exolab.org

CASTOR XML
SOURCE CODE GENERATOR

USER DOCUMENT

Author: Arnaud Blandin [blandin@intalio.com]

Revision: December 15, 2000

Abstract

As a complete **XML**[1] data binding framework, Castor XML offers the ability to generate Java code from an **XML Schema.**

The aim of this document is to describe the Castor XML Source Code Generator, including the main options and features, as well as to illustrate the mechanism used to generate java classes with a simple example. Open issues of the project are also discussed.

1: all **bold** names (except in section 2.3) are defined in the glossary.

1 Why a Source Code Generator?

1.1XML Data Binding

Many current applications which manipulates XML documents rely on XML Schemas which define the structure, the content and even the meaning of these XML documents.

In order to deal with the XML 'constraints' defined in the schema, applications need some tools to create and manipulate XML documents that are instances of the given XML Schema.

Such tools might be written using the **SAX API** or the **DOM API,** however these approaches are more focused on the structure of an XML document than the data itself.

Moreover all data in these APIs are treated as strings and must likely need to be cast to an appropriate data type.

It would be much easier if these applications could map directly an XML document to its in-memory object representation which contains all the information provided by the XML Schema, this is what we call **XML data binding**.

1.2The role of the source generator

To represent the data model of an XML document in memory, developers need to hard-code the description of the XML document. They need to describe the structure and the data of the document provided by the XML Schema.

Sometimes it could be easy when you only need to map a String or a Boolean, you can find the exact mapping in any Object Oriented language but when it is time to describe a more complex structure with some inner XML Schema types it can become very tedious and complex.

The aim of the source generator is to provide the necessary code to describe XML instances of a specific XML Schema with the proper fields and access methods.

To sum up, we can draw a parallel between the relations XML Schema-XML and Class-Object: an XML document is an ***instance*** of an XML Schema and an Object is an ***instance*** of a Class. Thus to represent an XML document as an Object in memory, we need to provide the Class that describes this object.

The Source Code Generator is merely generating the code for this class.

The Castor XML Source Code Generator – from now on referred to the Source Generator – is generating Java source code from a W3C XML Schema. The generated source includes an object model of the schema as well as the necessary Class Descriptors used by the **marshalling framework** to obtain information about the generated classes.

2 Usage, options & XML Schema support

2.1 Usage

The source generator can be used as a **command-line** tool that can simply be invoked by calling :

java org.exolab.castor.builder.SourceGenerator –options

or using the script files SourceGenerator.bat or SourceGenerator.sh.

The API can also be used, for more information refer to the Javadoc of the SourceGenerator class.

Note: the generated source files need to be compiled by hand.

2.2 Source Generator Options

The source code generator has a number of different options which may be set. Some of these are done using the command line, and others are done using a properties file (castorbuilder.properties).

2.2.1 Command Line Options

Option	Args	Description	Status
i	filename	The input XML Schema file	Required
package	*package-name*	The package for the generated source	Optional
dest	*path*	The destination in which to put the generated source	Optional
line-separator	*unix | mac | win*	Sets the line separator style for the desired platform. This is useful if you are generating source on one platform, but will be compiling/modifying on another platform.	Optional
types	*type-factory*	Sets which type factory to use. This is useful if you want JDK 1.2 collections instead of JDK 1.1) (see below)	Optional
h		Shows the help/usage screen	Optional
f		Forces the source generator to supress all non-fatal errors, such as overwriting of pre-existing files.	Optional
nodesc		Do not generate the class descriptors	Optional

Collection types:

The source code generator has the ability to use the following types of collections when generating source code:

- Java 1.1 (default) java.util.Vector.
- Java 1.2: if the option is types –j2, collection type will be java.util.Collections implemented as ArrayList.

- ODMG 3.0: if the option is types –odmg, collection type will be odmg.Darray.

2.2.3 Advanced options

These options are set up in the `org/exolab/castor/builder/castorbuilder.properties` file.

2.2.3.1 Bound Properties

Since version: 0.8.9

Bound properties are "properties" of a class, which when updated will send out a java.beans.PropertyChangeEvent to all registered java.beans.PropertyChangeListeners.

To enable bound properties, uncomment the appropriate line in the "org/exolab/castor/builder/castorbuilder.properties" file:

```
 # To enable bound properties uncomment the
 # following  line.
 # Please note that currently *all* fields will
 # be treated as bound properties
 # when enabled. This will change in the future
 # when we introduce fine grained control over
 # each class and its properties.
#
#org.exolab.castor.builder.boundproperties=true
```

When enabled, all properties will be treated as bound properties. For each class that is generated a setPropertyChangeListener method is created as follows:

```
/**
 * Registers a PropertyChangeListener with this class.
 * @param pcl The PropertyChangeListener to register.
 */

public void
addPropertyChangeListener(java.beans.PropertyChangeListener pcl)
{
propertyChangeListeners.addElement(pcl);
} //-- void
addPropertyChangeListener(java.beans.PropertyChangeListener)
```

Whenever a property of the class is changed, a PropertyChangeEvent will be sent to all registered listeners. The property name, the old value, and the new value will be set in the PropertyChangeEvent.

Note: To prevent unnecessary overhead, if the property is a collection, the old value will be null.

2.2.3.2 Class Creation/Mapping

Since version: 0.8.9

The source generator can treat the XML Schema structures such as complexType and element in two main ways.

The first, and currently default method is called the **"element"** method. The other is called the **"type"** method.

In the following we will illustrate the class creation with the following schema:

```
<?xml version='1.0'?>
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
          targetNamespace="http://castor.exolab.org/Examples">

     <complexType name="A">
          <sequence>
               <element name="B" type="string"/>
               <element name="C"     type="string"/>
          </sequence>
     </complexType>

</schema>
```

The 'element' method

The "element" method creates classes for all elements whose type is a complexType. Abstract classes are created for all top-level complexTypes. Any elements whose type is a top-level type will have a new class created that extends the abstract class which was generated for that top-level complexType.

Classes are not created for elements whose type is a simpleType.

This approach tends to describe the structure of an XML Document that is a particular instance of the XML Schema used to generate the source code.

In this case the generated class will be:

```
/*
 * This class was automatically generated with
 * <a href="http://castor.exolab.org">Castor 0.8.10</a>, using an
 * XML Schema.
 * $Id$
 */

package illustrate;

  //---------------------------------/
 //- Imported classes and packages -/
//---------------------------------/

import java.io.Reader;
import java.io.Serializable;
import java.io.Writer;
import org.exolab.castor.xml.*;
import org.exolab.castor.xml.ValidationException;

/**
 *
 * @version $Revision$ $Date$
**/
public abstract class A implements java.io.Serializable {


      //--------------------------/
     //- Class/Member Variables -/
    //--------------------------/

    private java.lang.String _b;

    private java.lang.String _c;
```

The 'type' method

The "type" method creates classes for all top-level complexTypes, or elements that contain an "anonymous" (in-lined) complexType.

Classes will not be generated for elements whose type is a top-level type.

This second approach tries to produce a class hierarchy that represents the XML Schema itself more than an instance of this XML Schema.

In this case the generated class will be:

```
package illustrate;

  //---------------------------------/
 //- Imported classes and packages -/
//---------------------------------/

import java.io.Reader;
import java.io.Serializable;
import java.io.Writer;
import org.exolab.castor.xml.*;
import org.exolab.castor.xml.MarshalException;
import org.exolab.castor.xml.ValidationException;
import org.xml.sax.DocumentHandler;

/**
 *
 * @version $Revision$ $Date$
**/
public class A implements java.io.Serializable {


      //--------------------------/
     //- Class/Member Variables -/
    //--------------------------/

    private java.lang.String _b;

    private java.lang.String _c;
```

To change the "method" of class creation simply edit the `castorbuilder.properties` file:

```
# Java class mapping of <xsd:element>'s and <xsd:complexType>'s
#
#org.exolab.castor.builder.javaclassmapping=element
```

2.2.3.3 Setting a super class

The source generator enables the user to set a super class to **all** the generated classes (of course class descriptors are not concerned by this option).

To set the super class, edit the `castorbuilder.properties` file:

```
# This property allows one to specify the super class of *all*
# generated classes
#
#org.exolab.castor.builder.superclass=com.xyz.BaseObject
```

2.3 XML Schema support

The source generator supports the W3C XML Schema Candidate Recommendation document (10/24/2000).

Roughly speaking the source generator maps a XML Schema type to a corresponding Java type.

It happens that a Schema type does not have its corresponding one in Java. Thus the Source Generator uses Castor implementation of these specific types (located in the package org.exolab.castor.types). For instance the TimeDuration type is implemented directly in Castor.

Many built-in types are supported but not all of them. You will find below a detailed list of supported built-in types and then some comments about the support.

Remember that the representation of XML Schema datatypes does not try to fit exactly the W3C XML Schema specifications, the aim is to map an XML Schema type to the java type that fit the most to the XML Schema type.

For instance this can explain why you won't find the support for the 'scale' facet in the Integer support.

2.3.1 Built-in types

2.3.1.1 Primitive Datatypes

The **bold** names refer to features supported by the Source Generator.

XML Schema Datatypes	Facets	Java corresponding types
string	**length** **minLength** **maxLength** **pattern** **enumeration** **whiteSpace**	java.lang.String
boolean	pattern whiteSpace	primitive boolean type
float	**pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive float type
double	**pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive double type
decimal	**scale** **precision** pattern enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	java.math.BigDecimal
timeDuration	pattern enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	org.exolab.castor.types.timeDuration
recurringDuration	pattern enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	org.exolab.castor.types.recurringDuration

XML Schema Datatypes	**Facets**	**Java corresponding types**
binary	encoding length max/min length pattern enumeration whiteSpace	primitive byte array
uriReference	length max/min length pattern whiteSpace enumeration	java.lang.String
ID	length max/min length pattern whitespace enumeration max/min Exclusive max/min Inclusive	java.lang.String
IDREF	length max/min length pattern whitespace enumeration max/min Exclusive max/min Inclusive	java.lang.Object
Entity	length max/min length pattern whitespace enumeration max/min Exclusive max/min Inclusive	
Qname	**length** **max/min length** **pattern** whiteSpace enumeration max/min Exclusive max/min Inclusive	

2.3.1.2 Derived datatypes

The **bold** names refer to features supported by the Source Generator.

XML Schema Datatypes	**Facets**	**Java corresponding types**
CDATA	**length** **max/min Length** **pattern** enumeration whiteSpace	
token	length max/min Length pattern enumeration whiteSpace	
language	length max/min Length pattern enumeration whiteSpace	
IDREFS	length max/min Length enumeration whiteSpace	java.util.Vector of XSIdRef
ENTITIES	length max/min Length pattern enumeration whiteSpace	
NMTOKEN	length max/min Length pattern enumeration whiteSpace	java.lang.String
NMTOKENS	length max/min Length pattern enumeration whiteSpace	java.util.Vector of XSNMToken

XML Schema Datatypes	**Facets**	**Java corresponding types**
Name	length max/min Length pattern enumeration whiteSpace	
NCName	length max/min Length pattern enumeration whiteSpace	java.lang.String
NOTATION	length max/min Length pattern enumeration whiteSpace max/min Exclusive max/min Inclusive	
integer	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive int type.
nonPositiveInteger	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive int type

XML Schema Datatypes	Facets	Java corresponding types
negativeInteger	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive int type.
long	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive long type
int	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive int type
short	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive short type

XML Schema Datatypes	**Facets**	**Java corresponding types**
byte	scale precision pattern enumeration whiteSpace max/min Exclusive max/min Inclusive	
nonNegativeInteger	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive int type
unsignedLong	scale precision pattern enumeration whiteSpace max/min Exclusive max/min Inclusive	
unsignedInt	scale precision pattern enumeration whiteSpace max/min Exclusive max/min Inclusive	

XML Schema Datatypes	Facets	Java corresponding types
unsignedShort	scale precision pattern enumeration whiteSpace max/min Exclusive max/min Inclusive	
unsignedByte	scale precision pattern enumeration whiteSpace max/min Exclusive max/min Inclusive	
positiveInteger	scale precision **pattern** enumeration whiteSpace **max/min Exclusive** **max/min Inclusive**	primitive int type
timeInstant	**duration** **period** pattern whiteSpace max/min Exclusive max/min Inclusive	java.util.Date
time	**duration** **period** pattern whiteSpace max/min Exclusive max/min Inclusive	org.exolab.castor.types.time
timePeriod	**duration** **period** pattern whiteSpace max/min Exclusive max/min Inclusive	org.exolab.castor.types.TimePeriod

XML Schema Datatypes	Facets	Java corresponding types
date	**duration** **period** pattern whiteSpace max/min Exclusive max/min Inclusive	org.exolab.castor.types.Date
month	**duration** **period** pattern whiteSpace max/min Exclusive max/min Inclusive	org.exolab.castor.types.Month
year	**duration** **period** pattern whiteSpace max/min Exclusive max/min Inclusive	org.exolab.castor.types.Year
century	**duration** **period** pattern whiteSpace max/min Exclusive max/min Inclusive	org.exolab.castor.Century
recurringDate	duration period pattern whiteSpace max/min Exclusive max/min Inclusive	
recurringDay	duration period pattern whiteSpace max/min Exclusive max/min Inclusive	

2.3.1.3 Conclusion – Comments

Primitive datatypes

The Source Generator supports 12 of the 13 W3C XML Schema primitive datatypes.

However this support is not complete and sometimes full support is not required.

- **decimal [section 3.2.5 XML Schema Part 2 :datatypes, W3C CR]**

Castor maps an XML Schema decimal to a java.math.BigDecimal. This mapping does not allow the full support. Although it is possible to validate a decimal towards the constraining facet precision **[section 2.4.2.11, XML Schema Part 2 :datatypes]**

It is not possible to create a default java.math.BigDecimal with the correct precision since this feature is not implemented in Java.

- **binary [section 3.2.8 XML Schema Part 2 :datatypes, W3C CR]**

The encoding constraint **[section 2.4.2.13 XML Schema Part 2 :datatypes]** is not yet supported. The current encoding of binary data is base64.

- **uriReference [section 3.2.9 XML Schema Part 2 :datatypes, W3C CR]**

A uriReference is currently represented as a string. No specific validation is done.

- **ENTITY [section 3.2.12 XML Schema Part 2 :datatypes, W3C CR]**

Not Supported

The type is inherited from the DTD model which tends to be replaced by XML Schema. Moreover the use of this type is not widespread enough for requiring support in the Source Generator.

- **Qname [section 3.2.13 XML Schema Part 2 :datatypes, W3C CR]**

This type is only represented as a String. A full representation of it requires a full representation of uriReference.

Derived datatypes

The Source Generator is supporting 20 of the 32 XML Schema derived datatypes

- **token [section 3.3.2 XML Schema Part 2:datatypes, W3C CR]**

Not Supported

The use of token is not widespread, its derived type (NMToken, Name) are more important.

- **language [section 3.3.3 XML Schema Part 2:datatypes, W3C CR]**

Not Supported

This type is rarely used.

- **ENTITIES [section 3.3.5 XML Schema Part 2:datatypes, W3C CR]**

Not Supported

c.f. comments on ENTITY primitive type

- **NOTATION [section 3.3.10 XML Schema Part 2:datatypes, W3C CR]**

Not Supported

The reasons are the same as ENTITY non-support.

- **unsignedLong, unsignedInt, unsignedShort, unsignedByte [section 3.3.19, 3.3.20,3.3.21, 3.3.22 XML Schema Part 2: datatypes, W3C CR]**

Not Supported

In Java, all numeric types are signed, it is not possible to deal properly with unsigned numeric types.

Structure

The following section is based on the **XML Schema Part 1: structure, W3C CR.**

Each paragraph number refers to the W3C document

Schema Component: Schema

- **§ 3.1 Schema details**

  Supported:

  - {type definitions}

  - {attribute declarations}

  - {attribute group definitions}

  - {model group definitions]

  - {annotations}

  Unsupported:

  - {notation declarations}

- **§ 4.1 XML Representations of Schemas**

  Unsupported features appear in *italics*

<schema
  attributeFormDefault = (qualified | unqualified) : unqualified
  blockDefault = (#all | List of (substitution | extension | restriction))
  elementFormDefault = (qualified | unqualified) : unqualified
   id = ID
  targetNamespace = uriReference
  version = string
  {any attributes with non-schema namespace . . . }>
  Content: ((include | import | *redefine* | annotation)*, ((attribute|attributeGroup | complexType | element | group | *notation* | simpleType), annotation*)*)
</schema>

Schema Component: Attribute Declaration

- **§ 3.2 Attribute Declaration details**

  Supported:

  - {name}

  - {target namespace}

  - {type definition}

  - {scope}

  - {value constraint}

  - {annotation}

- **§ 4.3.1 XML Representations of Attribute Declaration Schema Components**

  Unsupported features appear in *italics*

```
<attribute
    form = (qualified | unqualified)
    id = ID
    name = NCName
    ref = QName
    type = QName
    use = (prohibitied | optional | required | default | fixed) : optional
    value = string
    {any attributes with non-schema namespace . . . }>
    Content: (annotation?, (simpleType?))
</attribute>
```

Schema Component: Element Declaration

- **§ 3.3 Element Declaration details**

  Supported:

  - {name}
  - {target namespace}
  - {type definition}
  - {scope}
  - {annotation}

  Unsupported:

  - {abstract}
  - {value constraint}
  - {nullable}
  - {identity-constraintdefinitions}
  - {substitution group affilitaion}
  - {substitution group exclusions}
  - {disallowed substitutions}

- **§ 4.3.2 XML Representation of Element Declaration Schema Components**

  Unsupported features appear in *italics*

<element
    abstract = boolean : false
    block = (#all | List of (substitution | extension | restriction))
    default = string
    final = (#all | List of (extension | restriction))
    fixed = string
    form = (qualified | unqualified) : unqualified
    id = ID
    maxOccurs = (nonNegativeInteger | unbounded) : 1
    minOccurs = nonNegativeInteger : 1
    name = NCName
    nullable = boolean : false
    ref = QName
    substituionGroup = QName
    type = QName
    {any attributes with non-schema namespace . . . }>
    Content: (annotation?, ((simpleType | complexType)?, (*key* | *keyref* | *unique*)*))
</element>

Schema Component: Complex Type Definition

- **§ 3.4 Complex Type Definition Details**

  Supported:

  - {name}
  - {target namespace}
  - {base type definition}
  - {derivation method} - extension
  - {abstract}
  - {attribute use pairs}
  - {content type}
  - {annotations}

  Unsupported:

  - {derivation method} - restriction
  - {final}
  - {attribute wildcard}
  - {prohibited substitutions}

Schema Component: Attribute Group Definition

- **§ 3.5 Attribute Group Definition Details**

  Supported:

  - {name}
  - {target namespace}
  - {attribute use pairs}
  - {annotation}

  Unsupported:

  - {attribute wildcard}

Schema Component: Model Group Definition

We currently support the syntax for Model Group Definitions, however groups are currently flattened, and all are treated as "all".

- **§ 3.6 Model Group Definition Details**

  Supported:

  - {name}
  - {target namespace}
  - {model group}
  - {annotation}

Schema Component: Model Group

- **§ 3.7 Model Group Details**

Supported:

- {compositor}
- {particles}
- {annotation}

Schema Component: Particle

- **§ 3.8 Particle Details**

Supported:

- {min occurs}
- {max occurs}
- {term}

Schema Component: Wildcard

Wildcards are currently not supported.

- **§ 3.9 Wildcard Details**

  Unsupported:

  - {namespace constraint}

  - {process contents}

  - {annotation}

Schema Component: Identity-constraint Definition

Identity-constraints are currently not supported.

- **§ 3.10 Identity-constraint Definition Details**

  Unsupported:

  - {name}

  - {target namespace}

  - {identity-constraint category}

  - {selector}

  - {fields}

  - {referenced key}

  - {annotation}

Schema Component: Notation Declaration

Notations are currently not supported.

- **§ 3.11 Notation Declaration Details**

  Unsupported:

  - {name}
  - {target namespace}
  - {system identifier}
  - {public identifier}
  - {annotation}

Schema Component: Annotation

- **§ 3.12 Annotation Details**

  Supported:

  - {application information}
  - {user information}

Schema Component: Simple Type Definition

- **§ 3.13 Simple Type Definition Details**

  Supported:

  - {name}
  - {target namespace}
  - {base type definition}

- {facets}
- {variety}
    - atomic
- {annotation}

Unsupported:

- {variety}
    - list
    - union

2.4 Requirements

Castor XML is compliant with all JDK1.1 and above.

In order to run the Source Generator you'll need to set the following libraries in your classpath.

2.4.1 Castor packages

- org.exolab.castor.builder
- org.exolab.castor.builder.types
- org.exolab.castor.builder.util
- org.exolab.castor.mapping
- org.exolab.castor.types
- org.exolab.castor.xml

2.4.2 External libraries

- org.exolab.javasource : this package contains a full description of a java object
- Xerces-Java parser: the Apache XML parser is the parser used by castor
- Jakarta regular expression.

These external libraries are included in the Castor distribution.

3 Example

In this section we illustrate the use of the Source Generator by explaining the generated classes from a given schema.

The Source Generator is going to be used with the “java class mapping” property (cf section 2.2.3.2) set to ‘element’ (default value).

3.1 The schema file

The input file is the schema file given with the Source Generator example in the distribution of Castor (under /src/examples/SourceGenerator/invoice.xsd)

```
<?xml version="1.0"?>
    <schema xmlns="http://www.w3.org/2000/10/XMLSchema"
      targetNamespace="http://castor.exolab.org/Test/Invoice">

   <annotation>
      <documentation>
         This is a test XML Schema for Castor XML.
      </documentation>
   </annotation>

   <!--
       A simple representation of an invoice. This is simply an
example
       and not meant to be an exact or even complete
representation of an invoice.
     -->
   <element name="invoice">
      <annotation>
         <documentation>
             A simple representation of an invoice
         </documentation>
      </annotation>

      <complexType>
            <sequence>
                <element name="ship-to">
                <complexType>
                     <sequence>
                          <element name="name" type="string"/>
                          <element ref="address"/>
                          <element name="phone"
type="TelephoneNumber"/>
                     </sequence>
```

```
                 </complexType>
           </element>
           <element ref="item" maxOccurs="unbounded"
minOccurs="1"/>
           <element ref="shipping-method"/>
           <element ref="shipping-date"/>
           </sequence>
      </complexType>
   </element>

   <!-- Description of an item -->
   <element name="item">
           <complexType>
                 <sequence>
                       <element name="Id" type="ID" minOccurs="1"
maxOccurs="1"/>
                 <element name="Quantity" type="integer"
minOccurs="1" maxOccurs="1"/>
                 <element name="Price" type="PriceType"
minOccurs="1" maxOccurs="1"/>
           </sequence>
           <attributeGroup ref="ItemAttributes"/>
      </complexType>
      </element>

   <!-- Shipping Method -->
   <element name="shipping-method">
      <complexType>
             <sequence>
           <element name="carrier" type="string"/>
           <element name="option"  type="string"/>
           <element name="estimated-delivery"
type="timeDuration"/>
             </sequence>
      </complexType>
   </element>

   <!-- Shipping date -->
   <element name="shipping-date">
           <complexType>
                 <sequence>
                       <element name="date" type="date"/>
                       <element name="time" type="time"/>
                 </sequence>
           </complexType>
      </element>

   <!-- A simple U.S. based Address structure -->
   <element name="address">
      <annotation>
         <documentation>
```

```
            Represents a U.S. Address
         </documentation>
      </annotation>

      <complexType>
            <sequence>
           <!-- street address 1 -->
           <element name="street1" type="string"/>
           <!-- optional street address 2 -->
           <element name="street2" type="string" minOccurs="0"/>
           <!-- city-->
           <element name="city" type="string"/>
           <!-- state code -->
           <element name="state" type="state-code"/>
           <!-- zip-code -->
           <element ref="zip-code"/>
           </sequence>
      </complexType>
   </element>

   <!-- A U.S. Zip Code -->
   <element name="zip-code">
      <simpleType>
         <restriction base="string">
            <pattern value="[0-9]{5}(-[0-9]{4})?"/>
            </restriction>
      </simpleType>
   </element>

   <!-- State Code (obviously not a valid state code.) -->
     <simpleType name="state-code">
           <restriction base="string">
                <pattern value="[A-Z]{2}"/>
           </restriction>
   </simpleType>

   <!-- Telephone Number -->
   <simpleType name="TelephoneNumber">
           <restriction base="string">
                <length value="toto"/>
                <pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
           </restriction>
   </simpleType>

   <!-- Cool price type -->
   <simpleType name="PriceType">
           <restriction base="decimal">
                <scale value="2"/>
           <precision value="5"/>
           <minInclusive value="1"/>
                <maxInclusive value="100"/>
           </restriction>
   </simpleType>

   <!-- The attributes for an Item -->
   <attributeGroup name="ItemAttributes">
           <attribute name="InStock" type="boolean" use="default"
value="false"/>
           <attribute name="Category" type="string"
use="required"/>
    </attributeGroup>
</schema>
```

The structure of this schema is simple: it is composed of a top-level element which is a complexType with references to other elements inside.

This schema represents a simple invoice: an invoice is a *customer* ('ship-to' element), an *article* ('item' element), a *shipping method* ('shipping-method' element) and a *shipping date* ('shipping-date' element).

Notice that the 'ship-to' element uses a reference to an 'address' element. This 'address' element is a top-level element which contains a reference to a non-top-level element (the 'zip-code' element).

At the end of the schema we have two simpleTypes for representing a telephone number and a price.

The Source Generator is used with the 'element' property set for class creation (c.f.section 2.2.3.2) so a class is going to be generated for all top-level elements. No classes are going to be generated for complexTypes and simpleTypes since the simpleType is not an enumration.

To summarize, we can expect 6 classes : Invoice, Address, Item. ShipTo, ShippingMethod and ShippingDate and the 6 corresponding class descriptors.

To run the source generator and create the source from the invoice.xsd file in a package test, we just call in the command line:

```
java -cp %CP% org.exolab.castor.builder.SourceGenerator -i
invoice.xsd  -package test
```

3.2 The generated code

To simplify this example we now focus on the item element.

```
   <!-- Description of an item -->
   <element name="item">
         <complexType>
              <element name="Id" type="ID" minOccurs="1"
maxOccurs="1"/>
               <element name="Quantity" type="integer"
minOccurs="1" maxOccurs="1"/>
              <element name="Price" type="PriceType"
minOccurs="1" maxOccurs="1"/>
         </complexType>
   </element>

   <!-- Cool price type -->
   <simpleType name="PriceType" base="decimal">
     <scale value="2"/>
     <precision value="5"/>
     <minInclusive value="1"/>
     <maxInclusive value="100"/>
   </simpleType>
```

To represent an Item object, we need to know its 'Id', the 'Quantity' ordered and the 'Price' for one item.

So we can expect to find a least three private variables: a string for the 'Id' element, an `int` for the 'quantity' element (see the section on XML Schema support if you want to see the mapping between a W3C XML Schema type and a java type) but what type for the 'Price' element?

While processing the 'Price' element, Castor is going to process the type of 'Price' i.e. the simpleType 'PriceType' which base is 'decimal'. Since derived types are automatically mapped to parent types and W3C XML Schema 'decimal' type is mapped to a `java.math.BigDecimal`, the price element will be a `java.math.BigDecimal`.

Another private variable is created for 'quantity': quantity is mapped to a primitive java type so a `boolean` `has_quantity` is created for monitoring the state of the quantity variable.

The rest of the code is the 'getter-setter' methods and the Marshalling framework specific methods.

Here the whole Item class:

```
/*
 * This class was automatically generated with
 * <a href="http://castor.exolab.org">Castor 0.8.10</a>, using an
 * XML Schema.
 * $Id$
 */

package test;

  //---------------------------------/
 //- Imported classes and packages -/
//---------------------------------/

import java.io.Reader;
import java.io.Serializable;
import java.io.Writer;
import java.math.BigDecimal;
import org.exolab.castor.xml.*;
import org.exolab.castor.xml.MarshalException;
import org.exolab.castor.xml.ValidationException;
import org.xml.sax.DocumentHandler;

/**
 *
 * @version $Revision$ $Date$
**/
public class Item implements java.io.Serializable {


      //--------------------------/
     //- Class/Member Variables -/
    //--------------------------/

    private java.lang.String _id;

    private int _quantity;

    /**
     * keeps track of state for field: _quantity
    **/
    private boolean _has_quantity;

    private java.math.BigDecimal _price;
```

```
      //----------------/
     //- Constructors -/
    //----------------/

    public Item() {
        super();
    } //-- test.Item()


      //-----------/
     //- Methods -/
    //-----------/

    /**
    **/
    public java.lang.String getId()
    {
        return this._id;
    } //-- java.lang.String getId()

    /**
    **/
    public java.math.BigDecimal getPrice()
    {
        return this._price;
    } //-- java.math.BigDecimal getPrice()

    /**
    **/
    public int getQuantity()
    {
        return this._quantity;
    } //-- int getQuantity()

    /**
    **/
    public boolean hasQuantity()
    {
        return this._has_quantity;
    } //-- boolean hasQuantity()

    /**
    **/
    public boolean isValid()
    {
        try {
            validate();
        }
        catch (org.exolab.castor.xml.ValidationException vex) {
            return false;
        }
        return true;
    } //-- boolean isValid()

    /**
     *
     * @param out
    **/
    public void marshal(java.io.Writer out)
        throws org.exolab.castor.xml.MarshalException,
org.exolab.castor.xml.ValidationException
    {

        Marshaller.marshal(this, out);
    } //-- void marshal(java.io.Writer)

    /**
     *
     * @param handler
    **/
    public void marshal(org.xml.sax.DocumentHandler handler)
        throws org.exolab.castor.xml.MarshalException,
org.exolab.castor.xml.ValidationException
    {

        Marshaller.marshal(this, handler);
    } //-- void marshal(org.xml.sax.DocumentHandler)

    /**
     *
     * @param _id
    **/
    public void setId(java.lang.String _id)
    {
        this._id = _id;
    } //-- void setId(java.lang.String)

    /**
     *
     * @param _price
    **/
    public void setPrice(java.math.BigDecimal _price)
    {
        this._price = _price;
    } //-- void setPrice(java.math.BigDecimal)

    /**
     *
     * @param _quantity
    **/
    public void setQuantity(int _quantity)
    {
        this._quantity = _quantity;
        this._has_quantity = true;
    } //-- void setQuantity(int)

    /**
     *
     * @param reader
    **/
    public static test.Item unmarshal(java.io.Reader reader)
        throws org.exolab.castor.xml.MarshalException,
org.exolab.castor.xml.ValidationException
    {
        return (test.Item)
Unmarshaller.unmarshal(test.Item.class, reader);
    } //-- test.Item unmarshal(java.io.Reader)

    /**
    **/
    public void validate()
        throws org.exolab.castor.xml.ValidationException
    {
        org.exolab.castor.xml.Validator.validate(this, null);
    } //-- void validate()

}
```

The ItemDescriptor class is a bit more complex. This class is containing inner classes which are the XML field descriptors for the different components of an 'Item' element i.e. id, quantity and price.

4 FAQ

- **I have many simpleTypes in my schema but there is no classes generated to represent these simpleTypes?**

A simpleType remains merely a type. It has no particular meaning in the structure of an XML Schema or an instance if an XML Schema.

Right now the Source Generator generates source for a simpleType only when this simpleType is an enumeration.

- **How does the Source Generator handle the attribute 'maxOccurs' ?**

If you use in your schema an element with the attribute 'maxOccurs' set to a number greater than 1, the Source Generator will create a collection that can contain many elements. This collection depends on the property set up in the castorbuilder.properties file. For more details on collections, you can refer to section 2.2.1.

For instance, if you use the Source Generator with the default property and you want generate source for the following element:

```
<element name="Value" type="integer"
maxOccurs="unbounded"/>
```

The result will be

```
Private java.util.Vector ValueList
```

with some specific methods for dealing with the Vector:

```
addValue(int vValue),java.util.Enumeration enumerateValue(),
int   getValue(int index), int[] getValue(),
int getValueCount(),removeAllValue(),
int removeValue(int index),
setValue(int vVlaue, int index), setValue(int[] ValueArray).
```

Note: an array setter method is *automatically* created.

For example, given the following schema fragment:

```
…
  <complexType>
        <element name="products" type="product" minOccurs="0"
maxOccurs= "unbounded">
  </complexType>
…
```

The source code generator will produce the following "array setter" method:

```
…
   public void setProducts(Product[] productArray) {
      …
   }
…
```

- **What is the compatibility with the different JDKs?**

The Source Generator supports the different version of the JDKs since version 1.1.

But some features can work only a Java 2 Platform.

- **I want to use the Source Generator with a Castor Schema Object Model not with a data file, does the API allow that?**

You can create a new instance of the Source Generator and simply call the method generateSource(Schema schema, String packageName)

- **I want to use another implementation of the SAX Parser but Castor seems to rely on Xerces, is there a way to specify another parser?**

Castor is not relying on the SAX Parser implementation of Xerces. You can use another implementation by setting the following property in the castor.properties file:

```
# Defines the default XML parser to be used by castor
# The parser must implement org.xml.sax.Parser
#
org.exolab.castor.parser=org.apache.xerces.parsers.SAXParser
```

Castor relies on Xerces only for the serializer.

- **Is it possible to generate comments in the generated source ?**

If you want to insert comments on a class or element, you can use the ANNOTATION tag in the XML Schema.

The Source Generator is generating javadoc-style comments for all the contents of an ANNOTATION tag.

For instance in the example of section 3, the following:

```
<annotation>
      <documentation>
           This is a test XML Schema for Castor XML.
      </documentation>
            </annotation>
```

generates:

```
/**
 *
 * A simple representation of an invoice
 *
 * @version $Revision$ $Date$
**/
```

5 Open Issues: possible improvements

- Propose a Fine grained control on each class
- Provide a target (in Ant) so that we can compile directly the generated classes
- Provide an XML input file to control some properties:
    - Choose the name of the generated files
    - Decide what is the super class of the generated class
    - Decide what is the type of the generated classes (abstract, final, private…)
- Provide a stylesheet to convert old schemas to new ones
- Plug with a GUI schema editor

6 Glossary

[1] DOM (Document Object Model)

Document Object Model provides a standard set of objects for representing and manipulating HTML and XML documents.

[2] SAX (Simple API for XML)

SAX is a standard interface for event-based XML parsing.

[3] XML (Extensible Markup Language)

The Extensible Markup Language (XML) is a subset of SGML. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML.

[4] XML Schema

An XML Schema is a specific XML language that describes the structure and the types of an XML document.

[5] XML Data binding

Representing an XML document directly in-memory.

[6] Marshalling Framework

The marshalling framework is responsible for doing the conversion between Java and XML.

7 References

W3C XML SCHEMA

XML Schema Part 1: Structures

XML Schema Part 2: Datatypes

W3C Candidate Recommendation 24 October 2000

See http://www.w3.org/TR/2000/CR-xmlschema-1-20001024

http://www.w3.org/TR/2000/CR-xmlschema-2-20001024

SAX 2.0

Simple API for XML Version 2.0

David Megginson et al. 5th May 2000

See http://www.megginson.com/SAX/

Sun XML Data Binding WhitePapers

An XML Data Binding Facility for the Java Platform.

Sun Microsystems, Inc. 30 July 1999.

See http://java.sun.com/xml/docs/bind.pdf

Castor XML

Castor XML.

The Exolab group.

See http://castor.exolab.org