



FAKULTÄT FÜR **INFORMATIK**

# Object–XML mapping with JAXB2

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Informatik**

eingereicht von

**Joachim Grüneis**

Matrikelnummer 9126330

an der  
Fakultät für Informatik der Technischen Universität Wien

Betreuung:

Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Alexander Schatten

Wien, 16.10.2008

\_\_\_\_\_  
(Unterschrift Verfasser)

\_\_\_\_\_  
(Unterschrift Betreuer)



# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Kurzdarstellung</b>	<b>3</b>
<b>3</b>	<b>Motivation</b>	<b>5</b>
3.1	Research question . . . . .	5
3.2	Research activity . . . . .	6
<b>4</b>	<b>Extensible Markup Language (XML)</b>	<b>7</b>
4.1	Background of XML . . . . .	7
4.1.1	Markup Language . . . . .	7
4.1.2	Semi-Structured Data . . . . .	8
4.1.3	XML . . . . .	10
4.2	XML for data modeling . . . . .	12
4.2.1	Document Type Definition (DTD) . . . . .	12
4.2.2	W3C XML Schema . . . . .	17
4.3	Usage of XML . . . . .	30
4.3.1	Narrative Documents . . . . .	30
4.3.2	Record Like Data . . . . .	30
4.4	XML Standards . . . . .	31
4.5	Summary . . . . .	31
4.5.1	DTD Summary . . . . .	32
4.5.2	W3C XML Schema Summary . . . . .	32
<b>5</b>	<b>Java for Data Modeling</b>	<b>35</b>
5.1	Object Oriented Approach . . . . .	35
5.2	OO for Data Modeling . . . . .	37
5.3	Java Basics . . . . .	38
5.4	Java for Data Modeling . . . . .	39
5.5	Summary . . . . .	41
<b>6</b>	<b>Mapping</b>	<b>43</b>
6.1	Motivation of Mapping . . . . .	43
6.2	Mapping without business domain knowledge . . . . .	44

## Contents

6.2.1	Standards	44
6.2.2	Implementations	45
6.3	Mapping with business domain knowledge	45
6.3.1	Standards	46
6.3.2	Implementations	47
6.4	Summary	50
<b>7</b>	<b>Castor JAXB2 implementation</b>	<b>51</b>
7.1	JAXB2 Overview	51
7.2	Analysis of JAXB2 and Castor	56
7.2.1	Coverage of W3C XML Schema	56
7.2.2	General differences	57
7.2.3	Mapping specification	57
7.2.4	Binding API	64
7.2.5	Schema compiler	68
7.2.6	Schema generator	68
7.3	Castor JAXB2 Implementation	68
7.3.1	Major <i>JAXB2</i> execution flows	69
7.3.2	The issues to achieve the <i>Castor JAXB2 Implementation</i> are	69
7.4	Critical Discussion	74
<b>8</b>	<b>Conclusion</b>	<b>79</b>
8.1	XML	79
8.2	Java	79
8.3	Mapping	80
8.4	Castor	80
8.5	JAXB2 Standard	81
8.6	Comparison Castor and JAXB2 Standard	81
8.7	Implementation of Castor JAXB2	82
8.8	Implementation Results	82
8.9	Future Work	83
<b>A</b>	<b>Appendix</b>	<b>85</b>
A.1	Document Object Model (DOM) Specifications	85
A.2	Java Specification Requests related to XML	85
A.3	Tag cloud	88

# 1 Abstract

The focus of today's business software is management of information. In a mainstream IT system the information is represented either object oriented, relational or using XML depending on the durability of the information. During processing the information needs to be mapped from one model to the other. The transformation between the models brings up problems like completeness and validation of information as well as impedance mismatch.

The subarea of Object-XML mapping and within it the mapping of Java-XML based on *JAXB2 Standard* and *Castor Framework* is the contents of the thesis.

The thesis introduces into XML in general and provides a deeper look into data modeling for XML (building *XML Applications*) using *Document Type Definition (DTD)* and *W3C XML Schema Definition Language*. The latter providing the possibility to build strong typed data models by allowing the user to create own types and enforces that elements base on types.

For Java the building blocks like `class`, `interface`, `field`, `method` and *visibility* and the concepts of object orientation like *encapsulation*, *abstraction*, *inheritance* and *polymorphism* are introduced.

Mapping brings together Java and XML and explains which styles of mapping, with or without business domain knowledge, are possible. Mapping standards like *DOM*, *JAXP* and *JAXB2* are listed, *O/X Mapping* is explained and implementations like *XStream*, *JAXME*, *XMLBeans*, *Castor* and *JAXB2 Reference Implementation* are discussed.

The practical part of the thesis starts with a thorough discussion of the *JAXB2 Standard* and an analysis of *Castor's* capabilities to fulfill the standard requirements. The (partial) implementation of the *Castor JAXB2 Layer* and the discussion of the implementation form the last and major part of the thesis.

*1 Abstract*

## 2 Kurzdarstellung

Der Fokus von kommerzieller Datenverarbeitung liegt klar in der Verarbeitung und Verwaltung von Daten. Diese Daten liegen zumeist in unterschiedlichen Zuständen als Laufzeitobjekte, Datensätze einer relationalen Datenbank oder XML vor. Im Laufe der Verarbeitung der Daten werden diese zwischen den Systemen und Zuständen transformiert. Dabei ergeben sich Problemstellungen wie: Ist die Transformation vollständig möglich? Sind die Modelle der verschiedenen Systeme gegenseitig abbildbar? Wie können die Daten validiert werden?

Mit dem Teilbereich der Abbildung zwischen Objekten und XML und noch spezieller mit der Abbildung von Java und XML basierend auf den JAXB2 Standard und dem Framework Castor beschäftigt sich diese Diplomarbeit.

Die Diplomarbeit bietet eine Einführung in XML mit Konzentration auf Datenmodellierung. Ein solches in XML erstelltes Datenmodell wird als *XML Application* bezeichnet. Zur Erstellung von *XML Applications* stehen zwei Sprachen zur Verfügung: *Document Type Definition (DTD)* und *W3C XML Schema*. Letzteres erlaubt das Erstellen von strikt typisierten Modellen. Dieses besagt, dass jedes Element und jedes Attribut einen zugrundeliegenden Typ haben muss.

Die Java Einführung erklärt die Basis Elemente wie `class`, `interface`, `field`, `method` und *visibility*. Sowie die wichtigen Konzepte der objektorientierten Sprachen: *encapsulation*, *abstraction*, *inheritance* und *polymorphism*.

Im *Mapping* Kapitel wird erklärt wie der Datenaustausch zwischen Java und XML realisiert werden kann. Die grundlegende Unterscheidung ist, ob Wissen über das Datenmodell die Zuordnung beeinflusst. Es werden die Standards *DOM*, *JAXP* und *JAXB2* erklärt und die teils zugehörigen Implementierungen wie *XStream*, *JAXME*, *XMLBeans*, *Castor* und *JAXB2 Reference Implementation* vorgestellt.

Der praktische Teil der Diplomarbeit startet mit einer Einführung in und Diskussion des *JAXB2 Standard* und einer Analyse ob und inwieweit *Castor* die Anforderungen des Standards bereits erfüllt. Die Realisierung des *Castor JAXB2 Layer* und dessen Diskussion bilden den Hauptteil der praktischen Diplomarbeit.

## 2 Kurzdarstellung

## 3 Motivation

The goal of this thesis is to enhance *Castor* to be fully *JAXB2* compliant and discuss the completeness of mapping between Java and XML. Object-XML mapping is an established technology. The most recent standard of Object-XML mapping in Java is *JAXB2*. The thesis starts with an introduction into XML (focus on XML for data modeling) and Java (focus on Java for data modeling) continues with a discussion of mapping possibilities to finally reach the *JAXB2* compliance implementation for *Castor*, which has been implemented as part of this thesis.

*JAXB2* is a standard released in 2006 by Sun developed by the working group of JSR 222. It had a predecessor version 1 which was defined through JSR 31 and released on March 4, 2003. Some of the major goals of *JAXB2* are:

- Support all definitions of the *W3C XML Schema Language*
- Use and support Java 5 language features (e.g. annotations and generics)
- Portability of annotated classes between *JAXB2* implementations

The exact list of goals can be read in *The Java Architecture for XML Binding (JAXB) 2.0* – [SV06, page 4ff].

*Castor* is a widely used framework for mapping between Java and relational databases and between Java and XML. The first release dates back to 1999 and *Castor* is still in active development (maintenance and enhancements). *Castor* has its own style how to map between a *W3C XML Schema* definition and Java classes. The mapping follows no standard and is (should be) complete.

### 3.1 Research question

*Object-XML mapping with JAXB2* discussed by implementing *JAXB2* compliance for *Castor*.

The questions discussed in this thesis are:

- What are the limits of Object-XML mapping?

### 3 Motivation

- What are the limits of *JAXB2*?
- Can *Castor* reach full *JAXB2* compliance? Is a framework that has been developed and enhanced over nearly a decade flexible enough to be extended for *JAXB2* compliance?
- What are the features of *Castor* that outmatch a pure *JAXB2* implementation?

## 3.2 Research activity

*Object-XML mapping with JAXB2* builds on a stack of technologies. To clarify the theoretical background this thesis provides a brief introduction into:

- Extensible Markup Language (XML)
- Data modeling with/for XML
- Usage scenarios of XML
- Java
- Data modeling with/for Java
- Mapping between Java and XML
- *JAXB2*
- *Castor*

The applied part of the thesis is an implementation of an add-on for *Castor* that:

- Provides the *JAXB2* compliant API
- Uses *Castor* underneath to execute marshaling and unmarshaling
- Can read and interpret the *JAXB2* mapping definitions
- Can create *W3C XML Schema Definition* files from Java classes and mapping information

The implementation aspects are discussed in this thesis.

## 4 Extensible Markup Language (XML)

XML (Extensible Markup Language) was released (recommended) in 1998 by the *World Wide Web Consortium (W3C)*. It is a base definition meant to be extended for application usage. XML was developed from *Standard Generalized Markup Language* by *reducing it to the max*. XML is currently one of the corner stones of many modern applications. In many articles it is even named *the lingua franca of the Web*.

The focus of this chapter is to outline the capabilities of XML in means of data format and data modeling.

It first describes the background of XML and then steps into data modeling and usage of XML.

### 4.1 Background of XML

XML is a *Markup Language* and it can be used to store information as *Semi Structured Data*.

#### 4.1.1 Markup Language

*Markup Language* got first mentioned by *William W. Tunnicliffe* at a conference in 1967 then called *generic coding*. The purpose in mind was to have a generic *marking up* of text to express the presentation style to be used without using printer (or more generally: output) specific codes.

Historically, electronic manuscripts contained control codes or macros that caused the document to be formatted in a particular way ("specific coding"). In contrast, generic coding, which began in the late 1960s, uses descriptive tags (for example, "heading", rather than "format-17"). Many credit the start of the generic coding movement to a presentation made by William Tunnicliffe, chairman of the Graphic Communications Association (GCA) Composition Committee, during a meeting at the Canadian Government Printing Office in September 1967: his topic – the separation of information content of documents from their format. – [tun96]

## 4 Extensible Markup Language (XML)

The focus of *Markup Language* was text processing for books, articles, newspapers – all kinds of narrative documents.

The probably most popular *Markup Language* is HTML – even it is not of the print sector – all screen presentation is controlled by the tags enclosing the text.

Listing 4.1: HTMLSample.html

```
1 <html>
2   <head>
3     <title>A simple HTML page</title>
4   </head>
5   <body>
6     <h1>A simple sample page</h1>
7     <p>
8       This shows a <b>simple</b> example of
9       how <b>HTML</b> is used as <i>Markup
10      language</i>.
11    </p>
12  </body>
13 </html>
```

Another prominent *Markup Language* is TeX. Originally developed by Donald Knuth it is mostly used in the academic world. It is very strong for typesetting mathematical formulars, to build complex tables and to build large documents often split into multiple sub-documents. Also this thesis is written using LaTeX which is a customised version of TeX.

### 4.1.2 Semi-Structured Data

*Semi Structured Data* means that data and information about the structure of the data are stored together. Relational databases on the other hand mostly have a data dictionary holding the structure information which is separated from the data itself. Interpreting data from a relational database without having the data dictionary is nearly impossible. The main advantage of *Semi Structured Data* is that the data can 'always' be interpreted as it contains the rules how to interpret. The only condition is that the structure description must be understood.

*Semi Structured Data* can e.g. be used for data exchange and for long time storage of data.

At data exchange the advantage is that the data format doesn't have to be agreed on by all parties. The data provider is offering the data and the receiving parties will be able to extract the data (or parts of it) as the structure information is sent with the data. As soon as the interpreter of the structure information is implemented it can be used for all further implementations.

At long-time storage of data it is often a problem that a set of exported data needs to be imported by a software that has replaced the one which created the data set. Using *Semi Structured Data* should make it easier for the replacing software to import such data sets.

The following examples show the same data (an address record) represented three times. Once in a flat format and twice using *Semi Structured Data* languages. The examples show that data represented as *Semi Structured Data* can be easily interpreted by a human reader and easily contain more complex information.

### Address record as Comma Separated Values(CSV)

The flat format is *Comma Separated Values(CSV)*. Using CSV only record like data can be stored which has no sub structure. In CSV a 'record' of data is represented as one line of a file, the 'column' values can be with double quotes and are separated using comma signs:

Listing 4.2: CSVSample

```
1 "Mary Lebow", "5 Main Street", "San Diego, CA", 91912, "619 332-3452", "664 223-4667"
```

### Address record as JavaScript Object Notation (JSON)

The first *Semi Structured Data* format is *JavaScript Object Notation (JSON)* [jso08]. JSON is a very young data format based on the programming language JavaScript and on the language feature that JavaScript objects can be created using JSON syntax. The following sample would allow to instantiate objects and fill them with the provided data:

Listing 4.3: JSONSample

```
1 {"addressbook": {"name": "Mary Lebow",  
2   "address": {  
3     "street": "5 Main Street"  
4     "city": "San Diego, CA",  
5     "zip": 91912,
```

## 4 Extensible Markup Language (XML)

```
6     },
7     "phoneNumbers": [
8         "619_332-3452",
9         "664_223-4667"
10    ]
11 }
12 }
```

### Address record as XML

The second *Semi Structured Data* format is XML (which will be explained in this thesis):

Listing 4.4: XMLSample.xml

```
1 <addressbook>
2   <name>Mary Lebow</name>
3   <address>
4     <street>5 Main Street</street>
5     <city zip="91912"> San Diego, CA </city>
6     <phoneNumbers>
7       <phone>619 332-3452</phone>
8       <phone>664 223-4667</phone>
9     </phoneNumbers>
10  </address>
11 </addressbook>
```

Information about SMGL, GML, *Markup Language* and *Semi Structured Data* can be found in: [sgm08a], [sgm08b], [w3c08a], ....

### 4.1.3 XML

XML the *Extensible Markup Language* was first released in 1998 by *World Wide Web Consortium (W3C)*. XML it is a *Markup Language* and it can be used to store information as *Semi Structured Data*. But XML is not meant to be used directly in applications. It is a meta language to be derived for specific application purposes which are then called *XML Applications*.

The history of XML seen on a timeline is as in Table 4.1.3.

*XML Documents* are trees of elements with exactly one root element. Every element in the tree has to have exactly one parent but a parent can have multiple children – so XML forms hierarchical trees. Overlapping of elements is not allowed. Besides child elements an element can also contain attributes and text content. Syntactically

Table 4.1: XML History

1967	Generic Coding
1968	GML by Goldfarb, Mosher, Lorie (IBM)
1986	SGML gets ISO 8879
1989	HTML by Tim Berners-Lee (CERN)
1998	Extensible Markup Language (XML) Version 1.0

elements consist of a start-tag, the element content and an end-tag. Start-tags begin with `<` and end with `>` or `/>` for empty elements. Besides the square brackets a start-tag contains the element name and a list of attribute name and value pairs. After the start-tag, and if it is not an empty element, text content and child element nodes follow in any order and multiplicity until the end-tag of the element is reached. The end-tag begins with `</`, ends with `>` and only contains the element name. For data modeling and mapping purposes only elements, attributes and text content is used. *XML Documents* can contain also processing instructions, comments and entity references – but these are handled by lower layers of XML parsing and not used for data representation or mapping.

XML names, which are e.g. used for element names and attribute names, can be built of nearly every letter or number out of every character set available (at least since XML 1.1). The only characters which are not allowed are white space characters and punctuation characters (e.g. `<` and `&`).

For detailed information about XML syntax please consult: *Extensible Markup Language (XML) 1.0* [xml98], *Extensible Markup Language (XML) 1.1* [xml04] or *XML in a Nutshell* [Mea04, chapter: XML Fundamentals].

*XML Documents* have to be at least *well formed* otherwise they must not be considered to be *XML Documents* at all and must not be processed. Well formed means that all rules for structure, names and character-set are followed. In addition a *XML Document* can be *valid*, which means it is well formed and conforms to a schema definition. Schema definitions can be written in Document Type Definition (which is part of the XML standard), W3C Schema Definition Language, RelaxNG or any other schema language.

## 4.2 XML for data modeling

To use XML for data modeling means to create an *XML Application* using one of the schema languages available. In this section the capabilities of *Document Type Definition* and *W3C XML Schema* are discussed.

### 4.2.1 Document Type Definition (DTD)

The *Document Type Definition (DTD)* language is defined in the *XML 1.0 Recommendation* [xml98]. It allows to define a *XML Application*. The big advantages of DTD are:

- It is defined in and available since the first XML recommendation.
- The DTD can be included into a document.
- The DTD can be referenced.

DTD allows to define a valid node tree by

- Specifying the list of child elements for an element.
- Optionally specifying the multiplicity of child elements in the categories: *zero or one*, *zero or more*, *one or more* and *exactly one*.
- Specifying the list of attributes of an element.
- Marking an element to be empty.
- Marking an element to have text content.

There are some disadvantages in using DTD for data modeling:

- All declared elements can be used as root element in *XML Documents*.
- The text content of an element cannot be constraint.
- Element definitions cannot be reused by extension or restriction.

And some general disadvantages of writing schema definitions using DTD:

- DTD is not written in XML syntax, which means you have to learn a new syntax in order to write it.
- DTD does not support namespaces.
- DTD has no constraints imposed on the kind of character data allowed, so datatyping is not possible.

- DTD has minimal support for code modularity and none for inheritance.
- Large DTDs are hard to read and maintain.
- DTD has limited ability to control whitespace.
- DTD has only limited documentation support, as you cannot use the structured documentation features available for schema notation.

Possible definitions using DTD are:

**Element with PCDATA** Defines an element that contains text content and no child elements. The text content can not be constraint in any way and the semantic meaning of the text can not be defined.

Listing 4.5: ElementWithPcdata.dtd

```
1 <!ELEMENT Hugo (#PCDATA)>
```

An element defined like above can be used as follows:

Listing 4.6: ElementWithPcdata.xml

```
1 <Hugo>
2   A PCDATA element can contain any text without further
3   information about the content.
4 </Hugo>
5 <Hugo>2008-06-07</Hugo>
6 <Hugo>3.14</Hugo>
```

The example above shows that any values given to `Hugo` are valid, even that the semantic meaning of these values varies from text to date to number.

**Element with sequence of children** DTD allows to define a sequence of child elements for an element. All child elements in the *XML Document* have to occur exactly in the same order as they are specified in the definition. A sequence of children has to be within parenthesis and all elements are separated by comma signs.

Listing 4.7: DTDSequenceOfChildren.dtd

```
1 <!ELEMENT FirstName (#PCDATA)>
2 <!ELEMENT MiddleName (#PCDATA)>
3 <!ELEMENT LastName (#PCDATA)>
4 <!ELEMENT Name (FirstName , MiddleName , LastName)>
```

The element definition above would lead to a *XML Document* like below:

## 4 Extensible Markup Language (XML)

Listing 4.8: DTDSequenceOfChildren.xml

```
1 <Name>
2   <FirstName>Leonhard</FirstName>
3   <MiddleName>Christoph</MiddleName>
4   <LastName>Schmidt</LastName>
5 </Name>
```

**Element with choice of children** The other possibility is to specify a list of children of which only one should be used at a time – a kind of mutual exclusion. Choices are built by a list of children within parenthesis separated using the | sign. A sample definition for a choice of children:

Listing 4.9: DTDChoiceOfChildren.dtd

```
1 <!ELEMENT Book (#PCDATA)>
2 <!ELEMENT Article (#PCDATA)>
3 <!ELEMENT WebPage (#PCDATA)>
4 <!ELEMENT Cite (Book | Article | WebPage)>
```

The element definition above would lead to a *XML Document* like below:

Listing 4.10: DTDChoiceOfChildren.xml

```
1 <Cite>
2   <Book>XML in a Nutshell</Book>
3 </Cite>
```

**Multiplicity of children** The multiplicity of children can be specified by three different symbols or no symbol at all. The possible values are:

- ? the child might occur not at all or once,
- + the child might occur once or often,
- \* the child might occur not at all or often,
- no sign added** the child has to occur exactly once.

A group of children is built by specifying them within parenthesis. The group follows the same multiplicity rules.

Listing 4.11: DTDChildrenWithMultiplicity.dtd

```
1 <!ELEMENT Hugo (child1, child2?, child3+, child4*)>
```

The above element declaration together with the assumption that all `child` elements are empty elements would lead to the following valid XML fragments:

Listing 4.12: DTDCChildrenWithMultiplicity.xml

```

1 <Hugo>
2   <child1/>
3   <child3/>
4   <child3/>
5 </Hugo>
6 <Hugo>
7   <child1/>
8   <child2/>
9   <child3/>
10  <child4/>
11 </Hugo>
12 <Hugo>
13  <child1/>
14  <child2/>
15  <child3/>
16  <child3/>
17  <child3/>
18  <child3/>
19  <child3/>
20  <child4/>
21  <child4/>
22  <child4/>
23  <child4/>
24 </Hugo>

```

**Element with mixed content** An element can contain a mix of text content and child elements.

Listing 4.13: DTDMixedContent.dtd

```

1 <!ELEMENT Biography ((#PCDATA | DateOfBirth | DateOfDeath)*)
  >

```

Listing 4.14: DTDMixedContent.xml

```

1 <Biography>
2   XML, the franca lingua of data exchange, was born
3   in <DateOfBirth>1998</DateOfBirth> and is still
4   under heavy use.
5 </Biography>

```

**Empty Element** Defines that the element must not contain child elements or text content, but may contain attributes.

#### 4 Extensible Markup Language (XML)

Listing 4.15: DTDEmptyElement.dtd

```
1 <!ELEMENT Hugo EMPTY>
```

Listing 4.16: DTDEmptyElement.xml

```
1 <Hugo />
```

**Any Element** An any element can contain any content and any child elements. The only condition is that the child elements are declared within the DTD. If an element can contain anything it is mostly a sign for a poor design.

Listing 4.17: DTDAnyElement.dtd

```
1 <!ELEMENT Book (#PCDATA)>
2 <!ELEMENT Article (#PCDATA)>
3 <!ELEMENT Hugo ANY>
```

Listing 4.18: DTDAnyElement.xml

```
1 <Hugo>
2   Content is also allowed as <Book>MyBook</Book> and event
3   the any element again<Hugo><Article>MyArticle</Article>
4 </Hugo>
```

**Attributes for an element** Attributes for an element are not declared all together in one line (like with element children) but each attribute has to be declared alone. There are data types for attributes and some characteristics like default value.

Listing 4.19: DTDAttribute.dtd

```
1 <!ELEMENT Hugo ANY>
2 <!ATTLIST Hugo Hugo-Attribute CDATA "hugo-default">
```

Listing 4.20: DTDAttribute.xml

```
1 <Hugo hugo-default="some_text" />
```

DTDs can also contain entity declarations – which provides a kind of string substitution mechanism that is used as far as to create generic DTDs – but this thesis focuses onto mapping and therefor onto data modeling with DTD – entity declarations are not discussed.

### 4.2.2 W3C XML Schema

#### Introduction

When XML was released DTD was the only schema language to define *XML Applications*. Three years later (2001) *W3C World Wide Web Consortium* released a new language to define *XML Applications: W3C XML Schema*.

*W3C XML Schema* is itself an *XML Application* which means that its syntax is pure XML and schema definitions written in *W3C XML Schema* can be validated with the same mechanisms like any other *XML Application*. It introduces the concept of types which means that every element is of a certain type. Types are provided by *W3C XML Schema* (built-in-types) and can be defined by the user.

All types are part of a hierarchy with a single root type: **anyType** and two branches: *simple-types* and *complex-types*.

The *simple-types* constrain the text content for a single element or attribute but do not allow child-elements. *W3C XML Schema* ships with a list of built-in *simple-types* which are contained in every programming language – e.g. `xs:integer`, `xs:double`, `xs:decimal`, `xs:float`, `xs:date`, `xs:dateTime`, `xs:time`, `xs:duration`, `xs:string`, `xs:base64Binary`, `xs:hexBinary`. User-defined *simple-types* are based on built-in or other user-defined *simple-types* and can be derived by restriction, union or list.

- *Restrictions* restrain the super-type with facets like `length`, `maxLength`, `minLength`, `minExclusive`, `maxExclusive`, `minInclusive`, `maxInclusive`, `pattern`, `fractionDigits`, `whiteSpace`.
- *Unions* combine the lexical space of multiple *simple-types* together to one *simple-type*.
- *Lists* form new *simple-types* that base on exactly one other *simple-type* but allow that multiple values of this type occur within the element or attribute separated by white spaces.

The other branch of types are *complex-types*, which are built to contain child elements, attributes and text content. *Complex-types* are derived from *complex-types* and *simple-types* or composed of *model-groups*.

- Derived from a *complex-type* by restriction or extension.
- Derived from a *simple-type* by restriction or extension.
- Composed from *model-groups* which collect child elements by `sequence`, `all`, `choice` or `group`.

## 4 Extensible Markup Language (XML)

Using *W3C XML Schema* to define a valid structure for *XML Documents* must not mean to explicitly define types, it allows to define valid structures using global element definitions that base on anonymous type definitions, element references and substitution. But as all globally defined elements can be used as root elements for documents this leads to the problem that multiple elements are valid to be used as root element.

*W3C XML Schema* thoroughly supports namespaces.

### Examples of capabilities of W3C XML Schema

The following section gives a list of *W3C XML Schema* definition examples showing the possibilities. This list is not meant to show every possible definition but to show all data modeling capabilities. It is also not listing all exact syntax definitions and rules. Please refer [sch01a], [sch01b] and [sch01c] to get all exact definitions and rules. The samples are for explanation purposes and are not meant to be best practice!

**Element of built-in type** The simplest possible definition – an element based on a built-in type.

Listing 4.21: ElementBasedOnSimpleType.xsd

```
1 <xs:element name="IntegerElement" type="xs:integer" />
```

**Simple-type named or anonymous** Simple-types can either have a name and be then referenced in element definitions or the simple-type is embedded in an element definition having no own name and being an anonymous simple-type.

Listing 4.22: SimpleTypeNamedVsAnonymous.xsd

```
1 <xs:simpleType name="AustrianZipCodeType">
2   <xs:restriction base="xs:integer">
3     <xs:length value="4" />
4   </xs:restriction>
5 </xs:simpleType>
6 <xs:element name="AustrianZipCode"
7           type="AustrianZipCodeType" />
8
9 <xs:element name="GermanZipCode">
10  <xs:simpleType>
11    <xs:restriction base="xs:integer">
12      <xs:length value="5" />
13    </xs:restriction>
14  </xs:simpleType>
```

```
15 </xs:element>
```

The resulting element definitions both solve the same problem but for `AustrianZipCode` the underlying type `AustrianZipCodeType` can be used multiple times. The type of `GermanZipCode` is anonymous and cannot be reused.

**Simple-type by restriction** Create a user-defined named simple-type by restricting a built-in type.

Listing 4.23: SimpleTypeByRestriction.xsd

```
1 <xs:simpleType name="AustrianZipCodeType">
2   <xs:restriction base="xs:integer">
3     <xs:length value="4" />
4   </xs:restriction>
5 </xs:simpleType>
6 <xs:element name="AustrianZipCode"
7   type="AustrianZipCodeType" />
```

Listing 4.24: SimpleTypeByRestriction.xml

```
1 <AustrianZipCode>2240</AustrianZipCode>
```

**Simple-type by union** Create a user-defined named simple-type by combining two different simple-types allowing values that are allowed by any of the combined types.

Listing 4.25: SimpleTypeByUnion.xsd

```
1 <xs:simpleType name="IntegerOrUnknownType">
2   <xs:union memberTypes="xs:integer">
3     <xs:simpleType>
4       <xs:restriction base="xs:string">
5         <xs:enumeration value="UNKNOWN" />
6       </xs:restriction>
7     </xs:simpleType>
8   </xs:union>
9 </xs:simpleType>
10 <xs:element name="IntegerOrUnknown"
11   type="IntegerOrUnknownType" />
```

Listing 4.26: SimpleTypeByUnion.xml

```
1 <IntegerOrUnknown>UNKNOWN</IntegerOrUnknown>
```

**Simple-type by list** A list of simple-types is used when all values are occurring within the same element or attribute content and are delimited by white space characters.

Listing 4.27: SimpleTypeByList.xsd

```
1 <xs:simpleType name="FirstNameListType">
2   <xs:list itemType="xs:string" />
3 </xs:simpleType>
4 <xs:element name="FirstNameList" type="FirstNameListType" />
```

Listing 4.28: SimpleTypeByList.xml

```
1 <FirstNameList>Leonhard Christoph</FirstNameList>
```

**Complex-type named or anonymous** Complex-types can either have a name and be then referenced in element definitions or the complex-type is embedded in an element definition having no own name and being an anonymous complex-type (completely equal to simple-types).

**Complex-type extending simple content** Mostly means that an element (without children) is extended to have attributes.

Listing 4.29: CmplxExtendingSmpl.xsd

```
1 <xs:complexType name="StringWithLanguageAttributeType">
2   <xs:simpleContent>
3     <xs:extension base="xs:string">
4       <xs:attribute name="language" type="xs:language" />
5     </xs:extension>
6   </xs:simpleContent>
7 </xs:complexType>
8 <xs:element name="StringWithLanguageAttribute"
9   type="StringWithLanguageAttributeType" />
```

Listing 4.30: CmplxExtendingSmpl.xml

```
1 <StringWithLanguageAttribute language="de">
2   Ein Text markiert mit seiner Sprache
3 </StringWithLanguageAttribute>
```

**Complex-type restricting simple content** Means that an element without children gets type facets.

Listing 4.31: CmplxRestrictingSmpl.xsd

```

1 <xs:complexType name="
  RestrictedStringWithLanguageAttributeType">
2   <xs:simpleContent>
3     <xs:restriction base="xs:string">
4       <xs:length value="30" />
5       <xs:attribute name="language" type="xs:language" />
6     </xs:restriction>
7   </xs:simpleContent>
8 </xs:complexType>
9 <xs:element name="RestrictedStringWithLanguageAttribute"
10           type="RestrictedStringWithLanguageAttributeType"
           />

```

Listing 4.32: CmplxRestrictingSmpl.xml

```

1 <RestrictedStringWithLanguageAttribute language="de">
2   Ein Text markiert mit seiner Sprache
3 </RestrictedStringWithLanguageAttribute>

```

**Complex-type of group reference, all, choice or sequence** Used to define the child element structure useable for element definitions. Using **all** means that all child elements can occur once or not and in any order. **sequence** builds a list of children that have to occur in the given order and each child with any multiplicity. Of all child elements listed in a **choice** only one can be used at once. Within complex type definitions on references to defined groups are allowed. **All**, **sequence** and **choice** can be hierarchically combined still the resulting XML is a flat list of child elements.

**all** All child elements of the complex-type that are grouped in **all** have to occur zero or one time.

Listing 4.33: CmplxTypeAll.xsd

```

1 <xs:element name="Name">
2   <xs:complexType>
3     <xs:all>
4       <xs:element name="given-name"
5                 type="xs:string" />
6       <xs:element name="middle-name"
7                 type="xs:string" minOccurs="0" />
8       <xs:element name="last-name"
9                 type="xs:string" />
10    </xs:all>
11  </xs:complexType>
12 </xs:element>

```

## 4 Extensible Markup Language (XML)

Listing 4.34: CmplxTypeAll.xml

```
1 <Name>
2   <given-name>Amelie</given-name>
3   <last-name>Schmidt</last-name>
4 </Name>
```

**sequence** Child elements that are grouped together using **sequence** have to keep the order in which they are defined also in the resulting XML fragment. The multiplicity of child elements in a **sequence** can be defined freely.

Listing 4.35: CmplxTypeSequence.xsd

```
1 <xs:element name="Name">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="given-name"
5                 type="xs:string"
6                 maxOccurs="unbounded" />
7       <xs:element name="middle-name"
8                 type="xs:string"
9                 minOccurs="0" maxOccurs="1" />
10      <xs:element name="last-name"
11                type="xs:string"
12                minOccurs="1" maxOccurs="1" />
13    </xs:sequence>
14  </xs:complexType>
15 </xs:element>
```

Listing 4.36: CmplxTypeSequence.xml

```
1 <Name>
2   <given-name>Amelie</given-name>
3   <given-name>Isabell</given-name>
4   <last-name>Schmidt</last-name>
5 </Name>
```

**choice** All child elements that are grouped together using **choice** exclude each other. Only one element of those within the choice can occur where the choice is used.

Listing 4.37: CmplxTypeChoice.xsd

```
1 <xs:element name="Name">
2   <xs:complexType>
3     <xs:choice>
```

```

4     <xs:element name="simple-name"
5           type="xs:string" />
6     <xs:sequence>
7       <xs:element name="given-name"
8           type="xs:string" />
9       <xs:element name="middle-name"
10          type="xs:string" />
11      <xs:element name="last-name"
12          type="xs:string" />
13    </xs:sequence>
14  </xs:choice>
15 </xs:complexType>
16 </xs:element>

```

Listing 4.38: CmplxTypeChoice.xml

```

1 <Name><simple-name>Hugo</simple-name></Name>
2
3 <Name>
4   <given-name>Amelie</given-name>
5   <middle-name>Isabell</middle-name>
6   <last-name>Schmidt</last-name>
7 </Name>

```

**group reference** group definitions can only be used through group references in complex types. The next description explains how to create a group.

Listing 4.39: CmplxTypeGroupRef.xsd

```

1 <xs:group name="nameGroup">
2   <xs:choice>
3     <xs:element name="simple-name"
4         type="xs:string" />
5     <xs:sequence>
6       <xs:element name="first-name"
7           type="xs:string" />
8       <xs:element name="middle-name"
9           type="xs:string" minOccurs="0" />
10      <xs:element name="last-name"
11          type="xs:string" />
12    </xs:sequence>
13  </xs:choice>
14 </xs:group>
15 <xs:element name="name">
16   <xs:complexType>
17     <xs:group ref="nameGroup" />

```

#### 4 Extensible Markup Language (XML)

```
18 </xs:complexType>
19 </xs:element>
```

Listing 4.40: CmplxTypeGroupRef.xml

```
1 <name><simple-name>Pink Floyd</simple-name></name>
2
3 <name>
4   <first-name>Leonhard</first-name>
5   <last-name>Schmidt</last-name>
6 </name>
```

The beauty of **sequence**, **all**, **choice** and **group** shows when they are used together. A flat XML structure, just one level, can be described using multiple layers of compositors building complex rules of which elements can occur together and which must not.

If layers of complex types are used to implement such rules it would lead to multiple layers in the resulting *XML Document* which might not be what is wanted.

It is a trade-off: multiple layers of compositors are more elegant, but harder to understand – the same solution with complex types is easier to understand but the resulting XML structure is not so nice.

**Group definition** Groups can only be defined globally and used through group-references that are part of **complexType**, **choice**, **sequence**, **restriction** or **extension**. At a group reference the content of the group is inserted at the place of the reference.

Listing 4.41: NameGroup.xsd

```
1 <xs:group name="nameGroup">
2   <xs:choice>
3     <xs:element name="simple-name"
4       type="xs:string" />
5     <xs:sequence>
6       <xs:element name="first-name"
7         type="xs:string" />
8       <xs:element name="middle-name"
9         type="xs:string" minOccurs="0" />
10      <xs:element name="last-name"
11        type="xs:string" />
12    </xs:sequence>
```

```

13 </xs:choice>
14 </xs:group>
15 <xs:element name="name">
16   <xs:complexType>
17     <xs:group ref="namegroup" />
18   </xs:complexType>
19 </xs:element>

```

Listing 4.42: NameGroup.xml

```

1 <name><simple-name>Pink Floyd</simple-name></name>
2
3 <name>
4   <first-name>Leonhard</first-name>
5   <last-name>Schmidt</last-name>
6 </name>

```

**Complex-type extending complex content** Means that an element with children is extended by additional children – element or attribute.

Listing 4.43: CmplxTypeExtension.xsd

```

1 <xs:complexType name="interpret">
2   <xs:sequence>
3     <xs:element name="stage-name"
4               type="xs:string" />
5   </xs:sequence>
6 </xs:complexType>
7
8 <xs:complexType name="artist">
9   <xs:complexContent>
10    <xs:extension base="interpret">
11      <xs:sequence>
12        <xs:element name="first-name"
13                  type="xs:string" />
14        <xs:element name="last-name"
15                  type="xs:string" />
16      </xs:sequence>
17    </xs:extension>
18  </xs:complexContent>
19 </xs:complexType>
20
21 <xs:element name="artist" type="artist" />

```

Listing 4.44: CmplxTypeExtension.xml

#### 4 Extensible Markup Language (XML)

```
1 <artist>
2   <stage-name>Falco</stage-name>
3   <first-name>Johann</first-name>
4   <last-name>Hoelzl</last-name>
5 </artist>
```

**Complex-type restricting complex content** Means that an element with children will be restricted to have only a subset of its children.

Listing 4.45: CmplxTypeRestriction.xsd

```
1 <xs:complexType name="interpret">
2   <xs:sequence>
3     <xs:element name="stage-name"
4               type="xs:string" />
5   </xs:sequence>
6 </xs:complexType>
7
8 <xs:complexType name="artist">
9   <xs:complexContent>
10    <xs:extension base="interpret">
11      <xs:sequence>
12        <xs:element name="first-name"
13                  type="xs:string" />
14        <xs:element name="last-name"
15                  type="xs:string" />
16      </xs:sequence>
17    </xs:extension>
18  </xs:complexContent>
19 </xs:complexType>
20
21 <xs:element name="artist" type="artist" />
```

Listing 4.46: CmplxTypeRestriction.xml

```
1 <name><simple-name>Pink Floyd</simple-name></name>
2
3 <name>
4   <first-name>Leonhard</first-name>
5   <last-name>Schmidt</last-name>
6 </name>
```

**Complex-type with mixed content** Allows to create elements that contain text content, child elements and attributes.

Listing 4.47: CmplxTypeMixed.xsd

```

1 <xs:complexType name="artist-biography" mixed="true">
2   <xs:all>
3     <xs:element name="stage-name"
4       type="xs:string" minOccurs="0" />
5     <xs:element name="birth-date"
6       type="xs:date" minOccurs="0" />
7     <xs:element name="dead-date"
8       type="xs:date" minOccurs="0" />
9     <xs:element name="first-name"
10      type="xs:string" minOccurs="0" />
11    <xs:element name="last-name"
12      type="xs:string" minOccurs="0" />
13  </xs:all>
14 </xs:complexType>
15 <xs:element name="artist-biography" type="artist-biography"
   />

```

Listing 4.48: CmplxTypeMixed.xml

```

1 <artist-biography>
2   <stage-name>A</stage-name> was born on <birth-date>
3     1961-10-05</birth-date> in London. His real name is <
4     first-name>Karl</first-name> <last-name>Hugo</last-name
5     > and he started his carrer as singer with 23.
6 </artist-biography>

```

**Any type** Any type is the root type of all types including user-defined simple or complex types. If an element is defined to be of type *anyType* this element can hold values of literally any type.

**Any element** Is a kind of wildcard – when used inside a complex type it allows to be replaced by any element.

Listing 4.49: AnyElement.xsd

```

1 <element name="purchaseReport">
2   <complexType>
3     <sequence>
4       <element name="regions"
5         type="RegionsType" />
6       <element name="parts"
7         type="PartsType" />
8       <element name="htmlExample">
9         <complexType>
10          <sequence>

```

## 4 Extensible Markup Language (XML)

```
11         <any namespace="http://www.w3.org/1999/xhtml"
12             minOccurs="1" maxOccurs="unbounded"
13             processContents="skip"/>
14     </sequence>
15 </complexType>
16 </element>
17 </sequence>
18 <attribute name="period" type="duration"/>
19 <attribute name="periodEnding" type="date"/>
20 </complexType>
21 </element>
```

**Any attribute** Also a kind of wildcard – when used inside a complex type it allows to be replaced by any attribute.

**Attribute group** Attribute groups (`attributeGroup`) are the only way to define a fixed set of attributes that can be reused in exactly this definition. Attributes can be part of complex-types and therefore be reused by reusing the complex-type. But to have a set of attributes that can be reused is only possible by creating an attribute group and use it by reference (similar to `group`). Attribute group definitions have to be global.

Listing 4.50: AttributeGroup.xsd

```
1 <xs:attributeGroup name="message-attributes">
2   <xs:attribute name="status" type="xs:string" />
3   <xs:attribute name="persistence-id" type="xs:integer" />
4 </xs:attributeGroup>
5
6 <xs:complexType name="person">
7   <xs:sequence>
8     ...
9   </xs:sequence>
10  <xs:attributeGroup ref="message-attributes" />
11 </xs:complexType>
```

**Redefine of a schema** A schema (the definition within) can be included by redefining the global `simpleType`, `complexType`, `group` or `attributeGroup` definitions. Definitions that have not been redefined are included as they are.

**Substitution groups** Substitution groups can be used to first define a (maybe abstract) base element definition and then declare another element which can be used as substitute. The critical point is that the two definitions do not have to have anything in common. The second is allowed to be used where ever the first is required but does not have to extend or restrict the first one.

Listing 4.51: substitution-group.xsd

```

1 <xs:element name="name" />
2
3 <xs:element name="simple-name"
4           type="string32"
5           substitutionGroup="name" />
6
7 <xs:element name="full-name"
8           substitutionGroup="name">
9   <xs:complexType>
10  <xs:all>
11    <xs:element name="first"
12              type="string32" minOccurs="0" />
13    <xs:element name="last"
14              type="string32" />
15  </xs:all>
16 </xs:complexType>
17 </xs:element>
18
19 <xs:element name="character">
20   <xs:complexType>
21     <xs:sequence>
22       <xs:element ref="name" />
23       <xs:element ref="born" />
24       <xs:element ref="qualification" />
25     </xs:sequence>
26   </xs:complexType>
27 </xs:element>
28
29 <xs:element name="author">
30   <xs:complexType>
31     <xs:sequence>
32       <xs:element ref="name" />
33       <xs:element ref="born" />
34       <xs:element ref="dead" minOccurs="0" />
35     </xs:sequence>
36   </xs:complexType>
37 </xs:element>

```

**Element reference** Besides specifying a type for an element or including an anonymous type definition into the element definition a third approach exists. It is the element reference. Element reference means that at places where an element definition is expected only a reference to an existing global element definition is put. The major advantage of this approach is to mix namespaces within e.g. a sequence of elements.

## 4 Extensible Markup Language (XML)

Listing 4.52: ElementRef.xsd

```
1 <xs:element name="title" type="xs:string" />
2
3 <xs:sequence>
4   <xs:element ref="title" />
5 </xs:sequence>
```

### 4.3 Usage of XML

The two major usages of XML are *Narrative Documents* and *Record Like Data*.

#### 4.3.1 Narrative Documents

*Narrative Documents* have the goal to publish information in a rather loose structure and for human reading. The characteristics of *Narrative Documents* are:

- The XML tree of *Narrative Documents* tends to be wide but not deep
- Mixed content is in heavy use
- Data types are nearly not used
- The documents are processed in a parsing style but seldom used to be mapped into a data model
- Not used in Web Services
- Documents are created by humans (e.g. writers) not by applications
- Documents are processed to be presented to and read by humans

#### 4.3.2 Record Like Data

*Record Like Data* are highly structured and meant to be used for communication between applications – not to communicate with or by humans. E.g. many applications use databases to persist into and also read data from. Many application design principles like *Entity Relationship Modeling* or *Object Oriented Modeling* base on *Record Like Data*. The characteristics of record like XML data are:

- All elements and attributes are typed and typing is enforced
- Mixed content is rarely used
- Much structure – less content

- Mapping to other applications is highly desired
- *XML Documents* of this kind are mostly created by mapping of one system and consumed by mapping again
- Often used as contracts between systems – e.g. WebServices
- Compatibility is major

It is necessary to dig further and split up Record Like Data style into three more classifications:

**Messages** Especially with messages it is very important that the receiver can rely on the quality of the *XML Document* received to be able to successfully interpret it. A message provider commits itself to send only valid messages. Messages should also be self contained to allow the receiver to work with the information received without requesting further messages.

**Data dumps** Data dumps are characterized to contain a large number of records, the records inside are self contained and can be processed each at a time. The data needs to be interpretable even after a long period of time.

**Configuration** XML is also used for configuration purposes. Configuration files require to be easily human readable and editable! Configuration needs to be extendable because files from older releases should still be useable and have meaningful defaults.

## 4.4 XML Standards

The time table in Table 4.4 shows the *W3C World Wide Web Consortium* recommendations related to XML and data modeling with XML. The full list of all *W3C World Wide Web Consortium* recommendations and also all working drafts can be found at: [\[w3c08b\]](#).

## 4.5 Summary

*Markup Languages*, invented for the layouting and publishing sector, and *Semi Structured Data*, which combine structure information and data into the same file, are the roots of XML. *Extensible Markup Language* was recommended (released) in 1998 and was quickly adopted in all kinds of applications. The usage ranging from *Narrative Documents*, *Configuration Files*, *Programming Languages* to *Record Like Data*. Data exchange (e.g. messaging and WebServices) and long time persistence are the two sub categories of *Record Like Data*.

## 4 Extensible Markup Language (XML)

For further information read:

- *XML recommendation* [[xml98](#)]
- *XML recommendation 1.1* [[xml04](#)]
- *XML in a Nutshell* [[Mea04](#)]
- *Effective XML* [[Har04](#)]

The wide range of usage and quick adoption is possible because XML is itself a meta format that needs to be extended for usage. Such an extension is call a *XML Application* which combines all rules for one specific data model. Documents created according to the rules of XML and a *XML Application* are well-formed and valid. *XML Applications* can be defined using languages like *Document Type Definition (DTD)* and *W3C XML Schema Definition Language*.

### 4.5.1 DTD Summary

DTD is sufficient to generate simple *XML Applications* – using child element structures, choices of children, constraint multiplicity and define attributes for an element. It is available where ever XML is present and can be included into the *XML Document*. It fails when the content of elements should be constraint, namespaces are required or modularity is needed.

For further information read:

- *XML recommendation* [[xml98](#)]
- *XML in a Nutshell* [[Mea04](#)] – chapter *Document Type Definitions (DTDs)*
- *Effective XML* [[Har04](#)] – chapter: *Part 1 Syntax*

### 4.5.2 W3C XML Schema Summary

*W3C XML Schema* has been designed to be the language of choice for defining *XML Applications*. It is pure XML and itself an *XML Application*. It introduces types and demands that every element has to base on a type. Types are organized in a hierarchy with two major branches: simple-types and complex-types. *W3C XML Schema* provides built-in types (that are simple-types) and allows that users define their own types (simple or complex). Type definition can be based on another type by restriction, extension, union and list or by specifying the child elements in model-groups.

For further information read:

- *XML Schema* [[Har02](#)]

## 4.5 Summary

- *XML Schema Part 0: Primer Second Edition* [sch01a]
- *XML Schema Part 1: Structures Second Edition* [sch01b]
- *XML Schema Part 2: Datatypes Second Edition* [sch01c]
- *XML in a Nutshell* [Mea04][Chapter *XML Schemas*]
- *Effective XML* [Har04][Chapter *The W3C XML Schema Language*]

## 4 Extensible Markup Language (XML)

Table 4.2: History of XML Standards

### 1998

- Extensible Markup Language (XML)  
[\[xml98\]](#)
- Namespaces in XML 1.0  
[\[xml06a\]](#)
- Document Object Model (DOM) Level 1

### 1999

- XSL Transformations (XSLT) Version 1.0  
[\[xsl99\]](#)
- XML Path Language (XPath) Version 1.0  
[\[xpa99\]](#)

### 2000

- XHTML 1.0: The Extensible HyperText Markup Language  
[\[xht00\]](#)
- Document Object Model (DOM) Level 2  
[\[dom00a\]](#), [\[dom00b\]](#), [\[dom00c\]](#), [\[dom00d\]](#), [\[dom00e\]](#)

### 2001

- XML Schema  
[\[sch01a\]](#), [\[sch01b\]](#), [\[sch01c\]](#)
- XML Information Set  
[\[xml01b\]](#)
- RELAX NG  
[\[rel01\]](#)

### 2004

- Extensible Markup Language (XML) 1.1  
[\[xml04\]](#)
- Namespaces in XML 1.1  
[\[xml06b\]](#)

### 2007

- XQuery 1.0 and XPath 2.0 Data Model (XDM)  
[\[xml07b\]](#)

# 5 Java for Data Modeling

This chapter provides a rapid introduction into object orientation and Java with a focus on data modeling.

## 5.1 Object Oriented Approach

Object orientation as a programming approach was introduced to solve the problem of increasing complexity in hardware and software. Its goal is to divide a big and complex software into small pieces that can easily be understood and implemented by humans. The first object oriented programming language was *Simula* in 1967. Object orientation has the following basic elements:

**Class** A *Class* is the smallest logical unit in which object oriented languages divide. Classes have an identifying name and contain attributes and operations. The name of a class, the operations and the attributes have to build a logical unit. Attributes and operations within a class have to have distinct names and each attribute may occur only once in a class. Examples for classes would be **Interpreter**, **Artist**, **Band**, **Song**, **Recording**, **Label** if the goal is to build a music library. Often the terms *Class* and *Object* are used interchangeable – this thesis will distinguish between them.

**Object** Is an instance of a *Class*. *Objects* are also called *Instance*. E.g. **David Bowie** is an instance of class **Artist**. Instances live at maximum as long as their environment survives – for that time instances also have a unique identifier.

**Attribute** Attributes represent the state of an object/instance and are usually not accessible outside the class. In many object oriented languages:

- An attribute is either a scalar value or a reference to an instance of a class.
- Attributes within a class have no order and no multiplicity.
- If an attribute should allow multiple values it is required to reference to a kind of **array** or a special class that can handle multiple values e.g. a **Collection**.

**Operation** Operations form the behavior of a *Class*. Operations are also called *Methods* and *Messages*. Any state change within an object oriented system is triggered

by calling an operation of a class. Operations may have input parameters and may have a return value. Operations can be divided into three categories:

- Constructor / Destructor** The actions required to create / destroy an instance.
- Setter or Getter** Operations that are used to set or receive the value of a certain attribute without additional logic – *Setters* and *Getters* are required by some frameworks but no general must and violate encapsulation, see paragraph *Encapsulation* below for more information.
- Others** Operations that perform any behavior.

A simple class with attributes and operations can be seen in Figure 5.1.

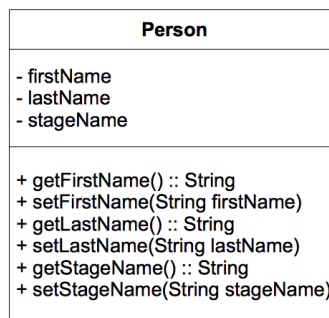


Figure 5.1: A simple Person class

The concepts of object orientation are:

**Encapsulation** Means that the implementation details of a class are hidden and only a specified interface can be used from outside. Encapsulation seems very simple but is very hard to achieve. For example *Getter* and *Setter* operations already reveal exactly which attributes are inside a *Class* and therefore violate *Encapsulation*. Some mapping (relational or XML) frameworks require *Getter* and *Setter* operations to access the attributes of a class but this can also be achieved with other approaches - e.g. aspect oriented programming, privileged access of private attributes.

**Abstraction** A class has to abstract real world entities. The goal is to find a level of abstraction that divides into classes of different behavior but still meaningful classes. Dividing too much would be confusing, dividing too less will lead to bloated classes with many control structures.

E.g. if the business model should model a music industry then the classes `Artist`, `Song`, `Recording`, `Band` and `Label` would be a good start but classes like

David Bowie, China Girl, Sony aren't abstract enough and `TwoPersonBand`, `ThreePersonBand` would be too fine grained.

**Inheritance** Inheritance is a hierarchy relation between classes in which each child is an extended version of the parent (e.g. `Artist` and `Band` are extended (more specialized) versions of `Interpreter`). Inheritance is also named generalization.

A sub-class (child class) inherits all attributes and operations of its super-class (parent class). A sub-class might overwrite (overload) parts of the behavior leading to a more specialized class, or a sub-class extends the super-class by adding features (attributes).

Together with inheritance a special kind of class is also introduced: *abstract class*. An abstract class cannot be instantiated by itself but through sub-classes. A sample class model for inheritance can be seen in Figure 5.2.

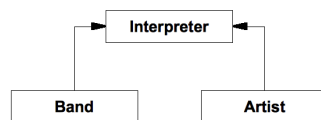


Figure 5.2: Inheritance example

**Polymorphism** Allows to treat sub-classes as super-classes (e.g. let `Espresso` be a sub-class of `Coffee`; then in every implementation expecting instances of `Coffee` will also accept instances of `Espresso`).

## 5.2 OO for Data Modeling

The base building blocks of object oriented languages are classes that may contain attributes and operations. The attributes are private (hidden from other classes, encapsulated) and can be accessed through operations only. Operations that allow direct manipulation of a single attribute are called setter and getter.

Usually an object oriented language comes with a set of base classes representing types like `String`, `Integer` and `Boolean` and also more complex definitions like `Collection`, `Set`, `Hash`, `Tree` and `Map`. Many object oriented languages have some kind of *standard library* providing complex building blocks as a generalized implementation. With this foundation *every* data model requirement can be implemented.

## 5 Java for Data Modeling

Design patterns document well proven and generalized solution fragments that can be used in own implementations. The most popular book about design patterns is: *Design Patterns* [EG95], it gives a good introduction into design patterns in general and also introduces 23 patterns. Another book focusing onto patterns to build enterprise applications is *Patterns of Enterprise Application Architecture* [Fow03]. And for example *Analysis Patterns: Reusable Object Models* [Fow96] which focuses on how repudiative business requirements can be solved.

### 5.3 Java Basics

Java is an object oriented programming language originally developed by Sun Microsystems. The first official version of Java was released in 1995. Some articles describing the history of Java are: *Java Technology: The Early Years* [ear03] and *The Java Saga* [jav95]. Java has syntactical roots in C and C++. But there are far more differences than equalities – some of the differences are:

- Code written in Java is compiled into bytecode and then executed on a (the) Java Virtual Machine (JVM).
- Object deallocation is done by a garbage collector and not by the application itself.
- Pointers – at least as they exist in C and C++ – are not available in Java
- Java has no header files
- Java does not support `struct` or `union`
- Java does not support multiple inheritance
- Java has the concept of `interfaces`; an interface is purely abstract defining only operations and a class can implement multiple interfaces (meaning that it provides all operations of different interfaces) but it can extend only on other class
- In Java object instances are always references only – no variable or attribute actually holds an object instance – this is a precondition for garbage collection
- As every usage of an object instance in Java is by reference it is possible to build dependency cycles e.g. `David Bowie::Artist` references many recordings, one of them would be `Blue Jean::Recording` and at the same time `Blue Jean::Recording` references `David Bowie::Artist` as its interpreter.
- Operations in Java are called *methods*

## 5.4 Java for Data Modeling

Besides classes Java supports primitive (scalar) data types like: `int`, `long`, `char`, `float` and *arrays*. Objects are always handled through references.

When using Java to build a data model the following building blocks are used:

**Simple class** A simple class with an identifying name and some attributes which all are of scalar types.

Listing 5.1: Artist.java

```

1 package at.grueneis.musiclibrary;
2
3 import ...
4
5 public class Artist {
6     private String firstName;
7     private String lastName;
8     private String stageName;
9
10    public Artist(
11        final String firstName,
12        final String lastName,
13        final String stageName
14    ) {
15        super();
16        this.firstName = firstName;
17        this.lastName = lastName;
18        this.stageName = stageName;
19    }
20
21    public String getFirstName() {
22        return firstName;
23    }
24
25    public String getLastName() {
26        return lastName;
27    }
28
29    public String getStageName() {
30        return stageName;
31    }
32 }
```

**Class with association to another class** As soon as a class holds one attribute which

is of a class-type the class has an association. Class-typed attributes always hold references to the class instance (object) only, not the complete object. So it is possible the object "Let's dance":Song points to object "David Bowie":Artist and at the same time object "David Bowie":Artist points to "Let's dance":Song. But none of the two objects holds a copy of the other.

Listing 5.2: ShowClassAssociation.java

```
1 Artist a = new Artist("David", "Bowie", "David_Bowie");
2 Recording r = new Recording("Let's_dance");
3 a.addRecording(r);
```

**Class extends another class** In Java if class A extends class B the following is true:

- A inherits all operations of B
- A can be used where ever B is expected
- A might extend B by adding operations
- A might overwrite the operations of B (but be aware that the rules above have to be kept still)

**Class implements an interface** If a class implements an interface that means that the signature of operations of the interface is provided by the class...

**Private vs protected vs public** Java allows to qualify attributes and operations to have a certain visibility.

- An operation or attribute marked to be private can be access only from other operations of the same class. Even derived classes are not allowed to access private attribute or operations.
- Protected operations or attribute are visible to all classes of the same package and to derived classes.
- Visibility public means full access.

Visibility is the tool to enforce encapsulation. E.g. making attributes anything else then `private` is more or less forbidden – a class consisting of public attributes only would mock `structs` known from C. In general it is good to keep as much of a class `private` as possible leading to strict encapsulation.

**Field access vs method access** In Java access to fields is usually granted through `set` and `get` methods (setters and getters). Direct access of attributes is not possible as those are private. Some classes should be immutable, meaning that after an object was instantiated its state can no longer be modified. There is a way in Java to even access private attributes and it is used by persistency and mapping frameworks.

## 5.5 Summary

Java is an *Object Oriented Programming Language*, released in 1995 by Sun Microsystems, consisting of **classes**, **fields** and **methods**. It follows the object oriented principles of *Encapsulation*, *Abstraction*, *Inheritance* and *Polymorphism*.

For data modeling the building blocks are **classes**, **fields** of scalar types, **fields** that reference other classes, *inheritance*, **interfaces**, *visibility* and **methods** that allow the implementation of business logic. As object instances are associated by reference it is possible and even probable that cyclic dependencies occur in an object graph.

To build more complex data models Java provides a standard library including classes like: **Map** to store name, value pairs, **ArrayList** to keep a flat list of objects and preserving the order and **HashTree** representing a simple tree implementation.

Beyond that standard library *Design Patterns* document well proven and generalized solution fragments that can be used in own implementations.

## 5 *Java for Data Modeling*

## 6 Mapping

In the previous chapters two different technology stacks have been introduced: XML and Java. XML is a meta data format to store information – used as silver bullet for many cases of saving and exchanging data. Java an object oriented programming language and a large ecosystem to build enterprise applications.

Enterprise applications tend to store information and to exchange data with others. For short and mid term data storage relational databases are the first choice but for data exchange and long time storage XML is chosen. The advantages of XML in these cases are that a closed data format like it is used by most relational database systems cannot be interpreted by recipients.

Java needs to provide easy and robust mechanisms to store Java runtime objects as *XML Document* or to interpret *XML Documents* into Java runtime objects.

### 6.1 Motivation of Mapping

Java needs a way to store Java runtime objects as *XML Documents* and to load *XML Documents* into Java runtime objects. The *simplest* way to read *XML Documents* is parsing it with a XML aware parser. Parsers create events for each XML chunk (like start tag, end tag, characters, ...). This leaves the responsibility of interpreting the XML chunks and managing the XML hierarchy (mostly a stack) to the application.

Mapping means that a XML fragment is completely mapped to Java objects. This approach requires the implementation of steps like interpretation of the parsing events, keeping the parsing state and transforming of character sets which are equal for all applications and can be implemented generically. The user of a mapping framework just specifies how elements and attributes of an XML fragment should be mapped into classes and class attributes and the rest is done by the framework. All the repetitive work that would be required when working directly on a parser can be reused.

## 6.2 Mapping without business domain knowledge

The simplest form of mapping is to map the XML tree nodes into Java objects that represent a XML node model. Consisting of e.g. `Document`, `Element`, `Attribute`, `Content Node` and `Processing Instruction`.

This kind of mapping has some major **advantages**:

- The Java object model is equal for all *XML Documents* mapped and completely independent of the application.
- It is not required to know anything about the *XML Application*.
- It can be used for every well formed *XML Document* without any adoptions at the implementation or configuration.
- The mapping can be considered to be complete – XML is a hierarchical tree of nodes and such structures are available in Java.

The biggest **disadvantage** is that the mapped Java tree represents the XML node structure but not the business model. Usually it is required that a specific business model is filled with the information read from the *XML Document*. To do this, starting with tree node mapping, requires big implementation effort to map from the tree to the business model.

### 6.2.1 Standards

#### Document Object Model (DOM)

*W3C World Wide Web Consortium* has defined a standard for low level XML mapping: **Document Object Model (DOM)** [DOM]. DOM is programming (implementation) language agnostic and defines a set of interfaces to provide an API. The center-piece (core) of the DOM standard is an object model representing the XML structures to explore and manipulate the XML. It consist of interfaces like: `Document`, `Node`, `CharacterData`, `Attr`, `Element`, `Text`, `Comment` and more. DOM also provides definitions to load and store documents, XPath processing, and more. DOM implementations exist in nearly every programming language.

The DOM standard is programming language agnostic and started with a set of interfaces to manipulate XML or HTML trees. Currently DOM is available in the third edition (Level 3) covering far more requirements and consisting of multiple modules each responsible for a set of functionality.

### Java API for XML Processing (JAXP)

Sun has created a standard named **Java API for XML Processing (JAXP)** that defines the Java specific APIs for SAX, StAX, DOM, XSTL processing and related topics. Any vendor can provide an implementation of this API. Since Java 5 the API and the reference implementation are part of JDK/JRE.

JAXP 1.0 was developed by the working group of JSR 4 ([jsr00]), released in 2000 and supported SAX 1.0 and DOM level 1. Release 1.1 and 1.2 were results of JSR 63 ([jsr01]) upgrading to the then latest DOM and SAX specification. Release 1.3 supports DOM level 3 and was extended by StAX which had been the result of JSR 173 ([jsr04]).

#### 6.2.2 Implementations

Java implementations of node level mapping are e.g.:

**JAXP reference implementation** Besides supporting other APIs also JAXP provides a DOM implementation following the DOM standard. The JAXP reference implementation is still under active development – a release 1.4 on the way. Please consult [jax08b] for further information.

**Dom4J** dom4j is a Java library supporting DOM standard API but also enhancing it e.g. to allow parsing documents until a point of interest and from that point on have it as DOM document; its latest release is 1.6.1 dating to May 16, 2005; please consult [dom05] for further information

**JDOM** JDOM is another Java implementation to map at node level. It supports the DOM standard API but also enhances it by e.g. using `Collection` instead of `arrays` (e.g. `Element[]`).

*While JDOM interoperates well with existing standards such as the Simple API for XML (SAX) and the Document Object Model (DOM), it is not an abstraction layer or enhancement to those APIs. Rather, it provides a robust, light-weight means of reading and writing XML data without the complex and memory-consumptive options that current API offerings provide. – [jdo07]*

### 6.3 Mapping with business domain knowledge

The logical next step is to map with business know-how – this kind of mapping is also known as *O/X Mapping*. Mapping with business know-how means that business model specific Java objects and business model specific *XML Documents* are

## 6 Mapping

bidirectionally transformed in one step. The mapping framework is responsible to handle all XML parsing, translation from XML chunk to Java object, translation from Java object to XML chunk and XML writing. This approach requires a definition step before the actual marshaling (Java to XML) or unmarshaling (XML to Java) where the mapping rules are defined.

Three approaches are possible how the business know-how is provided to the mapping framework:

**Schema First** At *Schema First* the business model is designed as a *XML Application* and then the Java object model is generated from it. It is mostly used when general services are in place (existing) which will be consumed by various applications implemented in different languages. This approach is often encountered when working with WebServices.

There might be situations with very complex schema definitions that are hard to map but today's frameworks can handle every schema definition when the resulting object model can be defined freely.

**Object First** The *Object First* approach assumes that the Java business model exists and the XML schema definition is derived from it. This approach is more likely used for long time persistence or for interfaces within an application. Mapping the state of Java objects to XML and create a schema definition representing the object state is solved. Mapping of the business logic cannot be done as there is no place to put business logic into a schema definitions.

**Both predefined** The XML schema definition and the Java business model have both been developed independently and need to be mapped.

This approach is likely when an existing application (existing object model) gets connected to an existing data provider (existing schema definition). This situation tends to be the most complex use case as many transformations have to be expected.

### 6.3.1 Standards

There is only one standard for Java-XML mapping: JAXB.

#### Java API for XML Binding (JAXB)

JAXB version 1.0 was created as result of JSR 31 at March 4, 2003. There had been a reference implementation from Sun and JaxMe from Apache Group.

## 6.3 Mapping with business domain knowledge

JAXB 2 followed at May 11, 2006 as a result of JSR 222. JAXB2 introduced the following new features:

- Full support of Java 5 language features (e.g. generics, annotations)
- Full support of *W3C XML Schema*

The JAXB2 standard is since developed further with new releases 2.1. and 2.2 but without an active JSR working group just by the JAXB2 reference implementation team.

### 6.3.2 Implementations

#### **XStream**

The approach of *XStream* is – as the name wants to imply – to stream objects into XML. The first versions of it gave little possibility to control how the resulting *XML Document* will look like, it serialized the Java objects into a XML file as is. In current release 1.3 of February 27, 2008 it already allows a lot of customizations on how the resulting XML should look like, e.g. aliases, serialize as element or attribute, custom conversion of classes and more. Still the focus is on serialization and de-serialization of Java objects. It still supports no *Schema First* approach, no *XML Application* and no XML validation.

For more details about *XStream* please visit: [\[xst08\]](#).

*XStream* seems to be an active project with a latest release 1.3 on February 27, 2008.

#### **XMLBeans**

*XMLBeans is a technology for accessing XML by binding it to Java types.*  
– [\[xml07a\]](#)

This quote describes the focus of *XMLBeans*, it is designed for the *Schema First* approach and doesn't support any other. *XMLBeans* requires that first a *W3C XML Schema* exists, this is then *compiled* into a set of Java interfaces that are used at runtime. *XmlBeans* does not create plain Java objects.

For more details about *XMLBeans* please visit: [\[xml07a\]](#).

No news since June 1, 2007. Latest release is 2.3.0.

## 6 Mapping

### **JAXMe**

Is an implementation of the *JAXB1* standard but there had been no development to reach *JAXB2* compliance.

For more details about *JAXMe* please visit: [\[jax06\]](#).

No news since October 24, 2006. Latest release is 0.5.2.

### **JAXB RI**

The reference implementation of *JAXB2 Standard* by Sun Microsystems was completely from scratch using and supporting all language features of Java 5. The first implementation was part of the *GlassFish* project and downloadable as add-on for Java 5. JDK 6 includes (and JDK 7 will include) a release of the *JAXB2 Reference Implementation* – it is no longer an add-on.

*Welcome to the JAXB Reference Implementation Project. This project is part of Project Metro and is in the Glassfish community at java.net. This project develops and evolves the code base for the reference implementation of the JAXB specification. The current code base supports JAXB 1.0, 2.0, and 2.1 but the project will track future versions of the JAXB specifications.*  
– [\[jax08a\]](#)

For more details about the *JAXB2 Reference Implementation* please visit: [\[jax08a\]](#).

The project is still active with maintenance, further development and enhancements – the latest releases are 2.0.5 from January 22, 2007 and 2.1.8 from August 29, 2008.

The ill of this project is that it also develops and enhances the standard without an active working group.

### **Castor**

*Castor* handles Object-Relational mapping and Object-XML mapping.

*Castor is an open source data binding framework for moving data from XML to Java programming language objects and from Java to databases.*  
[\[cas08\]](#)

An architecture overview of the *Castor* Object-XML part can be seen in Figure 6.1. The XML specific modules are:

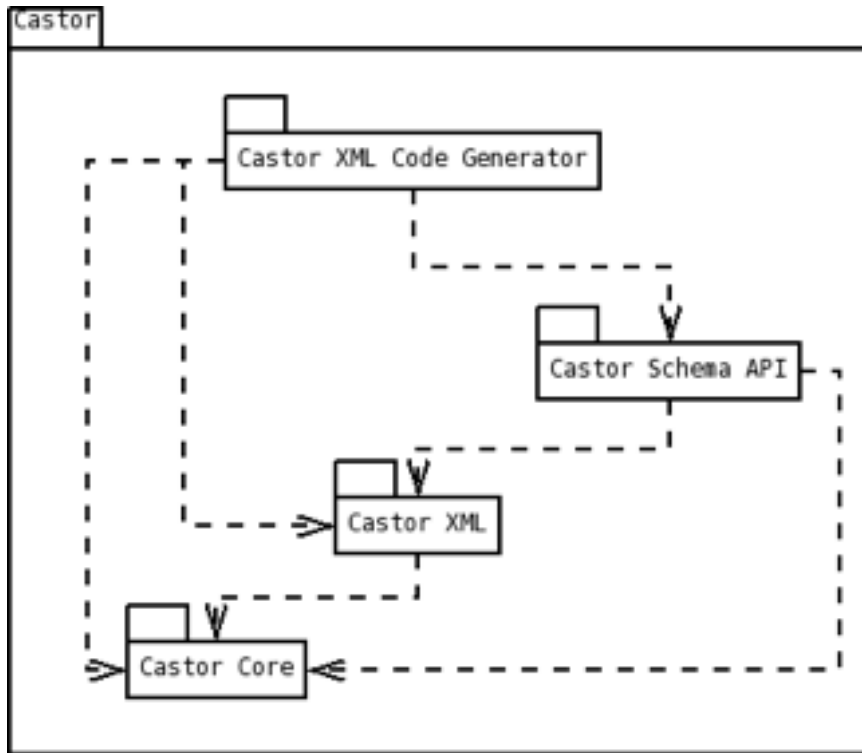


Figure 6.1: Castor XML architecture overview

**Castor XML** The part of *Castor* that actually performs the marshaling (Java to XML) and unmarshaling (XML to Java).

The mapping definition for marshaling and unmarshaling could be provided (is determined) through three approaches:

- A mapping configuration file in XML syntax
- Descriptor classes – those classes are *Castor* specific; mostly created by code generation; allow that the mapping information resides within compiled classes
- Ad hoc introspection of classes using introspection

**Castor XML Code Generator** The code generator allows to create Java classes for a specific *W3C XML Schema* definition. When Java code is generated for each of the domain classes a descriptor class is generated holding the mapping information of this domain class.

**Castor Schema API** A Java model representing the *W3C XML Schema* language. Using this project *W3C XML Schema* files can be created, read and written.

## 6 Mapping

The *Castor* project provides support to use it together with other tools through sub projects like:

**Castor Spring for XML** To allow usage of *Castor* from *Spring OXM*.

**Support for ANT and Maven** To execute *Castor Code Generator* as part of a project build cycle. The build can be implemented using ANT or Maven – both are supported.

Castor supports all *XML Application* definitions written in *W3C XML Schema*. Castor development started at 1999 and it is still enhanced and maintained. Its latest release is 1.2 which still developed to support Java 1.4 or above. A new release 1.3 starts its release candidate phase in October 2008. The 1.3 release will support Java 5 and above (but no longer Java 1.4).

### 6.4 Summary

To efficiently work with Java and XML it is required to have a simple and potent way to transform *XML Documents* into a Java object graph and vice versa. The simplest approach is to use a Java model that reflects the XML node tree. Such an implementation can be used for every *XML Document* without any modification but it doesn't solve the need to map a business model from XML to Java and back.

When a business model should be mapped and transformed from XML to Java (and back) in one step a so called *O/X Mapping* framework is required. Such frameworks start with a definition of the business domain as either *W3C XML Schema Definition*, Java object model or both, use this information to generate missing sources or mapping definitions and then work with the generated sources and mapping definition to provide one step transformation. Such frameworks are e.g. *Castor* and *JAXB2 Reference Implementation*.

## 7 Castor JAXB2 implementation

To get *Castor JAXB2* compliant it is required to:

- Analyze what *JAXB2* compliance implies.
- Check how far *Castor* already fulfills the requirements.
- Find out in which areas are modifications required and how to implement these modifications.
- Modify (enhance) *Castor* and implement a *Castor JAXB2 Layer*
- Examine where the limits are?

### 7.1 JAXB2 Overview

The overview of the *JAXB2* architecture is as shown in Figure 7.1 and can be found at [SV06][Page 18 Figure 3.1].

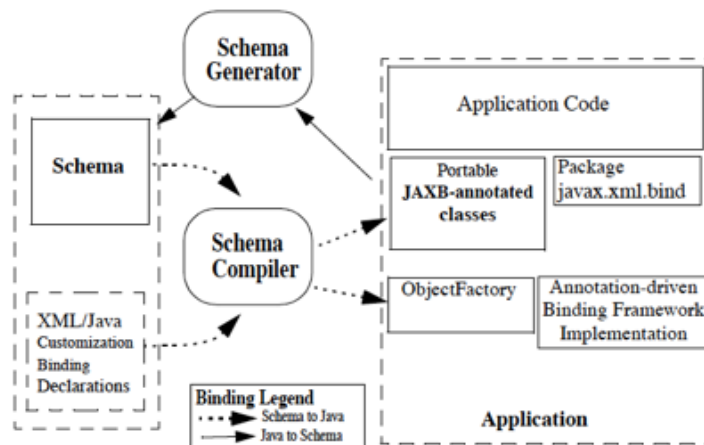


Figure 7.1: JAXB2 Architecture Overview

## 7 Castor JAXB2 implementation

*JAXB2* consists of four major building blocks:

**JAXB2 Annotations** are how the mapping between Java objects and XML is defined in *JAXB2*.

Some of the major annotations are:

`XmlElement` describes the mapping between a field and a XML element.

`XmlType` describes the mapping between a class and a XML type.

`XmlAttribute` describes the mapping between a field and a XML attribute.

`XmlTransient` describes that a field should not be mapped.

**Schema Compiler** is responsible to generate an annotated Java object model for a given *W3C XML Schema* definition.

The following example shows a simple *W3C XML Schema* definition and how the resulting Java classes will look like. The sample schema definition contains two global element declarations without any named complex type definitions.

Listing 7.1: MusicService.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="artist">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="stage-name" type="xs:string"
7           minOccurs="0" />
8         <xs:element name="first-name" type="xs:string"
9           minOccurs="0" />
10        <xs:element name="last-name" type="xs:string"
11          minOccurs="0" />
12      </xs:sequence>
13    </xs:complexType>
14  </xs:element>
15
16  <xs:element name="label">
17    <xs:complexType>
18      <xs:sequence>
19        <xs:element name="name" type="xs:string" />
20      </xs:sequence>
21    </xs:complexType>
22  </xs:element>
23  ...
24 </xs:schema>
```

One resulting class is: `Artist.java`. It is generated with three attributes reflecting the three child elements of the schema definition. The three attributes are accessed directly and the XML name is specified.

Listing 7.2: Artist.java

```
1 @XmlAccessorType(XmlAccessType.FIELD)
2 @XmlType(name = "", propOrder = {
3     "stageName",
4     "firstName",
5     "lastName"
6 })
7 @XmlRootElement(name = "artist")
8 public class Artist {
9
10     @XmlElement(name = "stage-name")
11     protected String stageName;
12     @XmlElement(name = "first-name")
13     protected String firstName;
14     @XmlElement(name = "last-name")
15     protected String lastName;
16
17     public String getStageName() {
18         return stageName;
19     }
20
21     public void setStageName(String value) {
22         this.stageName = value;
23     }
24
25     public String getFirstName() {
26         return firstName;
27     }
28
29     public void setFirstName(String value) {
30         this.firstName = value;
31     }
32
33     public String getLastName() {
34         return lastName;
35     }
36
37     public void setLastName(String value) {
38         this.lastName = value;
39     }
40 }
```

## 7 Castor JAXB2 implementation

The other generated class is: `Label.java`. It reflects the global element `label` with the child element name.

Listing 7.3: `Label.java`

```
1 @XmlAccessorType(XmlAccessType.FIELD)
2 @XmlType(name = "", propOrder = {
3     "name"
4 })
5 @XmlRootElement(name = "label")
6 public class Label {
7
8     @XmlElement(required = true)
9     protected String name;
10
11     public String getName() {
12         return name;
13     }
14
15     public void setName(String value) {
16         this.name = value;
17     }
18 }
```

The samples have been generated using the JAXB2 reference implementation from Sun – for details about this implementation see: [\[jax08a\]](#).

**Schema Generator** Responsible to generate a *W3C XML Schema* definition for a given annotated Java object model. This step can be executed at compile time or at runtime.

**Binding Framework** The runtime API to transfer Java objects into XML (marshaling) and XML into Java objects (unmarshaling). The API is kept abstract to allow the transformations without knowledge about the underlying processing steps and technology.

This example shows an annotated class ...

Listing 7.4: `Artist.java`

```
1 @XmlAccessorType(XmlAccessType.FIELD)
2 @XmlType(name = "", propOrder = {
3     "stageName",
4     "firstName",
5     "lastName"
6 })
7 @XmlRootElement(name = "artist")
```

```

8 public class Artist {
9
10     @XmlElement(name = "stage-name")
11     protected String stageName;
12     @XmlElement(name = "first-name")
13     protected String firstName;
14     @XmlElement(name = "last-name")
15     protected String lastName;
16
17     public String getStageName() {
18         return stageName;
19     }
20
21     public void setStageName(String value) {
22         this.stageName = value;
23     }
24
25     public String getFirstName() {
26         return firstName;
27     }
28
29     public void setFirstName(String value) {
30         this.firstName = value;
31     }
32
33     public String getLastName() {
34         return lastName;
35     }
36
37     public void setLastName(String value) {
38         this.lastName = value;
39     }
40 }

```

...together with the following code fragment which creates an `Artist` instance, fills it with values and then uses JAXB API to marshal it into ...

Listing 7.5: ArtistToXmlSample.java

```

1 javax.xml.bind.JAXBContext context = JAXBContext.newInstance
   (Artist.class);
2 javax.xml.bind.Marshaller marshaller = context.
   createMarshaller();
3
4 StringWriter out = new StringWriter();
5
6 Artist a = new Artist();

```

## 7 *Castor* JAXB2 implementation

```
7 a.setStageName("Falco");
8 a.setFirstName("Hans");
9 a.setLastName("Hoelzel");
10
11 marshaller.marshal(a, out);
12
13 System.out.println(out.toString());
```

... the following *XML Document* ...

Listing 7.6: artist.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <artist>
3   <stage-name>Falco</stage-name>
4   <first-name>Hans</first-name>
5   <last-name>Hoelzel</last-name>
6 </artist>
```

## 7.2 Analysis of JAXB2 and Castor

An in depth look at *JAXB2* and *Castor* reveals what is required to achieve compliance and what *Castor* provides. This analysis is created based on the following releases:

- JSR 222: Java Architecture for XML Binding (JAXB) 2.0 – Final Release at May 11, 2006 – [jsr06]
- JAXB reference implementation from Sun – [SV06]
- Castor 1.1 – [cas08]

### 7.2.1 Coverage of W3C XML Schema

**JAXB standard** Full compliance to the *W3C XML Schema* definitions was a major goal of the JSR working group which created the *JAXB2* standard. So it is assumed that all definitions are possible. The *JAXB2 Reference Implementation* has been tested to clarify details of the standard.

**Castor 1.1** Also *Castor* claims to be fully *W3C XML Schema* compliant – tests which had been developed in the practical part of the thesis have shown some bugs that are reported and most of them are already fixed.

## 7.2.2 General differences

### Java compliance

**JAXB standard** JAXB2 is meant to support Java 5 and above. From Java 6 on it is even part of Java Standard Edition (JSE).

**Castor 1.1** Castor release 1.1.x and 1.2.x are still developed to be Java 1.4 compliant.

## 7.2.3 Mapping specification

**JAXB standard** *JAXB2* holds the mapping definition (mapping between Java and XML) inside the Java classes in form of *JAXB2 Annotations*. The annotations are specified at:

- Package level annotations are found in a special file `package-info.java` and define things like e.g. the namespace a package is mapped to.
- Class level annotations define e.g. to which XML type a certain class corresponds.
- Fields are the state holders of classes and are mapped to child elements, attributes or element content.
- Methods that capsule the access of a field – annotating a method is mostly equal to annotating the field itself. One exception is e.g. `XmlTransient` which marks a method to be ignored in marshaling, unmarshaling and schema creation.

**Castor 1.1** Castor supports three ways to define the mapping information for a class:

- By *Descriptor Classes* which are Castor specific classes derived from `XMLClassDescriptor` that hold the complete mapping information for one class of the business model. *Descriptor classes* are generated by the *Castor Code Generator* when a *W3C XML Schema Definition* is translated into Java classes. *Castor Code Generator* creates Java classes for element and type definitions and for each class generated also a corresponding *Descriptor Class* is generated – e.g. For the element definition: `<xs:element name="ab" >...</xs:element>` the classes `Ab` and `AbDescriptor` are generated.
- A mapping file holding all Java to XML mapping information – this is a *Castor* specific mapping description format in XML.
- By reflection – *Castor* determines all *mappable* fields and methods of a class by reflection and uses default mechanisms to map Java types and names into XML.

## 7 Castor JAXB2 implementation

The approach *by reflection* is the closest to the *JAXB2* approach but it doesn't read annotation information and allows only very simple XML structures (e.g. only child elements are used but no attributes).

**The *JAXB2* Annotations are:**

### **XmlAccessorType**

**JAXB standard** Can be used on package or class level. Controls the order of child elements in the generated schema definition.

**Castor 1.1** *Castor* doesn't have a feature like it but it is possible to create the mapping informations in a way to influence the element order.

### **XmlAccessorType**

**JAXB standard** Defines which class fields should be processed at marshaling, unmarshaling and schema generation. Four values are possible:

**FIELD** All class fields are taken; no annotation of the fields is required; no access methods (get/set) are required; visibility is ignored

**NONE** No class fields are taken; only exception are class fields which are explicitly annotated with e.g. `XmlElement`

**PROPERTY** All class fields that have access methods (get/set) are taken regardless of the visibility of the field and access method

**PUBLIC MEMBER** All class fields with public access are taken.

**Castor 1.1** Castor can work only with class fields that have public access methods (get/set).

### **XmlAnyAttribute**

**JAXB standard** A field annotated with `XmlAnyAttribute` has to be of type `java.util.Map<QName, Object>` and will be filled with all attributes found at the place of the `xs:anyAttribute`. At schema creation an `xs:anyAttribute` is inserted.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlAnyElement**

**JAXB standard** A field annotated with `XmlAnyElement` has to be of type `org.w3c.dom.Element` or `java.util.List<org.w3c.dom.Element>` and will be filled with all elements found at the place or the `xs:any`.

**Castor 1.1** Is supported with a *Castor* specific style of DOM nodes that are returned.

### **XmlAttachmentRef**

**JAXB standard** To handle external MIME compatible binary attachments.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlAttribute**

**JAXB standard** To bind a Java field as a XML attribute. Name and namespace of the attribute can be given in the annotation.

**Castor 1.1** Supported in *Castor* in a similar way.

### **XmlElement**

**JAXB standard** To bind a Java field as a XML element. Name, namespace, default value and other information can be defined in the annotation.

**Castor 1.1** Supported in *Castor* in a similar way.

### **XmlElementDecl**

**JAXB standard** Only occurs in front of the factory methods in the `ObjectFactory` class. JAXB creates Java classes for XML types but not for elements – for elements only factory methods within the `ObjectFactory` class are created. The `XmlElementDecl` annotation adds the XML specific information (e.g. name, namespace, default value) for each XML element a factory method reflects.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlElementRef**

**JAXB standard** To bind a Java field as a XML element. But in case of `XmlElementRef` the element declaration is by reference (e.g. `<xs:element ref="hugo" />`).

**Castor 1.1** Supported in *Castor* in a similar way.

## XmlElementRefs

**JAXB standard** To bind a Java field to multiple XML elements that are declared by reference. This is the case when:

- The Java field is a `java.util.List` or `java.util.Set`
- The XML declaration allows multiple elements in undefined (zero or many) multiplicity.

Listing 7.7: ForXmlElements.xsd

```
1 <xs:element name="for-xml-elements">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:choice
5         minOccurs="0"
6         maxOccurs="unbounded">
7         <xs:element
8           ref="ref-one" />
9         <xs:element
10          ref="ref-two" />
11       </xs:choice>
12     </xs:sequence>
13   </xs:complexType>
14 </xs:element>
```

**Castor 1.1** Nothing similar available in *Castor*.

## XmlElementRefs

**JAXB standard** To bind a Java field to multiple XML elements. This is the case when:

- The Java field is a `java.util.List` or `java.util.Set`
- The XML declaration allows multiple elements in undefined (zero or many) multiplicity.

Listing 7.8: ForXmlElements.xsd

```
1 <xs:element name="for-xml-elements">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:choice
5         minOccurs="0"
6         maxOccurs="unbounded">
7         <xs:element
8           name="choice-one"
```

```

9         type="xs:string" />
10        <xs:element
11            name="choice-two"
12            type="xs:string" />
13        </xs:choice>
14    </xs:sequence>
15 </xs:complexType>
16 </xs:element>

```

**Castor 1.1** Nothing similar available in *Castor*.

### XmlElementWrapper

**JAXB standard** The annotation `XmlElementWrapper` defines an element which is used as parent element for the list of elements that are generated for the field values. This implicates that the field that can be annotated is either `java.util.List` or `java.util.Set` and the values of the list are them self translated into XML elements. The desired XML structure will look like:

Listing 7.9: WrapperSample.xml

```

1 <the-wrapper>
2   <inside>1</inside>
3   <inside>2</inside>
4   <inside>3</inside>
5 </the-wrapper>

```

**Castor 1.1** Castor has a different style how this is supported.

### XmlEnum

**JAXB standard** Used to bind a Java `enum` to a XML enumeration. Values within the Java `enum` can be mapped using `XmlEnumValue`.

**Castor 1.1** Supported in *Castor* in a similar way.

### XmlEnumValue

**JAXB standard** Used to map a Java `enum` value to a corresponding XML enumeration value.

**Castor 1.1** Supported in *Castor* in a similar way.

## 7 *Castor* JAXB2 implementation

### **XmlID**

**JAXB standard** To map a Java field as `xs:ID`.

**Castor 1.1** Supported in *Castor* in a similar way.

### **XmlIDREF**

**JAXB standard** To map a Java field as `xs:IDREF`.

**Castor 1.1** Supported in *Castor* in a similar way.

### **XmlInlineBinaryData**

**JAXB standard** To disable consideration of XOP encoding for datatypes that are bound to base64-encoded binary data in XML.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlJavaTypeAdapter**

**JAXB standard** To introduce one `XmlAdapter` implementation.

**Castor 1.1** *Castor* has a different approach. In *Castor* specific un-/marshalers can be implemented and bound to a Java type – these are called *Custom Handlers* in *Castor*.

### **XmlJavaTypeAdapters**

**JAXB standard** To introduce multiple `XmlAdapter` implementations.

**Castor 1.1** See `XmlJavaTypeAdapter`.

### **XmlMimeType**

**JAXB standard** To specify a *MIME type* for a binary content Java field.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlMixed**

**JAXB standard** Used when a complex type allows mixed content – meaning to have child elements, attributes and content within one element. `XmlMixed` can be used when the annotated Java field is of type `java.util.List` or `Object[]`. The Java field is filled with entries of:

- `java.lang.String` for content particles
- Instances of `JAXBElement` or `XmlRootElement` annotated classes for recognized elements
- Or `org.w3c.dom.Element` entries for not recognized elements

The entries occur in the same order as they occurred in the XML structure.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlNs**

**JAXB standard** To assign a namespace prefixes with a namespace URI.

**Castor 1.1** Supported in *Castor* in a similar way.

### **XmlRegistry**

**JAXB standard** To use another class for the purpose of `ObjectFactory` it needs to be annotated with `XmlRegistry`.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlRootElement**

**JAXB standard** A class annotated with `XmlRootElement` represents a XML root element – usually the generated Java classes represent XML types but not XML elements.

**Castor 1.1** Castor allows to use every class with a known mapping as root class.

### **XmlSchema**

**JAXB standard** Can only be used for packages and defines the schema parameters like the list of namespace prefix and URI.

**Castor 1.1** Nothing similar available in *Castor*.

### **XmlSchemaType**

**JAXB standard** All *W3C XML Schema* built in types have a default mapping to Java classes – using `XmlSchemaType` one schema type can be explicitly mapped to a Java class. This mapping can be specified either for a package or only for a certain field.

**Castor 1.1** Not supported in this style – the *Castor* solution would again be *Custom Handlers*.

### **XmlSchemaTypes**

**JAXB standard** To bundle multiple `XmlSchemaType` annotations together.

**Castor 1.1** See `XmlSchemaType`.

### **XmlTransient**

**JAXB standard** A field or method annotated with `XmlTransient` should not be considered for marshaling, unmarshaling and schema generation.

**Castor 1.1** Castor can ignore attributes – simply create no mapping description for it.

### **XmlType**

**JAXB standard** Used to map a class to an XML type. With the annotation the corresponding XML type name and namespace, the order of the properties and the factory class and method can be specified.

**Castor 1.1** Castor creates classes for XML types, a specific order of the properties cannot be given, factory class and method are not supported.

### **XmlValue**

**JAXB standard** An attribute (field or method) annotated with `XmlValue` is meant to hold the content of a `xs:simpleContent`.

**Castor 1.1** Nothing similar available in *Castor*.

## **7.2.4 Binding API**

### **JAXBContext**

**JAXB standard** The role of `JAXBContext` is as single entry point of JAXB API. It is created for a set of classes or a set of packages and can then be used to create initialized instances of `Marshaller`, `Unmarshaller`, `JAXBIntrospector` for further processing. Initialized means that all mapping information is parsed and available.

**Castor 1.1** Castor has no such central context holding the parsed mapping information. Also there is no central factory for `Marshaller`, `Unmarshaller` and such.

## Marshaller

**JAXB standard** Main purpose is to provide various `marshal` methods to transform class instances into a *XML Document*. Two parameters are expected by the marshaling methods: the Java object to marshal and an output option. The Java object has to be either annotated as `XmlRootElement` or be wrapped in a `JAXBElement`. The output options available are: `Result`, `OutputStream`, `File`, `Writer`, `ContentHandler`, `Node`, `XMLStreamWriter` and `XMLEventWriter`.

**Castor 1.1** *Castor* provides a similar set of marshaling methods. But the Java object to marshal can be *any* object and the output options `XMLStreamWriter` and `XMLEventWriter` are missing.

## Unmarshaller

**JAXB standard** Main purpose is to provide various `unmarshal` methods to instantiate class instances from an *XML Document*. The `unmarshal` methods take one parameter: the input source and returns the resulting Java object. The supported input sources are: `File`, `InputStream`, `Reader`, `URL`, `InputStream`, `Node`, `Source`, `XMLStreamReader` and `XMLEventReader`. The resulting Java object is either a class annotated as `XmlRootElement` or wrapped into a `JAXBElement`.

**Castor 1.1** *Castor* supports a similar set of unmarshaling methods. But not exactly the same input sources and it doesn't have the *root element* mechanism as *JAXB2* has it.

## JAXBIntrospector

**JAXB standard** A helper class to work with the binding information that resides in the `JAXBContext`.

**Castor 1.1** Nothing similar available in *Castor*.

## XmlAdapter

**JAXB standard** To allow custom marshaling for any Java type a specific `XmlAdapter` can be implemented and registered.

**Castor 1.1** To support user defined un-/marshaling of XML element content (or attribute values) *Castor* users implement an own *custom field handler* (`org.exolab.castor.mapping.FieldHandler`) that performs the specific un-/marshaling. The custom field handler can currently only be used in combination with a mapping file.

### **AttachmentMarshaller**

**JAXB standard** If, at marshaling, binary data is handled via attachments this interface has to be implemented by the user of the JAXB API.

**Castor 1.1** Nothing similar available in *Castor*.

### **ValidationEventHandler**

**JAXB standard** Event callback interface to implement by the JAXB user to receiver validation events during marshal, unmarshal and validation.

**Castor 1.1** Nothing similar available in *Castor*.

### **Marshaller.Listener**

**JAXB standard** An interface which can be implemented by the JAXB user to listen for marshal callback events. The listener will receive a callback before and after every object that is marshaled. The API looks like:

Listing 7.10: Marshaller.java

```
1 package javax.xml.bind;
2 ...
3 public interface Marshaller {
4     ...
5     public static abstract class Listener {
6         public void beforeMarshal(Object source) {}
7         public void afterMarshal(Object source) {}
8     }
9 }
```

**Castor 1.1** Castor has a similar interface:

Listing 7.11: MarshalListener.java

```
1 package org.castor.xml;
2 ...
3 public interface MarshalListener {
4     boolean preMarshal(Object object);
5     void postMarshal(Object object);
6 }
```

**AttachmentUnmarshaller**

**JAXB standard** If, at unmarshaling, binary data is handled via attachments this interface has to be implemented by the user of the JAXB API.

**Castor 1.1** Nothing similar available in *Castor*.

**Unmarshaller.Listener**

**JAXB standard** An interface which can be implemented by the JAXB user to listen for unmarshal callback events. The listener will receive a callback before and after the object instance will be filled with the result of the XML unmarshaling. The API looks like:

Listing 7.12: Unmarshaller.java

```

1 package javax.xml.bind;
2 ...
3 public interface Unmarshaller {
4     ...
5     public static abstract class Listener {
6         public void beforeUnmarshal(Object target, Object parent
7             ) {}
8         public void afterUnmarshal(Object target, Object parent)
9             {}
10    }
11 }

```

**Castor 1.1** Castor has a similar interface which provides more callback methods but can be mapped:

Listing 7.13: UnmarshalListener.java

```

1 package org.castor.xml;
2 ...
3 public interface UnmarshalListener {
4     void initialized (final Object target, final Object parent
5         );
6     void attributesProcessed(final Object target, final Object
7         parent);
8     void fieldAdded (String fieldName, Object parent, Object
9         child);
10    void unmarshalled (final Object target, final Object
11        parent);
12 }

```

## 7 Castor JAXB2 implementation

### JAXBElement

**JAXB standard** Wrapper for classes which aren't annotated as `XmlRootElement` but still should be marshaled and unmarshaled by them self.

**Castor 1.1** Nothing similar available in *Castor*.

### 7.2.5 Schema compiler

**JAXB standard** The *JAXB Schema Compiler* is responsible to read a *W3C XML Schema Definition* and create annotated Java classes for the XML types identified. The class creation can be controlled (influenced) by using an own *JAXB binding configuration* that can be either part of the input schema definition file or in a separate file.

**Castor 1.1** *Castor* has a similar tool called *Castor Code Generator*. It creates Java class files in a *Castor* specific way with *Castor Descriptor Classes* and supports modification through a *Castor* specific binding file. *JAXB2* annotations are, of course, not created.

### 7.2.6 Schema generator

**JAXB standard** *JAXB2* allows to create a *W3C XML Schema Definition* from the information read from the annotated Java classes.

**Castor 1.1** *Castor* has an API to read and write *W3C XML Schema Definitions* but has no feature to create such a definition from the mapping information of the Java classes.

## 7.3 Castor JAXB2 Implementation

The approach of the *Castor JAXB2 Implementation* is to built a layer on top of existing *Castor* implementation. The architecture overview of the *Castor JAXB2 Implementation* can be seen in Figure 7.2.

The *Castor JAXB2 Implementation* components are:

- The *Binding API Implementation* as specified in the *JAXB2 Standard* and implementing the API provided by the *JAXB2 Binding API* package – mostly communicates with *Castor XML*
- The *Schema Compiler API* – mostly using *Castor XML Code Generator*

## 7.3 Castor JAXB2 Implementation

- The *Schema Generator API* – mostly using a new *Castor* component: *Castor Schema Generator*
- The *JAXB2 Annotation Interpreter* is a plugin that is called by *Castor XML* to determine the mapping definition for a class
- The *Castor Schema Generator Extensions* is a plugin to extend the *Castor Schema Generator* for *JAXB2*
- The *Castor XML Code Generator Extension* is a plugin to extend the *Castor XML Code Generator* for *JAXB2*
- The *Support for JAXB2 Binding* provides the interpretation and creation of the *JAXB2* binding schema

### 7.3.1 Major JAXB2 execution flows

Most of the *JAXB2* functionality is covered by four major flows – these are:

Marshaling – Showing the flow required to marshal annotated classes into a *XML Document* – Figure 7.3

Unmarshaling – Showing the flow required to unmarshal annotated classes into a *XML Document* – Figure 7.4

Java Code Generation – Showing the flow required to generate Java classes from a *W3C XML Schema Definition (XML Application)* – Figure 7.5

W3C XML Schema Generation – Showing the flow required to create a *XML Application* from annotated classes – Figure 7.6

### 7.3.2 The issues to achieve the Castor JAXB2 Implementation are

#### Introduce XMLContext

*Description* In *JAXB2* the *JAXBContext* is the central entity providing configuration and mapping information to the framework (*Marshaler*, *Unmarshaler* and other parts).

*Solution* This issue focuses to introduce a similar entity, called *XMLContext*, into *Castor*. It should hold and provide the configuration and mapping information for *Castor*.

*Status* The issue is solved and will be part of the upcoming release 1.3.

## 7 Castor JAXB2 implementation

### Refactor descriptor resolution

*Description* *Castor* has various methods how the mapping of a class is determined. For the *Castor JAXB2 Implementation* an additional method is required: Reflection together with *JAXB2 Annotation* evaluation. This method cannot be implemented inside *Castor* which is still Java 1.4 compliant.

*Solution* The solution is to refactor the existing mapping resolution implementation to allow plugins and to implement the *JAXB2* specific resolution as a plugin.

*Status* The issue is solved and will be part of the upcoming release 1.3.

### Refactor Introspector

*Description* To partially reuse the mapping resolution based on reflection of *Castor* it is required to massively refactor the responsible module.

*Status* The issue is solved and will be part of the upcoming release 1.3.

### Refactor Java naming and XML naming

*Description* *Castor* has multiple source code fragments that implement naming conversion mechanisms and some of them are not exchangeable as they are implemented as utility classes with static methods only.

*Solution* The solution is to refactor the naming implementation to be exchangeable by configuration and in addition bring together the different implementations.

*Status* The issue is solved and will be part of the upcoming release 1.3.

### Fix support of xs:list

*Description* The transformation of a `Collection` into a XML Element that is of type `xs:list` was wrong.

*Solution* -

*Status* The issue is solved and will be part of the upcoming release 1.3.

### Extend Unmarshalllistener to send parent also

*Description* The *Castor Unmarshalllistener* did not include the parent object into the callback events.

*Solution* Extend the *Unmarshalllistener* (but guarantee that implementations of the previous interface will still be compileable and functional)

## 7.3 Castor JAXB2 Implementation

*Status* The issue is solved and will be part of the upcoming release 1.3.

### **Refactor xsi:type handling**

*Description* The `xsi:type` handling during marshaling is inconsistent and depends on wrong control mechanisms.

*Solution* Rewrite the marshaling.

*Status* The issue is in progress.

### **Support class-level marshal and unmarshal callbacks**

*Description* It is required (in the *JAXB2 Standard*) that marshal and unmarshal callback methods on each class are called if available. Such a feature is not available in *Castor*.

*Solution* Extend the classes that hold the mapping information to manage in addition if a class provides these callback methods. Extend the marshaling and unmarshaling implementation to call the class level callback methods.

*Status* The issue is in progress.

### **Implement JAXB2 Annotation processing**

*Description* Implement the evaluation of the *JAXB2 Annotations*.

*Solution* Implement evaluation of *JAXB2 Annotations* and map the result into the *Castor* mapping descriptor classes.

*Status* The issue is solved and is part of the *Castor JAXB2 Module* sources.

### **Implement Binding API 1**

*Description* In the first step the following parts of the *JAXB2 Binding API* are provided: `JAXBContext`, `Marshaller`, `Unmarshaller`, `JAXBIntrospector`, `Marshaller.Listener` and `Unmarshaller.Listener`.

*Status* The issue is solved and is part of the *Castor JAXB2 Module* sources.

### **Implement Castor Schema Generator**

*Description* Implement the possibility to generate *W3C XML Schema Definition* files from the mapping definition.

*Status* The issue is in progress.

## 7 Castor JAXB2 implementation

### **Implement JAXB2 Schema Generator**

*Description* Implement the *JAXB2* specific flavor including support of the *JAXB2* binding description extensions.

*Status* The issue is in progress.

### **Extend Castor Code Generator**

*Description* Implement support for Java 5 annotations, introduce templates for the code generated and introduce plugins to make the generation process extendable.

*Solution* -

*Status* The issue is solved and will be part of the upcoming release 1.3.

### **Implement JAXB2 Schema Compiler**

*Description* Implement all *JAXB2 Annotations*, the *JAXB2* specific naming conversion rules and other code generation specifics of *JAXB2*. Also the evaluation of the *JAXB2* binding description needs to be supported.

*Status* The issue is in progress.

### **Implement SchemaType feature**

*Description* Implement the support for the `XmlSchemaType` annotation.

*Status* The issue is open.

### **Implement XmlAdapter feature**

*Description* Implement the support for the `XmlJavaTypeAdapter` annotation.

*Status* The issue is open.

### **Implement JAXB2 root element handling**

*Description* *JAXB2* allows only such Java classes to be un-/marshaled as root element that are marked with `XmlRootElement`. In addition the classes `ObjectFactory` and `JAXBElement` and the annotations `XmlRegistry` need to be supported.

*Solution* *Castor* mapping information has to be extended and the marshaling and unmarshaling routines need to be adapted.

*Status* The issue is open.

**Implement attachment and binary data handling**

*Description* JAXB2 implements attachment handling and inline binary data handling through two annotations (`XmlAttachmentRef` and `XmlInlineBinaryData`) and additional API classes (`AttachmentMarshaller` and `AttachmentUnmarshaller`).

*Solution* Castor needs to introduce this functionality.

*Status* The issue is open.

**Support mixed element**

*Description* Castor needs to support elements that have mixed content.

*Status* The issue is open.

## 7.4 Critical Discussion

Full *JAXB2* compliance for *Castor* was not reached during the practical part of this thesis.

The reasons are:

- *Castor framework* has to provide back- and forward compatibility for their users – at least generated sources and implementations based on the existing API need to be useable without adaptations
- Modifications like: introduction of a central context, root element handling, introduction of `XmlAdapters` and more are to complex when sticking to back- and forward compatibility
- *Castor XML Code Generator* required complex extensions
- Interpretation of *JAXB2* binding specification is new to *Castor*
- *Schema Generation* was not existing in *Castor*

The resulting implementation of this thesis provides the part of the *JAXB2 Binding API* to do marshaling and unmarshaling, support for *JAXB2 Annotations* to control *Castor* mapping descriptors and *JAXB2* specific code generation.

The problems at reaching compliance started another discussion:

Is full *JAXB2* compliance required for *Castor*? And will *Castor* be more attractive to users by just providing *JAXB2* compliance? Or is it sufficient to provide compatibility at *Binding API* and *JAXB2 Annotations*?

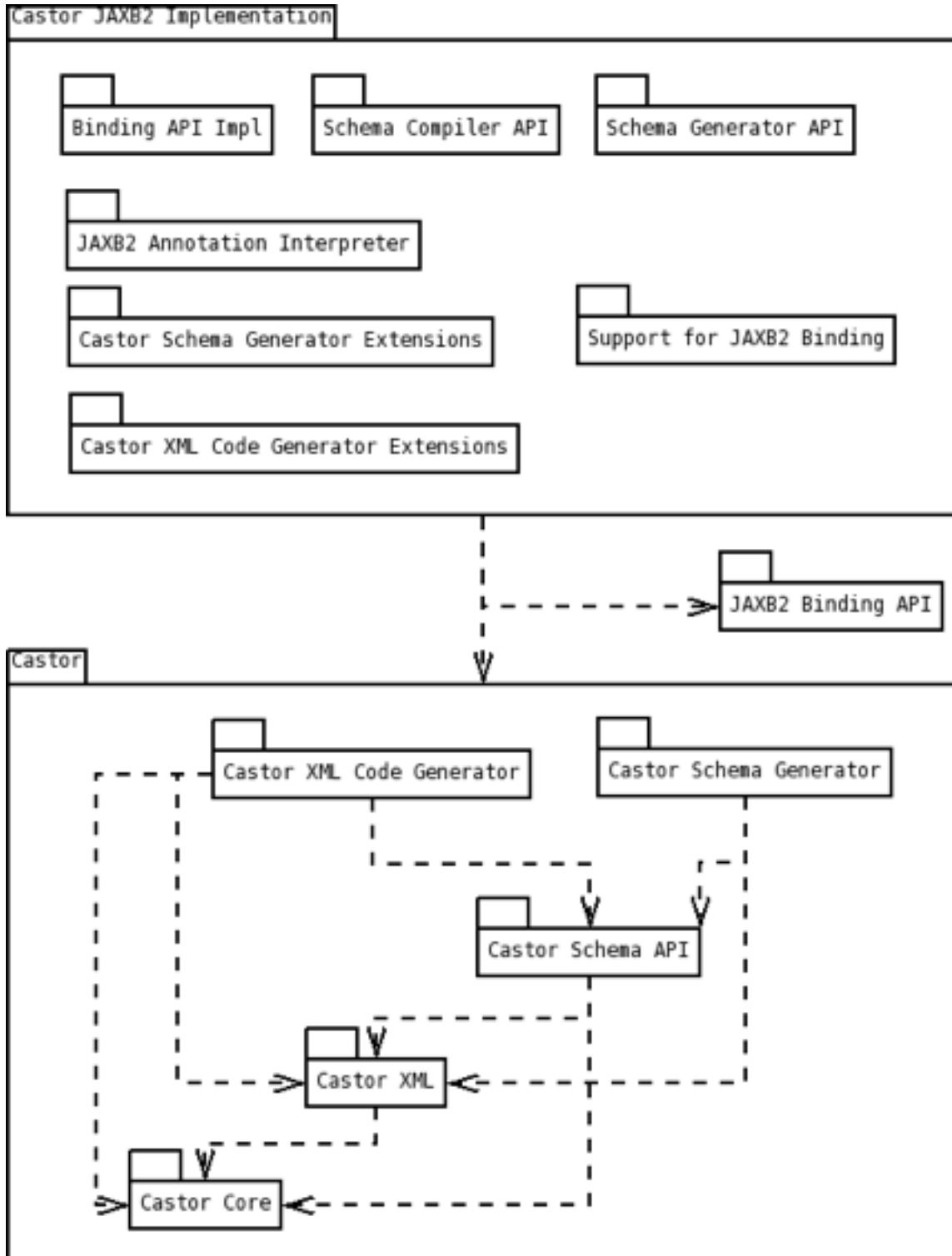


Figure 7.2: Castor JAXB2 Architecture Overview

## 7 Castor JAXB2 implementation

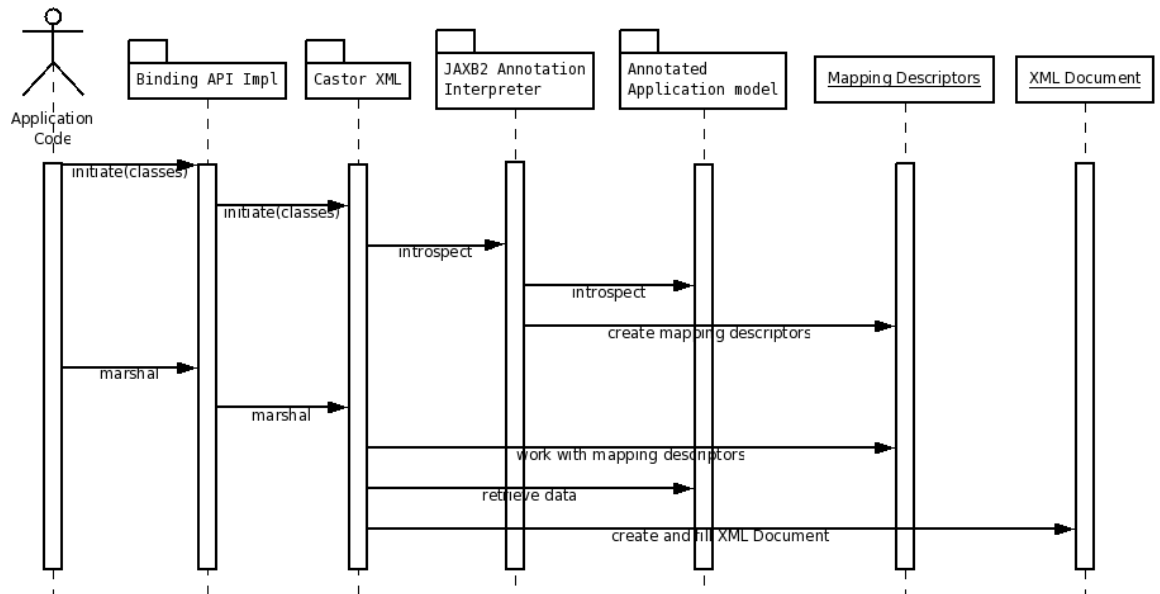


Figure 7.3: Castor JAXB2 Flow for Marshaling

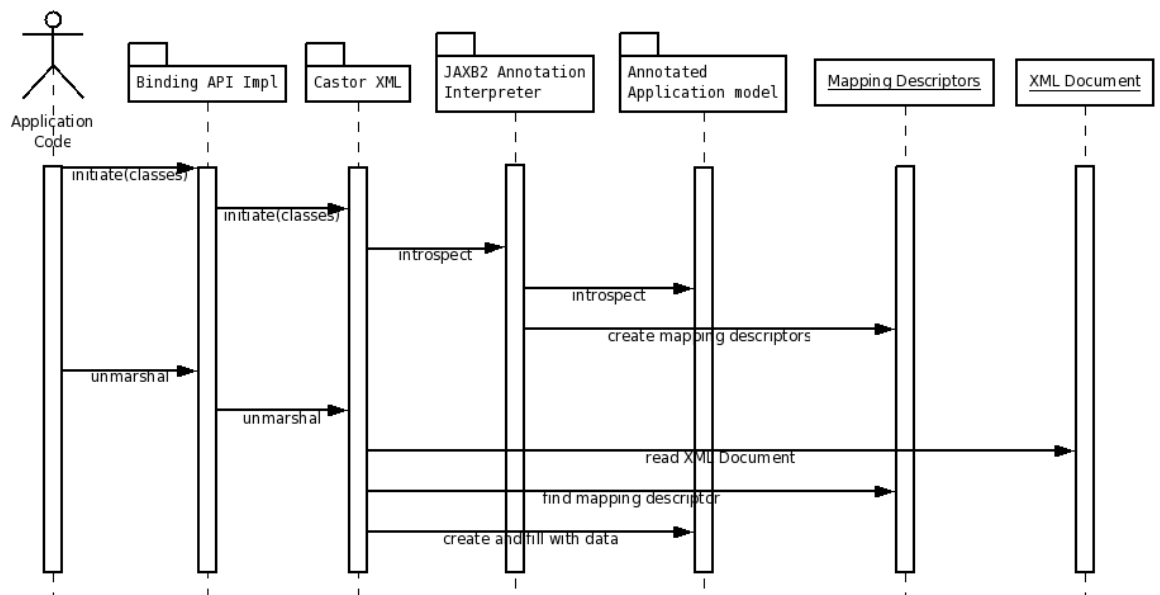


Figure 7.4: Castor JAXB2 Flow for Unmarshaling

7.4 Critical Discussion

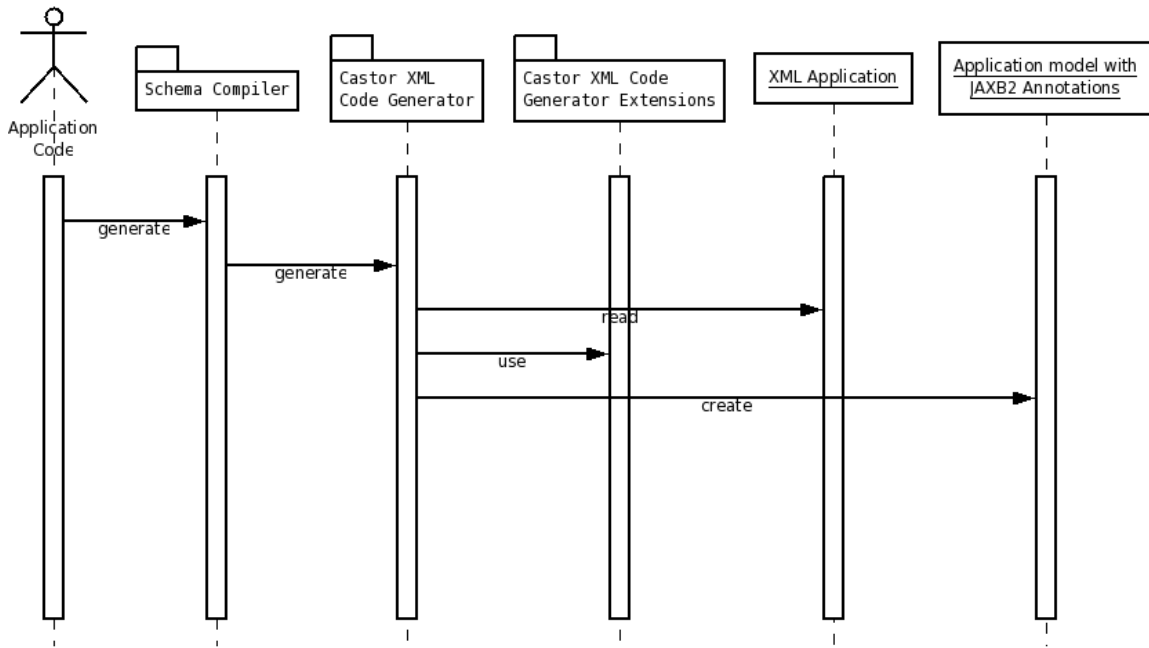


Figure 7.5: Castor JAXB2 Flow for Java Code Generation

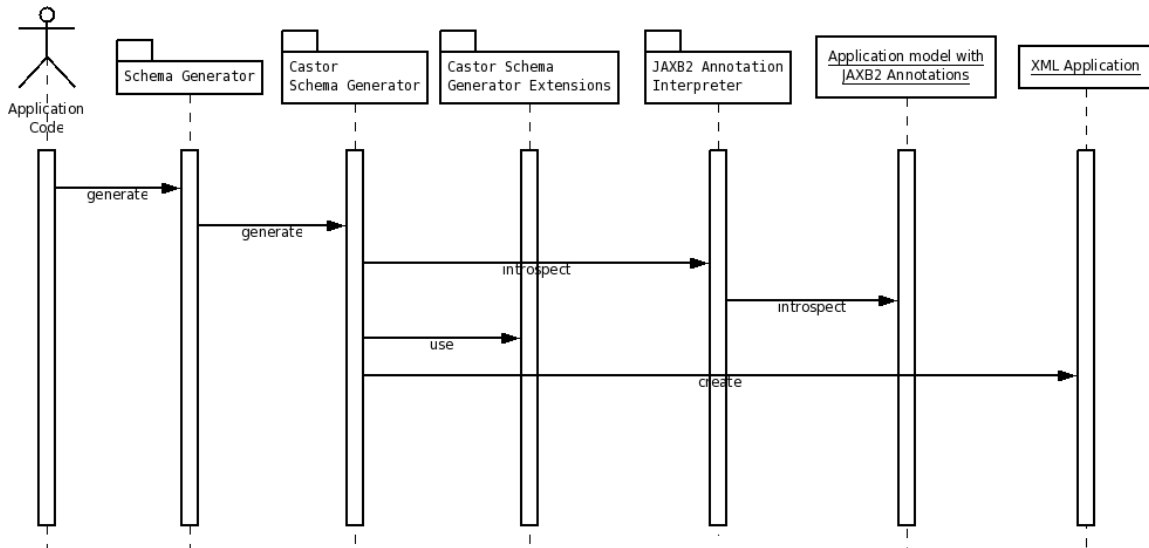


Figure 7.6: Castor JAXB2 Flow for W3C XML Schema Generation

## 7 *Castor JAXB2 implementation*

## 8 Conclusion

The motivation of the thesis was to enhance existing *Castor* to reach full *JAXB2* compliance and to explore *Object-XML Mapping* in general.

### 8.1 XML

XML is a standard (recommendation) released in 1998 by *World Wide Web Consortium*. It defines a meta format for data that was built on the basis of *Semi Structured Data* and *Markup Languages*. The idea of XML is that for each application an own derivative is created – a so called *XML Application*. Documents that accord to the rules of an *XML Application* are so called *well-formed and valid XML Documents*.

The two most popular languages to write *XML Applications* are *Document Type Definition (DTD)* and *W3C XML Schema*. For usages that include mapping to Java *W3C XML Schema* is the most popular – this originates in the fact that the *WebService* related standards all use *W3C XML Schema* for format description. The usage of XML in general ranges from *Narrative Documents*, *Configuration Files*, *Programming Languages* to *Record Like Data*.

### 8.2 Java

*Object Oriented Programming Languages* date back to 1967 with *Simula* as first language having features of object orientated languages. Java, as one object oriented language, was released in 1995 by Sun Microsystems. The basic building blocks of Java are `class`, `field` and `method`. Fields hold the state of a class and methods allow access and modification of the class state. In addition Java also supports `interfaces`, *abstract* classes and the control of class, field and method *visibility*. Java follows the object oriented principles like *encapsulation*, *abstraction*, *inheritance* and *polymorphism*. To allow the representation of complex data models Java provides a rich library of base classes like: `Map` to store lists of name, value pairs, `ArrayList` to allow a list of values that keep the order of insertion, `HashTree` representing a simple tree and many more. Beyond standard libraries also collections of well documented

## 8 Conclusion

*Patterns* can be found explaining best practice object oriented solutions.

### 8.3 Mapping

Mapping is where *Object Oriented Programming Languages* (Java in this case) and XML meet. Mapping between Java and XML can be done without business domain knowledge leading to an object model that only represents the XML structure – a tree of elements, attributes and content. To use the result of this kind of mapping in a business application a subsequent (additional) step is required which maps from the XML tree model to the business model.

The other mapping approach is with business domain knowledge. This approach is also called *O/X Mapping*. It starts with the business model as input – either as *W3C XML Schema*, Java model or both – and provides one step transformation between the business specific Java object model and the business specific *XML Document*.

*Castor* and *JAXB2 Standard* focus both on *O/X Mapping*.

### 8.4 Castor

*Castor* an open source framework providing Object-XML and Object-relational database mapping. It dates back to 1999 and is still enhanced and maintained. The XML mapping part follows no standard but provides all features required for *O/X Mapping*:

- Flexible mapping configuration through either a mapping file, *Castor* specific mapping descriptor classes or ad hoc introspection
- Marshaling of Java object graphs into *XML Documents*
- Unmarshaling of *XML Documents* into Java object graphs
- Java source generation out of *W3C XML Schema Definitions*
- Support to read and write *W3C XML Schema Definitions*

Within the practical part of this thesis *Castor* was challenged against the *W3C XML Schema* definitions found in Chapter 4.2.2. Some bugs had been found, reported and fixed in *Castor*.

## 8.5 JAXB2 Standard

*JAXB2 Standard* is the only *O/X Mapping* standard for Java. It was released at May 11, 2006 as a result of JSR 222. The main goals of the standard had been:

- Full support of Java 5
- Full support of *W3C XML Schema*

The standard specifies four major building blocks:

- The *JAXB2 Annotations* which are used to represent the description of the Java-XML mapping. All *W3C XML Schema* constructs are supported through annotations.
- The *Schema Compiler* which takes a *W3C XML Schema* definition as input and creates the according Java classes with *JAXB2 Annotations*.
- The *Schema Generator* allows to create a *W3C XML Schema* definition based on a set of Java classes and their *JAXB2 Annotations*.
- The *Binding API* which is meant to be used to write applications with. It provides access to marshal and unmarshal functionality and also supporting classes.

The list of *JAXB2 Annotations* was checked against the *W3C XML Schema* definitions found in Chapter 4.2.2 – all definitions can be covered. Also the *JAXB2 Reference Implementation* was tested with the set of *W3C XML Schema* definitions to analyze and compare e.g. the code generation results.

## 8.6 Comparison Castor and JAXB2 Standard

An immediate comparison of *Castor* and *JAXB2 Standard* shows many differences:

- Different *Binding API*
- Different approach how the mapping information needs to be provided
- Different Java structures for the same *W3C XML Schema* definition
- Different approach to influence the Java code generation from *W3C XML Schema* definitions
- *Castor* allowing *every* class as root element – *JAXB2* allowing only those classes which are specifically marked (`XmlRootElement` or `JAXBElement`)
- *Castor* validates the Java objects – *JAXB2* only validates the *XML Documents*

## 8 Conclusion

- *Castor* is still Java 1.4 compliant – *JAXB2* has a strong focus on Java 5 and is compliant with Java 5
- *JAXB2* also supports SOAP attachments
- Different approach to extend the marshaling and unmarshaling – at *Castor* through `FieldHandlers` – at *JAXB2* through `XmlAdapters`
- No *Schema Generator* in *Castor*

## 8.7 Implementation of Castor JAXB2

The implementation approach bases on the wish of the *Castor* community to preserve and use as much of the existing *Castor* implementation as possible and see the *Castor JAXB2 Implementation* as an add on that doesn't change *Castor* for existing users.

As a result the architecture of *Castor JAXB2 Implementation* consists of:

- A layer that provides *JAXB2* API and annotations at the outside and calls *Castor* at the inside.
- Some plugins that extend existing *Castor* functionality to support the new *JAXB2* flavor – e.g. the resolution of *JAXB2 Annotations* for mapping description is such a plugin.
- And finally extensions (e.g. enable plugins at mapping resolution) inside existing *Castor*.

## 8.8 Implementation Results

The accompanying implementation of *Castor JAXB2* succeeded in the following parts:

- Providing a first set of the *Binding API* consisting of e.g. `JAXBContext`, `Marshaller`, `Unmarshaller`, `Marshaller.Listener`, `Unmarshaller.Listener` and `JAXBIntrospector`.
- Interpretation of *JAXB2 Annotations* to use this mapping information when working with the *Castor* `Marshaller` and `Unmarshaller`.
- Extend *Castor XML Code Generator* to generate Java classes annotated with *JAXB2 Annotations*.
- A first version of *Schema Generator*

- Fixing some bugs found in the *W3C XML Schema* compliance check

The following issues prevented from reaching full *JAXB2* compliance:

- The number of features to implement was too large to finish it within the time constraints of this thesis.
- *Castor* already has a style of mapping and an API that is stable and in use. It cannot radically be changed without influencing existing users. E.g. introducing the *JAXB2* concept of classes for root elements would break all existing code that relies on the *Castor* feature that every class can be used to create a root element.

Neither the implementation of the full *JAXB2 Binding API* nor support for all *JAXB2 Annotations* was achieved.

## 8.9 Future Work

*Castor* will stay on track to implement *JAXB2* compliance – but it is no longer full compliance that is sought. But instead provide an easy way for users, having classes annotated with *JAXB2 Annotations* and implementations that are built on top of *JAXB2 Binding API*, to switch to *Castor*.

Providing just another *JAXB2* implementation is not enough – *Castor* has to differentiate itself from the *JAXB2 Reference Implementation*. Such features are:

- Supporting additional sources of mapping information like in a XML file instead of inside the Java sources only – thus allowing to map the same Java classes to different *XML Applications*.
- Generating not only Java classes but also validation code to support validation of the Java objects before marshaling.
- Allow close coupling of Java-XML and Java-database mapping – e.g. by allowing to provide the database mapping information in the binding definition of the *W3C XML Schema* used to generate the Java source base.

## 8 *Conclusion*

# A Appendix

## A.1 Document Object Model (DOM) Specifications

The DOM specifications available:

- DOM Level 1
  1. Document Object Model (DOM) Level 1 Specification – [[dom98](#)]
- DOM Level 2
  1. Document Object Model (DOM) Level 2 Core Specification – [[dom00a](#)]
  2. Document Object Model (DOM) Level 2 Events Specification – [[dom00b](#)]
  3. Document Object Model (DOM) Level 2 HTML Specification – [[dom03](#)]
  4. Document Object Model (DOM) Level 2 Style Specification – [[dom00c](#)]
  5. Document Object Model (DOM) Level 2 Traversal and Range Specification – [[dom00d](#)]
  6. Document Object Model (DOM) Level 2 Views Specification – [[dom00e](#)]
- DOM Level 3
  1. Document Object Model (DOM) Level 3 Abstract Schema Specification – [[dom02](#)]
  2. Document Object Model (DOM) Level 3 Core Specification – [[dom04a](#)]
  3. Document Object Model (DOM) Level 3 Events Specification – [[dom04b](#)]
  4. Document Object Model (DOM) Level 3 Load and Save Specification – [[dom04c](#)]
  5. Document Object Model (DOM) Level 3 Validation Specification – [[dom04d](#)]
  6. Document Object Model (DOM) Level 3 Views and Formatting Specification – [[dom04e](#)]
  7. Document Object Model (DOM) Level 3 XPath Specification – [[dom04f](#)]

## A.2 Java Specification Requests related to XML

The following list gives an overview about how many JSRs are currently directly related to XML. Of course there are many more that use XML for configuration or base on the JSRs mentioned in the list...

## A Appendix

To get the complete list of Java Specification Requests or to get details for the requests of the following list please visit: [\[jsr08\]](#).

**JSR 5 – XML Parsing Specification** The Java™ API for XML Parsing (JAXP) allows developers to easily use XML Parsers in their applications via the industry standard SAX and DOM APIs

**JSR 31 – XML Data Binding Specification** A facility for compiling an XML schema into one or more Java classes which can parse, generate, and validate documents that follow the schema.

**JSR 63 – Java™ API for XML Processing 1.1** The proposed specification will define a set of implementation independent portable APIs supporting XML Processing

**JSR 67 – Java™ APIs for XML Messaging 1.0** JAXM provides an API for packaging and transporting business transactions using on-the-wire protocols being defined by ebXML.org, Oasis, W3C and IETF.

**JSR 93 – Java™ API for XML Registries 1.0 (JAXR)** JAXR provides an API for a set of distributed Registry Services that enables business-to-business integration between business enterprises, using the protocols being defined by ebXML.org, Oasis, ISO 11179.

**JSR 101 – Java™ APIs for XML based RPC** Java APIs to support emerging industry XML based RPC standards

**JSR 102 – JDOM 1.0** JDOM is a way to represent an *XML Document* for easy and efficient reading, manipulation, and writing

**JSR 104 – XML Trust Service APIs** This defines a standard set of APIs and a protocol for a Trust Service, minimizing the complexity of applications using XML Signature.

**JSR 105 – XML Digital Signature APIs** This defines and incorporates a standard set of high-level implementation-independent APIs for XML digital signatures services. The XML Digital Signature specification is defined by the W3C.

**JSR 106 – XML Digital Encryption APIs** This JSR is to define a standard set of APIs for XML digital encryption services. This proposal is to define and incorporate the high level implementation independent Java APIs.

**JSR 110 – Java™ APIs for WSDL** JWSDL provides a standard set of Java APIs for representing, manipulating, reading and writing WSDL (Web Services Description Language) documents, including an extension mechanism for WSDL extensibility.

## A.2 Java Specification Requests related to XML

- JSR 155 – Web Services Security Assertions** To provide a set of APIs, exchange patterns & implementation to securely (integrity and confidentiality) exchange assertions between web services based on OASIS SAML.
- JSR 156 – Java API for XML Transactions** JAXTX provides an API for packaging and transporting ACID transactions (as in JTA) and extended transactions (e.g., the BTP from OASIS) using the protocols being defined by OASIS, W3C.
- JSR 157 – ebXML CPP/A APIs for Java** This JSR is to provide a standard set of APIs for representing and manipulating Collaboration Profile and Agreement information described by ebXML CPP/A (Collaboration Protocol Profile/Agreement) documents.
- JSR 173 – Streaming API for XML** The Streaming API for XML (StAX) is a Java based API for pull-parsing XML
- JSR 181 – Web Services Metadata for the Java™ Platform** This JSR defines an annotated Java™ format that that uses Java™ Language Metadata (JSR 175) to enable easy definition of Java Web Services in a J2EE container
- JSR 183 – Web Services Message Security APIs** This JSR is to define a standard set of APIs for Web services message security. The goal of this JSR is to enable applications to construct secure SOAP message exchanges.
- JSR 206 – Java™ API for XML Processing (JAXP) 1.3** JAXP 1.3 is the next version of JAXP, an implementation independent portable API for processing XML with Java™.
- JSR 222 – Java™ Architecture for XML Binding (JAXB) 2.0** JAXB 2.0 is the next version of JAXB, The Java™ Architecture for XML Binding. This JSR proposes additional functionality while retaining ease of development as a key goal.
- JSR 224 – Java™ API for XML-Based Web Services (JAX-WS) 2.0** The JAX-WS 2.0 specification is the next generation web services API replacing JAX-RPC 1.0
- JSR 225 – XQuery API for Java™ (XQJ)** Develop a common API that allows an application to submit queries conforming to the W3C XQuery 1.0 specification and to process the results of such queries.
- JSR 261 – Java™ API for XML Web Services Addressing (JAX-WSA)** The Java API for XML Web Services Addressing (JAX-WSA) 1.0 specification will define APIs and a framework for supporting transport-neutral addressing of Web services.



## List of Figures

5.1	A simple Person class . . . . .	36
5.2	Inheritance example . . . . .	37
6.1	Castor XML architecture overview . . . . .	49
7.1	JAXB2 Architecture Overview . . . . .	51
7.2	Castor JAXB2 Architecture Overview . . . . .	75
7.3	Castor JAXB2 Flow for Marshaling . . . . .	76
7.4	Castor JAXB2 Flow for Unmarshaling . . . . .	76
7.5	Castor JAXB2 Flow for Java Code Generation . . . . .	77
7.6	Castor JAXB2 Flow for W3C XML Schema Generation . . . . .	77
A.1	Tag cloud for the thesis text . . . . .	88

*List of Figures*

# List of Tables

4.1 XML History . . . . .	11
4.2 History of XML Standards . . . . .	34

*List of Tables*

# Listings

4.1	HTMLSample.html	8
4.2	CSVSample	9
4.3	JSONSample	9
4.4	XMLSample.xml	10
4.5	ElementWithPcdata.dtd	13
4.6	ElementWithPcdata.xml	13
4.7	DTDSequenceOfChildren.dtd	13
4.8	DTDSequenceOfChildren.xml	14
4.9	DTDChoiceOfChildren.dtd	14
4.10	DTDChoiceOfChildren.xml	14
4.11	DTDChildrenWithMultiplicity.dtd	14
4.12	DTDChildrenWithMultiplicity.xml	15
4.13	DTDMixedContent.dtd	15
4.14	DTDMixedContent.xml	15
4.15	DTDEmptyElement.dtd	16
4.16	DTDEmptyElement.xml	16
4.17	DTDAnyElement.dtd	16
4.18	DTDAnyElement.xml	16
4.19	DTDAttribute.dtd	16
4.20	DTDAttribute.xml	16
4.21	ElementBasedOnSimpleType.xsd	18
4.22	SimpleTypeNamedVsAnonymous.xsd	18
4.23	SimpleTypeByRestriction.xsd	19
4.24	SimpleTypeByRestriction.xml	19
4.25	SimpleTypeByUnion.xsd	19
4.26	SimpleTypeByUnion.xml	19
4.27	SimpleTypeByList.xsd	20
4.28	SimpleTypeByList.xml	20
4.29	CmplxExtendingSmpl.xsd	20
4.30	CmplxExtendingSmpl.xml	20
4.31	CmplxRestrictingSmpl.xsd	20
4.32	CmplxRestrictingSmpl.xml	21
4.33	CmplxTypeAll.xsd	21
4.34	CmplxTypeAll.xml	22

Listings

4.35 CmplxTypeSequence.xsd . . . . .	22
4.36 CmplxTypeSequence.xml . . . . .	22
4.37 CmplxTypeChoice.xsd . . . . .	22
4.38 CmplxTypeChoice.xml . . . . .	23
4.39 CmplxTypeGroupRef.xsd . . . . .	23
4.40 CmplxTypeGroupRef.xml . . . . .	24
4.41 NameGroup.xsd . . . . .	24
4.42 NameGroup.xml . . . . .	25
4.43 CmplxTypeExtension.xsd . . . . .	25
4.44 CmplxTypeExtension.xml . . . . .	25
4.45 CmplxTypeRestriction.xsd . . . . .	26
4.46 CmplxTypeRestriction.xml . . . . .	26
4.47 CmplxTypeMixed.xsd . . . . .	27
4.48 CmplxTypeMixed.xml . . . . .	27
4.49 AnyElement.xsd . . . . .	27
4.50 AttributeGroup.xsd . . . . .	28
4.51 substitution-group.xsd . . . . .	29
4.52 ElementRef.xsd . . . . .	30
5.1 Artist.java . . . . .	39
5.2 ShowClassAssociation.java . . . . .	40
7.1 MusicService.xsd . . . . .	52
7.2 Artist.java . . . . .	53
7.3 Label.java . . . . .	54
7.4 Artist.java . . . . .	54
7.5 ArtistToXmlSample.java . . . . .	55
7.6 artist.xml . . . . .	56
7.7 ForXmlElement.xsd . . . . .	60
7.8 ForXmlElement.xsd . . . . .	60
7.9 WrapperSample.xml . . . . .	61
7.10 Marshaller.java . . . . .	66
7.11 MarshalListener.java . . . . .	66
7.12 Unmarshaller.java . . . . .	67
7.13 UnmarshalListener.java . . . . .	67

## Bibliography

- [Adv06] Thinking XML: Good advice for creating XML. <http://www.ibm.com/developerworks/xml/library/x-think35.html>, January 2006.
- [ann98] The annotated XML specification. <http://www.xml.com/axml/testaxml.htm>, February 1998.
- [Bos97] XML, Java, and the future of the web. <http://www.xml.com/pub/a/w3j/s3.bosak.html>, October 1997.
- [cas08] The Castor project. <http://www.castor.org/index.html>, March 2008.
- [cir] Erik Meijer, Wolfram Schulte, Gavin Bierman *Programming with circles, triangles and rectangles*. <http://research.microsoft.com/emeijer/Papers/XML2003/xml2003.html>, Proceedings DP-COOL 2003.
- [com] Gavin Bierman, Erik Meijer, Wolfram Schulte. *The essence of data access in Cw*. <http://research.microsoft.com/Users/gmb/Papers/ecoop-corrected.pdf>, captured in April 2008.
- [csa06] C# application markup language (csaml): A preview. <http://www.charlespetzold.com/etc/CSAML.html>, April 2006.
- [cul] The cultural impedance mismatch between data professionals and application developers. <http://www.agiledata.org/essays/culturalImpedanceMismatch.html>, captured in April 2008.
- [DOM] W3C Document Object Model. <http://www.w3.org/DOM/>, 2008.
- [dom98] Document object model (DOM) level 1 specification. <http://www.w3.org/TR/REC-DOM-Level-1/>, October 1998.
- [dom00a] Document object model (DOM) level 2 core specification. <http://www.w3.org/TR/DOM-Level-2-Core/>, November 2000.
- [dom00b] Document object model (DOM) level 2 events specification. <http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/>, November 2000.

## Bibliography

- [dom00c] Document object model (DOM) level 2 style specification. <http://www.w3.org/TR/DOM-Level-2-Style/>, November 2000.
- [dom00d] Document object model (DOM) level 2 traversal and range specification. <http://www.w3.org/TR/DOM-Level-2-Traversal-Range/>, November 2000.
- [dom00e] Document object model (DOM) level 2 views specification. <http://www.w3.org/TR/DOM-Level-2-Views/>, November 2000.
- [dom02] Document object model (DOM) level 3 abstract schemas specification. <http://www.w3.org/TR/DOM-Level-3-AS/>, July 2002.
- [dom03] Document object model (DOM) level 2 html specification. <http://www.w3.org/TR/DOM-Level-2-HTML/>, January 2003.
- [dom04a] Document object model (DOM) level 3 core specification. <http://www.w3.org/TR/DOM-Level-3-Core/>, April 2004.
- [dom04b] Document object model (DOM) level 3 events specification. <http://www.w3.org/TR/DOM-Level-3-Events/>, April 2004.
- [dom04c] Document object model (DOM) level 3 load and save specification. <http://www.w3.org/TR/DOM-Level-3-LS/>, April 2004.
- [dom04d] Document object model (DOM) level 3 validation specification. <http://www.w3.org/TR/DOM-Level-3-Val/>, January 2004.
- [dom04e] Document object model (DOM) level 3 views and formatting specification. <http://www.w3.org/TR/DOM-Level-3-Views/>, February 2004.
- [dom04f] Document object model (DOM) level 3 XPATH specification. <http://www.w3.org/TR/DOM-Level-3-XPath/>, February 2004.
- [dom05] DOM4J - DOM4J: the flexible XML framework for java. <http://www.dom4j.org/>, May 2005.
- [ear03] Java technology: The early years. <http://java.sun.com/features/1998/05/birthday.html>, April 2003.
- [EG95] Ralph Johnson, John Vlissides, Erich Gamma, Richard Helm. *Design Patterns - Elements of reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Eva04] Eric Evans. *Domain-Driven Design*. Addison Wesley, 2004.
- [Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

- [Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [Har02] Elliotte Rusty Harold. *XML Schema*. O'Reilly Associates, Inc., 2002.
- [Har04] Elliotte Rusty Harold. *Effective XML*. Addison Wesley, 2004.
- [jav95] The Java saga. <http://www.wired.com/wired/archive/3.12/java.saga.html>, December 1995.
- [jax06] Welcome to JAXME 2. <http://ws.apache.org/jaxme/>, October 2006.
- [jax08a] JAXB reference implementation. <https://jaxb.dev.java.net/>, 2008.
- [jax08b] JAXP reference implementation. <https://jaxp.dev.java.net/>, 2008.
- [jdo07] JDOM. <http://www.jdom.org/>, November 2007.
- [jse] Elisabetta Poleo, Gianfranco Rossi. JSetL: Declarative Programming in Java with Sets. <http://www.math.unipr.it/gianfr/PAPER-S/JSetL.DPCOOL03.ps>, captured in April 2008.
- [jso08] JSON. <http://www.json.org/>, June 2008.
- [jsr00] JSR 5: XML parsing specification. <http://www.jcp.org/en/jsr/detail?id=05>, March 2000.
- [jsr01] JSR 63: Javatm API for XML processing 1.1. <http://www.jcp.org/en/jsr/detail?id=63>, February 2001.
- [jsr03] JSR 31: XML data binding specification. <http://www.jcp.org/en/jsr/detail?id=31>, March 2003.
- [jsr04] JSR 173: Streaming API for XML. <http://www.jcp.org/en/jsr/detail?id=173>, March 2004.
- [jsr06] JSR 222: Java architecture for XML binding (JAXB) 2.0. <http://jcp.org/en/jsr/detail?id=222>, 2006.
- [jsr08] Java specification requests - list of all JSRs. <http://jcp.org/en/jsr/all>, 2008.
- [Lau02] Simon St. Laurent. Monastic xml.org. <http://monasticxml.org/one.html>, 2002.
- [ME06] Brett D. McLaughlin and Justin Edelson. *Java & XML*. O'Reilly, 2006.
- [Mea04] Elliotte Rusty Harold & W.Scott Means. *XML in a Nutshell*. O'Reilly, 2004.

## Bibliography

- [MS07] Samuel Michaelis and Wolfgang Schmiesing. *JAXB 2.0*. Hanser, 2007.
- [Nel97] Theodor Holm Nelson. XML.com: Embedded markup considered harmful. <http://www.xml.com/pub/a/w3j/s3.nelson.html>, October 1997.
- [ori] The object-relational impedance mismatch. <http://www.agiledata.org/essays/impedanceMismatch.html>, captured in April 2008.
- [Pun07] Franz Puntigam. Skriptum zu objektorientierte programmierung. 2007.
- [rel01] RelaxNG specification. <http://relaxng.org/spec-20011203.html>, December 2001.
- [sax08] SAX project homepage. <http://www.saxproject.org/>, June 2008.
- [sch01a] XML schema part 0: Primer second edition. <http://www.w3.org/TR/xmlschema-0/>, May 2001.
- [sch01b] XML schema part 1: Structures second edition. <http://www.w3.org/TR/xmlschema-1/>, May 2001.
- [sch01c] XML schema part 2: Datatypes second edition. <http://www.w3.org/TR/xmlschema-2/>, May 2001.
- [sgm08a] A gentle introduction to SGML. <http://www.isgmlug.org/sgmlhelp/g-sg.htm>, June 2008.
- [sgm08b] SGML: General introductions and overviews. <http://xml.coverpages.org/general.html>, June 2008.
- [Siv05] Henri Sivonen. Howto avoid being called a bozo when producing XML. <http://hsivonen.iki.fi/producing-xml/>, September 2005.
- [SV06] Joe Fialli Sekhar Vajjhala. *The Java Architecture for XML Binding (JAXB) 2.0*. Sun Microsystems, Inc., final release edition, April 2006.
- [tax] Makoto Murata, Dongwon Lee, Murali Mani Taxonomy of XML Schema Languages using Formal Language Theory. <http://www.cobase.cs.ucla.edu/tech-docs/dongwon/mura0619.pdf>, 2000.
- [tod] Erik Meijer, Wolfram Schulte. *Unifying Tables, Objects and Documents*. <http://www.research.microsoft.com/emeijer/Papers/XS.pdf>, Proceedings DP-COOL 2003.

- [esx] Jerome Simeon, Philip Wadler. *The Essence of XML*. <http://www.research.avayalabs.com/user/wadler/papers/xml-essence/xml-essence.pdf>, Proceedings of POPL 2003, New Orleans, January 2003.
- [toi] Dave Thomas. *The Impedance Imperative Tuples + Objects + Infosets = Too Much Stuff!*. <http://www.jot.fm/issues/issue200309/column1.pdf>, JOT, 2003 Vol. 2, No. 5, September-October 2003.
- [tra] Luca Cardelli. Transitions in programming models. ICSE '05: Proceedings of the 27th international conference on Software engineering, 2005.
- [tun96] SGML: In memory of William W. Tunnicliffe. <http://xml.coverpages.org/tunnicliffe.html>, September 1996.
- [w3c08a] Overview of SGML resources. <http://www.w3.org/MarkUp/SGML/>, June 2008.
- [w3c08b] W3C technical reports and publications by date. <http://www.w3.org/TR/tr-date>, June 2008.
- [xht00] XHTML 1.0: The extensible hypertext markup language (second edition). <http://www.w3.org/TR/xhtml1/>, January 2000.
- [xml96] Design principles for XML. <http://www.textuality.com/sgml-erb/dd-1996-0001.html>, August 1996.
- [xml98] Extensible markup language (XML) 1.0. <http://www.w3.org/TR/xml/>, February 1998.
- [xml01a] Canonical XML version 1.0. <http://www.w3.org/TR/xml-c14n>, March 2001.
- [xml01b] XML information set. <http://www.w3.org/TR/xml-infoset>, October 2001.
- [xml04] Extensible markup language (XML) 1.1. <http://www.w3.org/TR/xml11/>, February 2004.
- [xml06a] Namespaces in XML 1.0 (second edition). <http://www.w3.org/TR/REC-xml-names/>, August 2006.
- [xml06b] Namespaces in XML 1.1 (second edition). <http://www.w3.org/TR/xml-names11/>, August 2006.
- [xml07a] Welcome to XMLBEANS. <http://xmlbeans.apache.org/index.html>, June 2007.

## Bibliography

- [xml07b] XQUERY 1.0 and XPATH 2.0 data model (XDM). <http://www.w3.org/TR/xpath-datamodel/>, January 2007.
- [xoi] Ralf Lammel and Erik Meijer *Revealing the X/O impedance mismatch*. <http://homepages.cwi.nl/~ralf/xo-impedance-mismatch/paper.pdf>, captured in April 2008.
- [xpa99] XML path language (XPath) version 1.0. <http://www.w3.org/TR/xpath>, November 1999.
- [xpa07] XML path language (XPath) 2.0. <http://www.w3.org/TR/xpath20/>, January 2007.
- [xsl99] XSL transformations (XSLT). <http://www.w3.org/TR/xslt>, November 1999.
- [xst08] XSTREAM - about XSTREAM. <http://xstream.codehaus.org/>, June 2008.