XML2J XML Generator

USER MANUAL VERSION 2.4

Table of Contents

1. Overview

This document describes the steps to set up the XML2J XML Generator v. 2.4 and the steps required generating code using the generator.

2. Prerequisites

To use the XML2J XML Generator 2.4:

- You MUST have Java 1.7 or later. For backward 1.5 compatibility contact support (only small differences, particularly the Java multi-catch statement).
- You MUST have a valid license for the XML2J XML Framework 2.4 (this may be an evaluation license)
- You MUST have Saxon XSLT and XQuery Processor: SAXON 6.5.5, which can be obtained here: http://prdownloads.sourceforge.net/saxon/saxon6-5-5.zip

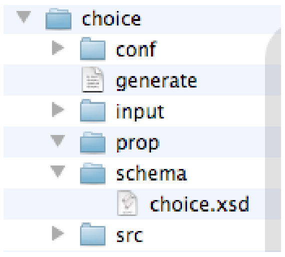
3. Preparation

Unpack the file **xml2j.zip** into a local directory on the development workstation.
Define an environment variable **XML2J_HOME**. This must point to the root of your XML2J installation.

After unpacking (seen from the directory in which you unpacked the zip file) you will see the following directory structure:

conf	Contains the specification of the generator configuration (XSD). This file is used by the generator to detect errors in the configuration file.
doc	The documentation for the XML2J Generator.
docs	Reference documentation (HTML).
samples	Set of samples (iso20022) illustrating the code generator.
tutorial	A set of examples illustrating the capabilities/features of the XML2J Generator.
xml2j.jar	The generator binary

The Tutorial directory contains a set of examples, designed to illustrate how the code generator maps XML onto Java. Each subdirectory follows the same structure illustrated by the *choice* example:



conf	The configuration for the XML2J Generator
docs	The optionally generated Java reference documentation
input	Sample input files (XML)
prop	Property file(s) for this example, used by the example program

schema	The XML Schema, which is both used for validation and by the generator
src	The generated source code

In some of the examples an ANT build file is provided within the respective sub-directory. We provided a generate batch file that generates all the sources directly under Tutorial (in a *src* sub-directory).

4. Usage

4.1. Command Line Options

4.1.1. Custom Header -h

The header the code generator inserts in the generated source files can be customized, using the -hfilename commandline option.

E.g. **java -jar xml2jg.jar -w./tutorial/choice -cconf/cfg.xml -s3 -hcustom.txt**

```
/*******************************************************************************
  ---------------------------------------------------------------------------
  XML2J XML to Java code generator
  ---------------------------------------------------------------------------
  +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++
  CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER
  CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER
  +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++

  This code was generated using xml2jg v. 2.2.0
  Generated code is compatible with xml2j-framework v. 2.2
  License: Lolke B. Dijkstra
  Module: CHOICE
  Generation date: Sat Feb 14 12:12:04 CET 2015

*******************************************************************************/
```

In the example above, the file custom.txt contains the following:

```
  +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++
  CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER
  CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER * CUSTOM HEADER
  +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++
```

4.1.2. Custom serialVersionUID -s

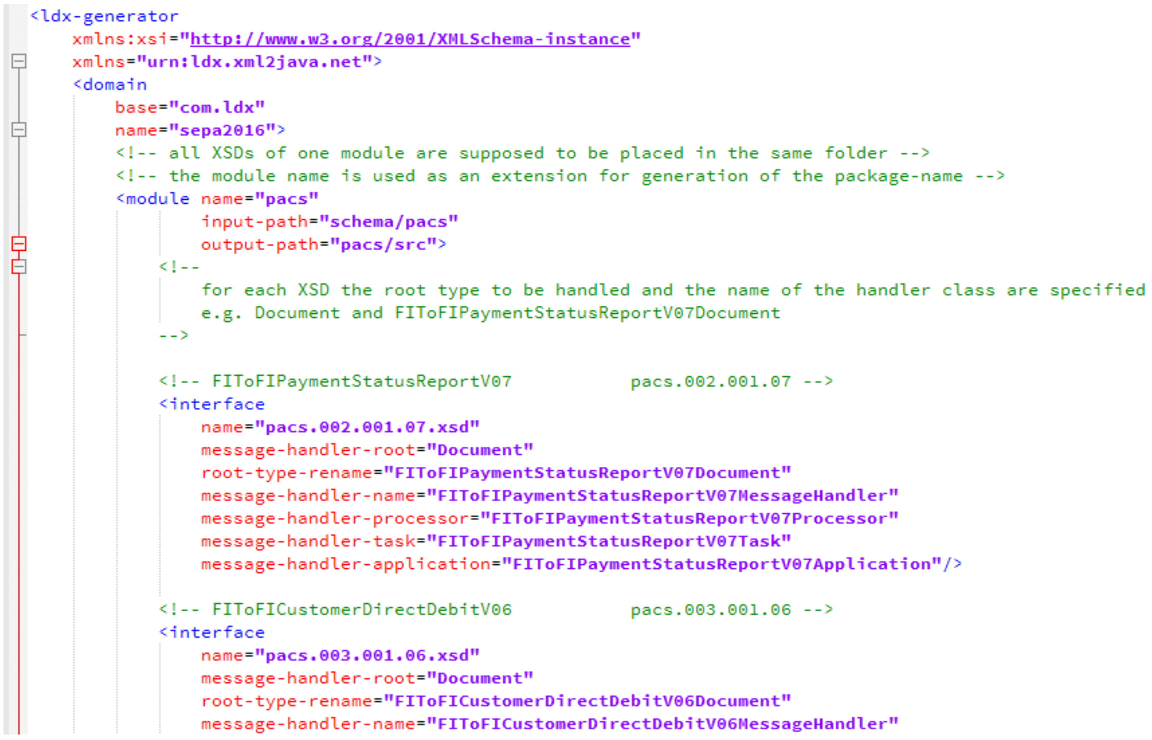
The option -s followed by a number causes the generator to use that number as the serialVersionUID. The above command would generate:

```
        /**
         * default serial version UID
         */
        private static final long serialVersionUID = 3L;
```

in the source code of the generated classes. This enables the use of difference serialVersionUIDs for different versions of the underlying XML schema.

4.2. Configuration

Before you can use the generator to generate code for your project you need to configure it. The configuration file is shown in *figure 1*.



```
<ldx-generator
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="urn:ldx.xml2java.net">
    <domain
        base="com.ldx"
        name="sepa2016">
        <!-- all XSDs of one module are supposed to be placed in the same folder -->
        <!-- the module name is used as an extension for generation of the package-name -->
        <module name="pacs"
                input-path="schema/pacs"
                output-path="pacs/src">
            <!--
                for each XSD the root type to be handled and the name of the handler class are specified
                e.g. Document and FIToFIPaymentStatusReportV07Document
            -->

            <!-- FIToFIPaymentStatusReportV07           pacs.002.001.07 -->
            <interface
                name="pacs.002.001.07.xsd"
                message-handler-root="Document"
                root-type-rename="FIToFIPaymentStatusReportV07Document"
                message-handler-name="FIToFIPaymentStatusReportV07MessageHandler"
                message-handler-processor="FIToFIPaymentStatusReportV07Processor"
                message-handler-task="FIToFIPaymentStatusReportV07Task"
                message-handler-application="FIToFIPaymentStatusReportV07Application"/>

            <!-- FIToFICustomerDirectDebitV06           pacs.003.001.06 -->
            <interface
                name="pacs.003.001.06.xsd"
                message-handler-root="Document"
                root-type-rename="FIToFICustomerDirectDebitV06Document"
                message-handler-name="FIToFICustomerDirectDebitV06MessageHandler"
```

Figure 1 – Generator Configuration

To detect errors made during editing, the parser checks the configuration file against the configuration XSD (conf/xml2j.xsd).

The ***domain*** indicates the business domain to which your project belongs. Domain attribute *name* is appended to the Java package name (whose base is **com.xml2j** by default) and applies for all contained elements. The *base* package 'com.xml2j' can be changed by using the *base* attribute. Using the instructions shown in figure 1 the generator generates ***com.xyz***. Per configuration file the generator supports only one domain. To generate code for multiple domains, use multiple configuration files. Both *domain.base* and *domain.name* are optional; if *name* is not present the package is the base package (in this case **com.xyz**).

The ***module*** indicates the sub-domain within the domain. The package in which the *classes* are generated is given by the Java package name followed by the module attribute *name* (e.g. com.xyz.tutorial.***ext***). The *messageHandlers* are generated in a subpackage *handlers* e.g. com.xyz.tutorial.ext.***handlers***).

For each **module** the following information must be provided:

name	the *module* name (see above)
input-path	the directory where the generator looks for the *schemata* (one per interface)

output-path	the directory where the generator outputs the *source code*

A module has one or more **interfaces**, each of which is specified by the **interface** element.

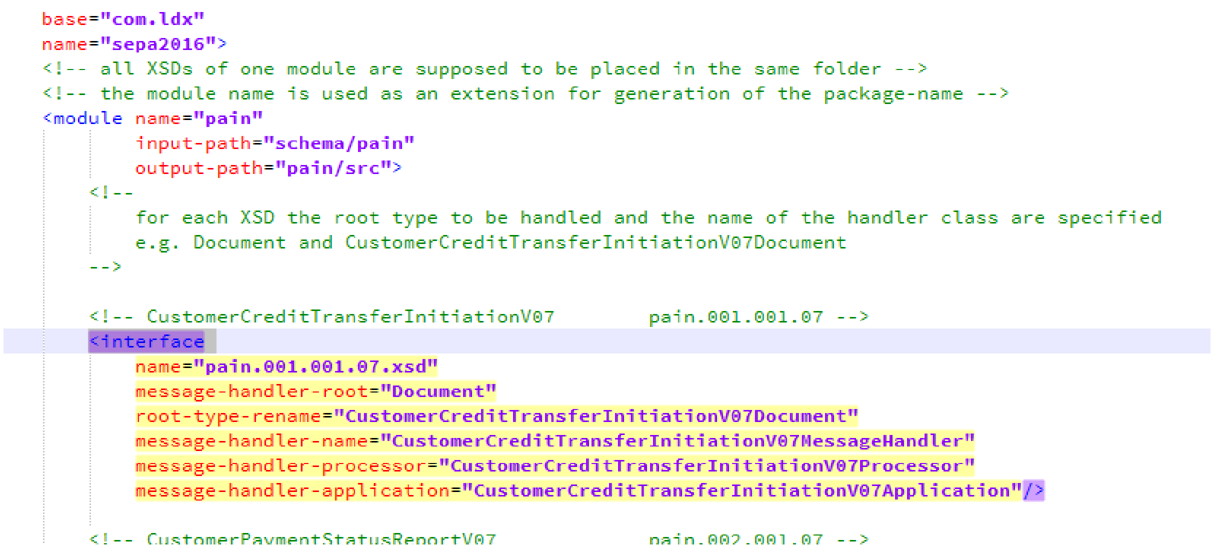
➔ **Each interface relates to precisely one schema file (XSD)**.

For each **interface** the following information is provided:

name	The name of the corresponding *schema* (XSD).
message-handler-root	The name of the XML *element* that is the root element of the instance document.
root-type-rename (*optional*)	The name the *type* the generator will substitute for the root type. It is used to prevent type name collisions. (This will also change the root-type handler since it is derived from the root-type by appending *Handler* to the name.) For example within SEPA for all interfaces (XML Schema files) the *type* of the root element is '*Document*' (not to be confused with the *name* of the root element, which also happens to be Document). Figure 2 shows it is renamed to be unique. For more information please refer to section 5.4.
message-handler-name	The name of the generated *messageHandler* class. This class reads the XML document from an input source.
message-handler-processor (*optional*)	The name of the (optionally) generated *MessageProcessor* class.
message-handler-runnable	The name (optionally) of the runnable class that is generated. The runnable class extends `ParserRunnable`, which implements `Runnable`.
message-handler-application-task	The name (optionally) of the application task class.
message-handler-application (*optional*)	The name of the (optionally) generated *ParserApplication* class. • If message-handler-application is not present, no main is generated. • If both runnable and application are specified, the generator generates a sample main using multi-threading. • If both message-handler-application-task and message-handler-application are present, the generator generates a sample main with no threading. • If only message-handler-application is present (no runnable and no application-task) then the generator assumes message-handler-application-task = message-handler-application + "Task"

The *messageHandler* automatically positions at the message-handler-root (which is the first element in the hierarchy that is processed).

As of version 2.4 the application class does no longer contain the generated main function, instead the main is generated as part of the applicationMain class.



```
base="com.ldx"
name="sepa2016">
<!-- all XSDs of one module are supposed to be placed in the same folder -->
<!-- the module name is used as an extension for generation of the package-name -->
<module name="pain"
        input-path="schema/pain"
        output-path="pain/src">
    <!--
        for each XSD the root type to be handled and the name of the handler class are specified
        e.g. Document and CustomerCreditTransferInitiationV07Document
    -->

    <!-- CustomerCreditTransferInitiationV07          pain.001.001.07 -->
    <interface
        name="pain.001.001.07.xsd"
        message-handler-root="Document"
        root-type-rename="CustomerCreditTransferInitiationV07Document"
        message-handler-name="CustomerCreditTransferInitiationV07MessageHandler"
        message-handler-processor="CustomerCreditTransferInitiationV07Processor"
        message-handler-application="CustomerCreditTransferInitiationV07Application"/>

    <!-- CustomerPaymentStatusReportV07               pain.002.001.07 -->
```

Figure 2 – Generator Configuration

According to the example configuration, the XML2J Generator reads the schemata from the subdirectory *sepa/schema* and generates the sources into subdirectory *sepa/pain/src*.

The highlighted section in **Figure 2** illustrates the usage of the optional *message-handler-application* and *message-handler-processor* attributes. In this case (because runnable is not specified) the main application is generated using a just the main thread. A main task is *assumed* (because message-handler-application-task is not specified) to take the name CustomerCreditTransferInitiationV07ApplicationTask.

4.3. Configuration Examples

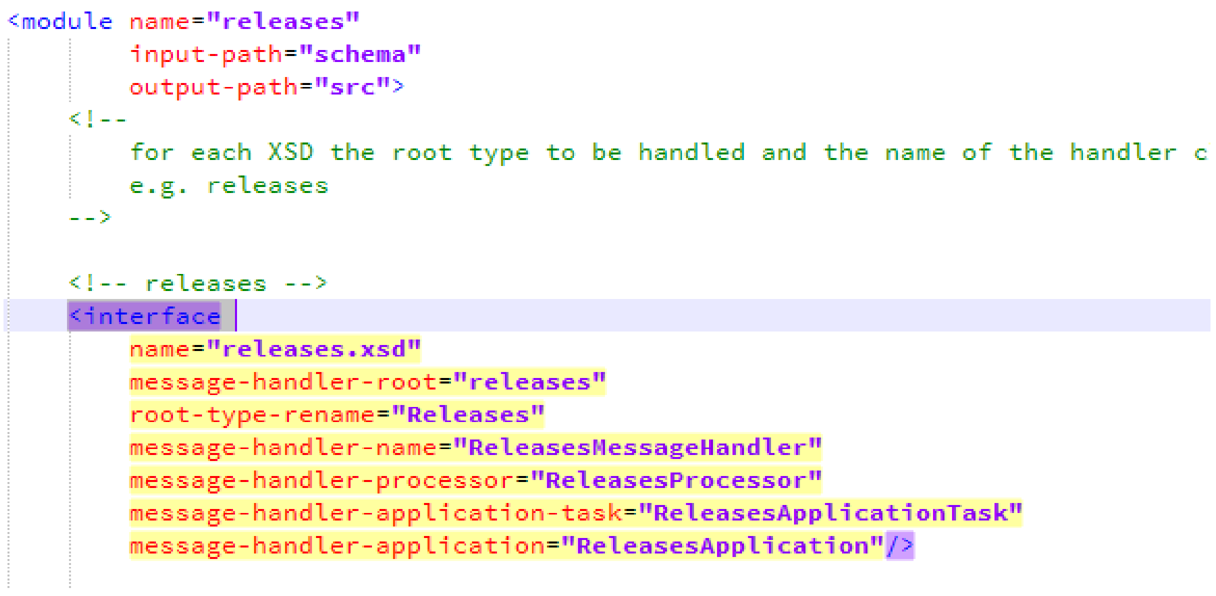
Example 1:

Generate code for all of the pain messages in a package named *com.mycompany.sepa.payments*.
To do this domain@*base* would be *'çom.mycompany' and* module@*name* would be *'payments'* instead of *'pain'*.

Example 2:

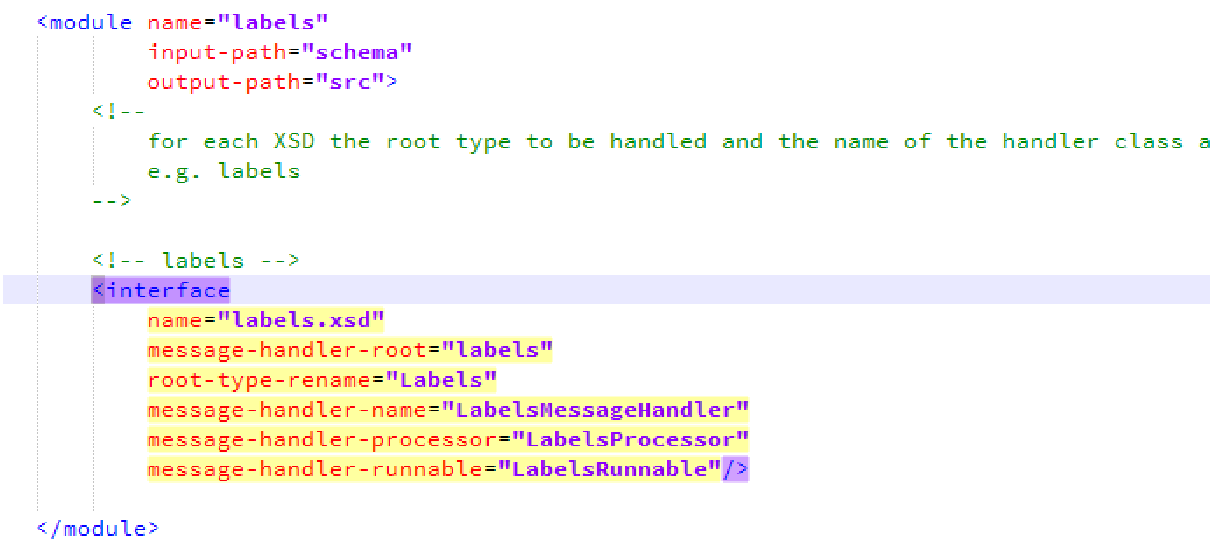
Generate all of the pain and pacs messages in a package named *com.xml2j.sepa.payments*.
To do this for each of the pacs XSD files an interface element must be added to the configuration file.

Three more examples:



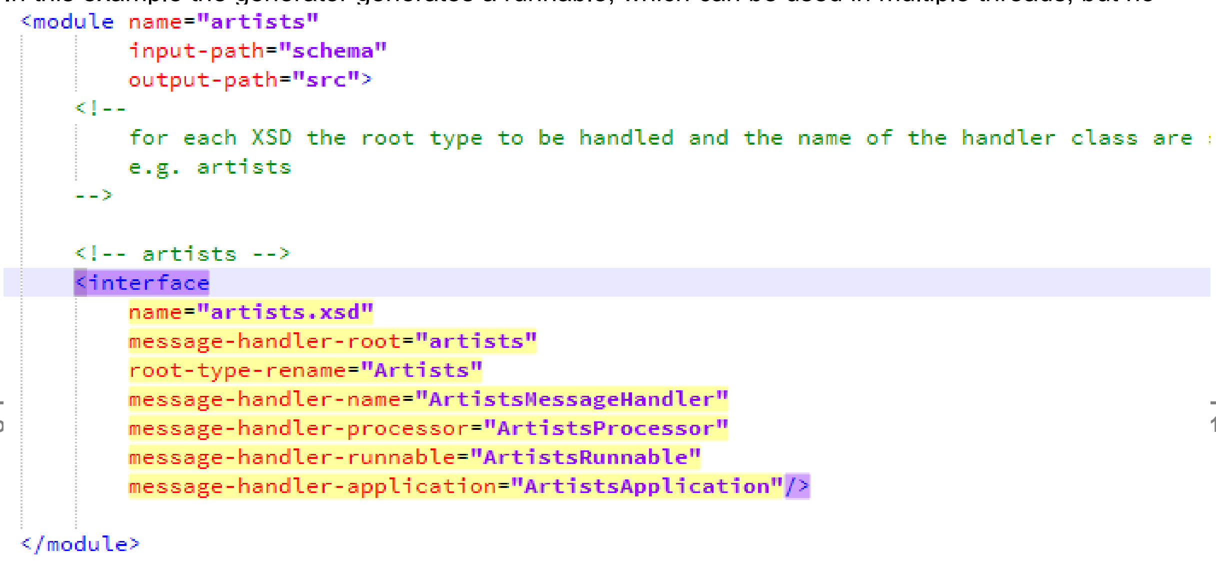
```
<module name="releases"
        input-path="schema"
        output-path="src">
    <!--
        for each XSD the root type to be handled and the name of the handler c
        e.g. releases
    -->

    <!-- releases -->
    <interface
        name="releases.xsd"
        message-handler-root="releases"
        root-type-rename="Releases"
        message-handler-name="ReleasesMessageHandler"
        message-handler-processor="ReleasesProcessor"
        message-handler-application-task="ReleasesApplicationTask"
        message-handler-application="ReleasesApplication"/>
```

In this example both message-handler-application and message-handler-application-task are specified, the generator generates the application-task and main with the task directly used within the main thread.



```
<module name="labels"
        input-path="schema"
        output-path="src">
    <!--
        for each XSD the root type to be handled and the name of the handler class a
        e.g. labels
    -->

    <!-- labels -->
    <interface
        name="labels.xsd"
        message-handler-root="labels"
        root-type-rename="Labels"
        message-handler-name="LabelsMessageHandler"
        message-handler-processor="LabelsProcessor"
        message-handler-runnable="LabelsRunnable"/>

</module>
```

In this example the generator generates a runnable, which can be used in multiple threads, but no



```
<module name="artists"
        input-path="schema"
        output-path="src">
    <!--
        for each XSD the root type to be handled and the name of the handler class are
        e.g. artists
    -->

    <!-- artists -->
    <interface
        name="artists.xsd"
        message-handler-root="artists"
        root-type-rename="Artists"
        message-handler-name="ArtistsMessageHandler"
        message-handler-processor="ArtistsProcessor"
        message-handler-runnable="ArtistsRunnable"
        message-handler-application="ArtistsApplication"/>

</module>
```

The xml2j.jar is the generator. The generator takes the following options:

-c	the name of the configuration the generator should use
-p	instructs the code generator to include the methods for printing (the default is to exclude them)

XML2J_HOME which is to be set to the path where XML2J is installed.

4.5. MessageHandler

The *message-handler-root* tells the generator which element is the root element. By default the generator uses the corresponding type name from the XML schema to generate the Java data class and Java handler class. However, in the case of SEPA in every XML schema the root type is *Document*. Consequently, following the default, the generator would *overwrite* the messageHandler for every schema (i.e. pain.001, pain.002, etc.)! For SEPA the best way to prevent this from happening is by instructing the generator to use an *alternative name* for the type using the *root-type-rename* attribute. For example (fig. 1) the generator substitutes *Pain001V03Document* for type *Document* in the case of pain.001.001.03.xsd. This is because within the standard all types have unique names, except for the root type. An alternative often used is a different module for each interface/schema. The module's name is then used to create the fully qualified name of the class.

5. Example

In this section we illustrate different runtime configurations by examining the *subst* example. You can find this example in the tutorials subdirectory.

5.1. The subst XML schema

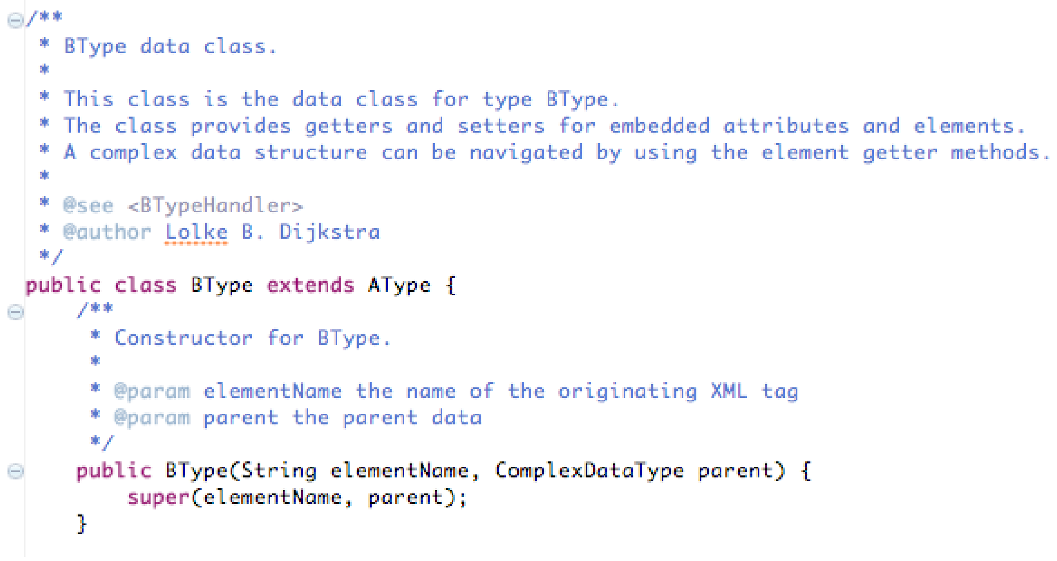
```
<xsd:complexType name="AContainedType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="AType">
  <xsd:sequence>
    <xsd:element name="first" type="xsd:string"/>
    <xsd:element name="containedA" type="AContainedType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="BType">
<xsd:complexContent>
  <xsd:extension base="AType">
  <xsd:sequence>
    <xsd:element name="second" type="xsd:string"/>
  </xsd:sequence>
  </xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="CType">
<xsd:complexContent>
  <xsd:extension base="BType">
  <xsd:sequence>
    <xsd:element name="third" type="xsd:string"/>
  </xsd:sequence>
  </xsd:extension>
</xsd:complexContent>
</xsd:complexType>
```

The above excerpt illustrates the family of types *AType*, *BType* and *CType* of which *AType* is the super type. In Java we modelled this using an extends relationship.

```
<xsd:element name="A" type="AType" />
<xsd:element name="B" substitutionGroup="A" type="BType" />
<xsd:element name="C" substitutionGroup="B" type="CType" />

<xsd:element name="container">
<xsd:complexType>
  <xsd:sequence>
    <xsd:element ref="A" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

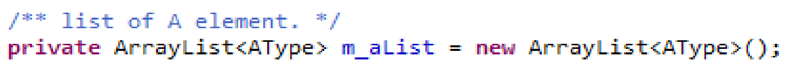
The element declaration section tells the Schema processor that B and C may be substituted wherever A occurs. The precondition for such element substitution being that the corresponding sub types are extensions from the base type.

The final *container* type embeds a collection of A elements.
Now let’s have a quick look at the generated Java code:



```
/**
 * BType data class.
 *
 * This class is the data class for type BType.
 * The class provides getters and setters for embedded attributes and elements.
 * A complex data structure can be navigated by using the element getter methods.
 *
 * @see <BTypeHandler>
 * @author Lolke B. Dijkstra
 */
public class BType extends AType {
    /**
     * Constructor for BType.
     *
     * @param elementName the name of the originating XML tag
     * @param parent the parent data
     */
    public BType(String elementName, ComplexDataType parent) {
        super(elementName, parent);
    }
```

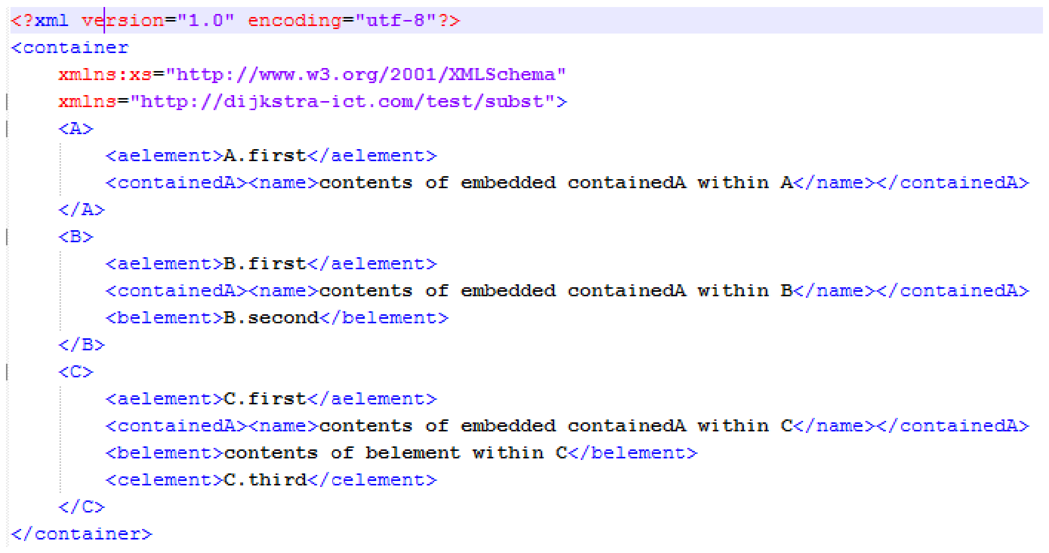
We find that the *BType extends AType*. Further within the container class there is a list of *AType*, resembling the XML Schema model.



```
    /** list of A element. */
    private ArrayList<AType> m_aList = new ArrayList<AType>();
```

5.2. Input and runtime configuration

Now we have a look at the *input*:



```
<?xml version="1.0" encoding="utf-8"?>
<container
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://dijkstra-ict.com/test/subst">
    <A>
        <aelement>A.first</aelement>
        <containedA><name>contents of embedded containedA within A</name></containedA>
    </A>
    <B>
        <aelement>B.first</aelement>
        <containedA><name>contents of embedded containedA within B</name></containedA>
        <belement>B.second</belement>
    </B>
    <C>
        <aelement>C.first</aelement>
        <containedA><name>contents of embedded containedA within C</name></containedA>
        <belement>contents of belement within C</belement>
        <celement>C.third</celement>
    </C>
</container>
```

We see that we have a container with tree elements, one of each type. The processor we built just prints out the Java objects sent by the framework.

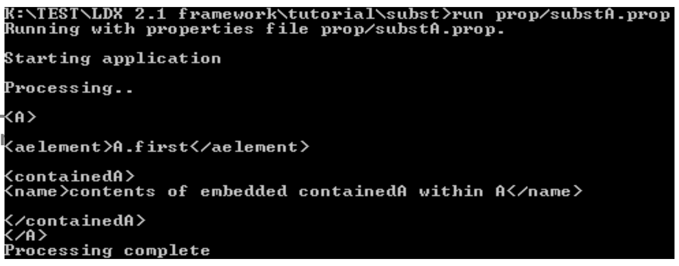
We can run this sample in a variety of ways:

5.2.1. Property file substA.prop

Configuration:

# process A

container/A/@detach=true



```
K:\TEST\LDX 2.1 framework\tutorial\subst>run prop/substA.prop
Running with properties file prop/substA.prop.

Starting application

Processing..

<A>

<aelement>A.first</aelement>

<containedA>
<name>contents of embedded containedA within A</name>

</containedA>
</A>
Processing complete
```

container/A/@process=true

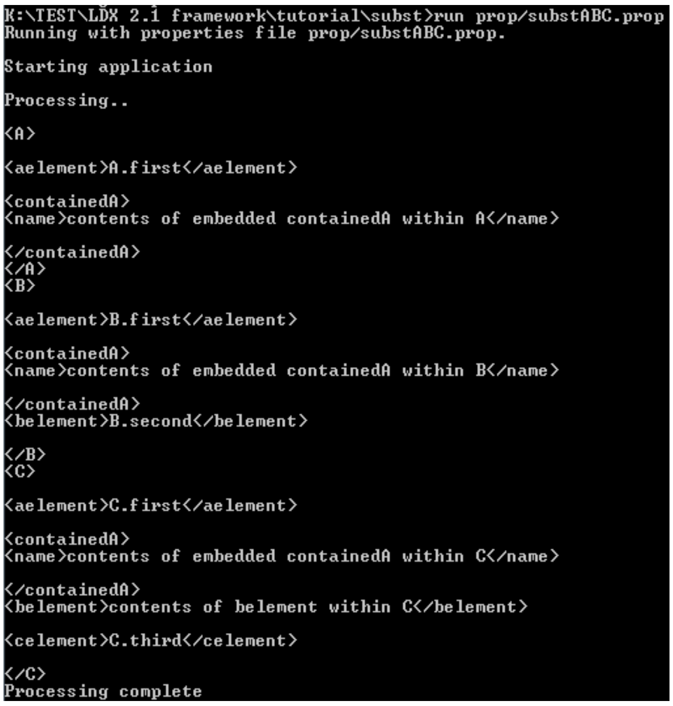
This tells the framework two things:
- "A" should be *detached* from its parent container
- "A" should be processed (that is sent to the processor)

Output:

5.2.2. Property file substABC.prop

Configuration:

```
# process A
container/A/@detach=true
container/A/@process=true

# process B
container/B/@detach=true
container/B/@process=true

# process C
container/C/@detach=true
container/C/@process=true
```

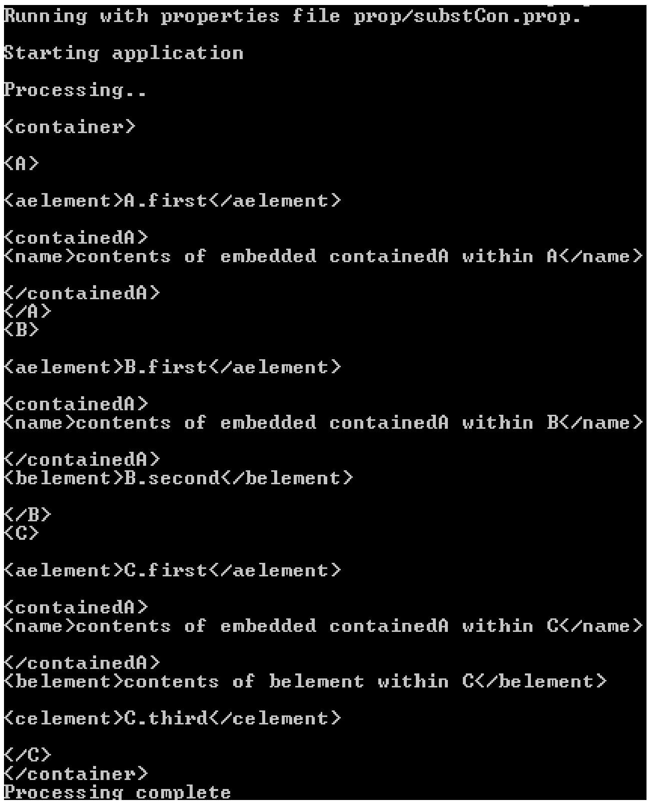


```
K:\TEST\LDX 2.1 framework\tutorial\subst>run prop/substABC.prop
Running with properties file prop/substABC.prop.

Starting application

Processing..

<A>

<aelement>A.first</aelement>

<containedA>
<name>contents of embedded containedA within A</name>

</containedA>
</A>
<B>

<aelement>B.first</aelement>

<containedA>
<name>contents of embedded containedA within B</name>

</containedA>
<belement>B.second</belement>

</B>
<C>

<aelement>C.first</aelement>

<containedA>
<name>contents of embedded containedA within C</name>

</containedA>
<belement>contents of belement within C</belement>

<celement>C.third</celement>

</C>
Processing complete
```

5.2.3. Property file substCon.prop

Configuration:
# process container
container/@process=true

Command:
Java -jar subst.jar input/subst.xml schema/subst.xsd prop/substCon.prop



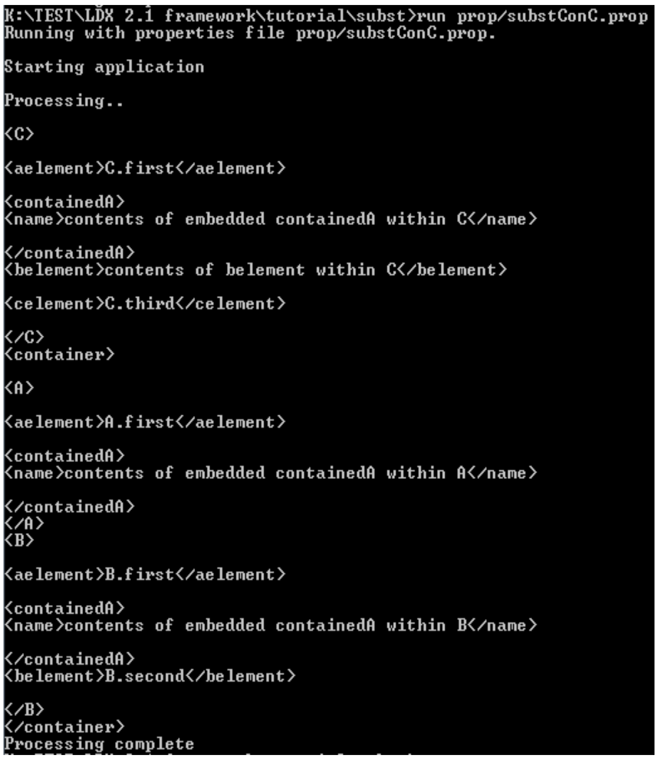
```
Running with properties file prop/substCon.prop.

Starting application

Processing..

<container>

<A>

<aelement>A.first</aelement>

<containedA>
<name>contents of embedded containedA within A</name>

</containedA>
</A>
<B>

<aelement>B.first</aelement>

<containedA>
<name>contents of embedded containedA within B</name>

</containedA>
<belement>B.second</belement>

</B>
<C>

<aelement>C.first</aelement>

<containedA>
<name>contents of embedded containedA within C</name>

</containedA>
<belement>contents of belement within C</belement>

<celement>C.third</celement>

</C>
</container>
Processing complete
```

5.2.4. Property file substConC.prop

Configuration:
# process container
container/@process=true

# process C
container/C/@detach=true
container/C/@process=true

This one is interesting: it tells the framework to process the container, and to process and detach C. Both A and B will be contained in the container, whereas C will be processed separately.



```
K:\TEST\LDX 2.1 framework\tutorial\subst>run prop/substConC.prop
Running with properties file prop/substConC.prop.

Starting application

Processing..

<C>

<aelement>C.first</aelement>

<containedA>
<name>contents of embedded containedA within C</name>

</containedA>
<belement>contents of belement within C</belement>

<celement>C.third</celement>

</C>
<container>

<A>

<aelement>A.first</aelement>

<containedA>
<name>contents of embedded containedA within A</name>

</containedA>
</A>
<B>

<aelement>B.first</aelement>

<containedA>
<name>contents of embedded containedA within B</name>

</containedA>
<belement>B.second</belement>

</B>
</container>
Processing complete
```

6. Features and Limitations

Currently *not* supported:

- XML *anyType, union, list*
- import within schemata
- targetNamespace other than the default