Beeklaan 444
2562 BK Den Haag
www.darwin-it.nl
info@darwin-it.nl
KvK 27283780
ING 65.35.40.663

DARWIN IT-PROFESSIONALS

Martien van den Akker
Technical Architect

Code Generation with XSLT

Introduction

Last week I read a nice article of my con-colleague Lucas Jellema about "PL/SQL Table-to-Java Bean (and Data to Java Bean Manager) generator - useful for data driven demos without database " at the AMIS-blog on:http://technology.amis.nl/blog/?p=3272.

In this blog Lucas writes how he generated java-code from Pl/Sql. It drew me back to a little project I did for myself and my former employer as a BusinessDevelopment-job. I wanted to make a simple method to enable a convential custom development application, like a Designer/Developer Forms application, like we build a million times in the past. Since it is primarly aimed for the hundreds of Forms applications at our customers that are primarly build with Designer and Headstart, I could do it in pl/sql also or even create a Headstart utility for it. But for the latter option I would learn myself Headstart-utility principals again, that did not look to useful for me. Pl/Sql would do. But I would have it flexible having the ability to generate several pieces of code from one source.

The problem with the code of Lucas, that is when you see it as a problem, is that all the code generation is in pl/sql. To change the outcome would mean that you should change your pl/sql code. I would like to have a more template-based solution.

A few years ago I wrote an article on "XML using Sql" for our monthly consulting paper at Oracle. See my post: http://darwin-it.blogspot.com/2008/07/xml-using-sql.html. With this knowledge it is easy to generate XML out of the datadictionary of the database. Then based on this xml you can generate all you want using … XSLT.

Let me show how I did the job.

Table definitions to XML

The table definitions are easily queried from the all_tables view from the datadictionary. The thing is only that we want to have the result in xml. For this Oracle has created XDB with its XMLtype and XML-Sql functions. So the query would look like the following:

```
select tbl.owner
     ,      tbl.table_name
     ,      xmlelement( "TableDefinition"
                      , xmlelement("Name", tbl.table_name)
                      ) xml
     from all_tables tbl
     where tbl.owner      = bTableOwner
     and   tbl.table_name = bTableName
```

```
    group by tbl.owner, tbl.table_name
  ;
```

This simple query gives the following xml:

```
<TableDefinition><Name>HSD_EMPLOYEES</Name></TableDefinition>
```

You see that the trick is to add the 'xmlelement' function that delivers an xmltype, and it gets the elements name as a first parameter and then can get any number of xmltype parameters following it. Or a reference to a column.

Now we want to add the columns of the table. We want to have that in a hierarchical lower level. This is quite hard to put into one query. But also I like to modularize my queries in a fancy way. So let's first take a look on how we query just the columns.

```
select xmlelement( "Columns"
                        , xmlagg(xmlelement( "Column"
                                           , xmlelement("ColName", col.column_name)
                                           , xmlelement("ColId", col.column_id)
                                           , xmlelement("DataType", col.data_type)
                                           , xmlelement("DataLength", col.data_length)
                                           , xmlelement("DataPrecision", col.data_precision)
                                           , xmlelement("DataScale", col.data_scale)
                                           )
                                 )
                        ) xml
    from all_tab_cols col
    where col.owner      = 'HDEMO65'
    and   col.table_name =  'HSD_EMPLOYEES';
```

Here there is another function the 'xmlagg'. This function aggregates multiple rows into an xmltype. Doing so we get an element 'Columns' and within it per column a 'Column' element with its information.

Then we have to add the primary key information. This we will need to generate parameters for the constructor method in the object type, for example.
The query for that will look like:

```
select cnt.owner
       ,      cnt.table_name
       ,      cnt.constraint_name
       ,      xmlelement( "PrimaryKey"
                        , xmlelement("PrimaryKeyName", cnt.constraint_name)
                        , xmlelement( "PrimaryKeyColumns"
                                    , xmlagg( xmlelement("Column"
                                                       , xmlelement("Name", cln.column_name)
                                                       , xmlelement("Position", cln.position)
                                                       , xmlelement("DataType", tcl.data_type)
                                                       , xmlelement("DataLength", tcl.data_length)
                                                       , xmlelement("DataPrecision",
tcl.data_precision)
                                                       , xmlelement("DataScale", tcl.data_scale)
                                                       )
                                            )
                                    )
                        ) xml
       from all_constraints cnt
       ,    all_cons_columns cln
       ,    all_tab_columns  tcl
       where cnt.constraint_type = 'P'
       and   cln.owner           = cnt.owner
       and   cln.table_name      = cnt.table_name
       and   cln.constraint_name = cnt.constraint_name
       and   tcl.table_name      = cln.table_name
       and   tcl.owner           = cln.owner
       and   tcl.column_name     = cln.column_name
```

```
      and   cnt.owner           = b_Owner
      and   cnt.table_name      =  b_TableName
      group by cnt.owner, cnt.table_name, cnt.constraint_name;
```

I think this isn't to complicated either. What you basically have to know is that we need the constraints of type 'P' (primary key) and the related columns. For these columns we will need the datatype information again. Strictly speaking you could say that this is already embedded in the columns-part. But it is easier for the xslt's to add the info here again.

The queries I put into a few functions in a package. Then the overall query will look like:

```
select tbl.owner
     ,      tbl.table_name
     ,      xmlelement( "TableDefinition"
                      , xmlelement("Name", tbl.table_name)
                      , xxx_table_definitions.TabColumns(tbl.owner, tbl.table_name)
                      , xxx_table_definitions.PKColumns(tbl.owner, tbl.table_name)
                      ) xml
     from all_tables tbl
     where tbl.owner      = bTableOwner
     and   tbl.table_name = bTableName
     group by tbl.owner, tbl.table_name
```

Then the xml will look like as follows. I reformatted it for convenience using jDeveloper:

```
<TableDefinition>
  <Name>HSD_EMPLOYEES</Name>
  <Columns>
    <Column>
      <ColName>CREATED_BY</ColName>
      <ColId>21</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>30</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>CREATION_DATE</ColName>
      <ColId>20</ColId>
      <DataType>DATE</DataType>
      <DataLength>7</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>LAST_UPDATE_DATE</ColName>
      <ColId>19</ColId>
      <DataType>DATE</DataType>
      <DataLength>7</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>LAST_UPDATED_BY</ColName>
      <ColId>18</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>30</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>CIVIL_STATE</ColName>
      <ColId>17</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>1</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
```

```
      <ColName>BANK_ACCOUNT</ColName>
      <ColId>16</ColId>
      <DataType>NUMBER</DataType>
      <DataLength>22</DataLength>
      <DataPrecision>15</DataPrecision>
      <DataScale>0</DataScale>
    </Column>
    <Column>
      <ColName>E_MAIL</ColName>
      <ColId>15</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>30</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>TELEPHONE</ColName>
      <ColId>14</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>20</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>COUNTRY</ColName>
      <ColId>13</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>24</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>ZIP_CODE</ColName>
      <ColId>12</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>10</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>CITY</ColName>
      <ColId>11</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>24</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>STREET</ColName>
      <ColId>10</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>24</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>COMMISSION</ColName>
      <ColId>9</ColId>
      <DataType>NUMBER</DataType>
      <DataLength>22</DataLength>
      <DataPrecision>7</DataPrecision>
      <DataScale>2</DataScale>
    </Column>
    <Column>
      <ColName>SALARY</ColName>
      <ColId>8</ColId>
      <DataType>NUMBER</DataType>
      <DataLength>22</DataLength>
      <DataPrecision>7</DataPrecision>
      <DataScale>2</DataScale>
    </Column>
    <Column>
      <ColName>HIRE_DATE</ColName>
      <ColId>7</ColId>
```

```
      <DataType>DATE</DataType>
      <DataLength>7</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>JOB</ColName>
      <ColId>6</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>9</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>NAME</ColName>
      <ColId>5</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>10</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>EMP_ID</ColName>
      <ColId>4</ColId>
      <DataType>NUMBER</DataType>
      <DataLength>22</DataLength>
      <DataPrecision>4</DataPrecision>
      <DataScale>0</DataScale>
    </Column>
    <Column>
      <ColName>DEP_ID</ColName>
      <ColId>3</ColId>
      <DataType>NUMBER</DataType>
      <DataLength>22</DataLength>
      <DataPrecision>2</DataPrecision>
      <DataScale>0</DataScale>
    </Column>
    <Column>
      <ColName>ORACLE_USERNAME</ColName>
      <ColId>2</ColId>
      <DataType>VARCHAR2</DataType>
      <DataLength>30</DataLength>
      <DataPrecision></DataPrecision>
      <DataScale></DataScale>
    </Column>
    <Column>
      <ColName>ID</ColName>
      <ColId>1</ColId>
      <DataType>NUMBER</DataType>
      <DataLength>22</DataLength>
      <DataPrecision>4</DataPrecision>
      <DataScale>0</DataScale>
    </Column>
  </Columns>
  <PrimaryKey>
    <PrimaryKeyName>HSD_EMP_PK</PrimaryKeyName>
    <PrimaryKeyColumns>
      <Column>
        <Name>ID</Name>
        <Position>1</Position>
        <DataType>NUMBER</DataType>
        <DataLength>22</DataLength>
        <DataPrecision>4</DataPrecision>
        <DataScale>0</DataScale>
      </Column>
    </PrimaryKeyColumns>
  </PrimaryKey>
</TableDefinition>
```

For your convenience here is the code of the package body (I take it that you can create the specification out of it your self:

```
create or replace package body xxx_table_definitions is

-- Author  : Martien van den Akker, Darwin-it Professionals
-- Created : 15-7-2008
-- Purpose : Get table definitions in XML
function getTableDefAsXML( pTableOwner varchar2
                         , pTableName varchar2)
return clob is
  Result clob;
  cursor cTbl ( bTableOwner varchar2
              , bTableName varchar2)
  is select tbl.owner
     ,      tbl.table_name
     ,      xmlelement( "TableDefinition"
                      , xmlelement("Name", tbl.table_name)
                      , xxx_table_definitions.TabColumns(tbl.owner, tbl.table_name)
                      , xxx_table_definitions.PKColumns(tbl.owner, tbl.table_name)
                      ) xml
     from all_tables tbl
     where tbl.owner      = bTableOwner
     and   tbl.table_name = bTableName
     group by tbl.owner, tbl.table_name
  ;
  rTbl cTbl%RowType;
begin
  open cTbl( bTableOwner => pTableOwner
           , bTableName  => pTableName);
  fetch cTbl into rTbl;
  if cTbl%found
  then
    Result := '<?xml version="1.0"?>'||chr(10);
    dbms_lob.append(src_lob =>  rTbl.xml.getClobVal
                   , dest_lob => Result);
  else
    Result := null;
  end if;
  return(Result);
end getTableDefAsXML;

  function TabColumns( p_Owner     varchar2
                     , p_TableName varchar2)
  return xmltype
  is
    cursor c_xml( b_Owner     varchar2
                , b_TableName varchar2)
    is select /*col.owner
       ,      col.table_name
       ,      */xmlelement( "Columns"
                          , xmlagg(xmlelement( "Column"
                                             , xmlelement("ColName", col.column_name)
                                             , xmlelement("ColId", col.column_id)
                                             , xmlelement("DataType", col.data_type)
                                             , xmlelement("DataLength", col.data_length)
                                             , xmlelement("DataPrecision", col.data_precision)
                                             , xmlelement("DataScale", col.data_scale)
                                             )
                                   )
                          ) xml
    from all_tab_cols col
    where col.owner      = b_Owner
    and   col.table_name =  b_TableName
--    group by col.owner, col.table_name;
order by col.column_id desc;
    r_xml c_xml%rowtype;
  begin
    open c_xml( b_Owner     => p_Owner
              , b_TableName => p_TableName);
    fetch c_xml into r_xml;
    if c_xml%notfound
    then
      raise no_data_found;
    end if;
    close c_xml;
    return(r_xml.xml);
```

```
  exception
    when others then
      if c_xml%isopen then close c_xml; end if;
      raise;
  end;
  function PKColumns( p_Owner     varchar2
                    , p_TableName varchar2)
  return xmltype
  is
    cursor c_xml( b_Owner     varchar2
                , b_TableName varchar2)
    is select cnt.owner
       ,      cnt.table_name
       ,      cnt.constraint_name
       ,      xmlelement( "PrimaryKey"
                        , xmlelement("PrimaryKeyName", cnt.constraint_name)
                        , xmlelement( "PrimaryKeyColumns"
                                    , xmlagg( xmlelement("Column"
                                                         , xmlelement("Name", cln.column_name)
                                                         , xmlelement("Position", cln.position)
                                                         , xmlelement("DataType", tcl.data_type)
                                                         , xmlelement("DataLength", tcl.data_length)
                                                         , xmlelement("DataPrecision",
tcl.data_precision)
                                                         , xmlelement("DataScale", tcl.data_scale)
                                                         )
                                            )
                                    )
                        ) xml
       from all_constraints cnt
       ,    all_cons_columns cln
       ,    all_tab_columns  tcl
       where cnt.constraint_type = 'P'
       and   cln.owner           = cnt.owner
       and   cln.table_name      = cnt.table_name
       and   cln.constraint_name = cnt.constraint_name
       and   tcl.table_name      = cln.table_name
       and   tcl.owner           = cln.owner
       and   tcl.column_name     = cln.column_name
       and   cnt.owner           = b_Owner
       and   cnt.table_name      =  b_TableName
       group by cnt.owner, cnt.table_name, cnt.constraint_name;
    r_xml c_xml%rowtype;
  begin
    open c_xml( b_Owner     => p_Owner
              , b_TableName => p_TableName);
    fetch c_xml into r_xml;
    if c_xml%notfound
    then
      raise no_data_found;
    end if;
    close c_xml;
    return(r_xml.xml);
  exception
    when others then
      if c_xml%isopen then close c_xml; end if;
      raise;
  end;
begin
  -- Initialization
  null;
end xxx_table_definitions;
```

XSLT for generation

Having all the definitions in XML, the code generation can begin. All you need is a code editor and an xml-parser that can do intermediate transformations to test your stylesheets.

You can use my XMLTester tool that you can download at: www.darwin-

it.nl/downloads/xmltester_v0.1.zip. In the article XSLT in Java with Oracle Parser at http://darwin-it.blogspot.com/2008/06/xslt-in-java-with-oracle-parser.html, I explained how I did it in Java.

Later in this article I'll explain how to do the same thing in the database.

Basic Template

I'm not going to give a complete XSLT course here. But just the basics. Here comes a basic template:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- Root template -->
  <xsl:template match="/">


  </xsl:template>
</xsl:stylesheet>
```

This is what you get if you create a new "XSL Style Sheet" in jDeveloper. It is actually doing nothing, yet. But you see it is in fact a pretty simple xml-document. And it starts with a template with in the match attribute an xpath-expression that queries the root-element of the xml-document that has to be transformed. Important for us is to add an output style:

```
<xsl:output method="text" indent="no"/>
```

It denotes that out of the transformation we expect plain ascii text, since we generate free-formatted source (not xml or html).

Traversing the XML

There are basically two ways in XSL to traverse through your source-xml and build up your XSL to generate the output. The most straightforward is using a "xsl:for-each":

```
  <xsl:for-each select="/ns1:SOAOrderBookingProcessRequest/po:PurchaseOrder/po:OrderItems/po:Item">
```

within this xml-element you can query the elements from the source xml using a <xsl:value-of select="po:price"/> element.
Advantage is that you can do an "order by" by adding an <xsl:sort > element as a first child element. This is the way you would do it when using the Transformation-tool in the BPEL and ESB desginer of jDeveloper.

But for building up my code generating stylesheets I prefer the other method: using sub-templates.
In the basic stylesheet you already see a first template. But you can add as many as you like. I mostly add at least a template per "functional level" in the source xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="text" indent="no"/>
 <!-- Main template -->
 <xsl:template match="/">
  <xsl:apply-templates select="/TableDefinition" mode="ObjectType"/>
 </xsl:template>
 <!-- Template to create the object type -->
 <xsl:template name="ObjectType" match="/TableDefinition" mode="ObjectType">
  <xsl:call-template name="CreateType"/>
  <xsl:call-template name="CloseType"/>
 </xsl:template>
 <xsl:template name="CreateType">
 <!-- Declaration of the Type -->
  <xsl:value-of select="concat('create or replace type ', Name)"/>
```

```
 </xsl:template>
 <!-- Close the Type -->
 <xsl:template name="CloseType">
  <xsl:text>
);</xsl:text>
 </xsl:template>
</xsl:stylesheet>
```

What you see is that in the "main template" the command "apply-templates" is added with an Xpath expression. This states that every template that matches the xpath expression is executed. In this case the xsl-processor will find a template called "ObjectType" with a match-attribute that matches the select attribute of the "apply-templates" command.

There are also a third and a fourth template that do the declaration and the closing of the type. These templates are called with "xsl:call-template" and can just contain text or any complex templating structure containing xpath-expressions starting from the node that is selected at the calling template. In the CreateType template there is a "value-of" command with a "select"-attribute denoting a concatenation of "create or replace type" and the Name of the table. "Name" is a sub-node of "/TableDefinition" that is selected at the calling template.

With the CreateType-template and the "apply-templates" also a mode-attribute is given. In this case it doesn't have any use. But with this attribute you can control the order that the apply-templates is performed. There are some rules that the xsl-processor follows to determine which sub-template to perform first. So you could compute this order, but in fact you can't do much about it. However, you can control it to give a mode in which to apply the templates. Then it will only apply those templates that matches the xpath-expression but also matches the mode. I'll give an example a little later.

Variables

I found that variables are pretty handy. Earlier I just put all my text in sub-templates that I called. But variables are sometimes handier to contain default-texts and characters, like newlines, quotes etc. These variables can then be used in xpath expressions.

So, to add some:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="text" indent="no"/>
<!-- Variables -->
<!-- Object type suffix -->
       <xsl:variable name="ObjectSuffix"><xsl:text>_type </xsl:text></xsl:variable>
 <!-- Close the Type -->
 <xsl:variable name="CloseType">
  <xsl:text>
);</xsl:text>
 </xsl:variable>
 <!-- Object Heading -->
 <xsl:variable name="ObjectTypeHeading">
              <xsl:text> as object
(
  -- Author  : Martien van den Akker, Darwin-IT Professionals
  -- Created : 8/15/2008 12:00:00 PM
  -- Purpose : Table to Object Mapping

  -- Attributes
</xsl:text>
    </xsl:variable>
 <!-- Main template -->
```

```
 <xsl:template match="/">
  <xsl:apply-templates select="/TableDefinition" mode="ObjectType"/>
 </xsl:template>
 <!-- Template to create the object type -->
 <xsl:template name="ObjectType" match="/TableDefinition" mode="ObjectType">
  <xsl:call-template name="CreateType"/>
  <xsl:value-of select="$CloseType"/>
 </xsl:template>
 <xsl:template name="CreateType">
  <xsl:value-of select="concat('create or replace type ', Name, $ObjectSuffix, $ObjectTypeHeading)"/>
 </xsl:template>
</xsl:stylesheet>
```

Here you see that I transformed the CloseType template into a variable, added an ObjectSuffix and an ObjectTypeHeading variable. These are referenced from an Xpath expression by preceding the name with a dollar-sign. I added the ObjectSuffix to make the name unique from the table-name.

The output of the stylesheet thus far is:

```
create or replace type HSD_EMPLOYEES_type  as object
(
  -- Author  : Martien van den Akker, Darwin-IT Professionals
  -- Created : 8/15/2008 12:00:00 PM
  -- Purpose : Table to Object Mapping

  -- Attributes

);
```

Maybe not too exciting, but it gives an idea. Let's make it a little more exciting.

Using Modes: adding attributes

We want the object type to have attributes resembling the columns of the source table.

To do that we add a total of three templates. One that actually masters the generation of the attributes. But we have to separate the attributes with a comma. We could end each attribute declaration with a comma, but then the last one must not have one. My own coding-standards prefer to have each line prefixed with a comma. And that makes it a little easier to generate. I use one template that generates the first attribute and then a second that does the rest. I steer the xsl-processing using the "mode" attributes and the xpath expression with the xpath-function "position()".

```
 <!-- Generate the Attributes -->
 <xsl:template name="Attributes">
  <xsl:apply-templates select="Columns/Column[1]" mode="AttributeFirst"/>
  <xsl:apply-templates select="Columns/Column[position()>1]" mode="AttributeRest"/>
 </xsl:template>
 <!-- Generate First of the Attributes -->
 <xsl:template name="AttributeFirst" match="Columns/Column" mode="AttributeFirst">
  <xsl:value-of select="concat('  ',ColName, ' ' , DataType)"/>
  <xsl:call-template name="DataTypeDefinition"/>
 </xsl:template>
 <!-- Generate Rest of the Attributes -->
 <xsl:template name="AttributeRest" match="Columns/Column" mode="AttributeRest">
  <xsl:value-of select="concat($NewLineComma, ColName, ' ' , DataType)"/>
  <xsl:call-template name="DataTypeDefinition"/>
 </xsl:template>
 <!-- DataType Definition -->
 <xsl:template name="DataTypeDefinition">
  <xsl:if test="DataType!='DATE'">
   <xsl:choose>
    <xsl:when test="string-length(DataPrecision)>0">
     <xsl:value-of select="concat('(', DataLength, ', ' ,DataPrecision, ')')"/>
    </xsl:when>
```

```
    <xsl:otherwise>
     <xsl:value-of select="concat('(', DataLength, ')')"/>
    </xsl:otherwise>
   </xsl:choose>
  </xsl:if>
 </xsl:template>
```

The steering of the xsl-processor could also easily be done the xsl:if construction you see in the "DataTypeDefinition" template. There I test if there need to be a length and a precision part of the datatype (for varchars and numbers). When you want the attributes also sorted, you should do an "xsl:for-each" with an "xsl:sort". Then with a smart xsl:if you should test if it is the first loop-cycle of a latter one. Besides the "xsl:if" you find here also an example of "xsl:choose".

Adding a constructor and a method

The idea of the object is to have it instantiated and put it on an AQ queue. To instantiate this you could just create a procedure that does a select and give the results to the default constructor of the object type. It is however more convenient to have a constructor based on the primary key of the table and let it query the particular row into the attributes. Creating the declaration of the constructor occurs in the same way as the attributes earlier.

An extra method ToXML is added to have a method to get the contents of the object type into an XMLType.

The complete xslt is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="text" indent="no"/>
<!-- Stylesheet to Generate an Oracle Object Type from an XML Query on a table in the database
     Author: Martien van den Akker, Darwin-IT Professionals
     Version 1.0, juli 2008
-->

 <!-- Variables -->
 <!-- Object type suffix -->
 <xsl:variable name="ObjectSuffix">
  <xsl:text>_type </xsl:text>
 </xsl:variable>
 <!-- Close the Type -->
 <xsl:variable name="CloseType">
  <xsl:text>
);</xsl:text>
 </xsl:variable>
 <!-- Object Heading -->
 <xsl:variable name="ObjectTypeHeading">
  <xsl:text> as object
(
  -- Author  : Martien van den Akker, Darwin-IT Professionals
  -- Created : 8/15/2008 12:00:00 PM
  -- Purpose : Table to Object Mapping

  -- Attributes
</xsl:text>
 </xsl:variable>
 <xsl:variable name="NewLineComma">
  <xsl:text>
, </xsl:text>
 </xsl:variable>
 <!-- Main template -->
 <xsl:template match="/">
  <xsl:apply-templates select="/TableDefinition" mode="ObjectType"/>
 </xsl:template>
 <!-- Template to create the object type -->
 <xsl:template name="ObjectType" match="/TableDefinition" mode="ObjectType">
```

```
  <xsl:call-template name="CreateType"/>
  <xsl:call-template name="Attributes"/>
  <xsl:value-of select="$NewLineComma"/>
  <xsl:call-template name="PKConstructorDecl"/>
  <xsl:value-of select="$NewLineComma"/>
  <xsl:call-template name="ToXMLDecl"/>
  <xsl:value-of select="$CloseType"/>
 </xsl:template>
 <xsl:template name="CreateType">
  <xsl:value-of select="concat('create or replace type ', Name, $ObjectSuffix, $ObjectTypeHeading)"/>
 </xsl:template>
 <!-- Generate the Attributes -->
 <xsl:template name="Attributes">
  <xsl:apply-templates select="Columns/Column[1]" mode="AttributeFirst"/>
  <xsl:apply-templates select="Columns/Column[position()>1]" mode="AttributeRest"/>
 </xsl:template>
 <!-- Generate First of the Attributes -->
 <xsl:template name="AttributeFirst" match="Columns/Column" mode="AttributeFirst">
  <xsl:value-of select="concat('  ',ColName, ' ' , DataType)"/>
  <xsl:call-template name="DataTypeDefinition"/>
 </xsl:template>
 <!-- Generate Rest of the Attributes -->
 <xsl:template name="AttributeRest" match="Columns/Column" mode="AttributeRest">
  <xsl:value-of select="concat($NewLineComma, ColName, ' ' , DataType)"/>
  <xsl:call-template name="DataTypeDefinition"/>
 </xsl:template>
 <!-- DataType Definition -->
 <xsl:template name="DataTypeDefinition">
  <xsl:if test="DataType!='DATE'">
   <xsl:choose>
    <xsl:when test="string-length(DataPrecision)>0">
     <xsl:value-of select="concat('(', DataLength, ', ' ,DataPrecision, ')')"/>
    </xsl:when>
    <xsl:otherwise>
     <xsl:value-of select="concat('(', DataLength, ')')"/>
    </xsl:otherwise>
   </xsl:choose>
  </xsl:if>
 </xsl:template>
 <!-- Primary Key based Constructor -->
 <xsl:template name="PKConstructorDecl">
  <xsl:value-of select="concat('constructor function ', Name, $ObjectSuffix)"/>
  <xsl:apply-templates select="PrimaryKey/PrimaryKeyColumns/Column[1]"
mode="PKConstructorParametersFirst"/>
  <xsl:apply-templates select="PrimaryKey/PrimaryKeyColumns/Column[position()>1]"
mode="PKConstructorParametersRest"/>
  <xsl:text>) return self as result</xsl:text>
 </xsl:template>
 <!-- Primary Key Constructor, first PK-parameter -->
 <xsl:template name="PKConstructorParametersFirst" match="PrimaryKey/PrimaryKeyColumns/Column"
               mode="PKConstructorParametersFirst">
  <xsl:value-of select="concat('( p_',Name, ' ', DataType)"/>
 </xsl:template>
 <!-- Primary Key Constructor, rest of the PK-parameters -->
 <xsl:template name="PKConstructorParametersRest" match="PrimaryKey/PrimaryKeyColumns/Column"
               mode="PKConstructorParametersRest">
  <xsl:value-of select="concat(', p_',Name, ' ', DataType)"/>
 </xsl:template>
 <!-- To XML Method -->
 <xsl:template name="ToXMLDecl">
  <xsl:text>member function to_xml return sys.xmltype</xsl:text>
 </xsl:template>
</xsl:stylesheet>
```

The result of the transformation is:

```
create or replace type HSD_EMPLOYEES_type  as object
(
  -- Author  : Martien van den Akker, Darwin-IT Professionals
  -- Created : 8/15/2008 12:00:00 PM
  -- Purpose : Table to Object Mapping

  -- Attributes
  CREATED_BY VARCHAR2(30)
```

```
, CREATION_DATE DATE
, LAST_UPDATE_DATE DATE
, LAST_UPDATED_BY VARCHAR2(30)
, CIVIL_STATE VARCHAR2(1)
, BANK_ACCOUNT NUMBER(22, 15)
, E_MAIL VARCHAR2(30)
, TELEPHONE VARCHAR2(20)
, COUNTRY VARCHAR2(24)
, ZIP_CODE VARCHAR2(10)
, CITY VARCHAR2(24)
, STREET VARCHAR2(24)
, COMMISSION NUMBER(22, 7)
, SALARY NUMBER(22, 7)
, HIRE_DATE DATE
, JOB VARCHAR2(9)
, NAME VARCHAR2(10)
, EMP_ID NUMBER(22, 4)
, DEP_ID NUMBER(22, 2)
, ORACLE_USERNAME VARCHAR2(30)
, ID NUMBER(22, 4)
, constructor function HSD_EMPLOYEES_type ( p_ID NUMBER) return self as result
, member function to_xml return sys.xmltype
);
```

The object type body

Knowing this, you could easily create the object type body yourself. But I'm a nice guy so I'll present it to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" indent="no"/>
<!-- Stylesheet to Generate an Oracle Object Type body from an XML Query on a table in the database
     Author: Martien van den Akker, Darwin-IT Professionals
     Version 1.0, juli 2008
-->

  <!-- Variables -->
  <xsl:variable name="newline">
    <xsl:text>
</xsl:text>
  </xsl:variable>
  <xsl:variable name="quote">
    <xsl:text disable-output-escaping="yes">"</xsl:text>
  </xsl:variable>
  <xsl:variable name="CreateTypeBody">
    <xsl:text disable-output-escaping="no">create or replace type body </xsl:text>
  </xsl:variable>
  <xsl:variable name="ObjectSuffix">
    <xsl:text disable-output-escaping="no">_type </xsl:text>
  </xsl:variable>
  <xsl:variable name="ObjectTypeBodyHeading">
    <xsl:text>is
  /* Member constructor, procedures and functions  */
</xsl:text>
  </xsl:variable>
  <xsl:variable name="CloseType">
    <xsl:text>
end;</xsl:text>
  </xsl:variable>
  <xsl:variable name="ImplementationStart">
    <xsl:text> is
  begin
    </xsl:text>
  </xsl:variable>
  <xsl:variable name="ConstructorImplEnd">
    <xsl:text>    return;
  end;
  </xsl:text>
  </xsl:variable>
```

```
  <xsl:variable name="FunctionToXMLDecl">
    <xsl:text>   member function to_xml return sys.xmltype
        is
    l_result sys.xmltype;
  begin
    </xsl:text>
  </xsl:variable>
  <xsl:variable name="intolresult">
    <xsl:text>
                    )
       into l_result
    from dual;
</xsl:text>
  </xsl:variable>
  <xsl:variable name="FunctionImplEnd">
    <xsl:text>    return l_result;
  end;
  </xsl:text>
  </xsl:variable>
  <!-- Main Template -->
  <xsl:template match="/">
    <xsl:apply-templates select="/TableDefinition" mode="ObjectTypeBody"/>
  </xsl:template>
  <!-- Object Type Body -->
  <xsl:template name="ObjectTypeBody" match="/TableDefinition" mode="ObjectTypeBody">
    <xsl:value-of select="concat($CreateTypeBody,Name,$ObjectSuffix, $ObjectTypeBodyHeading)"/>
    <xsl:call-template name="PKConstructor"/>
    <xsl:call-template name="FunctionToXml"/>
    <xsl:value-of select="$CloseType"/>
  </xsl:template>
  <!-- Template to build a Primary Key based constructor -->
  <xsl:template name="PKConstructor">
    <xsl:call-template name="PKConstructorDecl"/>
    <xsl:value-of select="$ImplementationStart"/>
    <xsl:call-template name="SelectInto"/>
    <xsl:value-of select="$ConstructorImplEnd"/>
  </xsl:template>
  <!-- Primary Key based Constructor Declaration -->
  <xsl:template name="PKConstructorDecl">
    <xsl:value-of select="concat('constructor function ',Name)"/>
    <xsl:apply-templates select="PrimaryKey/PrimaryKeyColumns/Column[1]"
mode="PKConstructorParametersFirst"/>
    <xsl:apply-templates select="PrimaryKey/PrimaryKeyColumns/Column[position()>1]"
mode="PKConstructorParametersRest"/>
    <xsl:text>) return self as result</xsl:text>
  </xsl:template>
  <!-- First parameter of the parameter list (starts with a opening bracket) -->
  <xsl:template name="PKConstructorParametersFirst" match="PrimaryKey/PrimaryKeyColumns/Column"
                mode="PKConstructorParametersFirst">
    <xsl:value-of select="concat('( p_',Name, ' ', DataType)"/>
  </xsl:template>
  <!-- Rest of the parameters of the parameter list (parameters start with comma, list ends with
closing bracket) -->
  <xsl:template name="PKConstructorParametersRest" match="PrimaryKey/PrimaryKeyColumns/Column"
                mode="PKConstructorParametersRest">
    <xsl:value-of select="concat(', p_',Name, ' ', DataType)"/>
  </xsl:template>
    <!-- Select into part of constructor -->
  <xsl:template name="SelectInto">
    <xsl:apply-templates select="Columns/Column[1]" mode="SelectColumnFirst"/>
    <xsl:apply-templates select="Columns/Column[position()>1]" mode="SelectColumnRest"/>
    <xsl:apply-templates select="Columns/Column[1]" mode="IntoColumnFirst"/>
    <xsl:apply-templates select="Columns/Column[position()>1]" mode="IntoColumnRest"/>
    <xsl:call-template name="From"/>
    <xsl:call-template name="Where"/>
  </xsl:template>
  <!-- Select columns, first column with select statement -->
  <xsl:template name="SelectColumnFirst" match="Columns/Column" mode="SelectColumnFirst">
    <xsl:value-of select="concat('Select ', ColName)"/>
  </xsl:template>
  <!-- Select columns, rest of columns with comma's  -->
  <xsl:template name="SelectColumnRest" match="Columns/Column" mode="SelectColumnRest">
    <xsl:value-of select="concat($newline,'    ,      ',ColName)"/>
  </xsl:template>
```

```
  <!-- Into variables clause, first column, with into clause -->
  <xsl:template name="IntoColumnFirst" match="Columns/Column" mode="IntoColumnFirst">
    <xsl:value-of select="concat($newline, '    into   self.', ColName)"/>
  </xsl:template>
  <!-- Into variables clause, rest of columns column, with comma's -->
  <xsl:template name="IntoColumnRest" match="Columns/Column" mode="IntoColumnRest">
    <xsl:value-of select="concat($newline, '    ,      self.',ColName)"/>
  </xsl:template>
  <!-- From clause -->
  <xsl:template name="From">
    <xsl:value-of select="concat($newline, '    from ', Name)"/>
  </xsl:template>
  <!-- where clause -->
  <xsl:template name="Where">
    <xsl:apply-templates select="PrimaryKey/PrimaryKeyColumns/Column[1]" mode="WhereClauseFirst"/>
    <xsl:apply-templates select="PrimaryKey/PrimaryKeyColumns/Column[position()>1]"
mode="WhereClauseRest"/>
    <xsl:value-of select="concat(';',$newline)"/>
  </xsl:template>
  <xsl:template name="WhereClauseFirst" match="PrimaryKey/PrimaryKeyColumns/Column"
mode="WhereClauseFirst">
    <xsl:value-of select="concat($newline, '    where ', Name, ' = p_', Name)"/>
  </xsl:template>
  <xsl:template name="WhereClauseRest" match="PrimaryKey/PrimaryKeyColumns/Column"
mode="WhereClauseRest">
    <xsl:value-of select="concat($newline, '    and   ', Name, ' = p_', Name)"/>
  </xsl:template>
  <!-- Template to build a Primary Key based constructor -->
  <xsl:template name="FunctionToXml">
    <xsl:value-of select="$FunctionToXMLDecl"/>
    <xsl:call-template name="SelectIntoXML"/>
    <xsl:value-of select="$FunctionImplEnd"/>
  </xsl:template>
  <!-- Select Object Type attributes Into XMLType -->
  <xsl:template name="SelectIntoXML">
    <xsl:call-template name="SelectXMLName"/>
    <xsl:apply-templates select="Columns/Column" mode="SelectXMLColumn"/>
    <xsl:value-of select="$intolresult"/>
  </xsl:template>
  <!-- Select Object Type: Select clause -->
  <xsl:template name="SelectXMLName">
    <xsl:value-of select="concat('select xmlelement( ',$quote, Name,$quote)"/>
  </xsl:template>
  <!-- Select Object Type: Columns -->
  <xsl:template name="SelectXmlColumnFirst" match="Columns/Column" mode="SelectXMLColumn">
    <xsl:value-of select="concat($newline,'                    , xmlelement(',$quote,ColName,$quote,',
',ColName, ')')"
                  disable-output-escaping="yes"/>
  </xsl:template>
</xsl:stylesheet>
```

The outcome of the transformation will then be:

```
create or replace type body HSD_EMPLOYEES_type is
  /* Member constructor, procedures and functions  */
constructor function HSD_EMPLOYEES( p_ID NUMBER) return self as result is
  begin
    Select CREATED_BY
    ,      CREATION_DATE
    ,      LAST_UPDATE_DATE
    ,      LAST_UPDATED_BY
    ,      CIVIL_STATE
    ,      BANK_ACCOUNT
    ,      E_MAIL
    ,      TELEPHONE
    ,      COUNTRY
    ,      ZIP_CODE
    ,      CITY
    ,      STREET
    ,      COMMISSION
    ,      SALARY
    ,      HIRE_DATE
    ,      JOB
    ,      NAME
```

```
    ,      EMP_ID
    ,      DEP_ID
    ,      ORACLE_USERNAME
    ,      ID
    into   self.CREATED_BY
    ,      self.CREATION_DATE
    ,      self.LAST_UPDATE_DATE
    ,      self.LAST_UPDATED_BY
    ,      self.CIVIL_STATE
    ,      self.BANK_ACCOUNT
    ,      self.E_MAIL
    ,      self.TELEPHONE
    ,      self.COUNTRY
    ,      self.ZIP_CODE
    ,      self.CITY
    ,      self.STREET
    ,      self.COMMISSION
    ,      self.SALARY
    ,      self.HIRE_DATE
    ,      self.JOB
    ,      self.NAME
    ,      self.EMP_ID
    ,      self.DEP_ID
    ,      self.ORACLE_USERNAME
    ,      self.ID
    from HSD_EMPLOYEES
    where ID = p_ID;
    return;
  end;
     member function to_xml return sys.xmltype
         is
    l_result sys.xmltype;
  begin
    select xmlelement( "HSD_EMPLOYEES"
                     , xmlelement("CREATED_BY", CREATED_BY)
                     , xmlelement("CREATION_DATE", CREATION_DATE)
                     , xmlelement("LAST_UPDATE_DATE", LAST_UPDATE_DATE)
                     , xmlelement("LAST_UPDATED_BY", LAST_UPDATED_BY)
                     , xmlelement("CIVIL_STATE", CIVIL_STATE)
                     , xmlelement("BANK_ACCOUNT", BANK_ACCOUNT)
                     , xmlelement("E_MAIL", E_MAIL)
                     , xmlelement("TELEPHONE", TELEPHONE)
                     , xmlelement("COUNTRY", COUNTRY)
                     , xmlelement("ZIP_CODE", ZIP_CODE)
                     , xmlelement("CITY", CITY)
                     , xmlelement("STREET", STREET)
                     , xmlelement("COMMISSION", COMMISSION)
                     , xmlelement("SALARY", SALARY)
                     , xmlelement("HIRE_DATE", HIRE_DATE)
                     , xmlelement("JOB", JOB)
                     , xmlelement("NAME", NAME)
                     , xmlelement("EMP_ID", EMP_ID)
                     , xmlelement("DEP_ID", DEP_ID)
                     , xmlelement("ORACLE_USERNAME", ORACLE_USERNAME)
                     , xmlelement("ID", ID)
                     )
       into l_result
    from dual;
    return l_result;
  end;

end;
```

Performing the XSL in the database

Basically there are two methods to do the transformation from the database. Having the XSLT's in a CLOB-column in a table in the database, you could do it as follows:

```
select xtb.xml.transform( xmltype(doc.xmldoc) ) from (
```

```
 select tbl.owner
     ,      tbl.table_name
     ,      xmlelement( "TableDefinition"
                      , xmlelement("Name", tbl.table_name)
                      , xxx_table_definitions.TabColumns(tbl.owner, tbl.table_name)
                      , xxx_table_definitions.PKColumns(tbl.owner, tbl.table_name)
                      ) xml
     from all_tables tbl
     where tbl.owner      = 'HDEMO65'
     and   tbl.table_name = 'HSD_EMPLOYEES'
     group by tbl.owner, tbl.table_name) xtb
     ,  xxx_xml_documents doc
where doc.docname='CreateObjectTypeBody';
```

The disadvantage of this is the performance due to several parses that are done over and over again. So you'd probably rather program it in Pl/Sql using the dbms_xslprocessor package.

First you'll have to parse the xslt-stylesheet. Remember: an xsl-stylesheet is in itself an xml-document. This is done in Pl/Sql in the following way:

```
    retDoc xmldom.DOMDocument;
    parser xmlparser.Parser;
  BEGIN
    parser := xmlparser.newParser;
    xmlparser.parseCLOB(parser, xml);
    retDoc := xmlparser.getDocument(parser);
    xmlparser.freeparser(parser);
    return retDoc;
  END;
```

Then from the resulting DOMDocument you'll have to create an XMLStylesheet:

```
    xmldoc   xmldom.DOMDocument;
    newsheet xslprocessor.Stylesheet;
  BEGIN
    xmldoc   := xml.parse(doc); -- See the Pl/Sql code above
    newsheet := xslprocessor.newStylesheet(xmldoc, NULL);
    xmldom.freeDocument(xmldoc);
    RETURN newsheet;
```

And then at the end the transformation itself, after parsing the source-xml-document:

```
retval VARCHAR2(32767);
  BEGIN
    xmldoc := xml.parse(source); -- See the parsing code above
    retval := transform(xmldoc, style, params);
    xml.freeDocument(xmldoc);
    RETURN retval;
```

When the resulting code is a valid DDL command, like our create type, then you can just issue an 'execute immediate retval;' to get it in the database right away.

Conclusions and comments

As you see, it is not too hard to generate code using XSL and an xml-query on the database. I also created a java-application that actually parses a package-specification to XML. It's not too hard to do that either. Having your definitions into an xml-file, you can transform it in anything you want.

From the code above you should not have too much trouble in creating an xsl that creates the java-beans of Lucas.

I just found however one big difference between the parser in the database and the parser shipped with jDeveloper. The java-xml-parser of jDeveloper transforms the xml just as I expected and showed in the examples of this documents. The database-xml-parser, however, transforms quotes, "greater than" characters in '=>' code, etc. in to excaped XML-characters like '"', '>', etc. I have not found anything to steer that yet. Even an XSL-attribute like 'disable-output-escaping="yes"' won't help a bit. So for the resulting code from database-parser you should replace those occurrences with the corresponding character like:

```
function replaceEntities(pDoc in varchar2)
return varchar2
is
  lDoc varchar2(32767);
begin
  lDoc := replace(srcstr => pDoc, oldsub => '&quot;', newsub => '"');
  lDoc := replace(srcstr => lDoc, oldsub => '&apos;', newsub => '''');
  lDoc := replace(srcstr => lDoc, oldsub => '&gt;'  , newsub => '>');
  return lDoc;
end;
```

For the xml-parsing and the xslt-transformation in the database I must say I used the example packages from Steve Muench's book "Building Oracle XML Applications".