhotel.at |
| 46 | office@salzburg-hotel.at |

| Description | | | |
|---|---|---|---|
| DescriptionId | *AccId* | Rating | GuestComment |
| 1 | 45 | 3 | noisy |
| 2 | 45 | 2 | nice rooms |
| 3 | 46 | 1 | beautiful |
| 4 | 46 | 1 | very nice landscape |

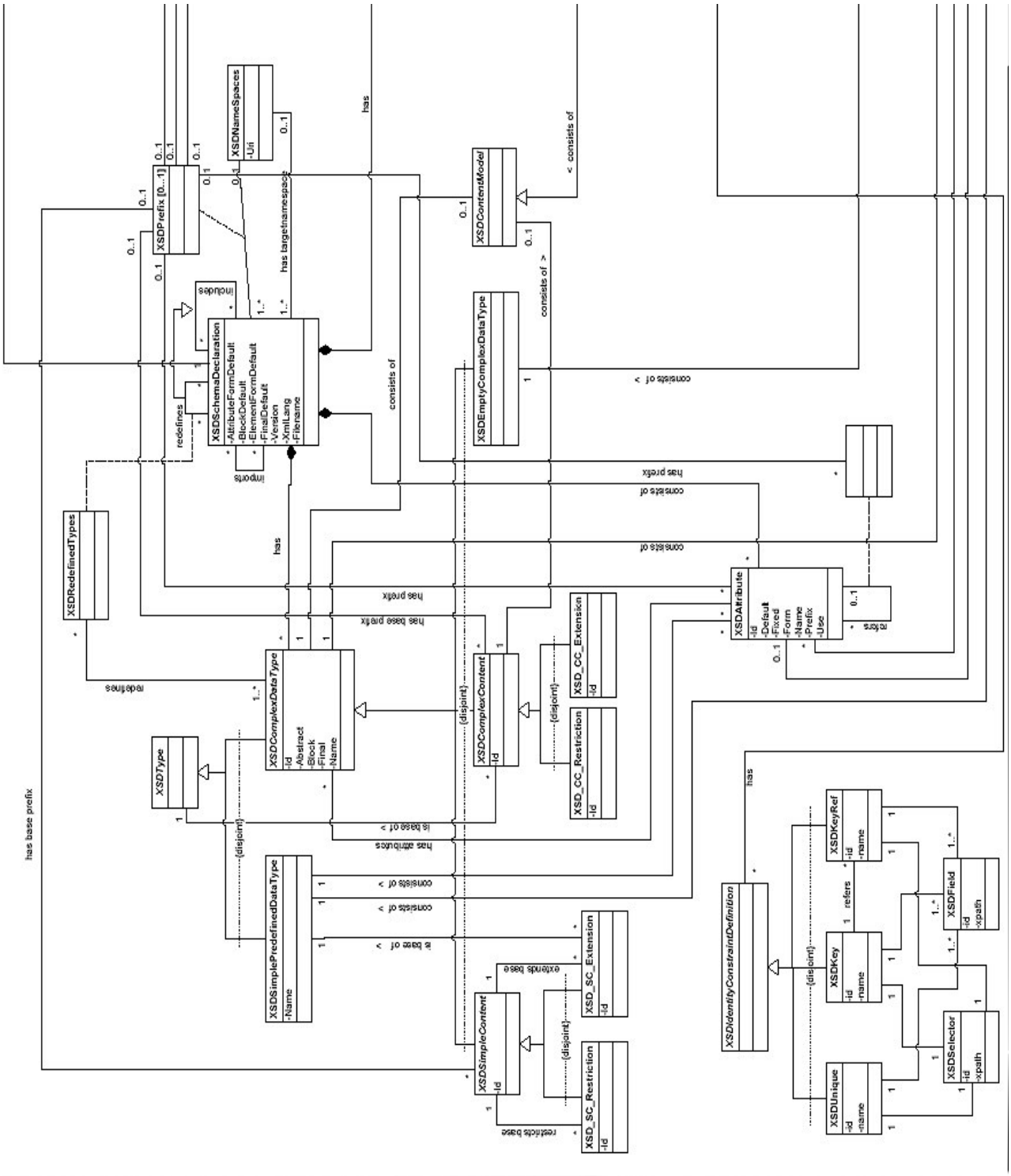
| AccAwards | |
|---|---|
| *Winner* | Year |
| 46 | 2003 |
| 45 | 2004 |

# Anhang C: UML Diagramme

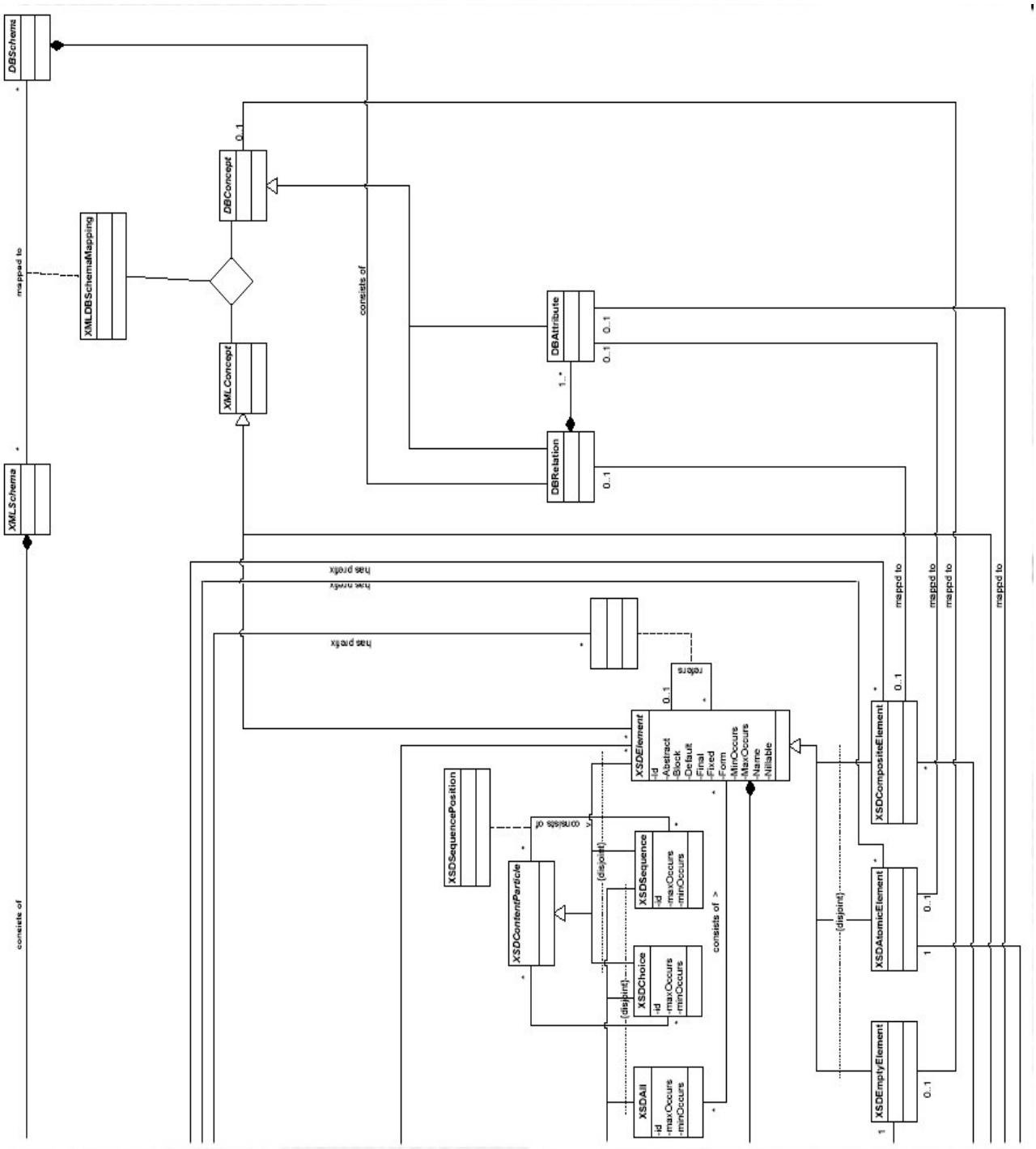
## UML Diagramm des Metaschemas von X-Rayxs - Gesamtansicht



## UML Diagramm des Metaschemas von X-Rayxs Teil 1/2



## UML Diagramm des Metaschemas von X-Rayxs Teil 2/2



## UML Diagramm des relationalen Schemas