

TEMA

Tema 4. Persistencia en BDOO – BDOR

Desarrollo de aplicaciones
multiplataforma

Acceso a datos

Autora: Silvia Macho



Tema 4: Persistencia en BDOO – BDOR

¿Qué aprenderás?

- Instalar y configurar el SGBD-OO Matisse.
- Crear una aplicación Java que utilice como persistencia una BBDD-OO.
- Instalar un SGBD-OR como DB4o.
- Crear una aplicación Java que utilice DB4o como objeto de persistencia.

¿Sabías que...?

- Matisse es un SGBD-OO libre y multiplatforma que proporciona una librería de clases para poder acceder a él desde una aplicación en Java.
- Trabajar con un lenguaje orientado a objetos como Java y un SGBD-OO como Matisse, hace que solo se trabaje con objetos, eliminando la necesidad de mapeos.
- DB4o no se puede considerar un SGBD como tal, ya que no cumple con criterios propios de cualquier SGBD como puede ser la concurrencia.
- DB4o es uno de los sistemas más fáciles de integrar en una aplicación Java como elemento de persistencia, con poco código se consigue.



4.1. Persistencia en BDOO

Las bases de datos relacionales, al inicio del año 2000 eran las más utilizadas en la mayoría de los ámbitos de aplicación. La estructura de datos básica que ofrece, la tabla relacional, es apropiada para muchas aplicaciones habituales.

Sin embargo, existen casos de uso en los que presenta serios inconvenientes prácticos, generalmente si se requiere gestionar datos muy complejos o no convencionales (imágenes, documentos...), para los que las estructuras relacionales resultan muy complejas e inefficientes.

Las bases de datos orientadas a objetos intentan dar respuesta a estos problemas, incorporando las siguientes características:

- Adoptan como modelo de datos el de los lenguajes orientados a objetos, permitiendo así el uso de estructuras de datos tan complejas como sea necesario y eliminando en gran medida las barreras entre el desarrollo de aplicaciones y la gestión de datos.
- Permiten la extensibilidad con nuevos tipos de datos complejos, permitiendo incorporar operaciones arbitrarias sobre ellos.

Estas características han motivado el desarrollo de numerosos sistemas orientados a objetos. Juntamente al avance vertiginoso de desarrollo de WebApps que requieren estructuras de datos muy flexibles.



4.2. Sistemas gestores de BBDD-OO

El hecho de que las BDOO y LPOO comparten el mismo modelo de datos permite desarrollar aplicaciones sobre BDOO de manera casi transparente.

Sin embargo, hay que tener en cuenta los siguientes factores:

- Desde el lenguaje de programación orientado a objetos es necesario indicar qué objetos se guardarán en la base de datos (serán persistentes) y cuáles no.
- Según los requisitos de la aplicación a desarrollar, se deberá utilizar un SGBDOO con determinadas características.

Todo SGBD-OO tiene que cumplir con las siguientes características:

1. **Objetos complejos:** 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 sólo 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 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. **Su DML debe ser completo.**
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 después de que la aplicación que los creo haya finalizado. El usuario no tiene que hacer ningún movimiento o copia de datos explícita para ello.
10. Debe ser capaz de manejar grandes BD: 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 convencionales.
13. Método de consulta sencillo: debe poseer un sistema de consulta ad-hoc de alto nivel, eficiente e independiente de la aplicación.

Los SGBD-OO nacen de la necesidad de proporcionar persistencia a las aplicaciones implementadas en lenguajes de programación OO. Hay varias alternativas para proporcionar dicha persistencia, pero la alternativa natural es utilizar un sistema gestor que permita conservar y explotar todas las posibilidades que la POO permite. Esta alternativa son los SGBD-OO.



4.2.1. ODMG y su modelo de datos

ODMG son las siglas de Object Data Management Group y es el consorcio de creadores de SGBD-OO. No es una asociación acreditada tipo ISO, pero tiene influencia a la hora de establecer los estándares relacionados con los SGBD-OO.

Entre muchas de sus especificaciones, el estándar ODMG define el modelo de objetos que debe ser soportado un todo SGBD-OO. Para crear este estándar, se basaron y crearon:

- El modelo de datos del OMG.
- Lenguaje ODL para especificar el lenguaje de la BBDD.
- Lenguaje OML para la manipulación de datos.
- Lenguaje OQL para crear consultas sobre la BBDD.

El modelo de datos ODMG permite que los diseños OO y las implementaciones utilizando lenguajes OO sean portables entre los diferentes sistemas. El modelo de datos dispone de unas primitivas de modelado.

Las primitivas fundamentales de una BBDD-OO son los objetos y los literales.

- Un objeto es una instancia de una clase que representa un concepto del mundo real. Los objetos necesitan un identificador único OID (Object Identifier).
- Un literal es un valor específico. Los literales no tienen identificadores. Un literal no tiene por que ser un único valor, puede ser un conjunto de valores relacionados entre si que se almacenan bajo el mismo nombre.



Los objetos se dividen en tipos, donde un tipo es como una clase en la POO. Los objetos del mismo tipo tienen el mismo comportamiento y tienen un rango de estados común:

- El comportamiento viene determinado por el conjunto de funciones que puede ejecutar el objeto, es decir, los métodos.
- El estado de los objetos viene establecido por los valores que tienen su conjunto de propiedades. Las propiedades pueden ser de dos tipos:
 - Atributos: los atributos tienen literales por valores y son accedidos por operaciones del tipo get/set.
 - Relaciones entre el objeto y uno o más objetos. Son propiedades que se definen entre tipos de objetos, no entre instancias. Estas relaciones pueden ser uno a uno, uno a muchos, 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 de las instancias del tipo y los métodos que implementan las operaciones definidas en la interfaz.



4.2.2. Lenguaje ODL

ODL son las siglas de Object Definition Language. ODL es un lenguaje de definición de la estructura de los sistemas ODMG, es decir, especifica los tipos de objetos que existirán en un sistema que cumple con la especificación ODMG. ODL es el lenguaje equivalente a DDL en SGBD relacionales.

ODL define los atributos, las relaciones entre objetos y la cabecera de las operaciones. También lo utilizamos para especificar la estructura e integridad de la BBDD. En una BBDD-OO, ODL define los objetos, métodos, jerarquías, herencias y el resto de elementos propios del mundo OO.

Una característica importante que debe tener un ODL, es proporcionar a la BBDD un sistema de tipos de datos parecidos a los utilizados por los lenguajes de programación OO. Estos tipos son:

- Tipos básicos: Int, Char, Boolean, Float, Short, Long ...
- Tipos estructurados o interfaces: son tipos complejos formados por un conjunto de tipos básicos.

ODL nos permite crear el esquema de una BBDD-OO siguiendo el estándar ODMG. Después de crear el esquema, podremos crear, modificar, eliminar y consultar objetos en esa BBDD-OO.



4.2.3. Lenguaje OML

OML son las siglas de Object Manipulation Language. Es un lenguaje que permite la modificación de objetos en una BBDD-OO. La particularidad de ODMG es que no define un lenguaje OML, deja esta tarea a los lenguajes de programación, es decir, son los propios lenguajes de programación OO los que tienen que proporcionar su propio OML. La razón por la que la ODMG no proporciona este tipo de lenguaje es para que una aplicación no trate de forma diferente los objetos guardados en la BBDD de los objetos creados en memoria.

4.2.4. Lenguaje OQL

OQL son las siglas de Object Query Language. Es un lenguaje parecido a SQL que nos permite realizar consultas sobre las BBDD-OO. Tanto ODL como OQL son lenguajes que surgieron cuando SQL estaba ya extendido de forma similar a un estándar, cosa que hizo que la definición de ODL y OQL siguieran las bases de SQL, para facilitarnos su aprendizaje.

En los sistemas relaciones en estándar es SQL, pero éste puede variar en función del SGBD, lo mismo ocurre con OQL. La ODMG define el estándar, pero dependiendo del SGBD-OO, éste puede variar.



4.3. Matisse

Los SGBD-OO no están muy extendidos en el mercado, pero al igual que ocurre con los SGBD relaciones, podemos encontrar tanto libres como propietarios. En este documento, el SGBD-OO que utilizaremos será Matisse.

Matisse es un SGBD-OO que respeta en gran medida el estándar ODMG, proporciona una API para acceder desde aplicaciones programadas en Java y es multiplataforma, así que es una buena opción para poner en práctica los lenguajes ODL, OML y OQL comentados anteriormente.

Matisse lo podemos descargar de forma gratuita desde la web de www.fresher.com.



4.3.1. ODL en Matisse

La mejor forma de mostrar el lenguaje ODL adaptado a Matisse es con un ejemplo. El siguiente ejemplo define dos tipos complejos llamados Album y Autor.

- Album tiene como atributos título de tipo básico String, año y número de pistas como tipo básico Integer.
- Autor: tiene como atributos nombre y nacionalidad de tipo simple String.
- Entre estos dos tipos complejos (Album y Autor) existe una relación de tipo Set: un Album es creado_por un conjunto de autores y un Autor crea un conjunto de álbumes.



```
interface Album{
    /* definición de atributos */
    attribute string titulo;
    attribute integer año;
    attribute integer num_pistas;

    /* definición de relaciones*/
    relationship Set<autor> creado_por inverse Autor::crea;
}

interface Autor{
    /* definición de atributos */
    attribute string nombre;
    attribute string nacionalidad;

    /* definición de relaciones*/
    relationship Set<Album> crea inverse Album::creado_por;
}
```

4.3.2. OQL en Matisse

Igual que en el apartado anterior, la mejor forma de ver OQL con Matisse es con un ejemplo. Seguiremos con el mismo ejemplo anterior de tipos Album y Autor.

1. Ejemplo OQL que devuelve el título de todos los álbumes cuyo número de pistas sea mayor de 10:

```
select a.titulo
from album a
where a.num_pistas > 10;
```

2. Ejemplo OQL que devuelve la nacionalidad (sin repetir) de los autores cuyo nombre empiece por “M”.

```
select distinct a.nacionalidad
from autor a
where nombre like 'M%';
```



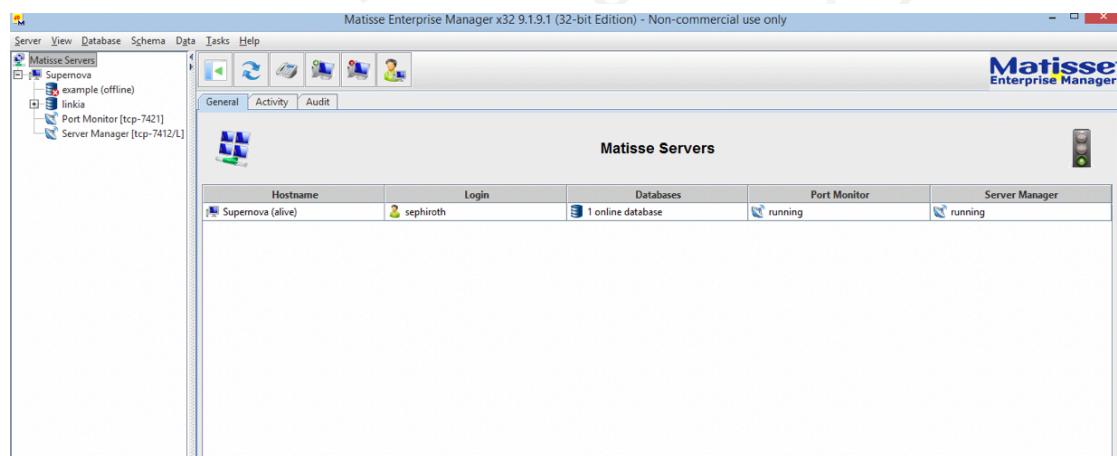
3. Ejemplo OQL que devuelve el título de los álbumes creados por todos los autores cuyo apellido empieza por “M”.

```
select al.titulo, au.nombre
from autor au, album al
where au.crea=al.OID and nombre like 'M%';
```

4.3.3. Instalación y configuración de Matisse

Matisse lo podemos descargar desde su página oficial www.fresher.com. Al ser multiplataforma, encontraremos la versión para nuestro sistema operativo. La instalación no tiene misterio, más que las particularidades propias del sistema operativo donde lo estemos instalando.

Al iniciar el sistema, nos aparecerá una pantalla como la siguiente:

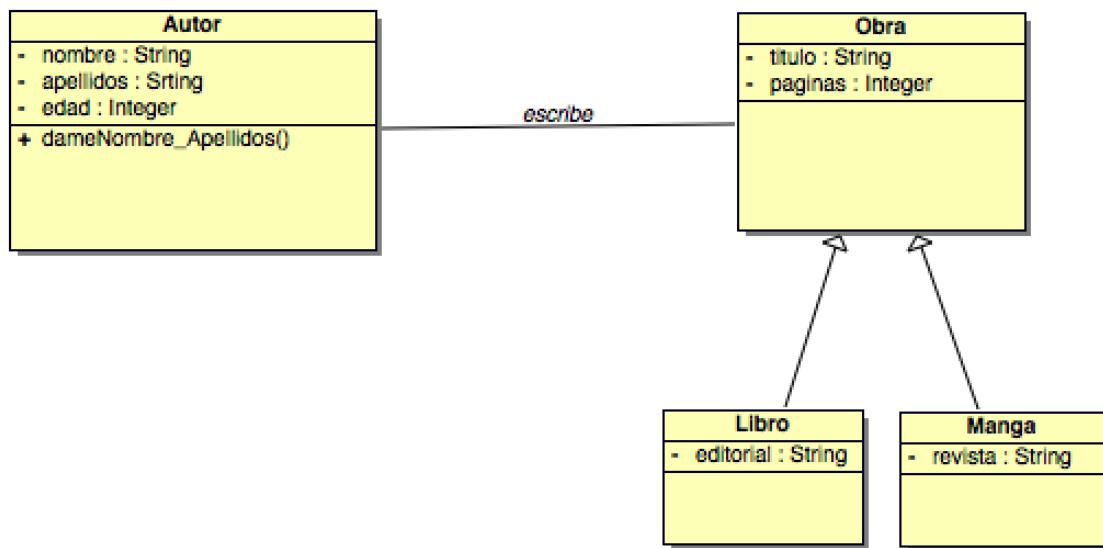


En esta ventana, tendremos dos partes bien diferenciadas:

- El panel de la izquierda nos muestra un listado con las BBDD existente en el sistema. Accediendo a cada una de ellas, se nos desplegará un listado con los diferentes elementos de las componen.
- En el panel de la derecha veremos el contenido de cada elemento que vayamos seleccionando en el panel de la izquierda. Además desde este panel podremos ejecutar código en lenguajes ODL y OQL sobre la BBDD seleccionada en el panel de la izquierda.

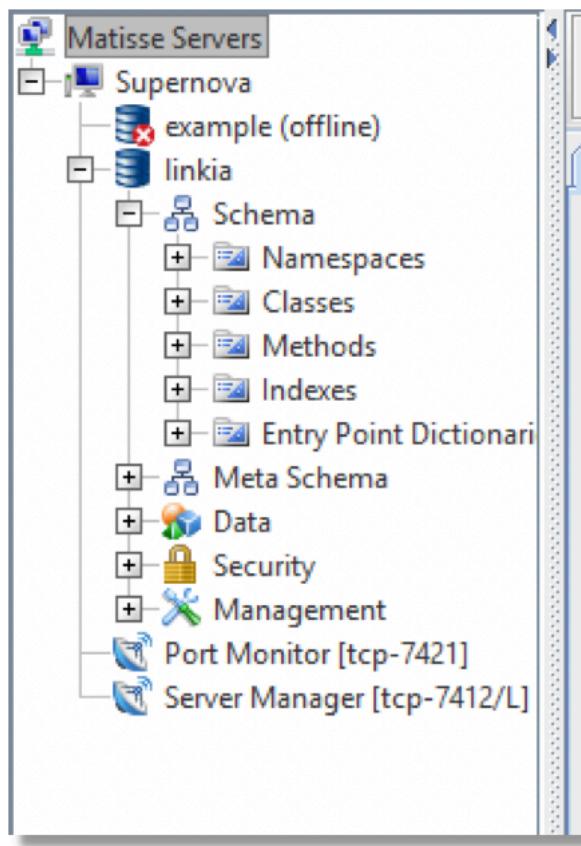


Para explicar la configuración de una BBDD en Matisse para dejarla preparada para su uso desde una aplicación en Java, seguiremos el siguiente ejemplo:



Para crear este esquema en Matisse, seguiremos los siguientes pasos:

1. Creamos una nueva BBDD. Para ello, en el panel de la izquierda, seleccionaremos la opción Server → New Database (sobre nuestro servidor). Aparecerá una ventana donde tenemos que poner el nombre que queremos que tenga la BBDD.
2. Cuando hayamos creado la BBDD, nos aparecerá en el panel de la izquierda. Como es una BBDD recién creada, inicialmente estará parada (veremos una x blanca con fondo rojo que nos lo indica). Para poder seguir con los siguientes pasos, tenemos que poner la BBDD en funcionamiento. Para ello, pulsaremos con el botón derecho sobre la BBDD y seleccionaremos la opción Start. La BBDD se iniciará y desaparecerá el símbolo que antes nos indicaba que estaba parada.
3. Como hemos iniciado la BBDD, ya podremos desplegar sus opciones para empezar a crear su esquema.



Como podemos ver en la imagen, este esquema está formado por varias secciones:

- Namespaces: es el espacio de nombres donde creamos las clases que representarán los diferentes objetos que guardaremos en al BBDD.
- Classes: son los elementos que nos van a permitir crear los objetos de los que guardaremos los datos en la BBDD.
- Methods: aquí tendremos los métodos de cada una de las clases creadas en el espacio de nombres. Como trabajamos con un SGBD-OO, guardaremos en la BBDD tanto los atributos como los métodos de los objetos.



4. El siguiente paso es crear un espacio de nombres. Para ello pulsaremos con el botón derecho sobre la opción Namespaces del panel de la izquierda y seleccionaremos la opción New Namespace. En la parte derecha, aparecerá una plantilla en lenguaje ODL para crear la sentencia que creará dicho Namespace. Lo que tendremos inicialmente es eso, una plantilla, por lo que tendremos que modificarla para crear exactamente la sentencia que queremos ejecutar.

La plantilla que vemos inicialmente es la siguiente:

```
-- Create a Namespace
--
-- When your namespace has been successfully created,
-- Click Refresh from the Namespaces Node to see
-- the newly created namespace.
--

CREATE NAMESPACE <mainmodule>.<submodule1>
```

En la imagen vemos un campo desplegable a la derecha. Este campo nos sirve para seleccionar dónde queremos crear el Namespace. Si es el primero que creamos, la opción que tendremos que dejar será root, que representa la raíz. Si más adelante queremos crear nuevos Namespaces, podremos crearlos también en la raíz o enlazarlos en forma de árbol seleccionando el Namespace padre en este campo desplegable.

Una vez modificamos la plantilla de ejemplo, nos quedaría el siguiente código:

```
-- Create a Namespace
--
-- When your namespace has been successfully created,
-- Click Refresh from the Namespaces Node to see
-- the newly created namespace.
--

CREATE NAMESPACE biblioteca
```



Para que el Namespace se cree, solo nos queda ejecutar la sentencia ODL que hemos creado. Para ello, tenemos que pulsar sobre el botón de 'Play' que tenemos arriba del todo del panel derecho:



Después de la ejecución, ya tendremos creado el nuevo Namespace.

5. Una vez creado el Namespace, ya podemos empezar a crear las clases que representarán el esquema de nuestra BBDD. Para ello, tenemos que pulsar con el botón derecho sobre la opción Classes y seleccionar la opción New Class. Igual que pasaba al crear un nuevo Namespace, de nuevo se cargará en el panel de la derecha una plantilla con lenguaje ODL para la creación de la nueva clase.

La plantilla inicial para la creación de una clase, nos ofrece todas las opciones para poder crearla, nosotros simplemente tenemos que dejar solo aquellas que necesitemos para la creación de la clase en cuestión.



La plantilla inicial que nos ofrece Matisse, es la siguiente:

The screenshot shows the 'Query Objects' interface in Matisse. At the top, there are four dropdown menus: 'In snapshot', 'Latest Version', 'In namespace', and 'Root'. The 'Root' menu is currently selected. Below these is a section titled 'SQL statement' containing the following ODL code template:

```
-- Create Class Specifying inheritance, relationships and constraints
-- Right click in the editor to select the possible attribute types
-- When your class has been successfully created,
-- Click Refresh from the Classes Node to see it in the tree.
-- 

CREATE CLASS <class>
[INHERIT <superclass> [, ...] ]
(
<attribute_1> <attribute_type> [DEFAULT literal] [NOT NULL],
<attribute_2> <attribute_type> [DEFAULT literal] [NOT NULL],
<relationship_1> [READONLY] RELATIONSHIP
    [LIST | SET] ( <succ_class> [, ...] )
    [CARDINALITY (<min>, <max>)]
    INVERSE <inv_class>.〈inverse_rshp>,
[CONSTRAINT <name>] PRIMARY KEY (<attribute> [, ...]),
[CONSTRAINT <name>] FOREIGN KEY (<attribute> [, ...])
REFERENCES <referenced_class> (<attribute> [, ...])
)
```

Lo primero que tenemos que modificar a la hora de crear una clase es el Namespace. Por defecto, la plantilla tendrá seleccionada la opción de Root, pero en ese campo desplegable podemos seleccionar el Namespace que corresponda. Lo siguiente es modificar la plantilla para crear la sentencia que creará la nueva clase tal y como nos interesa.

Siguiendo el ejemplo del esquema anterior, para crear la clase Autor, la sentencia ODL a ejecutar quedaría de la siguiente forma:

The screenshot shows the 'Query Objects' interface in Matisse with the 'namespace' dropdown set to 'biblioteca'. The 'Root' dropdown is still selected. The 'SQL statement' section contains the modified ODL code:

```
-- Create Class Specifying inheritance, relationships and constraints
-- Right click in the editor to select the possible attribute types
-- 
-- When your class has been successfully created,
-- Click Refresh from the Classes Node to see it in the tree.
-- 

CREATE CLASS Autor
(
    nombre String,
    apellidos String,
    edad String
)
```



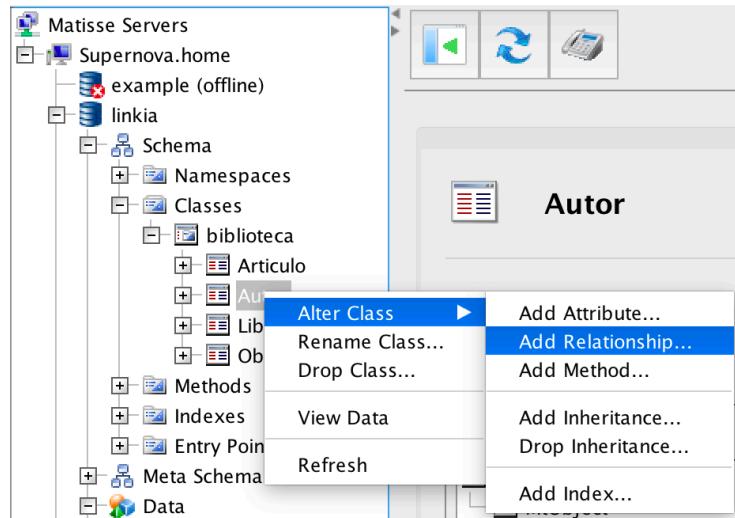
Ahora ya podemos ejecutar la sentencia para crear la clase. Al igual que con el Namespace, para ejecutarla, pulsaremos sobre el botón play que tenemos en la parte superior del panel derecho.

En el caso que alguna sentencia ODL diese un error, la sentencia no se ejecutaría y veríamos los errores en un nuevo panel que aparecerá en la parte inferior del panel derecho. En este panel, veremos la información sobre el error producido.

Para crear el resto de clases necesarias, solo tenemos que repetir este punto tantas veces como clases queramos crear en nuestro esquema. En el caso del ejemplo que estamos siguiendo, el código sería el siguiente:

```
CREATE CLASS Obra {  
    titulo String,  
    paginas integer  
}  
  
CREATE CLASS Libro INHERIT Obra {  
    editorial String  
}  
  
CREATE CLASS Manga INHERIT Obra {  
    revista String  
}
```

6. Ahora que ya tenemos las clases, los siguientes que tenemos que crear son las relaciones entre estas clases. Para ello, pulsaremos con el botón derecho sobre la clase a la que queremos añadir la relación y seleccionaremos la opción 'Alter Class'.



Nuevamente, lo primero que veremos será la plantilla para la creación de relaciones entre clases.

```
Query Objects
In snapshot Latest Version In namespace Root
SQL statement
-- Update Class: adding relationships
--
-- When your class has been successfully updated,
-- Click Refresh from the Class Node to see your newly created
-- relationships.
--

ALTER CLASS "biblioteca"."Autor"
ADD RELATIONSHIP <relationship>
[[READONLY] RELATIONSHIP [LIST | SET]] ( <succ_class> [, ...] )
[CARDINALITY (<min>, <max>)]
INVERSE <inv_class>.<inverse_rshp>
```

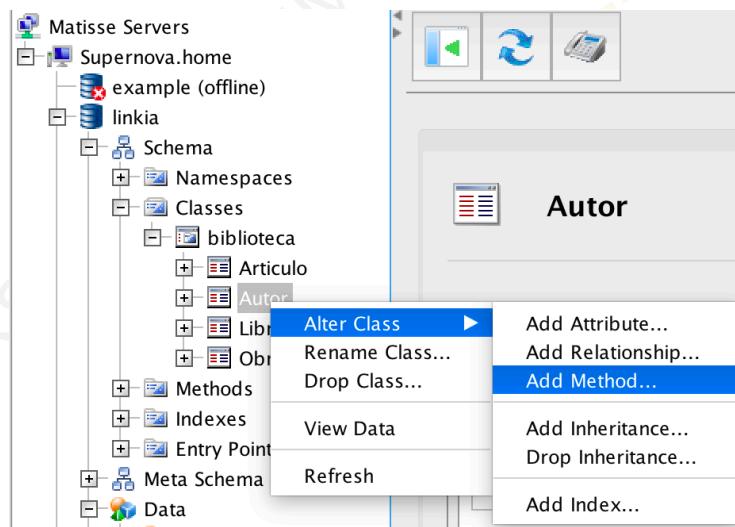
Modificaremos lo necesario para tener la sentencia ODL que necesitamos para crear la relación que queremos entre las clases. Esta relación la tenemos que crear en los dos sentidos, ya que son dos clases las que están involucradas en la relación.



En Matisse y siguiendo con el ejemplo, la relación entre las clases Autor y Obra la crearíamos con las siguientes sentencias:

```
--  
-- Update Class: adding relationships  
--  
-- When your class has been successfully updated,  
-- Click Refresh from the Class Node to see your newly created  
-- relationships.  
--  
  
ALTER CLASS Autor  
ADD RELATIONSHIP escribe  
RELATIONSHIP SET (Obra)  
INVERSE Obra.escribo_por;  
  
--  
-- Update Class: adding relationships  
--  
-- When your class has been successfully updated,  
-- Click Refresh from the Class Node to see your newly created  
-- relationships.  
--  
  
ALTER CLASS Obra  
ADD RELATIONSHIP escrito_por  
RELATIONSHIP SET (Autor)  
INVERSE Autor.escribe;
```

7. Lo último que nos queda son los métodos de las clases. Para añadir métodos a una clase, tenemos que pulsar con el botón derecho sobre la clase en la que queremos añadir el método y seleccionar la opción 'Add Method'.





Lo primero que veremos en el panel derecho es la plantilla para la creación de métodos:

```
Query Objects
In snapshot Latest Version In namespace Root
SQL statement
CREATE [INSTANCE | STATIC] METHOD <method name>
( <parameter declaration> [, ... ] )
RETURNS <data type>
FOR "biblioteca"."Autor"
<-- 
-- Describe your method here
<--
BEGIN
<statement>;
[ ... ]
END;
```

Ahora simplemente nos queda modificar la plantilla de forma correcta para tener la sentencia ODL que cree el método que necesitamos en la clase.

Para nuestro ejemplo, la sentencia en Matisse quedaría de la siguiente forma:

```
CREATE METHOD darNombreyApellidos ()
RETURNS String
FOR Autor
<--
-- Describe your method here
<--
BEGIN
    RETURN concat (nombre, apellido);
END;
```

En este punto, ya tendremos creado el esquema que representa nuestra BBDD-OO. No olvidar en cada uno de los pasos que hemos comentado, seleccionar el Namespace donde queremos ir creando los elementos.



4.4. Acceso a Matisse desde una aplicación Java

En este punto, ya tenemos creado el esquema en el SGDB-OO, ahora tenemos que preparar el código para que una aplicación Java pueda utilizar la BBDD para hacer persistentes sus objetos. Como hemos comentado en apartados anteriores, ODMG no define un lenguaje OML, es decir, un lenguaje que permita manipular los datos, deja esa responsabilidad a los SGBD-OO. En este caso el sistema gestor, Matisse, tampoco define un lenguaje OML, pasando esa responsabilidad a la aplicación, por lo que toda la manipulación de datos la tendremos que hacer desde Java.

No tenemos que olvidar, que el objetivo de un SGBD-OO es permitir a la aplicación tratar con todos los objetos de igual forma, ya estén almacenados en la BBDD o en memoria, dándole la sensación de que trabaja con objetos, sin que tenga que preocuparse por nada más. Este objetivo hace que al trabajar con un SGBD de este tipo, no utilizaremos API del estilo JDBC, todo lo contrario, la idea es que sea el SGBD-OO quien genere las clases en el lenguaje correspondiente a la aplicación, basándose en el esquema que contiene.

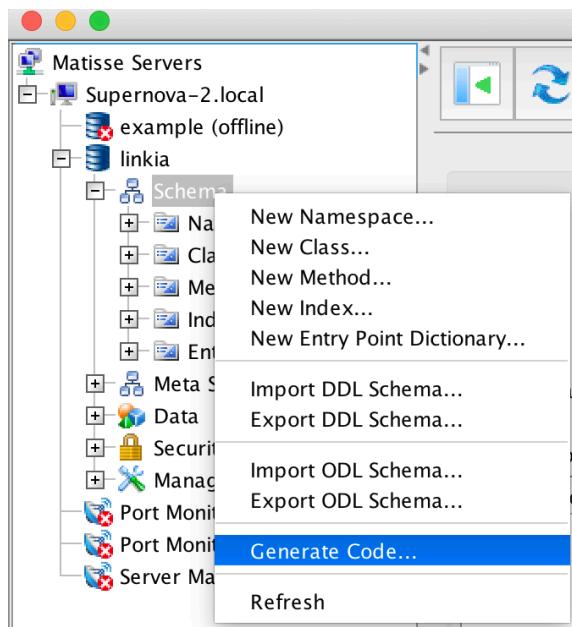
Por lo tanto, el SGBD-OO será el que genere las clases a partir del esquema que contiene en el lenguaje de programación de la aplicación. Este conjunto de clases, lo integraremos en la aplicación. Estas clases, junto con la librería que corresponda, serán los elementos que utilizaremos para gestionar la persistencia de la aplicación con la BBDD-OO, solamente llamando al método que corresponda en función de la acción que queramos realizar. La forma en la que esos objetos se hacen persistentes en la aplicación será completamente transparente para la aplicación, simplemente la aplicación llamará al método que corresponda en función de la acción a realizar y la acción se realizará, para la aplicación no habrá prácticamente diferencia entre gestionar un objeto en memoria o un objeto de la BBDD.



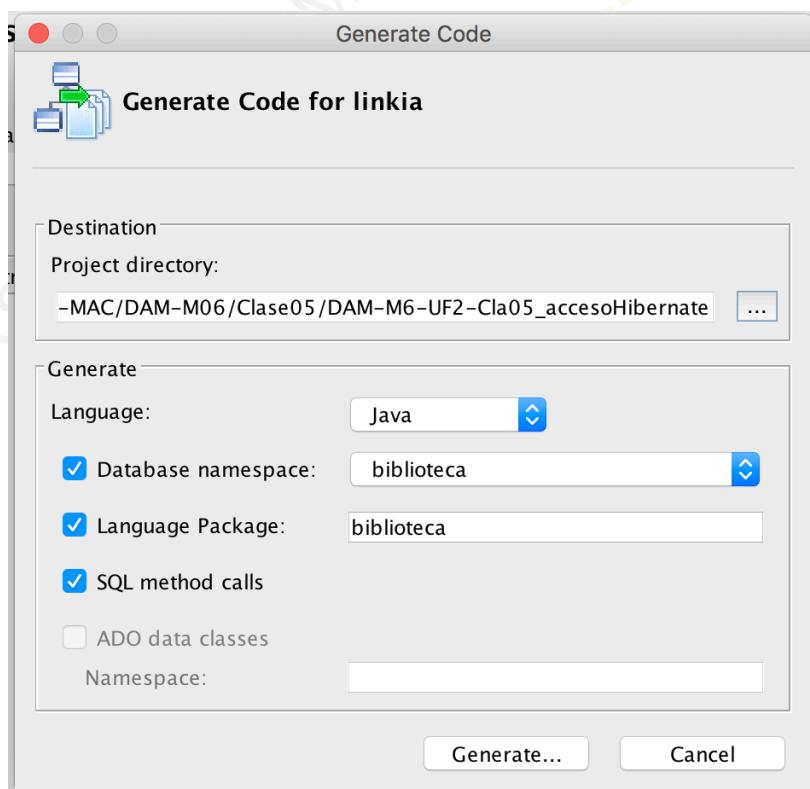
4.4.1. Creación del código de las clases

Para crear con Matisse la estructura de clases a partir del esquema e integrarlo en el proyecto en Java, seguiremos los siguientes pasos:

1. Sobre el esquema del que queremos crear las clases, pulsaremos con el botón derecho y seleccionaremos la opción Generate Code:



2. Al seleccionar esta opción, nos aparecerá la siguiente ventana:

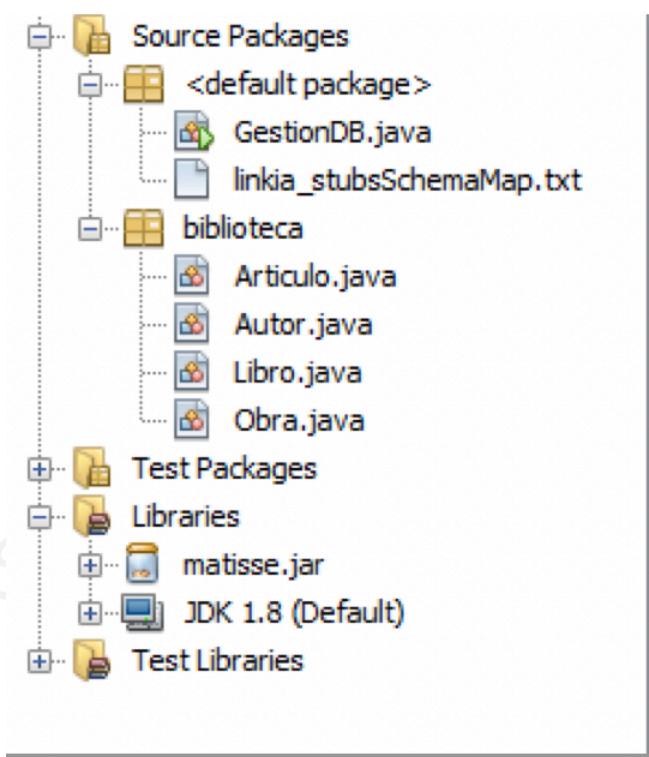




Los campos que tenemos que completar son los siguientes:

- Project directory: directorio del proyecto Java donde queremos que se generen las clases.
- Language: lenguaje de programación en el que queremos que se generen las clases, en nuestro caso, Java.
- Database namespace: el espacio de nombres de la BBDD del que queremos crear las clases.
- Language Package: el paquete dentro del proyecto Java en el que se crearán las clases.
- SQL method calls: al marcar esta opción indicaremos que queremos incluir las llamadas SQL a los métodos de las clases del esquema.

3. Una vez hemos completado la ventana anterior, ya tendremos las clases creadas en el proyecto Java.





4. El último paso que nos queda, es añadir en el proyecto la librería matissa.jar, para que éste reconozca las clases y métodos creados por Matisse. Esta librería la podemos encontrar en la carpeta de instalación de Matisse. En el caso de Windows, la ruta sería: C:\Products\Matisse\lib Además, tenemos que añadir los siguientes archivos al proyecto (en el caso de Windows) para que la conexión entre la aplicación Java y Matisse se realice correctamente: matisseJAVA.dll y matisseJAVA.lib.

4.4.2. Creación de objetos

Ahora que ya tenemos las clases creadas en el proyecto Java que representan al esquema de la BBDD, podemos empezar a programar la aplicación como tal solo centrándonos en la parte Java.

Una de las operaciones que realizamos en cualquier aplicación, es crear nuevos objetos. En el caso con el que estamos trabajando, es decir una SGBD-OO como es Matisse, seguiremos los siguientes pasos:

1. Crear el objeto en Java.
2. Crear la conexión a la BBDD.
3. Llamar a este nuevo objeto para almacenarlo en la BBDD.

Las clases y métodos para seguir estos pasos son:

- Clase MtDatabase: esta clase nos permite gestionar la conexión con la BBDD.
- Método MtDatabase: constructor de la clase anterior. Crea la conexión con la BBDD. Los parámetros que necesita son el servidor, la base de datos y el espacio de nombres al que queremos acceder.
- Método open: este método, abre la conexión con la base de datos definida en el constructor.



- Método startTransaction: este método nos permite iniciar una transacción para que incluya todas las sentencias de creación del objeto en la BBDD. Esta transacción finalizará al confirmarla o cuando se produzca un error.
- Método commit: este método nos permite indicar que la transacción ha finalizado con éxito y por lo tanto todos los cambios pueden hacerse persistentes en la BBDD.
- Método close: este método cierra la conexión con la BBDD.
- Clase MtException: esta clase nos permite gestionar las excepciones que se puedan producir durante la ejecución de los métodos anteriores.

En el siguiente ejemplo podemos ver el código para crear un nuevo objeto y almacenarlo en la BBDD:

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

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

        // Crea un objeto Autor
        Autor a1 = new Autor(db);
        a1.setNombre("Naruto");
        a1.setApellidos("Uzumaki");
        a1.setEdad("27");
        System.out.println("Autor creado...");

        // Crea un objeto Autor
        Autor a2 = new Autor(db);
        a2.setNombre("Pepe");
        a2.setApellidos("Sanchez");
        a2.setEdad("25");
        System.out.println("Autor creado...");
    }
}
```

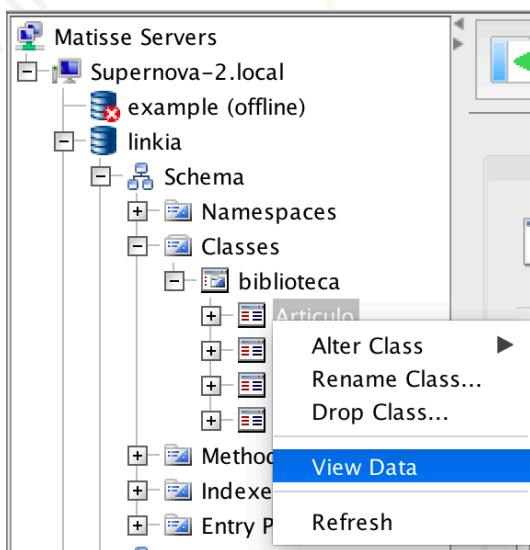


```
// Crea un objeto Libro
Libro l1 = new Libro(db);
l1.setTitulo("Konoha");
l1.setEditorial("Shinobi");
l1.setPaginas(512);
// Crea otro objeto Libro
Libro l2 = new Libro(db);
l2.setTitulo("Kyubi");
l2.setEditorial("Shinobi");
l2.setPaginas(498);
System.out.println("Libros creados...");

//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.
a2.setEscribe(o1);
//Ejecuta un commit para materializar las peticiones.
db.commit();
//Cierra la base de datos.
db.close();
System.out.println("\nHecho.");
} catch (MtException mte) {
    System.out.println("MtException : " + mte.getMessage());
}
}
```

Desde el propio Matisse, podemos ver el resultado de la ejecución del código del proyecto y ver qué objetos hay en la BBDD y cómo estos van cambiando en función de lo que ocurre en la aplicación.

Para ver qué objetos hay creados de una determinada clase, tenemos que pulsar con el botón derecho sobre la clase y seleccionar la opción View data.





Esta acción, creará una sentencia para hacer la consulta correspondiente. Esta consulta nos aparecerá en el panel derecho y para ejecutarla solo tenemos que pulsar sobre el botón play que tenemos en la barra de botones de la parte superior derecha.

The screenshot shows the Oracle SQL Developer interface. At the top, there is a toolbar with various icons. Below it is the 'Query Objects' pane, which includes dropdown menus for 'In snapshot' (set to 'Latest Version'), 'In namespace' (set to 'biblioteca'), and a search bar. The main area is titled 'SQL statement' and contains the following code:

```
--  
-- Click Execute (F5) to view instances of biblioteca.Autor.  
--  
SELECT * FROM "biblioteca"."Autor" LIMIT 100 OFFSET 0;|
```

Al ejecutar esta sentencia, el resultado lo veremos en la parte inferior del panel de la derecha.

The screenshot shows the results grid after executing the SQL query. The grid has columns for OID, nombre, apellidos, edad, and escribe. There is one row of data:

OID	nombre	apellidos	edad	escribe
0x109c Autor	Naruto	Uzumaki	27	NULL

4.4.3. Modificación de objetos

Para poder modificar objetos que ya existen en la BBDD, los pasos de crear la conexión, iniciar la transacción, confirmarla y cerrarlo todo al final, serán exactamente los mismos. A la hora de modificar un objeto, la diferencia con la creación es que antes de poder hacer la modificación, tenemos que localizar el objeto que queremos modificar.

Para poder localizar el objeto a modificar, una opción es utilizar un iterador MtObjectIterator. Una vez hayamos encontrado el objeto con este iterador, simplemente tendremos que cambiar los atributos correspondientes usando los métodos set.



El siguiente ejemplo realizar la modificación de un objeto almacenado en una BBDD Matisse:

```
public static void ModificaObjeto(String hostname, String dbname, String nombre, String nuevaEdad) {
    System.out.println("===== Modifica un objeto =====\n");
    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());
    }
}
```

En el ejemplo, podemos ver cómo creamos el iterador para buscar el objeto en concreto del que queremos cambiar los datos:

```
MtObjectIterator<Autor> iter = Autor.<Autor>instanceIterator(db);
```

A partir de este iterador, creamos un bucle que recorra todos los objetos de la clase Autor (por eso hacemos el cast al obtener el iterador). El bucle mira aquellos autores que tengan un nombre determinado y cambia su edad. En el ejemplo, tanto el nombre a buscar como la nueva edad a asignar, los pasamos como parámetro de entrada de la función.



4.4.4. Eliminación de objetos

Para eliminar objetos en una BBDD Matisse, utilizamos el método `deepRemove`.

Al igual que pasaba en los dos apartados anteriores, previo a la eliminación tenemos que establecer la conexión con la BBDD e iniciar la transacción.

Por otro lado, al igual que pasa cuando queremos modificar un objeto, antes de poder eliminarlo, tenemos que localizar dicho objeto utilizando el iterador `MtObjectIterator`.

En el siguiente ejemplo, tenemos el borrado de objetos en Matisse:

```
public static void borraObjetos(String hostname, String dbname) {
    System.out.println("===== Borra objetos =====\n");

    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" + 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());
    }
}
```

En el caso que queramos eliminar todos los objetos de una determinada clase, podemos utilizar el método `getClass(db).removeAllInstances()`, por ejemplo `Autor.getClass(db).removeAllInstances()`.



4.4.5. Consulta de objetos

La consulta es una de las operaciones básicas cuando trabajamos con una BBDD. En el caso de Matisse ocurre lo mismo.

A la hora de realizar consultas, podemos hacerlo de dos formas. La primera es ejecutar la consulta directamente sobre Matisse, tal y como hemos visto en secciones anteriores, seleccionando la clase sobre la que queremos ejecutar la consulta, creándola en el panel superior de la derecha y viendo los resultados de dicha consulta en el panel inferior de la derecha.

Pero para nosotros es más interesante la segunda forma, ya que la consulta la ejecutamos desde la aplicación, es decir, desde Java. Para ello seguiremos unos pasos muy similares a los que seguimos cuando usamos la API JDBC, pero utilizando clases y métodos propios de Matisse.

Para crear la consulta que queremos ejecutar, en el caso de Matisse utilizaremos el lenguaje OQL, que es la adaptación de SQL para trabajar con objetos.

A partir de aquí, los pasos que tenemos que seguir son los siguientes:

1. Creamos la conexión con la BBDD y abrimos dicha conexión.

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

2. Creamos una sentencia que será la que ejecute la consulta.

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



3. Ejecutamos la consulta con OQL. Para hacerlo, usaremos el método executeQuery pasándole la sentencia que queremos ejecutar como parámetro.

```
String commandText = "SELECT REF(a) from biblioteca.Autor a WHERE "
+ "a.nombre='Pepe' AND a.apellidos='Sanchez';";
```

```
ResultSet rset = stmt.executeQuery(commandText);
```

4. Los resultados de la consulta se guardan en el objeto ResultSet. Ahora solo nos queda recorrer este objeto con un bucle para extraer los resultados.

Lo importante que tenemos que tener en cuenta aquí es que una consulta OQL puede devolver un valor, un conjunto de valores o un objeto.

Aquí tenemos un ejemplo de consulta:

```
public static void ejecutaOQL(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. Esta consulta lo que hace es utilizar 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;";
        String commandText = "SELECT REF(a) from biblioteca.Autor a WHERE "
            + "a.nombre='Pepe' AND a.apellidos='Sanchez';";
        // Ejecuta la consulta y obtiene un ResultSet
        ResultSet rset = stmt.executeQuery(commandText);
        Autor a1;
        // Lee rset uno a uno.
        while (rset.next()) {
            // Obtiene los objetos Autor.
            a1 = (Autor) rset.getObject(1);
            // Imprime los atributos de cada objeto con un formato determinado.
            System.out.println("Autor: "
                + String.format("%16s", a1.getNombre())
                + String.format("%16s", a1.getApellidos())
                + " Spouse: "
                + String.format("%16s", a1.getEdad()));
        }
        // Cierra las conexiones
        rset.close();
        stmt.close();
    } catch (SQLException e) {
        System.out.println("SQLException: " + e.getMessage());
    }
}
```



4.5. Persistencia en BDOR

Una BBDD-OR es una base de datos objeto-relacional. Este tipo de BBDD han evolucionado desde el modelo relacional hasta una BBDD híbrida, ya que incluye tanto la tecnología relacional como la orientada a objetos. Son una alternativa que intenta dar las ventajas de la orientación a objetos pero sin dejar de lado el modelo relacional.

La diferencia fundamental entre una BBDD-OR y una BBDD-R es en que la primera el tipo de datos es un Objeto. Un tipo de objeto representa un elemento del mundo real y está compuesto por los siguientes elementos:

- Nombre: es el elemento que permite identificar al tipo de objeto.
- Atributos: definen la estructura de los objetos. El conjunto de atributos de un objeto junto con su valor, establecen el estado del objeto.
- Métodos: definen el comportamiento de los objetos. Deben estar escritos en un lenguaje de programación que soporte el SGBD-OR.

Al trabajar con objetos, los SGBD-OR también soportan los principios básicos de la OO, como pueden ser la encapsulación, la herencia, el polimorfismo, etc.



4.6. DB4o

DB4o son las siglas de Data Base For Objects, es una implementación OODB, bajo GPL y de código abierto, que permite de forma sencilla hacer persistentes objetos creados con Java y recuperarlos posteriormente en otra ejecución. Es un sistema que necesita muy pocas líneas de código y muy pocos recursos, ya que la forma de hacer persistentes los datos es mediante el uso de ficheros.

DB4o no es exactamente un SGBD-OO, ya que no sigue todas las especificaciones de la ODMG, como por ejemplo un lenguaje ODL. Por eso mismo, se le considera una BBDD-OR, es decir, una BBDD híbrida, pero una alternativa eficaz cuando lo que buscamos es la persistencia de pocos objetos en los que el tiempo de almacenaje y recuperación no es esencial.

Entre las ventajas de DB4o encontramos:

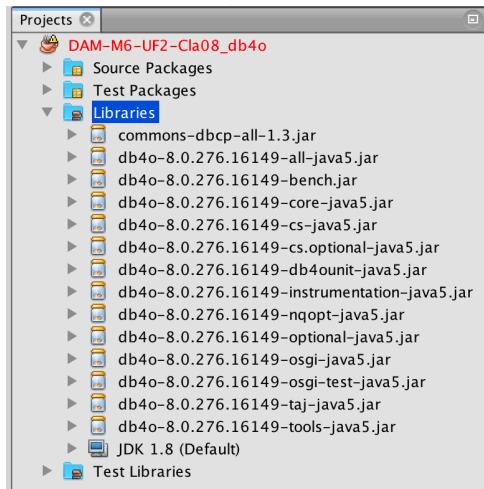
- Mayor velocidad de desarrollo (transparencia).
- Mejor rendimiento con objetos de negocio complejos:
 - Árboles, estructuras anidadas, relaciones N a N, relaciones recursivas.
- Fácil backup (la BBDD completa está en un único archivo).
- No necesita administración.
- Las búsquedas se hacen directamente usando objetos.
- Los cambios en los objetos (agregar o quitar atributos a una clase) se aplican directamente en la BBDD, sin tener que migrar datos ni reconfigurar nada.
- Multiplataforma.



Por otro lado, como todos los sistemas, también existen una serie de desventajas:

- No existe un lenguaje de consultas como sql (se deben realizar de forma programada).
- No existen restricciones, deben programarse (No implementa integridad referencial).
- Tamaño limitado de los ficheros de BBDD (2GB – 264GB).

Para poder implementar una aplicación Java con DB4o, tenemos que acceder a su página web y descargar la librería de clases que tendremos que añadir a nuestro proyecto Java para que pueda trabajar con las clases y métodos correspondientes.





4.6.1. Creación de las clases POJO

El concepto de clase POJO viene de la definición de las siglas POJO, Plain Old Java Object, lo que conocemos como una clase básica de Java.

Para trabajar con DB4o lo primero que tenemos que hacer es crear una clase POJO para cada uno de los objetos que queramos hacer persistentes. Estas clases, tienen que tener, como mínimo, los siguientes elementos:

- Atributos privados.
- Métodos get/set para cada uno de los atributos.
- Método `toString` para mostrar por pantalla un mensaje.

- Constructor que tenga tantos parámetros como atributos tiene la clase.
Este constructor tiene que inicializar todos y cada uno de los atributos con los parámetros de entrada.

A partir de aquí, la clase puede tener todos aquellos elementos que necesite para el funcionamiento de la aplicación. Otra cosa a remarcar es que estas clases no tienen código relacionado con DB4o.



Un ejemplo de clase POJO podría ser la siguiente:

```
public class Piloto {  
  
    private String name;  
    private int points;  
  
    public Piloto(String name, int points) {  
        this.name = name;  
        this.points = points;  
    }  
  
    public int getPoints() {  
        return points;  
    }  
  
    public void addPoints(int points) {  
        this.points += points;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String toString() {  
        return name + "/" + points;  
    }  
}
```

4.6.2. Apertura de la BBDD

Para acceder al fichero db4o o crear uno nuevo, utilizamos el método DB4oEmbedded.openFile().

- El primer parámetro es una plantilla de configuración. Para ello utilizamos:

DB4oEmbedded.newConfiguration()

- El segundo parámetro es el path al archivo.



Con esto obtenemos una instancia de ObjectContainer, que representa a la interfaz db4o. Para cerrar el objeto ObjectContainer utilizamos el método close().

```
// Atributo para la persistencia de 00
private ObjectContainer db;

public Gestordb4o() {
    db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "formula1");
}

public void cerrar() {
    db.close();
}
```

4.6.3. Almacenamiento de objetos

Para guardar un objeto, utilizamos el método *store()* pasándole el objeto a guardar.

```
public void guardarObjeto(Piloto p) {
    db.store(p);
    System.out.println("Objeto guardado " + p);
}
```

Llamando a la función del ejemplo tantas veces como necesitemos, iremos guardando los objetos en DB4o. Por ejemplo:

```
// GUARDAR OBJETOS
System.out.println("***** GUARDAR OBJETOS *****");
Piloto pilot1 = new Piloto("Pepe Perez", 100);
gestor.guardarObjeto(pilot1);
Piloto pilot2 = new Piloto("Fernando Alonso", 99);
gestor.guardarObjeto(pilot2);
Piloto pilot3 = new Piloto("Lewis Hamilton", 99);
gestor.guardarObjeto(pilot3);
Piloto pilot4 = new Piloto("Michael Schumacher", 100);
gestor.guardarObjeto(pilot4);
System.out.println("-----");
```



Siguiendo este código, tendríamos lo siguiente en DB4o:

Pepe Perez/100
Fernando Alonso/99
Lewis Hamilton/99
Michael Schumacher/100

4.6.4. Obtención de objetos

DB4o proporciona mecanismos para hacer una query a la BBDD. Uno de ellos es Query-By-Example. Con este sistema creamos un objeto prototipo para db4o que actúa como plantilla de los objetos que queremos obtener de la BBDD. DB4o devolverá todos los objetos que concuerden con el prototipo. Devuelve una instancia ObjectSet.

Para crear el objeto prototipo, utilizamos el constructor de la clase POJO. El valor que le pasamos a cada uno de los parámetros dependerá de si ese valor es necesario o no para hacer la búsqueda de los objetos que queremos recuperar. En el caso que el valor de un parámetro no sea necesario para realizar la búsqueda, éste se dejará en su valor por defecto. El valor por defecto dependerá del tipo del parámetro, por ejemplo, los números a 0, los objetos a null, etc. En el caso que queramos hacer una búsqueda de todos los objetos de una clase, como el valor de los parámetros no será determinante, dejaremos todos los valores a null. En el caso que queramos buscar los objetos que tengan un valor determinado en uno de sus atributos, éste será el valor que tendremos que pasarle al parámetro correspondiente al crear al objeto prototipo.



Según todo lo comentado, el siguiente ejemplo funcionará de la siguiente forma. El código del ejemplo es este:

```
public void obtenerPilotos(Piloto p) {  
    ObjectSet result = db.queryByExample(p);  
    System.out.println(result.size());  
    while (result.hasNext()) {  
        System.out.println(result.next());  
    }  
}
```

Esta función lo que hace es buscar todos los objetos Piloto que cumplan con el objeto prototipo pasado como parámetro de entrada. Podemos tener diferentes casos:

- Si queremos todos los pilotos. En este caso, tenemos que pasar un objeto Piloto con todos los valores por defecto, ya que ningún valor es determinante para realizar la búsqueda porque los estamos buscando todos.

```
Piloto prototipo = new Piloto(null, 0);
```

- Si queremos los pilotos con un nombre determinado. El nombre es el primer parámetro del constructor, por lo que pasaremos a ese primer parámetro el valor en concreto que queramos buscar. Si el resto de atributos no determinan la búsqueda porque vamos a buscar pilotos por nombre, los dejaremos con su valor por defecto.

```
Piloto prototipo = new Piloto("Pepe", 0);
```

Pasando este objeto como parámetro a la función del ejemplo (obtenerPilotos), buscará solo todos los pilotos cuyo nombre sea Pepe, independientemente del resto de valores de sus atributos.



- Si queremos los pilotos con unos determinados puntos. Los puntos son el segundo parámetro del constructor, por lo que pasaremos a ese segundo parámetro el valor en concreto que queramos buscar. Si el resto de atributos no determinan la búsqueda porque vamos a buscar pilotos por el número de puntos, los dejaremos con su valor por defecto.

```
Piloto prototipo = new Piloto(null, 100);
```

Pasando este objeto como parámetro a la función del ejemplo (obtenerPilotos), buscará solo todos los pilotos cuyo número de puntos sean 100, independientemente del resto de valores de sus atributos.

- Si queremos los pilotos con un determinado nombre y unos determinados puntos. En este caso queremos buscar con un valor para cada uno de los atributos, por lo que crearemos el objeto pasando cada uno de esos valores.

```
Piloto prototipo = new Piloto("Pepe", 100);
```

Pasando este objeto como parámetro a la función del ejemplo (obtenerPilotos), buscará solo todos los pilotos cuyo nombre sea Pepe y número de puntos sean 100.

Teniendo en cuenta todos estos ejemplos, además de una función que tenga como parámetro un objeto Piloto, podríamos tener varias funciones de búsqueda, una para cada criterio que podemos tener.



```
public void buscarPilotosPorNombre(String nombre) {  
    Piloto prototipo = new Piloto(nombre, 0);  
    ObjectSet result = db.queryByExample(prototipo);  
    System.out.println(result.size());  
    while (result.hasNext()) {  
        System.out.println(result.next());  
    }  
}  
  
public void buscarPilotosPorPuntos(int puntos) {  
    Piloto prototipo = new Piloto(null, puntos);  
    ObjectSet result = db.queryByExample(prototipo);  
    System.out.println(result.size());  
    while (result.hasNext()) {  
        System.out.println(result.next());  
    }  
}
```

El método queryByExample está sobrecargado. Una segunda versión de este método permite la búsqueda de todos los objetos de una determinada clase, para ello el parámetro que le pasamos es el nombre de la clase de la que queremos hacer la búsqueda.

```
public void obtenerTodosLosPilotos() {  
    ObjectSet result = db.queryByExample(Piloto.class);  
    System.out.println(result.size());  
    while (result.hasNext()) {  
        System.out.println(result.next());  
    }  
    /* List<Piloto> pilots = db.query(Piloto.class);  
    System.out.println(pilots);*/  
}
```

Otra opción de hacer una búsqueda de objetos es a través del método query. En este caso, el método también recibe como parámetro de entrada el nombre de la clase de los objetos que queremos realizar la búsqueda.

```
public void obtenerTodosLosPilotos() {  
    List<Piloto> pilots = db.query(Piloto.class);  
    System.out.println(pilots);  
}
```



4.6.5. Modificación de objetos

Modificar un objeto es tan sencillo como guardarlo, por lo tanto, volveremos a utilizar el método store. La diferencia con crear, es que previamente tenemos que localizar el objeto que queremos modificar. Los pasos son los siguientes:

1. Localizar el objeto que queremos utilizando el método queryByExample comentado anteriormente. En este caso, como estamos buscando un objeto en concreto, el que queremos modificar, tendremos que pasar un objeto prototipo que se ajuste al objeto que estamos buscando, es decir pasándole un valor determinado para cada uno de los parámetros en el constructor.
2. Cuando tengamos este objeto, simplemente lo tenemos que pasar como parámetro al método store y ya lo tendremos modificado en DB4o.

```
public void actualizarPiloto(String nombre, int puntos) {  
    ObjectSet result = db.queryByExample(new Piloto(nombre, puntos));  
    System.out.println(result.size());  
    while (result.hasNext()) {  
        Piloto found = (Piloto) result.next();  
        found.addPoints(11);  
        db.store(found);  
        System.out.println("Se han añadido 11 puntos a: " + found);  
    }  
    this.obtenerTodosLosPilotos();  
}
```



4.6.6. Eliminación de objetos

Para eliminar un objeto, utilizaremos el método delete pasándole el objeto que queremos eliminar. Al igual que pasa con el caso de modificar, primero tendremos que buscar el objeto que queremos modificar. Los pasos son los siguientes:

1. Localizar el objeto que queremos utilizando el método queryByExample comentado anteriormente. En este caso, como estamos buscando un objeto en concreto, el que queremos modificar, tendremos que pasar un objeto prototipo que se ajuste al objeto que estamos buscando, es decir pasándole un valor determinado para cada uno de los parámetros en el constructor.
2. Cuando tengamos este objeto, simplemente lo tenemos que pasar como parámetro al método delete y ya lo tendremos eliminado en DB4o

```
public void eliminarPiloto(String nombre, int puntos){  
    ObjectSet result = db.queryByExample(new Piloto(nombre, puntos));  
    System.out.println(result.size());  
    while (result.hasNext()) {  
        Piloto found = (Piloto) result.next();  
        db.delete(found);  
        System.out.println("Objeto borrado " + found);  
    }  
    this.obtenerTodosLosPilotos();  
}
```



Recursos y Enlaces

- [Java](#)



- [API Java 11](#)



- [NetBeans](#)



- [Matisse](#)





- [Matisse Java Programmer's Guide](#)



- [DB4o Java Guide](#)



Conceptos clave

- **ODL:** lenguaje de definición de objetos.
- **OML:** lenguaje de manipulación de objetos.
- **OQL:** lenguaje de consultas de objetos.
- **Matisse:** SGBD-OO.
- **DB4o:** Data Base for Objects. BBDD-OR, por lo tanto es una BBDD híbrida entre el mundo relacional y el mundo de la orientación a objetos.
- **Objeto prototipo:** objeto creado con el constructor de una clase POJO y que sirve de patrón de búsqueda de objetos en DB4o.
- **POJO:** Plain Old Java Object, lo que conocemos como una clase básica de Java.



Test de autoevaluación

1. Pon el significado de cada una de las siglas de ODMG:
 - a. O
 - b. D
 - c. M
 - d. G

2. ¿Cómo se llama el lenguaje que permite crear consultas para ejecutar en un SGBD-OO?
 - a. OML
 - b. ODL
 - c. OQL
 - d. SQL

3. ¿Cómo se llama la clase que permite crear un iterador para localizar un objeto en Matisse?
 - a. MtObjectIterator
 - b. MtIterator
 - c. MtIteratorObject
 - d. MtIteratorSearch

4. Pon el significado de cada una de las siglas de DB4o:
 - a. D
 - b. B
 - c. 4
 - d. O



5. ¿Qué método permite de la librería de clases de DB4o permite hacer persistente un objeto de la aplicación Java?
 - a. create
 - b. save
 - c. store
 - d. add

6. ¿Qué método sobrecargado permite hacer una búsqueda de objetos en DB4o?
 - a. queryByExample
 - b. query
 - c. queryByPrototipe
 - d. searchByExample



Ponlo en práctica

Actividad 1

Inserción de objeto en Matisse - Crear una función que añada un objeto a una BBDD Matisse.

Actividad 2

Realiza una función que realice una búsqueda de objetos almacenados en DB4o. La función buscará los objetos que tengan nombre “Naruto” y “Uzumaki”, teniendo en cuenta que nombre y apellidos son los atributos de la clase POJO.



SOLUCIONARIOS

Test de autoevaluación

1. Pon el significado de cada una de las siglas de ODMG:

- a. O – **Object**
- b. D – **Data**
- c. M – **Management**
- d. G – **Group**

2. ¿Cómo se llama el lenguaje que permite crear consultas para ejecutar en un SGBD-OO?

- a. OML
- b. ODL
- c. **OQL**
- d. SQL

3. ¿Cómo se llama la clase que permite crear un iterador para localizar un objeto en Matisse?

- a. **MtObjectIterator**
- b. MtIterator
- c. MtIteratorObject
- d. MtIteratorSearch

4. Pon el significado de cada una de las siglas de DB4o:

- a. D – **Data**
- b. B – **Base**
- c. 4 – **for**
- d. O – **Objects**



5. ¿Qué método permite de la librería de clases de DB4o permite hacer persistente un objeto de la aplicación Java?
 - a. create
 - b. save
 - c. **store**
 - d. add

6. ¿Qué método sobrecargado permite hacer una búsqueda de objetos en DB4o?
 - a. **queryByExample**
 - b. query
 - c. queryByPrototype
 - d. searchByExample



Ponlo en práctica

Actividad 1

Inserción de objeto en Matisse - Crear una función que añada un objeto a una BBDD Matisse.

Solución:

```
private void insertarModulo() throws IOException, SQLException {
    String nombre = Utilidades.pideModulo("Introduce Nombre del Módulo: ");

    try {
        MtDatabase db = new MtDatabase(hostname, TestBDOO.dbname,new MtPackageObjectFactory("",spacename));
        db.open();
        db.startTransaction();

        Modulo m = new Modulo(db);
        m.setId((int)ConsultarUltimoId("Modulo") + 1);
        m.setNombre(nombre);
        System.out.println("Modulo creado ...");

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

Actividad 2

Realiza una función que realice una búsqueda de objetos almacenados en DB4o. La función buscará los objetos que tengan nombre “Naruto” y “Uzumaki”, teniendo en cuenta que nombre y apellidos son los atributos de la clase POJO.

Solución:

```
public void obtenerPilotos() {
    ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "nombres");
    Persona p = new Persona("Naruto", "Uzumaki");
    ObjectSet result = db.queryByExample(p);
    System.out.println(result.size());
    while (result.hasNext()) {
        System.out.println(result.next());
    }
}
```