Master Thesis

Automatic Generation of OpenGIS-compliant Relational Database from XML Schema

Aivaras Pigaga
Matr. Nr.: 20726
Information and Media Technologies
Technical University of Hamburg-Harburg

Under the supervision of
Prof. Dr. Joachim W. Schmidt
Prof. Dr.-Ing. Erik Pasche
Rainer Marrone
Andreas von Dömming

Hamburg, March 22, 2004

Acknowledgements

Firstly, I wish to express my deep appreciation and sincere thanks to my wife Eleni, who understood and supported me in all my deeds.

I would like to express my sincere thanks to my advisors Rainer Marrone and especially Andreas von Dömming who always found time to advice and guide me during this work.

Many thanks also to my supervisors Prof. Dr. Joachim W. Schmidt and Prof. Dr.-Ing. Erik Pasche for accepting me to carry out this thesis.

I declare that:

I have carried out this work myself, all literally or content-related quotations from other sources are clearly pointed out, and no other sources or aids other than the ones specified are used.

Hamburg, March 2004

Aivaras Pigaga

Table of Content

1. Introduction 13
  1.1. Motivation 13
  1.2. Objectives 15
  1.3. Structure of the Work 16
2. Concepts and Technologies 17
  2.1. Model-View-Controller (MVC) 17
  2.2. XML Schema 18
  2.3. OpenGIS 22
  2.4. Summary of the Chapter 25
3. Model for Relational Database Schema Definition Generation from XML Schema 27
  3.1. Generic Data Importer 27
  3.2. Problem Statement 30
  3.3. Three-Layer Architecture of the Model and Model-View-Controller (MVC) Design Pattern 31
    3.3.1. Logic Layer 34
      3.3.1.1. Model 34
      3.3.1.2. Container 37
    3.3.2. Data Layer 39
      3.3.2.1. Mapping Specific XML Schema to Model objects 39
      3.3.2.2. Data Type Mapping 44
      3.3.2.3. Data Mappers 45
    3.3.3. Pseudo-Presentation Layer 49
      3.3.3.1. SQL Builders 49
      3.3.3.2. SQL Scripts Generation Process 51
  3.4. Limitations 53
  3.5. Summary of the Chapter 54
4. Implementation of the Model 55
  4.1. Technologies Used for the Implementation 55
    4.1.1. Java Programming Language 55
    4.1.2. JDBC technology 55
    4.1.3. JDOM 55
  4.2. Implementation Specificities 56
  4.3. Example of the Use of the Implemented Model 58
  4.4. Evaluation of the Implementation 61
  4.5. Summary of the Chapter 61

5. Conclusions 63
5.1. Summary 63
5.2. Outlook 64

Table Of Appendices

Appendix A. XML Schemas Related to Model.. 67
Appendix B. Instances of XML Schemas Related to Model 69
Appendix C. Implementation Related XML Documents.. 83
References 89

Table of Figures

Figure 2.1 Model-View-Controller Design Pattern 17
Figure 2.2 The Example *Dean* Schema [GMLv2.1] 20
Figure 2.3 Key Element Definition 22
Figure 2.4 Example of a Non-Spatial Feature *Dean* Definition [GMLv2.1] 24
Figure 2.5 Example of the Geometry Element Declaration [GMLv2.1] 24
Figure 3.1 Dependency Levels of the Three Database Schemas 28
Figure 3.2 Generic Data Importer (Generic Model) 29
Figure 3.3 Generic Data Importer 30
Figure 3.4 Three-Layer Architecture of Relational Database Schema Definition Generation from XML Schema Model 31
Figure 3.5 Relational Database Schema Generation from XML Schema Process Activity Diagram 32
Figure 3.6 Model for Relational Database Schema Generation from XML Schema. Class Diagram 34
Figure 3.7 Model Class Diagram 35
Figure 3.8 Container Class Diagram 38
Figure 3.9 Container Activity Diagram 38
Figure 3.10 Elements and Complex Type Definitions Example 40
Figure 3.11 Inline Elements’ and Simple Type Definitions Example 41
Figure 3.12 The Key Element of the XML Schema Example 42
Figure 3.13 The XML Schema for the Reserved SQL Keywords 43
Figure 3.14 The XML Schema for the Mapping of the Data Types 44
Figure 3.15 Data Mappers Class Diagram 46
Figure 3.16 Data Mappers Activity Diagram (Mapping Process) 48
Figure 3.17 SQL Builders Class Diagram 50
Figure 3.18 *PostgisSQLBuilder* Activity Diagram 52
Figure 3.19 Typical CREATE TABLE and SELECT AddGeometryColumn() SQL statements in Postgis SQL 53
Figure 4.1 The XML Schema for the Configuration File of the *Generator* 56
Figure 4.2 The XML Schema for the Report File of the *Generator* 57
Figure 4.3 Example of a Specific XML Schema Document Generated by GeoServer 60
Figure 4.4 Example of the Relational Database Schema Definition for Postgis RDBMS Generated by the *Generator* (RedGenie) 61
Figure A.1 The XML Schema for Storing Reserved SQL Keywords 67
Figure A.2 The XML Schema for Data Type Mapping Definition 67
Figure B.1 The XML Document Defining the Reserved SQL99 Keywords 73

Figure B.2 The XML Document Defining the Data Type Mapping from XML Schema Data Type to Java Data Types 78
Figure B.3 The XML Document Defining the Data Type Mapping from Java Data Types to Postgis SQL Data Types 81
Figure C.1 The XML Schema for the Configuration File for the *Generator* (RedGenie) 83
Figure C.2 The Configuration File for the *Generator* (RedGenie) 83
Figure C.3 The XML Schema for the Mapping Process Report Generation 84
Figure C.4 The Report File for the Buenzau_nutzung Relational Database Schema Definition Generation 87

1. Introduction

Scientific and business communities are generating considerably more data than ever before. The requirements of distributed information systems like Geographical Information Systems (GIS), Enterprise Resource Planning systems (ERP) or Business-to-Business (B2B) solutions grow more complex.

Geographical Information System (GIS) [GIS04] is a geographical data combined with other geographical information, like maps or photos, about the geographical area. The different geographical information is usually provided in layers and can be used for various reasons, like catastrophe management or environmental damage analysis. Such information usually comes from different sources, is collected using different methods, and is classified using different classification schemes. It is the case very often, that the information from one source can only be combined with the information of another source by putting a lot of manual effort. In other words, there is a lack of schematic and semantic interoperability.

The agencies and organizations, like Cadastre, Town Planning, or Water Management agencies, which are actively using GIS solutions, are in need of better schematic and semantic interoperability on the information management level. Those separate agencies and organizations have very valuable data, but most often they cannot use it effectively in the established models, like Rainfall Runoff Model for Flood Risk Management, for the decision-making, since they lack other data that has been collected and stored by another agency. So it is needed to search for the solutions to ease the cooperation between those agencies and organizations. Within the European Water Framework Directive [WFD03] all agencies related to Water Management must find a solution for schematic and semantic interoperability issue.

One solution that would help in the discussed matter would be to implement the services defined in the OpenGIS [OGC04] specifications. The implemented services could provide the schematic and semantic interoperability between different systems of the different agencies and organizations and enable building of a common web-based layer over the existing systems and databases, which would provide everyone with the information regardless of it source.

1.1. Motivation

Several projects defined within the EU Water Framework Directive are in progress in the Department of River and Coastal Engineering in Technical University Hamburg-

Harburg (TUHH). They are dealing with the issues of connecting the Rainfall Runoff Model and the Flood Forecast with the geographical data available in the following agencies in Hamburg Area:

- Behörde für Umwelt und Gesundheit (BUG) – Department of Environment and Health;
- Behörde für Bau und Verkehr (BBV) – Department of Civil Engineering and Transport;
- Landesbetrieb für Geoinformation und Vermessung (LGV) – State Enterprise for Geo-Information and Survey.

The main issue in these projects is the schematic and semantic interoperability of the geographical information originating from various sources. The department of River and Coastal Engineering in TUHH is determined to implement the OpenGIS specifications, since they provide such interoperability, in Java environment. However the OpenGIS specifications do not address some important issues, which arose during the implementation of the OpenGIS specification in the Department.

To enable a more dynamic exchange, archiving, and usage of the data, one has to provide the lacking functionalities for the importing data to the relational databases without human intervention in creating or altering relational database schemas for it. Such functionality would automate many tasks, which are normally performed by humans in distributed information systems. It would enable more dynamic data managing and archiving processes. In GIS systems, it would enable to create, remove, or alter geographical features dynamically without the human intervention in editing the relational database schema.

There are only two commercial and one open source solutions publicly available that enable the manipulation of the relational database structures dynamically. The following solutions, which enable such manipulation by supporting the generation of the relational database schema definition from XML Schema, are currently available: Oracle has implemented an integrated feature for such process, which is available in Oracle 9i [ORCL04], XMLSpy [XSPY04] tool has such ability also, and open source project xsd2db [XSDB04] implements the discussed generation, which is implemented in .NET environment [NET04].

One can ask the following question. Why is it needed to analyze, design and implement the solution for a problem, for which there are already several available solutions? This question can be answered with the following question and answer. Is there a solution from the above mentioned ones, which is platform- and RDBMS independent? Oracle feature can be used only for the Oracle RDBMS and so it is not RDBMS-independent. XMLSpy feature can be used only in the environment of the XMLSpy tool. So it cannot be use as a library or component, which one could integrate into ones own applications. Both, the Oracle 9i and the XMLSpy features, are not open source, but high-priced commercial products. The xsd2db tool is slightly better. It is free, open source product, which supports only several RDBMS now, but can be easily extended to support any RDBMS. Using its source code, one could reuse most of it for ones own applications. However, it is based on .NET framework, which is platform-dependent and is useless in Java-based systems. Moreover, for the OpenGIS-based projects in the department, there is a need for a component, which would support the generation of new geographical features defined in Open GIS [OGC04] Simple Features Specification for SQL [SFSv1.1]. Using such a component one would be able to create, remove, or alter geographical features dynamically without any human intervention in editing the relational database schemas in the persistence layer of the Geographical Information Systems (GIS), which is not only not available in any implementation of OpenGIS specifications, but also not even defined in any OpenGIS specification.

There is the lack of a platform-independent component for relational database schema definition generation from XML Schema, which would be written in Java environment and would support geographical features as it is defined in Open GIS Simple Features Specification for SQL [SFSv1.1]. It is intend to fill in this gap with this thesis.

1.2. Objectives

This thesis concerns itself with the following objectives:

- give an overview of thesis related concepts and technologies: Model-View-Controller (MVC), XML Schema, OpenGIS Specifications;
- establish a model for generic data importer and relational database schema definition generation from specific XML Schema;
- implement the established model for relational database schema definition generation from specific XML Schema;
- evaluate the established model for relational database schema definition generation from specific XML Schema and its implementation; identify and point out its limitations;
- summarize the results of this thesis and discuss the outlook on further development of the established model for relational database schema definition generation from specific XML Schema and its implementation.

1.3. Structure of the Work

In chapter 2 a short introduction to the concepts and technologies used during this thesis project is provided.

In chapter 3 a detailed description of the design of the relational database schema generation from specific XML Schema model is provided and the limitations of the designed model are identified.

In chapter 4 a description of the implementation of the model, which design is described in chapter 3, and the qualitative evaluation of this implementation are provided.

In chapter 5 a summary of the results of this thesis and an outlook in possible further development of the designed model and of the generic data importer is provided.

2. Concepts and Technologies

In this chapter the basics of the model-view-controller design pattern will be provided. The knowledge relevant to this thesis about XML Schema standard, and OpenGIS specifications will also be provided.

2.1. Model-View-Controller (MVC)

Model-View-Controller (MVC) is a software design pattern, which is rather old. It has been known from the early days of Smalltalk programming language. MVC is a high-level pattern. It concerns itself with the architecture of the application and tries to classify different kind of objects.

Figure 2.1 illustrates MVC design pattern. According to MVC there are three types of objects: model objects (model), view objects (view), and controller objects (controller). The pattern defines the role for each type of these three and software engineers design their object classes to fall in one of these three groups. The groups communicate with each other according to the role each group has in the MVC design pattern.

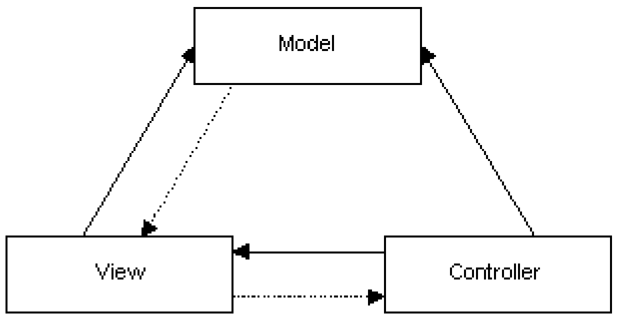


Figure 2.1 Model-View-Controller Design Pattern

Model objects serve as the abstraction of some real world process or system. They encapsulate the information that describes this real world process or system and provide the functionality to operate this information. So the model objects capture not only the state of a process or system, but also how it works. A well-designed application has all its data encapsulated in model objects. Any data that is stored in files or databases should reside in model objects once the data is loaded to the application.

View objects know how to display the given data from the application’s model objects. They should not be responsible for storing the data they are displaying. So the view objects are responsible for the presentation of the data of from the model objects.

Controller objects act as an intermediary between the application’s view and its model objects. The logic that controller objects encapsulate is specific to the application. They are responsible for managing the workflow of the application. For the view objects they ensure the access to the model objects they need to display. The controller objects usually notify the view objects when the model objects change their state.

The three types of objects are separated from each other by abstract boundaries and all the communication between them is conducted across these boundaries (see figure 2.1). The model is the core of the application and it does not know what kind of views observe it. The only weak relation that the model might have is the notification of the views about the change of its state, which could be also done by the controller. By contrast, the view knows exactly what kind of model does it observe. The view has a strong relation to the model and it can use all functionality provided by the model. It might have a weak relation to the controller to submit the requests for information. The controller has strong relation to both view and the model. Since the controller defines the behavior of the application, it needs to be able to use the functionalities provided by both view and model.

Shifting of the application specific code to the controller, the model and view objects become more general and reusable. Controllers are often the least reusable objects in an application. The separation of the objects into three types increases the understandability of the design and the code of the application and ensures easier extensibility and maintenance of the application.

2.2. XML Schema

XML Schema [XSD04] standard provides the XML constructs to write schemas, which define the shared vocabularies, the structure of XML documents which use those vocabularies, and provide links to associate semantics with them. It is an essential part for XML to reach its full potential. [XSINT01]

The purpose of schema is to define and describe a class of XML documents by using the constructs provided by XML Schema standard to constrain and document the meaning, usage, and relationships of the parts of these documents.

The XML Schema specification consists of three parts:

- *XML Schema Part 1: Structures* [XSDp1]. It proposes methods to describe structure and constraint the content of an XML Document. It also defines the rules for schema-validation of documents;
- *XML Schema Part 2: Datatypes* [XSDp2]. It defines a set of simple data types, which can be associates with XML element types and attributes;
- *XML Schema Part 0: Primer* [XSDp0]. It explains what schemas are, how they differ from DTDs, and how one builds a schema.

The introduction on XML Schema based on *XML Schema: Primer* [XSDp0] specification part will be provided further in this section. The introduction will be provided only on the concepts that are relevant to this thesis: element declaration, complex and simple type definitions, *minOccurs* and *nillable* attributes, substitution groups, extensions, and key definitions. The term *schema* will be used to refer to specific XML Schema further in this section. The schema defines the class of documents and so the term *instance document* is used to describe an XML document that conforms to a particular schema.

Complex Type Definitions, Element Declarations

In XML Schema, there is a difference between complex types, which can contain elements and attributes, and simple types, that do not allow elements or attributes in their content. In this section the definition of complex types and the declaration of elements that appear with them will be discussed.

The example of the Dean schema will be used in this section (see figure 2.2).

New complex types are defined by using a *complexType* element and they normally contain a set of element declarations. The declaration is not a type itself, but rather an association of the name of the element and the constraint defined by its type. Elements are declared using *element* element. For example, *DeanType* is defined as a complex type, and within the definition of *DeanType* there are four element declarations. So every element appearing in the instance of the Dean schema whose type is declared to be *DeanType* must consist of four elements as it is defined in the Dean schema. These elements must be called *id*, *familyName*, *age*, and *nickName* and must appear in the same sequence in which they are declared in the schema.

The *Dean* element declaration is associated with the complex type *DeanType*. The result is, that every appearance of the element *Dean* in the instance document of the Dean schema will contain four elements as it is defined by *DeanType* complex type.

So far the example of the element declaration was discussed which is associated with an existing type definition. Sometimes is preferable to use an existing element rather than declare a new one, like it is with one element declaration in *DeanType* complex type definition, which references the element declaration *age* using the attribute *ref*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema
  targetNamespace="http://www.deanurl.com/dean"
  xmlns:buenz="http://www.deanurl.com/dean"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified" version="1.0">
 <xs:import namespace="http://www.opengis.net/gml" schemaLocation=
  "http://schemas.opengis.net/gml/2.1.2/feature.xsd"/>
 <element name="Dean" type="ex:DeanType" substitutionGroup="gml:_Feature"/>
 <element name="age" type="integer"/>
 <complexType name="DeanType">
  <complexContent>
   <extension base="gml:AbstractFeatureType">
    <sequence>
     <element name="id" type="integer"/>
     <element name="familyName">
      <simpleType>
       <restriction base="string">
        <maxLength value="20">
       </restriction>
      </simpleType>
     </element>
     <element ref="age" nillable="true"/>
     <element name="nickName" type="string" minOccurs="0"/>
    </sequence>
   </extension>
  </complexContent>
 </complexType>
</schema>
```

Figure 2.2 The Example *Dean* Schema [GMLv2.1]

The value of the attribute *ref* must always reference a global element (the element declared under the element *schema*). The result of such element declaration using the attribute *ref* is, that the element *age* may appear in the element, which was associated with the type *DeanType* in the instance of the Dean schema.

Substitution Groups

The XML Schema provides the mechanism for elements to substitute the other elements. More specifically, elements can be assigned to a special group of elements that are substitutable for a particular named element, which is called the head element. In figure 2.2, an element called *Dean* is declared and assigned it to a substitution group whose head element is *_Feature*. The *Dean* element can be used any place where *_Feature* element can be used. The element is assigned to a substitution group by setting a *substitutionGroup* attribute of an element to the name of the head element.

minOccurs and nillable

The element is required to appear in the instance of the schema if the attribute *minOccurs* is set to *1* or more. The *nickName* element is optional within *DeanType*, because the value of the *minOccurs* attribute in its declaration is set to *0* (see figure 2.2). The default value of *minOccurs* attribute is *1*.

In some cases it is preferable to have the element appearing in the instance of the schema, but set to the *null* value. Such cases can be represented using XML Schema's *nil* mechanism, which enables an element to appear with or without a not-nil value. To declare an element as being able to appear in an instance of the schema carrying the *nil* value, one has to set its attribute *nillable* in the element declaration to *true*. For example, the *age* element is nillable within the *DeanType*, because its *nillable* attribute is set to *true*.

Simple Types

Some simple types, such as *string* or *decimal*, are build in to XML Schema. You can see all built-in simple types listed in created data type mapping files in appendix B. There are also other simple types that are derived from the built-in simple types. For example, the element *familyName* in the complex type *DeanType* is declared in the association with the simple type, which is derived from a built-in simple type *string* using a *restriction* method (see figure 2.2). So new simple types are defined by deriving them from existing simple types (built-in or derived). In particular, one can derive a new simple type by restricting the existing simple type. For example, narrowing its legal range of values or limiting its length. The *simpleType* element is used to define the new simple type. The *restriction* element is used to indicate the *base* simple type and to constrain the range of values or its length using different *facets*, depending on the simple type: *maxLength*, *minInclusive*, *maxInclusive*, *minExclusive*, *maxExclusive*, *pattern*, *enumeration*, etc. For example, the element *familyName* declaration in the complex type *DeanType* is associated with the simple type, which is the *restriction* of the *integer* built-in simple type. It is restricted using the *maxLength* attribute, which is set to *20*. The result of this restriction is, that the element *familyName* in the instance of the schema can take strings not longer than *20* characters as values.

Extensions

One can derive the complex types by extension of the base type, which actually means that the complex type definition inherits the content definition from the base type and adds its own content definition to it. For example, the *DeanType* complex type in figure 2.2 is an extension of *AbstractFeatureType* complex type.

Key Definitions

Using the *key* element one can constrain an element to be unique and not nillable. The name that is associated with the key makes the key able to be referenced from elsewhere. Keys are defined using the *key* element as it is illustrated in figure 2.3.

```
<key name="idKey">
 <selector xpath="Dean"/>
 <field xpath="id"/>
</key>
```

Figure 2.3 Key Element Definition

The *key* element, which name is *idKey* declares the element *id* from the element *Dean* as unique and not nillable. One can reference the *key* element name *idKey* from anywhere within the schema definition where it is defined.

2.3. OpenGIS

Modern information systems are distributed, interoperable, integrated, and web-based. The classical GIS solutions do not conform to any of the mentioned advantages of the modern information systems. OpenGIS initiative is working on moving the GIS to the modern information systems level. The vision of the OpenGIS initiative is the following: a world in which everyone benefits from geographic information and services made available across any network, application, or platform.

The OpenGIS initiative has released a number of specifications. Only the fragments of the Open GIS Geography Markup Language (GML) Implementation Specification

Version 2.1.2 [GMLv2.1] and the Open GIS Simple Features Specification for SQL Revision 1.1 [SFSv1.1] relevant to this thesis will be introduced.

The GML specification provides the necessary schemas to define the geographical features to use them in OpenGIS-compliant systems. There are two major schemas: geometry schema and feature schema. The geometry schema defines geometries, like *Point*, *LineString*, etc. The feature schema defines the structures to declare features and defines complex feature types, which contain the regular XML Schema elements and GML geometries defined in the geometry schema. The Open GIS Simple Features Specification for SQL defines the standard SQL schema that supports the storage, retrieval, query, and update of simple features. First the geometry list from the geometry schema will be provided. Then the feature schema will be introduced. Finally the list of SQL geometry types will be provided.

Geometries

GML provides geometry elements corresponding to the following geometry classes: *Point*, *LineString*, *LinearRing*, *Polygon*, *MultiPoint*, *MultiLineString*, *MultiPolygon*, and *MultiGeometry*.

Feature Schema

The feature schema models the geometric properties as association classes that link features with the geometries. Concrete geometric property types such as *PointProperty* constrain the geometry to a particular type, such as *Point*. There are six geometric properties defined in the feature schema: *PointProperty*, *LineStringProperty*, *PolygonProperty*, *MultiPointProperty*, *MultiLineStringProperty*, *MultiPolygonProperty*. All mentioned geometric properties are the restrictions from the type *GeometryProperty*.

Defining Features without Geometries

It is not a high possibility that many features will be defined without geometry properties using GML. However, to understand better the definition of a feature, a simple example of a non-spatial feature definition is provided in figure 2.5. The *Dean* element's declaration is associated with a complex type *DeanType* and is declared as a substitution to the *_Feature* element from the elements declared by GML, which makes the *Dean* element a feature type element. The *DeanType* complex type definition is an extension of a type *AbstractFeatureType* and it inherits all capabilities from the base class, like the feature's capability to be identified using *fid* attribute or to use a predefined *description* property to describe the feature. Every feature type definition must extend the *AbstractFeatureType* and every feature element declaration must substitute *_Feature* element from GML.

```
<element name="Dean" type="ex:DeanType" substitutionGroup="gml:_Feature" />
<complexType name="DeanType">
 <complexContent>
  <extension base="gml:AbstractFeatureType">
   <sequence>
    <element name="familyName" type="string"/>
    <element name="age" type="integer"/>
    <element name="nickName" type="string" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

Figure 2.4 Example of a Non-Spatial Feature *Dean* Definition [GMLv2.1]

Defining Features with Geometries

As it is mentioned above, the feature schema provides the pre-defined set of geometry properties, by which one can relate geometries of a particular type to features. For example, in figure 2.5 the example is provided where the *DeanType* feature definition has a point property type element declaration called *deanLocation*. The *type* attribute of the *deanLocation* element is set to the *PointPropertyType*.

```
<element name="deanLocation" type="gml:PointPropertyType"/>
```

Figure 2.5 Example of the Geometry Element Declaration [GMLv2.1]

Every geometry element declaration must be of a type of one of the pre-defined geometry properties.

SQL Geometry Types

The geometry types supported in SQL, as it is defined in Open GIS Simple Features Specification for SQL Revision 1.1 [SFSv1.1], are the following: GEOMETRY, POINT, CURVE, LINESTRING, SURFACE, POLYGON, COLLECTION, MULTIPOINT, MULTICURVE, MULTILINESTRING, MULTISURFACE, and MULTIPOLYGON. It might differ from one SQL implementation to another. For example, in Postgis implementation, only the following SQL geometry types are supported: POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, and additionally GEOMETRYCOLLECTION, which corresponds to the COLLECTION SQL geometry type provided by the specification.

2.4. Summary of the Chapter

In this chapter the introduction to the Model-View-Controller design pattern has been provided. The Model-View-Controller consists of three groups of objects: model, view, and controller. The introduction to the parts of XML Schema standard relevant to this thesis has also been provided, namely element declarations, simple and complex type definitions, element's attributes *nillable* and *minOccurs*, substitution groups, extensions, and key definitions. Finally the introduction to the thesis relevant parts of the following two OpenGIS specifications, namely Open GIS Geography Markup Language (GML) Implementation Specification Version 2.1.2 [GMLv2.1] and Open GIS Simple Features Specification for SQL Revision 1.1 [SFSv1.1] has been provided.

3. Model for Relational Database Schema Definition Generation from XML Schema

First the introduction to the model of a generic data importer with relational database schema definition generator will be given. Then the architecture of the relational database schema definition generation from XML Schema model will be defined. The overall process within this model will be described. Then the detailed design of the defined architecture and the detailed processes within the architecture will be defined. Finally, the limitations of the defined architecture of the model will be identified.

3.1. Generic Data Importer

The following definitions will be used in this section:

- *generic data importer* is the system responsible for a generation of relational database schemas and an import of data to those generated schemas without human intervention in creating or altering those generated schemas before the actual import of data;
- *platform–independent database schema definition* is a database schema definition independent from any database system;
- *RDBMS-dependent relational database schema definition* is a relational database schema definition for a specific relational database management system (RDBMS), e.g. SQL Data Definition Language (DDL) script [SQL01];
- *relational database schema* is a relational database table with their relations and constraints stored in the specific RDBMS on a specific platform.

For the convenience, the figure 3.1 is provided illustrating the abstraction levels of the last three definitions provided above. The least dependent schema is the platform-independent database schema definition, since it does not depend on neither platform nor database system. The RDBMS-dependent relational database schema definition depends on the specific RDBMS, but does not depend on the platform. The most dependent is the instance of the RDBMS-dependent relational database schema definition. The relational database schema depends on the specific platform and on the specific RDBMS.

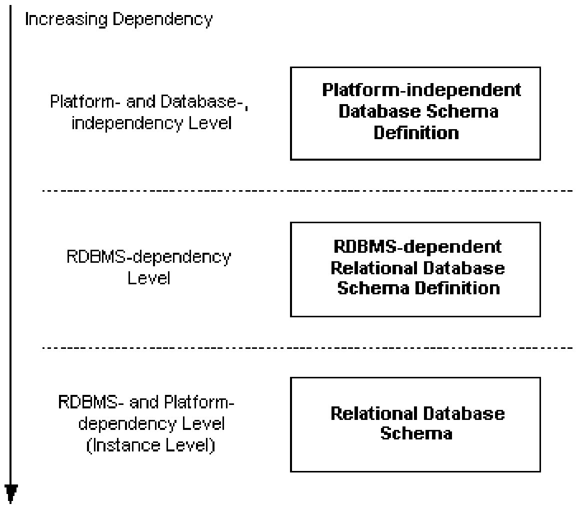

Increasing Dependency
Platform- and Database-, independency Level
Platform-independent Database Schema Definition
RDBMS-dependency Level
RDBMS-dependent Relational Database Schema Definition
RDBMS- and Platform-dependency Level (Instance Level)
Relational Database Schema

Figure 3.1 Dependency Levels of the Three Database Schemas

Generic data importer should be consisting of the following components (see figure 3.2):

- *relational database schema definition generator module* responsible for generating a RDBMS-dependent relational database schema definition based on given platform-independent database schema definition;
- *relational database schema creator module* responsible for creating a relational database schema from the RDBMS-dependent relational database schema definition;
- *data import module* responsible for importing data to the relational database schema created by the relational database schema creator module;
- *relational database schema update module* responsible for altering the relational database schema according to the differences between the platform-independent database schema definition and updated platform-independent database schema definition.

The process of data import should be executed in the following manner (see figure 3.2). The relational database schema definition generator module takes the platform-independent schema definition and generates RDBMS-dependent relational database schema definition from it. The result is then passed to the relational database schema creation module, which creates relational database schema in the given RDBMS. After the relational database schema has been created, the data import module can import data to the created relational database schema. The data import module validates data against platform-independent database schema definition before the actual data import occurs. Eventually the platform-independent database schema definition might change. The data won't validate against the current platform-independent database schema definition any longer. In this case relational database schema update module must alter the relational database schema, based on the differences between the current and updated platform-independent database schema definitions. After the relational database schema has been altered, the data import module imports the data to the altered relational database schema.

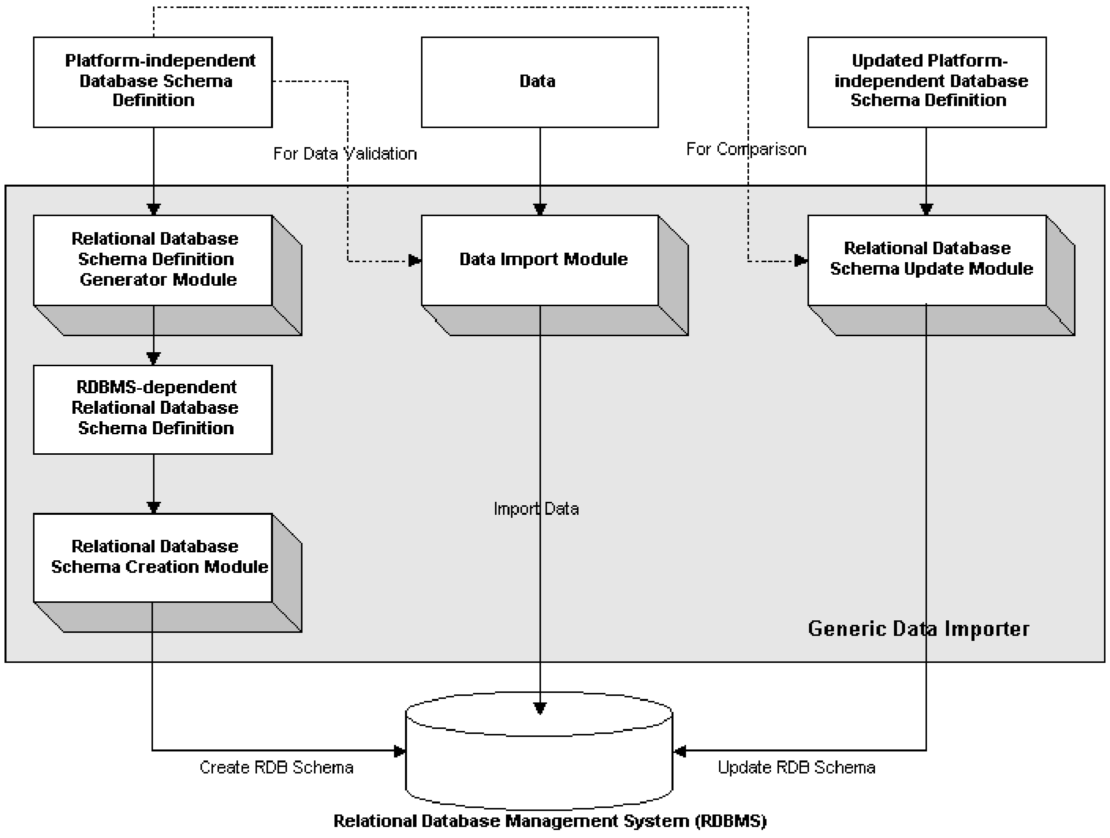

Platform-independent Database Schema Definition
Data
Updated Platform-independent Database Schema Definition
For Data Validation
For Comparison
Relational Database Schema Definition Generator Module
Data Import Module
Relational Database Schema Update Module
RDBMS-dependent Relational Database Schema Definition
Import Data
Relational Database Schema Creation Module
Generic Data Importer
Create RDB Schema
Update RDB Schema
Relational Database Management System (RDBMS)

Figure 3.2 Generic Data Importer (Generic Model)

The generic data importer must be platform-, data domain-, and RDBMS-independent (see figure 3.3). XML [XML04] can be used as a data format. It provides data domain independence. The system can be then applicable in any domain possible, e.g. it could archive a procurement process data in a B2B solution or store geographical objects from GIS system. XML is also platform-independent. The specific XML Schema [XSD04] can be used as a platform-independent database schema definition. Moreover, XML technology provides XML data validation against XML Schemas.

For the implementation of components of generic data importer the following technologies can be used: Java programming language [JAVA04] assuring platform-independency, JDBC [JDBC04] assuring independency of RDBMS, and the DOM [DOM04] assuring independency of XML Parser.

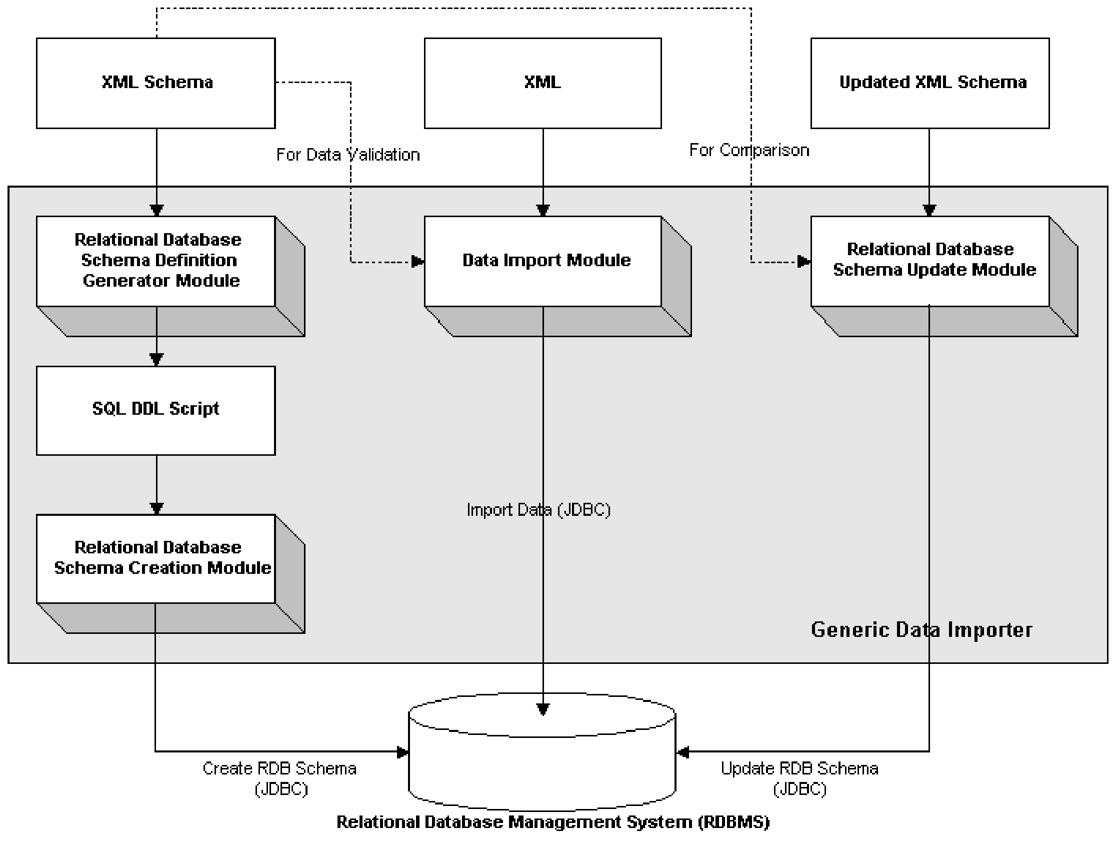


Figure 3.3 Generic Data Importer

The generic data importer is a collection of complex components. The detailed analysis of design and implementation of every component of generic data importer can be separated into several works. In this thesis the design and implementation of relational database schema definition generator module will be discussed and analyzed, since it is the least touched topic in scientific and engineering publications.

3.2. Problem Statement

The model for relational database schema definition generation from specific XML Schema must be established. This model must be extensible, platform- and RDBMS-independent, and must support OpenGIS-compliant features as it is defined in Open GIS Simple Features Specification for SQL [SFSv1.1].

The model for relational database schema definition generation from specific XML Schema must generate SQL DDL scripts, which would define relational database schema, from a given XML Schema as it is illustrated in figure 3.3.

3.3. Three-Layer Architecture of the Model and Model-View-Controller (MVC) Design Pattern

The architecture of the model for relational database schema definition generation from specific XML Schema is defined as it is depicted in figure 3.4. To define the architecture of this model the Three-Layer Architecture [MaHu02] has been used.

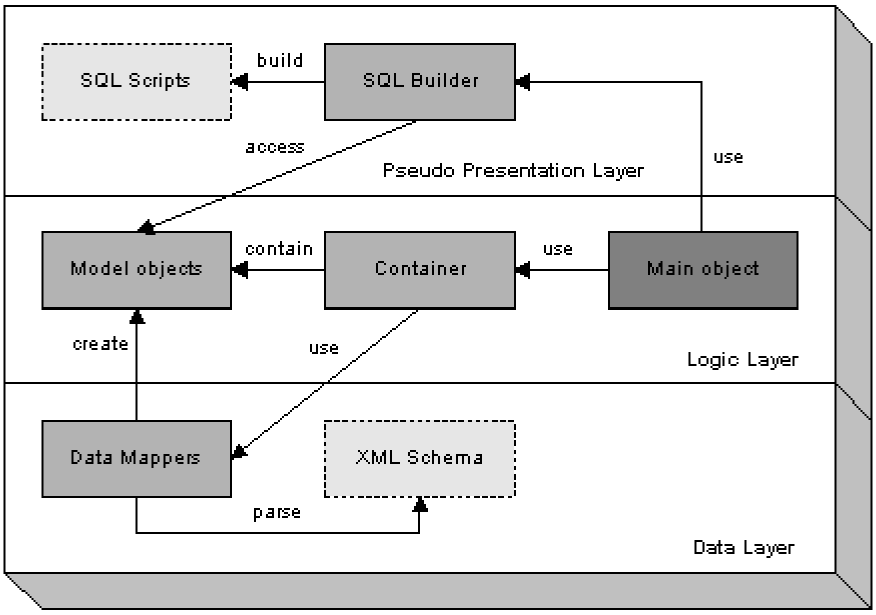


Figure 3.4 Three-Layer Architecture of Relational Database Schema Definition Generation from XML Schema Model

The model depicted in figure 3.4, consist of the following layers: data layer, logic layer, and pseudo-presentation layer. Each layer addresses different responsibilities:

- *data layer* is responsible for the functionality of platform-independent database schema definition by using XML Schema standard [XSD04]. It also encapsulates the Data Mappers, which are responsible for the functionality for parsing XML Schema and creating Model objects, which are defined in logic layer. In other words, it is responsible for mapping the structure definition from XML Schema to the Model objects;
- *logic layer* encapsulates the Model objects, which capture the relational database structure and are the core of the architecture. It also encapsulates the Container, which contains the Model objects and controls their creation and access to them. Finally, it encapsulates the Main object, which controls the process flow and the choice of the SQL Builder, which is defined in pseudo-presentation layer;

- *pseudo-presentation layer* encapsulates the SQL Builder, which is responsible for the generation of the RDBMS-specific SQL DDL scripts from the Model objects.

In the classical three-layer architecture model one finds a presentation layer instead of the pseudo-presentation layer. However, since the presentation layer does not provide any presentation layer functionalities in the sense of the classical understanding and purpose of this layer, but it is still somehow presenting the data from Model objects as SQL DDL scripts, it is call pseudo-presentation layer in this particular case.

The relational database schema generation from XML Schema process is illustrated in figure 3.5. The term *user* will be used for the application, which initiates the process by passing the XML Schema to the Main object and at the end receives the SQL DDL script. The process starts with the passing of the XML Schema to the Main object, which passes it to the Container, which, on his turn, passes it to the Data Mappers. The Data Mappers parse the XML Schema and create Model objects based on it. The Container returns the created Model objects to Main object. The Main object chooses the right SQL Builder based on the initial input from the user. The chosen SQL Builder builds the RDBMS-specific SQL DDL script based on the created Model objects. The Main object returns then the created SQL DDL script to the user.

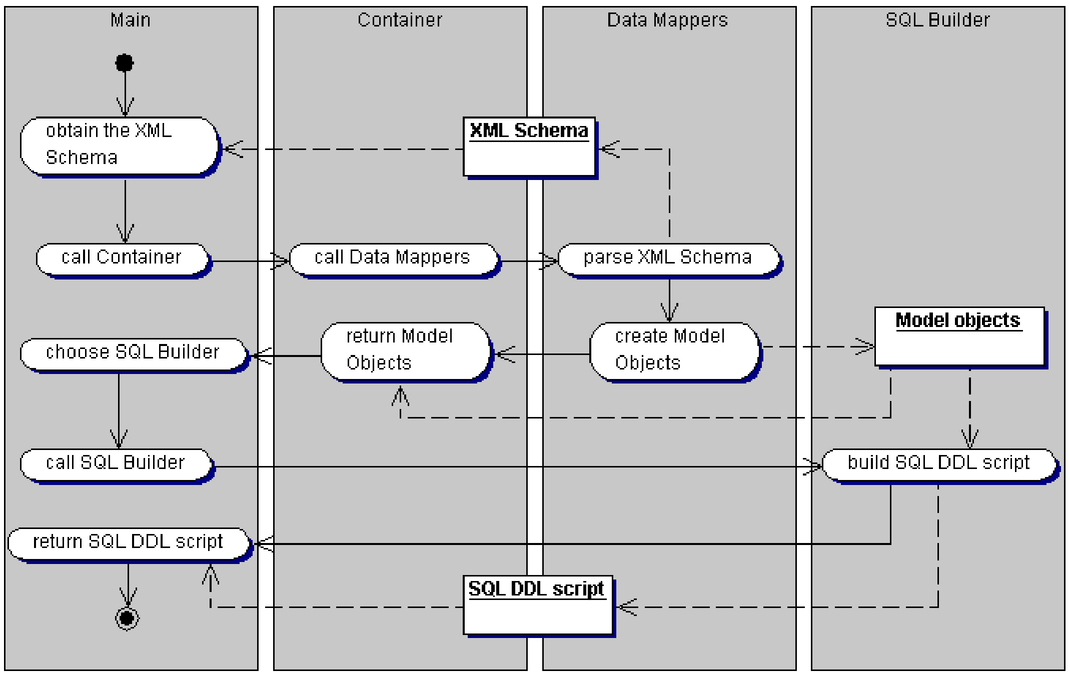


Figure 3.5 Relational Database Schema Generation from XML Schema Process Activity Diagram

The three-layer model was developed also according to a variation of the classical Model-View-Controller (MVC) pattern [SiSt02] as it is shortly introduced in chapter 2. The following describes the model from the point of view of MVC design pattern:

- *controller component.* The Main object is getting the input and according to it chooses the right SQL Builder. It also passes the XML Schema to the Container, which passes it to the Data Mappers and initiates the creation of the Model objects. The Container then returns the Model objects to the Main object. Therefore the Main object with the Container are parts of the controller component;
- *view component.* SQL Builder is some kind of view component. SQL Builder builds the SQL DDL script according to the Model objects. In other words, it presents the data from Model objects. For the SQL Builder the term *pseudo-view* is used, since the SQL Builder does not conform with the classical understanding of a view component;
- *model component.* The Model objects capture the structure of the relational database schema and the Data Mappers define the mapping from XML Schema structures to the Model objects. Therefore the Model objects with Data Mappers are parts of model component.

Using three-layer architecture of the model provides the following [MaHu02]:

- easy development and testing;
- scalability;
- easy maintenance;
- better performance, e.g. network load.

Using Model-View-Controller (MVC) pattern provides the following [SiSt02]:

- separation of user input, logic, and presentation logic;
- clean, easy understandable design;
- minimal coupling between components;
- extensibility;
- easy maintenance.

The relational database schema definition generator is designed as it is depicted in the class diagram in figure 3.6 and described above in this section using figure 3.4 and figure 3.5. The package *model* corresponds to the Model definition. The package *xml* corresponds to the Data Mappers definition. The package *logic* corresponds to the Container definition. The package *main* corresponds to the Main object definition. The package *sql* aggregates all SQL Builder definitions.

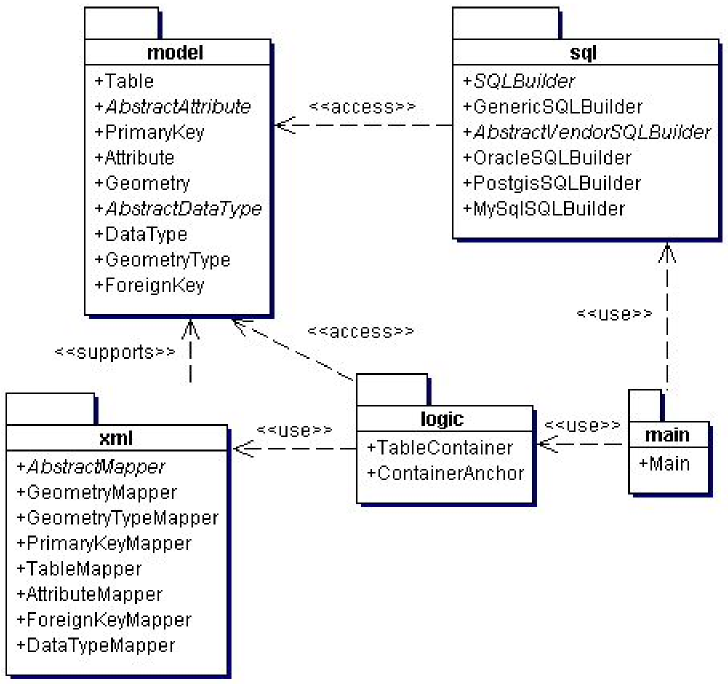


Figure 3.6 Model for Relational Database Schema Generation from XML Schema. Class Diagram

3.3.1. Logic Layer

First the Model will be discussed, since it is the core of the relational database schema definition generator model.

3.3.1.1. Model

The Model is designed as it is depicted in figure 3.7. The Model corresponds to the generic relational database schema with the support of geometries as it is defined in Open GIS Simple Features Specification for SQL [SFSv1.1].

In the Model only simple data types are supported. The nested complex data types in XML Schema corresponds to the user-defined data types in relational database schema. Relational database schema's user-defined data types are not supported in the Model, since, during the time the design of the model was in progress, no implementation of OpenGIS specifications did support the XML Schema's nested complex data types yet. By the end of this thesis period, it was brought to the attention, that one of the implementations of OpenGIS specifications: namely degree [DGREE04], has implemented the support of the nested complex types for several RDBMSs: Postgis, PointDB, MySQL, GMLDB. However, these implementations are not yet stable. So in the Model there is one data type type, namely simple data type, defined in generic relational database schema and one data type type, namely geometry data type, defined in OpenGIS Specifications. Therefore two classes are designed, namely *DataType* and *GeometryType*, which extend the abstract class *AbstractDataType* with the only property *name*, which is common for both extensions. The described hierarchy of three classes captures required data types structure.

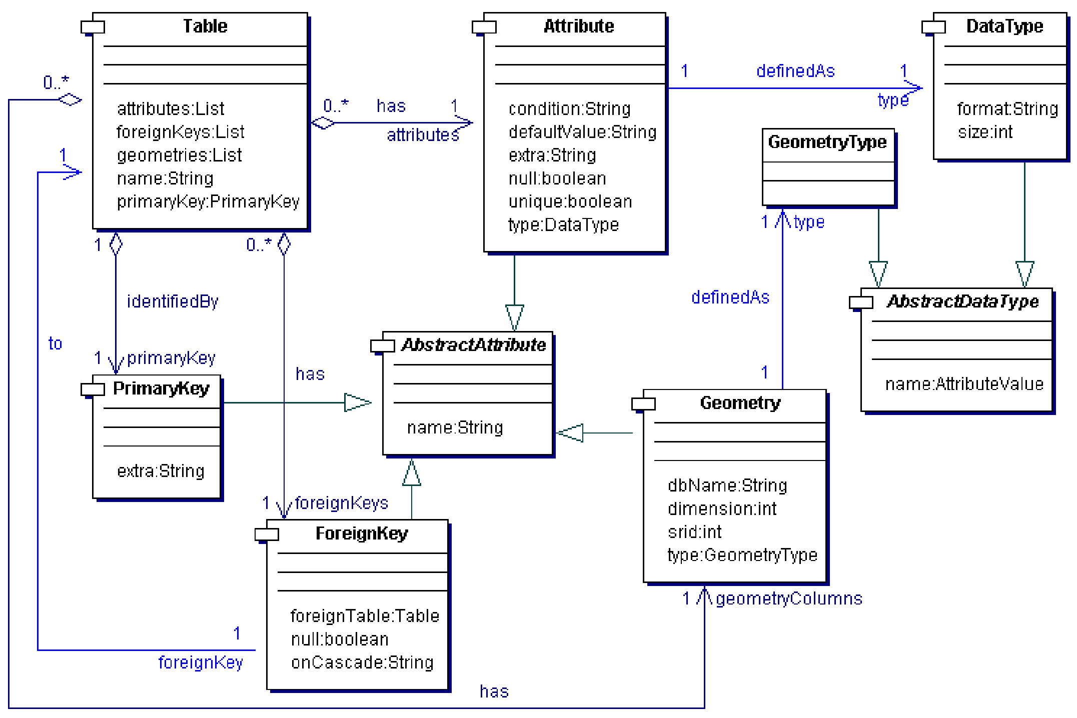


Figure 3.7 Model Class Diagram

The class *DataType* has the following additional properties: *format* and *size*. The property *format* is required for the data types, which are specified using patterns or formats, e.g. DATE can be specified using a pattern "yyyy.mm.dd". The property *size* is required for the data types, which are specified by length in number of characters or by size in bytes, e.g. the size of VARCHAR can be specified using number of characters.

The class *GeometryType* has no additional properties. The class *GeometryType* has only the property *name*, which it inherits from the abstract class *AbstractDataType*.

In the generic relational database schema there are the following attribute types: attribute, primary key, and foreign key. The OpenGIS specifications define one more attribute type, which is geometry. All those four attribute types are defined as classes in the Model, namely *Attribute*, *PrimaryKey*, *ForeignKey*, and *Geometry*, which extend the abstract class *AbstractAttribute* with the only property *name*, which is common for all four extensions. The described hierarchy of five classes captures required attributes structure.

The class *Attribute* has the following additional properties: *condition*, *defaultValue*, *extra*, *null*, *unique*, *type*. The property *condition* captures the generic relational database schema attribute's unidirectional constraint, which can be most of the cases defined using a SQL CHECK clause. The property *defaultValue* captures the default value of the generic relational database schema's attribute. The property *extra* can be used for any extra information about the generic relational database schema's attribute, e.g. automatic increment. The property *null* is used when defining if the generic relational database schema's attribute can be assigned a null value. The property *unique* is used when defining if the generic relational database schema's attribute is unique. The property *type* is defining the data type of the generic relational database schema's attribute and it is of a type of the class *DataType* from the Model.

The class *Geometry* has the following additional properties: *dbName*, *dimension*, *srid*, *type*. All these four properties are required for geometry definition as it is defined in OpenGIS specifications. The property *dbName* provides the information about the specific relational database name for which the geometry is defined. The property *dimension* defines if 2- or 3- dimensional geometries will be created and stored in the relational database schema. The property *srid* provides the id of the spatial referential system as it was described in chapter 2, which must be used as a base for the relative coordinates of the each geometry of this class. The property *type* is defining the geometry type of the geometry definition and it is of a type of the class *GeometryType* from the Model.

The class *PrimaryKey* has only one additional property *extra*, which is used for the same purpose as the property *extra* of the class *Attribute*. It is used for any extra information about the generic relational database schema's attribute, e.g. automatic increment. This particular property is very useful in the case when the generic relational database schema's attribute is a primary key.

The class *ForeignKey* has the following additional properties: *foreignTable*, *null*, *onCascade*. The property *foreignTable* is of a type of the class *Table* and captures the relation between tables defined in the generic relational database schema. It provides the name of the table, to which the table, which aggregates this foreign key, has a relation. The property *null* is used for the same purpose as the property *null* of the class *Attribute*. The property *null* is used when defining if the generic relational database schema's attribute can be assigned a null value. The property *onCascade* captures the SQL FOREIGN KEY clause's part, where one can define the behavior of the child table's records, in the case of deletion of the parent table's record. For example, one can define the following behavior: if the parent record is deleted all the child records must be deleted too.

The class *Table* captures the structure of the table as it is defined in the generic relational database schema and it has the following properties: *nam*e, *attributes*, *geometries*, *foreignKeys*, and *primaryKey*. The property *attributes* is a list of attributes of a type of the class *Attribute*. The property *geometries* is a list of geometries of a type of the class *Geometry*. The property *foreignKeys* is a list of foreign keys of a type of the class *ForeignKey*. The property *primaryKe*y is of a type of the class *PrimaryKey*. The multiple primary keys are not supported for the sake of simplicity.

The Model is easily extensible. One can add the support for the user-defined data types and multiple primary keys any time it is necessary by adding the necessary classes to the Model.

3.3.1.2. Container

The Container aggregates the Model objects and is responsible for controlling the mapping of the data structures from XML Schema to the Model objects and for providing the functionality to access the necessary Model objects. The Container is designed as follows (see figure 3.8):

- *ContainerAnchor* class defines the functionality for returning the specific container upon the request of the *user component* (the component that is using the object of a type of the class *ContainerAnchor*). The class *ContainerAnchor* creates all containers and can return any container upon the request;
- *TableContainer* class is a typical container class, which when instantiated contains the specific Data Mapper, in this particular case the object of the type of the class *TableMapper*, and the list of specific Model objects, in this particular case the list of objects of the type of the class *Table*, for which the contained *TableMapper* object defines the mapping from XML Schema. It also provides the functionality for returning necessary Model objects.

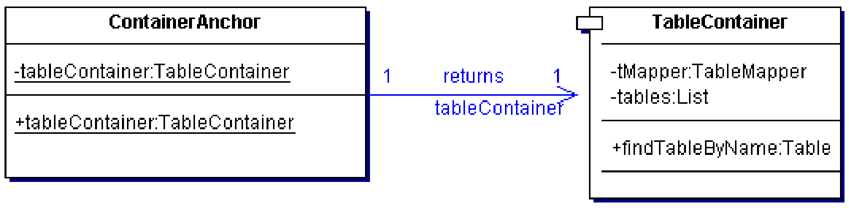


Figure 3.8 Container Class Diagram

Normally, for each Model class a container class is defined. However, in this particular case, the *TableMapper* instance calls other Data Mappers' instances, which, in their turn, call other Data Mappers' instances and so on. So the whole mapping process runs through after calling only *TableMapper* instance.

Since the access to the Model objects and Data Mappers' functionalities are controlled through the Container, the Model is strongly decoupled from the rest of the components. Therefore, when extending the Model, the Data Mappers, or the Container the other components are not influenced.

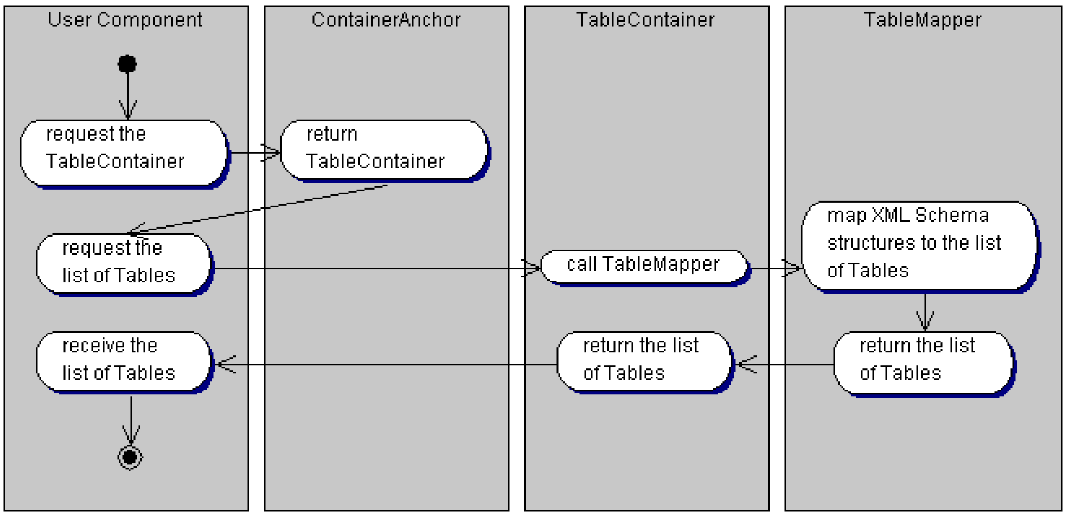

User Component
ContainerAnchor
TableContainer
TableMapper
request the TableContainer
return TableContainer
request the list of Tables
call TableMapper
map XML Schema structures to the list of Tables
receive the list of Tables
return the list of Tables
return the list of Tables

Figure 3.9 Container Activity Diagram

The process of creating the Model objects is executed as it is illustrated in figure 3.9. The *user component* requests the *TableContainer* object from *ContainerAnchor*. The *ContainerAnchor*, in his turn, returns the requested *TableContainer* object. Then the *user component* requests the lists of *Table* objects from the received *TableContainer*, which, in his turn, calls the *TableMapper* object to start the mapping process. The *TableMapper* object maps the XML Schema structures to the list of *Table* objects and returns this list to *TableContainer* object. The *TableContainer* object returns the received list of *Table* objects to the *user component* and the process of creating the Model objects is finished. Each Table object in the received list of Table objects holds the full information on specific relational database table definition.

3.3.2. Data Layer

As it was mentioned above, the data layer is responsible for the platform-independent database schema definition. The XML Schema standard is used to define the platform-independent database schema. The data layer is also responsible for the functionality for mapping XML Schema structures to Model objects. The Data Mappers are responsible for it.

3.3.2.1. Mapping Specific XML Schema to Model objects

The mapping of the structures from the specific XML Schema to the Model objects method has been developed in this thesis. The developed method will be described. In this method the following XML Schema structures' definitions [XSDp1] will be used: *element*, *complex type*, and *simple type*. The method provides the way to map the nullability. It also provides the way to map between any data types. It is used, in this particular case, for mapping XML Schema data types to Java data types.

Element, Complex Type, and Simple Type

The definitions of *elements* in XML Schema correspond to either a relational database table definition or a relational database table's column definition. The *element*, which type is a *complex type*, maps always to a relational database table definition. If its substitution group is _Feature, as it is defined in OpenGIS Geography Markup Language (GML) Implementation Specification [GMLv2.1], then the relational database table definition might include a column definition of the geometry type. If it has no substitution group, then the relational database table definition will not include a column definition of the geometry type. Please refer to the example in figure 3.10. The *element* with the name Pallas is of the PallasType type (the prefix msc6 only denotes the target namespace of this specific schema). PallasType is a *complex type*. Therefore the *element* Pallas maps the relational database table with the name Pallas.

The definition of the *complex type* is illustrated in figure 3.10. *Complex type* can define a complex content. The defined complex content is an extension of the base of AbstractFeatureType [GMLv2.1], if the *complex type* might contain the *element* definition of a geometry type. If the *complex type* does not define the complex content or is not an extension of the base of the AbstractFeatureType, the relational database table definition will not contain the definition of the column of the geometry type. The *element* definitions, found in the model group *sequence* or the model group *all*, define the columns of the relational database table. The *element* definition, which maps to the relational database table's column definition will be describe further in this section.

```
<element name="ID" type="integer"/>
<element name="Datum" type="integer"/>
<element name="Uhrzeit" type="integer"/>
<element name="Position" type="gml:PointPropertyType"
        substitutionGroup="gml:pointProperty"/>
<element name="Ereignis" type="string"/>
<element name="Pallas" type="msc6:PallasType"
        substitutionGroup="gml:_Feature"/>
<complexType name="PallasType">
        <complexContent>
                <extension base="gml:AbstractFeatureType">
                        <sequence>
                                <element ref="msc6:ID"/>
                                <element ref="msc6:Datum"/>
                                <element ref="msc6:Uhrzeit"/>
                                <element ref="msc6:Position"/>
                                <element ref="msc6:Ereignis"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>
```

Figure 3.10 Elements and Complex Type Definitions Example

The column of the relational database table is defined as an *element* of the *simple type* and can be defined directly in the definition of the *complex type* (see figure 3.11) or independently (see figure 3.10). For example in figure 3.10 one can see the definitions of the *elements* with the following names: ID, Datum, Uhrzeit, Position, and Ereignis. They are defined independently of the *complex type* definition and are referenced from the *complex type* definition with the name PallasType using the *element*'s attribute *ref*. Each of these *elements* defines the column of the relational database table. Each definition of the *element* is of the certain data type, which is denoted by the attribute *type*. The *element* with the name Position defines the column of the geometry type. The definition of the column of the geometry type will be presented further in this section. In the figure 3.11 the example of the following inline column definitions of the relational database table geom_test is presented: gid, geom, and name. The definitions of nullability and data types will be described further in this section.

The *element* definition, which defines the column of the relational database table, can be also of a *simple type*. The *simple type* can be defined independently of the *element*'s definition and then referenced using the *element*'s attribute *type*. The other way is to define the *simple type* in the body of the *element*'s definition. Such a definition of the *simple type* for the *element* definition *name* is depicted in figure 3.11. In the provided example the *simple type* is a restriction of the XML Schema data type string.

```
<xs:complexType name="geom_test_Type">
        <xs:complexContent>
                <xs:extension base="gml:AbstractFeatureType">
                        <xs:sequence>
                                <xs:element name="gid" type="xs:int"
                                        minOccurs="0" maxOccurs="1"/>
                                <xs:element name="geom" type="gml:PolygonPropertyType"
                                        minOccurs="1" maxOccurs="1"/>
                                <xs:element name="name" nillable="true">
                                        <xs:simpleType>
                                                <xs:restriction base="xs:string">
                                                        <xs:maxLength value="15"/>
                                                </xs:restriction>
                                        </xs:simpleType>
                                </xs:element>
                        </xs:sequence>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>
<xs:element name='geom_test' type='topp:geom_test_Type'
        substitutionGroup='gml:_Feature'/>
```

Figure 3.11 Inline Elements' and Simple Type Definitions Example

The *element* definition of the geometry type is recognized by its *type* attribute. If the value of the *type* attribute of the element is from the namespace, under which the definitions of the geometry types are stored as it is defined in OpenGIS Geography Markup Language (GML) Implementation Specification [GMLv2.1], this *element* is mapped to the column of the geometry type of the specific relational database table. For instance, the *element* definition *geom* in figure 3.11 is mapped to the column of the *Polygon* type of the relational database table *geom_test*. Data type mapping, which apart from simple data types includes geometry data types, will be explained explicitly further in this section.

Nullability

The nullability of the *element* of the *complex type* is not relevant to this particular method, since such element maps to the relational database table. The *elements* of the *simple type*, on the other hand, map to the columns of the relational database table. Therefore the nullability of the *elements* of the *simple type* will be discussed.

The nullability of the *element* can be defined using either the attribute *minOccurs* or the attribute *nillable* of the *element* definition. If the *minOccurs* attribute is set to ”0” or the *nillable* attribute is set to “true”, the corresponding *element* must be mapped to the nullable column of the relational database table. In all the other cases the *element* must be mapped to the non-nullable column of the relational database table. If neither the value of the attribute *minOccurs* nor the value of the attribute *nillable* is defined, the corresponding *element* is mapped to the nullable column of the relational database table by default.

In figure 3.11 the example of the use of the attributes *nillable* and *minOccurs* of the *element* definition is provided. After mapping the resulting relational database table *geom_test* definition will have the following columns:

- gid – nullable;
- geom – non-nullable;
- name – nullable.

The attribute *maxOccurs* of the *element* is not relevant for this particular method.

Primary Key and Foreign Keys

In this particular method the solution to define the primary and foreign keys of the relational database table is not provided. However, as you could see in the previous section, the Model, which includes the definition of the primary and foreign keys, is provided. It is designed in such a manner to ease the future extensions of the relational database schema definition generation from the specific XML Schema model.

```
<xs:key name="geom_test_key">
        <xs:selector xpath="geom_test"/>
        <xs:field xpath="gid"/>
</xs:key>
```

Figure 3.12 The Key Element of the XML Schema Example

Some suggestions regarding the definition of the primary and foreign keys of the relational database table will be provided. The primary key of the relational database table can be defined using the *key* structure of the XML Schema. The example is provided in figure 3.12. The name of the *key* element is not relevant to this particular method. Therefore it is ignored. The *selector* element using the means of XPath [XPATHv1] query language provides the information about the complex type element, which maps to a relational database table, to which the primary key is defined in this *key* element. The *field* element using the means of XPath [XPATHv1] provides the information, which sub-element of the complex type element must be mapped as a primary key column to the according relational database table. As the example in figure 3.12 shows, the element *gid* will be mapped as a primary key column of the relational database table *geom_test*.

The issue of foreign key definitions might be slightly more complicated. Therefore it is suggested to use the XLink [XLINKv1] structures to define links between the elements and then map it to the foreign key columns of the relational database tables. More details regarding this issue will not be covered in this thesis.

Naming Issues

There are some naming issues, when using the developed method. The relational database tables and columns within the table must have unique names. If this is not preserved during the generation or while defining the specific XML Schema, there maybe conflicts. The XML Schema preserves the uniqueness of the elements within the complex type element as long as those elements are in the same namespace. There also are other naming conventions that might produce conflicts. The issue of the reserved SQL keywords will be discussed in more detail in the next section.

Reserved SQL Keywords

The XML Schema for the reserved SQL keywords has been defined in this thesis. It can be used for checking the compatibility of the names of the specific XML Schema before the generation of the relational database schema definition. If at least one of the names identify with the reserved SQL keywords, the user receives the warning about it. The defined XML Schema for the reserved SQL words is illustrated in figure 3.13. The text view of this specific XML schema you can find in appendix A.

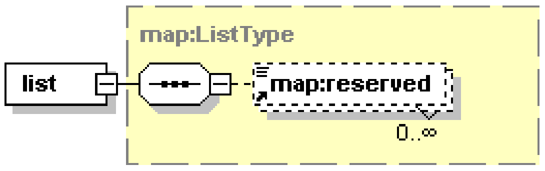


Figure 3.13 The XML Schema for the Reserved SQL Keywords

The XML Schema for the reserved SQL keywords is very simple. The root element is *list* and it aggregates the elements *reserved*. Each element *reserved* of the element *list* defines one reserved SQL keyword.

3.3.2.2. Data Type Mapping

The XML Schema for the mapping of any type of data types has been defined in this thesis. It is illustrated in figure 3.14. The text view of this specific XML schema you can find in appendix A. The root element *mappers* aggregates the elements *mapping*, *sql*, and *mapper*. The elements *mapping* and *sql* appear in each instance of this schema only once. The *mapping* element defines the mapping direction, namely *schema2java* or *java2sql*. The mapping direction *schema2java* implies the mapping from XML schema data types to Java data types. The mapping direction *java2sql*, on the other hand, implies the mapping from Java data types to SQL data types. The *sql* element is only required when the *mapping* element takes the value of *java2sql* and it provides the RDBMS name to which data types the java data types will be mapped.

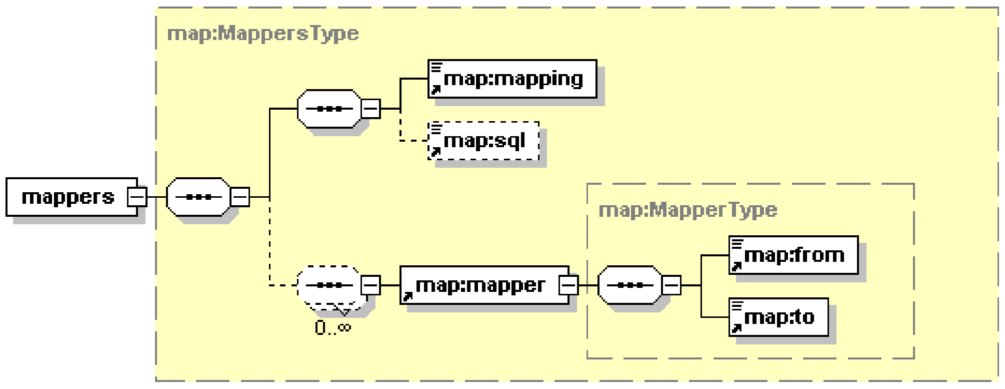


Figure 3.14 The XML Schema for the Mapping of the Data Types

The root element *mappers* also aggregates the elements *mapper*. Each of the *mapper* elements defines the mapping between different data types. For instance, if the *mapping* element takes the value of *schema2java*, then each *mapper* element will map one XML schema data type to one Java data type. The element *mapper* aggregates one element *from* and one element *to*, which hold the data type names. The data type denoted by the element *from* will be mapped to the data type denoted by the element *to*. For instance, if the *mapping* element takes the value of *schema2java* and the one of the *mapper* elements' element *from* holds the name of the XML schema data type *int*, then the element *to* of this *mapper* element holds logically the name of the Java data type *int*.

Using this XML schema's instance, all data types of the left side of the mapping direction can be mapped to all data types of the right side of the mapping direction.

The mapping of the geometry data types are also denoted in the instances of this schema together with the simple data types and in the same manner as simple data types. For instance, if the element *from* of a certain *mapper* element holds the name *PolygonPropertyType* of the geometry data type, then the element *to* of the same *mapper* element holds logically the name *Polygon* of the Java data type.

In the appendix B, one can find the text of XML file defining the mapping from XML Schema data types to Java data types, which also includes the mapping of the geometry data types.

3.3.2.3. Data Mappers

Data Mappers module performs the actual XML Schema structures to Model objects mapping process. The class diagram of this module is depicted in figure 3.15. Each Model class has its Data Mapper, which performs the mapping of the information specific to this class from the specific XML schema.

The *AbstractMapper* class is an abstract class, which captures the common functionality and stores the common information to all Data Mapper classes. The *AbstractMapper* class declares two abstract methods, namely *doLoad* and *load*, which must be implemented in a specific Data Mapper class for the information mapping, specific to the according Model class. The *doLoad* method returns the List of objects of the according Model class resulting after the mapping (refer to figure 3.7). The *load* method returns the object of the according Model class resulting after the mapping (refer to figure 3.7). Each specific Data Mapper class extends the *AbstractMapper* class and implements two above mentioned methods.

There are the following Data Mappers defined in the Data Mappers module (see figure 3.15 and figure 3.7):

- *TableMapper* class defines the mapping of the information relevant to the *Table* class;
- *AttributeMapper* class defines the mapping of the information relevant to the *Attribute* class;
- *GeometryMapper* class defines the mapping of the information relevant to the *Geometry* class;
- *PrimaryKeyMapper* class defines the mapping of the information relevant to the *PrimaryKey* class;
- *ForeignKeyMapper* class defines the mapping of the information relevant to the *ForeignKey* class;

- *DataTypeMapper* class defines the mapping of the information relevant to the *DataType* class;
- *GeometryTypeMapper* class defines the mapping of the information relevant to the *GeometryType* class.

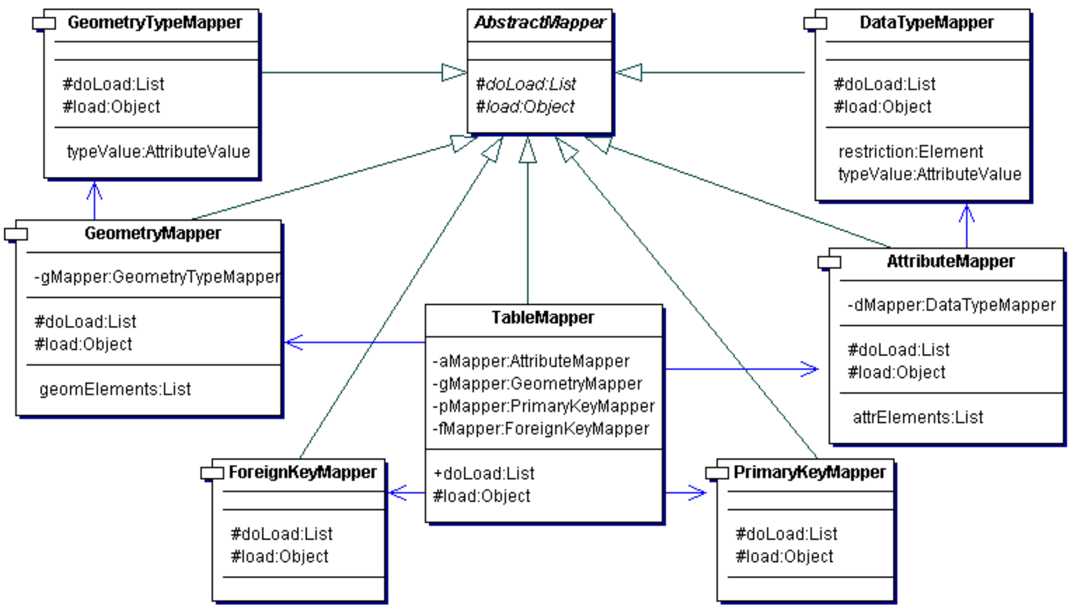


Figure 3.15 Data Mappers Class Diagram

To better understand how the Data Mappers work please refer to figure 3.16, which illustrates the mapping process. The term *user* will be used in this section to denote the module, which uses the services provided by Data Mappers module. The mapping process starts when the *user* of the Data Mappers module calls the *TableMapper's doLoad* method. The *TableMapper* takes one of the relational database table definitions, which are found in the specific XML Schema passed by the user. The *TableMapper* maps the name of the relational database table to the according *Table* object's property.

The *TableMapper* passes all column definitions, found for this specific relational database table definition, to the *AttributeMapper* and calls the *AttributeMapper*'s *doLoad* method. For each column definition received, the *AttributeMapper* maps its name and nullability to the according *Attribute* object's properties. It also passes each column's data type name and the data type restriction definitions to the *DataTypeMapper*, which maps the data type name and its restriction to the according *DataType* object's properties. Then *DataTypeMapper* returns the resulting *DataType* object to the *AttributeMapper*. The *AttributeMapper* includes it into the *Attribute* object and puts the *Attribute* object to the list of the *Attribute* objects. After all column definitions have been mapped to the *Attribute* objects and the *Attribute* objects reside in the list, the list of the *Attribute* objects represents all column definitions of the specific relational database table and is returned by the *AttributeMappe*r to the *TableMapper*, where it is saved in the according property of the specific *Table* object.

The primary key definition of the specific relational database table definition, found in the specific XML Schema, is then passed to the *PrimaryKeyMapper* and the *TableMapper* calls the *load* method of the *PrimaryKeyMapper*. The *PrimaryKeyMapper* maps the name of the primary key to the according property of the *PrimaryKey* object. This *PrimaryKey* object is then returned to the *TableMapper* by the *PrimaryKeyMapper* to be saved in the according property of the specific *Table* object.

The foreign key definitions of the specific relational database definition, found in the specific XML Schema, are passed to the *ForeignKeyMapper* and the *TableMapper* calls the *doLoad* method of the *ForeignKeyMapper*. For each foreign key definition for this specific relational database table definition, the *ForeignKeyMapper* maps the name of the foreign key and its nullability to the according properties of the *ForeignKey* object. The *ForeignKeyMapper* also maps the referenced table name to the according property of the *ForeignKey* object. The specific *ForeignKey* object is put to the list of *ForeignKey* objects. When all *ForeignKey* objects have been mapped, the *ForeignKeyMapper* returns the list of the *ForeignKey* objects to the *TableMapper*, where it is saved in the according property of the specific *Table* object.

The geometry definitions of the specific relational database definition, found in the specific XML Schema, are finally passed to the *GeometryMapper* and the *TableMapper* calls the *doLoad* method of the *GeometryMapper*. For each geometry definition of the specific relational database table definition, the *GeometryMapper* maps the name of the geometry to the according property of the *Geometry* object. It also passes the geometry type name of each geometry declaration to the *GeometryTypeMapper*, which maps the geometry type name to the according *GeometryType* object's property. The *GeometryTypeMapper* returns then the resulting *GeometryType* object to the *GeometryMapper*. The *GeometryMapper* includes it into the *Geometry* object and puts the *Geometry* object to the list of the *Geometry* objects. After all geometry definitions have been mapped to the *Geometry* objects and the *Geometry* objects reside in the list, the list of the *Geometry* objects represents all geometry definitions of the specific relational database table and is returned by the *GeometryMapper* to the *TableMapper*, where it is saved in the according property of the specific *Table* object.

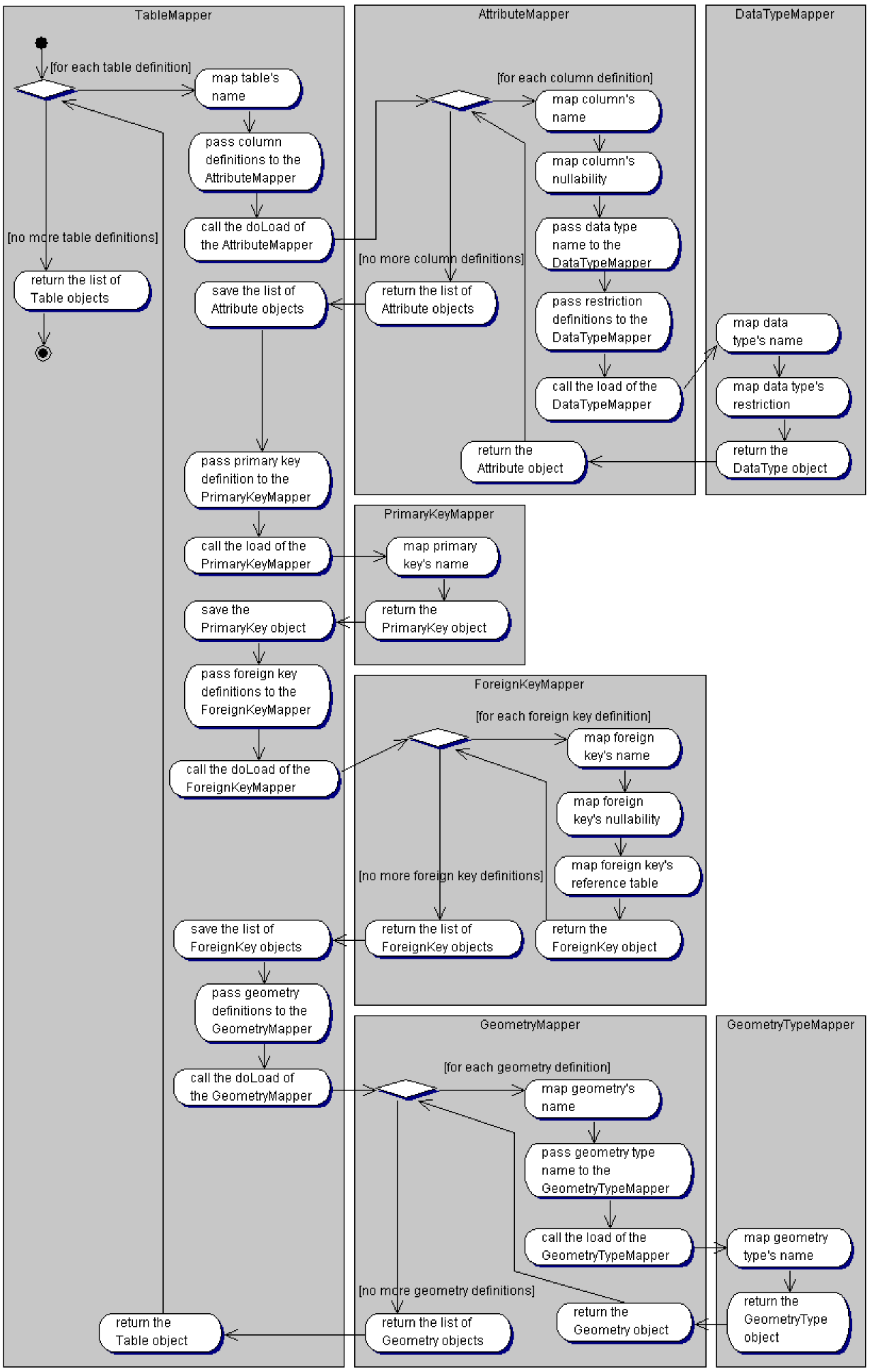

TableMapper
[for each table definition]
map table's name
pass column definitions to the AttributeMapper
call the doLoad of the AttributeMapper
[no more table definitions]
return the list of Table objects
save the list of Attribute objects
pass primary key definition to the PrimaryKeyMapper
call the load of the PrimaryKeyMapper
save the PrimaryKey object
pass foreign key definitions to the ForeignKeyMapper
call the doLoad of the ForeignKeyMapper
save the list of ForeignKey objects
pass geometry definitions to the GeometryMapper
call the doLoad of the GeometryMapper
return the Table object
AttributeMapper
[for each column definition]
map column's name
map column's nullability
pass data type name to the DataTypeMapper
pass restriction definitions to the DataTypeMapper
call the load of the DataTypeMapper
[no more column definitions]
return the list of Attribute objects
return the Attribute object
DataTypeMapper
map data type's name
map data type's restriction
return the DataType object
PrimaryKeyMapper
map primary key's name
return the PrimaryKey object
ForeignKeyMapper
[for each foreign key definition]
map foreign key's name
map foreign key's nullability
map foreign key's reference table
[no more foreign key definitions]
return the list of ForeignKey objects
return the ForeignKey object
GeometryMapper
[for each geometry definition]
map geometry's name
pass geometry type name to the GeometryTypeMapper
call the load of the GeometryTypeMapper
[no more geometry definitions]
return the list of Geometry objects
return the Geometry object
GeometryTypeMapper
map geometry type's name
return the GeometryType object

Figure 3.16 Data Mappers Activity Diagram (Mapping Process)

Finally all properties of the specific *Table* object are mapped and the *TableMapper* puts the resulting *Table* object to the list of *Table* objects by. The *TableMapper* runs the described mapping process for each relational database table definition found in the specific XML Schema and all resulting *Table* objects are at the end put to the list of the *Table* objects. When all relational database table definitions has been mapped to the *Table* objects and put to the list, the list of *Table* objects is representing the specific relational database schema definition in the specific XML Schema and is then returned to the *user* for the further use.

The mapping process executed by the Data Mappers module has been described. It is illustrated in figure 3.16.

3.3.3. Pseudo-Presentation Layer

As it was mentioned above, the Pseudo-Presentation Layer encapsulates SQL Builders, which are responsible for the generation of the RDBMS-specific SQL DDL scripts from the Model objects.

3.3.3.1. SQL Builders

The class diagram of the SQL Builders module is provided in figure 3.17. The interface *SQLBuilder* is defined, which declares the methods to create SELECT, INSERT, UPDATE, DELETE, CONSTRAINT, ALTER TABLE, and CREATE TABLE SQL statements.

The *GenericSQLBuilder* class implements the *SQLBuilder* interface and holds the names of the SQL keywords in its properties, which are most frequently used in the mentioned SQL statements. The *GenericSQLBuilder* class defines the way to build the most generic SQL structures. For example, it defines the concatenation of strings that hold SELECT block, FROM block, and WHERE block in the SELECT SQL statement, but it does not define the pattern of how the table names with their aliases are arranged in the FROM block. The latter task is left for the vendor-specific SQL Builders.

The *AbtractVendorSQLBuilder* abstract class extends the *GenericSQLBuilder* class. It declares the methods to build the more vendor-specific parts of the SQL statements like it was described above. It declares abstract methods for building table name definition, attribute definition, primary key definition, and foreign key definition for the CREATE TABLE SQL statement and table definitions and column definitions for the SELECT SQL statement. The *AbstractVendorSQLBuilder* abstract class knows about the RDBMS vendor type chosen for the mapping and it holds the list of the reserved SQL keywords. Although the reserved SQL keywords are originally stored in the XML file, which is an instance of the XML Schema illustrated in figure 3.13, but the real check of the names used in the relational database table definition is done in the SQL Builders module by the chosen vendor-specific SQL Builder.

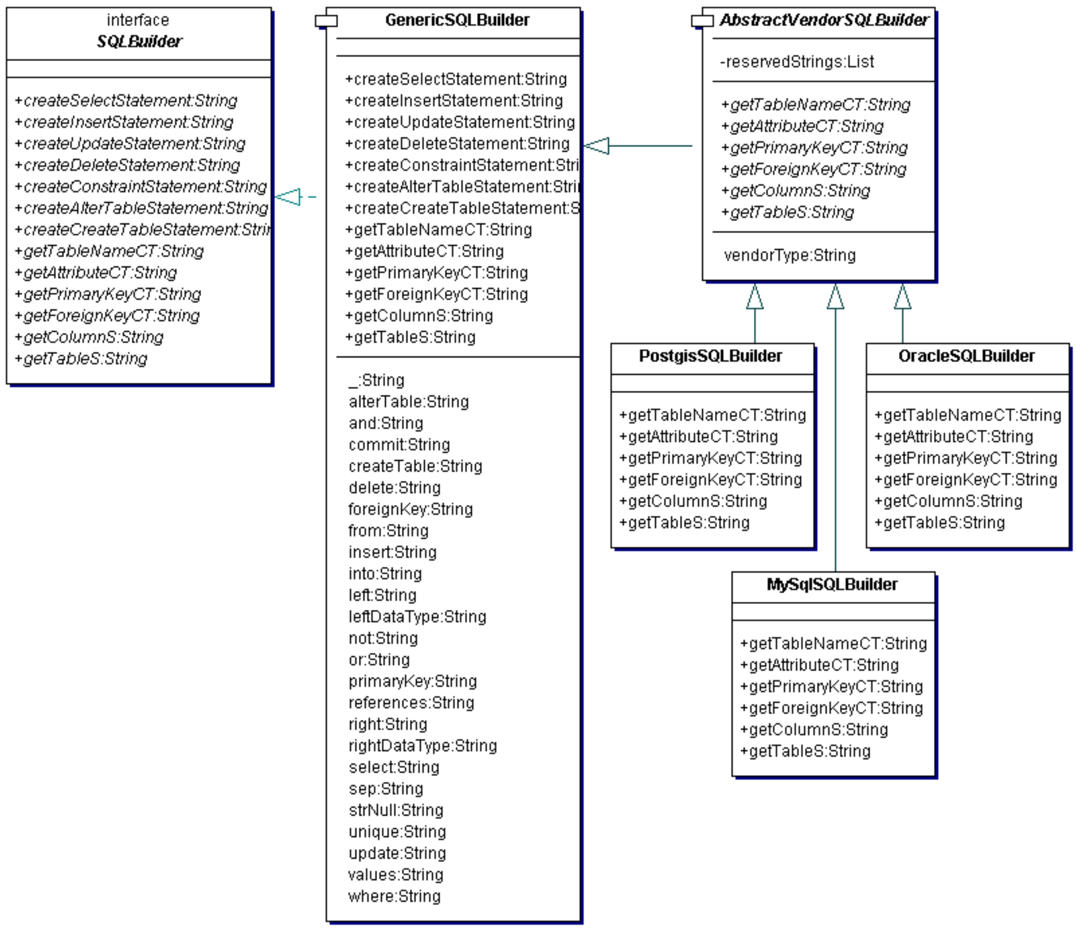


Figure 3.17 SQL Builders Class Diagram

The vendor-specific SQL Builders extend the *AbstractVendorSQLBuilder* abstract class and implement all the abstract methods declared in the *AbstractVendorSQLBuilder* abstract class. The vendor-specific SQL Builders can also reload the properties of the *GenericSQLBuilde*r class that differ from the vendor-specific SQL with the vendor-specific values. The properties of the *GenericSQLBuilder* class are inherited by the vendor-specific SQL Builders through the *AbstractVendorSQLBuilder* abstract class. This way the vendor-specific parts of the SQL statements are built in the vendor-specific SQL Builders. In figure 3.17 one can find three vendor-specific SQL Builders: *PostgisSQLBuilder*, *OracleSQLBuilder*, and *MySqlSQLBuilder*. Only *PostgisSQLBuilder* is fully designed, the others can easily be finished based on the example of the *PostgisSQLBuilder*. More vendor-specific SQL Builders can be easily added by extending the *AbstractVendorSQLBuilder* abstract class reloading the vendor-specific properties of the *GenericSQLBuilder* class and implementing the vendor-specific methods of the *AbstractVendorSQLBuilder* abstract class.

3.3.3.2. SQL Scripts Generation Process

The SQL scripts represent the RDBMS-specific relational database schema definition, which is the end-result of the designed product. The SQL scripts generation process is illustrated in figure 3.18 using the *PostgisSQLBuilder*. The *PostgisSQLBuilder* is the only vendor-specific SQL Builder, which has been fully designed in this thesis.

The term *user component* will be used to denote the component that is using the *PostgisSQLBuilder* for the SQL scripts generation. In the process the *PostgisSQLBuilder* is using the Model objects, created by other layers, as an information source about the existing relational database schema definition. The PostgisSQLBuilder fetches the necessary data from the according Model objects and performs the data type mapping from the Java data types to the Postgis SQL data types. It performs the data type mapping in the same manner as it is described in section 3.2.2.2 Data Type Mapping and uses the XML file, which is the instance of the XML Schema illustrated in figure 3.14. The example of such file one can find in the appendix B.

The SQL scripts generation process starts when the *user component* chooses one of the vendor-specific SQL Builders. In this particular case, the *PostgisSQLBuilder* will perform the generation. The *PostgisSQLBuilder* takes one of the *Table* objects from Model objects and creates Postgis-specific parts of the CREATE TABLE statement. The typical CREATE TABLE statement is provided in figure 3.19. The Postgis-specific parts are underlined, namely the name of the table, column definitions including the data types, primary key name, foreign key name and the referenced table’s name and column. After the Postgis-specific parts are generated, the generic CREATE TABLE statement parts are generated using the methods defined in *GenericSQLBuilder* and inherited by *PostgisSQLBuilder* through extension of *AbstractVendorSQLBuilder*. The generic CREATE TABLE statement parts in figure 3.19 are the following keywords: CREATE TABLE, PRIMARY KEY, FOREIGN KEY, the parentheses enclosing the relational database table definition, and the symbol “;”. After those are generated from the original or reloaded *GenericSQLBuilder*’s properties, the Postgis-specific and generic SQL parts are concatenated in the proper order, as it is depicted in figure 3.19, and the resulting CREATE TABLE statement is saved in the list of the SQL scripts.

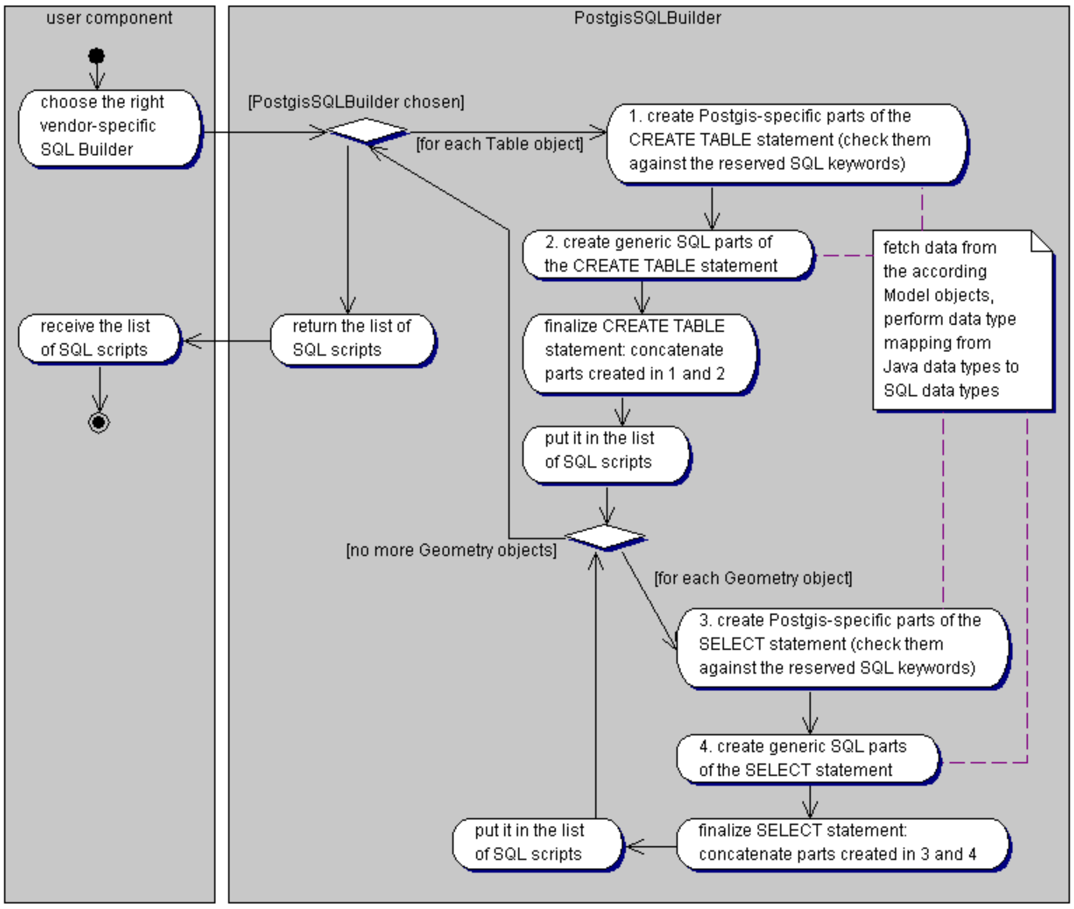
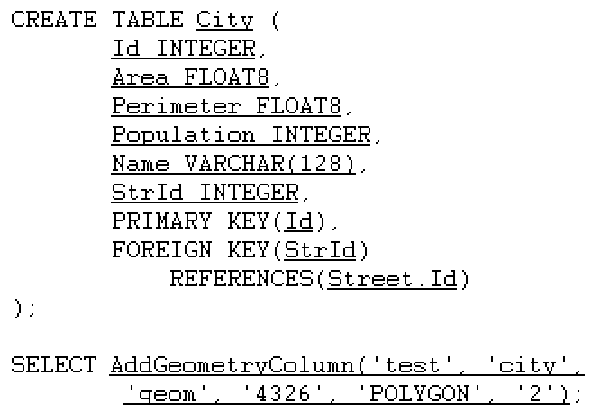

user component
PostgisSQLBuilder
choose the right vendor-specific SQL Builder
[PostgisSQLBuilder chosen]
[for each Table object]
1. create Postgis-specific parts of the CREATE TABLE statement (check them against the reserved SQL keywords)
2. create generic SQL parts of the CREATE TABLE statement
fetch data from the according Model objects, perform data type mapping from Java data types to SQL data types
receive the list of SQL scripts
return the list of SQL scripts
finalize CREATE TABLE statement: concatenate parts created in 1 and 2
put it in the list of SQL scripts
[no more Geometry objects]
[for each Geometry object]
3. create Postgis-specific parts of the SELECT statement (check them against the reserved SQL keywords)
4. create generic SQL parts of the SELECT statement
put it in the list of SQL scripts
finalize SELECT statement: concatenate parts created in 3 and 4

Figure 3.18 *PostgisSQLBuilder* Activity Diagram

The *PostgisSQLBuilder* performs the similar procedure with each *Geometry* object from the list found in the property geometries of this specific *Table* object. The typical resulting SELECT SQL script can be seen in figure 3.19 and it represents the definition of the *2-dimensional* geometry column *geom* for the relational database *test*, relational database table *city*, in the referential coordinate system identified by id *4326*, of the geometry type *POLYGON*. The underlined part is Postgis-specific and the keyword SELECT is a generic SQL keyword. After the SELECT SQL statements defining all geometry columns are generated and put into the SQL scripts list, the other *Table* object is taken from available list of *Table* objects and the described process of SQL scripts generation is repeated.



```
CREATE TABLE City (
       Id INTEGER,
       Area FLOAT8,
       Perimeter FLOAT8,
       Population INTEGER,
       Name VARCHAR(128),
       StrId INTEGER,
       PRIMARY KEY(Id),
       FOREIGN KEY(StrId)
           REFERENCES(Street.Id)
);

SELECT AddGeometryColumn('test', 'city',
        'geom', '4326', 'POLYGON', '2');
```

Figure 3.19 Typical CREATE TABLE and SELECT AddGeometryColumn() SQL statements in Postgis SQL

After all *Table* objects from the available list of *Table* objects are processed, the resulting list of SQL scripts is returned to the *user component* for the further use.

3.4. Limitations

There are several limitations of the model for relational database schema definition generation from XML Schema, which have been identified after the design phase. They are the following:

- there is no support for defining and mapping the uniqueness of the relational database table's column. The support must be provided in the further development of the model;
- there are only suggestions for the definition and mapping of the primary and foreign key. This issue must still be analyzed and the solution must be provided in the further development of the model;
- there is no support for the definition and mapping of the user-defined types. Such support would require nested complex type handling. The support must be provided in the further development of the model, since it is already partially supported in one of the implementations of the OpenGIS specifications (*deegree* [DGREE04]);
- the support of new RDBMSs is achieved by extending the *AbstractVendorSQLBuilder* abstract class, reloading the differing properties of the *GenericSQLBuilder* class, and implementing all abstract methods of the *AbstractVendorSQLBuilder* abstract class. Although this is quite flexible, but these responsibilities could be transferred to XML files. One could define the syntax of the new vendor-specific SQL as an XML

file and in such particular way support the new RDBMS. It must be considered in the further development of the model.

3.5. Summary of the Chapter

The model of the generic data importer has been introduced. The problem statement has been provided and the three-layer architecture of the relational database schema generation from XML Schema model has been defined to solve the provided problem. The three-layer architecture is consisting from the following layers: data layer, logic layer, and pseudo presentation layer. The overall process description within the defined three-layer architecture has been described. The design decisions that have been taken have been described and the detailed design of each layer of the defined architecture of the model has been defined. The processes within each layer of the defined architecture of the model have also been described. Several limitations of the defined architecture of the model for relational database schema generation from XML Schema have been identified. They are the following: no support for uniqueness, primary key, foreign key, user-defined types (nested complex types), and limited flexibility in extending for the support of new RDBMS.

4. Implementation of the Model

The list of technologies that will be used for the implementation of the designed model for relational database schema generation from XML Schema will be provided with the short description. The specificities of the implementation of this model will be also described. The example of using the implemented model will be provided and the qualitative evaluation of the implementation will be presented.

4.1. Technologies Used for the Implementation

The model for the relational database schema from XML Schema generation was implemented using Java programming language and JDOM [JDOM04] for Java representation of the XML document. For the use of the implemented component it is recommended to plug it into the JDBC [JDBC04] enabled component for the dynamic creation of the specific relational database schemas in chosen RDBMS.

4.1.1. Java Programming Language

Java programming language allows writing components, which can run on various platforms, since those programs are run on (interpreted by) the Java Virtual Machine (Java VM) program. So the Java programming language allows writing platform-independent components. [JAVA04]

4.1.2. JDBC technology

The JDBC technology is an API that provides the cross-DBMS connectivity to a wide range of the SQL databases. This technology was not used for the implementation of the designed model, but it is suggested to plug the implemented component into the JDBC enabled component for the dynamic creation of the specific relational database schemas in chosen RDBMS. The JDBC allows arranging the API calls to any database in the common manner. One can specify the RDBMS by choosing the RDBMS driver. In combination with the implemented component, JDBC Technology provides a kind of RDBMS-independency. [JDBC04]

4.1.3. JDOM

The JDOM is a Java representation of the XML document. It represents the XML document for efficient reading, manipulation, and writing. The JDOM has a lightweight and fast API, which is an alternative to DOM and SAX [SAX04]. On the other hand, it integrates with both, DOM and SAX. JDOM is a document object model that uses XML parsers to build documents. It can use nearly any XML parser available on the market. By default it uses JAXP parser. So the JDOM provides some kind of XML Parser independency. [JDOM04]

4.2. Implementation Specificities

Further in this section the term *generator* will be used to identify the implemented model for the relational database schema generation from XML Schema.

The *generator* implements most of the functionalities of the designed model for the relational database schema generation from XML Schema. The following functionalities where not implemented in the *generator*:

- primary key definition generation;
- foreign key definition generation.

The *generator* as it is implemented satisfies the requirements of the project, in the area of which the *generator* was designed and implemented.

The configuration file for the *generator* was defined as an XML file, which XML Schema is illustrated in figure 4.1. The text view of this XML schema and the configuration file example is provided in the appendix C.

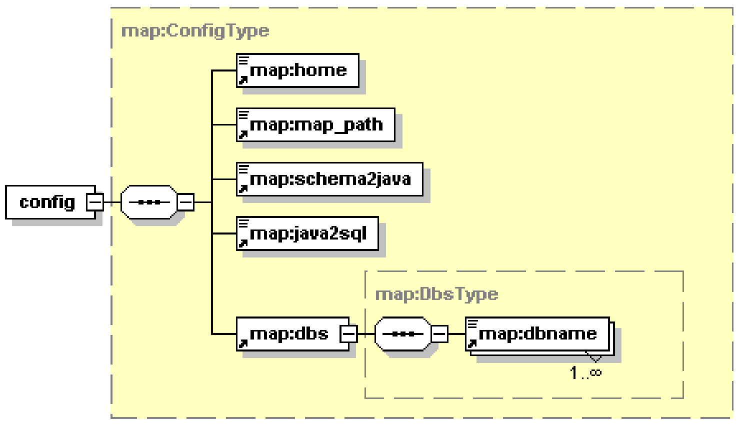


Figure 4.1 The XML Schema for the Configuration File of the *Generator*

The element *home* specifies the full path to the directory where the *generator* resides. The element *map_path* specifies the path relative to the *home* element's value where the mapping files reside. The element *schema2java* specifies the mapping file's name, which defines the data type mapping from XML Schema data types to Java data types. The element *java2sql* specifies the file name, which holds the reserved SQL keywords list. The last but not least, the *dbs* element holds the *dbname* elements, which specify the names of the RDBMS, which are supported by the *generator*. At the same time each *dbname* element specifies the mapping file name, which defines the data type mapping from Java data types to SQL data types of the specific RDBMS.

The extra functionality of the *generator* was implemented. It generates the report about the process of the generation. The report XML Schema is illustrated in figure 4.2. The text view of this XML schema and the example of the report file is provided in the appendix C.

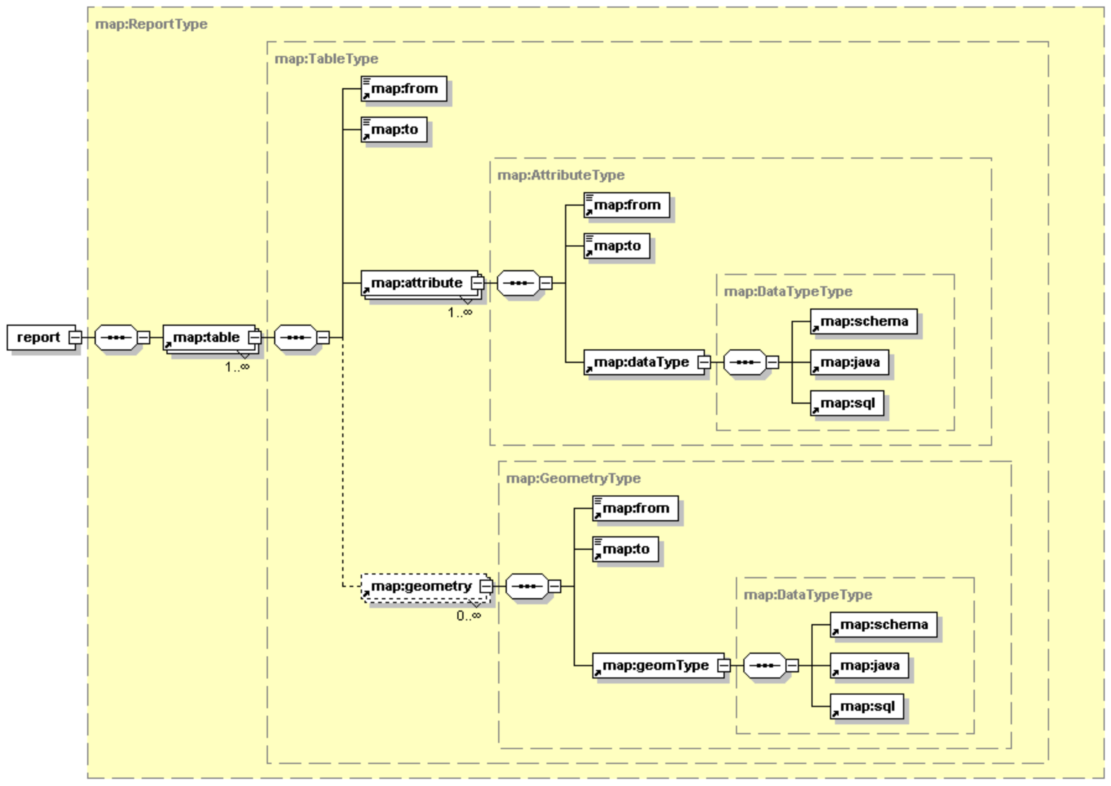


Figure 4.2 The XML Schema for the Report File of the *Generator*

The report XML Schema's root element is *report*, which consists of the sequence of the *table* elements. Each *table* element consists of the following sequence of the elements: *from*, *to*, *attribute* (multiple appearance), and *geometry* (multiple appearance). Element *from* provides the feature name from the original XML schema. Element *to* provides the table name to which the feature name was mapped by the *generator*. Each *attribute* element consists of the following sequence of the elements: *from*, *to*, and *dataType*. The same sequence of the elements is aggregated by each *geometry* element. The *geometry* elements are optional. The elements *from* and *to* from the *attribute* and *geometry* elements have the same semantics as the elements *from* and *to* from the element *table*. The element *from* provides the pre-generation name of the attribute or geometry and the element *to* provides the post-generation name of the attribute or geometry. The element *dataType* consist of the following sequence of the elements: *schema, java,* and *sql*. It provides the data type mapping information. The element *schema* provides the XML Schema data type name. The element *java* provides the Java data type name. The element *sql* provides the SQL data type name.

The model for relational database schema generation from XML Schema was implemented as a component, which can be used as an imported library in the Java code of ones own application by calling the static method *process()* of the *Main* object providing different parameters, or can be called as a executable providing the parameters in the command line.

The parameters needed for the generation to be executed are the following:

- the specific XML Schema document from which the relational database schema must be generated (can be provided as a full path to the file in the file system from the command line or the *java.io.Reader* object to the static *process()* method of the *Main* object);
- the RDBMS type, which exact possible values are specified in the configuration file of the generator under the element *dbname*: e.g. postgis, oracle, mysql;
- true if the reserved SQL keywords provided in the specific XML file must be taken into account, false if not;
- true if the report about the generation process, as it is defined by the report XML Schema illustrated in figure 4.2, must be generated, false if not;
- the name of the relational database for which the relational database schema will be created;
- spatial reference system id number (srid). There are many standards and types of spatial reference systems. One needs to specify, in which spatial reference system the coordinates of the geometries are provided;
- dimension of the geometries. One needs to specify if the geometries are provided as 2- or 3-dimentional objects.

4.3. Example of the Use of the Implemented Model

As an example, the specific XML Schema document, generated by the OpenGIS Web Feature Service (WFS) implementation called G*eoServer* [GEOS04], will be used.

The GeoServer provides the functionality to generate the default specific XML Schema document from the given data source using the following DescribeFeatureType [WFSv1.0] request to the GeoServer:
http://troubadix.wb.tu-harburg.de:8180/geoserver/wfs?request=DescribeFeatureType &typeName=buenz:Buenzau_nutzung.
The GeoServer has generated the XML Schema document, which is depicted in figure 4.3. The original data source was the ESRI Shapefile [SHP98], one of the standard to store the geographical data (geodata). The document defines the feature *Buenzau_nutzung*, which has the following elements:

- *the_geom* is an element of a geometry type *MultiPolygonPropertyTyp*e as it is defined in [SFSv1.1];
- *AREA* is an element of a simple type *double*;
- *PERIMETER* is an element of a simple type *double*;
- *NUTZUNG_* is an element of a simple type *int*;
- *NUTZUNG_ID* is an element of a simple type *int*;
- *NS* is an element of a simple type *int*;
- *BRD* is an element of a simple type *int*;
- *SZENE* is an element of a simple type *int*;
- *SPHEROID* is an element of a simple type *string*;
- *TKNR* is an element of a simple type *string*;
- *NS1* is an element of a simple type *int*;
- *NS2* is an element of a simple type *int*;
- *NS3* is an element of a simple type *int*;
- *NUTZUNG* is an element of a simple type *string*.

All the mentioned elements of the feature *Buenzau_nutzung* are defined with attributes *nillable* set to *true* and *minOccurs* set to *0*. This specific XML Schema resides in the file on the files system under the following full path: */{full_path}/buenzau.xsd*.

The *generator* will be called, which we have named RedGenie (RElational Database schema GENerator), as an executable from the command line as follows:
java RedGenie.main.Main /{full_path}/buenzau.xsd postgis true true test 31467 2
The parameters have the following meaning accordingly:

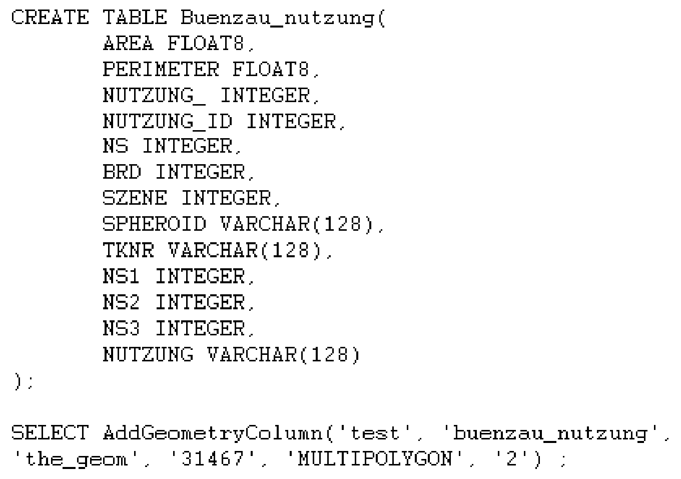
- the specific XML Schema resides under */{full_path}/buenzau.xsd*;
- generate relational database schema definition for *Postgis* RDBMS;
- check the keywords of the definition against reserved SQL keywords;
- generate the report file;
- generate the relational database schema definition for database with the name *test*;

- the coordinates of the geometries are provided in the spatial reference system, which id is *31467*;
- the geometries are *2-dimentional* objects;

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema
  targetNamespace="http://wb.tuhh.de/buenz"
  xmlns:buenz="http://wb.tuhh.de/buenz"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified" version="1.0">
<xs:import namespace="http://www.opengis.net/gml" schemaLocation="http://troubadix.wb.tu-
harburg.de:8180/geoserver/data/capabilities/gml/2.1.2/feature.xsd"/>
<xs:complexType xmlns:xs="http://www.w3.org/2001/XMLSchema" name="Buenzau_nutzung_Type">
    <xs:complexContent>
        <xs:extension base="gml:AbstractFeatureType">
            <xs:sequence>
                <xs:element name="the_geom" minOccurs="0" nillable="true"
                     type="gml:MultiPolygonPropertyType"/>
                <xs:element name="AREA" minOccurs="0" nillable="true" type="xs:double"/>
                <xs:element name="PERIMETER" minOccurs="0" nillable="true" type="xs:double"/>
                <xs:element name="NUTZUNG_" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="NUTZUNG_ID" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="NS" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="BRD" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="SZENE" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="SPHEROID" minOccurs="0" nillable="true" type="xs:string"/>
                <xs:element name="TKNR" minOccurs="0" nillable="true" type="xs:string"/>
                <xs:element name="NS1" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="NS2" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="NS3" minOccurs="0" nillable="true" type="xs:int"/>
                <xs:element name="NUTZUNG" minOccurs="0" nillable="true" type="xs:string"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name='Buenzau_nutzung' type='buenz:Buenzau_nutzung_Type'
      substitutionGroup='gml:_Feature'/>
</xs:schema>
```

Figure 4.3 Example of a Specific XML Schema Document Generated by GeoServer

The generated relational database schema definition is provided in figure 4.4. The report file is provided in the appendix C. The generated relational database schema definition's CREATE TABLE SQL statement defines the relational database table named *Buenzau_nutzung* with the columns corresponding to the elements defined in the specific XML Schema provided in figure 4.3. Each data type was mapped as it was defined in the data type mapping file for Postgis RDBMS. The contents of this data type mapping file can be found in appendix B. The geometry column is defined by relational database schema definition's SELECT SQL statement, which uses the function *AddGeometryColumn()* with the following parameters accordingly: *test* – the name of the relational database, for which the geometry column is created, *buenzau_nutzung* – the name of the relational database table, for which the geometry column is created, *the_geom* – the name of the geometry column, *31467* – the id of the spatial referential system, *MULTIPOLYGON* – the data type name of the geometry, *2* – the dimension of the geometry objects.



```
CREATE TABLE Buenzau_nutzung(
       AREA FLOAT8,
       PERIMETER FLOAT8,
       NUTZUNG_ INTEGER,
       NUTZUNG_ID INTEGER,
       NS INTEGER,
       BRD INTEGER,
       SZENE INTEGER,
       SPHEROID VARCHAR(128),
       TKNR VARCHAR(128),
       NS1 INTEGER,
       NS2 INTEGER,
       NS3 INTEGER,
       NUTZUNG VARCHAR(128)
);

SELECT AddGeometryColumn('test', 'buenzau_nutzung',
'the_geom', '31467', 'MULTIPOLYGON', '2') ;
```

Figure 4.4 Example of the Relational Database Schema Definition for Postgis RDBMS Generated by the *Generator* (RedGenie)

The resulting SQL scripts (see figure 4.4) must be executed in Postgis RDBMS to create the relational database schema to be able to populate it with data. One can do it manually, but it is suggested to integrate the *generator* to ones own application and execute the resulting SQL scripts using the JDBC technology. This way one would automate the process of data import to the relational databases by eliminating the necessity of human intervention to the process of creation of the relational database schemas.

4.4. Evaluation of the Implementation

The implementation of the *generator* was successful. All fully designed features of the model for relational database schema generation form XML Schema were implemented. The ones not fully designed were also not implemented, namely primary key definition and foreign key definition generation. So the generator (RedGenie) is rather precise implementation of the originally designed model for relational database schema generation from XML Schema.

4.5. Summary of the Chapter

The list of technologies that have been used for the implementation of the designed model for relational database schema generation from XML Schema has been provided, namely Java programming language and JDOM. It has been suggested always to use the implemented component together with the JDBC technology, to automate the data import to the relational databases. The specificities of the *generator* have been described. It has been noted, that the functionality of foreign key and primary key definitions generation was not implemented, since it was not fully designed. The example of using the *generator* with the specific XML Schema has been presented defining the feature *Buenzau_nutzung*, from which the relational database schema definition has been successfully created. The evaluation of the implementation of the *generator* has been described as rather precise, since all fully designed functionalities of the original model has been implemented in the *generator* (RedGenie).

5. Conclusions

5.1. Summary

The structures and the vocabulary relevant to this thesis from XML Schema standard have been introduced, namely element declarations, simple and complex type definitions, the attributes nillable and minOccurs of the element declaration, substitution groups mechanism, extension mechanism, and key definitions. The geometry and feature schemas defined in Open GIS Geography Markup Language (GML) Implementation Specification Version 2.1.2 [GMLv2.1] have been introduced and SQL geometry types defined in Open GIS Simple Features Specification for SQL Revision 1.1 [SFSv1.1] have been provided.

The requirements for the relational database schema generation from XML schema model have been formulated and the three-layer architecture for the design of the mentioned model has been defined. The three layers are the following: data layer, logic layer, and pseudo presentation layer. The model-view-controller pattern has also been used for the definition of the architecture. The overall process of the relational database schema generation from the XML Schema description within the three-layer architecture has been described and the design decisions have been taken. Furthermore, the detailed design of each layer of the defined three-layer architecture of the model has been defined and the processes within each of the layers have been described. Then several limitations of the defined model have been identified: no support for the uniqueness of the relational database table’s column, primary key, foreign key, user-defined types (nested complex types), and limited flexibility in extending the support to more RDBMSs.

A list and short description of the technologies we have used for the implementation of the designed model for relational database schema generation from XML Schema have been provided, namely Java programming language and JDOM. It has been suggested to use the implemented component together with the JDBC technology, to automate the data import process to the relational databases. The specific implementation decisions that have been made during the implementation have been discussed and it has been noted that the support for foreign and primary key definitions generation has not been implemented, because of the lack of design. The example case of the relational database schema generation from the geographical feature definition Buenzau_nutzung has been presented. The relational database schema definition has been successfully created by the implemented component.

Finally, the implementation has been evaluated as being rather successful, since all fully designed functionalities have been implemented.

This thesis has shown that there is a lack of components in Java development environment capable of generic data import to the relational databases with the relational database schema definition generation from specific XML Schema with a support for geographical features. One component from such a system has been designed and implemented, namely a component for OpenGIS-compliant relational database schema generation from the specific XML Schema. The implemented component can be used and on demand extended in the projects that are and/or will be in progress in River and Coastal Engineering Department in Technical University Hamburg-Harburg.

5.2. Outlook

In the future, the design for the OpenGIS-compliant relational database schema generation from the specific XML Schema model must be extended towards the support of the identified limitations. The support for the uniqueness mapping must be designed as soon as possible. The design of the support for the primary and foreign key definition generations must be defined on demand, which should increase soon, since some of the implementations of the OpenGIS specifications are already announcing the support for the nested complex types. The flexibility regarding the extension to the support of more RDBMSs is not very limited. The increase of the flexibility should be considered as “nice to have” feature.

All the future designs should be implemented on the spot for testing and usage.

The report XML document generated by the implemented component can be used and should be used not only for the overview of the results of the process, but also for the dynamic configuration of the Geoservices implemented according to OpenGIS Specifications, e.g. Web Feature Service (WFS) [WFSv1.0]. The mentioned report XML document contains all necessary information and with the help of XSL transformations [XSLTv1.0] it can be used for the mentioned configuration.

The other components of the generic data importer described in the introduction of this thesis should be also implemented, namely relational database schema creation module, data import module, and relational database schema update module.

Appendix A. XML Schemas Related to Model

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        targetNamespace="http://troubadix.wb.tu-harburg.de/map"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:map="http://troubadix.wb.tu-harburg.de/map"
        version="2.1.2">
        <element name="reserved" type="string"/>
        <element name="list" type="map:ListType"/>
        <complexType name="ListType">
                <sequence>
                        <element ref="map:reserved" minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
        </complexType>
</schema>
```

Figure A.1 The XML Schema for Storing Reserved SQL Keywords

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
        targetNamespace="http://troubadix.wb.tu-harburg.de/map"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:map="http://troubadix.wb.tu-harburg.de/map"
        version="2.1.2">
        <element name="mapping" type="map:MappingType"/>
        <element name="sql" type="NMTOKEN"/>
        <element name="mappers" type="map:MappersType"/>
        <element name="mapper" type="map:MapperType"/>
        <element name="from" type="token"/>
        <element name="to" type="token"/>
        <simpleType name="MappingType">
                <restriction base="NMTOKEN">
                        <enumeration value="schema2java"/>
                        <enumeration value="java2sql"/>
                        <!--more types to be added if needed-->
                </restriction>
        </simpleType>
        <complexType name="MapperType">
                <sequence minOccurs="1" maxOccurs="1">
                        <element ref="map:from"/>
                        <element ref="map:to"/>
                </sequence>
        </complexType>
        <complexType name="MappersType">
                <sequence minOccurs="1" maxOccurs="1">
                        <sequence minOccurs="1" maxOccurs="1">
                                <element ref="map:mapping" minOccurs="1" maxOccurs="1"/>
                                <element ref="map:sql" minOccurs="0" maxOccurs="1"/>
                                        <!--required when mappingType=java2sql-->
                        </sequence>
                        <sequence minOccurs="0" maxOccurs="unbounded">
                                <element ref="map:mapper"/>
                        </sequence>
                </sequence>
        </complexType>
</schema>
```

Figure A.2 The XML Schema for Data Type Mapping Definition

Appendix B. Instances of XML Schemas Related to Model

```
<?xml version="1.0" encoding="UTF-8"?>
<map:list
        xmlns="http://troubadix.wb.tu-harburg.de/map"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:map="http://troubadix.wb.tu-harburg.de/map"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://troubadix.wb.tu-harburg.de/map
                http://troubadix.wb.tu-harburg.de/map/java2sql.xsd">
        <!-- reserved SQL words: as it is given in SQL99 -->
        <map:reserved>ABSOLUTE</map:reserved>
        <map:reserved>ACTION</map:reserved>
        <map:reserved>ADD</map:reserved>
        <map:reserved>AFTER</map:reserved>
        <map:reserved>ALL</map:reserved>
        <map:reserved>ALLOCATE</map:reserved>
        <map:reserved>ALTER</map:reserved>
        <map:reserved>AND</map:reserved>
        <map:reserved>ANY</map:reserved>
        <map:reserved>ARE</map:reserved>
        <map:reserved>ARRAY</map:reserved>
        <map:reserved>AS</map:reserved>
        <map:reserved>ASC</map:reserved>
        <map:reserved>ASENSITIVE</map:reserved>
        <map:reserved>ASSERTION</map:reserved>
        <map:reserved>ASYMMETRIC</map:reserved>
        <map:reserved>AT</map:reserved>
        <map:reserved>ATOMIC</map:reserved>
        <map:reserved>AUTHORIZATION</map:reserved>
        <map:reserved>BEFORE</map:reserved>
        <map:reserved>BEGIN</map:reserved>
        <map:reserved>BETWEEN</map:reserved>
        <map:reserved>BINARY</map:reserved>
        <map:reserved>BIT</map:reserved>
        <map:reserved>BLOB</map:reserved>
        <map:reserved>BOOLEAN</map:reserved>
        <map:reserved>BOTH</map:reserved>
        <map:reserved>BREADTH</map:reserved>
        <map:reserved>BY</map:reserved>
        <map:reserved>CALL</map:reserved>
        <map:reserved>CALLED</map:reserved>
        <map:reserved>CASCADE</map:reserved>
        <map:reserved>CASCADED</map:reserved>
        <map:reserved>CASE</map:reserved>
        <map:reserved>CAST</map:reserved>
        <map:reserved>CATALOG</map:reserved>
        <map:reserved>CHAR</map:reserved>
        <map:reserved>CHARACTER</map:reserved>
        <map:reserved>CHECK</map:reserved>
        <map:reserved>CLOB</map:reserved>
        <map:reserved>CLOSE</map:reserved>
        <map:reserved>COLLATE</map:reserved>
        <map:reserved>COLLATION</map:reserved>
        <map:reserved>COLUMN</map:reserved>
        <map:reserved>COMMIT</map:reserved>
        <map:reserved>CONDITION</map:reserved>
        <map:reserved>CONNECT</map:reserved>
        <map:reserved>CONNECTION</map:reserved>
        <map:reserved>CONSTRAINT</map:reserved>
        <map:reserved>CONSTRAINTS</map:reserved>
        <map:reserved>CONSTRUCTOR</map:reserved>
        <map:reserved>CONTINUE</map:reserved>
        <map:reserved>CORRESPONDING</map:reserved>
```

```
<map:reserved>CREATE</map:reserved>
<map:reserved>CROSS</map:reserved>
<map:reserved>CUBE</map:reserved>
<map:reserved>CURRENT</map:reserved>
<map:reserved>CURRENT_DATE</map:reserved>
<map:reserved>CURRENT_DEFAULT_TRANSFORM_GROUP</map:reserved>
<map:reserved>CURRENT_PATH</map:reserved>
<map:reserved>CURRENT_ROLE</map:reserved>
<map:reserved>CURRENT_TIME</map:reserved>
<map:reserved>CURRENT_TIMESTAMP</map:reserved>
<map:reserved>CURRENT_TRANSFORM_GROUP_FOR_TYPE</map:reserved>
<map:reserved>CURRENT_USER</map:reserved>
<map:reserved>CURSOR</map:reserved>
<map:reserved>CYCLE</map:reserved>
<map:reserved>DATA</map:reserved>
<map:reserved>DATE</map:reserved>
<map:reserved>DAY</map:reserved>
<map:reserved>DEALLOCATE</map:reserved>
<map:reserved>DEC</map:reserved>
<map:reserved>DECIMAL</map:reserved>
<map:reserved>DECLARE</map:reserved>
<map:reserved>DEFAULT</map:reserved>
<map:reserved>DEFERRABLE</map:reserved>
<map:reserved>DEFERRED</map:reserved>
<map:reserved>DELETE</map:reserved>
<map:reserved>DEPTH</map:reserved>
<map:reserved>DEREF</map:reserved>
<map:reserved>DESC</map:reserved>
<map:reserved>DESCRIBE</map:reserved>
<map:reserved>DESCRIPTOR</map:reserved>
<map:reserved>DETERMINISTIC</map:reserved>
<map:reserved>DIAGNOSTICS</map:reserved>
<map:reserved>DISCONNECT</map:reserved>
<map:reserved>DISTINCT</map:reserved>
<map:reserved>DO</map:reserved>
<map:reserved>DOMAIN</map:reserved>
<map:reserved>DOUBLE</map:reserved>
<map:reserved>DROP</map:reserved>
<map:reserved>DYNAMIC</map:reserved>
<map:reserved>EACH</map:reserved>
<map:reserved>ELSE</map:reserved>
<map:reserved>ELSEIF</map:reserved>
<map:reserved>END</map:reserved>
<map:reserved>EQUALS</map:reserved>
<map:reserved>ESCAPE</map:reserved>
<map:reserved>EXCEPT</map:reserved>
<map:reserved>EXCEPTION</map:reserved>
<map:reserved>EXEC</map:reserved>
<map:reserved>EXECUTE</map:reserved>
<map:reserved>EXISTS</map:reserved>
<map:reserved>EXIT</map:reserved>
<map:reserved>EXTERNAL</map:reserved>
<map:reserved>FALSE</map:reserved>
<map:reserved>FETCH</map:reserved>
<map:reserved>FILTER</map:reserved>
<map:reserved>FIRST</map:reserved>
<map:reserved>FLOAT</map:reserved>
<map:reserved>FOR</map:reserved>
<map:reserved>FOREIGN</map:reserved>
<map:reserved>FOUND</map:reserved>
<map:reserved>FREE</map:reserved>
<map:reserved>FROM</map:reserved>
<map:reserved>FULL</map:reserved>
<map:reserved>FUNCTION</map:reserved>
<map:reserved>GENERAL</map:reserved>
<map:reserved>GET</map:reserved>
<map:reserved>GLOBAL</map:reserved>
```

```
        <map:reserved>GO</map:reserved>
        <map:reserved>GOTO</map:reserved>
        <map:reserved>GRANT</map:reserved>
        <map:reserved>GROUP</map:reserved>
        <map:reserved>GROUPING</map:reserved>
        <map:reserved>HANDLER</map:reserved>
        <map:reserved>HAVING</map:reserved>
        <map:reserved>HOLD</map:reserved>
        <map:reserved>HOUR</map:reserved>
        <map:reserved>IDENTITY</map:reserved>
        <map:reserved>IF</map:reserved>
        <map:reserved>IMMEDIATE</map:reserved>
        <map:reserved>IN</map:reserved>
        <map:reserved>INDICATOR</map:reserved>
        <map:reserved>INITIALLY</map:reserved>
        <map:reserved>INNER</map:reserved>
        <map:reserved>INOUT</map:reserved>
        <map:reserved>INPUT</map:reserved>
        <map:reserved>INSENSITIVE</map:reserved>
        <map:reserved>INSERT</map:reserved>
        <map:reserved>INT</map:reserved>
        <map:reserved>INTEGER</map:reserved>
        <map:reserved>INTERSECT</map:reserved>
        <map:reserved>INTERVAL</map:reserved>
        <map:reserved>INTO</map:reserved>
        <map:reserved>IS</map:reserved>
        <map:reserved>ISOLATION</map:reserved>
        <map:reserved>ITERATE</map:reserved>
        <map:reserved>JOIN</map:reserved>
        <map:reserved>KEY</map:reserved>
        <map:reserved>LANGUAGE</map:reserved>
        <map:reserved>LARGE</map:reserved>
        <map:reserved>LAST</map:reserved>
        <map:reserved>LATERAL</map:reserved>
        <map:reserved>LEADING</map:reserved>
        <map:reserved>LEAVE</map:reserved>
        <map:reserved>LEFT</map:reserved>
        <map:reserved>LEVEL</map:reserved>
        <map:reserved>LIKE</map:reserved>
        <map:reserved>LOCAL</map:reserved>
        <map:reserved>LOCALTIME</map:reserved>
        <map:reserved>LOCALTIMESTAMP</map:reserved>
        <map:reserved>LOCATOR</map:reserved>
        <map:reserved>LOOP</map:reserved>
        <map:reserved>MAP</map:reserved>
        <map:reserved>MATCH</map:reserved>
        <map:reserved>METHOD</map:reserved>
        <map:reserved>MINUTE</map:reserved>
        <map:reserved>MODIFIES</map:reserved>
        <map:reserved>MODULE</map:reserved>
        <map:reserved>MONTH</map:reserved>
        <map:reserved>NAMES</map:reserved>
        <map:reserved>NATIONAL</map:reserved>
        <map:reserved>NATURAL</map:reserved>
        <map:reserved>NCHAR</map:reserved>
        <map:reserved>NCLOB</map:reserved>
        <map:reserved>NEW</map:reserved>
        <map:reserved>NEXT</map:reserved>
        <map:reserved>NO</map:reserved>
        <map:reserved>NONE</map:reserved>
        <map:reserved>NOT</map:reserved>
        <map:reserved>NULL</map:reserved>
        <map:reserved>NUMERIC</map:reserved>
        <map:reserved>OBJECT</map:reserved>
        <map:reserved>OF</map:reserved>
        <map:reserved>OLD</map:reserved>
        <map:reserved>ON</map:reserved>
```

```
<map:reserved>ONLY</map:reserved>
<map:reserved>OPEN</map:reserved>
<map:reserved>OPTION</map:reserved>
<map:reserved>OR</map:reserved>
<map:reserved>ORDER</map:reserved>
<map:reserved>ORDINALITY</map:reserved>
<map:reserved>OUT</map:reserved>
<map:reserved>OUTER</map:reserved>
<map:reserved>OUTPUT</map:reserved>
<map:reserved>OVER</map:reserved>
<map:reserved>OVERLAPS</map:reserved>
<map:reserved>PAD</map:reserved>
<map:reserved>PARAMETER</map:reserved>
<map:reserved>PARTIAL</map:reserved>
<map:reserved>PARTITION</map:reserved>
<map:reserved>PATH</map:reserved>
<map:reserved>PRECISION</map:reserved>
<map:reserved>PREPARE</map:reserved>
<map:reserved>PRESERVE</map:reserved>
<map:reserved>PRIMARY</map:reserved>
<map:reserved>PRIOR</map:reserved>
<map:reserved>PRIVILEGES</map:reserved>
<map:reserved>PROCEDURE</map:reserved>
<map:reserved>PUBLIC</map:reserved>
<map:reserved>RANGE</map:reserved>
<map:reserved>READ</map:reserved>
<map:reserved>READS</map:reserved>
<map:reserved>REAL</map:reserved>
<map:reserved>RECURSIVE</map:reserved>
<map:reserved>REF</map:reserved>
<map:reserved>REFERENCES</map:reserved>
<map:reserved>REFERENCING</map:reserved>
<map:reserved>RELATIVE</map:reserved>
<map:reserved>RELEASE</map:reserved>
<map:reserved>REPEAT</map:reserved>
<map:reserved>RESIGNAL</map:reserved>
<map:reserved>RESTRICT</map:reserved>
<map:reserved>RESULT</map:reserved>
<map:reserved>RETURN</map:reserved>
<map:reserved>RETURNS</map:reserved>
<map:reserved>REVOKE</map:reserved>
<map:reserved>RIGHT</map:reserved>
<map:reserved>ROLE</map:reserved>
<map:reserved>ROLLBACK</map:reserved>
<map:reserved>ROLLUP</map:reserved>
<map:reserved>ROUTINE</map:reserved>
<map:reserved>ROW</map:reserved>
<map:reserved>ROWS</map:reserved>
<map:reserved>SAVEPOINT</map:reserved>
<map:reserved>SCHEMA</map:reserved>
<map:reserved>SCOPE</map:reserved>
<map:reserved>SCROLL</map:reserved>
<map:reserved>SEARCH</map:reserved>
<map:reserved>SECOND</map:reserved>
<map:reserved>SECTION</map:reserved>
<map:reserved>SELECT</map:reserved>
<map:reserved>SENSITIVE</map:reserved>
<map:reserved>SESSION</map:reserved>
<map:reserved>SESSION_USER</map:reserved>
<map:reserved>SET</map:reserved>
<map:reserved>SETS</map:reserved>
<map:reserved>SIGNAL</map:reserved>
<map:reserved>SIMILAR</map:reserved>
<map:reserved>SIZE</map:reserved>
<map:reserved>SMALLINT</map:reserved>
<map:reserved>SOME</map:reserved>
<map:reserved>SPACE</map:reserved>
```

```
        <map:reserved>SPECIFIC</map:reserved>
        <map:reserved>SPECIFICTYPE</map:reserved>
        <map:reserved>SQL</map:reserved>
        <map:reserved>SQLEXCEPTION</map:reserved>
        <map:reserved>SQLSTATE</map:reserved>
        <map:reserved>SQLWARNING</map:reserved>
        <map:reserved>START</map:reserved>
        <map:reserved>STATE</map:reserved>
        <map:reserved>STATIC</map:reserved>
        <map:reserved>SYMMETRIC</map:reserved>
        <map:reserved>SYSTEM</map:reserved>
        <map:reserved>SYSTEM_USER</map:reserved>
        <map:reserved>TABLE</map:reserved>
        <map:reserved>TEMPORARY</map:reserved>
        <map:reserved>THEN</map:reserved>
        <map:reserved>TIME</map:reserved>
        <map:reserved>TIMESTAMP</map:reserved>
        <map:reserved>TIMEZONE_HOUR</map:reserved>
        <map:reserved>TIMEZONE_MINUTE</map:reserved>
        <map:reserved>TO</map:reserved>
        <map:reserved>TRAILING</map:reserved>
        <map:reserved>TRANSACTION</map:reserved>
        <map:reserved>TRANSLATION</map:reserved>
        <map:reserved>TREAT</map:reserved>
        <map:reserved>TRIGGER</map:reserved>
        <map:reserved>TRUE</map:reserved>
        <map:reserved>UNDER</map:reserved>
        <map:reserved>UNDO</map:reserved>
        <map:reserved>UNION</map:reserved>
        <map:reserved>UNIQUE</map:reserved>
        <map:reserved>UNKNOWN</map:reserved>
        <map:reserved>UNNEST</map:reserved>
        <map:reserved>UNTIL</map:reserved>
        <map:reserved>UPDATE</map:reserved>
        <map:reserved>USAGE</map:reserved>
        <map:reserved>USER</map:reserved>
        <map:reserved>USING</map:reserved>
        <map:reserved>VALUE</map:reserved>
        <map:reserved>VALUES</map:reserved>
        <map:reserved>VARCHAR</map:reserved>
        <map:reserved>VARYING</map:reserved>
        <map:reserved>VIEW</map:reserved>
        <map:reserved>WHEN</map:reserved>
        <map:reserved>WHENEVER</map:reserved>
        <map:reserved>WHERE</map:reserved>
        <map:reserved>WHILE</map:reserved>
        <map:reserved>WINDOW</map:reserved>
        <map:reserved>WITH</map:reserved>
        <map:reserved>WITHIN</map:reserved>
        <map:reserved>WITHOUT</map:reserved>
        <map:reserved>WORK</map:reserved>
        <map:reserved>WRITE</map:reserved>
        <map:reserved>YEAR</map:reserved>
        <map:reserved>ZONE</map:reserved>
</map:list>
```

Figure B.1 The XML Document Defining the Reserved SQL99 Keywords

```
<?xml version="1.0" encoding="UTF-8"?>
<map:mappers
        xmlns="http://troubadix.wb.tu-harburg.de/map"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:map="http://troubadix.wb.tu-harburg.de/map"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:gml="http://www.opengis.net/gml"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://troubadix.wb.tu-harburg.de/map
                http://troubadix.wb.tu-harburg.de/map/mapping.xsd">
        <map:mapping>schema2java</map:mapping>
        <!--XML Schema data types-->
        <map:mapper>
                <map:from>xs:boolean</map:from>
                <map:to>boolean</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:byte</map:from>
                <map:to>byte</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:short</map:from>
                <map:to>short</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:int</map:from>
                <map:to>int</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:long</map:from>
                <map:to>long</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:float</map:from>
                <map:to>float</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:double</map:from>
                <map:to>double</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:integer</map:from>
                <map:to>java.math.BigInteger</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:decimal</map:from>
                <map:to>java.math.BigDecimal</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:string</map:from>
                <map:to>java.lang.String</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:dateTime</map:from>
                <map:to>java.util.Calendar</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:base64Binary</map:from>
                <map:to>byte[]</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>xs:hexBinary</map:from>
                <map:to>byte[]</map:to>
        </map:mapper>
        <map:mapper>
        <map:from>duration</map:from>
        <map:to>java.lang.String</map:to>
        <!-- weblogic.xml.schema.binding.util.Duration -->
</map:mapper>
<map:mapper>
        <map:from>xs:time</map:from>
        <map:to>java.util.Calendar</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:date</map:from>
        <map:to>java.util.Calendar</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:gYearMonth</map:from>
        <map:to>java.util.Calendar</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:gYear</map:from>
        <map:to>java.util.Calendar</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:gMonthDay</map:from>
        <map:to>java.util.Calendar</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:gDay</map:from>
        <map:to>java.util.Calendar</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:gMonth</map:from>
        <map:to>java.util.Calendar</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:anyURI</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:NOTATION</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:token</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:normalizedString</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:language</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:Name</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:NMTOKEN</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:NCName</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:NMTOKENS</map:from>
        <map:to>java.lang.String[]</map:to>
```

```
</map:mapper>
<map:mapper>
        <map:from>xs:ID</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:IDREF</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:ENTITY</map:from>
        <map:to>java.lang.String</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:IDREFS</map:from>
        <map:to>java.lang.String[]</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:ENTITIES</map:from>
        <map:to>java.lang.String[]</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:nonPositiveInteger</map:from>
        <map:to>java.math.BigInteger</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:nonNegativeInteger</map:from>
        <map:to>java.math.BigInteger</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:negativeInteger</map:from>
        <map:to>java.math.BigInteger</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:unsignedLong</map:from>
        <map:to>java.math.BigInteger</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:positiveInteger</map:from>
        <map:to>java.math.BigInteger</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:unsignedInt</map:from>
        <map:to>long</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:unsignedShort</map:from>
        <map:to>int</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:unsignedByte</map:from>
        <map:to>short</map:to>
</map:mapper>
<map:mapper>
        <map:from>xs:QName</map:from>
        <map:to>java.lang.String</map:to>
        <!-- javax.xml.namespace.QName -->
</map:mapper>
<!--GML data types-->
<map:mapper>
        <map:from>gml:Point</map:from>
        <map:to>Point</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:LineString</map:from>
        <map:to>LineString</map:to>
</map:mapper>
```

```
<map:mapper>
        <map:from>gml:LinearRing</map:from>
        <map:to>LinearRing</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:Polygon</map:from>
        <map:to>Polygon</map:to>
</map:mapper>
<!--<map:mapper>
        <map:from>gml:Box</map:from>
        <map:to>Box</map:to>
</map:mapper>-->
<map:mapper>
        <map:from>gml:MultiGeometry</map:from>
        <map:to>MultiGeometry</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:MultiPoint</map:from>
        <map:to>MultiPoint</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:MultiLineString</map:from>
        <map:to>MultiLineString</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:MultiPolygon</map:from>
        <map:to>MultiPolygon</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:PointType</map:from>
        <map:to>Point</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:LineStringType</map:from>
        <map:to>LineString</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:LinearRingType</map:from>
        <map:to>LinearRing</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:PolygonType</map:from>
        <map:to>Polygon</map:to>
</map:mapper>
<!--<map:mapper>
        <map:from>gml:BoxType</map:from>
        <map:to>Box</map:to>
</map:mapper>-->
<map:mapper>
        <map:from>gml:MultiGeometryType</map:from>
        <map:to>MultiGeometry</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:MultiPointType</map:from>
        <map:to>MultiPoint</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:MultiLineStringType</map:from>
        <map:to>MultiLineString</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:MultiPolygonType</map:from>
        <map:to>MultiPolygon</map:to>
</map:mapper>
<map:mapper>
        <map:from>gml:pointProperty</map:from>
        <map:to>Point</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:lineStringProperty</map:from>
                <map:to>LineString</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:polygonProperty</map:from>
                <map:to>Polygon</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:multiGeometryProperty</map:from>
                <map:to>MultiGeometry</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:multiPointProperty</map:from>
                <map:to>MultiPoint</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:multiLineStringProperty</map:from>
                <map:to>MultiLineString</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:multiPolygonProperty</map:from>
                <map:to>MultiPolygon</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:PointPropertyType</map:from>
                <map:to>Point</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:LineStringPropertyType</map:from>
                <map:to>LineString</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:PolygonPropertyType</map:from>
                <map:to>Polygon</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:MultiGeometryPropertyType</map:from>
                <map:to>MultiGeometry</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:MultiPointPropertyType</map:from>
                <map:to>MultiPoint</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:MultiLineStringPropertyType</map:from>
                <map:to>MultiLineString</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>gml:MultiPolygonPropertyType</map:from>
                <map:to>MultiPolygon</map:to>
        </map:mapper>
</map:mappers>
```

Figure B.2 The XML Document Defining the Data Type Mapping from XML Schema Data Type to Java Data Types

```
<?xml version="1.0" encoding="UTF-8"?>
<map:mappers
        xmlns="http://troubadix.wb.tu-harburg.de/map"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:map="http://troubadix.wb.tu-harburg.de/map"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://troubadix.wb.tu-harburg.de/map
                http://troubadix.wb.tu-harburg.de/map/mapping.xsd">
        <map:mapping>java2sql</map:mapping>
        <map:sql>postgis</map:sql>
        <!--JAVA data types-->
        <map:mapper>
                <map:from>int</map:from>
                <map:to>INTEGER</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>short</map:from>
                <map:to>SMALLINT</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>long</map:from>
                <map:to>BIGINT</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>float</map:from>
                <map:to>FLOAT4</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>double</map:from>
                <map:to>FLOAT8</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>byte</map:from>
                <map:to>SMALLINT</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>boolean</map:from>
                <map:to>BOOLEAN</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>char</map:from>
                <map:to>CHAR</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>java.lang.Integer</map:from>
                <map:to>INTEGER</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>java.lang.Short</map:from>
                <map:to>SMALLINT</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>java.lang.Long</map:from>
                <map:to>BIGINT</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>java.lang.Float</map:from>
                <map:to>FLOAT4</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>java.lang.Double</map:from>
                <map:to>FLOAT8</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>java.lang.Byte</map:from>
        <map:to>SMALLINT</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.lang.Boolean</map:from>
        <map:to>BOOLEAN</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.lang.Character</map:from>
        <map:to>CHAR</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.lang.String</map:from>
        <map:to>VARCHAR</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.lang.String[]</map:from>
        <map:to>TEXT</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.math.BigInteger</map:from>
        <map:to>DECIMAL</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.math.BigDecimal</map:from>
        <map:to>DECIMAL</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.util.Calendar</map:from>
        <map:to>VARCHAR</map:to>
</map:mapper>
<map:mapper>
        <map:from>java.util.Date</map:from>
        <map:to>DATE</map:to>
</map:mapper>
<map:mapper>
        <map:from>byte[]</map:from>
        <map:to>TEXT</map:to>
</map:mapper>
<!--<map:mapper>
        <map:from>weblogic.xml.schema.binding.util.Duration</map:from>
        <map:to></map:to>
</map:mapper>
<map:mapper>
        <map:from>javax.xml.namespace.QName</map:from>
        <map:to></map:to>
</map:mapper>-->
<!--GML data types-->
<map:mapper>
        <map:from>Point</map:from>
        <map:to>POINT</map:to>
</map:mapper>
<map:mapper>
        <map:from>Polygon</map:from>
        <map:to>POLYGON</map:to>
</map:mapper>
<map:mapper>
        <map:from>LineString</map:from>
        <map:to>LINESTRING</map:to>
</map:mapper>
<map:mapper>
        <map:from>LinearRing</map:from>
        <map:to>LINESTRING</map:to>
</map:mapper>
<map:mapper>
        <map:from>MultiPoint</map:from>
        <map:to>MULTIPOINT</map:to>
</map:mapper>
```

```
        <map:mapper>
                <map:from>MultiPolygon</map:from>
                <map:to>MULTIPOLYGON</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>MultiLineString</map:from>
                <map:to>MULTILINESTRING</map:to>
        </map:mapper>
        <map:mapper>
                <map:from>MultiGeometry</map:from>
                <map:to>GEOMETRYCOLLECTION</map:to>
        </map:mapper>
</map:mappers>
```

Figure B.3 The XML Document Defining the Data Type Mapping from Java Data Types to Postgis SQL Data Types

Appendix C. Implementation Related XML Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
       targetNamespace="http://troubadix.wb.tu-harburg.de/map"
       xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:xlink="http://www.w3.org/1999/xlink"
       xmlns:map="http://troubadix.wb.tu-harburg.de/map"
       version="2.1.2">
       <element name="home" type="string"/>
       <element name="map_path" type="string"/>
       <element name="schema2java" type="string"/>
       <element name="java2sql" type="string"/>
       <element name="dbname" type="NMTOKEN"/>
       <element name="dbs" type="map:DbsType"/>
       <element name="config" type="map:ConfigType"/>
       <complexType name="DbsType">
              <sequence minOccurs="1" maxOccurs="1">
                     <element ref="map:dbname" minOccurs="1" maxOccurs="unbounded"/>
              </sequence>
       </complexType>
       <complexType name="ConfigType">
              <sequence minOccurs="1" maxOccurs="1">
                     <element ref="map:home" minOccurs="1" maxOccurs="1"/>
                     <element ref="map:map_path" minOccurs="1" maxOccurs="1"/>
                     <element ref="map:schema2java" minOccurs="1" maxOccurs="1"/>
                     <element ref="map:java2sql" minOccurs="1" maxOccurs="1"/>
                     <element ref="map:dbs" minOccurs="1" maxOccurs="1"/>
              </sequence>
       </complexType>
</schema>
```

Figure C.1 The XML Schema for the Configuration File for the *Generator* (RedGenie)

```
<?xml version="1.0" encoding="UTF-8"?>
<map:config
       xmlns="http://troubadix.wb.tu-harburg.de/map"
       xmlns:xlink="http://www.w3.org/1999/xlink"
       xmlns:map="http://troubadix.wb.tu-harburg.de/map"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://troubadix.wb.tu-harburg.de/map
              http://troubadix.wb.tu-harburg.de/map/config.xsd">
       <map:home>x:\\Personal\\Thesis\\Release\\</map:home>
       <map:map_path>xml\\</map:map_path>
       <map:schema2java>schema2java.xml</map:schema2java>
       <map:java2sql>reserved.xml</map:java2sql>
       <map:dbs>
              <map:dbname>postgis</map:dbname>
       </map:dbs>
</map:config>
```

Figure C.2 The Configuration File for the *Generator* (RedGenie)

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
	targetNamespace="http://troubadix.wb.tu-harburg.de/map"
	xmlns="http://www.w3.org/2001/XMLSchema"
	xmlns:xlink="http://www.w3.org/1999/xlink"
	xmlns:map="http://troubadix.wb.tu-harburg.de/map"
	version="2.1.2">
	<element name="from" type="string"/>
	<element name="to" type="string"/>
	<element name="schema" type="map:DataType"/>
	<element name="java" type="map:DataType"/>
	<element name="sql" type="map:DataType"/>
	<element name="geomType" type="map:DataTypeType"/>
	<element name="geometry" type="map:GeometryType"/>
	<element name="dataType" type="map:DataTypeType"/>
	<element name="attribute" type="map:AttributeType"/>
	<element name="table" type="map:TableType"/>
	<element name="report" type="map:ReportType"/>
	<complexType name="DataType">
		<attribute name="name" type="string" use="required"/>
		<attribute name="size" type="string" use="optional"/>
	</complexType>
	<complexType name="DataTypeType">
		<sequence minOccurs="1" maxOccurs="1">
			<element ref="map:schema" minOccurs="1" maxOccurs="1"/>
			<element ref="map:java" minOccurs="1" maxOccurs="1"/>
			<element ref="map:sql" minOccurs="1" maxOccurs="1"/>
		</sequence>
	</complexType>
	<complexType name="AttributeType">
		<sequence minOccurs="1" maxOccurs="1">
			<element ref="map:from" minOccurs="1" maxOccurs="1"/>
			<element ref="map:to" minOccurs="1" maxOccurs="1"/>
			<element ref="map:dataType" minOccurs="1" maxOccurs="1"/>
		</sequence>
		<attribute name="null" type="boolean" use="optional"/>
	</complexType>
	<complexType name="GeometryType">
		<sequence minOccurs="1" maxOccurs="1">
			<element ref="map:from" minOccurs="1" maxOccurs="1"/>
			<element ref="map:to" minOccurs="1" maxOccurs="1"/>
			<element ref="map:geomType" minOccurs="1" maxOccurs="1"/>
		</sequence>
		<attribute name="dbname" type="string" use="required"/>
		<attribute name="srid" type="integer" use="required"/>
		<attribute name="dimension" type="integer" use="required"/>
	</complexType>
	<complexType name="TableType">
		<sequence minOccurs="1" maxOccurs="1">
			<element ref="map:from" minOccurs="1" maxOccurs="1"/>
			<element ref="map:to" minOccurs="1" maxOccurs="1"/>
			<element ref="map:attribute" minOccurs="1" maxOccurs="unbounded"/>
			<element ref="map:geometry" minOccurs="0" maxOccurs="unbounded"/>
		</sequence>
	</complexType>
	<complexType name="ReportType">
		<sequence minOccurs="1" maxOccurs="1">
			<element ref="map:table" minOccurs="1" maxOccurs="unbounded"/>
		</sequence>
	</complexType>
</schema>
```

Figure C.3 The XML Schema for the Mapping Process Report Generation

```
<?xml version="1.0" encoding="UTF-8"?>
<report
    xmlns=http://troubadix.wb.tu-harburg.de/map
    xmlns:xlink=http://www.w3.org/1999/xlink
    xmlns:map=http://troubadix.wb.tu-harburg.de/map
    xmlns:xs=http://www.w3.org/2001/XMLSchema
    xmlns:gml=http://www.opengis.net/gml
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:buenz=http://wb.tuhh.de/buenz
    xsi:schemaLocation="http://troubadix.wb.tu-harburg.de/map
        http://troubadix.wb.tu-harburg.de/map/report.xsd">
    <table>
        <from>buenz:Buenzau_nutzung</from>
        <to>Buenzau_nutzung</to>
        <attribute null="true">
            <from>buenz:AREA</from>
            <to>AREA</to>
            <dataType>
                <schema name="xs:double" />
                <java name="java.lang.Double" />
                <sql name="FLOAT8" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:PERIMETER</from>
            <to>PERIMETER</to>
            <dataType>
                <schema name="xs:double" />
                <java name="java.lang.Double" />
                <sql name="FLOAT8" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:NUTZUNG_</from>
            <to>NUTZUNG_</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:NUTZUNG_ID</from>
            <to>NUTZUNG_ID</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:NS</from>
            <to>NS</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:BRD</from>
            <to>BRD</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:SZENE</from>
            <to>SZENE</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:SPHEROID</from>
            <to>SPHEROID</to>
            <dataType>
                <schema name="xs:string" />
                <java name="java.lang.String" />
                <sql name="VARCHAR" size="128" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:TKNR</from>
            <to>TKNR</to>
            <dataType>
                <schema name="xs:string" />
                <java name="java.lang.String" />
                <sql name="VARCHAR" size="128" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:NS1</from>
            <to>NS1</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:NS2</from>
            <to>NS2</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:NS3</from>
            <to>NS3</to>
            <dataType>
                <schema name="xs:int" />
                <java name="java.lang.Integer" />
                <sql name="INTEGER" />
            </dataType>
        </attribute>
        <attribute null="true">
            <from>buenz:NUTZUNG</from>
            <to>NUTZUNG</to>
            <dataType>
                <schema name="xs:string" />
                <java name="java.lang.String" />
                <sql name="VARCHAR" size="128" />
            </dataType>
        </attribute>
        <geometry dbname="test" srid="31467" dimension="2">
```

```
            <from>gml:the_geom</from>
            <to>the_geom</to>
            <geomType>
                <schema name="gml:MultiPolygonPropertyType" />
                <java name="MultiPolygon" />
                <sql name="MULTIPOLYGON" />
            </geomType>
        </geometry>
    </table>
</report>
```

Figure C.4 The Report File for the Buenzau_nutzung Relational Database Schema Definition Generation

References

[XML04] Extensible Markup Language (XML). http://www.w3.org/XML/. Last visited: March 05, 2004.

[XSD04] XML Schema. http://www.w3.org/XML/Schema. Last visited: March 12, 2004.

[JDBC04] J2EE JDBC Technology. http://java.sun.com/products/jdbc/. Last visited: March 15, 2004.

[DOM04] Document Object Model (DOM). http://www.w3.org/DOM/. Last visited: January 20, 2004.

[XSPY04] XMLSpy 2004. Relational Database Integration. http://www.altova.com/features_database.html. Last visited: February 23, 2004.

[ORCL04] XML Technology Center. Oracle Technology Center. http://otn.oracle.com/tech/xml/index.html Last visited: February 23, 2004.

[NET04] Microsoft .NET. http://www.microsoft.com/net/. Last visited: February 23, 2004.

[XSDB04] Project: Manage DB Schema using XSD: Summary. SourceForge.net. http://sourceforge.net/projects/xsd2db/ Last visited: February 23, 2004.

[JAVA04] Java Technology. Products & Technologies. Sun Microsystems. http://java.sun.com/. Last visited: March 15, 2004.

[OGC04] Open GIS Consortium (OGC). http://www.opengis.org/. Last visited: March 18, 2004.

[SFSv1.1] Open GIS Simple Features Specification for SQL Revision 1.1. OpenGIS Specification 05/99. Open GIS Consortium, Inc. 1999. http://www.opengis.org/docs/99-049.pdf

[MaHu02] Matthes, Prof. Dr. Florian. Hupe, Parick. Software Architectures. Layered Architectures and Persistence Management. Lecture Slides. STS, TUHH. 2002.

[SiSt02] Singh, Inderjeet. Stearns, Beth. Johnson, Mark. Designing Enterprise Applications with the J2EE Platform, Second Edition. Addison-Wesley 03/02. Sun Microsystems, Inc. 2002.

[XSDp1] XML Schema Part 1: Structures. W3C Recommendation 05/01. W3C 2001. http://www.w3.org/TR/xmlschema-1/

[GMLv2.1] OpenGIS Geography Markup Language (GML) Implementation Specification Version 2.1.2. OpenGIS Implementation Specification

09/02. OpenGIS Consortium, Inc. 2002. http://www.opengis.org/docs/02-069.pdf

[XPATHv1] XML Path Language (XPath) Version 1.0. W3C Recommendation 11/99. W3C 1999. http://www.w3.org/TR/xpath

[XLINKv1] XML Linking Language (XLink) Version 1.0. W3C Recommendation 06/01. W3C 2001. http://www.w3.org/TR/xlink/

[DGREE04] Deegree – OpenGIS Specifications implementation initiative. GIS and Remote Sensing Unit of the Department of Geography, University of Bonn, and lat/lon. http://deegree.sourceforge.net/
Last visited: February 23, 2004

[SQL01] Melton, Jim. Simon, Alan. SQL:1999 – Understanding Relational Language Components. Morgan Kaufmann 2001

[JDOM04] JDOM Project. http://www.jdom.org/. Last visited: February 30, 2004

[SAX04] Simple API for XML (SAX). http://www.saxproject.org/.
Last visited: January 20, 2004

[GEOS04] The GeoServer Project. The Open Internet Gateway for Geographic Data. http://geoserver.sourceforge.net/html/index.php.
Last visited: February 23, 2004

[WFSv1.0] Web Feature Service Implementation Specification Version 1.0.0. OpenGIS Implementation Specification 09/02. Open GIS Consortium, Inc. 2002. http://www.opengis.org/docs/02-058.pdf

[SHP98] ESRI Shapefile Technical Description. ESRI White Paper 07/98. Environmental Systems Research Institute (ESRI), Inc. 1998.
http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[WFD03] Introduction to the New EU Water Framework Directive. European Commission 2003. http://europa.eu.int/comm/environment/water/water-framework/overview.html

[GIS04] Geographical Information Systems (GIS). http://www.gis.com/.
Last visited: February 30, 2004

[XSINT01] W3C XML Schema Published as a W3C Recommendation. Cover Pages 05/01. OASIS 2001.
http://xml.coverpages.org/ni2001-05-03-c.html

[XSDp2] XML Schema Part 2: Datatypes. W3C Recommendation 05/01. W3C 2001. http://www.w3.org/TR/xmlschema-2/

[XSDp0] XML Schema Part 0: Primer. W3C Recommendation 05/01. W3C 2001. http://www.w3.org/TR/xmlschema-0/

[XSLTv1.0] XSL Transformations Version 1.0. W3C Recommendation 11/99. W3C 1999. http://www.w3.org/TR/xslt