PFC:
Auto-generador de clases Java a partir de metadatos de una base de datos

ANEXOS

Aplicación xsltGenerator:

MANUAL DE USUARIO

Alejandro Lecina Laplana
Ingeniería en Informática
Junio 2012

Índice de contenido

Contenido del documento:

Este documento tiene dos principales objetivos:

- El primero es explicar la motivación y los objetivos de la aplicación *xsltGenerator*, para así comprender el uso para el que ha sido pensada la aplicación. Posteriormente se detallaran las opciones que da la aplicación y el como usar la misma para conseguir dichos objetivos.

- El segundo objetivo es el de detallar los procesos internos que se implementan en la aplicación.

Debido a que la finalidad de la aplicación es la de construir una herramienta para la ayuda a la hora de generar nuevas aplicaciones, es importante definir con exactitud la lógica de todos los procesos internos, para que así un usuario tenga la máxima información a cerca de ellos y pueda modificarlos con facilidad para adaptarlos a diferentes usos que se ajusten a sus necesidades.

La estructura interna del documento se divide en 3 grandes secciones:

- La primera consta de una introducción en la que se detalla la motivación para crear la aplicación y los objetivos que se pretenden cubrir con la misma.

- A continuación se comenta la aplicación *xsltGenerator*. Se detalla como configurar la aplicación y usar la misma, la estructura de los ficheros generados y su utilidad, y por ultimo los procesos internos de la aplicación para generar dichos ficheros.

- En ultimo lugar se ilustra un ejemplo de como usar los ficheros generados y las clases contenidas en la aplicación para construir la aplicación *xsltUse* que logra alcanzar los objetivos para los que la aplicación fue ideada.

Introducción

1- Motivación de la aplicación:

La motivación de la aplicación parte de la idea de desarrollar una herramienta capaz de generar automáticamente un interfaz básico (junto con las clases necesarias) para interactuar con una base de datos a partir de los meta datos de las tablas contenidas en ella, evitando el trabajo de tener que implementar a mano las clases y interfaces correspondientes.
Un problema añadido a la hora de generar estos interfaces y clases radica en la perdida de información a cerca de relación referencial (claves externas) entre las distintas tablas.
Debido a esto, se busca que a la hora de generar las clases y interfaces correspondientes se tenga un esquema claro de la relación referencial entre las tablas y una información completa del origen de las claves importadas, para así poder presentar los datos a los usuarios de una forma mas comprensible, añadiendo valores fácilmente identificables por un usuario correspondientes a tuplas cuyas claves primarias han sido importadas y no son fácilmente identificables a primera vista (códigos numéricos, abreviaturas, etc...)

2- Objetivos de la aplicación:

La aplicación *xsltGenerator* es una aplicación web cuyo objetivo es conectarse a una base de datos y generar unos ficheros xml (uno por tabla) con la estructura detallada de las tablas seleccionadas de dicha base de datos para poder generar ficheros a través de transformaciones XSL que puedan ser utilizados por otras aplicaciones.
La aplicación *xsltGenerator* ha sido pensada para ser usada junto a una aplicación *xsltUse*, la cual usa los ficheros generados por la primera aplicación para construir una aplicación web capaz de interactuar con una base de datos.
Los datos obtenidos de las tablas son la información de las columnas así como las claves primarias y importadas. En el caso de las claves importadas se realiza una búsqueda en la base de datos con intención de trazar el origen de los datos importados, es decir, la tabla de la cual proviene la columna original.
Usando las transformaciones XSL, se pueden generar cuatro tipos de ficheros de salida:

1. Una clase java simple (POJO: Plain Old Java Object) que represente la estructura de una tabla de la base de datos. Dicha clase tendrá como atributos las columnas de la tabla y como métodos los setters y getters correspondientes a los atributos.

2. Una clase java que contenga una información mas detallada de las columnas con objetivo de interactuar con la base de datos. Dicha clase necesitara de clases java auxiliares también desarrolladas en el proyecto y de otros ficheros generados por la aplicación para poder insertar, seleccionar y modificar datos en la base de datos.

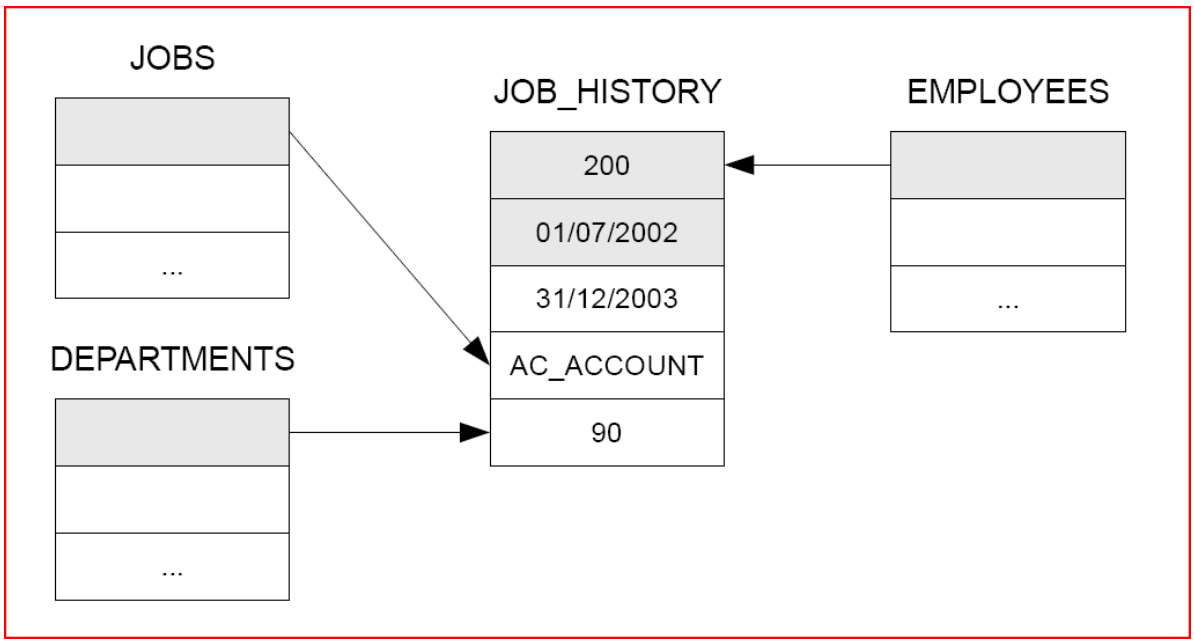
3. Paginas xhtml que contengan un formulario con los campos de texto de entrada correspondientes para una tabla y que permitan insertar, seleccionar y modificar datos en la base de datos. Dichas paginas xhtml usaran las clases correspondientes generadas para su correcto funcionamiento.

4. Paginas xhtml que permiten ver todos los datos de una tabla y acceder a información extra de los campos que sean claves importadas buscando la tupla de la cual provienes los datos importados en las tablas correspondientes.

2.1- Relación referencial entre tablas

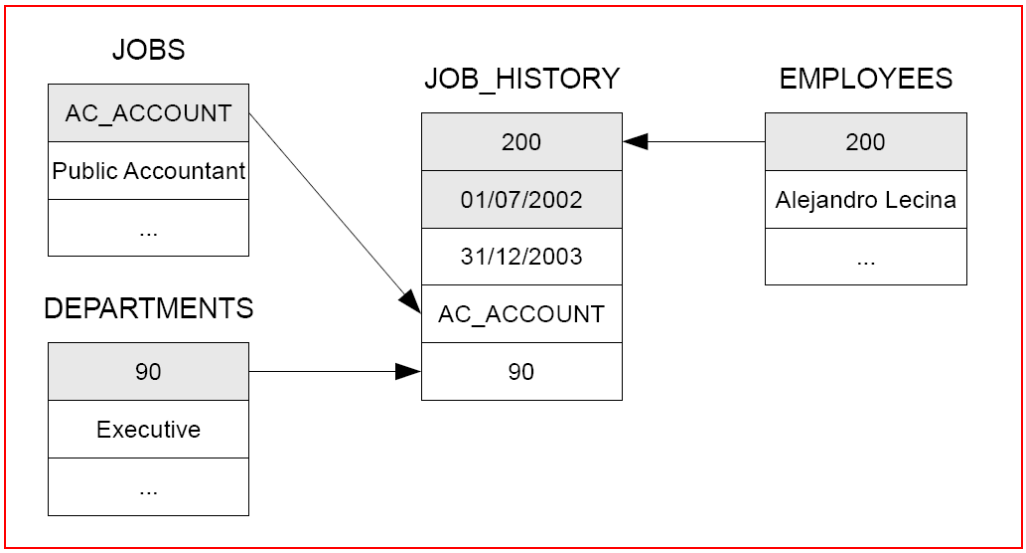
Puede suceder que al obtener directamente los datos de una tupla guardada en una base de datos, el observador no pueda identificar con claridad que significan alguno de los valores de las columnas. Esto sucede mas fácilmente en el caso de columnas que corresponden a claves importadas de otras tablas.

Veamos a continuación un ejemplo de una tupla de una tabla de una base de datos:

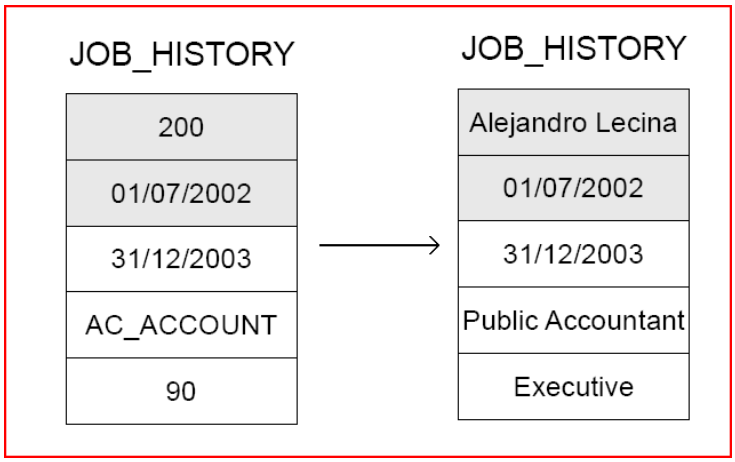


Las celdas en gris representan la clave primaria de la tabla, y las columnas apuntadas por flechas son columnas importadas de otras tablas (claves ajenas)
En este caso la tabla *JOB_HISTORY* contiene una clave primaria doble y tres columnas importadas. Ademas se da la particularidad de que una columna que forma parte de la clave primaria doble es una clave importada.
La tabla *JOB_HISTORY* guarda el historial de los trabajos que a llevado a cabo un empleado, guardando la fecha de inicio y la de finalización del mismo, así como el tipo de trabajo y el departamento para el que se desarrollo.
En el ejemplo se muestra una tupla de las que contiene la tabla. La información de las columnas de dicha tupla es difícilmente interpretable a primera vista por un observador.
Mirando únicamente la información interna de la tabla se obtiene que el trabajador *"200"* realizo un trabajo *"AC_ACCOUNT"* para el departamento *"90"* empezando el 1 de Julio de 2002. Seria interesante añadir información a los datos obtenidos para una comprensión mejor de los datos guardados en la tabla.
Si se ha inspeccionado la estructura de la base de datos y las referencias entre tablas podemos

obtener campos identificativos de cada tabla que aportarían información útil a la hora de interpretar los datos:



Usando estos datos podríamos obtener una tupla fácilmente interpretable por el usuario:



Ahora podemos observar que el trabajador Alejandro Lecina empezó un trabajo del tipo contabilidad publica el 1 de Julio de 2002 para el departamento ejecutivo. Esta información es mucho mas útil para un usuario, y mucho mas fácil de interpretar a simple vista.

Interesa poder usar este razonamiento en la aplicación, de modo que al ver los datos de una tabla, automáticamente nos cargue los valores identificativos relacionados con las claves que se importan de otras tablas, tal y como se muestra en el siguiente ejemplo:

Tupla contenida en la tabla

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
200	01/07/2002	31/21/2003	AC_ACCOUNT	90

Valores de la tupla añadiendo información adicional para las columnas importadas

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
200	01/07/2002	31/21/2003	AC_ACCOUNT	90	Alejandro Lecina	Public Accountant	Executive

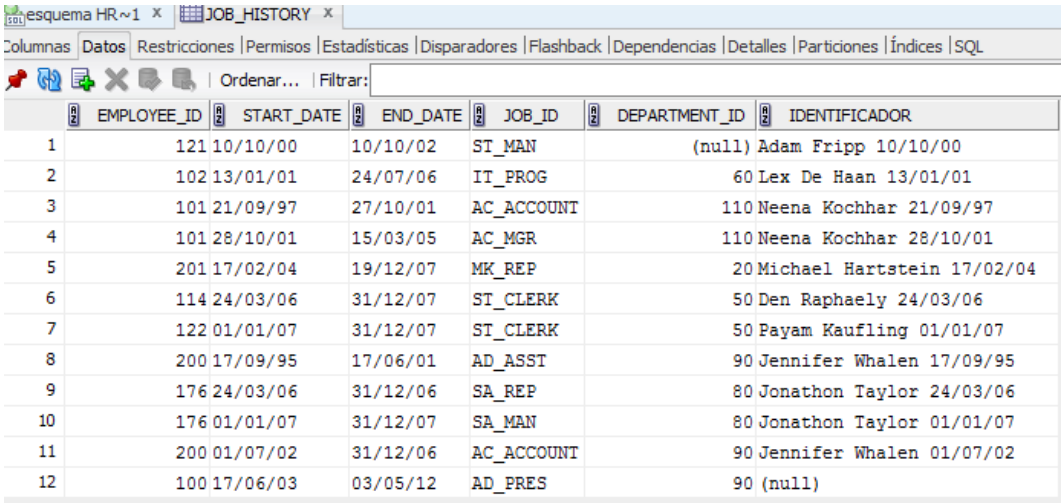
2.2- Generación de las paginas web correspondientes al interfaz

Como se ha explicado en la introducción de la aplicación, la idea es tener una herramienta que genere automáticamente un interfaz básico para el acceso a una base de datos, independientemente de la estructura de los mismos.
La aplicación esta pensada para generar dicho interfaz a partir de cualquier base de datos de tipo relacional accesible mediante una API JDBC (Java Database Connectivity).

A continuación se expone un ejemplo de un interfaz generado con la aplicación:

En este ejemplo se esta trabajando con una base de datos Ora*cle Database 11g Express Edition* y con el esquema *HR* que viene de ejemplo en la misma.

Nos fijamos en la estructura de la tabla *JOB_HISTORY*, con la ayuda del programa *sqldeveloper:*


esquema HR~1 | JOB_HISTORY

Columnas | Datos | Restricciones | Permisos | Estadísticas | Disparadores | Flashback | Dependencias | Detalles | Particiones | Índices | SQL

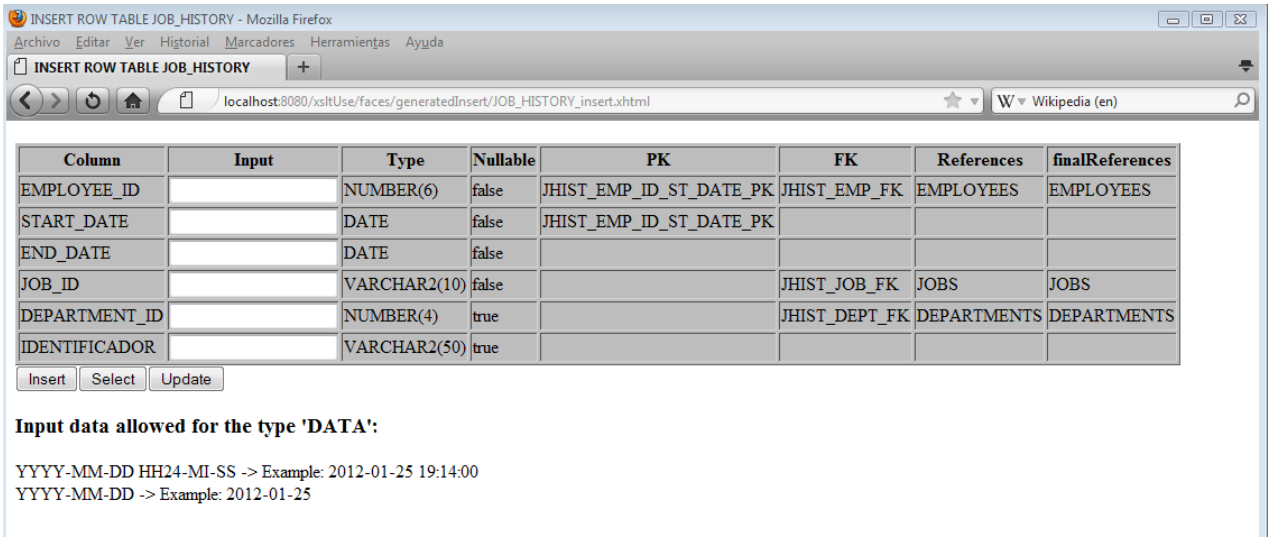
Ordenar... | Filtrar:

	EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID	IDENTIFICADOR
1	121	10/10/00	10/10/02	ST_MAN	(null)	Adam Fripp 10/10/00
2	102	13/01/01	24/07/06	IT_PROG	60	Lex De Haan 13/01/01
3	101	21/09/97	27/10/01	AC_ACCOUNT	110	Neena Kochhar 21/09/97
4	101	28/10/01	15/03/05	AC_MGR	110	Neena Kochhar 28/10/01
5	201	17/02/04	19/12/07	MK_REP	20	Michael Hartstein 17/02/04
6	114	24/03/06	31/12/07	ST_CLERK	50	Den Raphaely 24/03/06
7	122	01/01/07	31/12/07	ST_CLERK	50	Payam Kaufling 01/01/07
8	200	17/09/95	17/06/01	AD_ASST	90	Jennifer Whalen 17/09/95
9	176	24/03/06	31/12/06	SA_REP	80	Jonathon Taylor 24/03/06
10	176	01/01/07	31/12/07	SA_MAN	80	Jonathon Taylor 01/01/07
11	200	01/07/02	31/12/06	AC_ACCOUNT	90	Jennifer Whalen 01/07/02
12	100	17/06/03	03/05/12	AD_PRES	90	(null)

La tabla consta de 6 campos, de los que investigando con la aplicación *sqldeveloper* descubrimos que el campo *EMPLOYEE_ID* mas el campo *START_DATE* forman la clave primaria de la tabla, y que los campos *JOB_ID* y *DEPARTMENT_ID* corresponden a claves importadas.

La aplicación se conectara al esquema de la base de datos, y simplemente seleccionando la tabla *JOB_HISTORY*, generara los xhtml y clases java correspondientes para el acceso a dicha tabla en la base de datos.

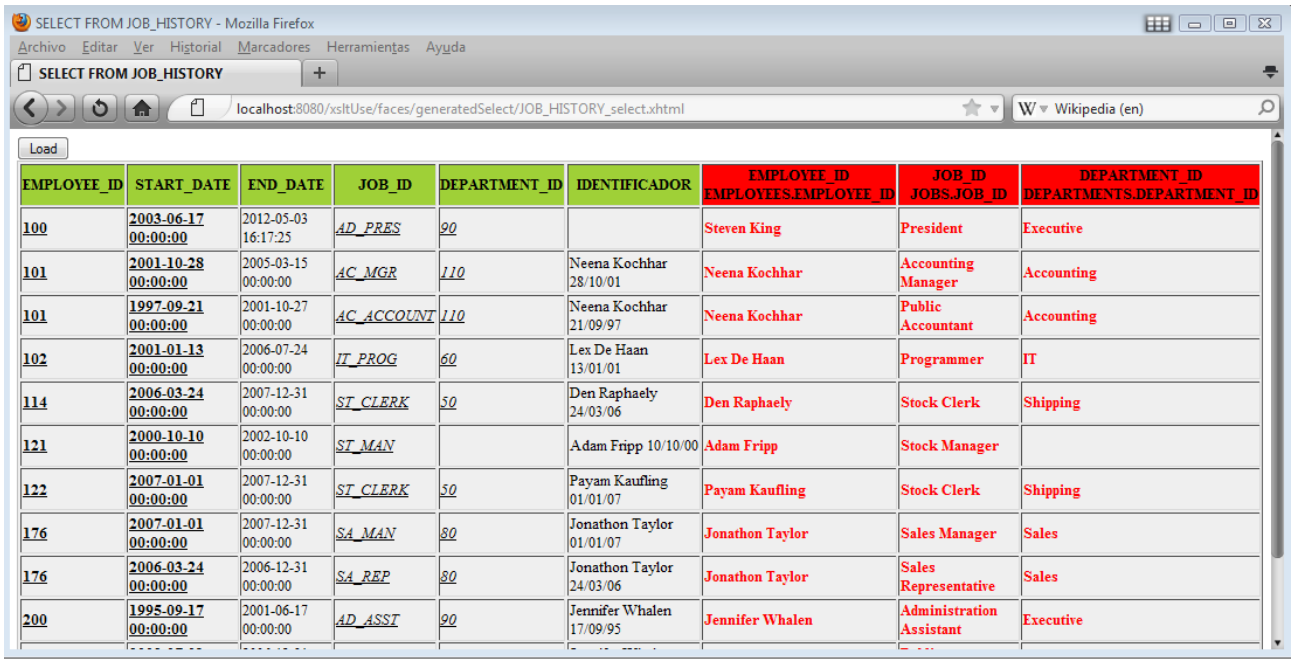
La pagina xhtml correspondiente a la inserción y modificación de datos tendrá el siguiente aspecto:



La pagina muestra toda la información obtenida de la base de datos, así como la procedencia de las claves importadas.
La columna *PK* muestra si una columna es clave primaria, si es así, muestra el nombre de la clave primaria a la que pertenece la columna. En este ejemplo se ve que la tabla tiene una clave primaria doble debido a que dos columnas comparten el mismo nombre de clave primaria.
La columna *FK* representa si la columna es una clave importada, así como el nombre de la clave importada. Si una columna es clave importada, también se muestra la columna *References*, correspondiente a la tabla directa de la que se importa la clave, y la columna *finalReferences*, correspondiente a la tabla original de la que procede la clave, que puede ser distinta en el caso de claves importadas con nivel de profundidad mayor de 1, como sera explicado mas adelante.

El botón "*insert*" permite introducir datos, siempre que los mismos cumplan todas las restricciones necesarias.
El botón "*Select*" permite seleccionar datos. En primer lugar se rellenan los campos que sean clave primaria, y al pulsar se cargara el registro que coincida con los campos rellenados si este existe en la base de datos.
Por ultimo, el botón "*Update*" permite modificar una tupla de la base de datos que ya existe. Después de cargar un registro con el botón "*select"*, se pueden cambiar valores que no sean clave primaria y confirmar los cambios en la base de datos pulsado el botón.

La pagina xhtml correspondiente a la visualización de los datos seria la siguiente:



EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID	IDENTIFICADOR	EMPLOYEE_ID EMPLOYEES.EMPLOYEE_ID	JOB_ID JOBS.JOB_ID	DEPARTMENT_ID DEPARTMENTS.DEPARTMENT_ID
100	2003-06-17 00:00:00	2012-05-03 16:17:25	AD_PRES	90		Steven King	President	Executive
101	2001-10-28 00:00:00	2005-03-15 00:00:00	AC_MGR	110	Neena Kochhar 28/10/01	Neena Kochhar	Accounting Manager	Accounting
101	1997-09-21 00:00:00	2001-10-27 00:00:00	AC_ACCOUNT	110	Neena Kochhar 21/09/97	Neena Kochhar	Public Accountant	Accounting
102	2001-01-13 00:00:00	2006-07-24 00:00:00	IT_PROG	60	Lex De Haan 13/01/01	Lex De Haan	Programmer	IT
114	2006-03-24 00:00:00	2007-12-31 00:00:00	ST_CLERK	50	Den Raphaely 24/03/06	Den Raphaely	Stock Clerk	Shipping
121	2000-10-10 00:00:00	2002-10-10 00:00:00	ST_MAN		Adam Fripp 10/10/00	Adam Fripp	Stock Manager	
122	2007-01-01 00:00:00	2007-12-31 00:00:00	ST_CLERK	50	Payam Kaufling 01/01/07	Payam Kaufling	Stock Clerk	Shipping
176	2007-01-01 00:00:00	2007-12-31 00:00:00	SA_MAN	80	Jonathon Taylor 01/01/07	Jonathon Taylor	Sales Manager	Sales
176	2006-03-24 00:00:00	2006-12-31 00:00:00	SA_REP	80	Jonathon Taylor 24/03/06	Jonathon Taylor	Sales Representative	Sales
200	1995-09-17 00:00:00	2001-06-17 00:00:00	AD_ASST	90	Jennifer Whalen 17/09/95	Jennifer Whalen	Administration Assistant	Executive

El botón *"load"* sirve para cargar los datos de la tabla correspondiente.
En verde se muestran los nombres de las columnas de la tabla. En rojo se muestran las columnas con información adicional de las columnas que son claves importadas. El primer valor representa el nombre de la columna de la tabla a la que se le añade la información, y debajo, la tabla y columna original pertenecientes a la clave de la cual los datos adicionales han sido obtenidos.
En negrita y subrayado se muestran los valores que son clave primaria, y solo subrayados los valores que son claves importadas.
También se muestra una columna llamada "identificador". Esta columna es la que se ha añadido a la tabla para usar a la hora de identificar la tupla. Por tanto, los datos en rojo corresponden a la columna "identificador" correspondiente a la tupla cuya clave primaria corresponde con el valor de la clave importada en la tabla.

La aplicación xsltGenerator

1- Entorno de desarrollo

En primer lugar se describe el entorno en el que se ha desarrollado la aplicación así como las herramientas utilizadas durante el mismo.

-Entorno de desarrollo: NetBeans IDE 7.0.1.
-Plataforma java: jdk 1.6.
-Servidor de aplicaciones: Glassfish Server 3.1.
-Base de datos: Oracle Database XE 11.2.
-Driver jdbc: ojdbc6.jar.
-Herramienta de ayuda para la Base de datos: sqldeveloper 3.0.
-Tipo de aplicación: aplicación web.
-Framework: Java Server Faces 2.0.
-PC: Intel core 2 duo, T5200, 1,60 Ghz, 2GB RAM.
-Sistema Operativo: Windows Vista Home Premium.

2- Interfaz y opciones de la aplicaciones

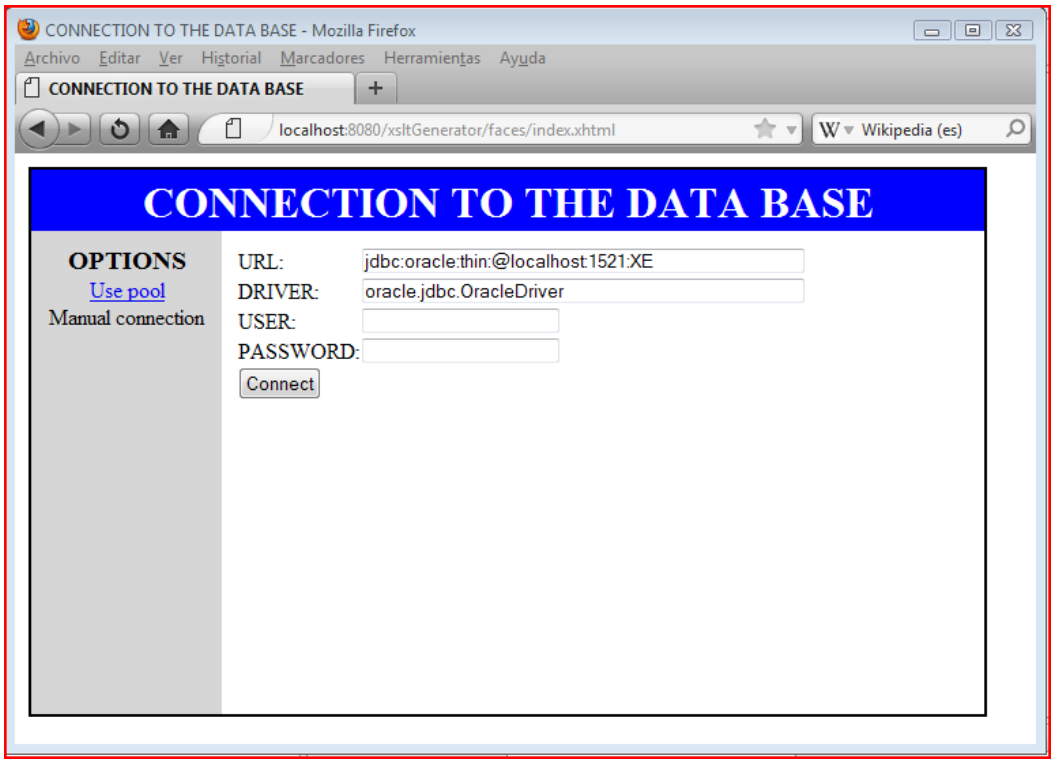
En esta sección se mostrara el interfaz de la aplicación así como una guiá para el uso de la misma. Esto ademas ayudara a ofrecer una visión global de las opciones que permite la misma.
Mas adelante se detallara el proceso interno que se requiere para llevar a cabo estas opciones.

2.1- conexión a la base de datos

En primer lugar, una vez lanzada la aplicación, lo primero que hay que hacer es conectarse a la base de datos. Existen dos opciones para conectarse a la base de datos. La primera se trata de una "*conexión manual*", la que consiste en introducir en unos campos de textos los datos necesarios para configurar la conexión. La segunda forma, y la mas recomendada, es usar una agrupación de conexiones (connection pool). Un pool de conexiones es una colección de conexiones abiertas a una base de datos de manera que puedan ser reutilizadas al realizar múltiples consultas o actualizaciones. El pool de conexiones ha de ser configurado previamente en el servidor de aplicaciones.

2.1.1- conexión manual:

Captura de la pagina *"manualConnection.xhtml"*, accesible mediante la opción del menú *"manual connection"*



En esta pantalla hay que introducir los valores URL, DRIVER, USER Y PASSWORD necesarios para conectare a una base de datos. Esta opción nos permite conectarnos a una base de datos sin tener que configurar un pool de conexiones.

2.1.1- conexión mediante un connection pool:

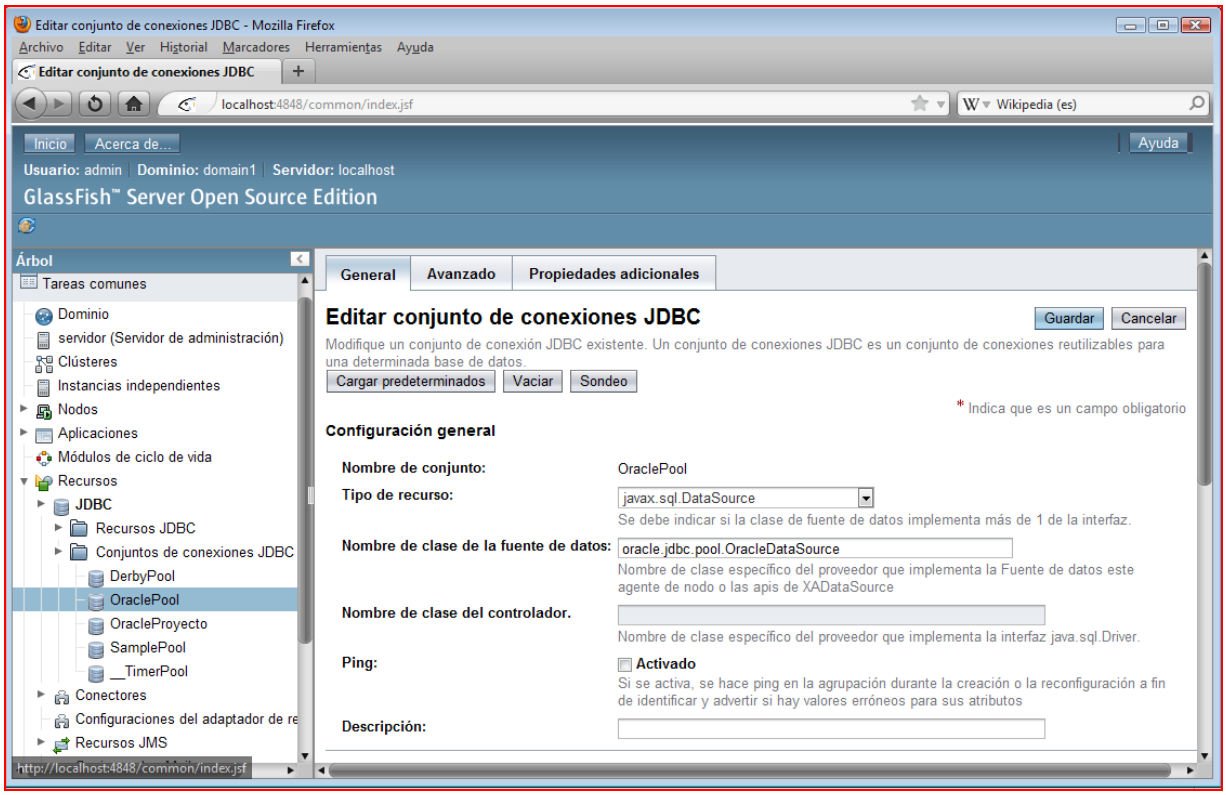
En el caso de trabajar con una conexión manual, cada vez que la aplicación se comunica con la base de datos tiene que abrir una conexión. El pool de conexiones supone una mejora ya que genera un grupo de conexiones que se mantiene abierta el tiempo que dura la ejecución del programa y sólo es cerrada al finalizar el trabajo de la aplicación con la base de datos. Al mantenerse abierto un grupo de conexiones, éstas son atribuidas a los diferentes hilos de ejecución únicamente el tiempo de una transacción con la base de datos. Al finalizar su utilización, la conexión se pone a disposición de otro hilo de ejecución que necesite de ese recurso, en lugar de cerrarla o de asignarla permanentemente a un único hilo de ejecución.

A continuación se muestra como se ha creado el connection pool usado durante el desarrollo de la aplicación.

En primer lugar hay que copiar el driver jdbc usado para la conexión (en este caso, el driver ojdbc6.jar) en la carpeta del dominio de Glassfish, concretamente en la carpeta lib/. En este caso en concreto el driver queda en la siguiente ruta:
C:\Users\propietario\.netbeans\7.0\config\GF3\domain1\lib
Esto es necesario para asegurarse de que *glassfish* usa el driver para configurar el pool de

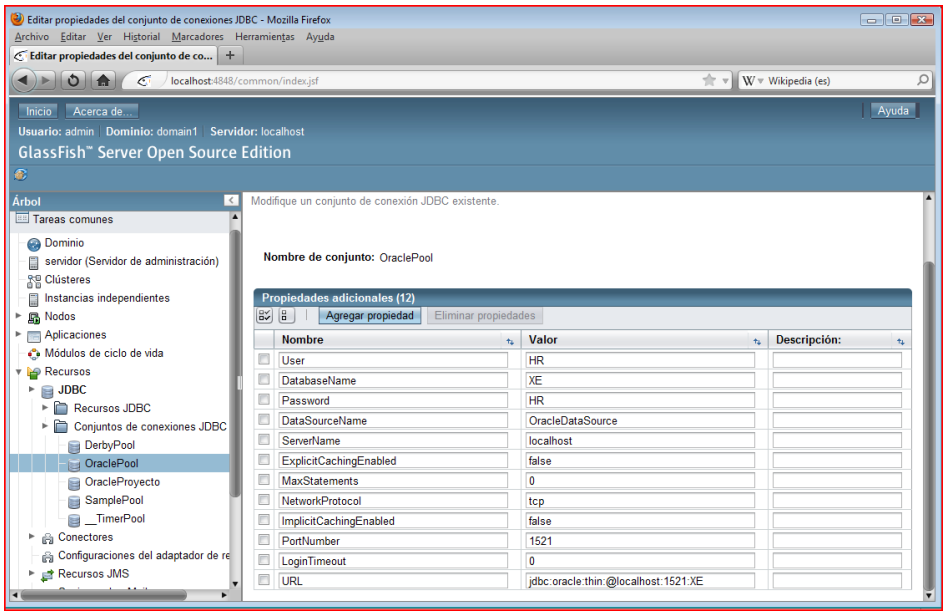
conexiones deseado, ya que solo viene con unos pocos driver por defecto.

A continuación hay que entrar en la consola de administración de *Glassfish* para crear el *conjunto de conexiones JDBC*:



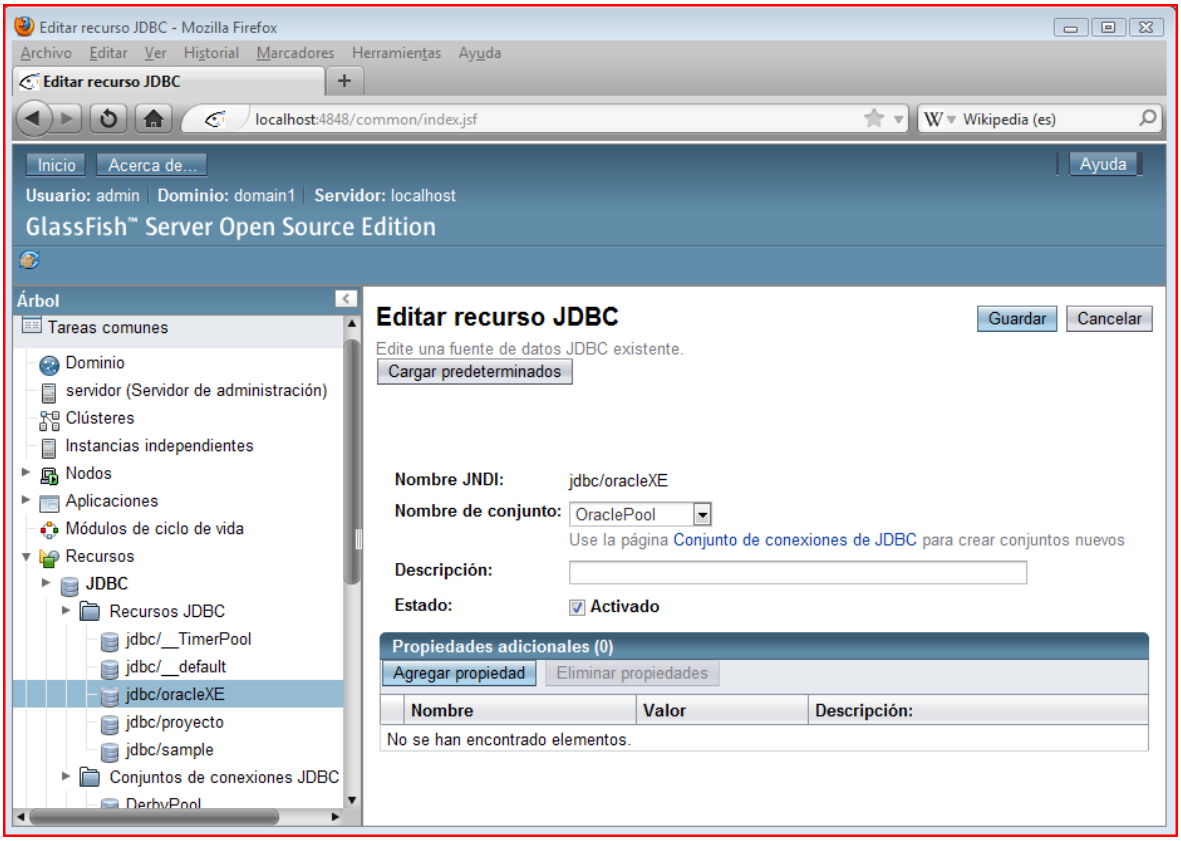
En la captura de pantalla superior vemos donde hay que configurar el conjunto de conexiones JDBC y los datos necesarios. En este caso se ha creado un conjunto de conexiones llamado *"OraclePool"*.

En la siguiente captura se muestran las propiedades adicionales que hay que añadir para finalizar de configurar el connection pool:



Con esto ya tenemos creado el conjunto de conexiones llamado *"OraclePool"*

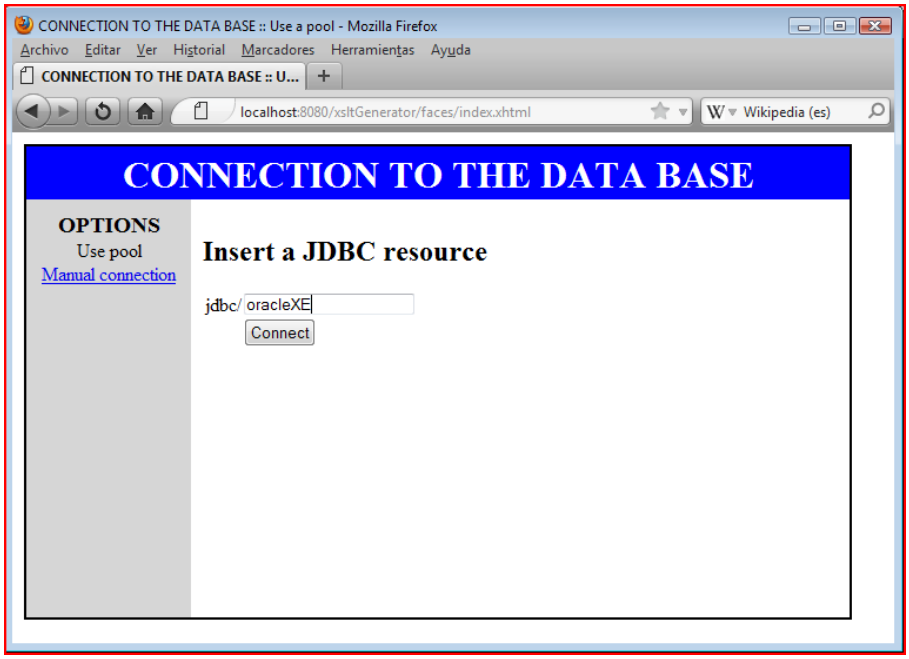
Ahora es el turno de crear el *recurso JDBC* que usaremos para conectado desde la aplicación a la base de datos.



En este caso el recurso creado se llama *"jdbc/oracleXE"* (todos los recursos empiezan por "*jdbc/*").

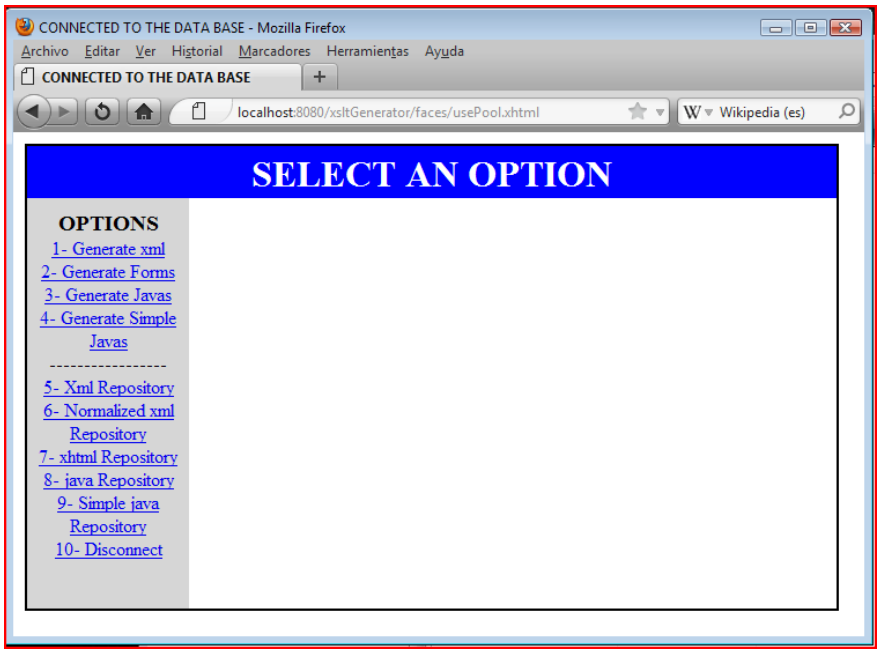
Como se ve en la aplicación hay que definir que el recurso pertenece al conjunto "*OraclePool*" previamente definido.

Ahora se muestra una captura de la pagina *usePool.xhtml*, a la que se llega mediante la opción del menú "*Use pool*". Simplemente hay que introducir el nombre del recurso creado para usar el pool de conexiones.



2.2- generar archivos

Como se ha señalado en la introducción, el programa esta diseñado para generar ficheros a partir de una base de datos. Después de conectarse a la base de datos con éxito, se accederá a la pagina con las acciones disponibles por la aplicación:

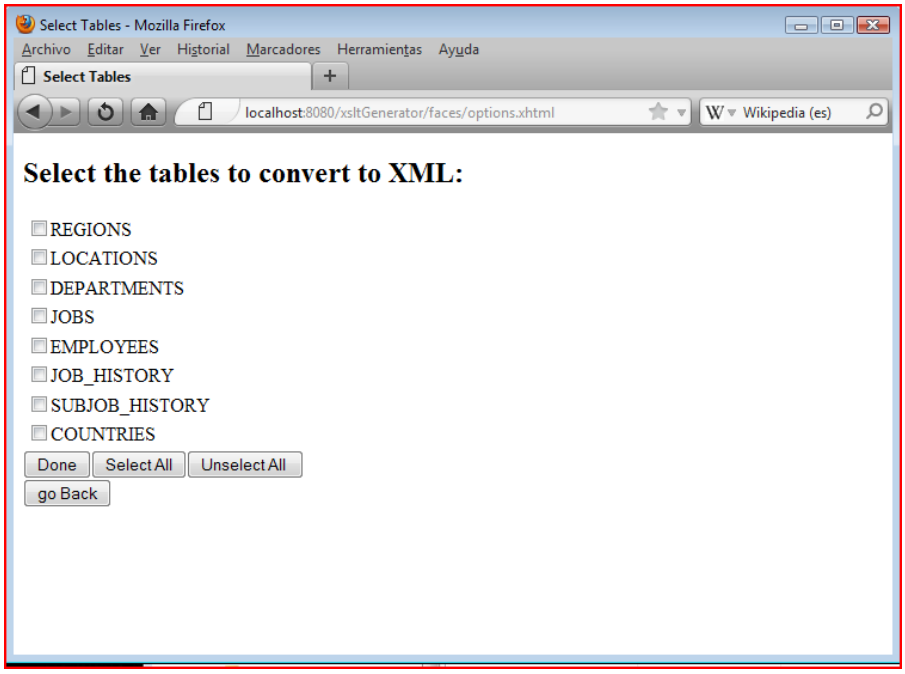


Las opciones del 1 al 4 del menú de opciones son las que nos permiten generar los ficheros que necesitamos. Las opciones del 5 al 9 permiten acceder al repositorio de datos para ver los ficheros generados. La opción 10 permite desconectarse de la base de datos (Cerrar el connection pool).

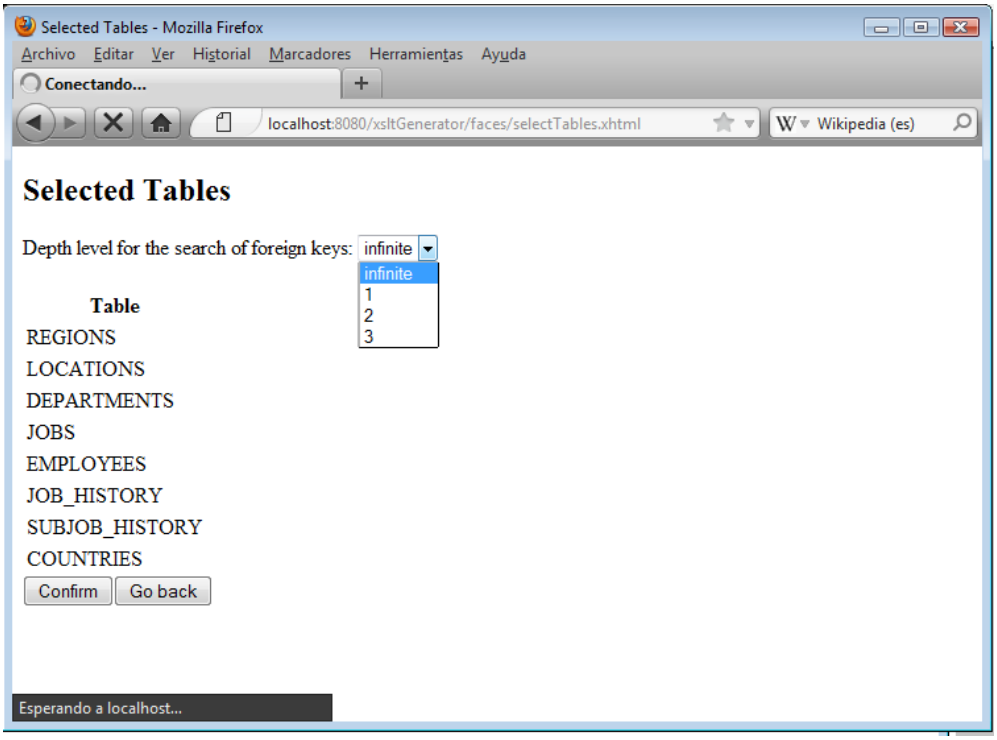
Existen tres tipos de ficheros de salida a generar: xml, paginas xhtml (formularios) y clases java. Los xhtml y las clases java se generan a partir de los xml generados mediante esta aplicación. Los xml se generan a partir de las tablas seleccionadas de la base de datos.

Por tanto, el primer paso a realizar es el de generar los ficheros xml (opción 1).

Captura de la pantalla resultante de seleccionar la opción *1-Generar xml:*



En esta pagina se muestran las tablas encontradas dentro del esquema de la base de datos al que nos hemos conectado. En este ejemplo se muestran las tablas del esquema que viene por defecto para pruebas en la base de datos utilizada durante el desarrollo.
Aquí es donde se pueden seleccionar las tablas de las que queremos generar los xml que utilizaremos para generar los ficheros que interesan.
Una vez seleccionadas las tablas deseadas, pulsando el botón "*Done*", accederemos a la pantalla de confirmación:

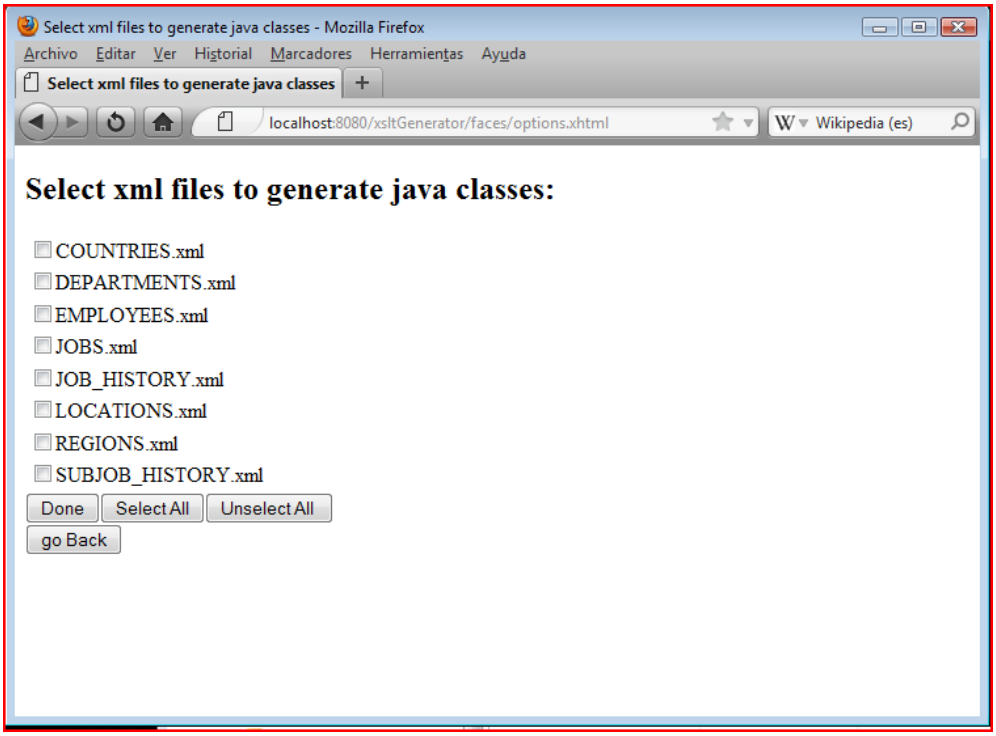


Antes de confirmar las tablas, debemos seleccionar la profundidad de la búsqueda de las claves ajenas. Si marcamos 1, para una columna que sea clave ajena, solo se buscara la información de la tabla y columna inmediata de la cual procede la clave ajena. Si se marca infinito, buscara la información de toda la cadena de tablas y columnas de la cual procede la clave exportada original.

Una vez que tenemos generados los xml, podemos generara los demás ficheros que nos interesan.

Como se ha vista en la pantalla principal de opciones, tenemos 3 tipos de ficheros a generar:
La opción 2 nos permite generar formularios (paginas xhtml para que el usuario pueda interactuar con la base de datos), la 3 genera clases java con información detallada de las columnas de las tablas, y por ultimo, la opción 4, que genera unas clases java mas básicas, teniendo en cuenta solo los nombres de las columnas de las tablas.

Si elegimos cualquiera de estas 3 opciones, la pantalla a la que accedemos tendrá una apariencia similar:

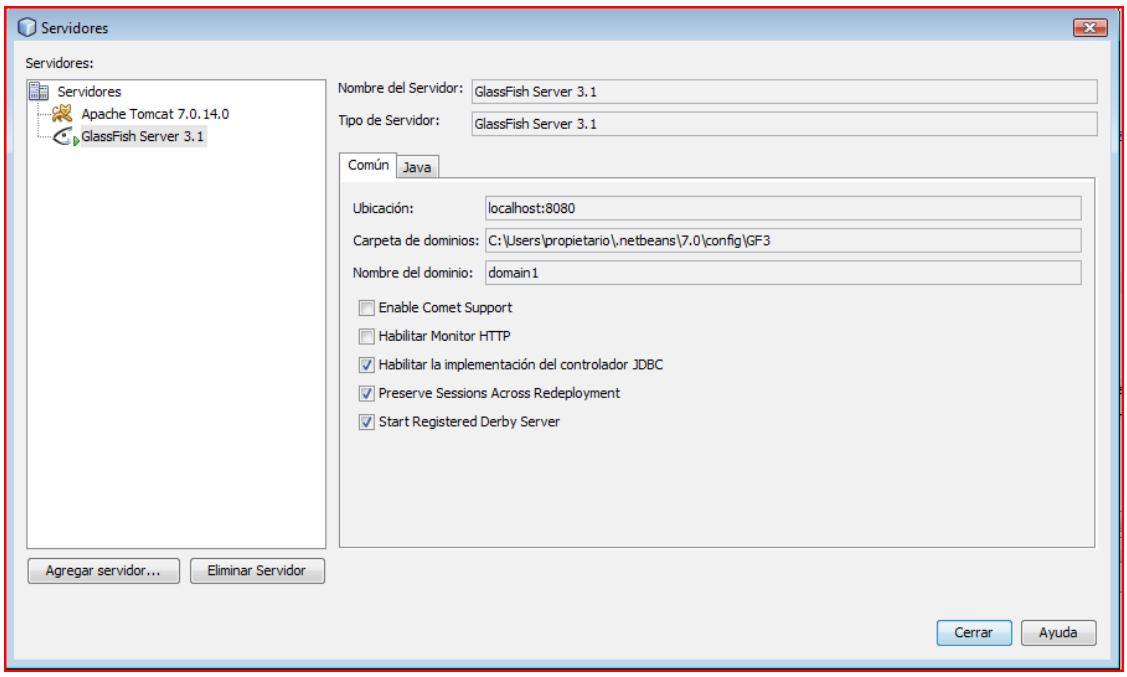


En esta pagina se muestran los xml generados mediante la opción 1 (generar xml) ya que los ficheros generados se generan a partir de estos xml. Solo queda elegir los ficheros xml y los archivos deseados (en este caso clases java) se generaran y almacenaran en el repositorio de datos.

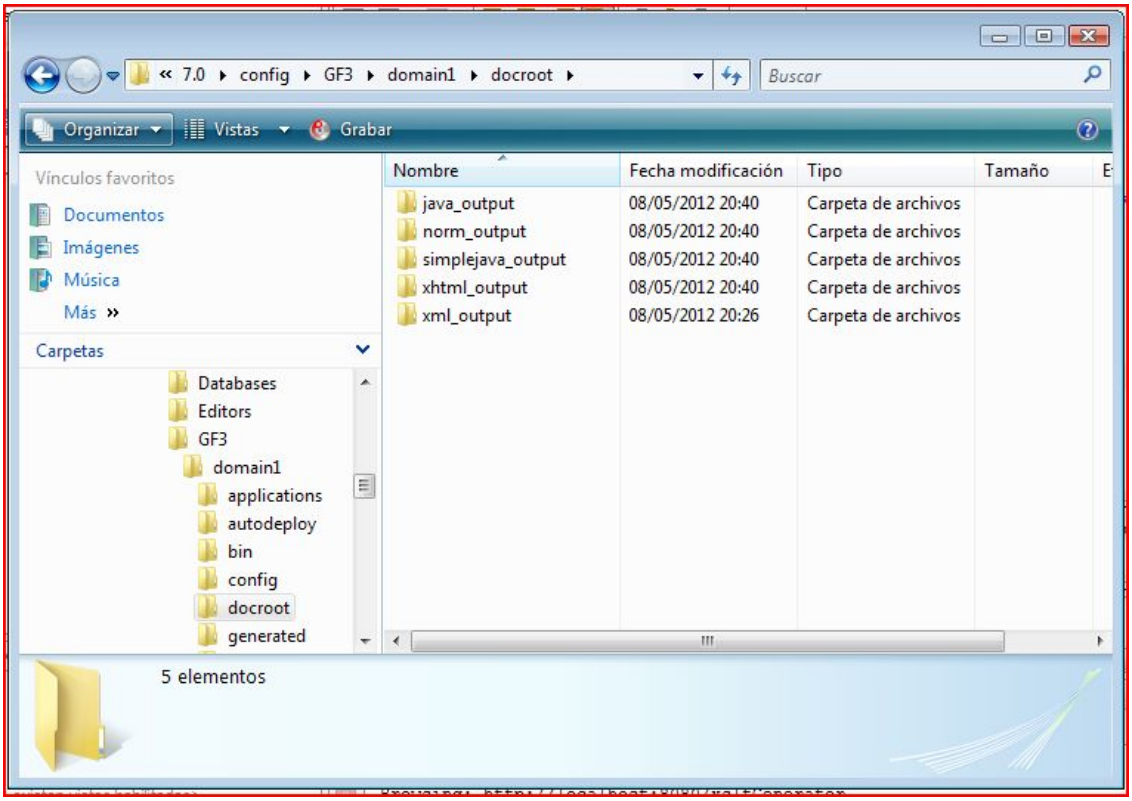
2.3- repositorio de datos

A continuación se hablara a cerca del repositorio de datos. El repositorio de datos es el lugar donde se colocan los ficheros generados por la aplicación: *.xml, *.xhtml y *.java.

En primer lugar se va a mostrar donde se encuentra el repositorio de datos y la estructura del mismo.
Para que el repositorio sea accesible desde la aplicación, hay que colocar los ficheros en un directorio accesible por la aplicación. Para ello, hay que fijarse en que ruta se encuentra el dominio de *Glassfish* en el que se ejecuta la aplicación:



Dentro del entorno de desarrollo *NetBeans*, en la pestaña "*prestaciones*", podemos acceder a los servidores. En este caso se esta usando el servidor de aplicación *Glassfish*, del cual podemos ver sus opciones. Se observa que el dominio en el que se ejecuta la aplicación es el siguiente: *C:\Users\propietario\.netbeans\7.0\config\GF3\domain1*
Dentro de esta carpeta, existe una llamada "*docroot*", en la que los ficheros generados son accesibles por la aplicación. Es aquí donde se colocaran los ficheros generados.



Al generar los archivos se crea una carpeta para cada uno de los tipos de ficheros generados.
Si se generan todos los archivos posibles que permite la aplicación, se obtendrán las 5 carpetas mostradas en la captura de pantalla de arriba:

java_output: Contiene las clases java con información detallada de las tablas.

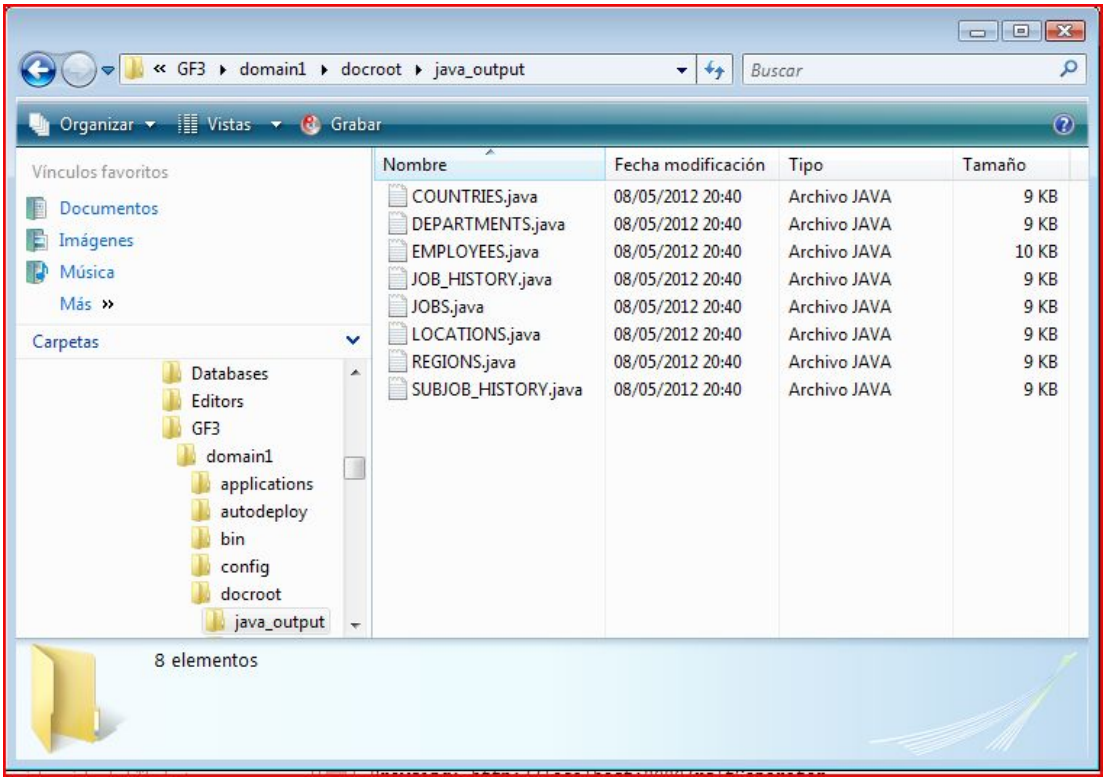
norm_output: Contiene ficheros xml intermedios usados para la generación de los archivos. Se tratan de xml en una estructura pensada para ser lo mas adecuada posible para obtener los ficheros deseados, ya que los xml que se generan a partir de la tabla están pensados para tener una estructura lo mas fácilmente interpretable por un usuario cuando los vea. Estos xml se obtiene, lógicamente, a partir de los xml que se generan con la opción 1 de la aplicación.

simplejava_output: Contiene las clases java con la información mas básica de las tablas.

xhtml_output: Contiene los ficheros xhtml generados, por cada tabla, se obtienen dos xhtml, uno para insertar y modificar datos, y otro para listar todos los datos de una tabla.

xml_output: Contiene los fichero xml generados directamente a partir de las tablas.

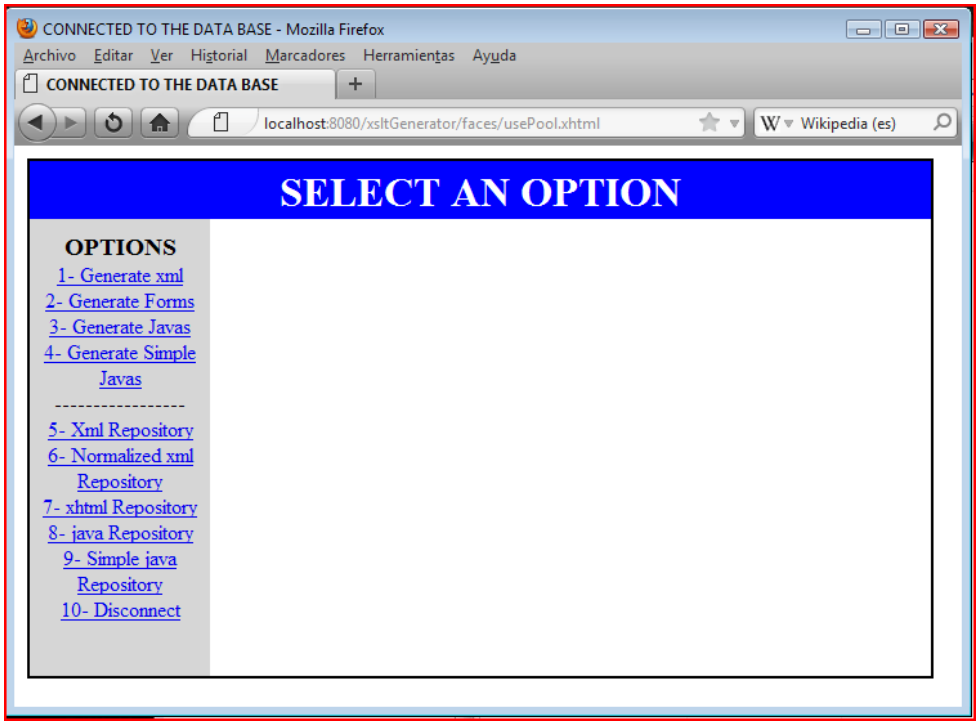
Si accedemos a una de estar carpeta, por ejemplo la carpeta *"java_output",* podemos ver en ella los ficheros que se han generado con la aplicación.



Los ficheros se generan con un nombre relacionado con la tabla a partir de los que se generan y es el siguiente según la carpeta en el que se generan:

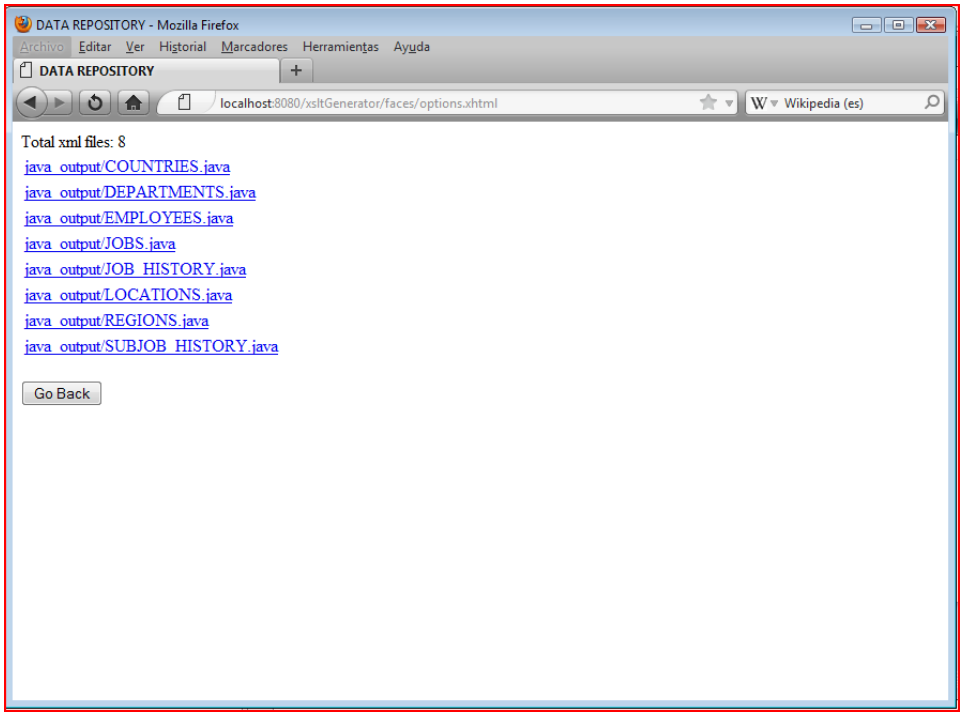
java_output: <*nombre_tabla*>*.java*
norm_output: <*nombre_tabla*>*.xml*
simplejava_output: <*nombre_tabla*>*.java*
xhtml_output: 2 ficheros por tabla: <*nombre_tabla*>*_insert.xhtml* y <*nombre_tabla*>*_select.xhtml*.
xml_output: <*nombre_tabla*>*.java*

Como se ha expuesto antes, los ficheros generados son accesibles desde la aplicación. Vamos a ver las opciones que tiene la aplicación para ello, fijándonos en la pantalla principal:

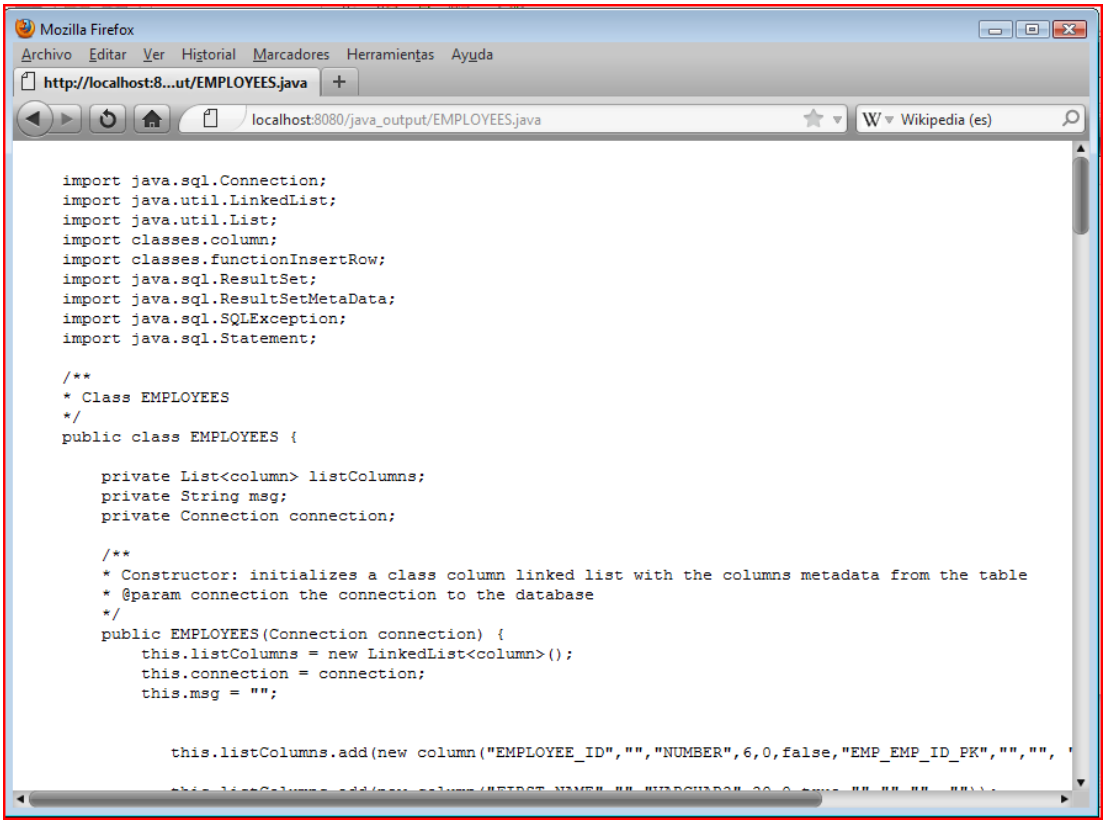


Las opciones desde el numero 5 hasta el numero 9 son la que nos permiten acceder al repositorio de datos y los ficheros generados.

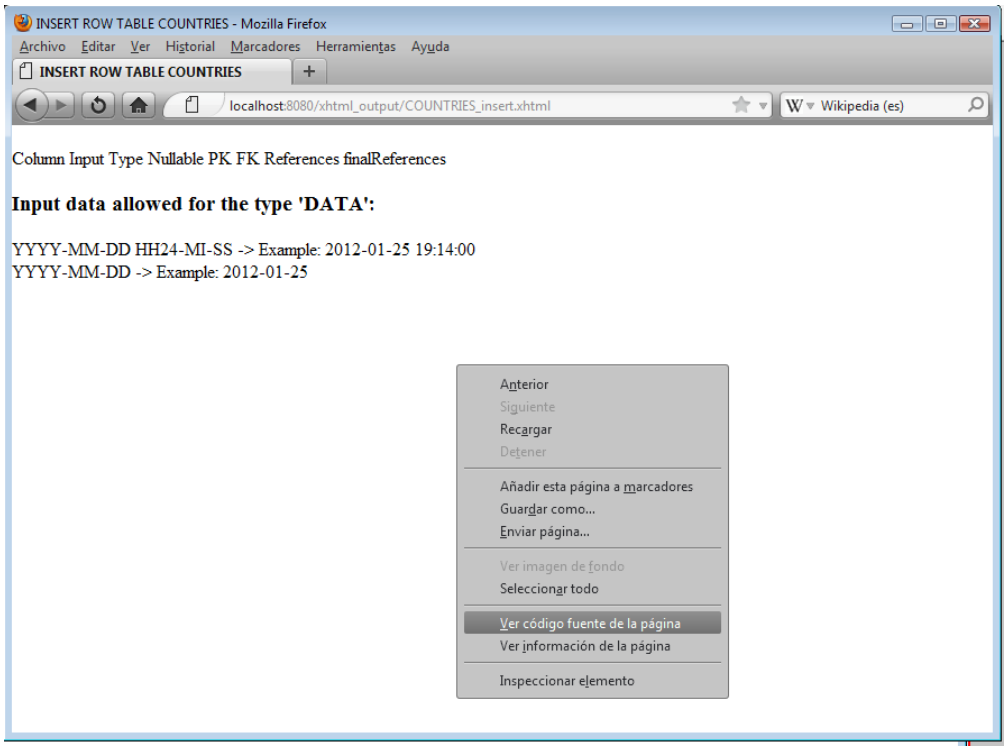
Si, por ejemplo, se accede al repositorio de javas (opción 8), se muestran los fichero contenidos en la carpeta correspondiente:



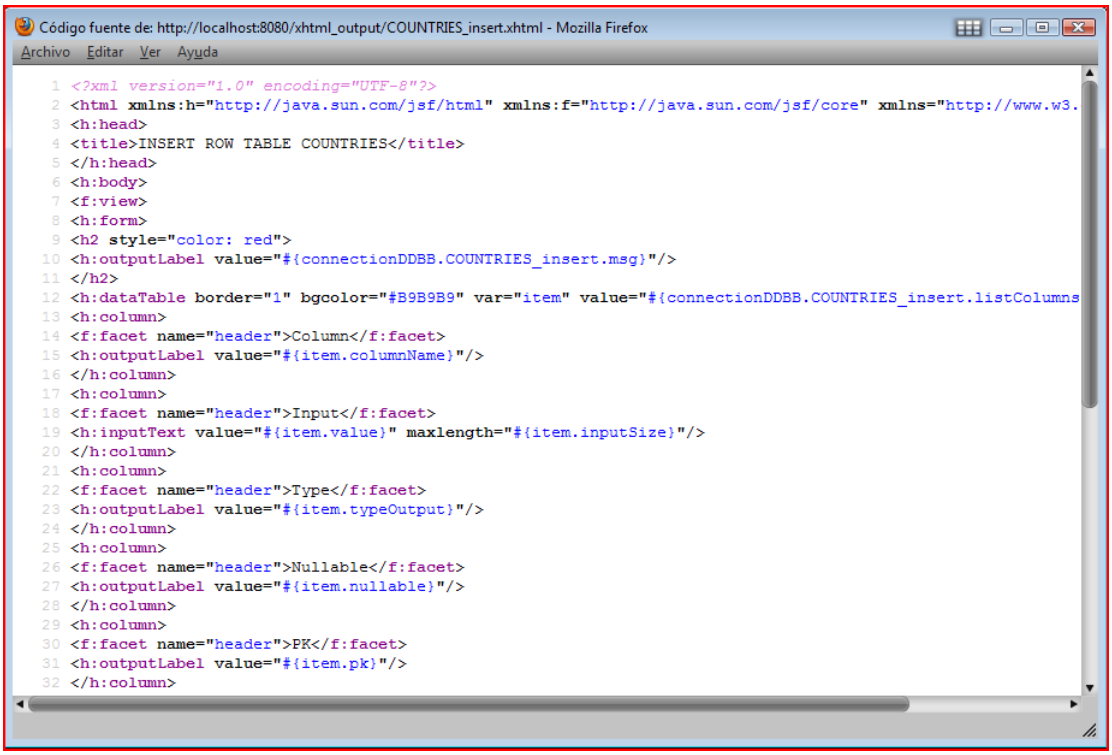
Si pulsamos sobre el fichero deseado (por ejemplo, *EMPLOYEES.java*), se accederá al contenido del mismo:



Ahora vamos a ver como acceder a los ficheros xhtml generados. En la siguiente captura se muestra la pagina que se abre al acceder a una pagina xhtml pensada para insertar datos:



La pagina mostrada no se corresponde con la que se vera a la hora de ejecutar un aplicación que use dicha pagina. Esto es debido a que las pagina *xhtml* usadas en una aplicación web necesitan datos obtenidos del control de la aplicación para rellenar los datos necesarios. Cuando se use la pagina junto a la clase java que gestiona la conexión con la base de datos (las clases java generadas por la aplicación) se podrá ver la pagina correctamente.
Si queremos acceder desde la aplicación al código del xhtml, basta con seleccionar la opción *"Ver código fuente de la pagina"*. El resultado seria el siguiente:



Hasta aquí la explicación de la aplicación *xsltGenerator*. Esta aplicación se centra únicamente en la generación de los archivos.
Una vez que tenemos todos los ficheros generados, solo tenemos que acceder al directorio de los datos y usarlos en una aplicación.
Mas adelante se hablara de la aplicación *xsltUse*, en la que se usan los ficheros generados para acceder a la base de datos.

3- Estructura de los ficheros generados:

A continuación se muestra la estructura de los xml que se generan a partir de los meta datos de las tablas de la base de datos. Para la generación de los archivos finales se aplica una primera transformación XSL del xml con la información de las tablas a otro xml con una estructura mas fácil de procesar por la aplicación. Estos xml son los que aquí de denominan *"xml normalizados"*. Estos xml también se guardan, a pesar de ser pasos intermedios en la generación de archivos, para que se disponga del mayor numero de ficheros de salida pensando en posibles reutilizaciones de los mismos por otras aplicaciones o variaciones de esta aplicación.

3.1- xml generados

La estructura de un xml generado tendrá una apariencia como la siguiente:

```
-<xml>
  -<TABLE catalog="" name="TABLA_1" schema="ESQUEMA_1">
     <!-- Información de la clave primaria -->
     -<PRIMARYKEY name="TABLA_1_PK">
        <COLUMN columnName="COL_1"/>
        <COLUMN .../>
      </PRIMARYKEY>
     <!-- Información de las columnas -->
      <COLUMN name="COL_1" nullable="0" size="6" sizeDec="0" type="NUMBER"/>
      <COLUMN name="COL_2" nullable="0" size="7" sizeDec="" type="DATE"/>
      <COLUMN .../>
      <!-- Información de las claves ajenas -->
     -<FOREIGNKEY name="TABLA_1_FK" referencedTable="TABLA_2">
        -<COLUMN columnName="COL_2"">
          <FKMETADATA depth="1" name="COL_1" nullable="0" size="7" sizeDec=""
                      table="TABLA_2" type="DATE"/>
        </COLUMN>
     </FOREIGNKEY>
  </TABLE>
</xml>
```

Las etiquetas y atributos guardan los siguientes valores:

+TABLE: Una etiqueta por tabla. Contiene la información de las columnas de la misa así como sus restricciones.
catalog = Catalogo de la base de datos a la que pertenece la tabla.
name = Nombre de la tabla.
schema = Esquema de la base de datos a la que pertenece la tabla.

+PRIMARYKEY: Contiene la columnas que forman parte de la clave primaria de la tabla
name = Nombre de la clave primaria.

Dentro de la etiqueta PRIMARYKEY podemos encontrar una lista de etiquetas como las siguientes:
<**COLUMN columnName** = "nombre de una columna que forma parte de la clave primaria" />

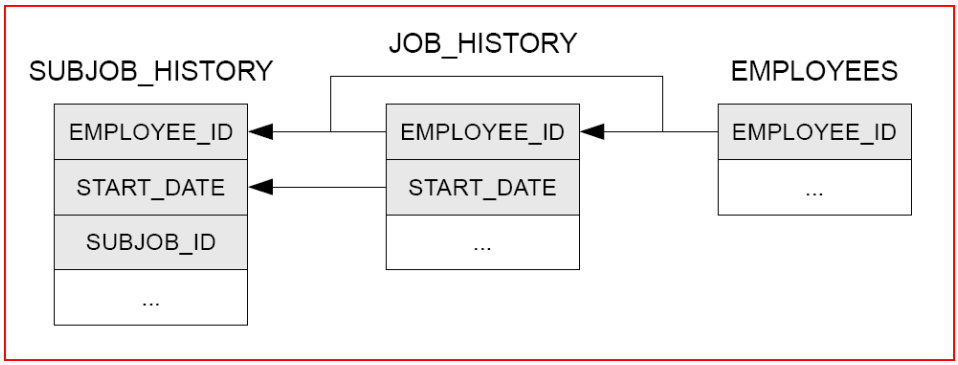
+COLUMN: Información de las columnas de la tabla.
name = Nombre de la columna.
nullable = 1 si la columna puede tener valor nulo, 0 si se requiere un valor.
size = Tamaño del tipo de la columna. Si es del tipo *NUMBER*, también incluye los decimales.
sizeDec = Decimales que puede tener el valor de la columna. Solo se usa para el tipo *NUMBER*.
type = Tipo de la columna.

+FOREIGNKEY: Contiene información a cerca de una clave ajena de la tabla.
name = Nombre de la clave ajena
referencedTable = Tabla de la que se importa la clave ajena.

Dentro de esta etiqueta encontramos las columnas que forman parte de dicha clave ajena. Pueden ser varias para el caso de que la clave importadas sea compleja (dos columnas o mas).
<**COLUMN columnName** = "nombre de la columna que forma parte de la clave importada">
</**COLUMN**>

Dentro de la etiqueta **COLUMN** encontramos mas información sobre la columna importada, los valores aquí mostrados tiene que ver con la tabla de la que se importa la clave
<**FKMETADATA**
depth = Nivel de profundidad de la búsqueda para la clave.
Name = Nombre de la columna de la tabla de la cual se exporta la clave.
Nullable = Muestra si la columna de la tabla de la cual se exporta la clave es nullable o no.
size = Tamaño del tipo de la columna exportada.
SizeDec = Decimales posibles de la columna exportada (tipo *NUMBER*)
table = Tabla de la cual se exporta la clave
type = Tipo de la columna importada
/>

El nivel de profundidad sirve para trazar las tablas de las cuales se van exportado las clave.
Puede suceder que una clave importada, sea a su vez una clave importada en su respectiva tabla. En este caso se mostraran una lista de etiquetas **FKMETADATA**, en la que el atributo **depth** se ira incrementando a la vez que se profundiza en la búsqueda del origen final de la clave.



En el ejemplo de la base de datos mostrada en el gráfico de arriba, cuando se muestre la información de la columna “EMPLOYEE_ID” que forma parte de la clave ajena de la tabla SUBJOB_HISTORY, se generaran dos etiquetas FKMETADATA, una con nivel de profundidad 1 para la columna EMPLOYEE_ID de la tabla JOB_HISTORY, y otra de nivel de profundidad 2 para la columna EMPLOYEE_ID de la tabla EMPLOYEES, que es el origen inicial de la columna que se exporta.

A continuación se muestran un ejemplo que muestran una definición de una tabla en SQL y su xml generado, en concreto el ejemplo para la tabla SUBJOB_HISTORY mostrada en el gráfico de arriba, la cual contiene una clave triple y una clave ajena doble con una columna importada de nivel máximo de profundidad 2.

SQL:

```
CREATE TABLE "HR"."SUBJOB_HISTORY"(
   "EMPLOYEE_ID"      NUMBER(6,0) NOT NULL,
   "START_DATE"       DATE NOT NULL,
   "SUBJOB_ID"        VARCHAR2(20 BYTE) NOT NULL,
   "SUBJOB_DESC"      VARCHAR2(50 BYTE),
   "IDENTIFICADOR"    VARCHAR2(50 BYTE),
   CONSTRAINT "SUBJOB_HISTORY_PK"
            PRIMARY KEY ("EMPLOYEE_ID", "START_DATE", "SUBJOB_ID"),
   CONSTRAINT "JOB_HISTORY_ID"
            FOREIGN KEY ("EMPLOYEE_ID", "START_DATE")
            REFERENCES "HR"."JOB_HISTORY" ("EMPLOYEE_ID", "START_DATE"))
```

XML:

```
-<xml>
  -<TABLE catalog="" name="SUBJOB_HISTORY" schema="HR">
    -<PRIMARYKEY name="SUBJOB_HISTORY_PK">
       <COLUMN columnName="EMPLOYEE_ID"/>
       <COLUMN columnName="START_DATE"/>
       <COLUMN columnName="SUBJOB_ID"/>
     </PRIMARYKEY>
     <COLUMN name="EMPLOYEE_ID" nullable="0" size="6" sizeDec="0" type="NUMBER"/>
     <COLUMN name="START_DATE" nullable="0" size="7" sizeDec="" type="DATE"/>
     <COLUMN name="SUBJOB_ID" nullable="0" size="20" sizeDec="" type="VARCHAR2"/>
     <COLUMN name="SUBJOB_DESC" nullable="1" size="50" sizeDec="" type="VARCHAR2"/>
     <COLUMN name="IDENTIFICADOR" nullable="1" size="50" sizeDec="" type="VARCHAR2"/>
    -<FOREIGNKEY name="JOB_HISTORY_FK" referencedTable="JOB_HISTORY">
      -<COLUMN columnName="EMPLOYEE_ID">
         <FKMETADATA depth="1" name="EMPLOYEE_ID" nullable="0" size="6" sizeDec="0"
                     table="JOB_HISTORY" type="NUMBER"/>
         <FKMETADATA depth="2" name="EMPLOYEE_ID" nullable="0" size="6" sizeDec="0"
                     table="EMPLOYEES" type="NUMBER"/>
       </COLUMN>
      -<COLUMN columnName="START_DATE">
         <FKMETADATA depth="1" name="START_DATE" nullable="0" size="7" sizeDec=""
                     table="JOB_HISTORY" type="DATE"/>
       </COLUMN>
     </FOREIGNKEY>
  </TABLE>
</xml>
```

3.2- xml normalizados

Se observa que el xml generado separa la información de las columnas respecto a la de las restricciones, de modo que podemos identificar fácilmente los datos de las columnas y la información completa de las restricciones por otro lado. Este sistema nos ofrece un xml fácil de interpretar por un usuario y que define de una manera lo mejor estructurada posible los meta datos de la tabla.

Ahora bien, de cara a generar los fichero de salida, es mas interesante tener la información completa de cada columna en el mismo elemento del xml, incluyendo si es clave primaria o ajena.
Mediante este sistema se pierde facilidad de interpretación de la estructura de la tabla, ya que por ejemplo, no tenemos una manera rápida de ver las columnas que forman una clave ajena. Habría que ver si una columna tiene una restricción de clave ajena y luego comparar el nombre de la restricción con otras para ver si existe una clave ajena doble.
También se añade información para poder generar las consultas necesarias en la base de datos para realizar la búsqueda en profundidad de las claves. Esta información relaciona las tablas y sus claves ajenas con las tablas a las que referencia y las columnas que son importadas en ellas.

La estructura de los xml normalizados es la siguiente:

```
-<TABLE>
   <NAME> nombre de la tabla</NAME>
   -<COLUMN>
      <NAME>nombre de la columna</NAME>
      <TYPE>tipo de la columna</TYPE>
      <SIZE>tamaño del tipo de la columna</SIZE>
      <SIZEDEC>tamaño de los decimales si el tipo de la columna es NUMBER</SIZEDEC>
      <NULLABLE>1 si no puede ser nulo, 0 si lo puede ser</NULLABLE>
      <PK>Si la columna forma parte de una restricción del tipo calve primaria, el nombre de la
            misma. Si no, etiqueta vacía </PK>
      -<FK> Si la columna forma parte de una restricción del tipo calve ajena, la información
aparecerá
            en los nodos hijos, si no, etiqueta vacía y sin nodos hijos
         <FKNAME>nombre de la restricción de la clave importada a la que la columna pertenece
         </FKNAME>
         <REFERENCES>tabla a la que referencia la columna (profundidad 1) </REFERENCES>
         <FINALREFERENCES>tabla final a la que referencia la columna (profundidad máximo)
         </FINALREFERENCES>
         <FINALCOLREFERENCES>nombre de la columna en la tabla señalada en la etiqueta
                                    finalreferences que es exportada
         </FINALCOLREFERENCES>
         <FKNUM>Se usa para ordenar correctamente las columnas en los xhtml
         </FKNUM>
      </FK>
   </COLUMN>

  -<COLUMN> ... </COLUMN>
```

```
-<FKSELECT> Información para la búsqueda en profundidad de las claves
    -<FKSELECTWHERE> Información a cerca de en que tablas y columnas estas relacionadas
       <FKCOLUMN> Columna de la tabla que es clave importada </FKCOLUMN>
       <TABLEREFERENCED> Tabla referenciada por la clave </TABLEREFERENCED>
       <PKCOLUMN> Columna de la tabla referenciada que es exportada </PKCOLUMN>
       <DEPTH> Nivel de la búsqueda en profundidad </DEPTH>
     </FKSELECTWHERE>

     -<FKSELECTWHERE> …  </FKSELECTWHERE>

   </FKSELECT>

  -<FKSELECT> … </FKSELECT>

</TABLE>
```

Por ejemplo, para el xml generado anteriormente, el inicio del xml normalizado generado seria el siguiente:

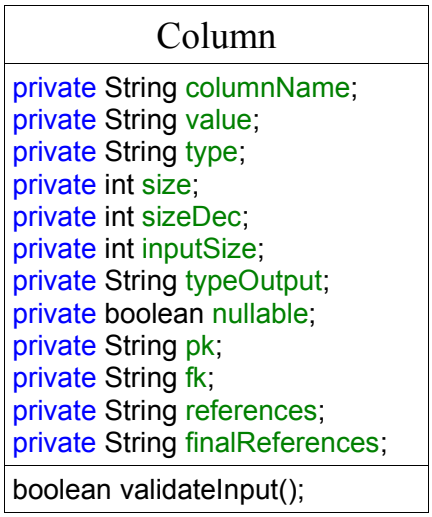
```
<TABLE>
  <NAME>SUBJOB_HISTORY</NAME>
 -<COLUMN>
    <NAME>EMPLOYEE_ID</NAME>
    <TYPE>NUMBER</TYPE>
    <SIZE>6</SIZE>
    <SIZEDEC>0</SIZEDEC>
    <NULLABLE>0</NULLABLE>
    <PK>SUBJOB_HISTORY_PK</PK>
   -<FK>
      <FKNAME>JOB_HISTORY_FK</FKNAME>
      <REFERENCES>JOB_HISTORY</REFERENCES>
      <FINALREFERENCES>EMPLOYEES</FINALREFERENCES>
      <FINALCOLREFERENCES>EMPLOYEE_ID</FINALCOLREFERENCES>
      <FKNUM>1</FKNUM>
    </FK>
  </COLUMN>
 …  (resto de columnas) …
</TABLE>
```

3.3- Ficheros java

Los ficheros java generados están pensados para guardar la máxima información posible a cerca de las tablas y para ser usados junto a las paginas xhtml generadas para interactuar con la base de datos.
Para ello, se desarrollaron dos clases auxiliares que se usaran junto a los javas generados para el correcto funcionamiento de las mismas.

3.3.1- La clase column

El fichero "*column.java*", situado en el paquete "*auxClasses*", contiene la clase "*column*":



Esta clase tiene como atributos los posibles meta datos para una columna extraíbles del xml normalizado.
El atributo *typeOutput* nos permite guardar en una cadena el tipo mas en tamaño en un formato pensado para ser presentado en un formulario. Por ejemplo, si *type* = "NUMBER", *size* = 4 y *sizeDec* = 2, entonces el valor de *typeOutput* sera "NUMBER(4,2)".
El atributo *value* es el valor de la columna. Sirve tanto como atributo de salida, después de hacer un select en la base de datos, como atributo de entrada, definiendolo antes de un insert.
Ademas, el método "*validateInput()"* nos permite ver si el atributo *value* es correcto para el tipo y el tamaño de la columna.

Esta clase se utilizara para guardar los meta datos de las columnas de una tabla como una lista enlazada de instancias de la clase "*column*".

La otra clase es *functionInsertRow* (functionInsertRow.java, en el paquete auxClasses).

functionInsertRow
private Connection connection; private List<column> listColumns; private String msg; private String tableName;
void executeInsertRow();

Esta clase es la que nos permitirá construir la sentencia SQL necesaria para introducir una tupla en una tabla de la base de datos, invocándola desde la clase java generada y pasándole como parámetros la lista con la información de las columnas y la conexión a la base de datos.

Una vez descritas las clases auxiliares, se explica la estructura de las clases java generadas:

Nombre_de_la_Tabla
private List<column> listColumns; private String msg; private Connection connection;

Nombre_de_la_Tabla(Connection);
void insertRow();
void searchRow();
void updateRow();

Como ya se ha explicado antes, los detalles de los meta datos de las columnas irán en la lista enlazada "*listColumns*".
Cuando se inicialice la clase, habrá que pasarle como parámetro el objeto "*connection*" que gestiona la conexión con la base de datos.
La lista enlazada se inicializara de la siguiente forma:

```
this.listColumns = new LinkedList<column>();
```

Se usara el método "add" sobre la lista enlazada para añadir las columnas.

A continuación se expone un ejemplo para la tabla EMPLOYEES de como añadir columnas:

```
listColumns.add(new column("FIRST_NAME","","VARCHAR2",20,0,true,"","","", ""));
//Columna FISRT_NAME, del tipo VARCHAR", con tamaño 20 y nullable

listColumns.add(new column("EMPLOYEE_ID","","NUMBER",6,0,false,"EMP_EMP_ID_PK","","", ""));
//Columna EMPLOYEE_ID, forma parte de la primary key  EMP_EMP_ID_PK

listColumns.add(new column("JOB_ID","","VARCHAR2",10,0,false,"","EMP_JOB_FK","JOBS", "JOBS"));
//Columna JOB_ID, es una clave importada de la tabla JOBS, con tabla final igual a JOBS (al ser la misma
significa que la clave tiene profundidad 1). La columna forma parte de la clave importada EMP_JOB_FK.
```

El parámetro “*msg*” servirá para guardar los mensajes resultantes de interactuar con la base de datos.
El método “*insertRow()*” llamara a la clase auxiliar *functionInsertRow* y nos permitirá insertar una tupla en la base de datos usando los valores del atributo *"value"* de cada columna.
"*searchRow()*" permite cargar una tupla de la tabla que coincida con los valores definidos para una tupla (Los valores en blanco se ignoraran).
Finalmente, el método "*updateRow()*" sirve para modificar una tupla de la base de datos. La mejor opción para modificar una tupla es usar el método “s*earchRow()*” definiendo los valores de las claves primarias, y una vez cargada la tupla en el formulario xhtml, modificar los campos deseados antes de usar el método.

3.4- Ficheros java simplificados

Puede darse el caso de que interese obtener una clase java mas simplificada de la estructura de la tabla, sin necesidad de tener una información tan amplia de como son los meta datos de la base de datos y fijándose únicamente en las columnas de la tabla.

En este caso se obtendría una clase con la siguiente estructura:

Nombre_de_la_Tabla
private [int/String] nombre_Columna_1; private [int/String] nombre_Columna_2; ... private [int/String] nombre_Columna_n;

Esta clase no contiene métodos a parte de los gettes y los setter.
Los atributos son los nombre de las columnas, y el tipo debe ser deducido a partir de la base de datos. Si el tipo de la columna en la base de datos es *NUMBER*, entonces el tipo del atributo java sera *int*, en cualquier otro caso sera *String*.

3.5- Formularios para la inserción de datos

Los formularios para la inserción de datos son paginas xhtml que tendrán el siguiente nombre: *"<nombre_de_la_tabla>_insert.xhtml"*

En la etiqueta html se define el tipo de documento:

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns="http://www.w3.org/1999/xhtml">
```

Esta cabecera define el documento como una pagina xhtml para usar junto al framework JSF (Java Server Faces).
El cuerpo de la pagina esta contenido dentro de una formulario de JSF.
En la parte superior de la pagina se coloca un texto de salida que se enlaza con el atributo "msg" de la clase, y que mostrara los mensajes resultantes de interactuar con la base de datos.
El formulario para la inserción de datos se constituye usando una *dataTable* enlazada con la lista de columnas de la clase java.
La tabla de datos se estructurara en 8 columnas. Los valores de las columnas serán extraídos de cada columna de la lista y son los siguientes:

-Column: Nombre de la columna.
-Input: Caja de texto para insertar datos.
-Type: Tipo de la columnas.
-Nullable: Columna nullable o not null.
-PK: Si forma parte de la clave primaria, nombre de la clave.
-FK: Si forma parte de una clave ajena, nombre de la clave.
-References: Tabla a la que referencia si es una clave ajena.
-finalReferences: Tabla final a la que referencia si es una clave ajena.

Todos los campos son *outputLabels* (texto de salida), menos la columna "*input"*. Esta columna es una caja de texto de entrada con máxima longitud para los datos insertados igual a la máxima longitud del tipo.

Al final del formulario existen tres botones, vinculados con las acciones de la clase java para insertar una tupla, buscar una tupla y actualizar una tupla.

3.6- Formularios para la visualización de datos

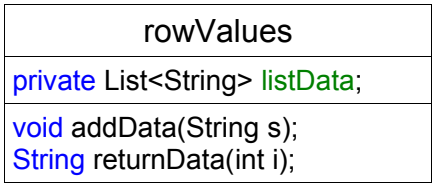
Los formularios para la inserción de datos son paginas xhtml que tendrán el siguiente nombre: *"<nombre_de_la_tabla>_select.xhtml"*.

Al igual que las paginas de inserción de datos, se tratan de paginas xhtml para usar junto al framework JSF.
En primer lugar hay información de estilo, dentro del elemento style, para aplicar a los datos mostrados y cabeceras de las columnas dependiendo del tipo de dato mostrado: clave primaria, clave ajena, información adicional para una clave ajena, cabeceras...
Hay que pulsar el botón para cargar los datos correctamente, ya que si no se pueden mostrar datos de anteriores consultas a la base de datos.
En la parte superior de la pagina se encuentra el botón para cargar los datos de la tabla. El botón se enlaza con una función auxiliar *executeSelect()* que sera la encargada de realizar la consulta.
Para las labores de obtener información adicional de las claves importadas se pasa una cadena al parámetro "*propertySelectQuery*" con los datos adicionales para realizar la consulta.
A continuación se enlaza una dataTable con los valores devueltos por la consulta, los cuales se encontraran en la clase "*funcionSelect*".
La tabla de datos mostrada en el xhtml tendrá tantas columnas como la tabla de la base de datos mas un numero igual a las claves ajenas de la tabla. Estas columnas adicionales tendrán información adicional sobre los valores de las claves ajenas importadas para cada tupla mostrada de la base de datos.

3.6.1- Ficheros auxiliares para los formularios de visualización de datos.

Las siguientes clases son necesarias para el correcto funcionamiento de los formularios y clases generadas para el uso que se les desea dar (insertar, modificar y consultar datos de las tablas). La aplicación que use dichos ficheros generados deberá incluir también estas clases auxiliares.

La primera clase auxiliar que se explica aquí es la que gestiona los datos devueltos por una consulta SQL y que permite mostrarlos en el formulario de visualización de datos:



Como vemos es una clase cuyo único parámetro es una lista enlazada de cadenas. Dicha lista contendrá los valores de las columnas de una tupla extraída de la base de datos.
Por tanto, al realizar una consulta SQL para ver todos los datos de una tabla, los valores devueltos se almacenaran en una lista de instancias de la clase rowValues.

La siguiente clase es una ayuda a la hora de estructurar las consultas SQL necesarias para las búsqueda de claves en profundidad.

selectData
private String foreirgKey; private String tableReferenced; private String tableReferencedPK; private int depth;

Esta clase no tiene métodos (aparte de los getters y setters), ya que se usara para crear una lista con instancias de la misma en la clase que ayudara a definir la manera en la que se estructura la consulta SQL.

Por ultimo, esta la clase que genera las consultas *select* en la base de datos. La clase incorpora los métodos y estructuras necesarias para poder realizar la búsqueda en profundidad de las claves y extraer los campos identificativos de las mismas en sus tablas originales.

functionSelect
private Connection connection; private List<rowValues> rowsValues; private List<String> selectQuerys; private String tableName; ...
functionSelect(Connection connection, String selectQuery); Private void restructureQuery(String query); Private void generateQuery(); Private void generateFinalSelectQuery(); Private void executeSelect(); Private void executeComplexQuery();

Este es un resumen del contenido de la clase. Debido a la complejidad de la clase y del proceso seguido para generar las consultas para la extracción de datos de las claves ajenas, aquí solo se hará una breve explicación del proceso. Mas adelante sera detallado de manera mas exhaustiva.

El atributo *connection* es el necesario para gestionar la conexión con la base de datos. *RowValues* serán los valores devueltos por la consulta y la lista *selectQuerys* almacenara todas las subconsultas necesarias para obtener la consulta final.
El atributo de entrada *selectQuery* es una cadena con una codificación especifica que contiene la información para las consultas y que ha sido generada en la transformacion XSLT que genera el xhtml para la visualización de datos.
Los métodos de la clase son privados, ya que cuando se instancia la clase se genera la consulta y se ejecuta automáticamente.
Al llamar al constructor, se ejecutara el método *"restructureQuery()"*, que decodificara la cadena "*query*" de entrada y la almacenara en unas estructuras internas para su posterior utilización.
Luego el método "*generateQuery()"* usara dichas estructuras para formar subconsultas.
El método "*generateFinalSelectQuery()"* generara la consulta final que use los valores devueltos por las subconsultas.
Finalmente, el método *executeComplexQuery* lanzara la consulta contra la base de datos y almacenara los valores devueltos para su posterior uso.

Puede darse el caso de que se quieran visualizar los datos de una tabla que no tenga claves ajenas. Este caso se detectara en el método "*restructureQuery()"*, y dado a que el proceso para generar la consulta se simplifica mucho, se llamara directamente al método "*executeSelect()"*.

Ademas de estas clases auxiliares, también hay que incluir un archivo de propiedades (myKeys.properties) a la aplicación que use los ficheros generados que contenga la siguiente clave:

infoColumn = identificador

La clase java necesaria para cargar dichas propiedades sera la misma que la usada para la aplicación y también se encuentra en el paquete con las clases java auxiliares.

El valor de la clave "infoColumn" es el nombre de la columna de las tablas que se usa como identificador del valor de una tupla y que se usara a la hora de mostrar información adicional a cerca de las claves importadas.

4- Procesos internos:

En esta sección se detallaran los procesos internos mas importantes que se dan en la aplicación para generar los ficheros deseados

4.1- Fichero myKeys.properties

El fichero "*myKeys.properties"* situado en el paquete *properties* define una serie de constantes del tipo cadena que se usan a la hora de generar los xml a partir de la base de datos, gestionar el repositorio de datos y aplicar las transformación XSL.
Para obtener las cadenas definidas aquí se usa la clase *"myProperties.java"*, situado en el mismo paquete.

Extracto de parte del fichero *myKeys.properties:*

```
(...)
# Keys used as a tags or attribute in the xml files
xmlTagTable = TABLE
xmlTagColumn = COLUMN
xmlTagPrimaryKey = PRIMARYKEY
xmlTagForeignKey = FOREIGNKEY
xmlTagForeignKeyMetaData = FKMETADATA
xmlAttribName = name
xmlAttribCatalog = catalog
xmlAttribSchema = schema
xmlAttribName = name
xmlAttribSize = size
xmlAttribSizeDec = sizeDec
xmlAttribType = type
xmlAttribNullable = nullable
xmlAttribColumnName = columnName
xmlAttribReferencedTable = referencedTable
xmlAttribTable = table
xmlAttribDepth = depth
(…)
```

Las propiedades que empiezan por *"xmlTag..."* definen el nombre que recibirá la etiqueta correspondiente en el xml de salida.
Por ejemplo, la propiedad *xmlTagColumn"*con valor *COLUMN* hace que las etiquetas que corresponden a información de columnas se llamen "*COLUMN*". Si a *xmlTagColumn* le asignamos otro valor, las etiquetas de los xml generados tendrían el nuevo valor asignado.
De manera análoga, las propiedades que empiezan por *"xmlAttrib..."* definen el nombre que tendrán los atributos de las etiquetas que los contienen.

De este modo, podemos cambiar el nombre de las etiquetas y sus atributos de los xml generados solo con modificar los valores de las propiedades, sin tener que modificar nada del código original.

La siguiente sección del fichero *"myKeys.Properties"* se usa para definir las rutas del repositorio de datos:

```
(...)
# Directory to save the output data
outputDirectory = docroot/xml_output/
outputRepositoryDirectory = xml_output/
outputNormalizeDirectory = docroot/norm_output/
outputNormalizeRepositoryDirectory = norm_output/
outputXhtmlDirectory = docroot/xhtml_output/
outputXhtmlRepositoryDirectory = xhtml_output/
outputJavaDirectory = docroot/java_output/
outputJavaRepositoryDirectory = java_output/
outputSimpleJavaDirectory = docroot/simplejava_output/
outputSimpleJavaRepositoryDirectory = simplejava_output/
(...)
```

Las propiedades *outputDirectory* y *outputRepositoryDirectory* controlan la dirección , dentro del dominio del servidor de aplicaciones, donde se encuentran los ficheros xml de salida generados. Los ficheros has de ser generados en el directorio docroot (En el caso del servidor de aplicaciones *Glassfish*) para que sean accesibles desde la aplicación.
Por cada tipo de fichero generado mediante transformaciones, existen 2 propiedades: *output<tipo_fichero>Directory* y *output<tipo_fichero>RepositoryDirectory*.
Se pueden cambiar las carpetas en las que se sitúan los ficheros de salida modificando los valores de estas propiedades. Por ejemplo, si queremos que los ficheros java se sitúen en la carpeta "misJavas" dentro del directorio docroot, deberíamos modificar los siguientes valores:

```
outputJavaDirectory = docroot/misJavas/
outputJavaRepositoryDirectory = misJavas/
```

La ultima sección del fichero de propiedades guarda las rutas de los ficheros *.xsl* que contiene las reglas para aplicar las transformaciones:

```
# XSLT Files Path
XSLTNormalize =
C:/Users/propietario/Documents/NetBeansProjects/xsltGenerator/src/java/XSLT/NormalizeXML.xsl
XSLTtoForm =
C:/Users/propietario/Documents/NetBeansProjects/xsltGenerator/src/java/XSLT/XSLTtoForm.xsl
XSLTtoSelect =
C:/Users/propietario/Documents/NetBeansProjects/xsltGenerator/src/java/XSLT/XSLTtoSelect.xsl
XSLTtoJava =
C:/Users/propietario/Documents/NetBeansProjects/xsltGenerator/src/java/XSLT/XSLTtoJava.xsl
XSLTtoSimpleJava =
C:/Users/propietario/Documents/NetBeansProjects/xsltGenerator/src/java/XSLT/XSLTtoSimpleJava.xsl
```

Estas rutas señalan la ubicación física donde se sitúan los ficheros *.xsl* contenidos en la aplicación, dentro del paquete XSLT.
Es necesario que la propiedad adecuada tenga el valor exacto de la ruta del fichero para poder aplicar la transformación durante la ejecución del programa.

4.2- Generar los xml

Algunos métodos usados a la hora de obtener los meta datos de las tablas no son muy eficientes en tiempo, particularmente el método "*getImportedKeys(**String** catalog, **String** schema, **String** table)*" definido en el interface *DatabaseMetaData* del paquete *java.sql*.
Dicho método se usa para, dada una tabla pasada como parámetro, obtener las columnas (y su información) de la base de datos que son importadas en la tabla.
En la aplicación es necesario usar este método, ya que se busca la máxima información a cerca de la estructura de una tabla. Ademas, para cada tabla puede que haya que invocar el método varias veces para investigar en profundidad las claves importadas.
Debido a esto, una vez que se ha seleccionado las tablas a transformar en xml, se lanza un *thread* por cada una para obtener los datos de la base de datos, logrando así un incremento del rendimiento en tiempo a la hora de generar los xml, al poder generarlos de manera paralela.

El proceso para generar los xml se divide en dos partes: Extraer la información de la base de datos y convertir la información en un xml.

4.2.1- Extraer los meta datos de la base de datos

ThreadMetadataBD es la clase que extrae los meta datos para cada tabla. Dicha clase se ejecuta como un hilo por cada tabla.

La conexión a la base de datos ha sido definida en el interface *Connection* del paquete java.sql.
Para obtener los meta datos de las tablas de la base de datos, se usan principalmente 4 métodos.

El primero es el método *"DatabaseMetaData getMetaData()",* definido para el interface *Connenction*. Este método devuelve un objeto del tipo "*DatabaseMetaData*" que contiene los meta datos de la base de datos. Este objeto incluye información acerca de las tablas de la base de datos.

A continuación, para la tabla deseada hay que extraer la información de las columnas, las claves primarias y las claves importadas. Para ello se usan los siguientes métodos, definidos para la clase "*DatabaseMetaData".*

"ResultSet getColumns(String catalog, String schema, String table, String column)"

Esta función nos devuelve la información deseada de las columnas para el elemento de la base de datos definido en los parámetros. En nuestro caso interesa la información de las columnas de una tabla, por lo que el método será invocado con los siguientes parámetros de entrada:
getColumns(null, null, tabla_deseada, null);

"ResultSet getPrimaryKeys(String catalog, String schema, String table)"

Este método nos devuelve la información de las columnas que forman parte de la clave primaria de la tabla definida en los parámetros de entrada.

"ResultSet getImportedKeys(String catalog, String schema, String table)"

Este ultimo método nos devuelve la descripción de las columnas que son clave primaria de las tablas referenciadas por las claves ajenas de la tabla.

La estructura que guarda todos estos meta datos es una lista enlazada de cadenas ("*LinkedList<String>()*").
Para cada tipo de meta dato extraído de la base de datos, se introduce en la lista una cadena que identifica el meta dato, y a continuación las cadenas con los valores para dicho dato.
A continuación se exponen los tipos de meta datos que se extraen (En negrita, la cadena clave que define el dato).
Para cada tipo de dato hay un numero fijo de cadenas que le siguen con la información correspondiente al mismo:

Linea i → **primaryKey** (Información de una clave primaria) *Linea i+1* → Nombre de la columna que forma parte de la clave primaria. *Linea i+2* → Nombre de la clave primaria.
Linea i → **catalog** (Catalogo al que pertenece la tabla) *Linea i+1* → Nombre del catalogo.
Linea i → **schema** (Esquema al que pertenece la tabla) *Linea i+1* → Nombre del esquema.
Linea i → **columnRow** (Información de los datos de una columna de la tabla) *Linea i+1* → Nombre de la columna. *Linea i+2* → Tipo. *Linea i+3* → Longitud definida para el tipo. *Linea i+4* → Longitud de los decimales en el caso del tipo NUMBER, para otro tipo = null. *Linea i+5* → 0 si la columna no puede ser nula, 1 en el caso contrario.
Linea i → **importedKey** (Información de una clave ajena) *Linea i+1* → Nombre de la columna que forma parte de la clave ajena. *Linea i+2* → Nombre de la clave ajena.
Linea i → **importedKeyMetaData** (Información de los datos de una columna que es referenciada por una clave importada. Los datos aquí mostrados son los que tiene la columna en la tabla referenciada) *Linea i+1* → Nombre de la columna. *Linea i+2* → Tipo. *Linea i+3* → Longitud definida para el tipo. *Linea i+4* → Longitud de los decimales en el caso del tipo NUMBER, para otro tipo = null. *Linea i+5* → 0 si la columna no puede ser nula, 1 en el caso contrario. *Linea i+6* → Nivel de profundidad e la búsqueda de la clave.

Una vez que se ha rellenado la lista enlazada con todos los datos, se invoca a la clase "*createXML*" pasándole como parámetros la lista enlazada y el nombre de la tabla.

4.2.2- Extraer los datos de la búsqueda de claves en profundidad

La manera en la que se realiza la búsqueda de las claves en profundidad es la siguiente:
Para que una clave ajena tenga profundidad mayor que 1, hay que fijarse en la tabla de la cual se importa la clave. Si en dicha tabla, la clave primaria exportada es a su vez una importada, o tiene valores importados, habrá que ir a la tabla de la que se importan estos valores para realizar la búsqueda en profundad, y así sucesivamente.

Debido a esto, cuando se obtiene una columna que forma parte de una clave importada, se invoca al siguiente método recursivo, definido en la clase *threadMetadataBD:*

private void foreignKeySearchDepth(String table, String pkColumnImported, int depth);

A dicho método hay que pasarle la tabla referenciada por la clave ajena y la columna de dicha tabla que es importada.
En la primera ejecución del método, el valor de profundidad sera 1.
El método comprueba si la columna exportada en la tabla es a su vez una clave importada. Si no es así, el método finalizara. Si determina que la clave exportada es a su vez importada, buscara la tabla y la columna de la cual se importa la clave, y volverá a ejecutar el método para la tabla y la columna que importa el dato, con un nivel de profundad igual al anterior + 1. El nivel de profundidad se usa para añadirlo a la información de las claves ajenas de los xml

4.2.3- Generar el xml

La clase *createXML* usa la lista enlazada generada con los meta datos para crear los xml con la información de las tablas.
Para crear los xml nos ayudamos de la librería java.xml.* que nos permite crear los nodos definiendo su etiqueta y sus atributos. También nos permite definir la jerarquía de los nodos añadiendo nodos hijos a otros ya existentes.

Para generar el xml basta con recorrer la lista, identificar cuando aparece una cadena que determina la creación de un nodo, crearlo con la etiqueta adecuada y recorrer los siguientes valores de la lista para rellenar los atributos del nodo.

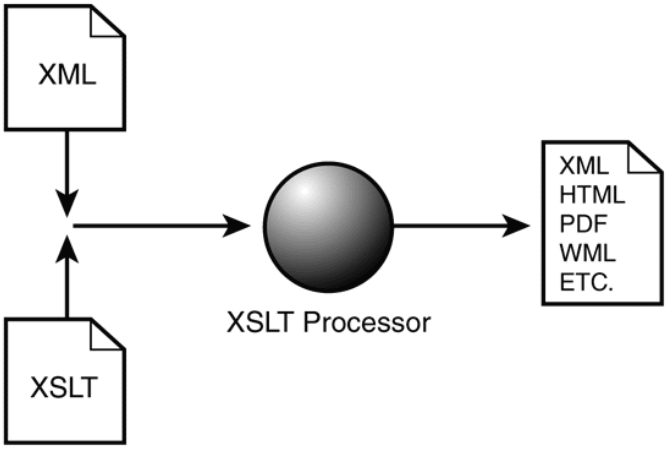
El único caso en el que hay que tener un cuidado especial es para los nodos que definen claves importadas. Como es posible que existan claves importadas complejas (que contengan dos o mas columnas), hay que chequear el valor del nombre de una clave ajena cada vez que aparezca en la lista. Si la clave no existe en el xml, se creara un nodo nuevo para la clave y su subsiguiente información. Si la clave ya existía, hay que incorporar los valores del nodo dentro del ya existente correspondiente a dicha clave ajena.

4.3- Transformaciones XSL:

Una vez que tenemos los xml generados, usaremos transformaciones XSL para generar los archivos de salida.

XSLT o Transformaciones XSL es un estándar de la organización W3C que presenta una forma de transformar documentos xml en otros e incluso a formatos que no son xml. Las hojas de estilo XSLT realizan la transformación del documento utilizando una o varias reglas de plantilla. Estas reglas de plantilla unidas al documento fuente a transformar alimentan un procesador de XSLT, el que realiza las transformaciones deseadas poniendo el resultado en un archivo de salida.

El esquema para la generación de los ficheros de salida es el siguiente:



Los archivos xml son los generados y explicados anteriormente.
Los archivos XSLT contiene las reglas para transformar el fichero xml en el de salida. El formato del fichero de salida se especifica en el archivo XSLT. Estos archivos XSLT serán detallados en la siguiente sección de este documento, y se encuentran en el paquete XSLT de los fuentes de la aplicación.

Dentro de la clase *"xmlRepository.java"*, existen las siguientes tres funciones:

```
"private void xsltForm(String xmlFile)"
"private void xsltJava(String xmlFile)"
"private void xsltSimpleJava(String xmlFile)"
```

Cada uno de los métodos sirve para transformar un xml en el fichero de salida deseado. Para ello primero se realiza un transformación del xml original al xml normalizado. A continuación se usa el xml normalizado para generar el fichero de salida deseando usando la plantilla XSLT adecuada.

El proceso que sigue la aplicación para realizar una transformación XSLT es el siguiente:

Para generar el xml normalizado necesitamos que los dos ficheros de entrada sean un xml original y la plantilla XSLT *"NormalizeXML.xsl"*, que define las reglas para generar el xml normalizado de salida.
Las dos clases java que nos permiten realizar las transformaciones están contenidas dentro del paquete *javax.xml.transform* y son la clase Transformer y la clase TransformerFactory.

El código comentado para realizar la transformación es:

```
File directory = new File(myProperties.getProperty("outputNormalizeDirectory"));
directory.mkdir();
// En primer nos aseguramos de que el directorio para el fichero de salida este creado.
File fileOut = new File(myProperties.getProperty("outputNormalizeDirectory").concat(xmlFile));
Result result = new StreamResult(fileOut);
// xmlFile contiene el nombre del fichero de salida. Se crea el fichero de salida en el directorio de salida deseado y se le asigna a un flujo de salida.
File fileIn = new File(myProperties.getProperty("outputDirectory").concat(xmlFile));
Source source = new StreamSource(fileIn);
// xmlFile contiene el nombre del fichero xml de entrada (En este caso el fichero de salida y el de entrada tiene mismo nombre, pero se encuentran en distinta carpeta ). Se abre el fichero xml de entrada y se le asigna a un flujo de entrada.
TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer =
tFactory.newTransformer(newStreamSource(myProperties.getProperty("XSLTNormalize")));
// Se crea una nueva instancia de la clase TransformerFactory. La instancia "transformer" de la clase "Transformer" es la que ejercerá de procesador XSLT. Para ello usamos el método newTransformer() usando como parámetro la ruta del fichero xsl deseado. En este caso el fichero  normalizeXML.xsl cuya ruta esta definida en la propiedad "XSLTNormalize"
transformer.transform(source, result);
// El método transform de la instancia del procesador XSLT es el que ejecuta la transformación. Usa el flujo de entrada (el fichero xml) y guarda el resultado en el flujo de salida (El fichero xml normalizado)
```

El proceso en análogo para todas las transformaciones, solo hay que especificar el fichero de entrada (un xml o un xml normalizado), el fichero de salida deseado (concatenandole la extensión deseada) y instanciar la clase *Transformer* con la plantilla XSLT adecuada.

4.3.1- Normalizar los xml

La plantilla usada para transformar un xml en un xml normalizado es *"NormalizeXML.xsl"*.
El proceso que se sigue para generar el xml normalizado es el siguiente:
Primero se crean dos estructuras que guardan la información de las claves primarias y las claves ajenas.
La estructura para las claves primarias se almacena en el parámetro "*pkList*". Aquí se almacenan una lista de elementos con la siguiente estructura:

```
<pk pkName="nombre de la clave primaria" pkColumn="nombre de la columna"/>
```

La información de las claves ajenas se guarda en el parámetro "*fkList*". En este caso se almacenan nodos con los siguientes atributos:

```
<fk fkColumn="columna que es clave ajena" fkName="nombre de la clave ajena"
    fkTable="tabla referenciada" fkFinalTable="Tabla final referenciada"
    fkFinalTableColumn="Columna referenciada de la tabla final"/>
```

Una vez que tenemos estos dos parámetros definidos, podemos construir el xml normalizado, en el que se busca agrupar toda la información de una columna dentro de un nodo.
Basta con recorrer los elementos con el tag "*COLUMN*" de los xml, usar la información contenida en ellos, y luego recorrer las dos listas y comparar el nombre de la columna con los nombres de columna que se encuentran en los elementos de ellas. Si coincide, se añade la información de clave correspondiente.

Para recorrer la lista con los valores de la calve se usa la siguiente función *"exsl:node-set()"*. Para ello se añadio el espacio de nombres "exsl" a la etiqueta <xsl:stylesheet>.
A continuación un ejemplo de como se recorre el parámetro "*pkList*" y se guarda en la variable "*unPk*" el nombre de la columna de la primary key.

```
<xsl:for-each select="exsl:node-set($pkList)/pk" >
<!-- Extrae un elemento con tag pk-->
  <xsl:variable name="unPk">
    <xsl:value-of select="@pkColumn"/>
    <!-- Extrae el atributo pkColumn del elemento y lo almacena en la variable "unPK"-->
  </xsl:variable>
<xsl:for-each/>
```

Recorriendo la lista "*pkList*" y "*fkList*" para cada columna que se encuentre en el xml original podemos comprobar si dicha columna forma parte de una clava primaria (*pkList*) o ajena (*fkList*) y almacenar los valores propios de la clave.

Al final de la plantilla se crea la estructura necesaria que servirá para formar las consultas en la base de datos necesarias para realizar las búsquedas en profundidad de las claves ajenas.
Por cada clave ajena que exista en la tabla se creara un nodo con tag "*FKSELECT*" que contendrá la información necesaria para la búsqueda en profundidad de la clave. Los elementos hijos estarán contenidos en elementos con tag "*FKSELECTWHERE*", el cual contiene información a cerca del criterio de búsqueda a aplicar.

4.3.2- Generar javas

Generar los java no conlleva gran dificultad .
Generaremos el fichero de salida en formato de texto.
Guardaremos en la variable table el nombre de la tabla:

```
<xsl:variable name="table">
  <xsl:value-of select="TABLE/NAME"/>
</xsl:variable>
```

Se usara el nombre de la tabla a través de la variable table para el nombre de la clase, el constructor y las referencias a una tabla de la base de datos.

En la clase se importan las siguientes clases:

```
import java.util.LinkedList;
import java.util.List;
import classes.column;
import classes.functionInsertRow;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
```

Las clases del paquete útil se usan para la lista de datos de las columnas. Las del paquete sql para interactuar con la base de datos. Finalmente las dos clases auxiliares que requiere la clase generada (*column* y *functionInsertRow*) son importadas de un paquete "classes". Estas clases auxiliares deberán estar en dicho paquete dentro de la aplicación que use este objeto. En el caso de que estuvieran en otro paquete habría que ajustar a mano la clase para importar correctamente los mismo.

El punto mas interesante a la hora de generar la clase se centra en el constructor. Aquí es donde se añadirá la información de los meta datos de las columnas a la clase:

```
public <xsl:value-of select="$table"/>(Connection connection) {
  this.listColumns = new LinkedList&lt;column>();
  <!-- &lt; = carácter '<', hay que utilizar el carácter especial para que el procesador xslt no lo tome como un elemento de la plantilla-->
  this.connection = connection;
  this.msg = "";
  <!-- Rellenar la estructura listColumns con la información de las columnas -->
  <xsl:for-each select="TABLE/COLUMN">
  <!-- Para cada columna del xml normalizado, crear un objeto column inicializandolo con los valores
       adecuados de la columna del xml y añadirlo a la lista de columnas-->
    this.listColumns.add(
      new column(
            "<xsl:value-of select="NAME"/>",
            "",
            "<xsl:value-of select="TYPE"/>",
            <xsl:value-of select="SIZE"/>,
            <xsl:value-of select="SIZEDEC"/>,
            <xsl:value-of select="NULLABLE = 1"/>,
            "<xsl:value-of select="PK"/>",
            "<xsl:value-of select="FK/FKNAME"/>",
            "<xsl:value-of select="FK/REFERENCES"/>",
            "<xsl:value-of select="FK/FINALREFERENCES"/>"
      )
    );
  </xsl:for-each>
}
```

4.3.3- Generar javas simples

En el caso de los java simplificados, solamente hay que generar los atributos (uno por columna), los setters y getters para los elementos y un constructor sin parámetros de entrada (no inicializa los atributos).

Se recorren todas las columnas del xml normalizado y se comprueba el tipo de la columna. Si el tipo es *NUMBER*, el atributo sera del tipo *float*. Si el tipo es *CHAR*, *VARCHAR2* o *DATE*, el tipo java sera *String*.

```
<!-- Forma de declarar los atributos-->
<xsl:for-each select="TABLE/COLUMN">
<!-- Para cada columna del xml normalizado, guardamos su tipo en la variable "type" -->
  <xsl:variable name="type">
    <xsl:value-of select="TYPE"/>
  </xsl:variable>
  <!-- Comprobar que tipo es-->
  <xsl:if test="$type = 'VARCHAR2'">
    <!-- Como el tipo es VARCHAR2, el parámetro sera del tipo String-->
    <!-- El valor de NAME es el nombre de la columna y sera el nombre del atributo-->
    <xsl:text>String </xsl:text><xsl:value-of select="NAME"/> <xsl:text>;</xsl:text>
  </xsl:if>
  <xsl:if test="$type = 'NUMBER'">
    <!-- Como el tipo es NUMBER, el parámetro sera del tipo float-->
    <xsl:text>float </xsl:text><xsl:value-of select="NAME"/> <xsl:text>;</xsl:text>
  </xsl:if>
  ...
  <!-- Código análogo al tipo VARCHAR2 para el tipo CHAR y DATE-->
  ...
</xsl:for-each>
```

A la hora de generar los setters y getters habrá que realizar la misma comprobación para establecer el tipo que devuelve un get y el tipo del parámetro de entrada para un set.

4.3.4- Generar formularios de inserción de datos

Los formularios generados para la inserción de datos y visualización de los mismos son paginas xhtml que se ejecutaran dentro del framework JSF. Para que las paginas tengan este formato hay que definir los siguientes elementos de la cabecera de la plantilla xsl de la siguiente manera:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:f="http://java.sun.com/jsf/core"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns="http://www.w3.org/1999/xhtml"
                version="1.0">
<xsl:output method="xml" indent="yes" encoding="UTF-8" />
```

La etiqueta *xsl:stylesheet* define el tipo de elementos que se pueden encontrar en la plantilla. El valor que va después de los dos puntos del atributo "*xmlns*" referencia a un prefijo de un elemento y ayuda a identificar el tipo del mismo.
- *xmlns:xsl* identifica a los elementos que empiezan por *xsl:* y que hacen referencia a las transformaciones XSLT.
- *xmlns:f* y *xmlns:h* referencia a elementos que empiezan por *f:* o *h:*, los cuales son elementos propios del framework JSF.
Por ultimo, el atributo *xmlns* sin sufijo hace referencia a todos los demás elementos, los cuales los asocia con el tipo de documento xhtml.
El método de salida sera xml, debido a que un xhtml es un html con una estructura XML valida.
Con estos elementos definidos y generando el archivo de salida como un xhtml, obtendremos los ficheros en el formato deseado.

La manera de asociar la tabla de datos mostrada la pagina y la estructura con la información de las columnas de la clase java correspondiente es la siguiente:

```
<xsl:variable name="class">
  #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.listColumns}
</xsl:variable>
<h:dataTable border="1" bgcolor="#B9B9B9" var="item" value="{$class}">
  <h:column> ... </h:column>
  <h:column> ... </h:column>
  ...
</h:dataTable>
```

La variable *class* contiene la ruta de la clase y su parámetro a la que se asocian los datos de la tabla. En el framework el parámetro al que se asocia un valor se señala encerrándolo entre llaves y con un corchete delante. En este caso los datos de la tabla se asociaran con la clase generada para la misma tabla, y que se encontrara definida en una clase llamada *connectionDDBB*.
En la etiqueta JSF *dataTable* asociamos el valor con la lista de la clase (etiqueta *value*) y la usaremos dentro de la estructura como el valor *item* (atributo *var*). *Item contendrá* un valor de la lista de columnas.
La etiqueta *column* contiene el valor mostrado para la columna.

Por ejemplo, el valor de la columna nombre sera:

```
<h:column>
   <!-- Cadena que aparece en la cabecera de la columna-->
   <f:facet name="header"><xsl:text>Column</xsl:text></f:facet>
   <!-- nombre de la columna-->
   <xsl:variable name="item">#{item.columnName}</xsl:variable>
   <!-- La variable item contiene el nombre de la columna-->
   <h:outputLabel value="{$item}"/>
   <!-- El nombre de la columna se mostrara como una etiqueta de texto->
</h:column>
```

De este mismo modo vamos generando las columnas de la tabla.

El único caso particular es el campo de texto disponible para introducir datos, el cual se define de la siguiente manera:

```
<h:column>
  <f:facet name="header"><xsl:text>Input</xsl:text></f:facet>
  <xsl:variable name="item">#{item.value}</xsl:variable>
  <xsl:variable name="size">#{item.inputSize}</xsl:variable>
  <h:inputText value="{$item}" maxlength="{$size}"/>
  <!-- En este caso creamos una caja de texto de entrada que admite un tamaño de caracteres igual
a la longitud del tipo  -->
</h:column>
```

Por ultimo, generaremos los botones para las acciones disponibles: insertar, buscar y modificar:

```
<xsl:variable name="action">
  #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.insertRow()}
</xsl:variable>
<h:commandButton value="Insert" action="{$action}"/>
<xsl:variable name="action2">
  #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.searchRow()}
</xsl:variable>
<h:commandButton value="Select" action="{$action2}"/>
<xsl:variable name="action3">
  #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.updateRow()}
</xsl:variable>
<h:commandButton value="Update" action="{$action3}"/>
```

Cabe destacar que en el fichero generado se da por echo que las clases java generadas se encuentran instanciadas dentro de una clase *"connectionDDBB"*. Si las clases java generadas se quieren usar dentro de otra clase, basta con modificar la plantilla cambiando *conectionDDBB* por el valor deseado.

4.3.5- Generar formularios de visualización de datos

Para generar el xhtml hay que usar las mismas definiciones para la etiqueta *xsl:stylesheet* y *xsl:output.*

La primera sección de la plantilla XSLT se centra en generar la cadena con la información necesaria para la búsqueda de información adicional para las claves ajenas en la estructura adecuada.
Dicha cadena la almacenaremos en la variable *"select"* añadiendo los valores adecuados en el orden deseado.
En primer lugar añadiremos el nombre de la tabla. A continuación añadiremos, separados por comas, los nombre de la columnas que forman parte de la clave primaria de la tabla:

```
<xsl:for-each select="TABLE/COLUMN">
<!-- Recorremos las columnas del xml normalizado, si el nodo PK tiene algún nodo hijo, entonces la columna es una clave primaria, asi que la añadimos-->
  <xsl:if test="count(PK/node()) > 0">
    <xsl:text>,</xsl:text>
    <xsl:text><xsl:value-of select="NAME"/></xsl:text>
  </xsl:if>
</xsl:for-each>
```

A continuación, para cada elemento con tag *FKSELECT*, añadimos el carácter '='. Esto significa que hay una clave ajena. Dentro de este elemento, por cada etiqueta *FKSELECTWHERE* añadiremos el carácter '|'. Este carácter informa de que a continuación van los datos obtenidos de la búsqueda en profundad de dicha clave importada.
Los datos que añadiremos a la cadena serán, separados por comas:
Columna que es clave importada, tabla a la que referencia, columna referenciada y nivel de profundad de los datos.

Una vez generada la cadena, se genera la etiquete style en la que se hayan los estilos *css* definidos para la pagina.
Ahora nos centraremos en el contenido del formulario JSF del cuerpo de la pagina.
Al igual que en el formulario para la inserción de datos, se da por echo que la clase generada para la tabla se encuentra instanciada en la clase *"connectionDDBB"*. Si se desea cambiar la ubicación de la misma basta con modificar la plantilla XSLT para poner el valor deseado.
En primer lugar se genera el botón para cargar los datos.

```
<xsl:variable name="valueButton">#{connectionDDBB.executeSelect()}</xsl:variable>
<!-- valueButton = método a ejecutar para realizar la consulta -->
<h:commandButton value="Load" action="{$valueButton}">
<!--  boton con texto load y accion asociada al método de cargar los datos -->
  <xsl:variable name="table">
    <xsl:value-of select="TABLE/NAME"/>
  </xsl:variable>
  <xsl:variable name="target">#{connectionDDBB.propertySelectQuery}</xsl:variable>
    <f:setPropertyActionListener target="{$target}" value="{$select}" />
    <!-- Cuando se pulse el botón, se pasara al atributo definido en la variable target (la cadena propertySelectQuery), el valor del parámetro select, que es la cadena generada al principio de la plantilla con la información de las claves importadas. Dicha información se usara al ejecutar el método select -->
</h:commandButton>
```

A continuación relacionaremos la tabla de datos con los valores devueltos por la consulta:

```
<xsl:variable name="valueTabla">
  #{connectionDDBB.functionSelect.rowsValues}
</xsl:variable>
<!-- La variable valueTabla contiene el parámetro de la clase functionSelect. El parámetro rowsValues
     es una lista enlazada de instancias de la clase rowValues -->
<h:dataTable border="1" bgcolor="#EEEEEE" var="item" value="{$valueTabla}"
             headerClass="heading">
<!--  Asociamos los datos devueltos con la tabla y la variable "item" -->
 (Contenido de la tabla)
</h:dataTable>
```

Dentro de la tabla generamos las columnas con los valores devueltos para los campos de la tabla. Se distingue el tipo de columna que es (normal, clave primaria o ajena) para aplicar un estilo diferente para cada caso:

```
<xsl:for-each select="TABLE/COLUMN">
<!-- Recorremos todas las columnas para saber que tipo de dato contendrá -->
  <h:column>
    <f:facet name="header"><xsl:value-of select="NAME"/></f:facet>
    <!-- La cabecera sera el nombre de la columna-->
    <xsl:variable name="value">
       #{item.returnData(<xsl:value-of select="position()-1"/>)}
    </xsl:variable>
    <!-- El valor devuelto se realizara  mediante el método returnData(i) -->
    <xsl:choose>
      <!-- El valor sera una clave primaria -->
      <xsl:when test="count(PK/node()) > 0">
        <h:outputLabel styleClass="pk" value="{$value}"/>
      </xsl:when>
      <!--  El valor sera una clave importada que no es clave primaria -->
      <xsl:when test="(count(FK/node()) > 0) and (count(PK/node()) = 0)">
        <h:outputLabel styleClass="fk" value="{$value}"/>
      </xsl:when>
      <xsl:otherwise>
      <!-- En otro caso no se aplica estilo -->
        <h:outputLabel  value="{$value}"/>
      </xsl:otherwise>
    </xsl:choose>
  </h:column>
</xsl:for-each>
```

Cabe explicar mejor el funcionamiento del método *"returnData(int i)"*.
La lista *"rowsValues"* es una lista de instancias de la clase *"rowValues"*. A su vez, la clase *rowValues* contiene una lista de cadenas, una para cada valor devuelto por columna de la tabla para una tupla.
Cuando mostramos el valor en la pagina xhtml usamos el método *"returnData(<xsl:value-of select="position()-1"/>)"* para cada columna.
Position() nos devuelve la posición de la etiqueta *COLUMN* dentro de su nivel en la jerarquía en el xml normalizado. Como las columnas has sido generadas en el xml del mismo orden en el que se has generado las columnas del xhtml, la columna 1 coincidirá con el valor de la posición 0 dentro

de la lista de la clase rowValues., la 2 con el valor de la posición 1, y así sucesivamente. De aquí el echo de devolver el valor de la posición – 1.

Esto se torna aun mas importante a la hora de crear las columnas con la información adicional para las columnas que son clave ajena. Es importante tener una ordenación del orden de las búsqueda en profundad para saber en que orden aparecerán los datos y poder así colocar los valores en la columna adecuada. Para esto existe para un elemento columna que sea clave importada el elemento *FKNUM*. Este valor nos ayudara a seleccionar el valor deseado para la columna al ejecutar el método *returnData(int i)*.

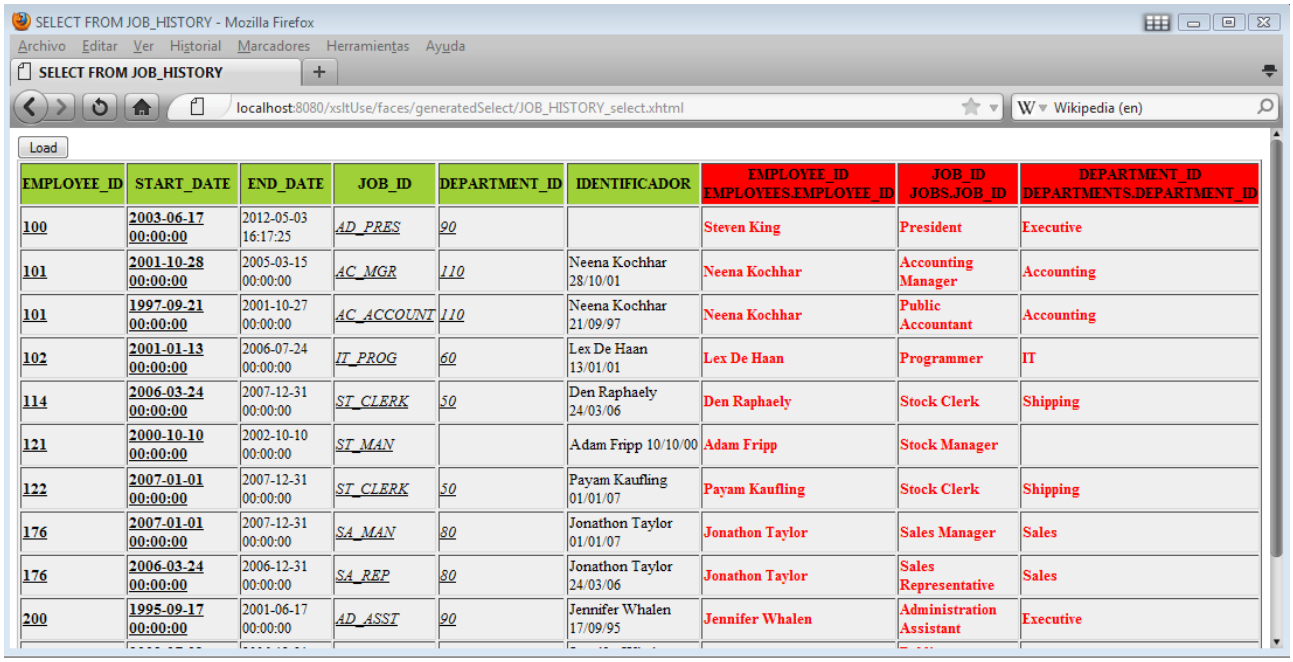
Después de añadir las columnas originales de la tabla, se añaden las que contiene la información adicional de las claves importadas.

```
<xsl:variable name="totCol" select="count(TABLE/COLUMN)"/>
<!-- variable totCol = numero de columnas -->
<xsl:for-each select="TABLE/COLUMN">
<!-- Para todas los columnas, si una es clave ajena, añadir una nueva columna -->
  <xsl:if test="count(FK/FKNUM/node()) > 0">
    <h:column headerClass="imported">
      <f:facet name="header">
        <xsl:value-of select="NAME"/>
        <br></br>
        <xsl:value-of select="FK/FINALREFERENCES"/>.
        <xsl:value-of select="FK/FINALCOLREFERENCES"/>
        <!-- Cabecera de la columna = nombre de la columna seguido de la tabla y columna de la
             que se obtiene la información de la clave importada  -->
      </f:facet>
      <xsl:variable name="fkpos" select="FK/FKNUM"/>
      <!-- variaable fkpos = valor de fknum -->
      <xsl:variable name="value">
         #{item.returnData(<xsl:value-of select="$totCol + $fkpos – 1"/>)}
      </xsl:variable>
      <!-- Valor devuelto = elemento totCol -1 + fkpos, es el que coincidirá con la cabecera de la
           columna -->
      <h:outputLabel  value="{$value}" styleClass="importedLabel"/>
    </h:column>
  </xsl:if>
</xsl:for-each>
```

4.4- La búsqueda en profundidad en la pagina de visualización de datos

Una de las funciones mas interesantes de la aplicación es el echo de poder usar la información obtenida de las claves importadas para obtener información adicional de las mismas a la hora de visualizar las mismas.

En un ejemplo de una captura de la pagina que muestra los valores de la tabla "*JOB_HISTORY*" de la base de datos usada como muestra durante el desarrollo del proyecto podemos ver esta información adicional:



EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID	IDENTIFICADOR	EMPLOYEE_ID EMPLOYEES.EMPLOYEE_ID	JOB_ID JOBS.JOB_ID	DEPARTMENT_ID DEPARTMENTS.DEPARTMENT_ID
100	2003-06-17 00:00:00	2012-05-03 16:17:25	AD_PRES	90		Steven King	President	Executive
101	2001-10-28 00:00:00	2005-03-15 00:00:00	AC_MGR	110	Neena Kochhar 28/10/01	Neena Kochhar	Accounting Manager	Accounting
101	1997-09-21 00:00:00	2001-10-27 00:00:00	AC_ACCOUNT	110	Neena Kochhar 21/09/97	Neena Kochhar	Public Accountant	Accounting
102	2001-01-13 00:00:00	2006-07-24 00:00:00	IT_PROG	60	Lex De Haan 13/01/01	Lex De Haan	Programmer	IT
114	2006-03-24 00:00:00	2007-12-31 00:00:00	ST_CLERK	50	Den Raphaely 24/03/06	Den Raphaely	Stock Clerk	Shipping
121	2000-10-10 00:00:00	2002-10-10 00:00:00	ST_MAN		Adam Fripp 10/10/00	Adam Fripp	Stock Manager	
122	2007-01-01 00:00:00	2007-12-31 00:00:00	ST_CLERK	50	Payam Kaufling 01/01/07	Payam Kaufling	Stock Clerk	Shipping
176	2007-01-01 00:00:00	2007-12-31 00:00:00	SA_MAN	80	Jonathon Taylor 01/01/07	Jonathon Taylor	Sales Manager	Sales
176	2006-03-24 00:00:00	2006-12-31 00:00:00	SA_REP	80	Jonathon Taylor 24/03/06	Jonathon Taylor	Sales Representative	Sales
200	1995-09-17 00:00:00	2001-06-17 00:00:00	AD_ASST	90	Jennifer Whalen 17/09/95	Jennifer Whalen	Administration Assistant	Executive

Los valores de las columnas con la cabecera en verde son los datos contenidos en la tabla. Existen 3 columnas que son claves importadas (*EMPLOYEE_ID, JOB_ID y DEPARTMENT_ID*).
Se observa que solo con esta información, al visualizar los datos de la tabla es difícil interpretar los datos si no se sabe relacionar los códigos de las claves con la tupla a la que referencian (Por ejemplo, ver que el trabajador 100 trabaja en el departamento 90 no aporta mucha información a priori).
Al generar la pagina web con la aplicación, y usándola junto a la clase auxiliar "*funcionSelect()*" podemos obtener un campo identificativo de la tupla a la que referencia la clave importada.
Estos campos claves son mostrados en las columnas de color rojo. Por cada clave importada hay una columna, dicha columna tiene en primer lugar el nombre de la columna que es clave ajena para poder relacionarlo con la misma, y a continuación la clave primaria origen del dato mostrado (nombre_de_la_columna.tabla).

En este ejemplo, fijándonos en la primera tupla mostrada, la información adicional de las claves importadas nos permite ver que el empleado con clave primaria 100 es Steven King, su identificador de trabajo "AD_PRES" referencia al puesto de trabajo de presidente y que pertenece al departamento 90, que es el departamento ejecutivo.

El proceso para obtener dicha información es complejo, y es detallado a continuación:

Al generar los xhtml para la visualización de datos, se usa la información de los xml de las claves importadas para formar una cadena codificada en una estructura con dicha información. Cuando se pulse el botón de cargar los datos de una tabla, dicha estructura se pasara al atributo *propertySelectQuery* definido en el *bean connectionDDBB* que controla la aplicación que usa los ficheros generados. Dicho atributo sera pasado como parámetro al constructor de la clase "*funcionSelect*", que se encarga de cargar los datos de la tabla. Dicha clase usara esta información para formar la consulta adecuada que añada la información de las claves importadas a la consulta final.

La cadena codificada con la información de las claves importadas tiene la siguiente estructura:

En primer lugar aparecerá el nombre de la tabla.
Después, separado por comas, los nombres de la columna que forman la clave primaria de dicha tabla.
A continuación, por cada clave importada se inserta el carácter '='. Dicho carácter representa el echo de que hay que realizar una búsqueda en la base de datos para obtener los valores identificativos para dicha clave.
Por cada carácter '=', aparecerán 1 o mas sucesiones de 4 cadenas, separadas por comas. Cada una de estas sucesiones de 4 cadenas empezaran con el carácter '|' y contendrán los datos necesarios para formar la consulta final. Dichos valores serán el nombre de la columna que forma parte de la clave importada, el nombre de la tabla a la que referencia, la columna a la que referencia y el nivel de profundidad para dichos datos.

A continuación se muestran dos de estas cadenas generadas y se detalla la estructura de la tabla a partir de la cual se han generado para el mayor entendimiento de las mismas:

Ejemplo 1:

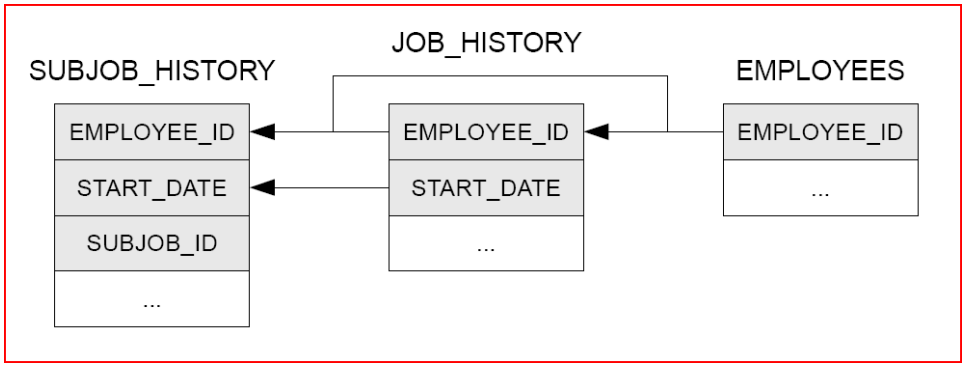
"DEPARTMENTS,DEPARTMENT_ID
=|MANAGER_ID,EMPLOYEES,EMPLOYEE_ID,1,
=|LOCATION_ID,LOCATIONS,LOCATION_ID,1,"

Esta cadena ha sido generada para la tabla DEPARTMENTS.
La clave primaria de la tabla es simple y es la columna DEPARTMENT_ID.
La tabla tiene dos claves importadas simples. Una es la columna MANAGER_ID, la cual referencia a la columna EMPLOYEE_ID de la tabla EMPLOYEES. La otra es la columna LOCATION_ID, la cual referencia a la columna LOCATION_ID de la tabla LOCATIONS. Ambas claves importadas tiene profundidad máxima 1.

Ejemplo 2:

"SUBJOB_HISTORY,EMPLOYEE_ID,START_DATE,SUBJOB_ID
=|EMPLOYEE_ID,JOB_HISTORY,EMPLOYEE_ID,1,
|EMPLOYEE_ID,EMPLOYEES,EMPLOYEE_ID,2,
|START_DATE,JOB_HISTORY,START_DATE,1,"

La estructura y relaciones de la tabla de la cual se obtiene esta cadena esta representada en el siguiente gráfico:



En este caso la cadena referencia a la tabla SUBJOB_HISTORY.
La clave primaria es compleja, ya que esta formada por 3 columnas: EMPLOYEE_ID, START_DATE y SUBJOB_ID.
Existe una clave importada en la tabla, la cual es doble y esta formada por las columnas EMPLOYEE_ID y START_DATE.
Las tres siguientes tuplas de 4 cadenas contienen la información necesaria para realizar la búsqueda de dicha clave ajena en la base de datos.
En primer lugar nos fijamos en las tuplas con nivel de profundidad 1, que son la primera y la tercera. Dichas tuplas hay que interpretarlas de manera conjunta, ya que forman la información completa necesaria para trazar el destino de la clave ajena. Obtenemos que la clave formada por las columnas (EMPLOYEE_ID, START_DATE) referencian a la clave primaria doble (EMPLOYEE_ID, START_DATE) de la table JOB_HISTORY.
Por ultimo vemos que la columna EMPLOYEE_ID de la tabla referencia a nivel de profundidad 2 a la columna EMPLOYEE_ID de la tabla EMPLOYEES.
En definitiva, deberemos construir dos subconsultas para obtener los datos de las claves importadas, una referenciando a la calve primaria doble de la tabla JOB_HISTORY y otra referenciando a la clave simple de la tabla EMPLOYEES.

Una vez formada la cadena, la clase *functionSelect.java* interpretara la misma para formar la consulta final:

Las estructuras de la clase que nos ayudaran a generar la consulta final son las siguientes:

```
rowsValues = new LinkedList<rowValues>();
```

Es una lista enlazada de instancias de la clase *rowValues*. Dicha lista contendrá los valores devueltos por la consulta final.

selectQuerys = new LinkedList<String>();

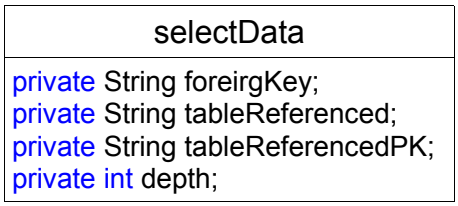
Es una lista enlazada que contiene cadenas con consultas que serán ejecutadas de manera independiente. Para generar la consulta final es posible que haya que crear una serie de vistas en la base de datos que contengan subconsultas para la consulta final. Tanto estas vistas como la consulta final serán almacenadas en esta estructura.

PKNames = new LinkedList<String>();

Lista con las columnas que forman la clave primaria de la tabla. Sera utilizada a la hora de crear las vistas con la información de las subconsultas.

listSelectData = new LinkedList<selectData>();

Esta lista almacena instancias de la clase *selectData*. Recordemos que esta clase tiene los siguientes atributos:



Estos 4 atributos coinciden con las 4 cadenas que siguen al carácter '|' en la cadena que se usa para generar la consulta. Por tanto, la lista listSelectData almacena estas tuplas de 4 cadenas usadas para la generación de las subconsultas.

Para generar la consulta final en el caso de tablas con claves importadas habrá que usar subconsultas que generaran vistas con clausulas “LEFT JOIN” debido a que si se utilizan selects con clausulas “AND” se perderá toda la información para las claves importadas para el caso de que una o mas de las claves importadas tenga valor null para una tupla (Si una de las columnas para una clausula “AND” tiene valor null nunca se encontrara una tupla que coincida y el valor devuelto sera siempre nulo aunque se encuentres coincidencias para las demás clausulas).

A continuación se muestran unos ejemplos usando datos de las tablas usadas durante el desarrollo del porque usar *LEFT JOINs* y no clausulas *AND*.

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
100	17/06/03	03/05/12	null	90

VISTA A = JOB_HISTORY AND EMPLOYEES

EMPLOYEE_ID	START_DATE	IDENTIFICADOR
100	17/06/03	Steven King

VISTA B = JOB_HISTORY AND JOBS

EMPLOYEE_ID	START_DATE	JOB_ID	IDENTIFICADOR

VISTA C = JOB_HISTORY AND DEPARTMENTS

EMPLOYEE_ID	START_DATE	DEPARTMENT_ID	IDENTIFICADOR
100	17/06/03	90	Executive

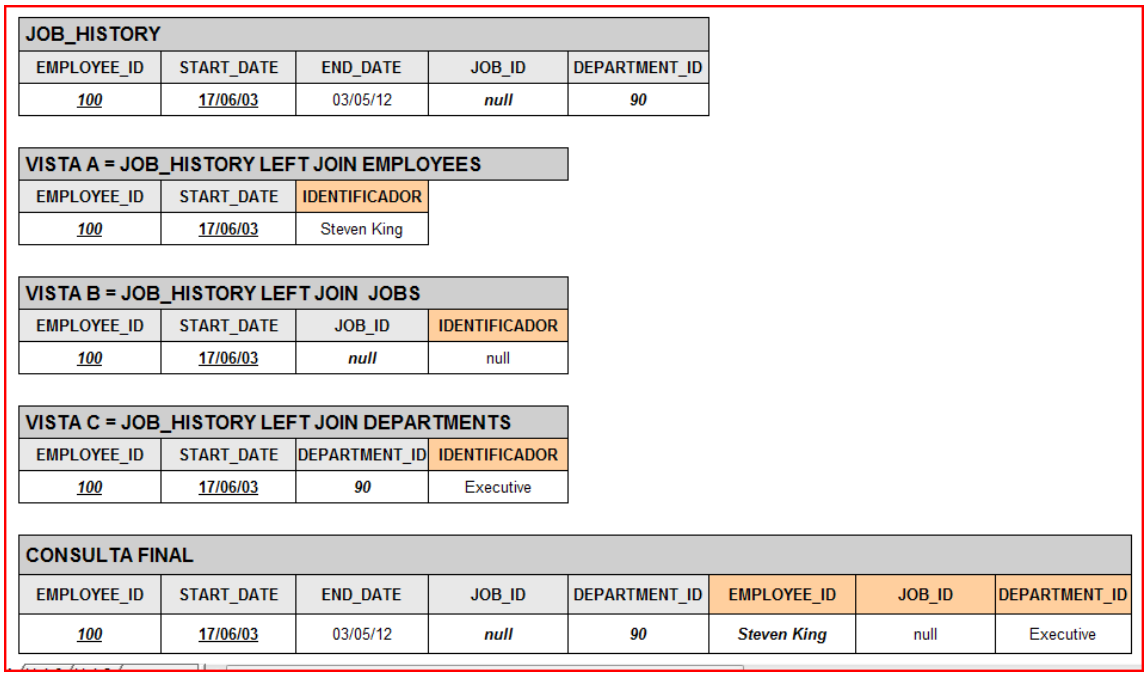
CONSULTA FINAL

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID

Resultado usando clausulas AND

La tupla original pertenece a la tabla JOB_HISTORY. Dicha tupla tiene 3 columnas que son clave importada: EMPLOYEE_ID, JOB_ID y DEPARTMENT_ID.
Al crear las 3 vistas (1 por clave importada) con las tablas a las que referencian las claves importadas, tenemos que la vista B que hace un select entre las tablas JOB_HISTORY y la tabla JOBS guardando las columnas que son clave primaria y usando la clausula *AND* entre el campo JOB_HISTORY.JOB_ID y JOBS.JOB_ID no devuelve ningún resultado ya que el valor de JOB_ID de la tupla original es nula.
Debido a esto la consulta final que hace un *AND* entre las tres columnas importadas y los campos IDENTIFICADOR correspondientes a las mismas no devuelve ninguna tupla ya que una de las clausulas *AND* no encuentra un valor.

Creando vistas usando la clausula *LEFT JOIN* el resultado de la consulta final sera el deseado ya que la vista B se genera correctamente debido que almacenara la clave primaria de la tupla original aunque el campo identificador de la tabla referenciada sea nulo:



JOB_HISTORY

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
100	17/06/03	03/05/12	null	90

VISTA A = JOB_HISTORY LEFT JOIN EMPLOYEES

EMPLOYEE_ID	START_DATE	IDENTIFICADOR
100	17/06/03	Steven King

VISTA B = JOB_HISTORY LEFT JOIN JOBS

EMPLOYEE_ID	START_DATE	JOB_ID	IDENTIFICADOR
100	17/06/03	null	null

VISTA C = JOB_HISTORY LEFT JOIN DEPARTMENTS

EMPLOYEE_ID	START_DATE	DEPARTMENT_ID	IDENTIFICADOR
100	17/06/03	90	Executive

CONSULTA FINAL

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
100	17/06/03	03/05/12	null	90	Steven King	null	Executive

Resultado usando clausulas LEFT JOIN

Los métodos de dicha clase son los siguientes:

restructureQuery(String query):

Este método se encarga de extraer los datos de la cadena con la información de la consulta y almacena los valores en las estructuras necesarias: Las claves primarias en la lista *PKNames* y los valores para las subconsultas en la lista *listSelectData*.
Una vez que estas estructuras estén rellenadas y por cada clave importada (carácter '=' en la cadena con la información de la consulta) se llamara al método *"generateQuery()"*.

GenerateQuery():

Este método es el encargado de crear la vista para la subconsulta.
En primer lugar se busca el mayor nivel de profundidad para las subconsultas que hay que realizar (mayor valor de *selectData.depth* de la lista *listSelectData*). Habrá que generar una vista por nivel de profundidad.

El código comentado que genera las vista es el siguiente:

```
for(int i = 0; i < maxLevel; i++){
// por cada nivel de profundidad una vista (para el caso de claves importadas compuestas)
   this.selectQuerys.add("DROP VIEW " + this.viewName);
   // Borrar la vista si es que existe para crear una con ese nombre
   String view = "";
   view = view.concat("CREATE VIEW " + this.viewName + " AS SELECT ");
   // Crear la vista con el nombre deseado
   for(int k = 0; k < this.PKNames.size(); k++){
      view = view.concat("alias1." + this.PKNames.get(k) + ",");
      // Por cada clave primaria, añadirla. La tabla sera renombrada con el alias "alias1" mas adelante
   }
   view = view.concat("alias2." + getMyProperties().getProperty("infoColumn") + " ");
   // Añadir la columna que esta definida como identificativa para la clave importada que sera parte de
la tabla con la que se realizara el left join y renombrada con el alias "alias2"
   view =
   view.concat("FROM " + this.tableName + " alias1 LEFT JOIN " + depthTableName[i] + " alias2");
   // Añadir la clausula FROM del LEFT JOIN entre la tabla origen y la referenciada y renombrarla con
los alias "alias1" y "alias2". La estructura depthTableName[i] contiene el nombre de la tabla
referenciada por cada nivel de profundidad de la clave.
   view = view.concat(" ON "); // Añadir clausula ON
   for(int k = 0; k < this.listSelectData.size(); k++){
   // Por cada elemento de la lista  listSelectData
      if((i+1) ==  this.listSelectData.get(k).getDepth()){
      // Si es el nivel de profundidad adecuado, añadir la clausula que compara la clave importada con
la clave primaria de la tabla referenciada para las columnas necesarias
         view = view.concat("alias1."+ this.listSelectData.get(k).getForeirgKey() +"=");
         view = view.concat("alias2."+ this.listSelectData.get(k).getTableReferencedPK() + " and ");
      }
   }
   view = view.substring(0, view.length()-5);
   // Elimino los ultimo 5 caracteres de la cadena (la ultima clausula 'and' que aquí sobra)
   this.selectQuerys.add(view);
   // añado a la lista la subconsulta generada
   this.viewName++;
   // incremento la variable viewName (la variable un un carácter que se inicializa con valor 'a')
}
this.listSelectData.clear();
// limpio la lista para que pueda volver a ser usada para otras claves ajenas
```

generateFinalSelectQuery():

Una vez que se han generado todas las listas que obtienen los datos de las claves importadas, hay que formar la consulta final que usara dichas vistas.
Para formar la consulta final hay que hacer un select entre la tabla original y las vistas creadas, mostrando solo los campos identificadores de dichas vistas y usando la clausula *AND* entre la clave primaria de la tabla original y las vistas.

El código comentado de este método es el siguiente:

```
char viewAux = 'a';
int numViews;
String finalSelect = "";
finalSelect = finalSelect.concat("SELECT " + this.tableName + ".*,");
// Seleccionar todos los campos de la tabla original
numViews = this.viewName – viewAux;
// numero de vistas = totales menos la primera
for(int i = 0; i < numViews; i++){
   finalSelect = finalSelect.concat(viewAux + "."+getMyProperties().getProperty("infoColumn")+",");
   viewAux++;
   // Por cada vista añadir la columna identificativa de las mismas
}
finalSelect = finalSelect.substring(0, finalSelect.length()-1);
// quito la ultima coma de la cadena con la consulta final para que este bien formada
viewAux = 'a';
finalSelect = finalSelect.concat(" from " + this.tableName + ",");
// clausula from de la tabla sobre la que se realiza la consulta
for(int i = 0; i < numViews; i++){
   finalSelect = finalSelect.concat(viewAux + ",");
   viewAux++;
   // añadir las vistas generadas a la clausula from
}
finalSelect = finalSelect.substring(0, finalSelect.length()-1);
finalSelect = finalSelect.concat(" where ");
// quitar la ultima coma y añadir la clausula where
viewAux = 'a';
for(int i = 0; i < numViews; i++){
   // Para cada vista
   for(int j = 0; j < this.PKNames.size(); j++){
   // por cada columna que forma parte de la clave primaria
      finalSelect = finalSelect.concat(this.tableName + "." + this.PKNames.get(j) + "="
                                       + viewAux + "." + this.PKNames.get(j) + " and ");
      }
      // Hacer un AND entre las claves primarias de la tabla original y las mismas columnas de las
vistas
   viewAux++;
}
finalSelect = finalSelect.substring(0, finalSelect.length()-5);
// Eliminar el ultimo and para que la consulta este bien formada
this.selectQuerys.add(finalSelect);
// Añadir la consulta final en la ultima posición de la lista con las subconsultas
```

executeComplexQuery():

Este método es el encargado de ejecutar las consultas.
Tenemos todas las subconsultas y la consulta final en la lista *selectQuerys*.
Primero se ejecutan todas las consultas menos la ultima sin tener en cuenta valores devueltos, ya que dichas subconsultas son vistas.
Finalmente, se ejecuta la ultima consulta y se almacenan los valores devueltos en la lista de instancias de la clase "*rowValues*".

executeSelect():

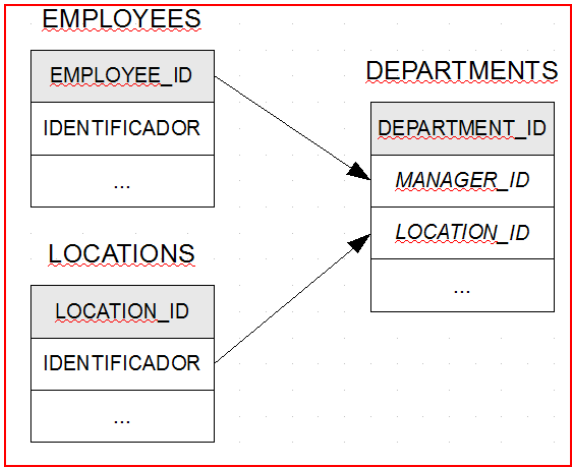
Puede suceder que la tabla de la cual se quieren consultar los datos no contenga ninguna clave ajena. En este caso se usara esta función para ejecutar el select debido a que dicha consulta es sencilla de implementar.

El procedimiento para generar la consulta final se implementa en el constructor de la clase y tiene la siguiente estructura:

```
restructureQuery(selectQuery); // Extraer los datos de la cadena y generar las vistas necesarias
if(this.simpleQuery == false){ // Si la tabla contiene claves importadas:
   generateFinalSelectQuery(); // Generar la consulta final
   executeComplexQuery(); // Ejecutar las vistas y la consulta final
} else { // Si la tabla no contenía clave importadas
   executeSelect(); // Ejecutar un select simple
}
```

En la siguiente pagina se mostraran dos ejemplos ilustrativos de el proceso anteriormente detallado:

El primer ejemplo es para una tabla que contiene dos claves importadas sencillas:



El proceso seguido para generar la consulta final sera el siguiente:

originalQuery (Generada mediante transformaciones XSLT) =
"DEPARTMENTS,DEPARTMENT_ID
=|MANAGER_ID,EMPLOYEES,EMPLOYEE_ID,1,
=|LOCATION_ID,LOCATIONS,LOCATION_ID,1,"

Método: RestructureQuery();
Estructura que almacena el resultado: listSelectData[0];
listSelectData[0].foreirgKey = MANAGER_ID
listSelectData[0].tableReferenced = EMPLOYEES
listSelectData[0].tableReferencedPK = EMPLOYEE_ID
listSelectData[0].depth = 1

Método: restructureQuery(); --> GenerateQuery();
Estructura que almacena el resultado: selectQuerys[0];
DROP VIEW a
CREATE VIEW a **AS SELECT** alias1.DEPARTMENT_ID,alias2.identificador **FROM** DEPARTMENTS alias1 **LEFT JOIN** EMPLOYEES alias2 **ON** alias1.MANAGER_ID=alias2.EMPLOYEE_ID

Método: restructureQuery();
Estructura que almacena el resultado: listSelectData[0];
listSelectData[0].foreirgKey = LOCATION_ID
listSelectData[0].tableReferenced = LOCATIONS
listSelectData[0].tableReferencedPK= LOCATION_ID
listSelectData[0].depth= 1

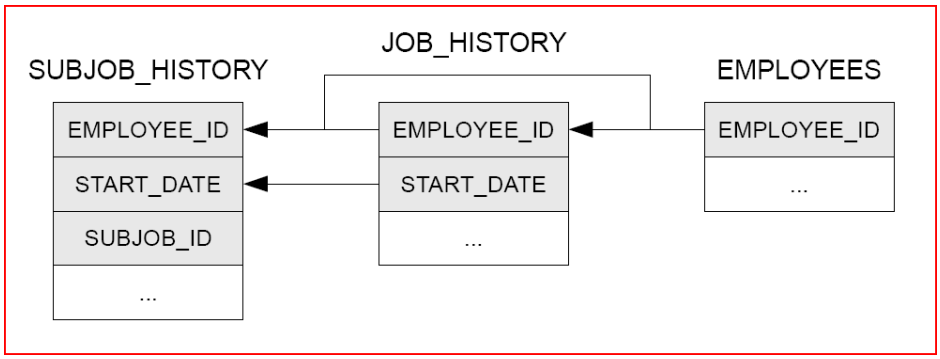
Método: restructureQuery(); --> generateQuery();
Estructura que almacena el resultado: selectQuerys[1]
DROP VIEW b
CREATE VIEW b **AS SELECT** alias1.DEPARTMENT_ID,alias2.identificador **FROM** DEPARTMENTS alias1 **LEFT JOIN** LOCATIONS alias2 **ON** alias1.LOCATION_ID=alias2.LOCATION_ID

Método: generateFinalSelectQuery();
Estructura que almacena el resultado: selectQuerys[2]
SELECT DEPARTMENTS.*,a.identificador,b.identificador from DEPARTMENTS,a,b **where** DEPARTMENTS.DEPARTMENT_ID=a.DEPARTMENT_ID **and** DEPARTMENTS.DEPARTMENT_ID=b.DEPARTMENT_ID

Método: executeComplexQuery();
Estructura usada: selectQuerys(); Se ejecutan todas las sentencias y se toman los valores devueltos de la ultima.

En el ejemplo 2, la tabla de la cual se quieren visualizar los datos es una tabla con una clave

primaria triple, y una clave ajena doble que a su vez contiene una columna con nivel de profundidad 2:



El proceso seguido para generar la consulta final sera el siguiente:

originalQuery(Generada con transformaciones XSLT) =
"SUBJOB_HISTORY,EMPLOYEE_ID,START_DATE,SUBJOB_ID
=|EMPLOYEE_ID,JOB_HISTORY,EMPLOYEE_ID,1,
|EMPLOYEE_ID,EMPLOYEES,EMPLOYEE_ID,2,
|START_DATE,JOB_HISTORY,START_DATE,1,"

Método: RestructureQuery();
Estructura que almacena el resultado: listSelectData[0];
listSelectData[0].foreirgKey = EMPLOYEE_ID
listSelectData[0].tableReferenced = JOB_HISTORY
listSelectData[0].tableReferencedPK = EMPLOYEE_ID
listSelectData[0].depth = 1

Método: RestructureQuery();
Estructura que almacena el resultado: listSelectData[1];
listSelectData[1].foreirgKey = EMPLOYEE_ID
listSelectData[1].tableReferenced = EMPLOYEES
listSelectData[1].tableReferencedPK = EMPLOYEE_ID
listSelectData[1].depth = 2

Método: RestructureQuery();
Estructura que almacena el resultado: listSelectData[2];
listSelectData[2].foreirgKey = START_DATE
listSelectData[2].tableReferenced = JOB_HISTORY
listSelectData[2].tableReferencedPK = START_DATE
listSelectData[2].depth = 1

Método: restructureQuery(); --> GenerateQuery();

Estructura que almacena el resultado: selectQuerys[0], Usa listSelectData[0] y listSelectData[2];
Nivel = 1
DROP VIEW a
CREATE VIEW a **AS SELECT**
alias1.EMPLOYEE_ID,alias1.START_DATE,alias1.SUBJOB_ID,alias2.identificador **FROM**
SUBJOB_HISTORY alias1 **LEFT JOIN** JOB_HISTORY alias2 **ON**
alias1.EMPLOYEE_ID=alias2.EMPLOYEE_ID and
alias1.START_DATE=alias2.START_DATE

Método: restructureQuery(); --> GenerateQuery();
Estructura que almacena el resultado: selectQuerys[1]; Usa listSelectData[1];
Nivel = 2;
DROP VIEW b
CREATE VIEW b **AS SELECT**
alias1.EMPLOYEE_ID,alias1.START_DATE,alias1.SUBJOB_ID,alias2.identificador **FROM**
SUBJOB_HISTORY alias1 **LEFT JOIN** EMPLOYEES alias2 **ON**
alias1.EMPLOYEE_ID=alias2.EMPLOYEE_ID

Método: generateFinalSelectQuery();
Estructura que almacena el resultado: selectQuerys[2]
SELECT SUBJOB_HISTORY.*,a.identificador,b.identificador **from**
SUBJOB_HISTORY,a,b **where** SUBJOB_HISTORY.EMPLOYEE_ID=a.EMPLOYEE_ID
and SUBJOB_HISTORY.START_DATE=a.START_DATE **and**
SUBJOB_HISTORY.SUBJOB_ID=a.SUBJOB_ID **and**
SUBJOB_HISTORY.EMPLOYEE_ID=b.EMPLOYEE_ID **and**
SUBJOB_HISTORY.START_DATE=b.START_DATE **and**
SUBJOB_HISTORY.SUBJOB_ID=b.SUBJOB_ID

Método: executeComplexQuery();
Estructura Usada: selectQuerys(); Se ejecutan todas las sentencias y se toman los valores devueltos de la ultima.

Como se aprecia en los ejemplos, las vistas generadas tienen la siguiente estructura para una clave importada y nivel:

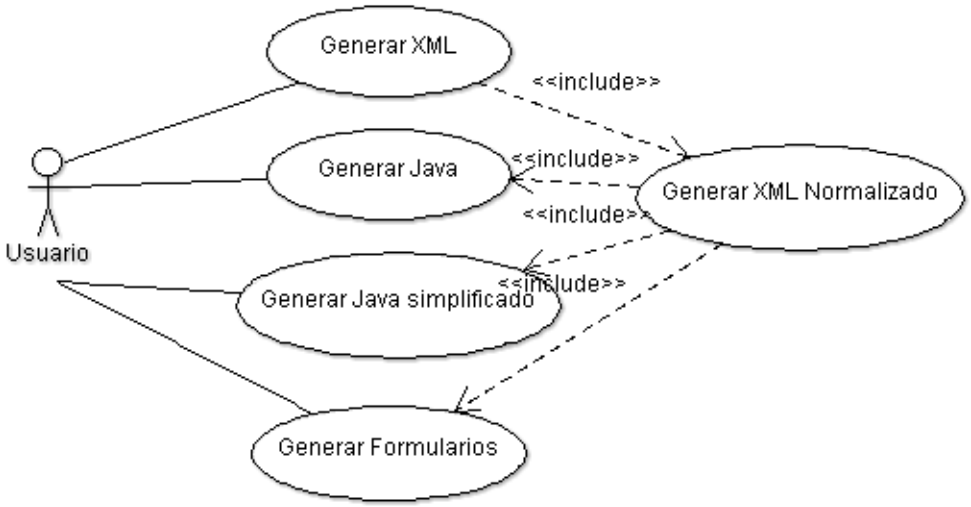
DROP VIEW [*nombre de la vista*]
CREATE VIEW [*nombre de la vista*] **AS SELECT**
  alias1.[*columnas que forman la clave primaria de la tabla original*],
  alias2.[*columna identificativa de la tabla referenciada*]
FROM
  [*tabla original*] alias1 **LEFT JOIN** [*tabla referenciada*] alias2
ON
  alias1.[*columnas que forman la clave importada*] =
  alias2.[*columnas que forman la clave primaria de la tabla referenciada*]

En el caso de la consulta final, esta tendrá la siguiente estructura:

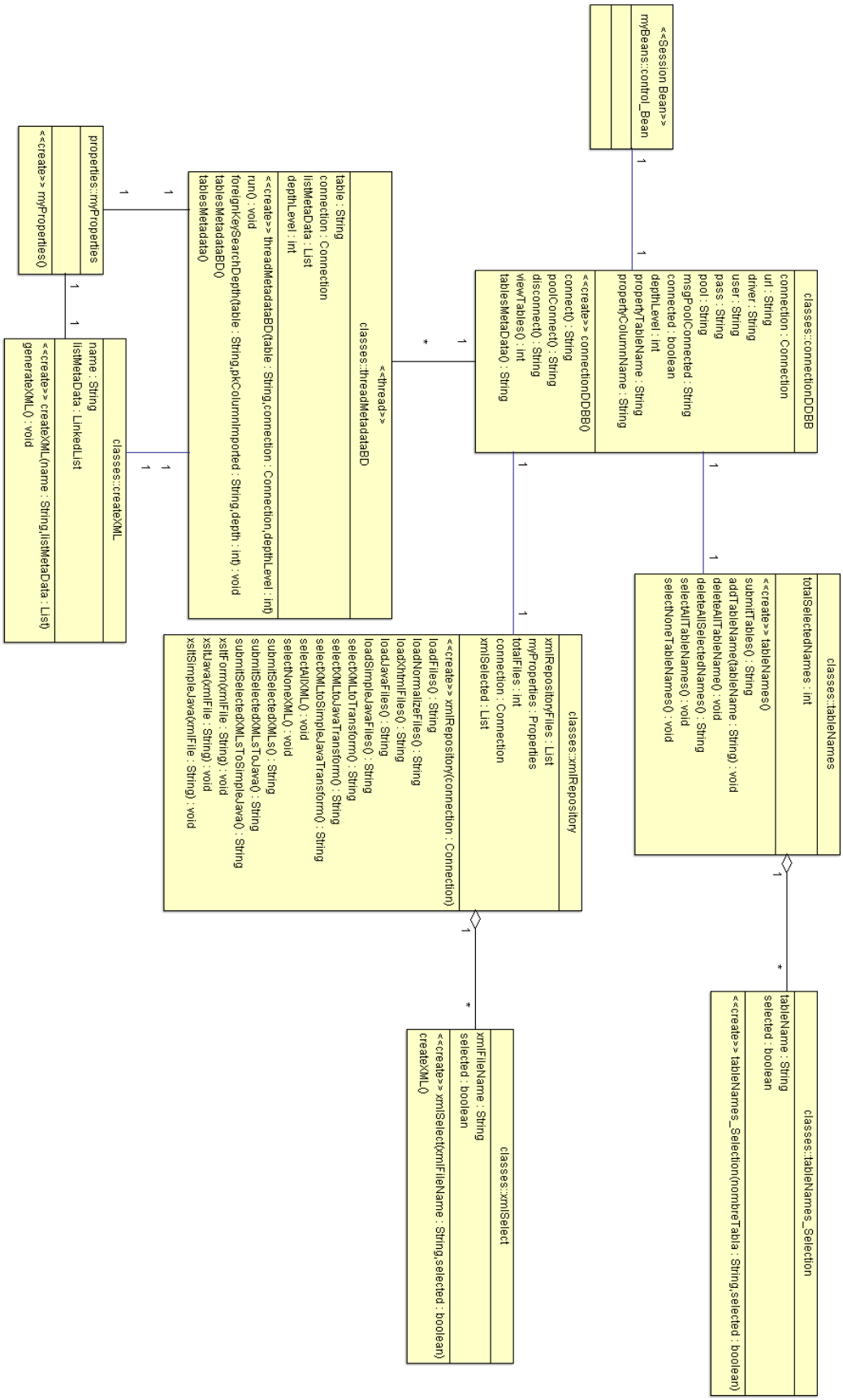
SELECT [*tabla original*].*, [*vistas generadas*].[*columna identificativa*] **FROM**
  [*tabla original*], [*vistas generadas*]
WHERE
  (Por cada vista generada)
  [*tabla original*].[*columnas que forman la clave primaria*] =
  [*vista generadas*].[*columnas que forman la clave primaria*]

5- Arquitectura del programa (Diagramas)

5.1- Diagrama de casos de uso

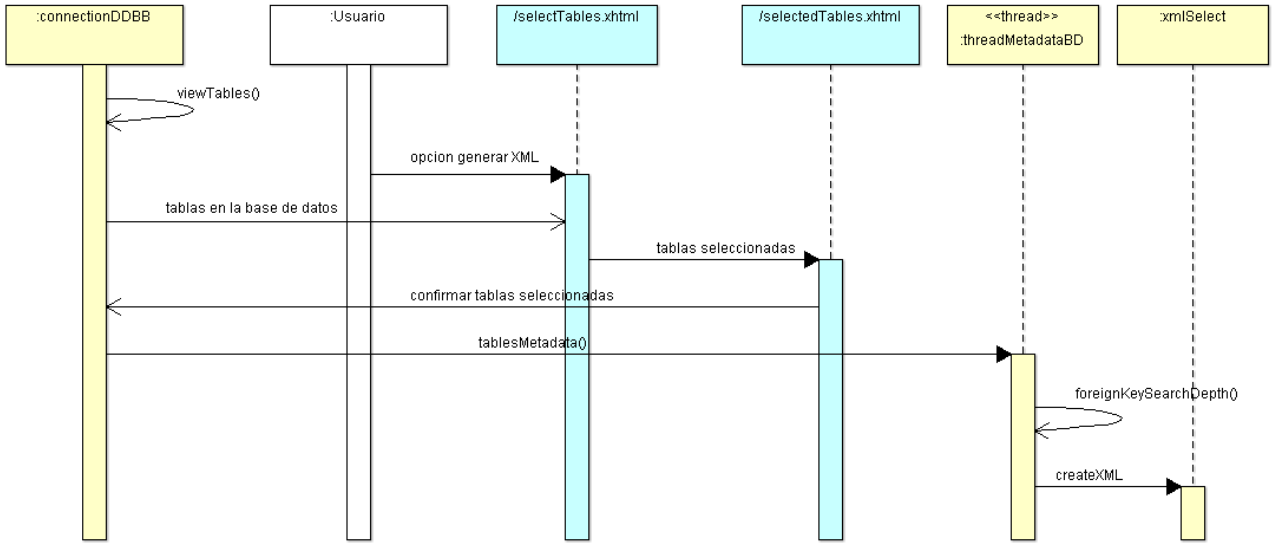


5.2- Diagrama de clases

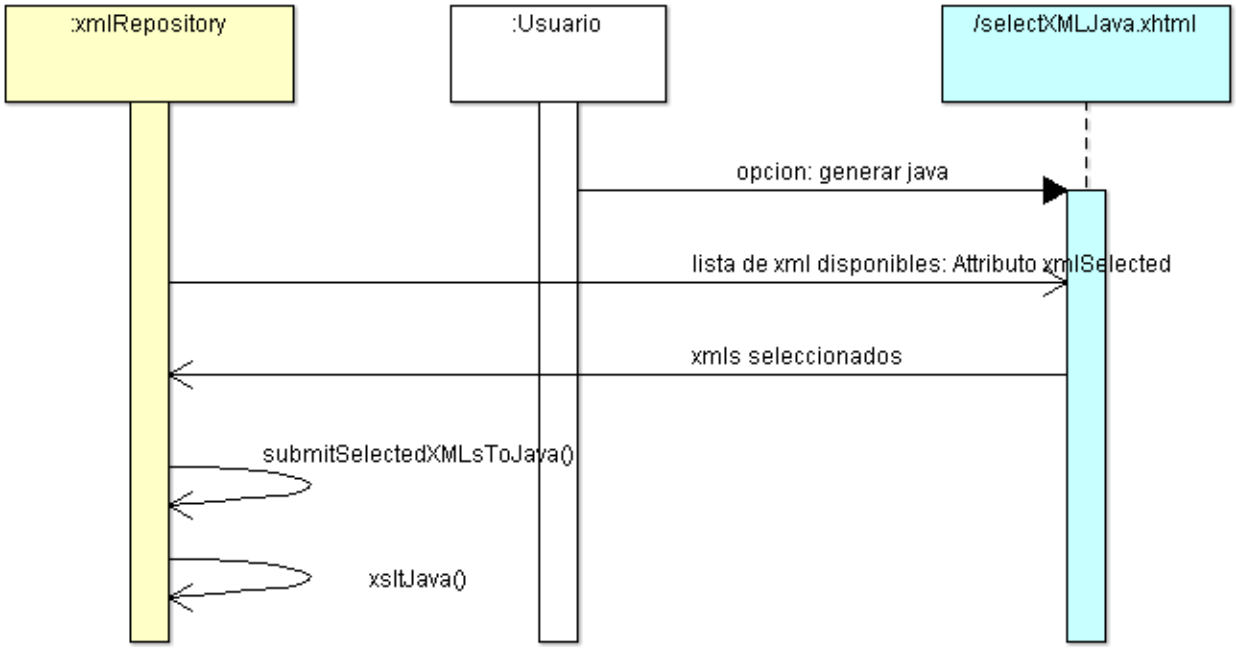


5.3- Diagramas de secuencia para la generación de archivos:

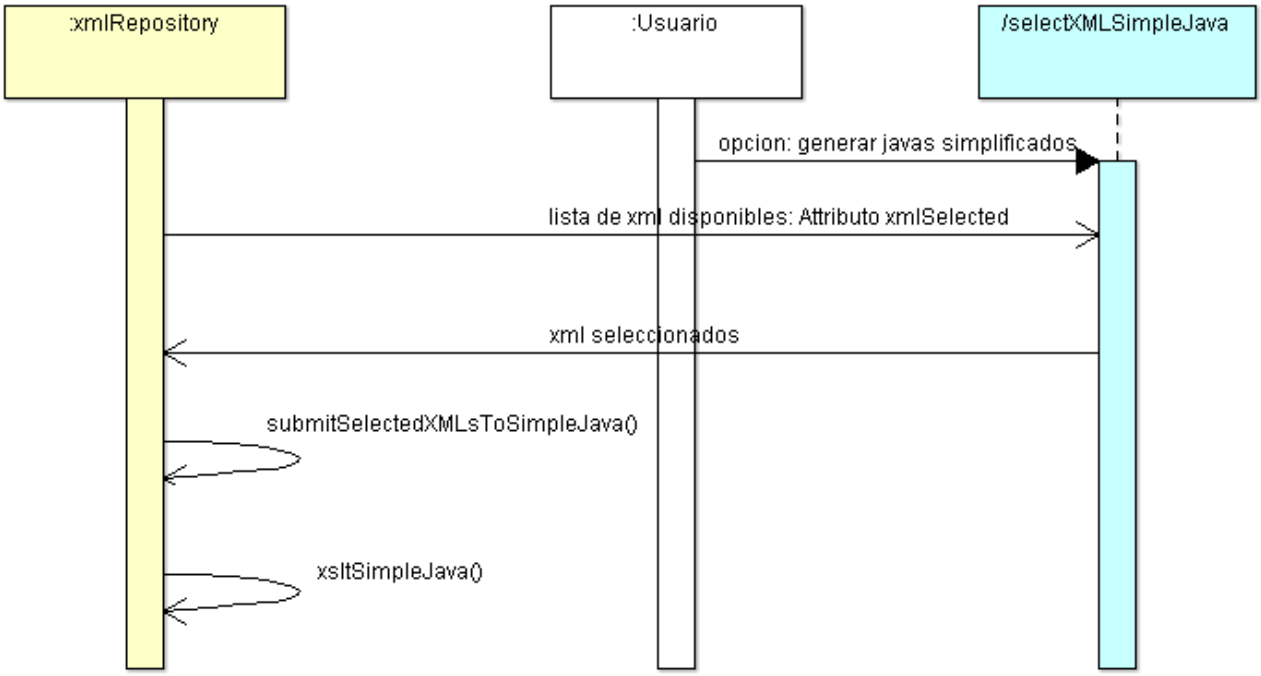
Generar los xml:



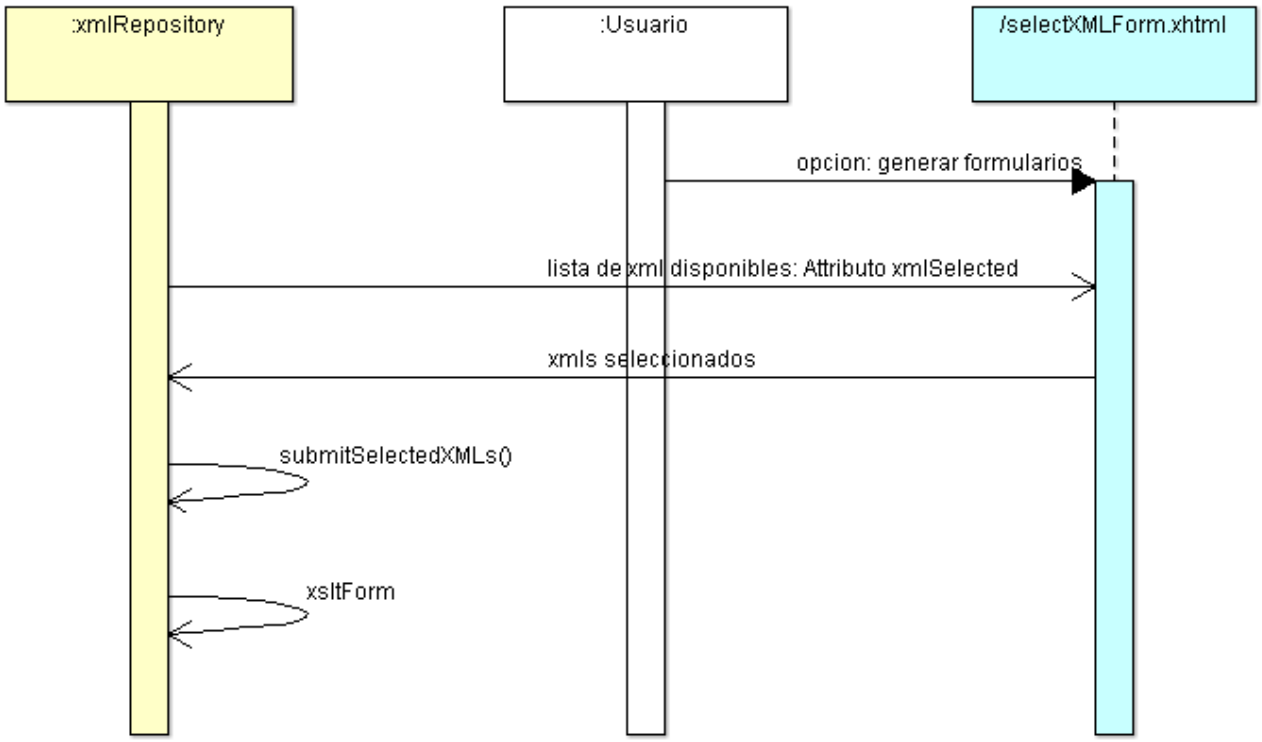
Generar las clases java:



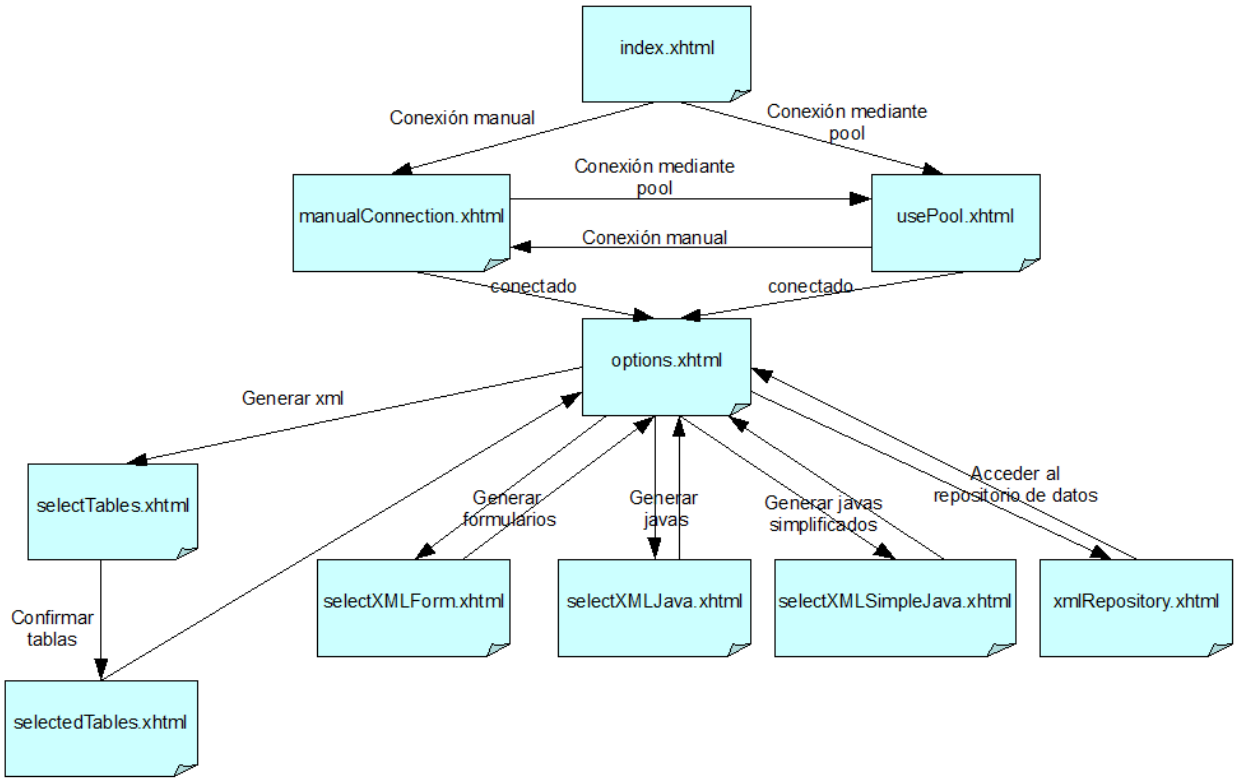
Generar las clases java simplificadas:



Generar los formularios:



5.4- Interfaz: Navegación



La aplicación xsltUse:

Un ejemplo de uso de los ficheros generados por la aplicación xsltGenerator

MANUAL DE USUARIO

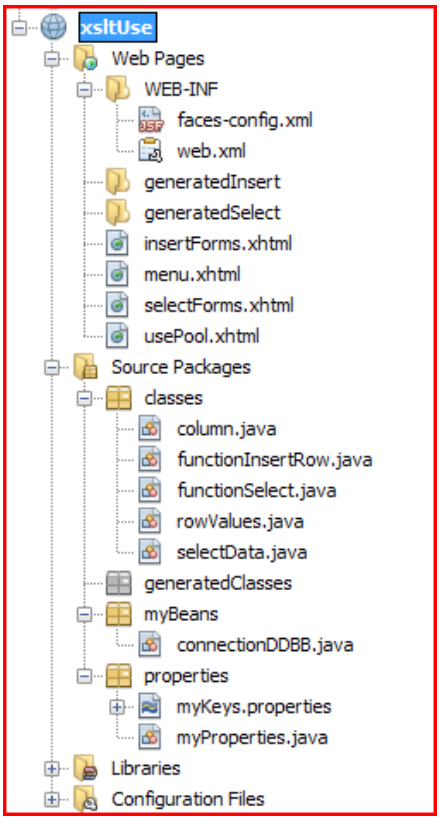
1- Introducción

Una vez que hemos generados los ficheros java y las paginas web para la inserción y visualización de los datos, llega el momento de hacer uso de los mismos.
La aplicación web *xsltUse* esta pensada para incluir los ficheros generados por la aplicación *xsltGenerator* y así poder interactuar con las tablas de la bases de datos de las que se generaron dichos ficheros.
Una vez que la aplicación haya incluido en su código estos ficheros sera capaz de insertar, modificar y consultar datos de una tabla.
Al igual que la aplicación *xsltGenerator*, la aplicación *xsltUse* es una aplicación web que usa el framework JSF2.0.

Para el correcto funcionamiento de la aplicación habrá que modificar alguna de las clases de la misma para incluir los ficheros java generados en la aplicación. Durante las siguientes secciones se partirá del esqueleto genérico del programa y se explicara el proceso a seguir para que la aplicación trabaje con unos ficheros generados para una base de datos de ejemplo. El proceso para otras tablas de una base de datos distinta seria el análogo al aquí detallado.

2- Esqueleto del programa

En el siguiente gráfico se muestra el contenido de la aplicación web *xsltUse* antes de que se le añadan los ficheros generados por la aplicaciones *xsltGenerator*:



La *carpeta "Web Pages"* contiene las paginas web que sirven como interfaz para la aplicación web. El contenido de esta carpeta es el siguiente:

Carpeta WEB_INF: Contiene ficheros de configuración de la aplicación.

Carpeta generatedInsert: En esta carpeta es donde se colocaran los formularios de inserción de datos generados.

Carpeta generatedSelect: En esta carpeta es donde se colocaran los formularios de visualización de datos generados.

usePool.xhtml: Pagina inicial de la aplicación. En ella se introduce el nombre del pool de conexiones a usar para conectarse a la base de datos

menu.xhtml: Menu principal de la aplicación, permite acceder a los formularios generados que se han añadido a la aplicación y desconectarse de la base de datos.

insertForms.xhtml: Pagina que lista los formularios de inserción añadidos a la aplicación.

selectForms.xhtml: Pagina que lista los formularios de visualización añadidos a la aplicación.

La *carpeta Source Packages* contiene el código de la aplicación en distintos paquetes:

Paquete classes:

Contiene las clases genéricas necesarias para el funcionamiento de la aplicación. Estas clases siempre se usaran por la aplicación *xsltUse* independientemente de los archivos generados por la aplicación *xsltGenerator*.
Estas son las clases auxiliares que contenía la aplicación *xsltGenerator* y que has sido comentadas y analizadas previamente en este documento.

Column.java: Información con los meta datos de una columna de una tabla de la base de datos.

funcionInsertRow.java: Clase encargada de insertar una tupla en la base de datos.

funcionSelect.java: Se encarga de procesar la cadena con la información para formar la consulta de los datos de la tabla proveniente del formulario de visualización de datos para luego generar y ejecutar la consulta.

rowValues.java: Estructura en la que se almacenan los valores devueltos por las consultas para mostrarlos en el formulario de visualización de datos.

SelectData.java: Es usada por la clase "*funcionSelect.java*" para generar las consultas.

Paquete generatedClasses:

En este paquete es donde se colocaran las clases java generadas por la aplicación *xsltGenerator*.

Paquete myBeans:

Contiene el *bean* de sesión connectionDDBB.java. Este *bean* contiene los métodos para la conexión con la base de datos y es donde se incluirán las clases generadas como parámetros y donde serán instanciadas. También es el encargado de invocar a las funciones del paquete clases.

Paquete properties:

Contiene el fichero myKeys.properties que incluye las claves usadas por la aplicación y la clase myProperties.java para el acceso a las mismas.

3- Conexión BBDD

La conexión con la base de datos se realizara mediante un pool de conexiones. La configuración y el uso del pool de conexiones ya ha sido detallado anteriormente en la sección de la aplicación *xsltGenerator*.
Como la aplicación se conectara a la base de datos de la que se han generado los ficheros se puede usar el mismo para esta aplicación.

4- Añadir archivos generados

Si no se añaden los ficheros generados, lo único que es capaz de hacer la aplicación es conectarse a la base de datos y desconectarse, así que si queremos obtener la funcionalidad deseada, debemos seguir el siguiente proceso para añadir los ficheros generados.

4.1- Añadir los ficheros xhtml

Añadir los ficheros xhtml es sencillo y no requiere ninguna modificación del código.
Basta con añadir los formularios de inserción de datos (los ficheros finalizados por _insert.xhtml) en la carpeta *generatedInsert* y los formularios de visualización de datos (los ficheros finalizados por _select.xhtml) en la carpeta *generatedSelect*.
La aplicación sera capaz de acceder a estos formularios cargándolos desde la carpeta en la que se encuentran.
Para que esto sea posible, habrá que asignar el valor correcto a las calves del fichero *myKeys.properties* "selectFormDirectory" y "insertFormDirectory".
Dichas claves deberán contener la ruta completa donde se ubica la carpeta del proyecto *generatedSelect* y *generatedInsert* respectivamente.

Por ejemplo, para la aplicación *xsltUse* completa usada de prueba las claves tiene el siguiente valor:

```
selectFormDirectory =
C:/Users/propietario/Documents/NetBeansProjects/xsltUse/web/generatedSelect/
insertFormDirectory =
C:/Users/propietario/Documents/NetBeansProjects/xsltUse/web/generatedInsert/
```

4.2- Añadir los ficheros java

Ahora es el momento de añadir las clases java generadas.
Para que los formularios generados a partir de las tablas puedan comunicarse con la base de datos hay que incluir la clase java generada para la tabla de la cual se generaron los formularios.
Las clases java a añadir serán las que contiene la información de los meta datos de las columnas (<NOMBRE_TABLA>.java).

En primer lugar hay que añadir los ficheros *.java generados al paquete *generatedClasses*

Para que las clases sean accesibles por los formularios habrá que añadir código al *bean* "*connectionDDBB.java*"

Esta clase tiene la siguiente estructura:

ConnectionDDBB <<session bean>>
Connection connection; Statement stmt; String pool; String msgPoolConnected; boolean connected; DataSource dataSource; String validationMsg; String DDBBMsg; Properties myProperties; List<String> listSelectForms; List<String> listInsertForms; String propertySelectQuery; functionSelect functionSelect;
connectionDDBB() <<constructor>> String **poolConnect()** void **instanceGeneratedJavas()** void **loadGenerateForms()** void **executeSelect()** String **disconnect()** void **poolNotFound()** void **poolFound()**

Los 6 primeros atributos (*connection*, *stmt*, *pool*, *msgPoolConnected*, *connected*, *dataSource*) son usados por la conexión con la base de datos.
Los 2 siguientes (*validationMsg*, *DDBBMsg*) son cadenas con mensajes informativos que se mostraran en las paginas web que forman en interfaz de la aplicación. El primero para el resultado de la validación de los datos de las tuplas y la siguiente para mensajes generados por la base de datos.
Los atributos *myProperties* y *funcionSelect* son instancias a las clases correspondientes.
La cadena *propertySelectQuery* es el atributo en el que se almacenara la cadena con la información para la generación de la consulta de los datos proveniente de los formularios de visualización de datos.
Por ultimo, las dos listas de cadenas *listSelectForms* y *listInsertForms* contendrán los nombres de los formularios de selección y inserción añadidos a la aplicación respectivamente, para luego poder mostrarlos en el interfaz y acceder a ellos.

En cuanto a los métodos, existen 4 usados para la conexión mediante el pool a la base de datos (*poolConnect()*, *disconnect()*, *poolNotFound()*, *poolFound()*).
El método *loadGenerateForms()* es el encargado de cargar en las listas *listSelectForms* y *listInsertForms* los nombres de los formularios agregados a la aplicación.
El método *executeSelect()* es el que se invoca por los formularios de visualización de datos y que llama a la clase *funcionSelect()* pasandole el objeto connection con la información de la conexión con la base de datos y la cadena con la información para generar la consulta contra la tabla.

El código del método es el siguiente:

```
public void executeSelect(){

        this.functionSelect = new functionSelect(this.connection, this.propertySelectQuery);
}
```

Por ultimo, esta el método *instanceGeneratedJavas()*.
Inicialmente este método se encontrara vació, ya que aquí es donde se instancian las clases que has sido generadas por la aplicaciones *xsltGenerator* y que han sido añadidas a la aplicación.

Vamos a definir la aplicación de ejemplo funcional que se implemento durante las pruebas de la aplicación para ver el proceso a seguir en este punto.

Se uso la base de datos de prueba que venia definida en la base de datos Oracle usada y se generaron con la aplicación *xsltGenerator* los formularios y javas correspondientes a 8 tablas:

COUNTRIES, DEPARTMENTS, EMPLOYEES, JOBS, JOB_HISTORY, LOCATIONS, REGIONS y SUBJOB_HISTORY.

Una vez añadidos los formularios y las clases en las carpetas y paquetes correspondientes, es hora de adaptar la clase *connectionDDBB.java* para usar dichos ficheros java.

En primer lugar, hay que importar el paquete en el que se encuentran las clases java generadas:

```
import generatedClasses.*;
```

Después, al final de la definición de los atributos de la clase, hay que añadir los atributos correspondientes a instancias de las clases generadas. El nombre del atributo ha de coincidir con el nombre de la tabla y el nombre de los ficheros java generados (que sera el mismo) y teniendo en cuenta que hay diferencia entre mayúsculas y minúsculas.
En nuestro caso hay que añadir los siguientes atributos:

```
  private COUNTRIES COUNTRIES;
  private DEPARTMENTS DEPARTMENTS;
  private EMPLOYEES EMPLOYEES;
  private JOBS JOBS;
  private JOB_HISTORY JOB_HISTORY;
  private LOCATIONS LOCATIONS;
  private REGIONS REGIONS;
  private SUBJOB_HISTORY SUBJOB_HISTORY;
```

Una vez que se han definido los atributos, es obligatorio generar todos los setter y getter para cada uno, ya que si no los formularios no podrán acceder a dichos objetos.

Por ultimo, habrá que instanciar las atributos correspondientes a las clases generadas dentro del método *instanceGeneratedJavas()*.
Al inicializar las clases generadas habrá que pasarle como parámetro el atributo connection que ya ha sido inicializado por la aplicación.

Para nuestro ejemplo el método quedaría así:

```
public void instanceGeneratedJavas(){

	/* inicialize classes
	* variable = new class(this.connection)
	*/

	this.COUNTRIES = new COUNTRIES(this.connection);
	this.DEPARTMENTS = new DEPARTMENTS(this.connection);
	this.EMPLOYEES = new EMPLOYEES(this.connection);
	this.JOBS = new JOBS(this.connection);
	this.JOB_HISTORY = new JOB_HISTORY(this.connection);
	this.LOCATIONS = new LOCATIONS(this.connection);
	this.REGIONS = new REGIONS(this.connection);
	this.SUBJOB_HISTORY = new SUBJOB_HISTORY(this.connection);
}
```

Con esto ya esta añadido todo el código necesario para el correcto funcionamiento de la aplicación usando los ficheros generados por la aplicación *xsltGenerator*.

5- Añadir información referencial

Para poder importar un campo identificativo de una correspondiente a una clave importada, dicho campo habrá de existir en las tablas de la base de datos. El siguiente apartado muestra el proceso que se realizo para insertar en las tablas de la base de datos las columnas con el valore identificativo de la tupla.

5.1- Campo identificador y preparación de la base de datos

En nuestro caso se escogió el nombre de "*identificador*" para la columna que contenga la cadena con la información identificativa de la tupla.
Las tablas de la base de datos usada en el ejemplo finalizado de la aplicación *xsltUse* no contienen dicha columna, así que habrá que añadirla y rellenar su valor.
Se decidió crear la columna 'identificador' del tipo VARCHAR2 y con tamaño para 50 caracteres.
Se definió la clave *infoColumn* con el valor "identificador" en el archivo *myKeys.properties* de la aplicación *xsltUse* para que la aplicación sepa buscar dicha columna al cargar los campos identificativos de las claves importadas.

Para preparar las tablas de la base de datos en necesario tener conocimiento de las tablas que nos interesan para generar los ficheros y de los datos que contienen.

En el caso de ejemplo interesaba una base de datos que contenía 8 tablas. Se creo un script SQL que añadiese el campo identificador a las tablas de la base de datos:

update identificador.sql:

```
alter table countries add identificador varchar2(50);
alter table departments add identificador varchar2(50);
alter table employees add identificador varchar2(50);
alter table job_history add identificador varchar2(50);
alter table jobs add identificador varchar2(50);
alter table locations add identificador varchar2(50);
alter table regions add identificador varchar2(50);
alter table subjob_history add identificador varchar2(50);
```

Ahora que ya hemos creado la columna 'identificador' en las tablas deseadas, hay que actualizarlas para asignar un valor identificativo para las tuplas de dichas tablas.

Examinando la base de datos, se añadir las siguientes sentencias al script SQL para actualizar el campo identificativo:

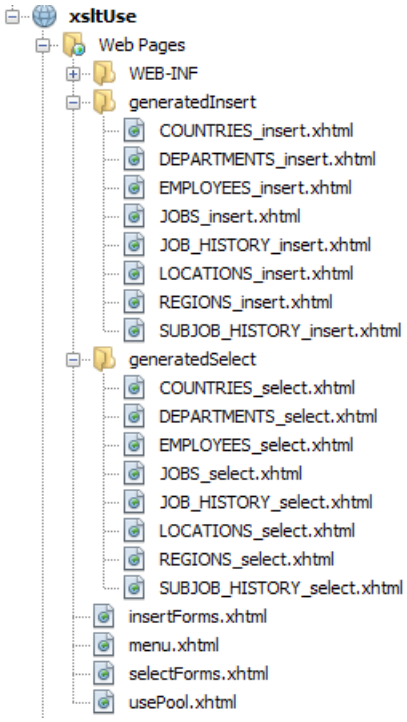
update identificador.sql (continuacion):

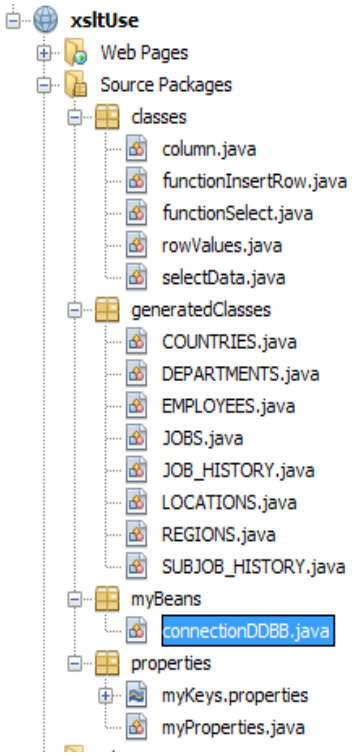
```
update countries set identificador=country_name;
-- Para la tabla paises, columna identificador = nombre del pais.

update departments set identificador=department_name;
-- Para la tabla departamentos, columna identificador = nombre del departamento.

update employees set identificador=first_name || ' ' || last_name;
-- Para la tabla empleados, columna identificador = nombre del empleado + primer apellido.

update job_history set identificador= (select first_name ||' '|| last_name from employees where
employee_id = job_history.employee_id) || ' '|| start_date;
-- Para la tabla historial de trabajos, columna identificador = nombre y apellido del trabajador + la
fecha en la que empezó el trabajo.

update jobs set identificador=job_title;
-- Para la tabla trabajos, columna identificador = nombre del trabajo.

update locations set identificador=city;
-- Para la tabla localización, columna identificador = nombre de la ciudad.

update regions set identificador=region_name;
-- Para la tabla regiones, columna identificador = nombre de la región.

update subjob_history set identificador=(select first_name ||' '|| last_name from employees where
employee_id = subjob_history.employee_id) ||' '|| start_date || ' ' || subjob_id;
-- Para la tabla historia de sub trabajos, columna identificador = nombre y apellidos de trabajador +
la fecha en que que empezo el trabajo principal y el codigo identificativo de la subtarea.
```

Una vez que este definido el script SQL, hay que ejecutarlo en la base de datos elegida y automáticamente creara las columnas en las tablas y las actualizara para insertar los valores deseados.

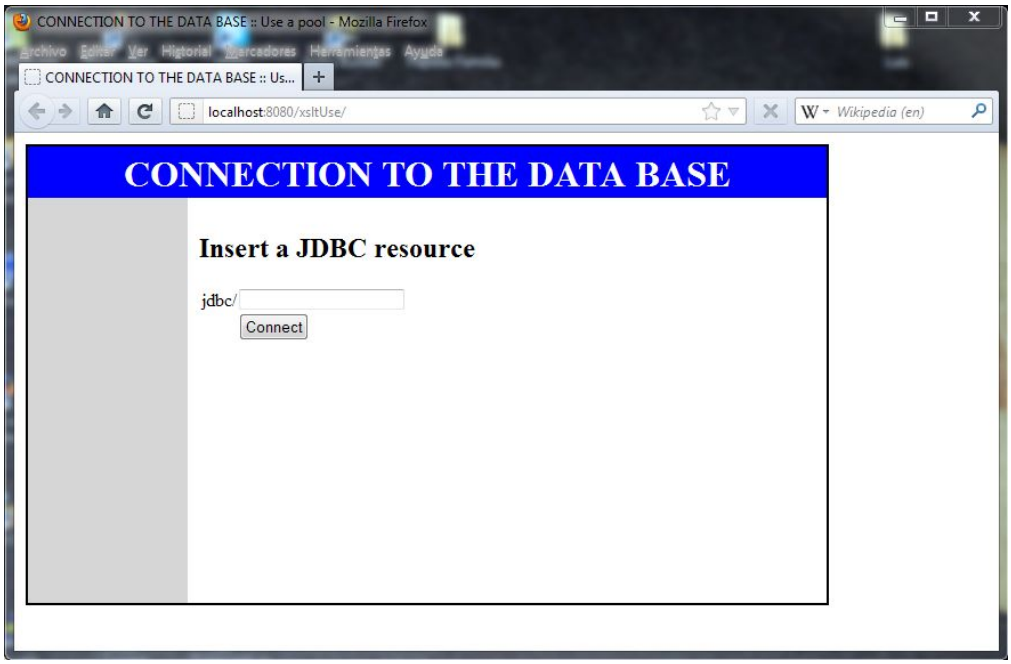
6- Interfaz y navegación del programa

Una vez añadidos los ficheros generados la carpeta W*eb Pages y Source Packages* quedarían de la siguiente manera:

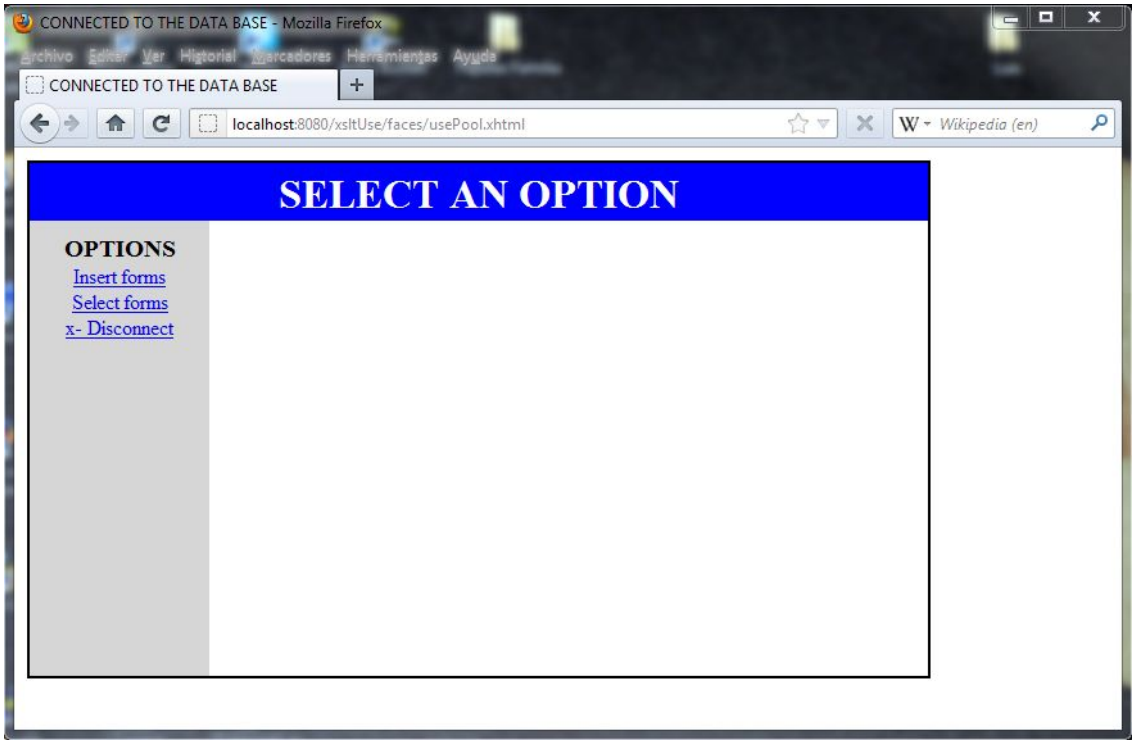




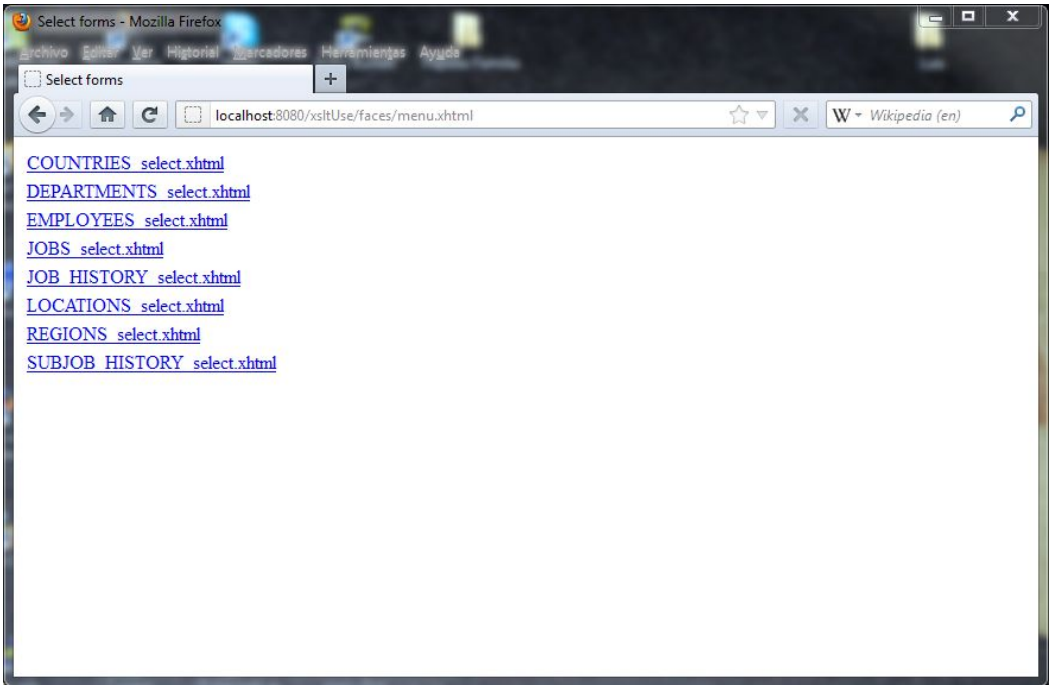
Una vez lanzada la aplicación, encontramos la pagina en la que introducir el nombre del pool de conexiones para conectarse a la base de datos:



Introducimos el nombre del recurso y nos conectamos a la base de datos:



Pulsando la opción *Select forms* accedemos a los formularios de visualización de datos añadidos a la aplicación:



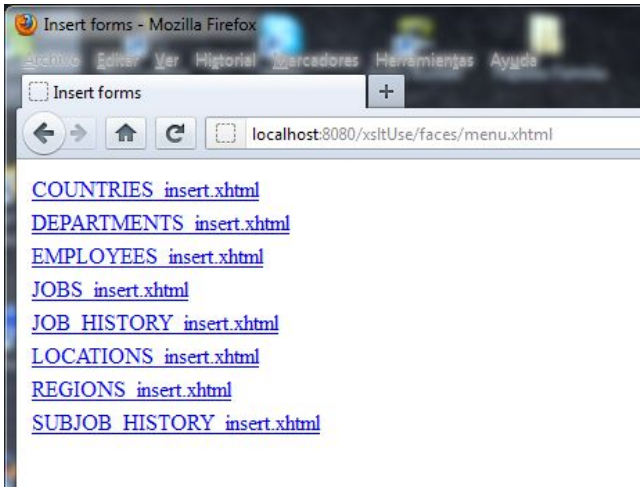
Vamos a abrir el formulario *DEPARTMENTS_select.xhtml.*

Pulsando el botón *load* contenido en la parte superior de la pagina, cargaremos los datos de la tabla DEPARTMENTS:

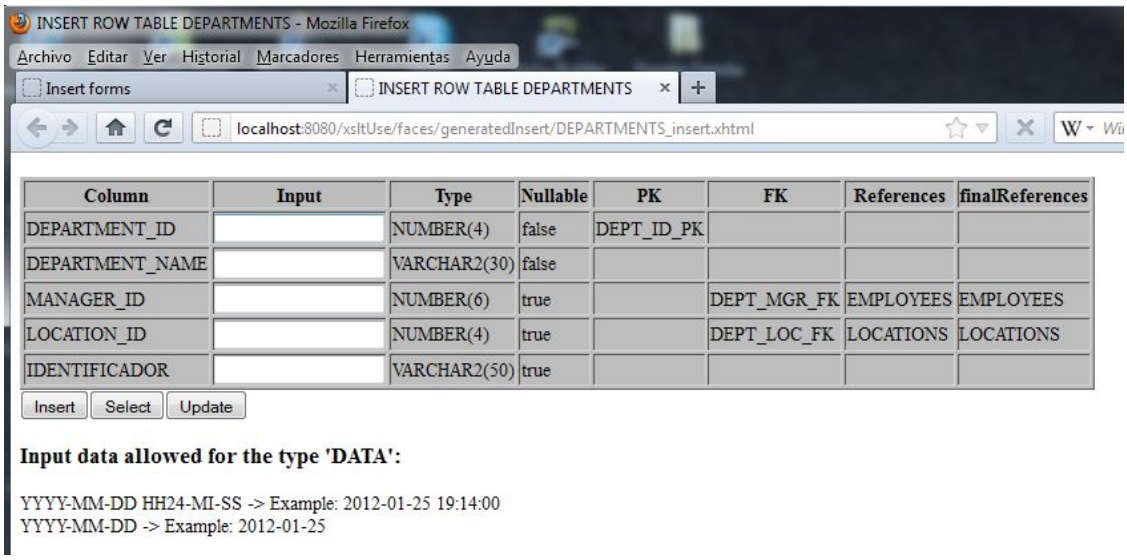
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID	IDENTIFICADOR	MANAGER_ID EMPLOYEES.EMPLOYEE_ID	LOCATION_ID LOCATIONS.LOCATION_ID
90	Executive	100	1700	Executive	Steven King	Seattle
60	IT	103	1400	IT	Alexander Hunold	Southlake
100	Finance	108	1700	Finance	Nancy Greenberg	Seattle
30	Purchasing	114	1700	Purchasing	Den Raphaely	Seattle
50	Shipping	121	1500	Shipping	Adam Fripp	South San Francisco
80	Sales	145	2500	Sales	John Russell	Oxford
10	Administration	200	1700	Administration	Jennifer Whalen	Seattle
20	Marketing	201	1800	Marketing	Michael Hartstein	Toronto
40	Human Resources	203	2400	Human Resources	Susan Mavris	London
70	Public Relations	204	2700	Public Relations	Hermann Baer	Munich
110	Accounting	205	1700	Accounting	Shelley Higgins	Seattle
120	Treasury		1700	Treasury		Seattle
130	Corporate Tax		1700	Corporate Tax		Seattle
140	Control And Credit		1700	Control And Credit		Seattle
150	Shareholder Services		1700	Shareholder Services		Seattle
160	Benefits		1700	Benefits		Seattle
170	Manufacturing		1700	Manufacturing		Seattle
180	Construction		1700	Construction		Seattle
190	Contracting		1700	Contracting		Seattle
200	Operations		1700	Operations		Seattle
210	IT Support		1700	IT Support		Seattle
220	NOC		1700	NOC		Seattle

En subrayado y negrita vemos los campos que son primary key, y en subrayado y cursiva los campos que son foreign key. Las columnas en rojo contienen la información importada para las columnas foreign key (El campo identificador de la tupla referenciada).
Podemos observar que para la primara tupla, la columna MANAGER_ID (El empleado que esta al cargo del departamento) corresponde al empleado “Steven King”, cuya clave primaria (columna EMPLOYEE_ID de la tabla referenciada EMPLOYEES) es el numero 100. A su vez, la columna LOCATION_ID (la localización del departamento) corresponde a “Seattle”, cuya clave primaria (columna LOCATION_ID de la tabla referenciada LOCATIONS) es el numero 1700.
Gracias a esta información adicional, los valores de las columnas de las tuplas listadas son mas fáciles de interpretar para un usuario, cosa que era uno de los objetivos principales de la aplicación.

Si en el menú principal pulsamos la opción *Insert forms* accederemos a los formularios para inserción de datos:



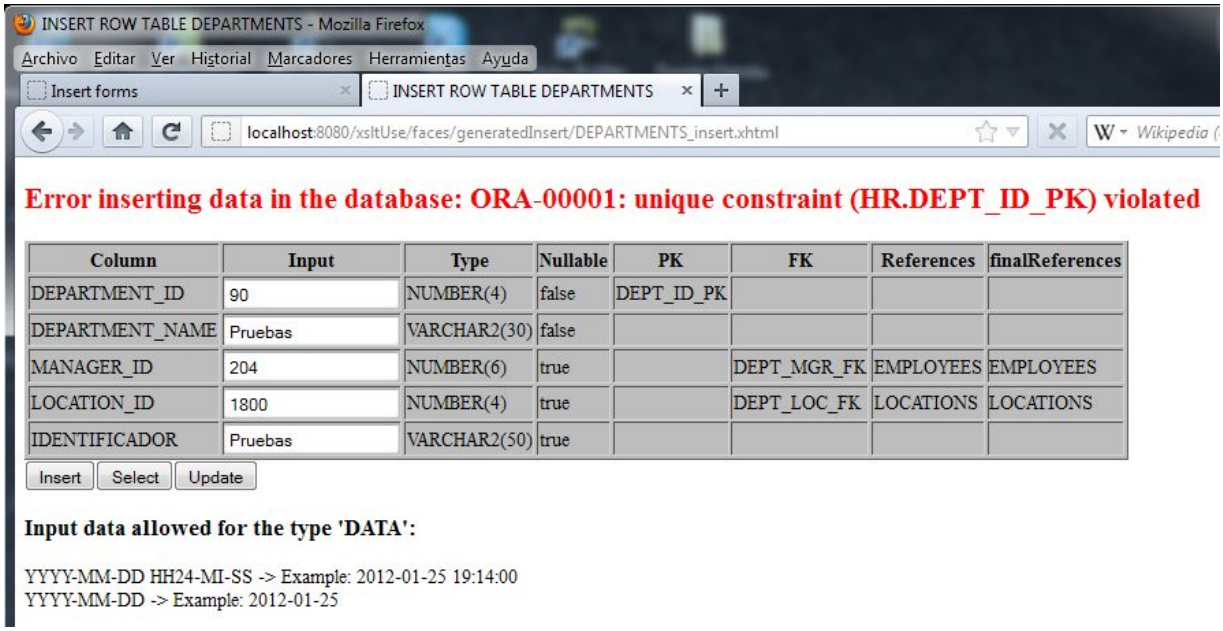
Seleccionamos el formulario *DEPARTMENTS_insert.xhtml:*



Este es el formulario para la inserción y actualización de datos de la tabla DEPARTMENTS. En el formulario podemos ver la información con los meta datos de las columnas de la tabla.

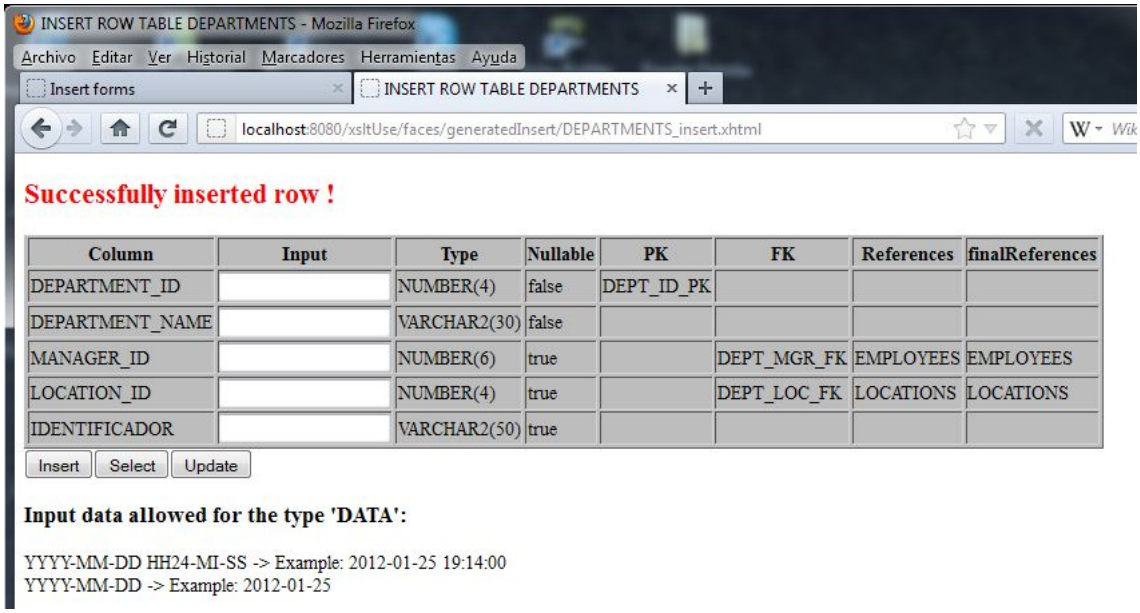
A continuación se realizan una serie de pruebas para ver las funcionalidades del formulario.

Si insertamos algún dato que viola alguna restricción de la base de datos o los datos insertados no son los correctos para el tipo esperado, se mostrara el tipo de error en la parte superior de la pagina. Si por ejemplo deseamos insertar un dato asignándole un valor para la clave primario que ya existe, se nos informaría mediante el siguiente mensaje de error:



Se muestra el error producido al intentar insertar el dato. La restricción DEPT_ID_PK que corresponde al echo de que la columna DEPARTMENT_ID es clave primaria, tiene que ser unica, lo que significa que ya existe una tupla con DEPARTMENT_ID = 90.

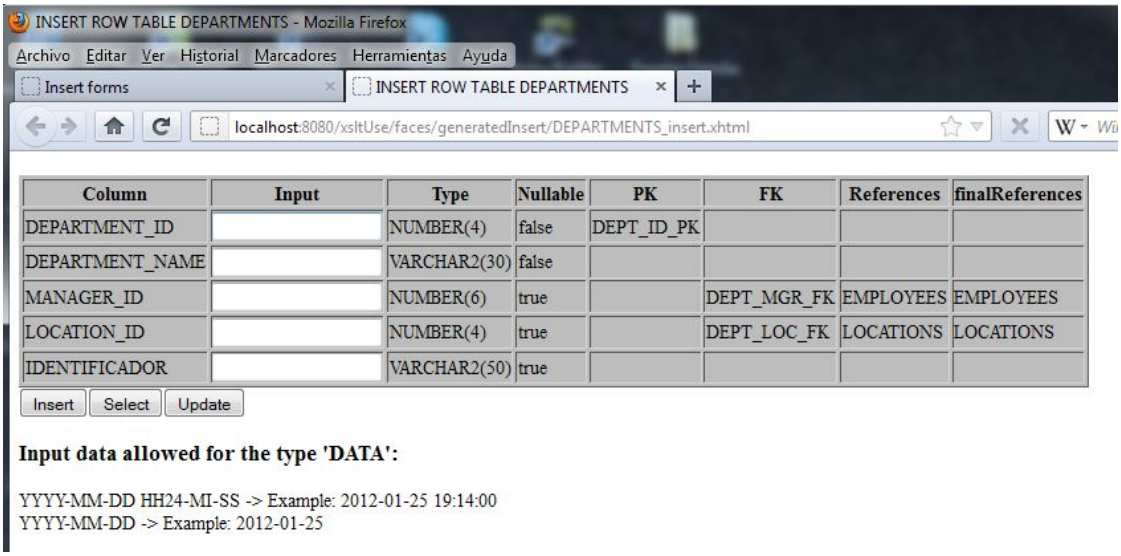
Si cambiamos solo el valor de esa columna por '300', (valor disponible para la clave primaria), se nos informa de que la tupla a sido insertada con éxito, y los campos se reinician para facilitar la inserción de una nueva tupla:



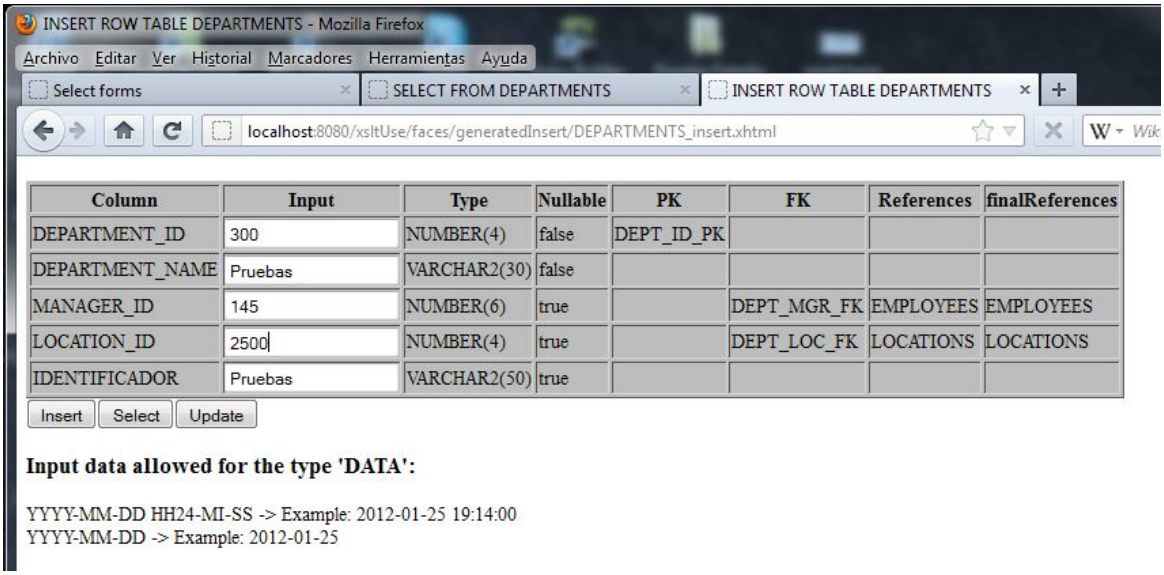
Si ahora seleccionamos el formulario de visualización de datos para la tabla DEPARTMENTS, podemos observar que los datos se han añadido con éxito:

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID	IDENTIFICADOR	MANAGER_ID EMPLOYEES.EMPLOYEE_ID	LOCATION_ID LOCATIONS.LOCATION_ID
90	Executive	100	1700	Executive	Steven King	Seattle
60	IT	103	1400	IT	Alexander Hunold	Southlake
100	Finance	108	1700	Finance	Nancy Greenberg	Seattle
30	Purchasing	114	1700	Purchasing	Den Raphaely	Seattle
50	Shipping	121	1500	Shipping	Adam Fripp	South San Francisco
80	Sales	145	2500	Sales	John Russell	Oxford
10	Administration	200	1700	Administration	Jennifer Whalen	Seattle
20	Marketing	201	1800	Marketing	Michael Hartstein	Toronto
40	Human Resources	203	2400	Human Resources	Susan Mavris	London
70	Public Relations	204	2700	Public Relations	Hermann Baer	Munich
300	Pruebas	204	1800	Pruebas	Hermann Baer	Toronto
110	Accounting	205	1700	Accounting	Shelley Higgins	Seattle
120	Treasury		1700	Treasury		Seattle
130	Corporate Tax		1700	Corporate Tax		Seattle
140	Control And Credit		1700	Control And Credit		Seattle

Volvemos al formulario de inserción de datos:

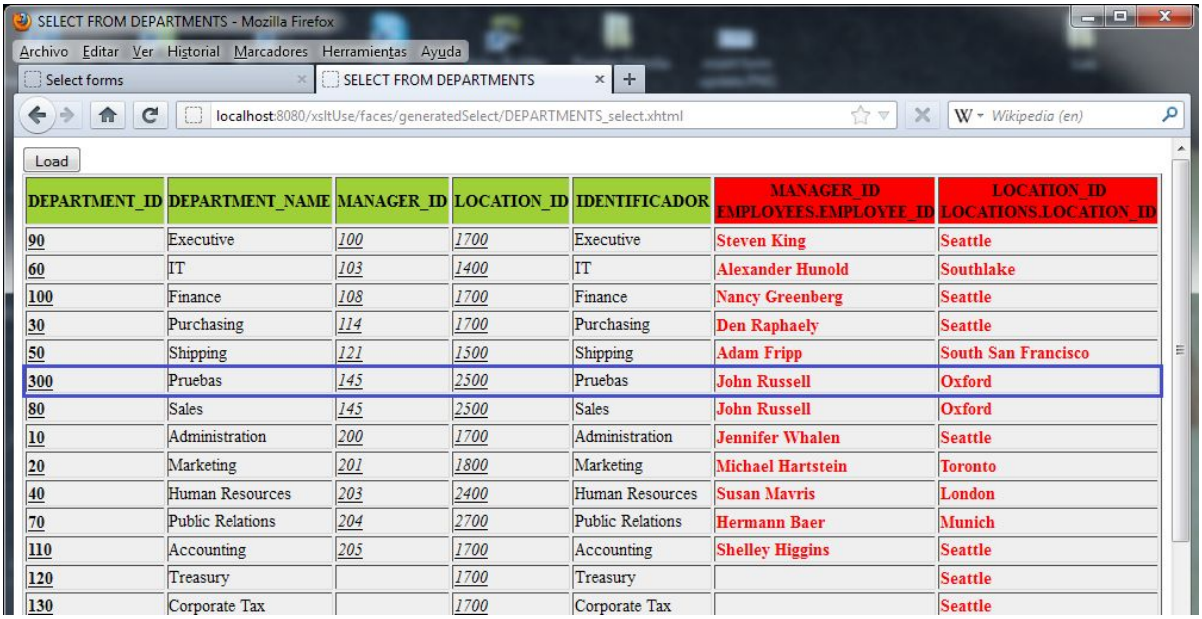


Si ahora insertamos en el campo DEPARTMENT_ID el valor 300 y pulsamos el botón “Select”, buscara en la tabla la tupla con valor DEPARTMENT_ID = 300, y cargara los datos de las columnas con los datos de la tupla si esta existe en la tabla (En este caso existe por que la acabamos de insertar con éxito):



De esta manera es sencillo modificar algún dato de la tupla, basta con dejar como estan los cargados y modificar los deseados.
Modificamos los campos MANAGER_ID (antes 204, Hermann Baer) por 145, que es el empleado John Russell, y el campo LOCATION_ID (antes 1800, Toronto) por 2500, Oxford, y pulsamos el botón *Update* para actualizar la tupla.
El botón *Insert* ejecuta una sentencia de inserción en la base de datos, por lo que no permite modificar datos ya que habrá una violación de la constante primary key para la tupla.
El botón *Update* ejecuta una sentencia update con los nuevos valores insertados, por lo que permite actualizar los valores de la tupla con clave primaria igual a la de la tupla actualizada.

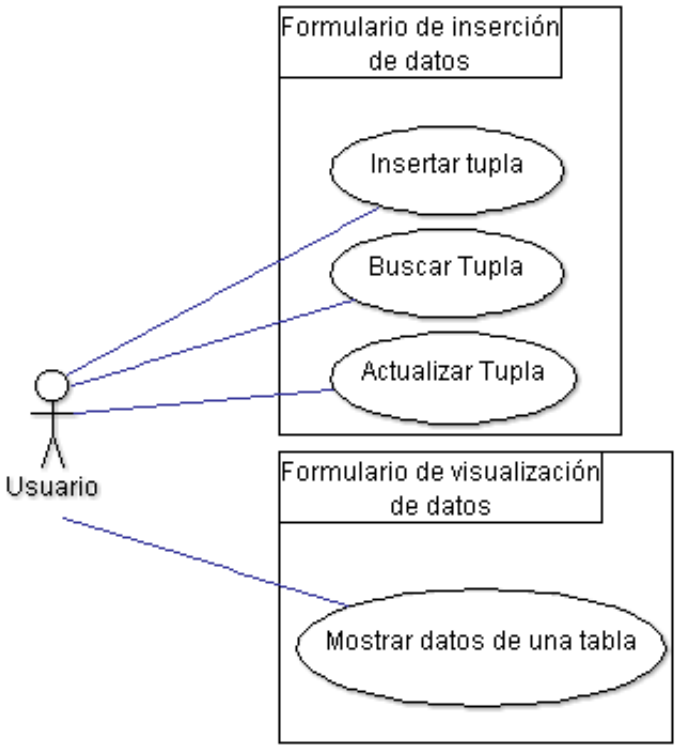
En el formulario de visualización de datos de la tabla DEPARTMENTS podemos ver que la tupla se a actualizado con éxito:



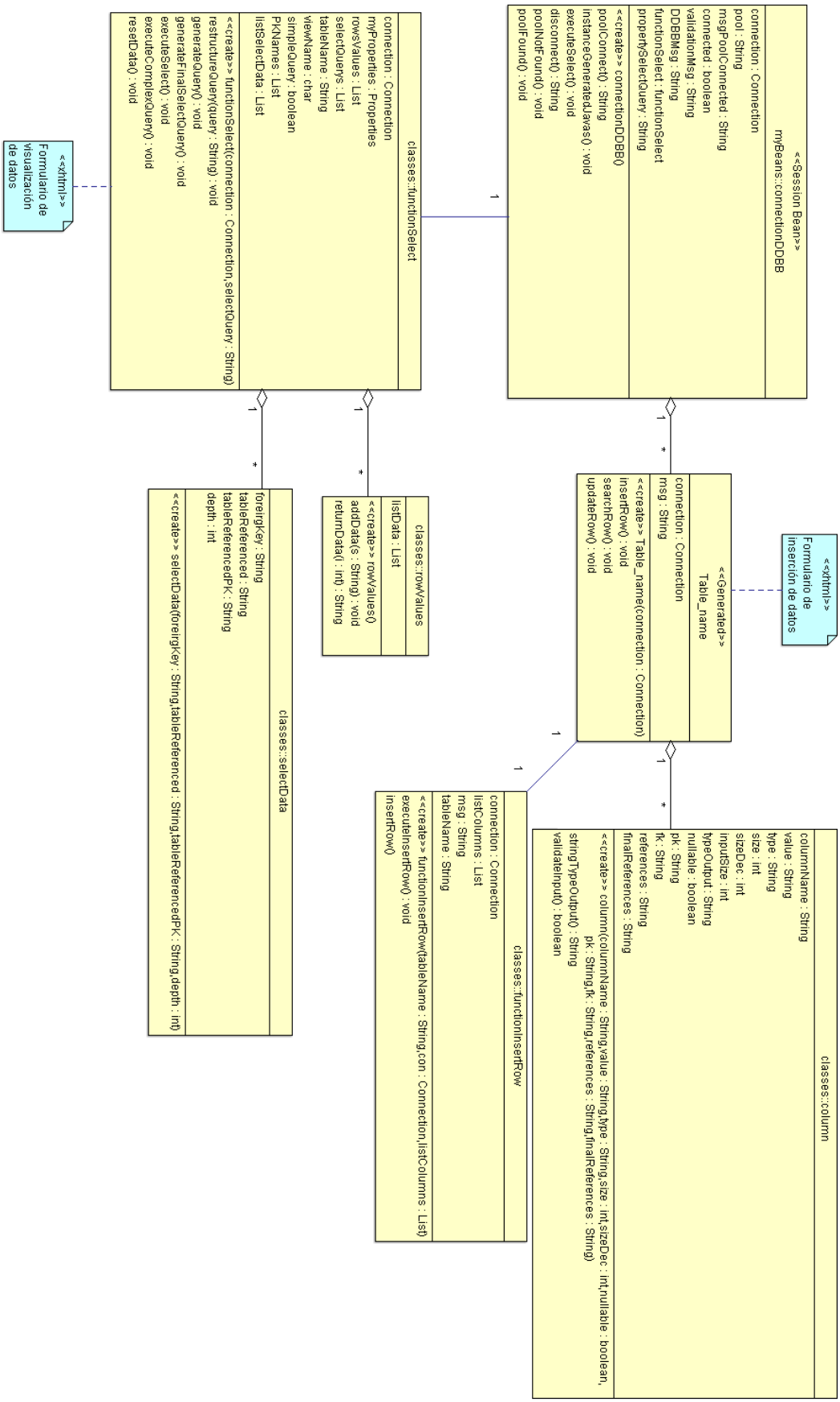
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID	IDENTIFICADOR	MANAGER_ID EMPLOYEES.EMPLOYEE_ID	LOCATION_ID LOCATIONS.LOCATION_ID
90	Executive	100	1700	Executive	Steven King	Seattle
60	IT	103	1400	IT	Alexander Hunold	Southlake
100	Finance	108	1700	Finance	Nancy Greenberg	Seattle
30	Purchasing	114	1700	Purchasing	Den Raphaely	Seattle
50	Shipping	121	1500	Shipping	Adam Fripp	South San Francisco
300	Pruebas	145	2500	Pruebas	John Russell	Oxford
80	Sales	145	2500	Sales	John Russell	Oxford
10	Administration	200	1700	Administration	Jennifer Whalen	Seattle
20	Marketing	201	1800	Marketing	Michael Hartstein	Toronto
40	Human Resources	203	2400	Human Resources	Susan Mavris	London
70	Public Relations	204	2700	Public Relations	Hermann Baer	Munich
110	Accounting	205	1700	Accounting	Shelley Higgins	Seattle
120	Treasury		1700	Treasury		Seattle
130	Corporate Tax		1700	Corporate Tax		Seattle

7- Arquitectura de la aplicación (Diagramas)

7.1- Casos de uso

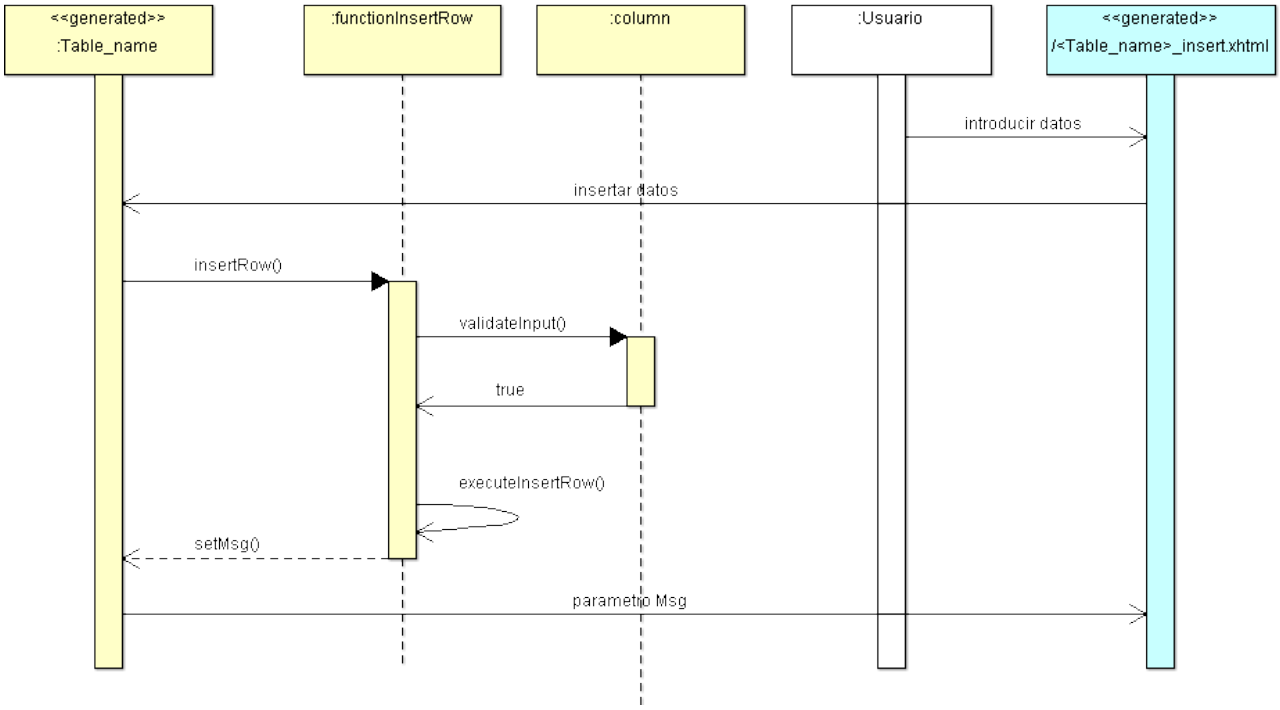


7.2- Diagrama de clases

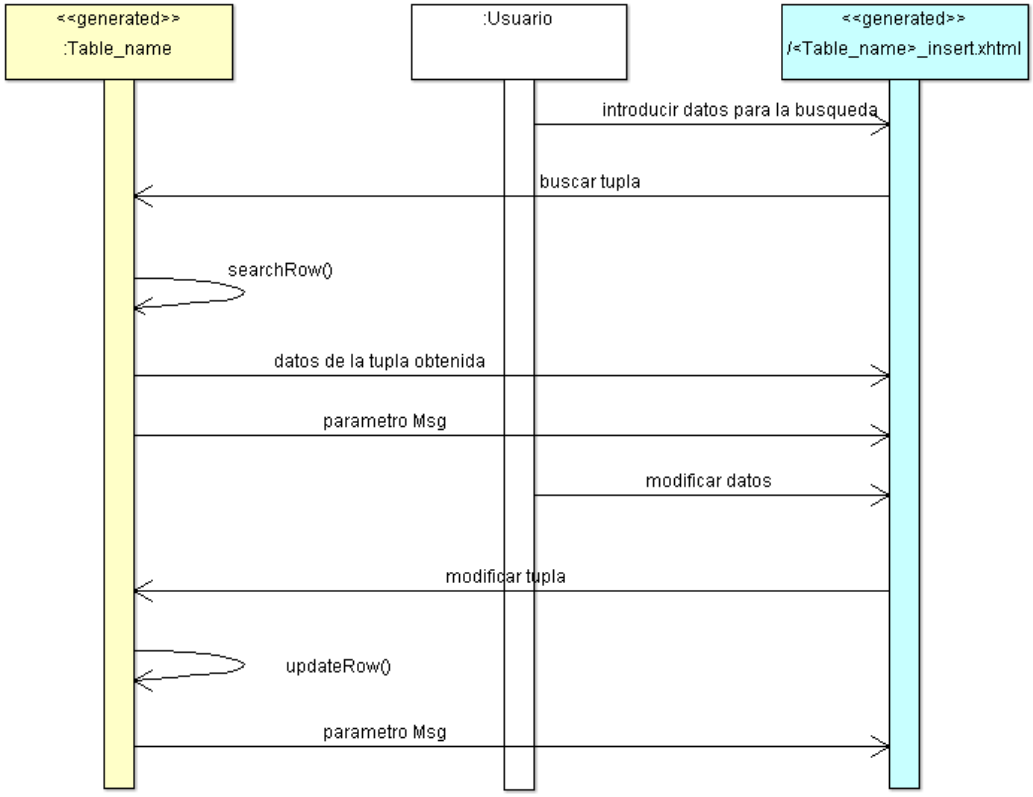


7.3- Diagramas de secuencia

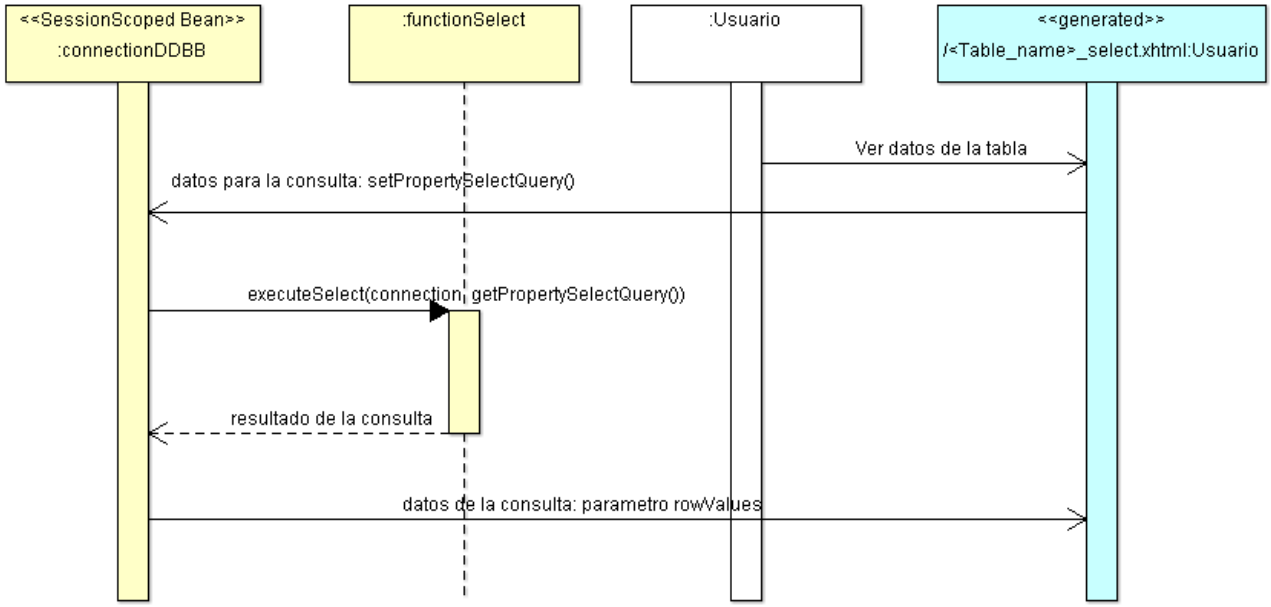
Introducir tupla



Buscar tupla y modificarla



Mostrar datos de una tabla



PLANTILLAS PARA LAS TRANSFORMACIONES XSL

NormalizeXML.xsl

Transforma los XML en XML “normalizados” para ser usados como XML de entrada para generar ficheros mediante transformaciones XSL

Codigo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- First transformation xml to xml. Extract from the orginal xml the columns
    values in a format oriented to create a xhtml form -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:exsl="http://exslt.org/common"
            version="1.0">

  <xsl:output method="xml"/>
  <!-- output type = xml -->

  <xsl:template match="/">

    <!-- Variable "pkList" with the primary keys -->
      <xsl:param name="pkList">
        <xsl:param name="pkName">
          <!-- name of the primary key-->
          <xsl:value-of select="xml/TABLE/PRIMARYKEY/@name"/>
        </xsl:param>
        <xsl:for-each select="xml/TABLE/PRIMARYKEY/COLUMN">
        <!-- for each column in the primary key element -->
          <pk>
          <!-- tag "pk" for the primary key info -->
            <xsl:attribute name="pkName">
            <!-- attribute pkName = name of the primary key-->
              <xsl:value-of select="$pkName"/>
            </xsl:attribute>
            <xsl:attribute name="pkColumn">
            <!-- attribute pkColumn = name of the column that is a primary key-->
              <xsl:value-of select="@columnName"/>
            </xsl:attribute>
          </pk>
        </xsl:for-each>
      </xsl:param >

    <!-- Variable "fkList" with the foreign keys -->
       <xsl:param name="fkList">
         <xsl:for-each select="xml/TABLE/FOREIGNKEY">
         <!-- for each foreign key -->
          <xsl:param name="fkName">
          <!-- name of the foreign key-->
            <xsl:value-of select="@name"/>
          </xsl:param>
          <xsl:param name="fkTable">
          <!-- name of the table referenced by the foreign key-->
            <xsl:value-of select="@referencedTable"/>
          </xsl:param>
          <xsl:for-each select="COLUMN">
          <!-- for each column that is part of the foreign key -->
            <fk>
```

```
            <!-- tag "fk" for the foreign key info -->
              <xsl:attribute name="fkColumn">
              <!-- attribute fkColumn = name of the column that is a foreign key-->
                <xsl:value-of select="@columnName"/>
              </xsl:attribute>
              <xsl:attribute name="fkName">
              <!-- attribute fkName = name of the foreign key-->
                <xsl:value-of select="$fkName"/>
              </xsl:attribute>
              <xsl:attribute name="fkTable">
              <!-- attribute fkTable = name of the table referenced-->
                <xsl:value-of select="$fkTable"/>
              </xsl:attribute>
               <xsl:attribute name="fkFinalTable">
               <!-- attribute fkFinalTable = name of the table referenced whit max depht value-->
                 <xsl:for-each select="FKMETADATA">
                   <xsl:choose>
                     <xsl:when test="position() = last()">
                        <xsl:value-of select="@table"/>
                     </xsl:when>
                   </xsl:choose>
                 </xsl:for-each>
              </xsl:attribute>
              <xsl:attribute name="fkFinalTableColumn">
              <!-- attribute fkFinalTableColumn =
                    name of the column at table referenced whit max depht value-->
                 <xsl:for-each select="FKMETADATA">
                   <xsl:choose>
                     <xsl:when test="position() = last()">
                        <xsl:value-of select="@name"/>
                     </xsl:when>
                   </xsl:choose>
                 </xsl:for-each>
              </xsl:attribute>
            </fk>
          </xsl:for-each>
        </xsl:for-each>
      </xsl:param >

<xsl:element name="TABLE">
<!-- element whit tag = TABLE -->
  <xsl:element name="NAME">
  <!-- element whit tag = NAME. Contains the table name -->
    <xsl:value-of select="xml/TABLE/@name"/>
  </xsl:element>

  <!-- Column information -->
  <xsl:for-each select="xml/TABLE/COLUMN">
        <!-- for each column -->
          <xsl:element name="COLUMN">
          <!-- element whit tag = COLUMN. Contains the column information -->
            <xsl:element name="NAME">
             <!-- element whit tag = NAME. Contains the column name -->
              <xsl:value-of select="@name"/>
            </xsl:element>
```

```
<xsl:element name="TYPE">
<!-- element whit tag = TYPE. Contains the column type -->
  <xsl:value-of select="@type"/>
</xsl:element>

<xsl:element name="SIZE">
<!-- element whit tag = SIZE. Contains the column type size -->
  <xsl:value-of select="@size"/>
</xsl:element>

<xsl:element name="SIZEDEC">
  <!-- element whit tag = SIZEDEC. Contains the type decimal size -->
  <xsl:choose>
    <xsl:when test="@sizeDec = ''">0</xsl:when>
    <!-- If sizeDec = null, then sizeDec = 0-->
    <xsl:otherwise>
      <xsl:value-of select="@sizeDec" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:element>

<xsl:element name="NULLABLE">
<!-- element whit tag = NULLABLE. Contains the column nullable value -->
  <xsl:value-of select="@nullable"/>
</xsl:element>

<!-- Set primary key if the column is one -->
<xsl:element name="PK">
 <!-- element whit tag = PK. Contains the column primary key info if the columns is one -->
  <xsl:variable name="name">
    <xsl:value-of select="@name"/>
  </xsl:variable>
  <xsl:for-each select="exsl:node-set($pkList)/pk" >
  <!-- go over the structure with the primary keys info -->
    <xsl:variable name="unPk">
      <xsl:value-of select="@pkColumn"/>
    </xsl:variable>
      <xsl:if test="$unPk = $name">
      <!-- If the name of the column is in the primary key data structure,
           the column is a primary key -->
        <xsl:value-of select="@pkName"/>
        <!-- add the primaty key name-->
      </xsl:if>
  </xsl:for-each>
</xsl:element>

<!-- Set foreign key if the column is one -->
<xsl:element name="FK">
<!-- element whit tag = FK. Contains the column foreign key info if the columns is one -->
  <xsl:variable name="name">
    <xsl:value-of select="@name"/>
  </xsl:variable>
  <xsl:for-each select="exsl:node-set($fkList)/fk" >
  <!-- go over the structure with the foreign keys info -->
    <xsl:variable name="unFk">
      <xsl:value-of select="@fkColumn"/>
```

```
            </xsl:variable>
              <xsl:if test="$unFk = $name">
              <!-- If the name of the column is in the foreign key data structure,
                   the column is a foreign key -->
                <xsl:element name="FKNAME">
                <!-- element whit tag = FKNAME. Contains the foreign key name -->
                  <xsl:value-of select="@fkName"/>
                </xsl:element>
                <xsl:element name="REFERENCES">
                <!-- element whit tag = REFERENCES. Contains the foreign key table referenced -->
                  <xsl:value-of select="@fkTable"/>
                </xsl:element>
                <xsl:element name="FINALREFERENCES">
                <!-- element whit tag = FINALREFERENCES.
                     Contains the foreign key table final referenced->
                  <xsl:value-of select="@fkFinalTable"/>
                </xsl:element>
                <xsl:element name="FINALCOLREFERENCES">
                <!-- element whit tag = FINALCOLREFERENCES.
                      Contains the column of the table final referenced by the key -->
                  <xsl:value-of select="@fkFinalTableColumn"/>
                </xsl:element>
                <xsl:element name="FKNUM">
                <!-- element whit tag = FKNUM. Order of the foreign key -->
                   <xsl:value-of select="position()"/>
                </xsl:element>

              </xsl:if>
          </xsl:for-each>
        </xsl:element>

      </xsl:element>
    </xsl:for-each>
```

```
<!-- Information for the select querys and the foreign keys-->
    <xsl:for-each select="xml/TABLE/FOREIGNKEY">
    <!-- For each foreign key -->
      <xsl:element name="FKSELECT">
      <!-- element whit tag = FKSELECT. Contains the info for a database select query -->
        <xsl:for-each select="COLUMN">
        <!-- For each column that contains the foreign key -->
          <xsl:variable name="FKName">
            <xsl:value-of select="@columnName"/>
          </xsl:variable>
          <xsl:for-each select="FKMETADATA">
          <!-- For each column found in the depth search -->
            <xsl:element name="FKSELECTWHERE">
            <!-- element whit tag = FKSELECTWHERE. Contains the info for a depth query-->
              <xsl:element name="FKCOLUMN">
              <!-- column to search in the table referenced -->
                <xsl:value-of select="$FKName"/>
              </xsl:element>
              <xsl:element name="TABLEREFERENCED">
              <!-- table referenced -->
                <xsl:value-of select="@table"/>
```

```
                    </xsl:element>
                    <xsl:element name="PKCOLUMN">
                    <!-- column in the current table that import the key -->
                      <xsl:value-of select="@name"/>
                    </xsl:element>
                    <xsl:element name="DEPTH">
                    <!-- Depth of the column in the in-depth search -->
                      <xsl:value-of select="@depth"/>
                    </xsl:element>
                  </xsl:element>
                </xsl:for-each>
              </xsl:for-each>
            </xsl:element>
          </xsl:for-each>

    </xsl:element>  <!-- End of element  TABLE -->
  </xsl:template>

</xsl:stylesheet>
```

XSLTtoJava.xsl

Transforma los XML “normalizados” en clases Java con los metadatos de una tabla

Codigo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="text" indent="no" encoding="UTF-8" />
    <!-- output type = text -->
   <xsl:template match="/">

   <xsl:variable name="table">
   <!-- variable table = table name -->
      <xsl:value-of select="TABLE/NAME"/>
   </xsl:variable>
   import java.sql.Connection;
   import java.util.LinkedList;
   import java.util.List;
   import classes.column;
   import classes.functionInsertRow;
   import java.sql.ResultSet;
   import java.sql.ResultSetMetaData;
   import java.sql.SQLException;
   import java.sql.Statement;

   /**
   * Class <xsl:value-of select="$table"/>
   */
   public class <xsl:value-of select="$table"/> {

      private List&lt;column> listColumns;
      private String msg;
      private Connection connection;

      <!-- Constructor -->
       /**
       * Constructor: initializes a class column linked list with the columns metadata from the table
       * @param connection the connection to the database
       */
            public <xsl:value-of select="$table"/>(Connection connection) {
               this.listColumns = new LinkedList&lt;column>();
               this.connection = connection;
               this.msg = "";
               <!-- Fill the columns with the columns metadata -->
               <xsl:for-each select="TABLE/COLUMN">
               <!-- For each column in the normalize XML, add the structure with all its values -->
                  this.listColumns.add(new column("<xsl:value-of select="NAME"/>",
                                                  "",
                                                  "<xsl:value-of select="TYPE"/>",
                                                  <xsl:value-of select="SIZE"/>,
                                                  <xsl:value-of select="SIZEDEC"/>,
                                                  <xsl:value-of select="NULLABLE = 1"/>,
                                                 "<xsl:value-of select="PK"/>",
                                                 "<xsl:value-of select="FK/FKNAME"/>",
                                                 "<xsl:value-of select="FK/REFERENCES"/>",
```

```
                                                    "<xsl:value-of select="FK/FINALREFERENCES"/>"));
        </xsl:for-each>
    }
```

```
<!-- Insert row in the data base. Uses the generic function insertRow -->
    /**
    * Call a class functionInsertRow with the columns linked list as a parameter
    * The class uses that list to generate a insert statement
    */
    public void insertRow(){
       functionInsertRow fiRow =
         new functionInsertRow("<xsl:value-of select="$table"/>",connection, listColumns);
       this.setMsg(fiRow.getMsg());
    }
```

```
<!-- Search in the database a row with the an specific primary key -->
    /**
     * Fill the primary key fields and select the appropiate row in the data base
     */
    public void searchRow(){

      String query = "Select * from <xsl:value-of select="$table"/> where ";
      int wheres = 0;
      int values = 0;

       for(int i = 0; i &lt; this.listColumns.size();i++){
        column col = this.listColumns.get(i);
        if(col.getPk().length()>0){
          if(wheres > 0){
            query = query.concat(" AND ");
          }
          wheres++;
          if(col.getType().equals("DATE")){
             if(col.getValue().length()==19){
               query = query.concat("to_char(" + col.getColumnName() + ",'yyyy-mm-dd hh24:mi:ss')=");
             } else
             if(col.getValue().length()==10){
               query = query.concat("to_char(" + col.getColumnName() + ",'yyyy-mm-dd')=");
             }
          } else {
            query = query.concat(col.getColumnName() + "=");
          }
          if(col.getType().equals("NUMBER")){
          // if type numbre, put value
            query = query.concat(col.getValue());
          } else
          if(col.getType().equals("DATE")){
          // if type date, to_date('value')
            query = query.concat("'" + col.getValue() + "'");
          } else {
          //String type
             query = query.concat("'" + col.getValue() + "'");
          }

        }
      }
```

```
        try {
          Statement statement = this.connection.createStatement();
          ResultSet res = statement.executeQuery("" + query);
          ResultSetMetaData rsmd = res.getMetaData();
          int numberOfColumns = rsmd.getColumnCount();
          while (res.next()) {
            values++;
            for (int j=1; j &lt;= numberOfColumns ; j++){
              column col = this.listColumns.get(j-1);
              col.setValue(res.getString(j));
              this.listColumns.set(j-1, col);

                                  }
                          }
          if(values == 0){
             this.msg = "No rows selected";
          } else {
             this.msg = "";
          }
        } catch (SQLException e) {
          System.out.println("Search error: " + e.getLocalizedMessage());
        }
      }
```

```
<!-- Update a row in the data base -->
      /**
       * Update a database row
       */
      public void updateRow(){
        /* UPDATE table_name SET column1=value, column2=value2,... WHERE
some_column=some_value */

        String query = "UPDATE <xsl:value-of select="$table"/> SET ";
        int wheres = 0;


        for(int i = 0; i &lt; this.listColumns.size();i++){
          column col = this.listColumns.get(i);

          query = query.concat(col.getColumnName() + "=");
          if(col.getType().equals("NUMBER")){
          // if type numbre, put value
            if(col.getValue().length() > 0){
              query = query.concat(col.getValue());
            } else {
               query = query.concat("null");
            }
          } else
          if(col.getType().equals("DATE")){
          // if type date, to_date('value')
            if(col.getValue().length() > 0){
              query = query.concat("to_date('" + col.getValue() + "','");
              if(col.getValue().length() == 19){
                // 2012-01-25 19:14:00
                query = query.concat("YYYY-MM-DD HH24:MI:SS')");
              } else if(col.getValue().length() == 10){
```

```
                // 2012-01-25
                query = query.concat("YYYY-MM-DD')");
              }
            } else {
               query = query.concat("null");
            }
          } else {
          //String type
             query = query.concat("'" + col.getValue() + "'");
          }
          query = query.concat(", ");
        }
        // Delete the last 2 characters = ", "
        query = query.substring(0, query.length()-2);
        query = query.concat(" WHERE ");

        // Add the primary keys values
         for(int i = 0; i &lt; this.listColumns.size();i++){
           column col = this.listColumns.get(i);
           if(col.getPk().length()>0){
             if(wheres > 0){
               query = query.concat(" AND ");
             }
             wheres++;
             if(col.getType().equals("DATE")){
               query = query.concat("to_char(" + col.getColumnName() + ",'");
               if(col.getValue().length() == 19){
                 // 2012/01/25:19:14:00
                 query = query.concat("YYYY-MM-DD HH24:MI:SS')=");
               } else if(col.getValue().length() == 10){
                 // 2012/01/25
                 query = query.concat("YYYY-MM-DD')=");
               }
             } else {
               query = query.concat(col.getColumnName() + "=");
             }
             if(col.getType().equals("NUMBER")){
             // if type numbre, put value
               query = query.concat(col.getValue());
             }  else {
             //String type
                query = query.concat("'" + col.getValue() + "'");
             }

           }
         }
         // insert in the data base
         try{
            Statement statement = this.connection.createStatement();
            statement.execute("" + query);
            statement.close();
            //resert input values
            for(int i = 0; i &lt; this.listColumns.size();i++){
               this.listColumns.get(i).setValue("");
            }
            this.setMsg("Successfully update row !");
```

```
            } catch (SQLException e){
                this.setMsg("Error updating row in the database: " + e.getLocalizedMessage());
            }

        }
```

```
<!-- Setters and Getters -->
        /* Setters and getters */
        public List&lt;column> getListColumns() {
            return listColumns;
        }
        public void setListColumns(List&lt;column> listColumns) {
            this.listColumns = listColumns;
        }
        public String getMsg() {
            return msg;
        }
        public void setMsg(String msg) {
            this.msg = msg;
        }
    }
```

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

XSLTtoSimpleJava.xsl

Transforma los XML “normalizados” en clases Java simplificadas fijándose únicamente en los nombres de las columnas de una tabla

Codigo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="text" indent="no" encoding="UTF-8" />
   <!-- output type = text-->
   <xsl:template match="/">

   <!-- Vataible $table = table name-->
   <xsl:variable name="table">
   <!-- table = table name -->
      <xsl:value-of select="TABLE/NAME"/>
   </xsl:variable>

   public class <xsl:value-of select="$table"/> {

      <!-- Declare class attributes -->
         //Attributes
         <xsl:for-each select="TABLE/COLUMN">
         <!-- for each column -->
           <xsl:variable name="type">
             <xsl:value-of select="TYPE"/>
           </xsl:variable>
           <!-- If the column is a oracle type = NUMBER then java type = float. Else, java type = String -->
           <xsl:if test="$type = 'VARCHAR2'">
            <xsl:text>String </xsl:text><xsl:value-of select="NAME"/> <xsl:text>;</xsl:text>
           </xsl:if>
           <xsl:if test="$type = 'CHAR'">
             <xsl:text>String </xsl:text><xsl:value-of select="NAME"/> <xsl:text>;</xsl:text>
           </xsl:if>
           <xsl:if test="$type = 'DATE'">
             <xsl:text>String </xsl:text><xsl:value-of select="NAME"/> <xsl:text>;</xsl:text>
           </xsl:if>
           <xsl:if test="$type = 'NUMBER'">
             <xsl:text>float </xsl:text><xsl:value-of select="NAME"/> <xsl:text>;</xsl:text>
           </xsl:if>
           <xsl:text>
           </xsl:text>
         </xsl:for-each>

      <!-- Constructor -->
         //Constructor
         public <xsl:value-of select="$table"/>() {

         }

      <!-- Setters and Getters -->
         <!-- Setters and getters according with the attributes type -->
         // Setters and getters
         <xsl:for-each select="TABLE/COLUMN">
```

```
        <xsl:variable name="type">
          <xsl:value-of select="TYPE"/>
        </xsl:variable>
        <xsl:if test="$type = 'VARCHAR2'">
          public String get<xsl:value-of select="NAME"/>(){
            return this.<xsl:value-of select="NAME"/>;
          }
          public void set<xsl:value-of select="NAME"/>(String <xsl:value-of select="NAME"/>){
             this.<xsl:value-of select="NAME"/>=<xsl:value-of select="NAME"/>;
          }
        </xsl:if>
        <xsl:if test="$type = 'CHAR'">
          public String get<xsl:value-of select="NAME"/>(){
            return this.<xsl:value-of select="NAME"/>;
          }
          public void set<xsl:value-of select="NAME"/>(String <xsl:value-of select="NAME"/>){
             this.<xsl:value-of select="NAME"/>=<xsl:value-of select="NAME"/>;
          }
        </xsl:if>
        <xsl:if test="$type = 'DATE'">
          public String get<xsl:value-of select="NAME"/>(){
            return this.<xsl:value-of select="NAME"/>;
          }
          public void set<xsl:value-of select="NAME"/>(String <xsl:value-of select="NAME"/>){
             this.<xsl:value-of select="NAME"/>=<xsl:value-of select="NAME"/>;
          }
        </xsl:if>
        <xsl:if test="$type = 'NUMBER'">
          public float get<xsl:value-of select="NAME"/>(){
            return this.<xsl:value-of select="NAME"/>;
          }
          public void set<xsl:value-of select="NAME"/>(float <xsl:value-of select="NAME"/>){
             this.<xsl:value-of select="NAME"/>=<xsl:value-of select="NAME"/>;
          }
        </xsl:if>
      </xsl:for-each>

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

XSLTtoForm.xsl

Transforma los XML “normalizados” en xhtml para la inserción y modificación de datos usando las clases java generadas.

Codigo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:f="http://java.sun.com/jsf/core"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns="http://www.w3.org/1999/xhtml"
                version="1.0">
 <xsl:output method="xml" indent="yes" encoding="UTF-8" />
 <!-- output type = xhtml -->
 <xsl:template match="/">
   <html>
     <!-- HEAD -->
     <h:head>
       <title>INSERT ROW TABLE <xsl:value-of select="TABLE/NAME"/>
       </title>
     </h:head>
     <!-- BODY -->
     <h:body>
       <f:view>
         <h:form>
           <!-- Messages from the class -->
                   <xsl:variable name="msg">
                     #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.msg}
                   </xsl:variable>
                   <h2 style="color: red"><h:outputLabel value="{$msg}"/></h2>
                   <!-- List of data columns from the class -->
                   <xsl:variable name="class">
                     #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.listColumns}
                   </xsl:variable>
                   <!-- variable class = java class with the columns info for the table-->
                   <h:dataTable border="1" bgcolor="#B9B9B9" var="item" value="{$class}">
                     <h:column>
                       <f:facet name="header"><xsl:text>Column</xsl:text></f:facet>
                       <!-- Column name-->
                        <xsl:variable name="item">#{item.columnName}</xsl:variable>
                       <h:outputLabel value="{$item}"/>
                     </h:column>
                      <h:column>
                       <f:facet name="header"><xsl:text>Input</xsl:text></f:facet>
                       <!-- Input text field, with attributes column value, size and required=(!nullable) -->
                       <xsl:variable name="item">#{item.value}</xsl:variable>
                       <xsl:variable name="size">#{item.inputSize}</xsl:variable>
                       <h:inputText value="{$item}" maxlength="{$size}"/>
                     </h:column>
                     <h:column>
                       <f:facet name="header"><xsl:text>Type</xsl:text></f:facet>
                       <!-- Type and size, in an string sturcture-->
                       <xsl:variable name="item">#{item.typeOutput}</xsl:variable>
                       <h:outputLabel value="{$item}"/>
                     </h:column>
```

```
                <h:column>
                  <f:facet name="header"><xsl:text>Nullable</xsl:text></f:facet>
                  <!-- column value nullable? -->
                  <xsl:variable name="item">#{item.nullable}</xsl:variable>
                  <h:outputLabel value="{$item}"/>
                </h:column>
                <h:column>
                  <f:facet name="header"><xsl:text>PK</xsl:text></f:facet>
                  <!-- if primary key, set the primary key name-->
                  <xsl:variable name="item">#{item.pk}</xsl:variable>
                  <h:outputLabel value="{$item}"/>
                </h:column>
                <h:column>
                  <f:facet name="header"><xsl:text>FK</xsl:text></f:facet>
                  <!-- if foreign key, set the foreign key name-->
                  <xsl:variable name="item">#{item.fk}</xsl:variable>
                  <h:outputLabel value="{$item}"/>
                </h:column>
                <h:column>
                  <f:facet name="header"><xsl:text>References</xsl:text></f:facet>
                  <!-- if foreign key, set the referenced table by the column-->
                  <xsl:variable name="item">#{item.references}</xsl:variable>
                  <h:outputLabel value="{$item}"/>
                </h:column>
                <h:column>
                  <f:facet name="header"><xsl:text>finalReferences</xsl:text></f:facet>
                  <!-- if foreign key, set the final referenced table by the column-->
                  <xsl:variable name="item">#{item.finalReferences}</xsl:variable>
                  <h:outputLabel value="{$item}"/>
                </h:column>
              </h:dataTable>
```

```
<!-- insert row method for the class-->
            <!-- insert row button -->
            <xsl:variable name="action">
               #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.insertRow()}
            </xsl:variable>
            <h:commandButton value="Insert" action="{$action}"/>
            <!-- search row button -->
            <xsl:variable name="action2">
               #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.searchRow()}
            </xsl:variable>
            <h:commandButton value="Select" action="{$action2}"/>
            <!-- update row button -->
            <xsl:variable name="action3">
              #{connectionDDBB.<xsl:value-of select="TABLE/NAME"/>.updateRow()}
            </xsl:variable>
            <h:commandButton value="Update" action="{$action3}"/>
```

```
    </h:form>
  </f:view>
  <!-- Text with info about the oracle type DATA input values -->
  <h3>Input data allowed for the type 'DATA':</h3>
  YYYY-MM-DD HH24-MI-SS  -> Example: 2012-01-25 19:14:00 <br></br>
  YYYY-MM-DD -> Example: 2012-01-25
</h:body>
```

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

XSLTtoSelect.xsl

Transforma los XML “normalizados” en xhtml para la visualización de los datos de una tabla.

Codigo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:f="http://java.sun.com/jsf/core"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns="http://www.w3.org/1999/xhtml"
                version="1.0">
  <xsl:output method="xml" indent="yes" encoding="UTF-8" />
  <!-- output type = xhtml -->
  <xsl:template match="/">
    <html>

      <!-- SET THE APPROPIATE SELECT QUERY (stored in the variable "select") -->
           <!-- Generate a string with the info of the foreign keys-->
           <!-- Example input:
             Value = "DEPARTMENTS,DEPARTMENT_ID=|
      MANAGER_ID,EMPLOYEES,EMPLOYEE_ID,1,=|LOCATION_ID,LOCATIONS,LOCATION_ID,1,"
           -->
          <xsl:variable name="tableName">
            <xsl:value-of select="TABLE/NAME"/>
          </xsl:variable>
          <xsl:variable name="select">
            <!-- First value = name of the table-->
            <xsl:text><xsl:value-of select="$tableName"/></xsl:text>
             <xsl:for-each select="TABLE/COLUMN">
             <!-- For each column, if is primary key, put the column name -->
              <xsl:if test="count(PK/node()) > 0">
                 <xsl:text>,</xsl:text>
                 <xsl:text><xsl:value-of select="NAME"/></xsl:text>
              </xsl:if>
            </xsl:for-each>
            <xsl:for-each select="TABLE/FKSELECT">
            <!-- for each select query to do in the data base-->
              <xsl:text>=</xsl:text>
              <xsl:for-each select="FKSELECTWHERE">
              <!-- for each info about the where clausule-->
                <xsl:text>|</xsl:text>
                <!-- foreign key in the table -->
                <xsl:text><xsl:value-of select="FKCOLUMN"/></xsl:text>
                <xsl:text>,</xsl:text>
                <!-- table referenced by the foreign key -->
                <xsl:text><xsl:value-of select="TABLEREFERENCED"/></xsl:text>
                <xsl:text>,</xsl:text>
                <!--  primary key of the table referenced-->
                <xsl:text><xsl:value-of select="PKCOLUMN"/></xsl:text>
                <xsl:text>,</xsl:text>
                <!-- depth of the key search -->
                <xsl:text><xsl:value-of select="DEPTH"/></xsl:text>
                <xsl:text>,</xsl:text>
              </xsl:for-each>
            </xsl:for-each>
```

```
    </xsl:variable>

  <!-- css styles -->
  <style type="text/css" >
    .heading{
      font-weight: bold;
      background-color: yellowgreen;
    }
    .pk{
       font-weight: bold;
       text-decoration: underline;

    }
    .fk{
       font-style: italic;
       text-decoration: underline;
    }
    .imported{
       background-color: red;
    }
    .importedLabel{
       font-weight: bold;
       color: red;
    }

  </style>

<!-- HEAD -->
<h:head>
  <title>SELECT FROM <xsl:value-of select="TABLE/NAME"/></title>
</h:head>
<!-- BODY -->
<h:body>
  <f:view>
    <h:form>
      <!-- source for the acction executeSelect-->
      <xsl:variable name="valueButton">#{connectionDDBB.executeSelect()}</xsl:variable>
      <!-- Button load = execute the select query in the database -->
      <h:commandButton value="Load" action="{$valueButton}">
         <xsl:variable name="table">
            <xsl:value-of select="TABLE/NAME"/>
         </xsl:variable>
         <!-- store in the property of the class the select string with the info to generate the select query -->
         <xsl:variable name="target">#{connectionDDBB.propertySelectQuery}</xsl:variable>
        <f:setPropertyActionListener target="{$target}" value="{$select}" />
      </h:commandButton>
      <!-- Link to attribute row values with the return info for the select query -->
      <xsl:variable name="valueTabla">#{connectionDDBB.functionSelect.rowsValues}</xsl:variable>
      <h:dataTable border="1" bgcolor="#EEEEEE" var="item" value="{$valueTabla}"
                   headerClass="heading">

        <xsl:for-each select="TABLE/COLUMN">
            <!-- For each column, a row values must be returned -->
              <h:column>
                 <f:facet name="header"><xsl:value-of select="NAME"/></f:facet>
                 <!-- variable value = position inside the returned vector for each row -->
```

```
                <xsl:variable name="value">
                  #{item.returnData(<xsl:value-of select="position()-1"/>)}
                </xsl:variable>
                <xsl:choose>
                  <!-- if the value is a primary key, apply a pk style -->
                  <xsl:when test="count(PK/node()) > 0">
                    <h:outputLabel styleClass="pk" value="{$value}"/>
                  </xsl:when>
                  <!-- if the value is a foreign key, apply a fk style -->
                  <xsl:when test="(count(FK/node()) > 0) and (count(PK/node()) = 0)">
                    <h:outputLabel styleClass="fk" value="{$value}"/>
                  </xsl:when>
                  <xsl:otherwise>
                    <h:outputLabel  value="{$value}"/>
                  </xsl:otherwise>
                </xsl:choose>
              </h:column>
            </xsl:for-each>
```

```
<!-- ADD THE FOREIGN COLUMNS WITH THE IDENTIFICADTOR FIELS -->
            <xsl:variable name="totCol" select="count(TABLE/COLUMN)"/>
            <xsl:for-each select="TABLE/COLUMN">
            <!-- for each column, if the column is a foreign key, add a column in the xhtml table
                whit the additional info for the foreign key-->
              <xsl:if test="count(FK/FKNUM/node()) > 0">
                <h:column headerClass="imported">
                  <f:facet name="header">
                    <xsl:value-of select="NAME"/><br></br>
                    <xsl:value-of select="FK/FINALREFERENCES"/>.
                    <xsl:value-of select="FK/FINALCOLREFERENCES"/></f:facet>
                  <xsl:variable name="fkpos" select="FK/FKNUM"/>
                  <xsl:variable name="value">
                    #{item.returnData(<xsl:value-of select="$totCol + $fkpos – 1"/>)}
                  </xsl:variable>
                  <h:outputLabel  value="{$value}" styleClass="importedLabel"/>
                </h:column>
              </xsl:if>
            </xsl:for-each>
```

```
        </h:dataTable>
      </h:form>
    </f:view>
  </h:body>
  </html>
</xsl:template>
</xsl:stylesheet>
```