A Schema-based XML Database Storageadapter

Claus P. Priese

Datenbanken und Informationssysteme
FB15, Institut für Informatik
J.W.G. Universität Frankfurt/Main
Robert-Meyer-Str. 11-15
D-60486 Frankfurt/Main
priese@dbis.informatik.uni-frankfurt.de
and cspriese@gmx.de

Abstract: After a short motivation within this paper a concise view on the basic concepts of XML-Schema, the Java architecture for XML Binding, Java Data Objects, the Universal Free Object-Relational DataBase adapter and the idea of a conceptual Object-Relational Schema is given. After this, persistency concerns for XML data are threatened in general. Then a mathematical well-founded mapping between graphs representing XML-Schema components and OR-Schema components is given by defining a graph-isomorphism. Finally the extended UFX-RDB adapter architecture and interfaces are depicted in short.

1 Introduction

Today`s choice for building innovative new informationsystems and the integration of existing destributed softwarecomponents appear to be webservices. Webservices provide a standardized technical plattform for establishing asynchronous interoperation and coupling of distributed parts of software, possibly over the worldwide web. Moreover dependend on the targeted application-domain of an informationsystem different system-design methodologies and achitectural alternatives may appear rational or approriate. This means webservices can be interconnected for example according to workflow-processdefinitions, by using semantic informations located somewhere in the web or by any other arbitrary composition-paradigm.
In all cases data is processed. System-metadata for running the informationsystem itself properly and user-/applicationdata "inside" the systems to fulfill the intended application-purpose. Even for the communication between distributed softwareparts (the services) data (the messages) is exchanged over networks.
Common chacteristics of various kind of data are the use of an appropriate encoding-language and usually the appearance in conjunction with structural descriptions. Further non-volatile data should be stored in a database-system for save and durable persistence and accessability.
Not only, but also in the context of webservices the data-encoding-language and the structural description language of choice is the extensible markup language (XML) [XML00] and XML-Schema [XSD-P0-01], [XSD-P1-01], [XSD-P2-01].

Therefore in this paper the universal free XML to relational database storage adapter (UFX-RDB) is introduced in section 7. But in front of doing so a short review of the related and required technologies and specifications is given. This is done to enable the reader to understand the relationships, ideas and concepts as used in the UFX-RDB adapter. Further to recognize the neccessity and thus appraise the work done with the UFX-RDB adapter.

2 XML and XML-Schema

As mentioned above data is encoded with an encoding language and usually appears in conjunction with structural descriptions. The encoding language used here is XML and the language used for the structural descripiton is XML-Schema. Therefore in this context XML-Schema is a *meta-language* for describing the grammar and vocabulary of another language. For example the widley know HyperTextMarkupLanguage (HTML) could be specified within one XML-Schema. The same is true for any other XML-Language. And because HTML and any other XML-Languages are itself *markup-languages* XML-Schema is moreover a *meta-markuplanguage* because it is a language for defining other markup-languages and is also decribing the structure and content of a class of XML-Documents. Futher it may be of interrest to be mentioned that XML-Schema itself is defined in XML instead of using an own language as done with DocumentTypeDefinitions (DTD). Therefore any XML-Schema always is a valid XML document with a cyclic-free[1] tree-structure. The schema-for-schemas is specified in [XSD-P1-01].
The development and introduction of XML-Schema was also caused by the poor data-typing- and data-encapsulation-capabilities of the predecessor XML in conjunction with DTDs. The main improvements of XML-Schema are:

- a rich set of primitive datatypes including all usual types and the additional possibility to specify the domain of types
- definition of complex elementtypes by composition of existing primitive and other complex types
- explicit grouping mechanism for attributes and elements
- definition of new elementtypes on basis of existing ones (inheritance) with extension and restriction mechanisms
- support for namespaces

Through the use of namespaces a XML-Schema can contain elements definded in other XML-Schemata or can contain different elements with identical name inside one XML-Schema if they exists in different namespaces. By this the use of namespaces supports rational reuse and data-encapsulation as known simmilar from object-oriented programming languages. For an short introduction into XML namespaces see [Bi01] pp.106-118 .

---

[1] In a preliminary version of the XML-Schema specification there was a specification-gap allowing unintentionally cycles with XML-Schemata. The former master thesis student A.A.Baker and the author of this paper therefore made a corresponding hint to the XML-Schema standardization group to close this gap.

Now follows a concise look on selected details of the relevant parts of the XML-Schema specification as need for section 3, 6 and 7.
Any XML-Schema is constructed of up to twelve components/blocks that make up the the abstract data model of the schema. According to [Bi01] they can be split in three groups which appear to be choosen well and are therefore adopted here:

- **Primary components**: element and attribute declarations, simple type and complex type definitions
- **Secondary components**: attribute groups, identity constraints, model group definitions and notation declarations
- **Helper components**: these are different form the first two, because they cannot be named or independently accessed

The basic building blocks of any XML document are elements and it`s attributes. Therefore declarations of elements with attributes are also the basic components for XML-Schema. While gobal elements are declare within the outermoust schema-element of an XML-Schema local ones are declared inside a complex type or element group. The name and type of an element is specified in attributes. Therefore the basic function of an element declaration is the association of its name with a type. A simple example could look like :

```
<xsd:element name=”amount” type=”xsd:float” />
```

If no type-attribute is specified the default type string is used. The prefix xsd: indicates that element and the typename float come from the W3C XML-Schema namespace which is declared in the root element of an XML schema (<xsd:schema xmlns:xsd=...>). Elements further can contain the attributes minOccurs and maxOccurs not explained in more detail here. Another property of elements is, that they can be subject of substitution with a substitution group. If substitution is used all participants have to be declared globally. Then the substitution group needs to be as of the same type as the replaced element or of a derived type of them. Substitution is transitive but not reflexive.

As primary components, beside elements and attributes, there are two kinds of typedefinitions in the XML-Schema language: primitive type definition and complex type definitions. As expected, the differnce is, that only complex type definitions can contain child elements in their definition while simple types can not. According to this two kinds of types there exist two root-types *anySimpleType* and *anyType* for inheritance by *extension* or *restriction*.
The XML-Schema specification contains 45 built-in data-types (20 primitive and 25 standardized derived types) which can be used as basis for the definition of user specific simple types. Simple types are defined bye using the *simpleType*-Element and one of three methods available for creating user defining simple types: atomic with restrictions, list-creation and union`s. An example for a list-type is:

```
<xsd:simpleType name=”listofnumbers” >
      <xsd:list itemType=”xsd:integer”/>
</xsd:simpleType>
```

In XML-Schema complex types are the means for building content models by adding child-elements and attributes to complex type definitions. A simple example for a complex type definition is:

```
<xsd:complexType name="size">
        <xsd:sequence>
                <xsd:element name="height" type="xsd:float" />
                <xsd:element name="width" type="xsd:float" />
        </xsd:sequence>
</xsd:complexType>
```

A complex type definition may contain one of three compositors anonymously(i.e. without explicit naming of the compositor-structure) or a reference to a gobally defined named group containing one compositor. The tree available compositors are: all, choice and sequence. The use of groups is usefull in cases where the same group of elements is needed in different places. The compositors itself can contain elements of primitive or again complex type. Exactly this is the mechanism used for building content models.

Beside elements also attributes can be defined in global groups for identical use in multiple types.

All three identity components provided from the XML-Schema language (ID, IDREF and key/keyref) does NOT match directly with the *XML persistent document identity* mechanism used in UFX-RDB and described in a later section. Therefore their exsitance is mentioned only here. A description of this identity-constructs takes at least 2 pages. See [Bi01] p. 216-218 if interrested in more details.

Further the possibility needs to be metioned to embed a complete XML-Schema into another by means of the *include*-directive, or embed parts of another XML-Schema by means of the *import*- or *redefine*-directives. The include-directive is used in the UFX-RDB adapter for the important task of importing an identity-sturcture XML-Schema.

Even if not all available constructs of the XML-Schema specification could be covered, it should be enough to comprehend the meaning of the following sections.

3 The Java Architecture for XML Binding (JAXB)

Webservices and related informationsystem design rationals are not the topic of this paper but they are the intended application-environment to be used with the Java Architecture for XML Binding[2][JAX02]. Therefore it is neccessary to be at least aware of it also for our aimed XML persistency mechanism. The thight integration with

[2] Do not mistake the JAXB for the Java API for XML processing (JAXP)! These are different specifications! JAXB is something new, different from the older JAXP.

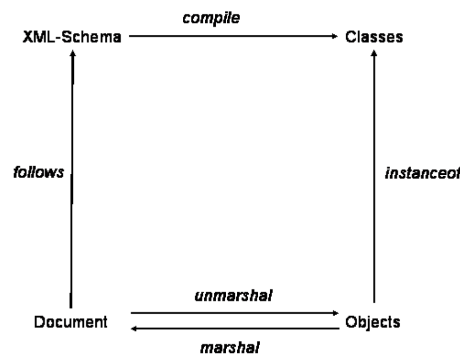
webservices can also be seen from the fact that Sun bundles the JAXB now with its latest Java Webservice Deveolper Pack (1.1)[3].

The JAXB specification standardize how to access and use a XML document (a file containing XML-tagged data) using the Java programming language, without considering any persistency requirements. But JAXB is aware of XML-Schema as the structural description language for the data contained in XML documents.
Moreover JAXB introduces a new API for reading, writing and manipulating of XML documents by means of the Java programming language while avoiding the tedious and error-prone use of the lowlevel SAX parser API or the somewhat higher-level DOM parse-tree API.

The Binding Framework

In general the JAXB specification maps the components of an XML document to in-memory objects instanciated from specifically created java-classes. JAXB uses the expressions of *Unmarshalling,* which means the process of reading an XML document and constructing a tree of content objects, *Marshalling,* which is the inverse of unmarshalling and *Validation,* which means the process of verifying that all constraints expressed in the source schema hold for a given content tree.

But prior to execution of any un-/marshalling or validation by the JAXB binding framework, the framework needs to know how to do this for any XML-Schema. Therefore always a *binding compiler* must be used to "bind" a XML-Schema to a set of content Java language constructs (i.e. interfaces+classes). This binding is described by a JAXB *binding language,* enabling also additional customization of this mapping where required. The interconnection of the mentioned parts can be seen from picture 1 below.



picture 1 – JAXB roundtrip overview

[3] WSDK can be downloaded fom Sun`s website http://java.sun.com

The JAXB specification introduces two new java-packages containing application programming interfaces and classes representing the JAXB. Implementations of the JAXB must fulfill these interfaces.

The *JAXBContext* class provides the client's entry point to the JAXB API. It holds references to the XML/Java binding information necessary for managing it and contains methods for the creation of instances of the interfaces *Marshaller*, *Unmarshaller* and *Validator*.
The interfaces metioned until now suggests how the content of an exsting XML document can be filled into a properly created tree of Java object instances. But it does not suggest how the opposite direction works. In fact a Java-application can create an XML document in two ways using a JAXB implementation:
The first alternative is that the application itself creates a tree of Java objects by instanciation from these Java classes previously generated from the binding compiler for a XML-Schema and afterwards pass this object-tree (representing the document) to an implementation of Marshaller interface.
The second, and for our purpose more important alternative, is the use of a factory mechanism. Each binding generated for a XML-Schema from the binding compiler contains a class *ObjectFactory*. This Factory class contains methods to generate objects for each of the schema-derived interfaces and classes. This means that any application can use a standardized mechanism using the constant ObjectFactory class instead of instanciating java objects itself from always different java classes derived from the actually used XML-Schema for the actual application.

JAXB concepts for XML to Java mapping

The Java Architecture for XML Binding separates the conceptual mapping from implementations by the use of java interfaces. The components of XML-Schema are mapped to java interfaces making up the content-model. Its a matter of a JAXB implementation to provide/generate implementing classes fulfilling these interfaces. We will see that this especially usefull for our aims to store the data contained persistent.

Now follows a concise look on the basic mappings from the basic parts of the XML-Schema to a Java representation as specified by the JAXP.

Spoken simple, the output of a JAXB binding compilers run is a Java package representing the XML-Schema. The name of the package ist either derived directly from the XML namespace URI, or specified by a binding customization. The package is at least made up of a set of Java *content* interfaces, a set of Java *element* interfaces and an *ObjectFactory* class. While the content interfaces represent the content model(i.e. complex types) declare in the XML-Schema, the element interfaces represent the elements declared in the XML-Schema.

Simple types and attributes either of a type or an element when declare in a XML-Schema are mapped to java properties (i.e. attributefields) in content or element

interfaces. This is always done in conjunction with a set of access functions (i.e. setter/getter methods).
As specified [JAX02] several property models can be used to represent the different simple types possible in a XML-Schema.
All java properties must have a base type, which may be a Java primitive type (*e.g.*, int) or a reference type according to the XML-Schema built in type or user derived type within the considered XML-Schema.
On the lowermost level JAXB specifies how the 45 XML-Schema built in data-types are mapped to primitive Java types. For example xsd:string is mapped to java.lang.String, xsd:integer is mapped to java.math.BigInteger, xsd:int is mapped to int, xsd.long is mapped to long ... see [JAX02] section 5.2.2 Atomic Datatypes.

Further, the JAXB specifications says a model(type) group definition is not bound to a Java content interface. Rather, when a named model group is referenced, the JAXB property set representing its content model is aggregated into the Java content interface representating the complex type definition that referenced the named model group definition.
Also attribute groups are spread the same way. As referenced, each attribute in the attribute group definition becomes a part of the referencing complex type definition.

4 Java Data Objects (JDO)

The Java Data Objects specification [JDO02] standardize a pure object-oriented database access mechanism and it`s embedding in the Java-Programming-Environment.[4]
The JDO specification does contain several parts describing neccessary aspects of an object-oriented database integration into Java. In this section we will concentrate on the fundamental aspects required for understanding the corresponding parts of the database adapters proposed in section 5.

Object Model, JDO Identity and Object States

The JDO specification distinguishes transient java programming language objects from persistent capable objects. The used object model split the set of persistent capable objects into two subsets. First Class Objects (FCS) and Second Class Objects (SCO). The main difference is, that first class objects possess a persistent object identity (POID), while second class objects does not. SCO`s doesn`t have own POID`s because they are always part of another FCS as attributes or in the inheritance-hierarcy.
Persistent object identity differs from the in memory volatile object instance identity which is assigned from the Java Virtual Machine at runtime. JDO specifies three alternatives for persistent object identity (POID): application identity, data store identity and nondurable identity. The default object identity mechanism used in the UFO-RDB adapter described in section 5 cornsponds to JDO data store identity. Which means, the

[4] JDO seems to be inspired by the ObjectDatabaseManagementGroup`s (ODMG) Object Database Standard3.0 with specialized Java-binding

POID is managed by the used data store without being tied to any field values of a JDO persistent capable object instance.
Persistent capable objects can be in one of the following states, according to their actual livecycle-phasis. Required states are transient, persistent-new, persistent-dirty, hollow, persistent-clean, persistent-deleted and persistent-new-deleted. Optional are the states nontransactional, persistent-nontransactional, transient-transactional, transient-clean, and transient-dirty. A detailed description of the meaning of this states is given in the JDO specification [JDO02] section 5.5 to 5.8 and is therefore omitted here.

Class Enhancer

Before being able to store instances of java classes in a persistent storage the JDO specification requires that the classes are enhanced. Enhancement means adding of attributes and code to enable the persistency mechanism to correctly manage the instances and it`s data content. There is (intentional) nothing said in the JDO specification about how this enhancement needs to be done. Therefore all possiblilities are available: source code extension, java-bytecode modification or perhpas a sophisticated on the fly modificication of instances on runtime whenever needed. While the first alternative would be the most trivial to implement it isn`t an elegant choice because of it`s code-intrusion visible also for the application programmer. Therefore the seconde approach of a java bytecode enhancer is choosen as the primary on in the UFO-RDB adapter of section 5.

Interfaces and their interconnection

An important part of the JDO specification is the introduction of a set of application programming interfaces accompanied with a description of their meaning.
The interface *PersistenceCapable* is the one which java classes whose object instances are intended to be stored in a persistent data store needs to implement. The implementation for this interface is added to java classes usually by the class enhancer and consists mainly of fields and generic methods for data content management.
The interface *StateManager* is “the other end” during runtime. This means an implementation of StateManager interacts with persistent capable object instances during runtime through the specified interface methods to manage the data content and object state correctly.
Before being able to manage a persistence capable object instance at runtime it has to be created or read form the underlying persistency mechanism. This is the task of an implementation of interface *PersistenceManager*. A PersistenceManager encapsulates the details of connection(s) to the used underlying enterprise information system (EIS), which can be directly a database management system or an even more complex persistent storage mechanism. To fulfill this tasks a PersistentManager implementation uses a vendor specific so called ResourceAdapter to create, store, retrieve, update or delete data objects in the underlying EIS. After doing this it passes the further handling of created JDO`s to StateManagers for runtime management.
PersistenceManager`s itself are created form implentations of the interface *PersistenceManagerFactory*. This factories contains or reads in from local configuration

files or retrieve from other resources informations about framework details like exact hosts and ports to use and which additional librarys to use etc.
Other interfaces found in the JDO specification are omitted here because of it`s weaker importance for the subject of this paper.

5 The UF O-R DataBase adapter

The UFO-RDB adapter, first introduced in [Pr99] and described more detailed in [Pr00], is an universal and highy customizable object-relational storage architecture framework for orthogonally object persistence [AM95] on the Java plattform. It does support the storage and retrieval of object-data by allowing Java applications to be programmed against a pure object-oriented API while using an arbitrary storage mechanism on the bottom. For example a JDBC-Driver[5] for the connection to a relational style database management system. For more details on the relational database model see [Co90].

Due to [Pr00] and [Pr99] already contains a description of the basic UFO-RDB architecture layers only the small changes induced from adding a light adapter-implementation, connecting the JDO-interfaces to the underlying UFO-RDB API and implementation, is shown in **picture 3**.
The advancements in the object-part are the introduction of a JDO compliant Java class enhancer and some changes in the functionality, boundaries and naming of the upper layers.

In the context of JDO the UFO-RDB adapter can be seen as a ResourceAdapter for relational DBMS.

Object-Relational Schemes and CCI Graph

For the subject of this paper the new innovative idea of an "object-relational schema" as also published first in [Pr00] is important and therefore explicated here again in brief.
According to (D1) [Pr00] an **object-relational schema** (OR-Schema) is definded as quadruple $S_j = < C(S_j), R(S_j), H(S_j), M(S_j) >$ where:

- $C(S_j) = \{c_{1j}, \ldots, c_{nj}\}$ is a finite set of persistent capable classes; forming the first half of the object schema part of the object-relational schema;
- $R(S_j) = \{r_{1j}, \ldots, r_{pj}\}$ is a finite set of tables; forming the relational schema part of the object-relational schema;
- $H(S_j) = \{h_{1j}, \ldots, h_{qj}\}$ is a finite set of hierarchy descriptors; forming the other half of the object schema part of the object-relational schema;
- $M(S_j) = \{m_{1j}, \ldots, m_{sj}\}$ is a finite set of mapping rules,
  with (naturally) $C(S_j) \cap R(S_j) \cap H(S_j) \cap M(S_j) = \emptyset$.

[5] JDBC stand for Java Data Base Connectivity and is an API and implemented mechanism for dynamic relational database access included in the Java 2 Plattform Standard Edition

As already done informally in [Pr00] there can be used several graphs[6] in association with (D1) to describe the structure securely on a mathematical stable basis. The one graph most important for an exact understanding of the type-mapping described in the next section is the directed complex class-interconnection graph as defined in the following:

(D3) A directed **complex class-interconnection graph** for an OR-Schema S_j is a triple

$CCIG(S_j) = (Cc(S_j), Me(S_j), vc_{Sj})$ where:

- $Cc(S_j) = \{Cc_{1j}, \ldots, Cc_{nj}\}$ is a finite set of nodes given by the set of persistent capable classes; from the object schema part of the object-relational schema S_j;
- $Me(S_j) = \{m_{1j}, \ldots, m_{sj}\}$ is a finite set of edges given by the subset of complex mapping rules of an object-relational schema S_j;
- vc_{Sj} is the (interconnection) function which does map the edges in Me onto orderend pairs out of Cc .

Note that this directed CCI Gaph as defined here is on a high abstraction level (or seen from the other end: not very fine-grained) and is intended only to be a means to demonstrate the principles. It does not contain explicit informations about complex attribute names, primitive types, inheritance or references.

6 XML persistency concerns

In this section mapping concerns and strategies are treated in general on an abstract level. After this is done, details of how an implementation looks like are given, in the next section.

XML persistency options

One of the fundamental decisions which have to be made, when persistent storage of XML documents valid under a XML-Schema in a database is required, is: which kind of database to use? The answer depends on the objectives of the designed application or informationsystem.
A specialized XML-database like Tamino[Ta03] seems to be the best choice in case that no mapping should be used and highest possible performance is required. The disadvantages are it`s proprietarity, it`s licensing-costs and the low persitent data portability.
If data-portability ist the most important aspect a mapping to relational structures probably would be the best choice, because relational database mgmt systems are the most widespread used DBMS. The disadvantage of this approach is the conceptual gap between the XML-Schema structure and relational data-structures. This would require a very mighty mapping-tool consuming probably huge efforts to develop and contains the danger to be very proprietary again because there is no standard or guideline on how to

[6] There is a large amount of literatur on graph-theory also in the english language. In this paper [Mu87] is used as reference.

do this correctly. Therefore this approach seems to be possible but impractically. (at least at the moment)
Therefore in this paper a third alternative is proposed for XML-schema structures to relational structures mapping by using an indirection through object-oriented structures in the middle. This approach is corroborated by the following advantages:

- First of all it appears natural, because when recalling the XML-schema typing system introduced in section 2 this looks very simmilar to ideas from object-oriented programming languages and therefore should be mutuable mappable.
- Further all major relational database management system vendors introduced object-features into their products guided by the SQL99-standard[ref needs to be inserted].
- And last but not least another advantage of this approach is that good concepts and mature implementations exists for object-relational mapping. For example the UFO-RDB adapter addressed in the previous section.

Fortunately the major part of the remaining XML-Schema to java-object-model mapping is done already by the JAXB Binding Framework. Therefore the remainder of this section will focus on the additional conceptual mechanisms neccessary for storing and retrieving the created java-objects, which contains the XML documents, with the UFX-R and UFO-R database adapters.

Adding XML persistent document identity (XPDI)

As explained already in section 4 an instanciated persistence capable object-structure (of arbitrary depth and complexity) always is assigned with and identified by a persistent object identifier. If a XML document is transformed into such object-structure, by means of unmarshalling of a JAXB implementation, it also can be identified by such POID when stored in a object-datastore. Later an application can retrieve an instanciated object-structure - containing the document - again using the assigned POID for manipuation or any other use.
But there is a need to lift the persistent identifier from the object level also up to the instance level of XML documents to enable any – possibly remote and distributed over a network, e.g. another webservice – application to identify a specific XML document by its unique id. For example an invoice would normally bear such unique identity.
Therefore the insertion of a corresponding XML persistency document identity (XPDI) structure is proposed here. The XPDI is included automatically into every instance of a XML document by means of implicit including a XPDI structure into every used XML-Schema. The small identity-structure XML-Schema always included corresponds to the used POID structure and looks like:

```
<?xml version”1.0” encoding=”UTF-8”?>
<xsd:schema xmlns:xsd=“http://www.w3.org/2001/XMLSchema” elementFormDefault=“qualified“>
        <xsd:element name=“XPDI“ type=“XPDIstructure“>
        <xsd:complexType name=“XPDIstructure“>
                <xsd:attribute name=”id” use=”required” type=”xsd:integer” />
                <xsd:attribute name=”typeName” use=”required” type=”xsd:string” />
                <xsd:attribute name=”segment” use=”required” type=”xsd:string” />
        </xsd:compexType>
</xsd:schema>
```

As you can see the complex type *XPDIstructure* does contain only attributes of primitive types specified in the XML-Schema language. These attributes are mapped onto java-class properties (i.e. attributes) in an class named XPDIstructure according to the JAXB specification and will then be stored/retrieved also as primitive typed class attributes by an JDO implementation (i.e. UFX-RDB with UFO-RDB) the same way as the rest of the document content. The content of the XPDI structure is set according to the content of the POIDs used and reverse by the UFX-RDB adapter during retrieval and storage of documents.
The inclusion of this identity structure XML-Schema is somewhat "instrusive" but should be acceptabel, expecting that using applications will ignore all additional parts of the resulting extended XML documents which they don't use. Ignoring is the standard behaviour know from other programming-area domains. And in the end this inclusion can be replaced and/or complemented by other identity handling mechanisms if wanted.

Mapping of XML-Schema strucures to OR-Schema structures

An instance of a XML document becomes a root based tree of object-instances after unmarshalling through a JAXB implementation. In doing so each of the interconnected object-instances fulfills either the content or element interface. The root of the object-instance-tree is an object representing the document itself. The same is true on the object-schema(i.e. class) level: the used classes - created from a JAXB implementation - for a XML document which validates against a XML-Schema are forming a root based tree. These classes forming the physical[7] object-model for the set of XML documents validating against a XML-Schema XS_j can be seen as a directed XML-Schema graph as defined in the following:

(D4) A directed **XML-Schema graph** for a XML-Schema XS_j is a triple

$XSG(S_j) = (Xc(XS_j), Xe(XS_j), vc_{XSj})$ where:

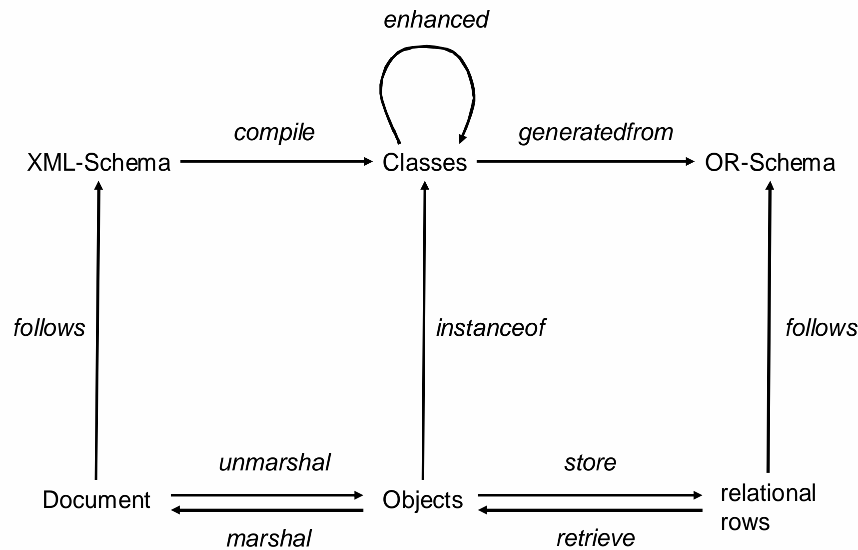
- $Xc(XS_j) = \{Xc_{1j}, \ldots, Xc_{nj}\}$ is a finite set of nodes given by the set of created classes from a JAXB implemention for a XML-Schema XS_{j};
- $Xe(S_j) = \{m_{1j}, \ldots, m_{sj}\}$ is a finite set of edges given by the subset of complex class relationships within the set of created classes from a JAXB implemention for a XML-Schema XS_j
- vx_{Sj} is the (interconnection) function which does map the edges in Xe onto orderend pairs out of Xc .

Informally it should be clear that a XSG can be "mapped" to a CCIG as given in (D3). More exactly the graphs defined (D3) and (D4) are isomorph. They can be "mapped" to each other by means of an XML2Object Isomorphism :

(D5) (h_c , h_m) with $h_c : Xc(XS_j)$? $Cc(S_j)$ and $h_m : Xe(S_j)$? $Me(S_j)$ and the property $vx(m) = (Xc_p, Xc_q)$? $vc(h_m(m)) = (h_c(Xc_p), h_c(Xc_q))$.

---

[7] In cotradiction to the more logical mapping to the content and element interfaces



picture 2 – extended roundtrip

Given this mapping from XML-Schema structures onto the object-part of an OR-Schema the required other parts of the OR-Schema can be generated by an object-relational mapping algorithm. Three of such standard algorithms according to [St99] are implemented already in the UFO-RDB adapter. The resulting extension of **picture 1** is shown in **picture 2**.

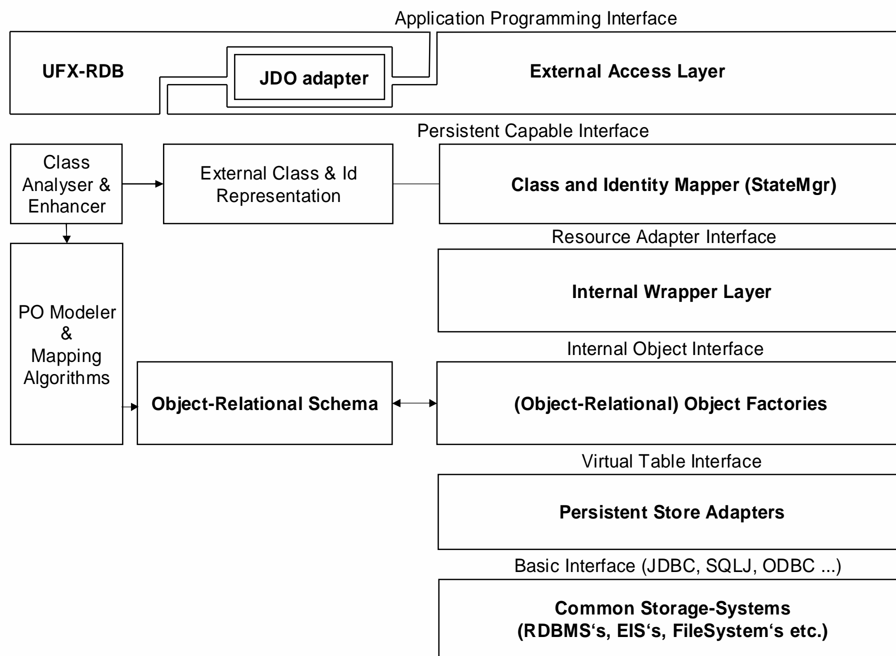
7 The UF X-R DataBase adapter

The Universal Free XML-Relational DataBase adapter (UFX-RDB) is the prototypical implementation of a XML storage adapter for relational databases using and incorporting in detail the ideas and concepts indicated in the previous sections.
In the context of JAXB the UFX-RDB adapter can be seen as a JAXB implementation enriched with facets required for persistency. Further the UFX-RDB is based on the light JDO branch of the UFO-RDB adapter.

Extensions and architecture

Simplified, the Universal Free XML–Relational DataBase adapter (UFX-RDB) is an addition XML-specific layer on top of the UFO-RDB adapter. It reuses all features of the

object-relational mapping mechanisms inside the UFO-RDB adapter and adds specific XML-functionality and an API extension specific for storing and retrieving XML-documents as you can see from **picture 3**.



picture 3 - UFO-RDB and UFX-RDB architecture

The shown archtitectur is very suitable for storing JAXB-unmarshalled XML documents because – as mentioned in section 3 – a JAXB-stye binding compiler always creates ObjectFactories for compiled XML-Schemas. This concept matches with the existing Objects-Factory concept used in the UF(O/X)-RDB architecture. This way enabling a smooth integration while preserving flexibility of customization if needed.

Interfaces

Because of the fact, that the UFX-RDB adapter stores and retrieves XML-documents in the form of instanciated object-structures, the application programming interface of the UFX-RDB adapter looks very simmilar to the one of the pure UFO-RDB adapter. The main API is (again) *DataBase* but containing now methods for storing, retrieving, updating, name-assigning, querying etc XML documents. Herein special methods for retrieving and handling of a XPDI structure from a document are included. Further a *SchemaManager* is introduced newly to assist the transformation of a XML-Schema into the required OR-Schema structures.

8 Conclusion and Outlook

In this paper a short view on the basic ideas and concepts of XML-Schema, the Java architecture for XML Binding, Java Data Objects, the Universal Free Object-Relational DataBase adapter and the idea of a conceptual Object-Relational Schema was given. After this, persistency concerns for XML data were mentioned in general and an isomorphism was definded between the graph of the JAXB-created object-representation of basic XML-Schema components and an graph of the object part of OR-Schema components to show the possibility of mapping. Finally the extended UFX-RDB adapter architecture and interfaces was shown.
While the defined graphs and the given isomorphism shows the fundamentals of a mathematical founded mapping, the given descripitions are very concise because of limited space in this paper. It would be of interrest to extend the given structures to include more details.

References

[AM95] Atkinson, M.P. and Morrison, R.: Orthogonally Persistent Object Systems. VLDB Journal, 4(3), 1995, pp319-401

[Bi01] Birbeck, M. et. al.: Professional XML 2nd Edition. Wrox Press Ltd, Birmingham, UK, 2002. p. 197

[Co90] E.F. Codd: The relational model for database management. version 2. Addison-Wesley Pub. Inc, 1990

[JAX02] The Java Architecture for XML Binding (JAXB); Proposed Final, V 0.90, December 2002, Editors: Joseph Fialli. Sekhar Vajjhala, Sun Microsystems Inc., Santa Clara, USA, 2002.

[JDO02] JavaDataObjectExpertGroup; Specification Lead: Craig Russel: Java Data Objects. Version 1.0 Sun Microsystems Inc., Palo Alto, U.S.A., 2002.

[Mu87] Müller, H.: Diskrete Algebraische Strukturen. Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung, Band 20, Nr. 5, Erlangen, 1987.

[Pr99] Priese, Claus P. A Flexible Type-Extensible Object-Relational DataBase Wrapper-Architecture. Internal Report 09/99 Univ. of Frankfurt, presented at the “Workshop on Java and Databases” at OOPSLA99, Denver

[Pr00] Priese, Claus P. Architecture of a reusable and extensible DataBase-Wrapper with rule-set based object-relational schemes. in journal “L'objet” Vol.6 issue N40, Hermes Science Publishing Ltd, Oxford, 2000

[Pr02] This journal-issue is republished in 2002 as book by Hermes Penton Science, 2002 with ISBN 1903996155

[Pr03a] And this journal-issue is republished again in 2003 as book from Stylus Publishing

[St99] Michael Stonebraker, Paul Brown: Object-Relational DBMSs. Morgan Kaufmann Publ. Inc., San Francisco, CA, USA, 1999.

[Ta03] “Tamino” is a commercial product from Software AG, Darmstadt Germany. For the actual version and other resources see http://www.tamino.com , 2003.

[XML00] Tim Bray, Jean Paoli, C.M.Sperberg-McQueen: W3C Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml , 2000.

[XSD-P0-01] Schema Part 0: Primer, W3C Recommendation 2 May 2001 Available at http://www.w3.org/TR/xmlschema-0/

[XSD-P1-01] XML Schema Part 1: Structures, W3C Recommendation 2 May 2001 Available at http://www.w3.org/TR/xmlschema-1/

[XSD-P2-01] XML Schema Part 2: Datatypes, W3C Recommendation 2 May 2001 Available at http://www.w3.org/TR/xmlschema-2/