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

Alejandro Lecina Laplana
Ingeniería en Informática
Junio 2012

Índice de contenido

1- Introducción

1.1- Contexto del PFC

Este Proyecto de Fin de Carrera fue ofrecido por el Observatorio Tecnológico HP durante el curso 2010-2011.
El Observatorio Tecnológico HP de la Universidad de Zaragoza es un espacio dentro de la universidad donde los estudiantes pueden realizar sus Proyectos Fin de Carrera supervisados por personal especializado de la empresa Hewlett-Packard. [1]
El director del proyecto, Javier Diaz, fue el empleado de HP que superviso el proyecto.
El proyecto surgió como una herramienta paralela de ayuda para el producto SILO desarrollado por el departamento de logística de HP, especialmente para la interacción con la base de datos del mismo.
HP Warehouse Management System (SILO) es un producto que proporciona software y servicios para gestionar las actividades de almacén, personal, desempeño, y el inventario, así como los recursos físicos, como las paletas, la robótica, los camiones y de almacenamiento. [2]
El proyecto parte de una motivación concreta y pretende alcanzar unos objetivos bien definidos usando unas determinadas herramientas. El proceso para alcanzar los objetivos no estaba definido, y es una de las partes mas importantes del proyecto.

1.2- Motivación

Las aplicaciones empresariales necesitan, para tareas de soporte, pantallas que accedan directamente a los datos "en crudo" contenidos en las tablas de la base de datos, y que permitan editar, visualizar o insertar nuevos. Existen algunas herramientas que permiten generar una versión inicial de las clases Java que conforman estas pantallas, pero requieren siempre de ajustes manuales para que sean debidamente utilizables por un usuario. Esto se debe a varias causas, que están relacionados con la falta de visibilidad sobre la relación referencial (claves externas) entre las diversas tablas de una aplicación.
Pensando en el software SILO, se propone crear un generador automático de clases Java que tenga en cuenta los metadatos que cualquier base de datos contiene acerca de la estructura referencial de las entidades que la componen, y los utilice para adecuar un conjunto de plantillas existentes y así generar clases Java que conformen pantallas de soporte finales, sin necesidad de ajustes posteriores por parte de desarrolladores.
Por otra parte, las aplicaciones Web has aumentado su popularidad en los últimos años debido a lo práctico del navegador web como cliente ligero, a la independencia del sistema operativo, así como a la facilidad para actualizar y mantener aplicaciones web sin distribuir e instalar software a miles de usuarios potenciales. [3]
El software SILO es una aplicación compleja y cerrada. Una idea que interesa en un futuro es poder interactuar con la compleja base de datos que gestiona a través de paginas web, tal como lo haría una aplicación web, por las razones previamente explicadas.
Como antes también se ha detallado, una herramienta que genere las clases y plantillas de acceso correspondientes a las tablas de una base de datos sin tener en cuenta los metadatos de las mismas ni las relaciones entre ellas tendrá una serie de carencias respecto a clases y plantillas que si los tengan en cuenta.
Las siguientes secciones (1.2.1 y 1.2.2) detallan una serie de situaciones que se dan en una base de datos y que llevaron al interés de abordar el problema tratado en este proyecto.

1.2.1- Relación referencial entre tablas

La base de datos que gestiona la aplicación SILO es compleja y consta de tablas con una gran cantidad de columnas importadas de otras tablas (claves ajenas). Resulta interesante que al generar una clase correspondiente a una tabla, exista información a cerca del origen de los datos de las columnas que son claves importadas para poder obtener una información mas detallada de los datos guardados en dicha tabla.
Ademas se da el caso de que en la base de datos gestionada por SILO la mayoría de las tablas contienen una clave primaria en formato de código numérico o alfanumérico. Esto hace que cuando se muestra al usuario un dato de una tupla de una tabla con columnas que son claves importadas de otras tablas, el valor de dichas columnas no sea fácilmente identificable por un usuario si no sabe relacionar el código que identifica a la

tupla referenciada, cosa que puede ser muy complicado si las tablas tienen una gran cantidad de datos almacenados, como en el caso de la aplicación SILO.
El gráfico 1.1 muestra un ejemplo que detalla este problema para una base de datos de ejemplo.

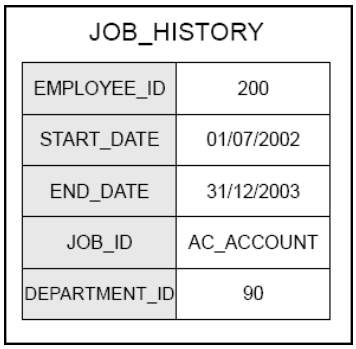


Gráfico 1.1- Tupla de una tabla sin conocer las referencias

El gráfico 1.1 corresponde a una tupla de la tabla JOB_HISTORY. La tabla corresponde al historial de trabajos de un trabajador. Tiene 5 columnas (Identificador del trabajador, fecha de inicio del trabajo, fecha de finalización del trabajo, identificador del trabajo y identificador del departamento para el que se realiza el trabajo). Sin mas información a cerca de que columnas contiene claves importadas y sin saber de que tabla se importan, los datos de la tupla son difícilmente interpretables por un usuario, ya que el dato de que el trabajador 200 realizo un trabajo de "AC_ACCOUNT" para el departamento 90 no aporta mucha información.

En el gráfico 1.2 vemos lo que sucede si tenemos la información que relaciona las columnas que son claves importadas con la tabla y clave de la que proceden.

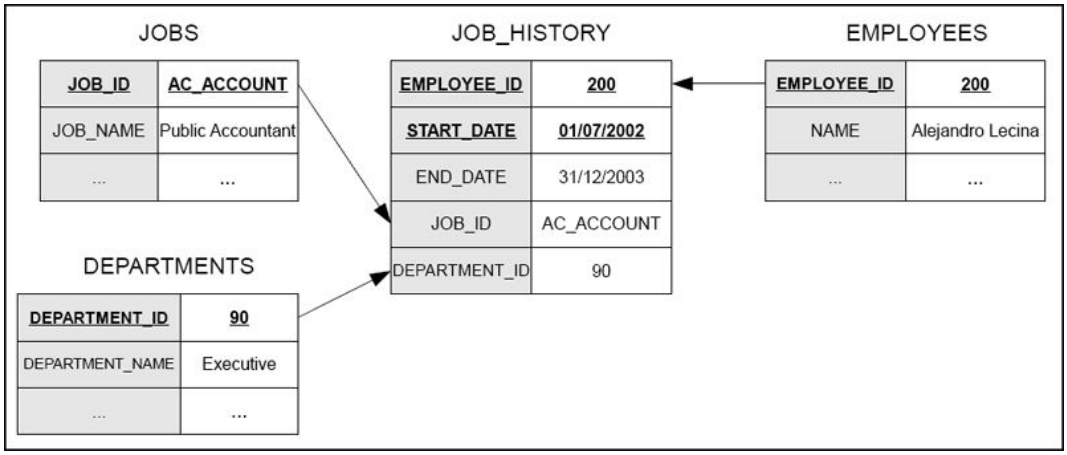


Gráfico 1.2- Tupla de una tabla de la que se conocen las referencias

Con toda la información a cerca de los metadatos y restricciones de la tabla JOB_HISTORY, podemos saber el origen de los datos de las columnas que son claves importadas (EMPLOYEE_ID, JOB_ID y DEPARTMENT_ID). Accediendo a la tupla correspondiente de las tablas referenciadas por las claves ajenas, que serán aquellas tuplas que tengan el valor de la clave primaria igual al valor de la clave importada de la tupla original, podemos acceder a datos adicionales que nos permiten interpretar con mayor facilidad la tupla mostrada, como se muestra en el gráfico 1.3

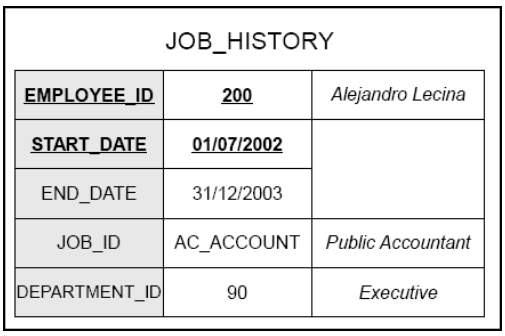


Gráfico 1.3- Tupla de una tabla añadiendo valores referenciados

Seria interesante que la aplicación que generase las clases y las plantillas de acceso a la base de datos fueran capaces de realizar esta función de añadir información a las columnas que sean claves importadas.

1.2.2- Claves "engordadas" y nivel de profundidad de una clave.

La base de datos que gestiona la aplicación ha de ser lo suficientemente genérica para adaptarse a el mayor numero posible de escenarios logísticos, que es para lo que esta pensada. Debido a esto y a la naturaleza de alguno de las tablas que se pueden encontrar en la base de datos, sucede el echo de que algunas tablas tienen a lo que en este proyecto se van a denominar "claves engordadas".
Llamaremos "*clave engordada"* a una clave primaria múltiple (formada por mas de una columna), en la que alguna de estas columnas es importada de otra tabla (Es decir, la clave primaria contiene una o mas claves importadas).
En el gráfico 1.4. vemos un ejemplo de la base de datos de la aplicación SILO de una clave "engordada".

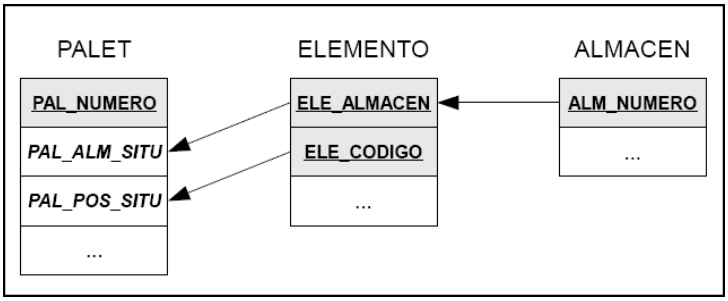


Gráfico 1.4- Ejemplo de clave "engordada"

En el gráfico 1.4 nos fijamos solo en las columnas que son clave primaria (con fondo gris) y en las columnas importadas. La tabla ELEMENTO contiene una clave primaria engordada, ya que la columna ELE_ALMACEN que forma parte de la clave primaria referencia a la clave primaria de la tabla ALMACEN. La tabla PALET importa esta clave engordada (Columnas PAL_ALM_SITU y PAL_POS_SITU). También puede darse el caso de que una clave engordada contenga una clave engordada y así sucesivamente. Si en este ejemplo las 3 columnas mostradas de la tabla PALET formaran la clave primaria (cosa que es perfectamente posible), seria una clave engordada que contendría la clave engordada de la tabla.
Durante el proyecto se hará referencia al termino *"nivel de profundidad de una clave"*. Este termino se aplica generalmente a las claves engordadas, ya que son las únicas que pueden tener un nivel de profundidad mayor que 1.
El nivel de profundidad de una clave es la distancia en numero de tablas de la que proviene la columna importada.
Volvemos a fijarnos en el gráfico 1.4. La columna PAL_POS_SITU de la tabla PALET tiene un nivel de profundidad 1 respecto a la columna ELE_CODIGO de la tabla ELEMENTO.
Por otro lado, la columna PAL_ALM_SITU de la tabla PALET respecto a la columna ELE_ALMACEN de la tabla ELEMENTO tiene nivel de profundidad 1, y respecto a la columna ALM_NUMERO de la tabla ALMACEN tiene nivel de profundidad 2, ya que la columna es importada a través de 2 tablas, la de origen, tabla ALMACEN y luego a través de la tabla ELEMENTO. Por lo tanto, también se dice que la columna

PAL_ALM_SITU de la tabla PALET tiene nivel máximo de profundidad 2.
A la hora de generar la solución para el problema planteado, habrá que tener mucha atención a estas claves "engordadas" y a las claves múltiples, ya que añaden dificultad al código a desarrollar para que la aplicación resultante pueda generar las clases a partir de cualquier base de datos con tablas con distintos tipos de estructuras
Se desea guardar esta información en las clases generadas para poder trazar con exactitud el origen de las columnas importadas.

1.3- Objetivos

El objetivo del proyecto es generar una aplicación web java que sea capaz de cubrir los tres siguientes objetivos:

1.3.1- Formalización de las tablas en ficheros XML

La aplicación web ha de ser capaz de conectarse a una base de datos y listar las tablas contenidas en ella para poder seleccionar aquellas de las que se desean generar las clases java y las plantillas de soporte para el accedo a los datos.
Si el proceso de generar las clases y las plantillas esta implementado dentro del código de la aplicación, cualquier modificación posterior a la hora de modificar la estructura de algún fichero de salida generado requeriría modificar el código. Esto siempre es complicado ya que exige conocer con exactitud el código de la aplicación.
Debido a esto, se generara un fichero XML por cada tabla deseada, con una estructura fija a definir, y que contenga toda la información a cerca de los metadatos y las restricciones (claves primarias, claves importadas y su nivel de profundidad) de dicha tabla.
A partir de estos XML se generaran los ficheros deseados.

1.3.2- Generación de clases java y formularios para el acceso a la base de datos

Usando los XML antes generados, y mediante transformaciones XSL [4], se generaran las clases java con la información de los metadatos y restricciones de las tablas, así como las plantillas que permitirán el acceso a las tablas.
Los ficheros de salida generados están pensados para ser usados por una aplicación web java. Debido a esto, las plantillas de soporte finales para el acceso a los datos serán paginas xhtml.
Por cada tabla deseada se generara una clase java, una pagina xhtml que nos permita visualizar los datos de la tabla y otra pagina xhtml que nos permite insertar y modificar datos. Las paginas xhtml usaran las clases java generadas para interactuar con la base de datos.

1.3.3- Obtener información adicional de las tablas referenciadas

En el caso de las paginas que listan los datos de una tabla, se creara una metodología que permita importar campos clave de las tuplas referenciadas por las columnas que son claves importadas, para así poder mostrar los datos de una manera mas fácilmente interpretable por un usuario, tal como se explico en el apartado 1.2.1 (*Relación referencial entre tablas*).

1.4- Productos obtenidos

Los productos obtenidos durante el desarrollo del proyecto son los siguientes:

1- Una aplicación Web llamada *xsltGenerator* que nos permite generar los siguientes ficheros de salida a partir de unas tablas seleccionadas en una base de datos:

- Ficheros XML con los metadatos de las tablas.
- Clase java con los metadatos.
- Xhtml para la inserción y modificación de datos.
- Xhtml para la visualización de datos importando datos adicionales par las columnas que son claves referenciadas.

2- Una aplicación Web llamada *xsltUse* pensada para usar los ficheros generados y crear una aplicación que los use para el propósito de los mismos. Incluyendo los ficheros generados de manera correcta, la aplicación permite insertar, modificar y listar los datos de las tablas de la base de datos de la que se han generado los ficheros.

1.5- Contenido de la memoria

Esta memoria contiene el trabajo realizado durante el proyecto. Esta información ha sido estructurada en 5 secciones principales.
La primera comenta algunos aspectos generales del proyecto, tales como los requisitos que la aplicación generada debe cubrir y las tecnologías usadas en el proyecto (Base de datos, entorno de desarrollo, frameworks, tipo de aplicación, transformaciones, etc)
Los puntos 3 y 4 detallan los métodos desarrollados en la aplicación *xsltGenerator*, que es la encargada de cumplir los objetivos del proyecto (Generación de clases java y formularios xhtml para el acceso a datos). También se describen las decisiones tomadas a la hora de implementar dichos métodos.
La sección 5 describe el modo en el que hay que usar los ficheros generados para que cumplan las funciones para las que han sido pensadas.
Finalmente, el punto 6 describe una valoración general del trabajo desarrollado, incluyendo posibilidades adicionales que ofrece y una valoración personal a cerca de las experiencias adquiridas durante el desarrollo del mismo.
Adicionalmente, se añade una indice que referencia paginas web consultadas para la elaboración del trabajo. Dichas referencias incluyen paginas web de las tecnologías usadas, tutoriales consultados, paginas usadas para resolver dudas concretas y paginas con información detallada a cerca de conceptos usados en la aplicación.
También se incluye un sección de Anexos, en la que principalmente se encuentra un manual de usuario de la aplicación final que genera los archivos (*xsltGenerator*) y el de una aplicación que usa los ficheros generados (*xsltUSe*). Estos manuales de usuario describen tanto la manera de usar las aplicaciones como los procesos internos que se dan en ellos, generalmente con mas detalle que los expuesto en la memoria.

2- Aspectos generales

2.1- Recopilación de requisitos del sistema

Durante distintas reuniones con el director del proyecto se fueron definiendo los requisitos que la aplicación final debía cumplir.
El gráfico 2.1 muestra la recopilación de estos requisitos. La columna “sección” referencia al apartado de la memoria donde se detalla la manera de cubrir el requisito.

Requisito	Descripción	Sección
RF 1	El programa debe dar la opción de elegir la base de datos a conectarse y el esquema o tablas a convertir a XML	2.3
RF 2	Formalización en XML de los meta datos de las tablas seleccionadas.	3
RNF 1	Meta datos a guardar de una tabla en los XML: tabla: (tabla + esquema + catalogo) columnas: (nombre + tipo + tamaño + posibilidad de campo nulo) restricciones: (clave primaria ={nombre de columna + nombre de la clave} claves ajenas ={nombre de columna + nombre de la clave + tabla referenciada + columna referenciada y sus datos + profundidad de la búsqueda})	3.1
RF 3	Tipo de aplicación: Aplicación web que usa el framework JSF 2.0.	2.2
RNF 2	Generar el código y la documentación en ingles.	*Código*
RF 4	Configurar la pantalla inicial para permitir usar un pool para la conexión a la base de datos.	2.3
RF 5	Una vez generados los XML, usar estos para generar un xhtml para la inserción de datos.	4.2
RF 6	Una vez generados los XML, usar estos para generar un xhtml para la actualización de datos.	4.2
RF 7	Una vez generados los XML, usar estos para generar un xhtml para mostrar los datos de las tablas.	4.3
RF 8	Una vez generados los XML, usar estos para generar las clases java necesarias para acceder a la base de datos a través de las paginas xhtml.	4.1
RF 9	Chequear la validez de los datos para poder insertar o modificar una fila en la base de datos a través de las paginas web.	4.1
RF 10	Gestionar todas las claves del programa en un archivo *.properties*.	2.4
RF 11	Permitir configurar el nivel de profundidad a investigar en la base de datos a la hora de obtener la información de las claves ajenas de las tablas.	3.2
RF 12	Usar un thread para generar el XML correspondiente a una tabla.	3.3
RF 13	Colocar los ficheros XML y xhtml de salida en una carpeta accesible para la aplicación para poder visualizarlos en tiempo de ejecución en el navegador.	2.5
RF 14	Usar transformaciones XSL para generar las clases java y las paginas xhtml a partir de los XML.	2.6

Gráfico 2.1- Requisitos del sistema

2.2- Entorno de desarrollo de la aplicación

Un requisito para la elaboración del proyecto es que la aplicación que generase las claves y los formularios fuera una aplicación web. Los ficheros generados también están pensados para ser usados por aplicaciones web.
Existen varios frameworks para el desarrollo de aplicaciones web en lenguaje Java.
Un framework para aplicaciones web es un paquete software diseñada para ayudar al desarrollo de webs dinámicas, aplicaciones web y servicios web. Un framework tiene como objetivo aliviar la sobrecarga asociada a las actividades comunes que se realizan en el desarrollo Web. [5]
En principio, el framework a usar constaba de tres opciones: GWT (Google Web Toolkit), Spring o JSF (Java Server Faces).
No tenia conocimiento de ninguno de los frameworks antes detallados. Después de investigar algo a cerca de ellos [6], y ayudado por las recomendaciones del director del proyecto, se decidió usar el framework JSF 2.0 (Java Server Faces, versión 2.0).
Se decidió usar este framework ya que era el que tenia una menor curva de aprendizaje que los otros dos. Ademas, es ideal para empezar a aprender conceptos del mundo de las aplicaciones web y viene integrado en el entorno de desarrollo NetBeans IDE 7.0.1, [7] que fue el usado para generar la aplicación web.
El primer paso antes de empezar a desarrollar la aplicación fue familiarizarse con el funcionamiento del framework JSF 2.0. El principal tutorial seguido fue el de la pagina de "*coreservlets*". [8]
El interfaz de la aplicación fue generado mediante paginas xhtml. JSF implementa las tareas de comunicación entre las paginas que forman el interfaz y las clases java que forman el modelo de la aplicación.

Se necesitaba una base de datos para realizar el proyecto. La elegida fue la base de datos gratuita de Oracle "*Oracle Database XE 11.2*". [9] Para realizar las pruebas a la hora de generar clases y formularios se uso las tablas del esquema "HR" que viene por defecto en la base de datos, ademas de un modelo simplificado de la base de datos usada por la aplicación SILO.
Para ejecutar una aplicación web se necesita un servidor de aplicaciones Java EE. El usado fue Glassfish Server 3.1 [10] debido a que viene integrado en el entorno de desarrollo NetBeans IDE 7.0.1.

2.3- conexión a la base de datos

El paquete *java.sql* [11] sera la API usada para conectarse y procesar los datos de las base de datos.
La aplicación *xsltGenerator* implementa dos maneras de conectarse a una base de datos: "*conexión manua*l" y conexión mediante un connection pool (agrupación de conexiones).

En la conexión manual se muestra un formulario (gráfico 2.2) con 4 campos de texto de entrada. Introduciendo los datos adecuados nos conectaremos a la base de datos deseada.

Gráfico 2.2- Conexión manual a la base de datos

La segunda opción para la conexión a la base de datos es la de usar 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.
El connection pool se configura en el servidor de aplicaciones. La manera de configurar un connection pool para el servidor de aplicaciones GlassFish esta detallada en el anexo "*Manual de usuario de la aplicación xsltGenerator*" *sección 2.1.1- conexión mediante un connection pool.*
El formulario en el que se inserta el nombre del pool de conexiones deseado para conectarse a una base de datos se muestra en el gráfico 2.3

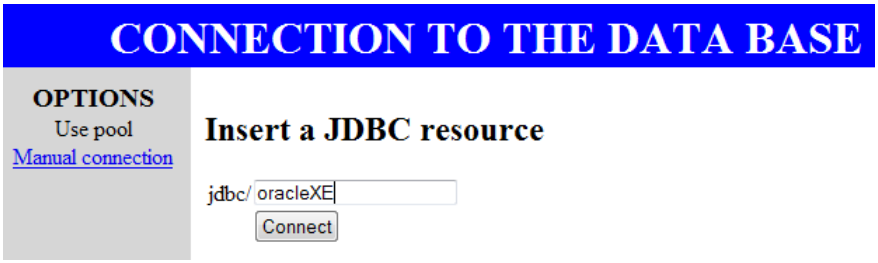


Gráfico 2.3- Uso de un connection pool

Una vez que el programa se conecte correctamente a la base de datos, llamara a un método que ejecutara la consulta SQL "*SELECT table_name FROM user_tables*" que devolverá todas las tablas de la base de datos. Estas se almacenaran en una lista. A través del interfaz de la aplicación se podrán seleccionar las tablas de la base de datos de las que se pretende generar un XML con sus metadatos.

2.4- Fichero *.properties*. Claves del sistema

Los ficheros *.properties* son archivos que nos permiten almacenar variables de configuración de nuestra aplicación. En la práctica, no deja de ser un fichero de texto donde almacenar por cada línea, un par clave valor, indicando el nombre de la variable y su valor. [12]

El fichero *myKeys.properties* situado en el paquete *properties* de la aplicación *xsltGenerator* contiene las claves del sistema.
Para obtener las cadenas definidas aquí se usa la clase *myProperties.java* situado en el mismo paquete.
En este fichero se establecen las claves que definen el nombre de las etiquetas y sus atributos que tendrán los xml generados, la ruta donde guardar los ficheros generados y la ruta de las plantillas xsl que se usaran para las transformaciones XSL.
Para información mas detallada del fichero *.properties* usado, mirar el anexo "*Manual de usuario de la aplicación xsltGenerator*" *sección 4.1- Fichero myKeys.properties.*

2.5- Repositorio para los ficheros generados

El repositorio de datos es el lugar donde se colocan los ficheros generados por la aplicación: ficheros xml, formularios xhtml y clases java.
Se quiere que los ficheros generados por la aplicación *xsltGenerator* se almacenen en un lugar accesible por la aplicación en tiempo de ejecución. El servidor de aplicaciones Glassfish no permite acceder a rutas locales por razones de seguridad. La única excepción a esto es la carpeta "*docroot*" situada en el dominio de ejecución de Glassfish.
Es en esta carpeta donde se guardaran los ficheros generados.
En el fichero *myKeys.properties* se definen las rutas donde se almacenaran los ficheros generados (carpetas dentro del directorio docroot del dominio de Glassfish).
En la carpeta docroot se generaran 5 carpetas (usando el fichero *myKeys.properties* por defecto) con los ficheros generados:

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 ficheros xml generados mediante transformaciones XSL a partir de los xml generados de las tablas de la base de datos.
simplejava_output: Contiene las clases java simplificadas 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.

Para información mas detallada a cerca del repositorio de datos, mirar el anexo "*Manual de usuario de la aplicación xsltGenerator*" *sección 2.3- repositorio de datos.*

2.6- Transformaciones XSL

Una vez que tengamos 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 (definidas en un fichero .xsl). 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 [13].

El esquema para la generación de los ficheros de salida se ilustra en el gráfico 2.4

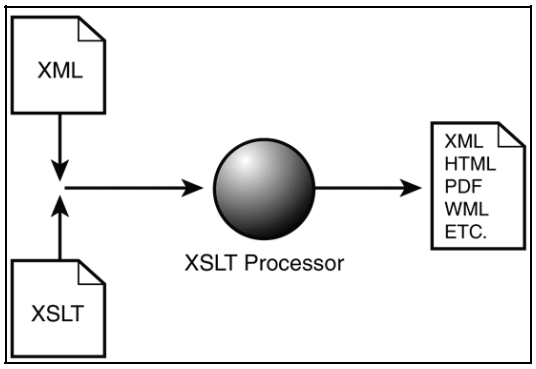


Gráfico 2.4- Transformaciones XSL

Existen numerosas paginas con información y tutoriales a acerca de las transformaciones XSLT. El mas utilizado fue el de w3schools.com [14], apoyado por la pagina oficial, de contenido mas extenso, de XSLT de la organización W3C [15].
Los ficheros .xsl son los que contienen las reglas de plantilla para transformar los ficheros XML.
Un fichero XSL consta principalmente de 3 elementos:

<xsl:stylesheet>: Este elemento especifica la versión XSLT y los espacios de nombres usados. Un espacio de nombres (namespace) es un identificador utilizado para distinguir entre los nombres de elementos XML y los nombres de los atributos usados en el fichero XSLT que pueden ser iguales. [16]
Los espacios de nombres usados en el fichero XSLT se definen en este elemento de la siguiente manera: xmlns:(prefijo de los elementos del namespace)="URI del namespace"

<xsl:output>: Define el formato del documento de salida según el valor del atributo "method". Este valor puede ser xml, html o text.

<xsl:template>: Este elemento define las plantillas. Estas plantillas se aplican a los elementos del xml de entrada que coincidan con el valor del atributo "match". Las acciones mas comunes que se pueden definir en

una plantilla son extraer valores de un nodo del xml de entrada, definir parámetros o variables locales, crear elementos con atributos en el fichero de salida, recorrer todos los nodos de un tipo y ejecutar sentencias condicionales sobre un nodo.

Principalmente, las transformaciones XSL permiten agregar / quitar elementos y atributos hacia o desde el archivo de salida. También puede reorganizar y ordenar elementos, realizar pruebas y tomar decisiones sobre cuáles son los elementos para ocultar y mostrar.

El entorno de desarrollo NetBeans incorpora un editor de ficheros XSL y un procesador XSLT accesible desde el interfaz. Basta con pulsar sobre un fichero XSL con el botón derecho y elegir la opción "transformación XSLT". Una pantalla como la mostrada en el gráfico 2.5 nos permitirá seleccionar el XML de origen y la salida.

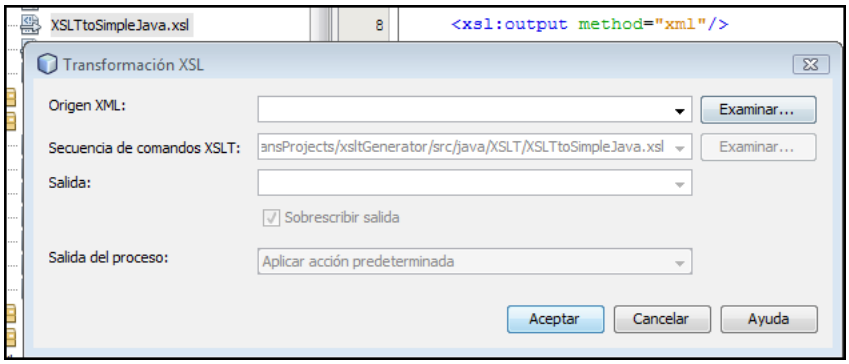


Grafico 2.5- Procesador XSLT integrado en NetBeans

Este procesador XSLT fue muy útil a la hora de realizar pruebas y lograr los resultados finales deseados. No obstante, se desea que las transformaciones sean producidas en tiempo de ejecución. El código java para aplicar una transformación XSL en tiempo de ejecución es el siguiente:

```
File fileOut = new File(<Ruta para fichero de salida> + <nombre del fichero de salida>);
Result result = new StreamResult(fileOut);
// 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(<Ruta del fichero de entrada> + <nombre del fichero XML de entrada>);
Source source = new StreamSource(fileIn);
// 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(<Ruta del fichero xsl con las plantillas XSLT>));
// 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.
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 de salida)
```

Para ver con detalle el código de un ejemplo concreto de como se aplica una transformación XSL en la aplicación *xsltGenerator*, mirar el anexo "*Manual de usuario de la aplicación xsltGenerator*" *sección 4.3-Transformaciones XSL.*

3- Formalización de las tablas en XML

El primer paso que hay que realizar con la aplicación *xsltGenerator* para poder generar las clases java y los formularios es generar las xml de las tablas seleccionadas para poderlos usarlos como ficheros de entrada para las transformaciones XSL

3.1- Estructura de los XML

Los metadatos de las tablas a almacenar en los XML fueron definidos como un requisito de la aplicación y son los siguientes:

Tabla: Nombre de la tabla + Esquema al que pertenece + Catalogo al que pertenece.
Columnas: Nombre + Tipo + Tamaño + Posibilidad de campo nulo (Nullable)
Clave primaria = Nombre de las columnas que la forman + Nombre de la clave.
Claves ajenas = Nombre de las columnas que la forman + Nombre de la clave + Tabla referenciada + Columna referenciada y sus datos + Nivel de profundidad de la búsqueda.

Al director del proyecto le interesaba el echo de que el XML generado tuviera una estructura que favoreciera comprender con facilidad la estructura de la tabla. Debido a esto, se separa la información de las columnas de la información de las claves ajenas y importadas.
Por cada columna se crea un elemento con la información de la columnas.
En el caso de la clave primaria, se crea un elemento que contiene el nombre de la clave y las columnas que la forman.
Por ultimo, para las claves importadas, se crea un elemento con el nombre de la clave y la tabla a la que referencia. Por cada columna que forma la clave, se señala que columna de la tabla forma parte de la clave importada y los valores de la columna y tabla a la que referencia por cada nivel de profundidad de la clave, hasta llegar al nivel de profundidad máxima.
Gracias a esto, podemos ver agrupada toda la información de las claves. Esto ayuda a comprender la estructura de la tabla, sobre todo para el caso de claves compuestas por mas de 1 columnas, ya que de una manera rápida podemos ver todas las columnas que forman parte de la clave.
La otra opción planteada fue incluir todos los datos, también los relativos a claves, en un elemento por columna. Este caso se desecho por lo antes mencionado, ya que había que comprobar todas las columnas y el nombre de la clave a la que pertenecen para deducir una clave compuesta.

Los elementos y sus atributos que podemos encontrar en los XML generados son los siguientes:

<TABLE>: Una etiqueta por tabla. Contiene la información de la tabla.

Atributos	Descripción
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.

Atributos	Descripción
name	Nombre de la clave primaria.

<FOREIGNKEY>: Contiene información a cerca de una clave importada.

Atributos	Descripción
name	Nombre de la clave ajena
referencedTable	Tabla de la que se importa la clave ajena.

<COLUMN>: Información de las columnas de la tabla.

Atributos	Descripción
name	Nombre de la columna.
type	Tipo de la columna.
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*.
nullable	1 si la columna puede tener valor nulo, 0 si se requiere un valor.
columnName	Nombre de la columna. Este atributo aparece solo para el elemento COLUMN, ya que describir una columna que forma parte de una clave. La etiquete sera un elemento hijo del elemento que define la clave a la que pertenece.

<**FKMETADATA>:** Por cada nivel de búsqueda en profundidad de una columna que es clave importada, almacena los valores de la columna a la que referencia la clave importada.

Atributos	Descripción
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 gráfico 3.1 muestra la jerarquía de los elementos generados en los xml.

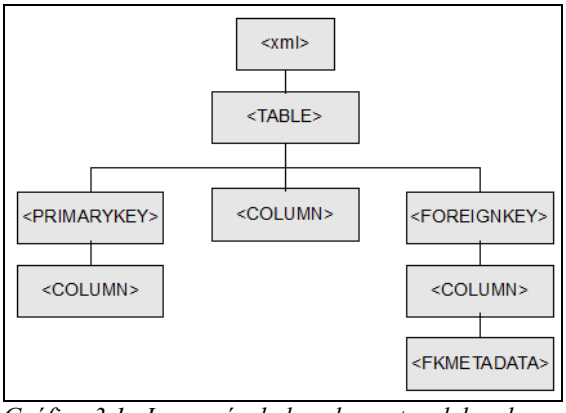


Gráfico 3.1- Jerarquía de los elementos del xml

En el anexo "*Manual de usuario de la aplicación xsltGenerator*" *sección 3.1- xml generados"* podemos ver un ejemplo de un xml generado para una tabla descrita en SQL.

3.2- Generación de los XML

Los xml generados tendrán el mismo nombre que la tabla a partir de las que se generan.
El primer paso es el de extraer de las tablas de la base de datos seleccionadas los metadatos que nos interesan. Se uso la API java.sql para esta labor.
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.

DatabaseMetaData getMetaData() throws SQLException

Este método esta definido para el interface *Connenction*. Devuelve un objeto del tipo "*DatabaseMetaData*" que contiene los meta datos 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"[17].

ResultSet **getColumns**(String catalog, String schema, String table, String column) throws SQLException

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) throws SQLException

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) throws SQLException

Este ultimo método nos devuelve la descripción de las columnas que forman parte de la clave primaria de las tablas referenciadas por las claves importadas de la tabla.

El método para implementar la búsqueda en profundidad de las claves importadas es el siguiente:
Cuando se obtiene una columna que forma parte de una clave importada, se invoca al siguiente método recursivo, definido en la aplicación *xsltGenerator*:

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

A dicho método hay que pasarle la tabla referenciada por la clave importada y la columna de dicha tabla que es importada (la cual formara parte de la clave primaria de la tabla referenciada).
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 de otra tabla, es decir, si la columna forma parte de la clave primaria y también de una clave ajena. Si no esa sí, 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 profundidad igual al anterior

La estructura que guarda todos estos meta datos es una lista enlazada de cadenas. Esta lista se pasara al constructor de una clase de la aplicación *xsltGenerator* para que genere el XML.
Con esta lista, y con la ayuda de la API DOM para Java (paquete *org.w3c.dom.**) se irán generando los elementos, añadiendo los atributos a los mismos y añadiéndolos al objeto XML de la manera correcta.
DOM (Document Object Model) es una plataforma que permite modelar objetos XML. Expone un documento XML como una estructura de árbol compuesta por nodos. El DOM permite navegar por el árbol de la programación y añadir, cambiar y borrar alguno de sus elementos.
Los estándares de programación de la interfaz DOM son definidos por el W3C. [18]

3.3- Eficiencia al generar los XML

Durante las pruebas realizadas sobre la base de datos instalada para generar los xml de las tablas, se observo que el proceso era algo lento (4 minutos 18 segundo para un esquema de 13 tablas). Después de investigar el código y consultar información, se encontró que el problema radica en el uso del método ***getImportedKey()***, ya que implica complejas condiciones de unión sobre tablas a nivel del sistema , y solo se recomienda su uso si es imprescindible. [19]
En nuestro caso es imprescindible usar este método, ya que es el único que ademas de devolvernos las columnas que forman parte de una clave importada de una tabla, también nos devuelve toda la información de las columnas originales que son importadas.
Originalmente se llamaba al método que extraía los metadatos de la base de datos (y que usa el método ***getImportedKeys***) de manera secuencial para todas las tablas elegidas. Se opto por mejorar la eficiencia del código llamando al método que extrae los metadatos paralelamente mediante hilos de ejecución. De esta manera se mejoraba el rendimiento a la hora de generar los xml, tal como se muestra en el gráfico 3.2.

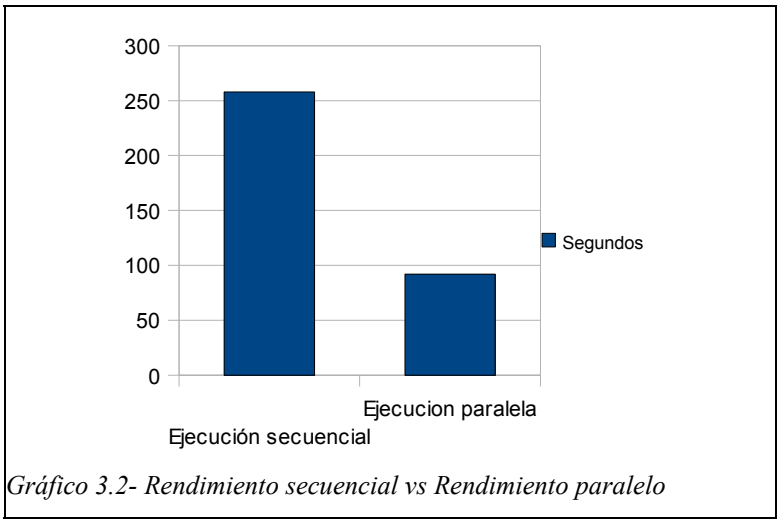


Gráfico 3.2- Rendimiento secuencial vs Rendimiento paralelo

El gráfico 3.2 muestra el rendimiento original para la ejecución secuencial que extraía los metadatos de una base de datos de 13 tablas y generaba los xml respecto al rendimiento para la misma base de datos si se invoca a un hilo que extraiga los datos y genere los xml.
El equipo en el que se desarrollaron estas pruebas fue un Intel core 2 duo, T5200, 1,60 Ghz, 2GB RAM y la base de datos Oracle Database XE 11.2.
Se observa que usando hilos se logra un incremento en un 180% aproximadamente para este caso.

3.4- XML Normalizados

Los xml generados a partir de los metadatos de las tablas debía tener una estructura con una serie de detalles que eran requisitos para la aplicación, tales como el echo de que fueran fácilmente interpretables por un usuario a simple vista.
A lo hora de generar las plantillas para las transformaciones XSLT para obtener los ficheros deseados, se observo que la estructura de los XML no era la mas adecuada debido a que separaba los metadatos de las columnas de las restricciones (claves primarias y importadas).
Debido a esto, se introdujo el concepto de "XML Normalizados". Estos XML normalizados son ficheros xml con los metadatos de las tablas que nos interesan pero agrupados todos por columnas. Es decir, a un elemento que representa una columna de una tabla se le añaden los datos necesarios si la columna forma parte de una clave primaria o una clave importada.

Estos XML normalizados se generan a partir de los XML generados y usando la plantilla XSLT "*NormalizeXML.xsl"*. Por tanto, serán un paso intermedio para las transformaciones XSLT entre los XML originales y los ficheros de salida.

Ademas, el xml original contiene la información que nos permite trazar una búsqueda para las columnas que son claves importadas. En los XML normalizados almacenaremos esta información con el fin de tener una estructura que nos ayude a generar automáticamente las consultas necesarias para importar datos adicionales de las columnas que son importadas para mostrarlos en los formularios para la visualización de los datos de una tabla.

Las etiquetas que encontramos en los XML normalizados son las siguientes:

Etiqueta	Contenido
<TABLE>	Metadatos de la tabla y la información para consultas de claves importadas.
<NAME>	Nombre del elemento padre.
<COLUMN>	Define los metadatos de una columna.
<TYPE>	Tipo de una columna.
<SIZE>	Tamaño del tipo de una columna.
<SIZEDEC>	Decimales para el tipo NUMBER.
<NULLABLE>	1 si no puede ser nulo, 0 si lo puede ser.
<PK>	Si la columna forma parte de la clave primaria, nombre de la clave.
<FK>	Si la columna forma parte de una clave importada, información de la clave.
<FKNAME>	Nombre de la clave importada a la que pertenece la columna.
<REFERENCES>	Tabla a la que referencia una columna.
<FINALREFERENCES>	Tabla a la que referencia una columna para el nivel máximo de profundidad.
<FINALCOLREFERENCES>	Columna de la tabla a la que referencia una columna para el nivel máximo de profundidad.
<FKNUM>	Usada para ordenar correctamente las columnas agregadas a los xhtml
<FKSELECT>	Información para la búsqueda en profundidad de claves importadas.
<FKSELECTWHERE>	Información a cerca de en que tablas y columnas estas relacionadas.
<FKCOLUMN>	Columna de la tabla que es clave importada.
<TABLEREFERENCED>	Tabla referenciada por la clave.
<PKCOLUMN>	Columna de la tabla referenciada que es exportada (clave primaria).
<DEPTH>	Nivel de la búsqueda en profundidad

El gráfico 3.3 muestra la jerarquía de los elementos de los xml normalizados.

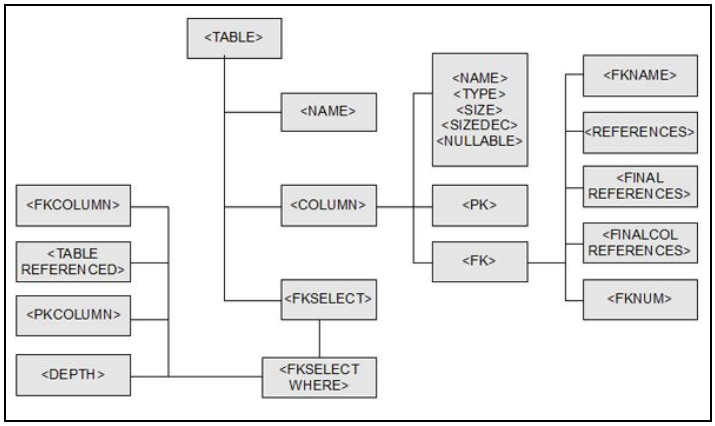


Gráfico 3.3- Jerarquía del XML normalizado

Resumiendo, los xml generados directamente de la base de datos están pensados desde el punto de vista del usuario, y los xml normalizados desde el punto de vista del código. Estos xml normalizados serán los que se usen de entrada junto a las plantillas XSLT para generar las clases java y los formularios.
La aplicación también guarda en el repositorio de datos estos xml normalizados.

Para mas detalles a cerca de la transformación XSL aplicada para generar estos XML normalizados, mirar el anexo "*Manual de usuario de la aplicación xsltGenerator" sección 4.3.1- Normalizar los xml.*

4- Generación de clases java y formularios

Para generar las clases java y los formularios xhtml es preciso haber generado previamente los xml con los metadatos de las tablas.
Cuando se elige la opción de generar algún tipo de estos archivos, se cargan los nombres de los xml generados y almacenados en el repositorio de datos de la aplicación *xsltGenerator*. De esta manera solo se permite generar uno de estos ficheros si el xml correspondiente a la tabla a sido previamente generado.
Como se ha explicado previamente, antes de generar la clase o formulario, se generara el xml normalizado a partir del xml como paso intermedio, y se usara el xml normalizado para generar el fichero deseado. Ambos serán generados aplicando transformaciones XSL.
Las plantillas XSLT usadas para la generación de archivos se pueden encontrar en el anexo "*Plantillas para las transformaciones XSL"*.

4.1- Generar las clases 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.

Las clases generadas tendrán el nombre de la tabla y los siguientes atributos y métodos:

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();

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.
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 "searchRow()" 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.
Para mas detalles a cerca de la transformación XSL aplicada para generar estos XML normalizados, mirar el anexo "*Manual de usuario de la aplicación xsltGenerator" sección 4.3.2- Generar Javas.*

4.1.1- Clases auxiliares para las clases Java generadas.

Column.java:

Column
private String columnName; private String value; private String type; private int size; private int sizeDec; private int inputSize; private String typeOutput; private boolean nullable; private String pk; private String fk; private String references; private String finalReferences;
boolean validateInput();

Esta clase tiene como atributos los posibles metadatos 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. 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 metadatos de las columnas de una tabla como una lista enlazada de instancias de la clase "*column*".

FunctionInsertRow.java:

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

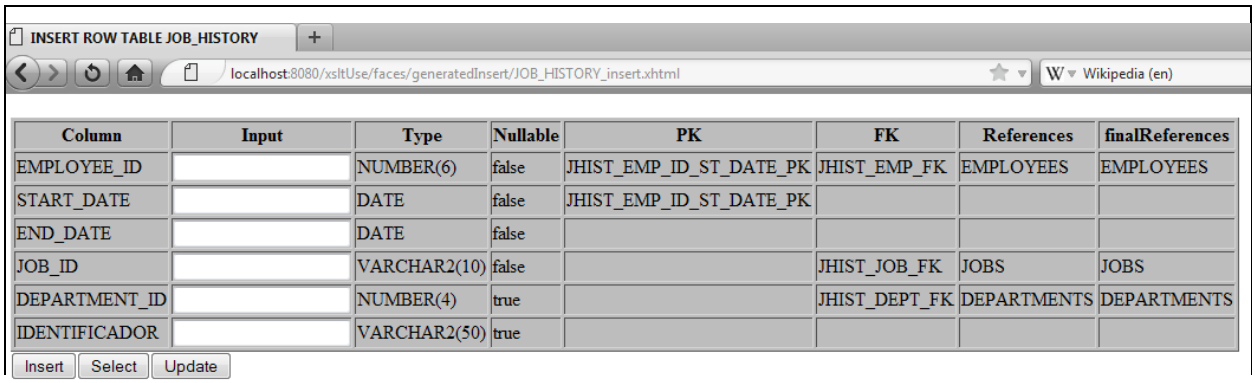
Esta clase es la que nos permitirá construir la sentencia SQL necesaria para introducir una tupla en una tabla de la base de datos, invocándola desde la clase java generada y pasándole como parámetros la lista con la información de las columnas y la conexión a la base de datos.

Si nos fijamos es la plantilla XSLT que genera las clases Java (*XSLTtoJava.xsl*), el punto mas interesante se centra en generar el constructor de la clase. En el constructor es donde se añadirá la información de los meta datos de las columnas a la clase:

```
<!-- Constructor de la clase generada -->
public <xsl:value-of select="$table"/>(Connection connection) {
  this.listColumns = new LinkedList&lt;column&gt;();
  <!-- &lt; = carácter '<' -->
  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>
}
```

4.2- Generar los formularios para la inserción de datos

El gráfico 4.1 muestra el ejemplo de una pagina xhtml generada por la aplicación *xsltGenerator* para una tabla de una base de datos (En este ejemplo la tabla JOB_HISTORY).



Un formulario para la inserción de datos (botón *Insert*) también nos permite cargar una tupla de la tabla (botón *Select*) introduciendo los valores de los campos de la tupla que se quiere buscar, y actualizar una tupla (botón *Update*), que modificara los valores de la tupla que tenga los valores introducidos para las columnas que forman la clave primaria.

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" />
```

Se añaden los espacios de nombres xmlns:f y xmlns:h a la información de estilo del documento XSL. Representan elementos propios de las paginas xhtml usadas por el frameworj JSF.

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>
    <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>
```

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

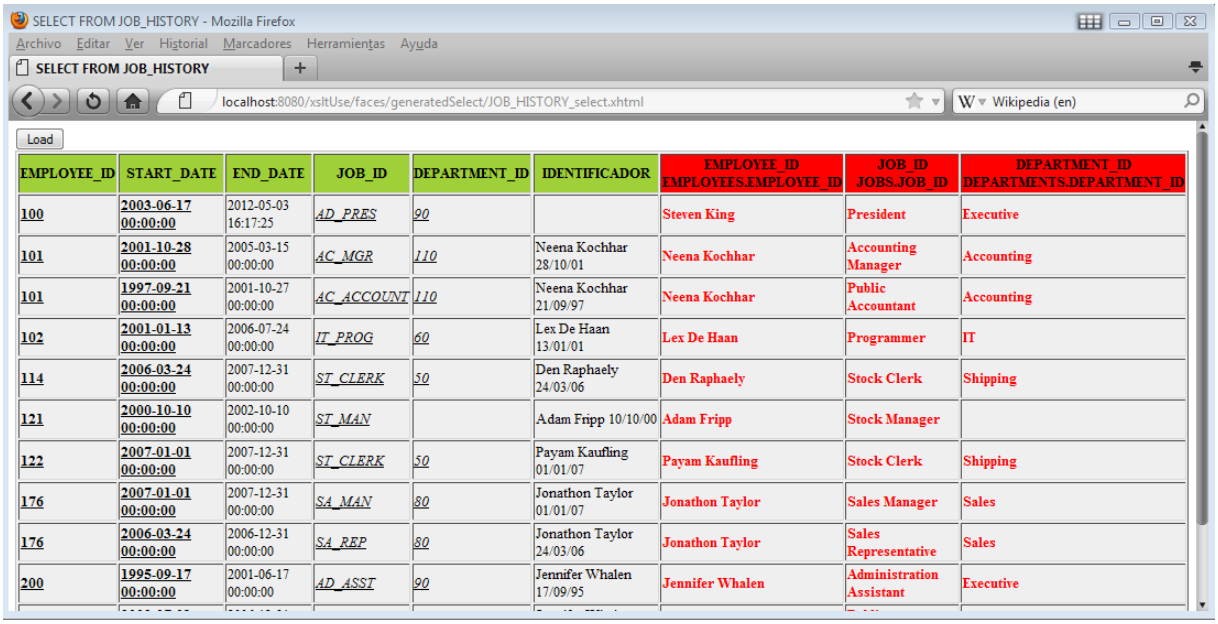
```
<h:column>
  <f:facet name="header"><xsl:text>Input</xsl:text></f:facet>
  <xsl:variable name="item">#{item.value}</xsl:variable>
  <xsl:variable name="size">#{item.inputSize}</xsl:variable>
  <h:inputText value="{$item}" maxlength="{$size}"/>
   <!-- caja de texto de entrada que admite un tamaño 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}"/>
```

4.3- Generar los formularios para la visualización de datos

El gráfico 4.2 muestra un formulario xhtml generado para la visualización de los datos de una tabla.



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

Gráfico 4.2- Formulario xhtml de visualización de datos de una tabla

Hay que pulsar el botón *Load* para cargar los datos correctamente, ya que si no se pueden mostrar datos de anteriores consultas a la base de datos.
Las columnas en verde representan los datos almacenados en la tabla. En negrita y subrayado se muestran los valores que perteneces a la clave primaria, y en cursiva y subrayado las columnas que son claves importadas.
Las columnas en color rojo corresponden a información adicional para las columnas correspondientes que son claves importadas.

La plantilla XSLT para la generación del formulario consta de los siguientes pasos:
El primer paso es generar una cadena en una estructura que sera interpretada por la aplicación, usando la información para consultas de los XML normalizados, y que servirá para construir la consulta que obtiene los datos, incluyendo las consultas adicionales que añaden información a cerca de las claves importadas.
A continuación se enlaza el botón *Load* con el método que ejecutara la consulta, la cadena generada con la información de la consulta se pasara como parámetro al invocar el método. El valor devuelto por la consulta se mostrara en una tabla de datos. A la tabla de datos se le añaden tantas columnas como columnas tenga la tabla mas el numero de columnas de la tabla que pertenecen a claves importadas.
Para información detallada de la plantilla XSLT que genera los formularios para la visualización de datos, mirar el anexo "*Manual de usuario de la aplicación xsltGenerator*" *sección 4.3.5- Generar formularios de visualización de datos.*

4.3.1- Clases auxiliares usadas por el formulario de visualización de datos

Las siguientes clases java serán usadas por el formulario de visualización de datos principalmente para generar y ejecutar las consultas y para almacenar los valores devueltos por las mismas.

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:

rowValues
private List<String> listData;
void addData(String s); String returnData(int i);

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 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 selectQuery); Private void generateQuery(); Private void generateFinalSelectQuery(); Private void executeSelect(); Private void executeComplexQuery();

La lista *RowValues* contiene 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* contiene la cadena con la información para generar la consulta y las subconsultas. Al llamar al constructor, se ejecutara el método "*restructureQuery()"*, que decodificara la cadena "selectQ*uery*" 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()"*.

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

El proceso para decodificar la cadena con la información para las consultas y formar la consulta y subconsultas que obtenga las tuplas de la tabla y la información adicional para las columnas que forman parte de las claves importadas es un proceso complejo, que esta definido con amplitud en el anexo "*Manual de usuario de la aplicación xsltGenerator*" *sección 4.4- La búsqueda en profundidad en la pagina de visualización de datos.*

En esta sección se repasaran los puntos mas importantes de este proceso, que lleva a cabo la clase *functionSelect*.

El proceso para añadir información a cerca de una columna que es clave importada es el siguiente:
La cadena con la información para la consulta almacena para una columna importada la tabla y la columna a la que referencia por cada nivel de profundidad. Con esta información, para una columna que es clave importada, se accede a la tabla referenciada y se busca la tupla que tiene un valor para la clave primaria igual al valor de la clave importada. Esta tupla contiene todos los valores a los que referencia la clave importada.

El problema que se encontró era qué información de esta tupla se podía importar para añadir información que fuera realmente fácil de identificar por un usuario.

La solución tomada fue la siguiente:
Para cada tupla referenciada se añadiría siempre el valor de una columna llamada "*identificador*", y que contendría una cadena con una descripción fácilmente interpretable por un usuario a cerca del contenido de la tupla.
Obviamente, las tablas de la base de datos no tienen por que tener esta columna, así que hay que crearla.
En un script SQL se modifican todas las tablas de la base de datos para añadir la columna "*identificador*" del tipo VARCHAR.
A continuación se estudian las tablas y su estructura para identificar que campos nos podrían dar una información fácil de identificar a cerca de una tupla contenida en la tabla.
Se añade al script SQL una sentencia que actualice la columna "*identificador*" de las tablas añadiéndole la información deseada que queremos mostrar en los formularios para la visualización de datos como información adicional para las claves importadas.
El nombre de la columna identificativa de la tupla (en nuestro caso "*identificador*") se encuentra definido en el fichero de claves *myKeys.properties*, de modo que si interesa usar otro nombre para esta columna basta con modificar el fichero de claves.
De este modo ya tendremos la base de datos preparada para ser usada por los ficheros generados por la aplicación.
En el anexo "*Manual de usuario de la aplicación xsltUse" sección 5.1- Campo identificador y preparación de la base de datos"* Podemos ver un ejemplo de como se preparo la base de datos usada para las pruebas durante el desarrollo del proyecto.

El otro punto a tener en cuenta a la hora de añadir información a las claves es la manera de estructuras las subconsultas para obtener los campos identificativos deseados.

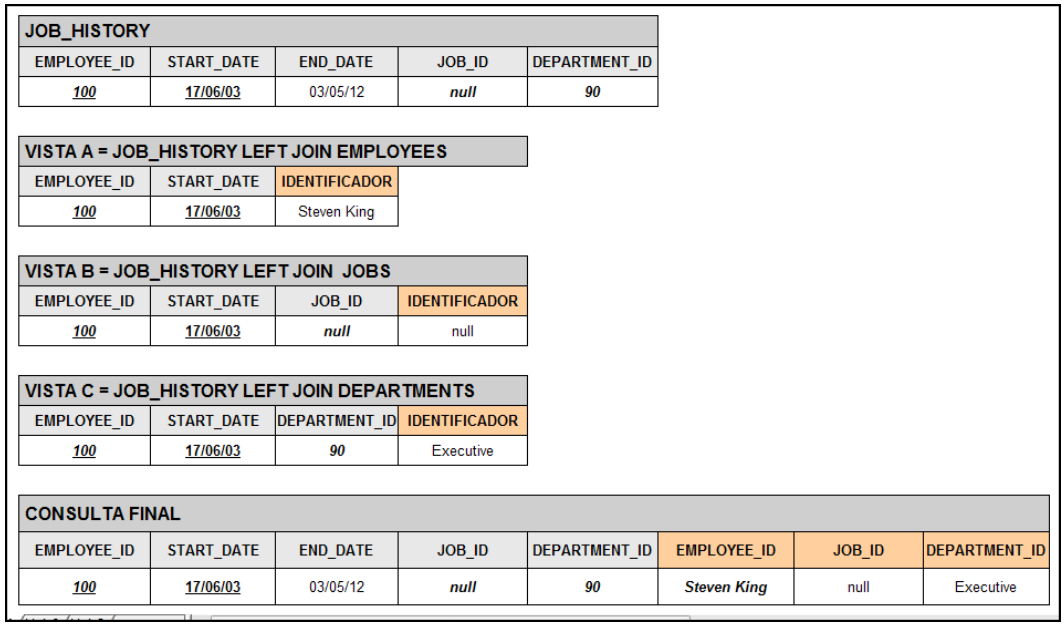
El primer problema a tratar era el caso de claves importadas compuestas (formadas por mas de una columna). En este caso las columnas que forman la clave importada apuntaran a las columnas de la clave primaria de la tabla a la que apuntan. Es imprescindible que para estas claves se busque la tupla referenciada que coincida con todos los valores de las columnas de la clave, ya que si solo coincide alguno, el valor devuelto no sera el deseado. Este problema se soluciono definiendo correctamente en los XML normalizados (que contienen la información que genera la cadena para la consulta) cuando una clave era compuesta y las columnas que la componían, de manera que la clase *funcionSelect* sera capaz de identificarlas y buscarlas de manera correcta.

La naturaleza de las consultas que obtenían la información de las claves referenciadas tuvo que ser modificada para arreglar unos problemas de las primeras versiones.

Originalmente se usaba un SELECT de todos los datos de la tabla mas la columna identificativa de las tablas referenciadas por las claves importadas usando una clausula AND entre las claves importadas de la tabla y la claves primarias de las tablas a las que referenciaban cada clave.
El problema de este método radica en que alguna columna que forma parte de una clave importada puede ser nula (columna con la restricción nullable). En este caso una de las clausulas AND no encuentra resultado, y por tanto devuelve un valor nulo para toda la consulta, por lo que la tupla de la tabla no se mostraba en el formulario.

Se opto por elaborar una solución mas compleja que resolviera este problema.
En primer lugar, por cada clave importada se genera una vista que obtendrá el valor identificativo de la tupla referenciada.
En cada vista se guardaran los valores de la clave primaria de la tabla original mas el valor identificador de la tupla apuntada por una clave importada. Para obtener este valor se usara una clausula LEFT JOIN entre la tabla y la tabla referenciada, de modo que, aunque el valor de la clave sea nulo, la vista obtenida almacenara la información de que para esa clave el valor es nulo, información que podrá ser usada para formar la consulta final.
En la consulta final se muestran todas las columnas de la tabla mas los valores importados, que se encuentran guardados en las vistas.

El gráfico 4.3 representa un ejemplo del proceso que se sigue para obtener información adicional de las claves importadas de la tabla JOB_HISTORY.



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

gráfico 4.3- Generación de la consulta con los datos adicionales para las claves importadas

Las columnas *EMPLOYEE_ID* y *START_DATE* forman la clave primaria de la tabla *JOB_HISTOTY.* Dicha tabla tiene 3 claves importadas: *EMPLOYEE_ID, JOB_ID* y *DEPARTMENT_ID.*
Las vista A, B y C se han generado haciendo un LEFT JOIN entre la tabla y la tabla referenciada por cada clave según el valor de la columna que forma parte de la clave. Observamos que estas vistas ya contienen la información identificativa para cada una de las claves importadas. Finalmente, obtenemos todos los datos de la tabla añadiéndole la información identificativa almacenada en las vistas.

5- Uso de las clases java y formularios generados

En la fase final del proyecto se desarrollo la aplicación *xsltUse*. Se trata de una aplicación web para el framework JSF que contiene un esqueleto que permite añadir directamente los ficheros generados para que cumplan el objetivo para los que has sido creados.
El anexo "*Manual de usuario de la aplicación xsltUSe*" contiene todos los detalles de como añadir los ficheros generados para poder usarlos.

Principalmente el proceso a seguir consta de 4 pasos y son el siguientes:

En primer lugar se han de añadir los formularios xhtml generados (Los de inserción y los de visualización de datos) dentro de la carpeta "Web Pages" de la aplicación. Estos formularios servirán como interfaz para interactuar con la base de datos.

En segundo lugar hay que añadir todas las clases auxiliares modeladas para usar por las clases java y los formularios. El esqueleto de la aplicación *xsltUse* ya las contiene.

A continuación se añaden las clases java generadas. Estas clases deberán ser instanciadas y inicializadas para poder ser usadas a través de los formularios xhtml.

Finalmente se adapto la base de datos de las tablas añadiéndole en campo identificativo y actualizándolo con los valores deseados usando un script SQL. También se añadió el fichero *myKeys.properties* con las claves necesarias: (El nombre a usar para las columnas identificativas y la ubicación de los formularios añadidos).

Siguiendo estos pasos, la aplicación *xsltUse* es capaz de interactuar con las tablas de la base de datos de prueba usada durante el desarrollo de la aplicación.

6- Conclusiones

6.1- Valoración del trabajo desarrollado

El objetivo del proyecto era el de generar una aplicación capaz de generar clases Java y formularios para el acceso a datos para poder ser usados por otras herramientas.
Como el trabajo desarrollado esta enfocado a una aplicación de ayuda al desarrollo de otras aplicaciones, era complicado hacer que los ficheros generados sirviesen para el amplio numero de aplicaciones Java que se pueden desarrollar.
Debido a esto, había que separar algunos procesos generales de otros que se enfocan a la tecnología usada en el proyecto (aplicación web java para el framework JSF).
Un paso critico a la hora de elaborar el programa era la generación de los XML con los metadatos. Con la ayuda del director del proyecto se definió una estructura que representase correctamente los metadatos de una tabla. Este paso es importante ya que es independiente del tipo de aplicación o del framework.
Las clases java generadas también son independientes al framework, y representan con exactitud los metadatos de la tabla. Para ello fue necesario crear la clase auxiliar *column.java,*ya que sin esta era difícil modelar con exactitud la tabla.
La principal ventaja de la aplicación radica en el uso de las transformaciones XSL que se usan para generar los formularios y los métodos de las clases java. Estos formularios están pensados para un framework en concreto, pero conociendo su funcionamiento y las plantilla XSLT que los generan, es fácil poder adaptarlas a otros tipos de frameworks o aplicaciones.
El proyecto también incluye lo que se podría denominar un "*generador de consultas SQL a partir de metadatos de una base de datos"* ya que las consultas SQL ejecutadas para obtener información adicional de las claves referenciadas se generan automáticamente. Esta función podría ser tenida en cuenta a la hora de sacar provecho al código de la aplicación o para poder tener la capacidad de generar consultas mas complicadas y personalizadas añadiendo el código y los procesos necesarios.
Los formularios xhtml están pensados para ser usadas junto a las clases para acceder a la base de datos sin tener que adaptarlas por un programador. No obstante, se generan con todas las columnas y su descripción para una tabla de la base de datos. A algún desarrollador le podría interesar separar la lógica con la que se conecta a la base de datos y quedarse con la estructura de la tabla de datos generada, para luego usarlos para los distintos intereses que desee. Aunque los formularios generados no tiene mucha información de estilo, se les puede añadir a través de hojas de estilo CSS para darles la apariencia deseada sin tener que tocar los elementos generados.

En resumen, el echo de usar transformaciones XSL hace que los ficheros generados se puedan adaptar a múltiples usos solo con modificar las plantillas XSL.

En cuanto a las posibilidades adicionales que se le pueden añadir al trabajo desarrollado, se podrían aumentar el numero de opciones disponibles para la interacción con la base de dato. Se podría añadir la opción para borrar un dato, o un formulario que permita editar los metadatos de una tabla que usara la información de las referencias de las claves para poder modificar los metadatos de una columna y que se modificaran también todas las columnas relacionadas con ella.
También se podrían generar formularios para modificar, añadir o borrar columnas de una tabla.
Usando toda la información referencial de las claves de las tablas de la base de datos almacenadas en los xml, se podría crear una aplicación que construyera un esquema general (gráficamente o en algún tipo de estructura) de las tablas de la base de datos y sus relaciones.

6.2- Valoración personal y experiencia adquirida

Las experiencias adquiridas durante el desarrollo del proyecto se pueden distinguir en tres puntos.

El primero es el echo de haber aprendido a utilizar tecnologías desconocidas antes de empezar el desarrollo del proyecto y muy útiles y usadas en el mundo real.
En primer lugar aprendí a desarrollar una aplicación web Java y a usar un framework (Java Server Facer) para el desarrollo de la aplicación. Ademas, el conocimiento del framework JSF servira para poder comprender mejor y aprender mas fácilmente otros framework pensados para el desarrollo de aplicaciones web.
También se aprendió a usar las transformaciones XSL, incluyendo espacios de nombres que definen elementos y funciones adicionales a los estándares de XSLT que ayudan a potenciar la funcionalidad de la tecnología.
Aunque ya tenia conocimientos sobre Java, bases de datos y entornos de programación. Pude profundizar en el conocimiento de los mismos, aprendiendo a usar mejor y mas eficientemente la documentación web oficial de Java y la de la API para el driver JDBC que gestiona la conexión de una base de datos de una aplicación Java. Se adquirieron a su vez nuevos conocimientos a la hora de usar el entorno de desarrollo NetBeans tales como el funcionamiento del servidor de aplicaciones GlassFish o el procesador XSLT que lleva integrado.

En segundo lugar, se aprendió a usar estas tecnologías para solucionar los problemas que planteaba el proyecto. Debido a que la estructura para las clases java y los formularios así como la manera de comunicarse entre ellas dependía de mi, tuve que tomar decisiones a la hora estructurar las clases y los métodos que lograrían que los ficheros generados consiguieran los objetivos planteados al inicio del proyecto. En resumen, la tarea no solo era desarrollar código, si no que lo mas difícil era determinar que necesidades tenia para así poder implementar el código que mejor las cubría.

Finalmente, tuve la oportunidad de interactuar con miembros de HP Zaragoza para tener una primera aproximación del mundo real en cuanto al trabajo se refiere. Me introdujeron en las tecnologías que se usan en el mundo real así como enseñarme las ventajas que estás ofrecen. La experiencia obtenida en las reuniones, tanto desde el punto de vista de la aplicación como en el personal, fueron de gran valor.

7- Referencias

1- Pagina web del Observatorio Tecnologico HP (http://bifi.es/observatorio/index.php)

2- Pagina del servicio SILO de HP (http://www8.hp.com/bo/es/services/services-detail.html?compURI=tcm:247-823415)

3- Wikipedia.es. Aplicaciones Web (http://es.wikipedia.org/wiki/Aplicación_web)

4- Wikipedia.en. XSLT (Extensible Stylesheet Language Transformations) (http://en.wikipedia.org/wiki/XSLT)

5- Wikipedia.en. Web application framework (http://en.wikipedia.org/wiki/Web_application_framework)

6- Wikipedia.en. Comparison of web application frameworks (http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#Java)

7- NetBeans IDE (http://netbeans.org/index.html)

8- Coreservlets.com. Tutorial JSF 2.0 (http://www.coreservlets.com/JSF-Tutorial/jsf2/)

9- Oracle Database Express Edition 11g Release 2 (http://www.oracle.com/technetwork/products/express-edition/downloads/index.html)

10- Pagina de Glassfish (http://glassfish.java.net/)

11- Documentación del paquete java.sql (Java Platform SE 6) (http://docs.oracle.com/javase/6/docs/api/java/sql/package-summary.html)

12- Java y los ficheros .properties (http://www.v3rgu1.com/blog/476/2011/programacion/java-y-los-ficheros-properties/)

13- Wikipedia.es. Extensible Stylesheet Language Transformations (http://es.wikipedia.org/wiki/Extensible_Stylesheet_Language_Transformations)

14- XSLT Tutorial de w3schools.com. (http://www.w3schools.com/xsl/)

15- W3C: XSL Transformations (XSLT) Version 1.0 (http://www.w3.org/TR/xslt)

16- Understanding XML Namespaces (http://www.xmlpdf.com/namespaces.html)

17- java.sql Interface DatabaseMetaData (http://docs.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html)

18- Java DOM Tutorial (http://www.roseindia.net/xml/dom/)

19- Consejos sobre el rendimiento del controlador JDBC (http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzaha/jdbcperf.htm)