

1. EVOLUCIÓN DE LAS BASES DE DATOS.....	2
2. BASES DE DATOS ORIENTADAS A OBJETOS.....	3
2.1. CARACTERÍSTICAS	3
2.2. EL ESTÁNDAR ODMG.....	6
2.2.1. <i>Modelo de Objetos</i>	6
2.2.2. <i>Lenguaje de Definición de Objetos (ODL)</i>	7
2.2.3. <i>Lenguaje de Consultas de Objetos (OQL)</i>	9
2.2.3.1. <i>Operadores de comparación</i>	11
2.2.3.2. <i>Cuantificadores y operadores sobre colecciones</i>	12
2.3. SISTEMAS GESTORES.....	13
3. NEODATIS ODB	14
3.1. ALMACENAMIENTO Y RECUPERACIÓN DE OBJETOS	15
3.2. EXPLORADOR DE OBJETOS	18
3.3. CONSULTAS CON CRITERIOS.....	24
3.4. CONSULTAS MÁS COMPLEJAS CON FUNCIONES	30

1. EVOLUCIÓN DE LAS BASES DE DATOS

Hace ya más de 40 años que Edgar F. Codd desarrolló el **modelo relacional** para la gestión de bases de datos, que se basa en los datos y en las relaciones existentes entre ellos. Sin lugar a dudas, este modelo es referencia a la hora de almacenar y recuperar información: es el más extendido e implementado.

Los **Sistemas Gestores de Bases de Datos Relacionales (SGBD-R)** son las aplicaciones software que gestionan los datos según el modelo relacional. Arquitecturas privativas (como *Oracle* y *Microsoft SQL Server*) o basadas en software libre (como *MySQL* y *PostgreSQL*) son ejemplos actuales del gran rendimiento que este tipo de sistemas ofrece a los desarrollos de software.

En la década de los 90, surgió de nuevo el paradigma de Programación Orientada a Objetos (POO), impulsado por el lenguaje de programación Java. La aceptación de la POO dio paso a una visión más amplia del problema, es decir, a un **modelo orientado a objetos** (que se centra en las operaciones realizadas sobre los datos) como herramienta para diseñar software, crear código y almacenar datos.

Para satisfacer el almacenamiento de datos, en el mercado aparecieron nuevas propuestas de **Sistemas Gestores de Bases de Datos Orientadas a Objetos (SGBD-OO)**, 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, pues en teoría no es necesaria una conversión entre el modelo de POO (por ejemplo con Java) y el modelo de base de datos (por ejemplo con O2).

Sin embargo, esta conversión sí es necesaria si se utiliza POO y un modelo relacional para el almacenamiento de objetos, pues requiere un mapeo entre los datos estructurados y las propiedades de los objetos a las tablas y atributos del modelo relacional.

El mayor problema de los SGBD-OO y el modelo orientado a objetos es la comparación con los SGBD-R y el modelo relacional. El principal inconveniente que ofrece el modelo orientado a objetos 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 orientado a objetos ofrece una alternativa no formal, que pone en duda la implementación óptima de los SGBD-OO y los relega a una posición inferior al compararlos con los SGBD-R.

En general, quizás los SGBD-OO no consiguen unos resultados tan buenos (espacio de almacenamiento, eficiencia en consultas, escalabilidad) a la hora de almacenar y recuperar información como sí ofrecen los SGBD-R.

Estas dudas sobre 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 la programación orientada a objetos como en la persistencia de los mismos. Estas soluciones de compromiso se llamaron **Sistemas Gestores de Bases de Datos Objeto-Relacionales (SGBD-OR)**. Ofrecen una interfaz que simula ser orientada a objetos, pero internamente los objetos se almacenan como en los sistemas relacionales clásicos. Con esta estructura, se pretende conseguir las ventajas que tienen ambos modelos.

2. BASES DE DATOS ORIENTADAS A OBJETOS

Las **Bases de Datos Orientadas a Objetos (BDOO)** son aquellas cuyo modelo de datos está orientado a objetos y soportan el paradigma orientado a objetos, almacenando datos y métodos. Su origen se debe principalmente a la existencia de problemas para representar cierta información y modelar ciertos aspectos del mundo real. Las BDOO simplifican la programación orientada a objetos (POO), almacenando directamente los objetos en la base de datos y empleando las mismas estructuras y relaciones que los lenguajes de POO.

Un Sistema Gestor de Bases de Datos Orientadas a Objetos (SGBD-OO) es un sistema gestor de bases de datos (SGBD) que almacena objetos.

2.1. CARACTERÍSTICAS

Las características asociadas a las bases de datos orientadas a objetos son:

- Los datos se almacenan como objetos.
- Cada objeto se identifica mediante un identificador único u **OID (Object Identifier)**, que no es modificable por el usuario.
- Cada objeto define sus atributos y métodos, y la interfaz con la que se puede acceder a ellos. El usuario puede especificar qué atributos y métodos se pueden usar desde fuera.

Un SGBD-OO debe contemplar las siguientes características:

- **Características propias de la Orientación a Objetos:** encapsulación, identidad, herencia y polimorfismo, junto con control de tipos y persistencia.
- **Características propias de un Sistema Gestor de Bases de Datos:** persistencia, concurrencia, recuperación ante fallos, gestión del almacenamiento secundario y facilidad de consultas.

En 1989, Malcolm Atkinson propuso el *Manifiesto de los Sistemas de Bases de Datos Orientadas a Objetos Puras*, que contiene 13 características obligatorias para los SGBD-OO:

- 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 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 base de datos orientada a objetos 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 se deben poder aplicar 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) **Complejidad de Cálculos.** El lenguaje de manipulación de datos (DML) debe ser completo.
- 8) **Conjunto de Tipos de Datos 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 se deben mantener (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) **Gestión de Gran Cantidad de Datos.** Debe proporcionar mecanismos transparentes al usuario, que aseguren 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 ante Fallos.** Debe poseer un mecanismo de recuperación ante fallos hardware y software similar al de los sistemas relacionales (igual de eficiente).
- 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).



Los SGBD-OO tienen las siguientes **ventajas**:

- Mayor capacidad de modelado. La utilización de objetos permite representar de una forma más natural los datos que se necesitan almacenar.
- Extensibilidad. Se pueden construir nuevos tipos de datos a partir de tipos existentes.
- Existe una única interfaz entre el lenguaje de manipulación de datos (DML) y el lenguaje de programación. Esto elimina el tener que incrustar un lenguaje declarativo como SQL en un lenguaje imperativo como Java o C.
- Lenguaje de consultas más expresivo. El lenguaje de consultas es navegacional de un objeto al siguiente, en contraste con el lenguaje declarativo SQL.
- Soporte a transacciones largas, necesario para muchas aplicaciones de bases de datos avanzadas.
- Adecuación a aplicaciones avanzadas de bases de datos (CASE, CAD, sistemas multimedia).

Los SGBD-OO tienen los siguientes **inconvenientes**:

- Falta de un modelo de datos universal. La mayoría de los modelos carecen de una base teórica.
- Falta de experiencia. El uso de los SGBD-OO es todavía relativamente limitado.

- Falta de estándares. No existe un lenguaje de consultas estándar como SQL, aunque está el lenguaje **OQL (Object Query Language)** de ODMG (Object Data Management Group), que se está convirtiendo en un estándar de facto.
- Competencia con los SGBD-R y los SGBD-OR.
- La optimización de consultas compromete la encapsulación. Optimizar consultas requiere conocer la implementación para acceder a la base de datos de una manera eficiente.
- Complejidad. El incremento de funcionalidad provisto para un SGBD-OO lo hace más complejo que un SGBD-R. La complejidad conlleva productos más caros y difíciles de usar.
- Falta de soporte a las vistas. La mayoría de los SGBD-OO no proveen mecanismos de vistas.
- Falta de soporte a la seguridad.

2.2. EL ESTÁNDAR ODMG

ODMG (Object Data Management Group) es un consorcio o grupo industrial formado por fabricantes de bases de datos con el objetivo de definir estándares para los SGBDOO. Su principal objetivo es sacar adelante un conjunto de especificaciones (estándares) que permitan a los desarrolladores escribir aplicaciones portables para bases de datos orientadas a objetos y herramientas ORM. Uno de sus estándares, el cual lleva el mismo nombre del grupo (ODMG) especifica los elementos que se definirán, y en qué manera se hará, para la consecución de persistencia en las BDOO que soporten el estándar.

Entre 1993 y 2001, publicó cinco revisiones de su estándar. La última revisión fue **ODMG versión 3.0**, publicada en 2000. Los principales componentes de este estándar ODMG versión 3.0 son los siguientes:

- Modelo de Objetos.
- Lenguaje de Definición de Objetos (ODL).
- Lenguaje de Consultas de Objetos (OQL).
- Conexión con el Lenguaje C++, Lenguaje Smalltalk y Lenguaje Java.

2.2.1. Modelo de Objetos

El **modelo de objetos** ODMG especifica las características de los objetos, cómo se relacionan, cómo se identifican, construcciones soportadas, etc. Las primitivas de modelado básicas son: los objetos caracterizados por un identificador único (OID) y los literales que son objetos que no tienen identificador, no pueden aparecer como objetos, están embebidos a ellos.

Como se ha comentado, 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, sino que puede ser una estructura o un conjunto de valores relacionados que se guardan bajo un solo nombre (por ejemplo, enumeraciones).

Los objetos se dividen en tipos. Los **tipos de objetos** se pueden entender como las clases en POO:

- **Tipos Atómicos:** *boolean, short, unsigned short, int, long, unsigned long, float, double, char, string, enum, octect.*
- **Tipos Estructurados:** *date, time, timestamp, interval.*
- **Colecciones:**
 - *set<tipo>*. Colección o grupo desordenado de objetos del mismo tipo que no admite duplicados.
 - *bag<tipo>*. Colección o grupo desordenado de objetos del mismo tipo que permite duplicados.
 - *list<tipo>*. Colección ordenada de objetos del mismo tipo que permite duplicados.
 - *array<tipo>*. Colección ordenada de objetos del mismo tipo a los que se puede acceder por su posición. El tamaño es dinámico.
 - *dictionary<clave,valor>*. Colección de objetos del mismo tipo en la que cada valor está asociado a una clave.

Los literales pueden ser atómicos (*long, short, boolean, unsigned long, etc.*), colecciones (*set, bag, list, array, dictionary*), estructuras (*date, interval, time, timestamp*) y NULL.

Mediante las **Clases** especificamos el estado y el comportamiento de un tipo de objeto, puede incluir métodos. Son instanciables, por lo que a partir de ellas se pueden crear instancias de objetos individuales. Son equivalentes a una clase concreta en los lenguajes de programación. Una clase es un tipo de objeto asociado a un “extend”.

Los objetos de un mismo tipo tienen un mismo comportamiento y muestran un rango de estados común:

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

Un tipo tiene una interfaz y una o más implementaciones. La **interfaz** define las propiedades visibles externamente y las operaciones soportadas para 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**. Es la 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).

2.2.2. Lenguaje de Definición de Objetos (ODL)

Object Definition Language (ODL) es el equivalente al DDL (lenguaje de definición de datos) de los SGBD-R tradicionales. 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. Es decir, 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 elemento del modelo orientado a objetos. La sintaxis de ODL extiende el lenguaje de definición de interfaces CORBA (*Common Object Request Broker Architecture*).

ODL ofrece al diseñador de bases de datos un sistema de tipos semejantes a los de otros lenguajes de programación orientados a objetos. Los tipos permitidos son:

- **Tipos Básicos**. Incluyen los tipos atómicos (*boolean, short, integer, long, float, double, char, string*) y las enumeraciones.
- **Tipos de Interfaz o Estructurados**. Son tipos complejos obtenidos al combinar tipos básicos mediante 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.
- *Lista (List<tipo>)*. Denota el tipo cuyos valores son listas ordenadas finitas conteniendo 0 o más elementos del tipo. Un caso especial es el tipo *String*, que es una abreviatura del tipo *List<char>*.
- *Vector (Array<tipo,n>)*. Denota el tipo cuyos elementos son vectores de n elementos del tipo.

Algunas de las palabras reservadas para definir objetos son:

- **class**. Declaración del objeto, define el comportamiento y el estado de un tipo de objeto.
- **extent**. Define la extensión, nombre para el actual conjunto de objetos de la clase. En las consultas se hace referencia al nombre definido en esta cláusula, no se hace referencia al nombre definido a la derecha de *class*.
- **key[s]**. Declara la lista de claves para identificar las instancias.
- **attribute**. Declara un atributo.
- **set / bag / list / array**. Declara un tipo de colección conjunto, bolsa, lista o vector.
- **struct**. Declara un tipo estructurado.
- **enum**. Declara un tipo enumerado.
- **relationship**. Declara una relación.
- **inverse**. Declara una relación inversa.
- **extends**. Define la herencia simple.

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 de ese esquema.

El siguiente ejemplo muestra la definición de un esquema que contiene las clases *Cliente*, *Producto*, *LineaVenta* y *Venta*. El objeto Cliente tiene como clave el NIF. Se definen atributos y un método; uno de los atributos es un tipo estructurado (*struct*), otro es enumerado (*enum*) y también hay un tipo colección (*set*):

```
class Cliente (extent Clientes key NIF) {
    /* definición de atributos */
    attribute string NIF;
    attribute struct Nombre_Completo {
        string apellidos,
        string nombre_pila
    } nombre;
    attribute date fecha_nacimiento;
    attribute enum Genero {H,M} sexo;
    attribute struct Direccion_Completa {
        string calle,
        string población,
        string provincia
    } direccion;
    attribute set<string> telefonos;
    /* definición de operaciones */
    short calcular_edad();
}
```

Definimos el objeto Producto:

```
class Producto (extent Productos key ID)
{
```



```

        /* definición de atributos */
        attribute short IDPRODUCTO;
        attribute string descripcion;
        attribute float precio;
        attribute short stock_minimo;
        attribute short stock_actual;
    }

```

Definimos el objeto Línea de Venta con los datos de la línea y la operación para calcular el importe de la línea:

```

class LineaVenta (extent LineasVentas)
{
    /* definición de atributos */
    attribute short numero_linea;
    attribute Producto product;
    attribute short cantidad;
    /* definición de operaciones */
    float calcular_importe();
}

```

A continuación definimos el objeto Venta y sus relaciones: una venta es realizada por un cliente (*es_realizada_por*) y la inversa, el cliente lleva a cabo o realiza una venta (*realiza*), también se define un atributo colección (*set*) para las líneas de venta:

```

class Venta (extent Ventas key ID)
{
    /* definición de atributos */
    attribute short IDVENTA;
    attribute date fecha_venta;
    attribute set<LineaVenta> lineas;
    /* definición de relaciones */
    relationship Cliente es_realizada_por inverse::realiza;
    /* definición de operaciones */
    float calcular_total_venta();
}

```

ODMG no define ningún lenguaje de manipulación de datos (OML). El motivo es claro: relegar esta tarea a los propios lenguajes de programación. Es decir, serán los lenguajes de programación orientados a objetos los que accederán a los objetos para modificarlos, cada uno con su sintaxis y sus posibilidades. Con esto, se persigue el objetivo de no diferenciar en la ejecución de un programa entre objetos persistentes almacenados en una base de datos y objetos no persistentes creados en memoria.

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

2.2.3. Lenguaje de Consultas de Objetos (OQL)

Object Query Language (OQL) es el lenguaje declarativo estándar de consultas de BDOO.

Tanto las definiciones de las bases de datos orientadas a objetos como de OQL fueron posteriores a las bases de datos relacionales y a SQL. En realidad, SQL estaba muy extendido y aceptado por los desarrolladores y clientes de bases de datos cuando apareció OQL. Por esta razón, OQL se definió lo más parecido a la sintaxis usada en SQL (*Select-From-Where*). Así, los nuevos usuarios no apreciarían en este lenguaje diferencias significativas con respecto a SQL y obtendrían una curva de aprendizaje más rápida.

Las principales características de OQL son:

- Es orientado a objetos y está basado en el modelo de objetos de ODMG.

- Es un lenguaje declarativo del tipo de SQL y con una sintaxis similar a SQL.
- Acceso declarativo a los objetos de la base de datos (propiedades y métodos).
- Semántica formal bien definida.
- No incluye operaciones de actualización, sólo de consulta. Las modificaciones se realizan mediante los métodos que los objetos poseen.
- Dispone de operadores sobre colecciones (min, max, count) y cuantificadores (for all, exists).

La sintaxis básica de OQL se basa en una sentencia SELECT:

```
SELECT <lista de valores>
FROM <lista de colecciones y miembros típicos>
[WHERE <condición>]
```

Las colecciones en FROM pueden ser extensiones (nombres que aparecen a la derecha de *extent*) o expresiones que evalúan una colección. Se suele utilizar una variable iterador que vaya tomando valores de los objetos de la colección. Las variables iterador se pueden especificar de varias formas utilizando la cláusula IN o AS:

```
FROM Clientes c
FROM c IN Clientes
FROM Clientes AS c
```

Para acceder a los atributos y objetos relacionados, se utilizan expresiones de camino. Una expresión de camino empieza normalmente con un nombre de objeto o una variable iterador, seguida de atributos conectados mediante un punto o nombres de relaciones. Por ejemplo, para obtener el nombre de los clientes que son mujeres, se puede escribir:

```
SELECT c.nombre.nombre_pila FROM Clientes c WHERE c.sexo = "M";
SELECT c.nombre.nombre_pila FROM c IN Clientes WHERE c.sexo = "M";
SELECT c.nombre.nombre_pila FROM Clientes AS c WHERE c.sexo = "M";
```

En general, supongamos que *v* es una variable cuyo tipo es *Venta*:

- *v.IDVENTA* es el identificador de venta del objeto *v*.
- *v.fecha_venta* es la fecha de venta del objeto *v*.
- *v.calcular_total_venta()* obtiene el total de venta del objeto *v*.
- *v.es_realizada_por* es un puntero al cliente mencionado en *v*.
- *v.es_realizada_por.direccion* es la dirección del cliente mencionado en *v*.
- *v.lineas* es una colección de objetos del tipo *LineaVenta*. El uso de *v.lineas.numero_linea* no es correcto, porque *v.lineas* es una colección de objetos y no un objeto simple.
- Cuando se tiene una colección como *v.lineas*, para acceder a los atributos de dicha colección se usa la cláusula FROM.

Ejemplos:

- 1) Obtener los datos del cliente cuyo IDVENTA sea 1.

```
SELECT v.es_realizada_por.nombre, v.es_realizada_por.direccion,
       v.fecha_venta, v.calcular_total_venta()
FROM Ventas v
WHERE v.IDVENTA = 1;
```

- 2) Obtener las líneas de venta de la venta cuyo IDVENTA sea 1.

```
SELECT lv.numero_linea, lv.product.descripcion,
```

```
lv.cantidad, lv.calcular_importe()
FROM Ventas v, v.lineas lv
WHERE v.ID = 1;
```

El resultado de una consulta OQL puede ser de cualquier tipo soportado por el modelo. Por ejemplo, la consulta anterior devuelve un conjunto de estructuras del tipo *short*, *string* y *float*; el resultado es del tipo colección: *bag(struct(numero_linea:short, descripción:string, cantidad:short, importe:float))*.

En cambio la consulta: *SELECT c.nombre.nombre_pila FROM c IN Clientes WHERE c.sexo = "M"*; devuelve un conjunto de nombres; el tipo devuelto es: *bag(string)*.

****Recordemos la diferencia entre las colecciones *set* y *bag*, *set* es el grupo desordenado de objetos del mismo tipo que no permite duplicados y *bag* permite duplicados.**

Se puede usar alias en las consultas, por ejemplo, la SELECT anterior: *SELECT lv.numero_linea, lv.product.descripcion, lv.cantidad, lv.calcular_importe()*; se puede expresar usando alias de la siguiente manera:

```
SELECT nlin:lv.numero_linea,
dpro:lv.product.descripcion,
can:lv.cantidad,
imp:lv.calcular_importe()
```

Y el tipo devuelto en este caso es: *bag(struct(nlin::short, dpro:string, can:short, imp:float))*.

Para obtener como resultado una *set* de estructuras (colección que no admita duplicados), se usa **DISTINCT** a la derecha de la sentencia **SELECT**:

```
SELECT DISTINCT c.nombre.nombre_pila
FROM c IN clientes
WHERE c.sexo = "M";
```

En este caso el tipo devuelto es: *set(string)*.

Para obtener como resultado una lista (un tipo *list*) de estructuras ordenada con un criterio, se usa la cláusula **ORDER BY**:

```
SELECT nl:lv.numero_linea, dpl:lv.product.descripcion,
cl:lv.cantidad, il:lv.calcular_importe()
FROM Ventas v, v.lineas lv
WHERE v.ID = 1
ORDER BY nl ASC;
```

El tipo devuelto es: *list(struct(nlin::short, dpro:string, can:short, imp:float))*.

2.2.3.1. Operadores de comparación

Para comparar valores numéricos se pueden utilizar los siguientes operadores:

- < (menor que).
- <= (menor o igual que).
- > (mayor que).
- >= (mayor o igual que).
- = (igual que).
- != (distinto de).

Para comparar cadenas de caracteres se pueden utilizar los operadores **IN** y **LIKE**:

- **IN.** Comprueba si existe un carácter en una cadena de caracteres (`caracter IN cadena`).
- **LIKE.** Comprueba si dos cadenas de caracteres son iguales (`cadena1 LIKE cadena2`). La segunda cadena puede contener caracteres especiales:
 - `_` o `?`. Indicador de posición que representa cualquier carácter.
 - `*` o `%`. Representa una cadena de caracteres.

Ejemplos:

- Obtener los datos de ventas de los clientes de la población de TOLEDO y cuyos apellidos empiecen por la letra A:

```
SELECT v.IDVENTA, v.fecha_venta, v.calcular_total_venta()
FROM Ventas v
WHERE (v.es_realizada_por.direccion.poblacion = "Toledo"
      AND v.es_realizada_por.nombre.apellidos LIKE "A%");
```

- Obtener para el IDVENTA 1 aquellas líneas de venta cuya descripción del producto contenga el carácter P en su descripción:

```
SELECT lv.numero_linea, lv.product.descripcion,
       lv.cantidad, lv.calcular_importe()
FROM Ventas v, v.lineas lv
WHERE (v.IDVENTA = 1 AND 'P' IN lv.product.descripcion);
```

2.2.3.2. Cuantificadores y operadores sobre colecciones

Mediante el uso de cuantificadores, se puede comprobar si todos los miembros, algunos miembros o al menos un miembro de una colección satisfacen una condición:

- Todos los Miembros: **FOR ALL** x **IN** coleccion : condicion
- Alguno/Cualquier Miembro: coleccion comparacion **SOME/ANY** condicion
(comparación puede ser: <, <=, =, >=, >)
- Al menos 1 Miembro: **EXISTS** x **IN** coleccion : condicion
EXISTS x
- Sólo 1 Miembro: **UNIQUE** x

Ejemplo:

- Obtener todas las ventas que tengan líneas de venta cuya descripción del producto sea "Pendrive 32 GB".

```
SELECT v.IDVENTA, v.fecha_venta, v.calcular_total_venta()
FROM Ventas v
WHERE EXISTS x IN v.lineas : x.product.descripcion = "Pendrive 32 GB";
```

- Obtener las ventas que solo tienen líneas de venta cuya descripción del producto sea "Pendrive 32 GB".

```
SELECT v.ID, v.fecha_venta, v.calcular_total_venta()
FROM Ventas v
WHERE FOR ALL x IN v.lineas : x.product.descripcion = "Pendrive 32 GB";
```

Los operadores **AVG**, **SUM**, **MIN**, **MAX** y **COUNT** se pueden aplicar a cualquier colección, siempre y cuando tengan sentido para el tipo de elemento. Por ejemplo, para calcular la media del total de ventas necesitaríamos asignar el valor devuelto a una variable:

```
Media = AVG (SELECT v.calcular_total_venta() from Ventas v)
```

El tipo devuelto es una colección de un elemento: `bag(struct(calcular_total_venta: float))`.

Como hemos visto **OQL** es bastante complejo y actualmente ningún creador de software lo ha implementado completamente. **OQL** ha influenciado el diseño de algunos lenguajes de consulta nuevos como **JDOQL** (*Java Data Object Query Lenguaje*) y **EJBQL** (*Enterprise Java Bean Query Lenguaje*), pero estos no pueden ser considerados como versiones de OQL.

2.3. SISTEMAS GESTORES

Existe una oferta significativa de **Sistemas Gestores de Bases de Datos Orientadas a Objeto (SGBD-OO)** en el mercado, aunque no es tan extensa como ocurre con los SGBD-R. Como en el caso de los sistemas gestores relacionales, existen:

- Sistemas privativos:
 - 1) ObjectStore: <https://www.ignitetechnology.com/solutions/information-technology/objectstore>
 - 2) Objectivity/DB: <http://www.objectivity.com/products/objectivitydb/>
 - 3) Versant: <http://www.actian.com/products/operational-databases/versant/>
- Sistemas bajo licencias de software libre:
 - 1) Matisse: <http://www.fresher.com/>
 - 2) NeoDatis ODB: <http://neodatis.wikidot.com/>
 - 3) EyeDB: <http://www.eyedb.org/>
 - 4) Ozone Database Project: <http://www.ozone-db.org/>
- Licencia dual GPL/comercial
 - 1) DB4O: <https://sourceforge.net/projects/db4o/>
<http://www.epidataconsulting.com/tikiwiki/tiki-index.php?page=db4o>

3. NEODATIS ODB

NeoDatis ODB es una base de datos orientada a objetos sencilla que actualmente corre en los lenguajes Java, .NET, Groovy y Scala, y que tiene una licencia LGPL de GNU.

Con NeoDatis ODB, se evita la falta de impedancia entre los paradigmas orientado a objetos y relacional, ya que actúa como una capa de persistencia nativa y transparente para Java, .NET y Mono. Los objetos se pueden almacenar y recuperar con una sola línea de código, sin necesidad de tener que mapearlos a tablas.

Sus principales características son las siguientes:

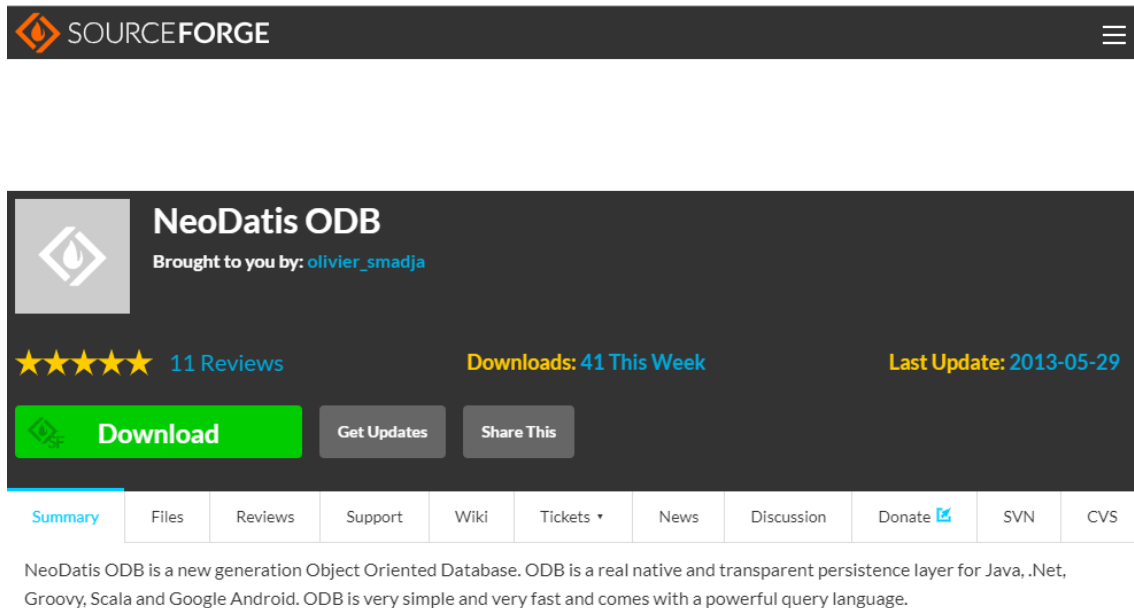
- Es muy simple e intuitiva, con un tiempo de aprendizaje mínimo.
- Su motor de bases de datos ocupa menos de 800 KB y se distribuye como un único fichero JAR/DLL que puede empaquetarse fácilmente en cualquier aplicación.
- Es rápida. Puede almacenar más de 30000 objetos en un segundo.
- Soporta transacciones ACID para garantizar la integridad de los datos.
- Utiliza un único fichero para almacenar la base de datos, incluyendo el meta-modelo, los objetos y los índices.
- Es multiplataforma. Funciona con Java, .NET, Groovy y Scala.
- Permite exportar e importar todos los datos a/de un fichero XML, con lo cual se garantiza la disponibilidad de los datos.
- La persistencia de datos se realiza con pocas líneas de código, sin modificar clases ni necesidad de mapeo.
- Se distribuye bajo la licencia LGPL de GNU.

Neodatos ODB es una base de datos orientada a objetos de código abierto que funciona con Java, .Net, Groovy y Android. Los objetos se pueden almacenar y recuperar con una sola línea de código, sin necesidad de tener que mapearlos a tablas.

Desde el sitio web oficial de NeoDatis ODB es: <http://neodatis.wikidot.com/>.



La sección *Download* conduce al repositorio de <https://sourceforge.net/projects/neodatis-odb/>. Desde aquí, se puede descargar la última versión (**fichero neodatis-odb-1.9.30.689.zip**).



NeoDatis ODB is a new generation Object Oriented Database. ODB is a real native and transparent persistence layer for Java, .Net, Groovy, Scala and Google Android. ODB is very simple and very fast and comes with a powerful query language.

Al descomprimir este fichero ZIP, se obtiene el fichero **neodatis-odb-1.9.30.689.jar**, que deberá ser ubicado en la carpeta correspondiente, para después definirlo en la variable de entorno CLASSPATH o para incluirlo en un proyecto del entorno de desarrollo Eclipse. Además, desde la carpeta `/doc/javadoc` se puede acceder a la documentación de la API de NeoDatis ODB.

3.1. ALMACENAMIENTO Y RECUPERACIÓN DE OBJETOS

El siguiente programa Java presenta la clase *Jugador*, de la cual se van a instanciar varios objetos para posteriormente almacenarlos y recuperarlos en/de una base de datos NeoDatis ODB.

```
//Clase Jugadores
public class Jugadores {
    //atributos
    private String nombre;
    private String deporte;
    private String ciudad;
    private int edad;
    //constructores
    public Jugadores() {
    }
    public Jugadores(String nombre, String deporte, String ciudad, int edad) {
        this.nombre = nombre;
        this.deporte = deporte;
        this.ciudad = ciudad;
        this.edad = edad;
    }
}
```

```

//métodos de acceso
public void setNombre(String nombre) {
    this.nombre = nombre;
}
public String getNombre() {
    return nombre;
}
public void setDeporte(String deporte) {
    this.deporte = deporte;
}
public String getDeporte() {
    return deporte;
}
public void setCiudad(String ciudad) {
    this.ciudad = ciudad;
}
public String getCiudad() {
    return ciudad;
}
public void setEdad(int edad) {
    this.edad = edad;
}
public int getEdad() {
    return edad;
}
}

```

Para realizar operaciones con la base de datos, se utiliza la clase **ODBFactory**, que devuelve una instancia de la interfaz **ODB**, que representa la interfaz pública con el usuario, es decir, lo que el usuario ve.

Para abrir la base de datos se usa el método *open()* (devuelve como se ha mencionado antes un ODB):

```
ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
```

Para almacenar objetos se usa el método *store()*:

```
// Crear instancias para almacenar en BD
Jugadores j1 = new Jugadores("Maria", "voleibol", "Madrid", 14);

// Almacenamos objetos
odb.store(j1);
```

Una vez almacenados los objetos, para recuperarlos usamos el método *getObjects()*, que recibe la clase cuyos objetos se van a recuperar y devuelve una colección de objetos que usa esa clase:

```
//recuperamos todos los objetos
Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
```

Para validar los cambios en la base de datos se usa el método *close()*.

```
odb.close(); // Cerrar BD
```



```

public class EjemploNeodatis {
    public static void main(String[] args) {

        // Crear instancias para almacenar en BD
        Jugadores j1 = new Jugadores("Maria", "voleibol", "Madrid", 14);
        Jugadores j2 = new Jugadores("Miguel", "tenis", "Madrid", 15);
        Jugadores j3 = new Jugadores
            ("Mario", "baloncesto", "Guadalajara", 15);
        Jugadores j4 = new Jugadores("Alicia", "tenis", "Madrid", 14);

        ODB odb = ODBFactory.open("neodatis.test");// Abrir BD

        // Almacenamos objetos
        odb.store(j1);
        odb.store(j2);
        odb.store(j3);
        odb.store(j4);

        //recuperamos todos los objetos
        Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
        System.out.printf("%d Jugadores: %n", objects.size());

        int i = 1;
        // visualizar los objetos
        while(objects.hasNext()){
            Jugadores jug = objects.next();
            System.out.printf("%d: %s, %s, %s %n",
                i++, jug.getNombre(), jug.getDeporte(),
                jug.getCiudad(), jug.getEdad());
        }
        odb.close(); // Cerrar BD
    }
}

```

Los principales métodos de la interfaz **ODB** son:

- **open(esquema)**. Abre la base de datos indicada y devuelve un objeto ODB.
- **store(objeto)**. Almacena un objeto en la base de datos.
- **delete(objeto)**. Elimina un objeto de la base de datos.
- **deleteCascade(objeto)**. Elimina un objeto y todos sus sub-objetos asociados.
- **getObjectId(objeto)**. Obtiene el identificador del objeto especificado.
- **getObjects(clase)**. Recupera una colección de objetos de la clase indicada.
- **commit()**. Valida los cambios realizados en la base de datos.
- **rollback()**. Deshace los cambios realizados y no validados de la base de datos.
- **close()**. Valida automáticamente los cambios realizados y cierra la base de datos asociada.

Se puede acceder a los objetos conociendo su **OID (Object Identifier)**. El siguiente programa Java muestra los datos del objeto cuyo OID es 3:

```
import org.neodatis.odm.ODB;
import org.neodatis.odm.ODBFactory;
import org.neodatis.odm.OID;
import org.neodatis.odm.core.oid.OIDFactory;

public class ejemploOid {
    public static void main(String[] args) {
        ODB odb = ODBFactory.open("neodatis.test"); // Abrir BD
        OID oid = OIDFactory.buildObjectOID(3); // Obtener objeto con OID 3
        //visualizar los datos del jugador recuperado
        Jugadores jug = (Jugadores) odb.getObjectFromId(oid);
        System.out.printf("%s, %s, %s, %d %n",
            jug.getNombre(), jug.getDeporte(), jug.getCiudad(), jug.getEdad());
        odb.close(); // Cerrar BD
    }
}
```

El **OID** de un objeto es devuelto también por los métodos *store(objeto)* y *getObjectId(objeto)*. Por ejemplo, para obtener el OID de un objeto j1, se puede hacer de dos maneras:

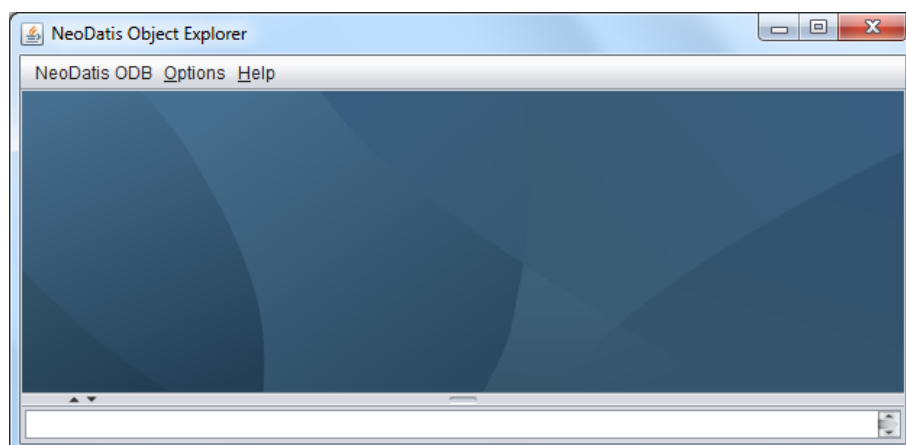
```
OID oid = odb.store(j1);
OID oid = odb.getObjectId(j1);
```

3.2. EXPLORADOR DE OBJETOS

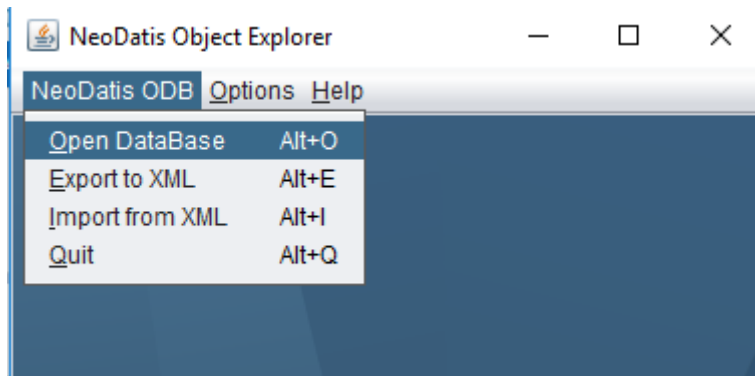
Neodatis disponer de un explorador, **NeoDatis Object Explorer**. Es una herramienta gráfica que viene con la base de datos NeoDatis para gestionar los datos. Esta herramienta permite:

- Exportar/importar una base de datos a/de un fichero XML.
- Navegar por los objetos de la base de datos.
- Manipular objetos (crear, actualizar y eliminar).
- Realizar consultas de objetos.
- Refactorizar la base de datos.

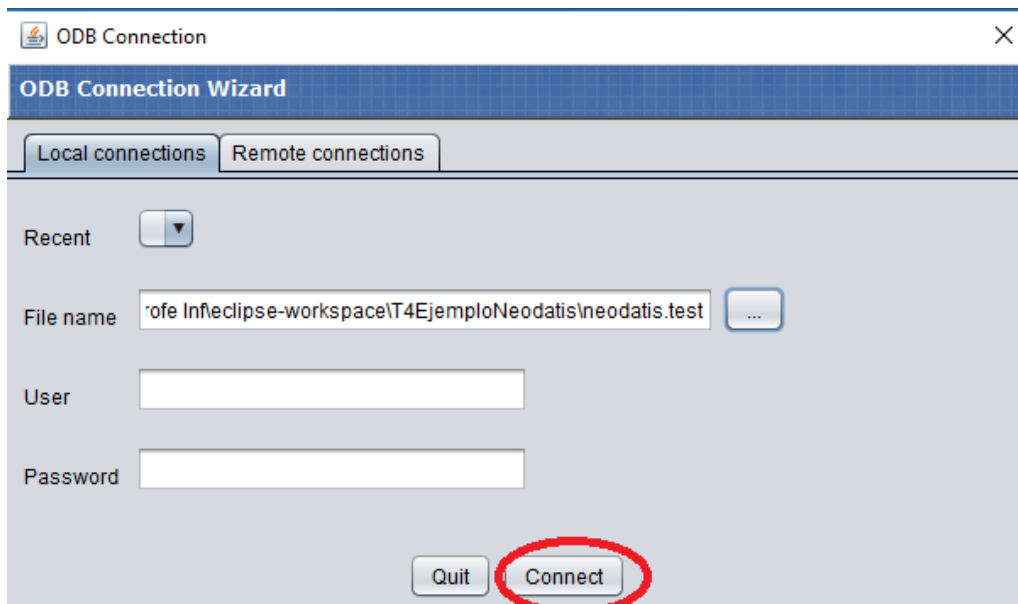
Para ejecutarla, se hace doble clic en el fichero **odb-explorer.bat** (sistemas Windows) u **odb-explorer.sh** (sistemas Linux).



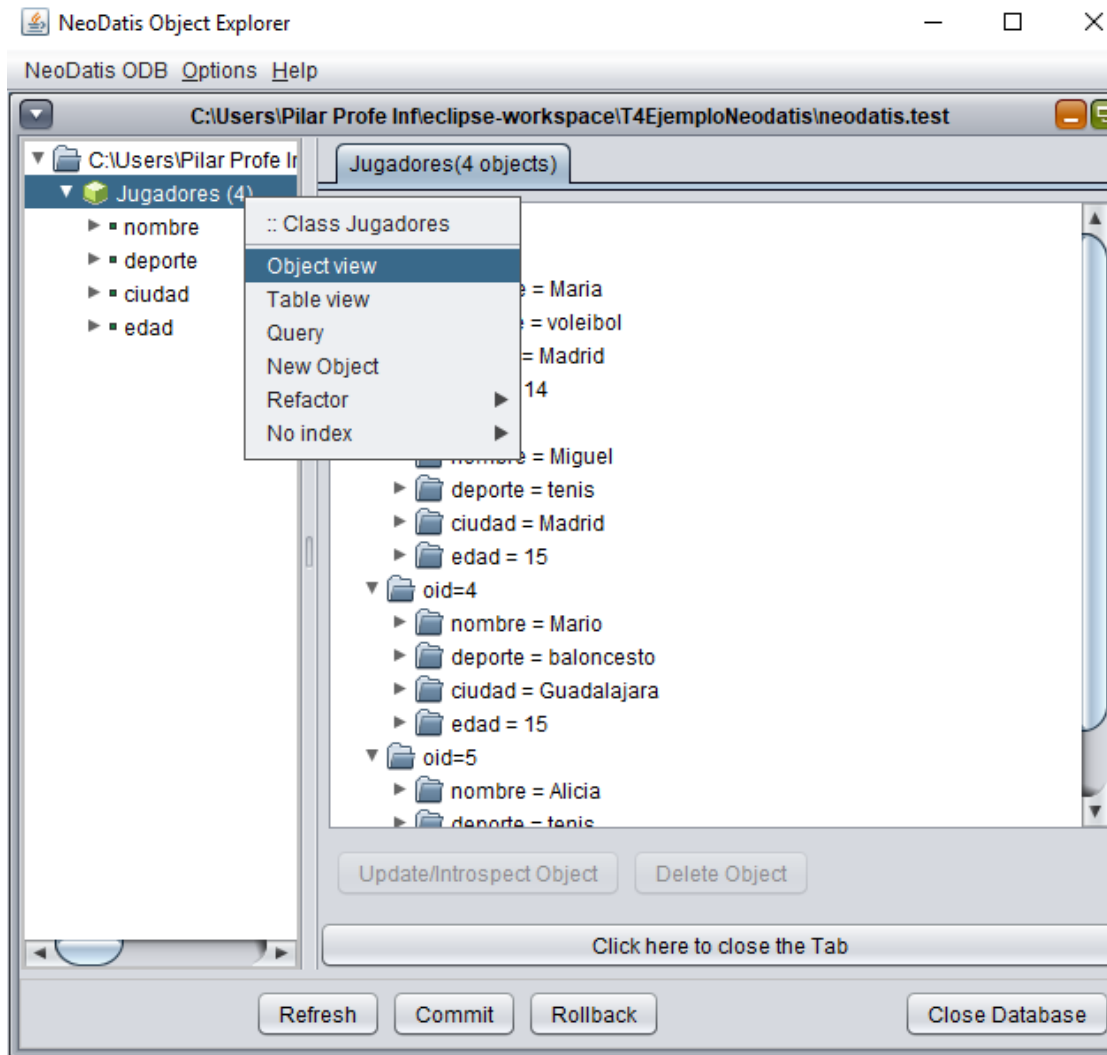
A continuación, en primer lugar es necesario abrir una base de datos para poder navegar por ella (pulsamos en el menú **NeoDatis ODB -> Open Database**).



Se abre una nueva ventana con 2 pestañas, nos quedamos en la primera (*Local connections*), localizamos en nuestro disco el fichero *neodatis.test* y pulsamos el botón *Connect*.

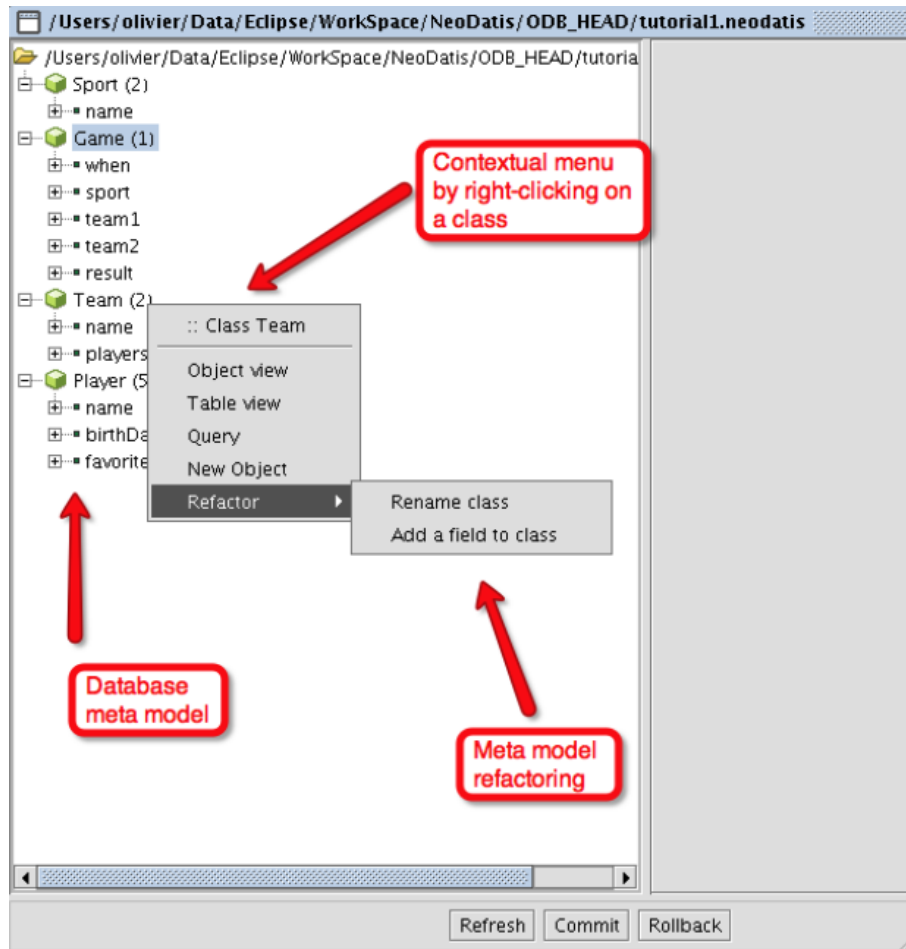


Se abre el explorador, al pulsar el botón derecho del ratón sobre la clase Jugadores podemos acceder a la vista de los objetos, vista en formato de tabla, realizar consultas, crear un nuevo objeto, etc..



Desde el explorador se pueden modificar los objetos, eliminarlos, etc. Después para realizar cada operación hemos de pulsar el botón *Commit* para validar los cambios. Para finalizar con la base de datos pulsamos el botón *Close DataBase*.

A continuación, se muestran más opciones del explorador de objetos: el meta-modelo de la base de datos en el panel izquierdo. Aquí se pueden ver las definiciones de las clases, con sus atributos y métodos:

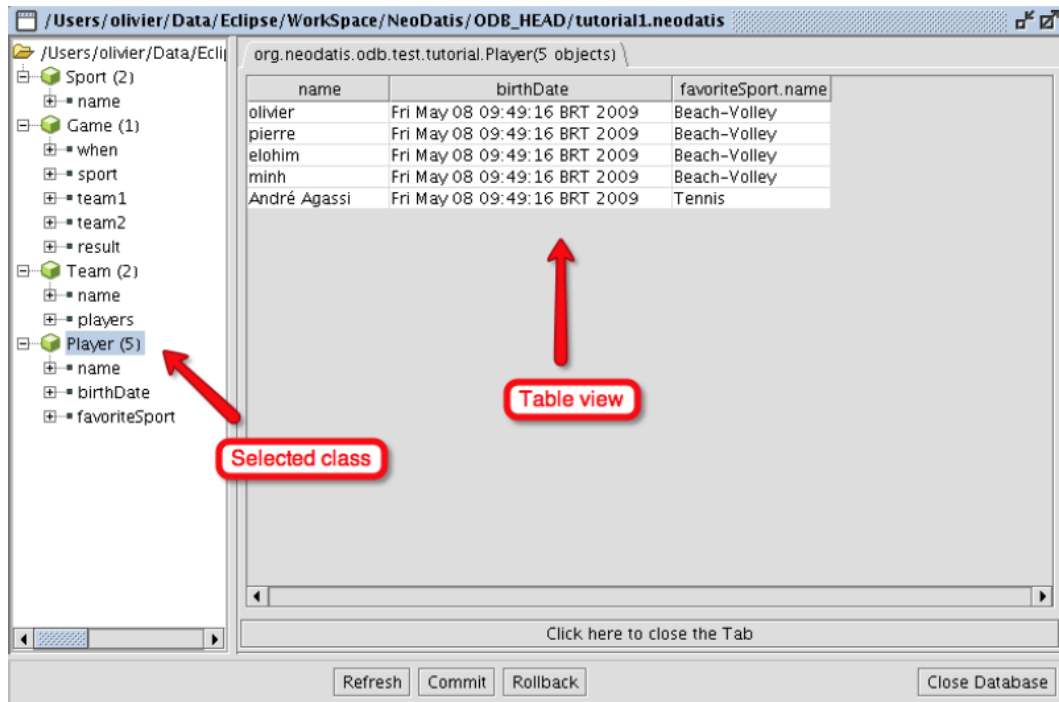


Pulsando sobre una clase aparece un menú contextual:

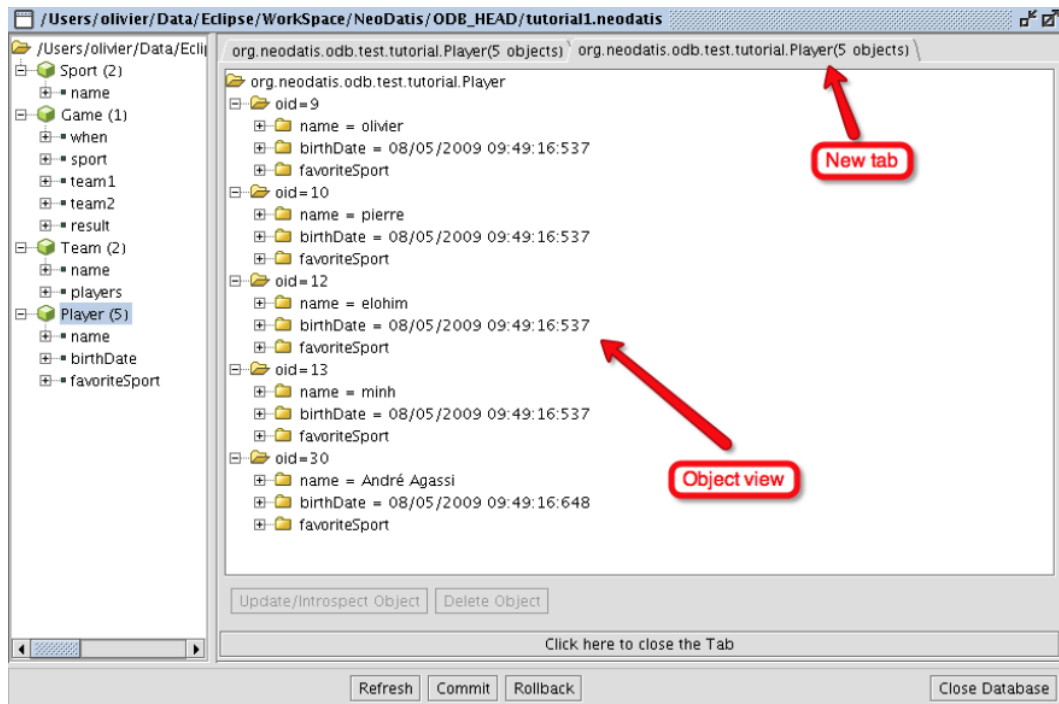
- La opción **Object View (Vista de Objetos)** muestra todos los objetos en un modo jerárquico.
- La opción **Table View (Vista de Tabla)** muestra los datos en un resultado de consulta parecida a SQL.
- La opción **Query (Consulta)** abre un asistente gráfico para realizar consultas con criterios.
- La opción **New Object (Nuevo Objeto)** abre una ventana para crear una nueva instancia de la clase especificada.
- La opción **Refactor (Renombrado)** sirve para cambiar el nombre de una clase de la base de datos y para añadir un campo a una clase.

Existen dos formas de visualizar los datos:

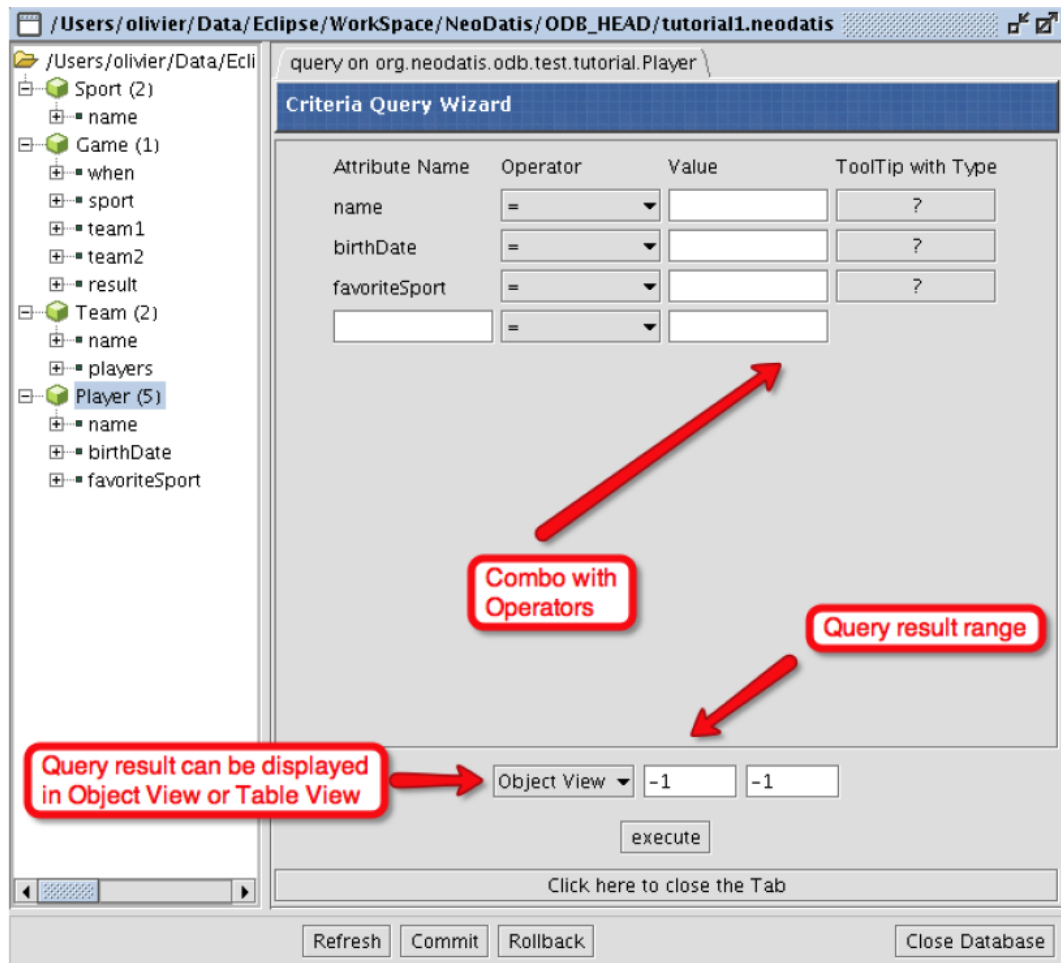
- **Table View (Vista de Tabla).** Muestra los resultados en un resultado de consulta parecida a SQL.



- **Object View (Vista de Objetos).** Muestra los objetos en una jerarquía de árbol según el modelo de objetos de la base de datos.



El Explorador de Objetos también dispone de una interfaz gráfica para realizar consultas con criterios sobre un subconjunto dado de objetos:



3.3. CONSULTAS CON CRITERIOS

Para realizar consultas con NeoDatis ODB se utiliza la clase **CriteriaQuery**, especificando la clase de la cual se buscan objetos y el criterio de selección de los mismos. Este criterio de selección es similar a la cláusula WHERE, en la cual se indica la condición que deben cumplir los objetos seleccionados. Necesitamos importar los siguientes paquetes:

```
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
```

El criterio de la consulta se especifica mediante la interfaz **ICriterion** y/o la clase **Where**, que dispone de los siguientes métodos para construir criterios y expresiones:

- **equal(atributo, valor)**. Indica una comparación de *igualdad* para los tipos primitivos (*boolean*, *short*, *int*, *long*, *byte*, *float*, *double* y *char*) y para objetos.
- **like(atributo, valor)**. Indica una comparación de *similitud* usando una patrón con caracteres.
- **gt(atributo, valor)**. Indica una comparación de *mayor que* para los tipos primitivos (*short*, *int*, *long*, *byte*, *float*, *double* y *char*) y para objetos.
- **ge(atributo, valor)**. Indica una comparación de *mayor o igual que* para los tipos primitivos (*short*, *int*, *long*, *byte*, *float*, *double* y *char*) y para objetos.
- **lt(atributo, valor)**. Indica una comparación de *menor que* para los tipos primitivos (*short*, *int*, *long*, *byte*, *float*, *double* y *char*) y para objetos.
- **le(atributo, valor)**. Indica una comparación de *menor o igual que* para los tipos primitivos (*short*, *int*, *long*, *byte*, *float*, *double* y *char*) y para objetos.
- **contain(atributo, valor)**. Comprueba si una colección (vector, lista) contiene un valor específico de tipo primitivo (*boolean*, *short*, *int*, *long*, *byte*, *float*, *double* y *char*) o un objeto.
- **isNull(atributo)**. Comprueba si un atributo es nulo.
- **isNotNull(atributo)**. Comprueba si un atributo es no nulo.

El siguiente programa Java realiza una consulta sobre los jugadores que practican el deporte tenis:

El método *CriteriaQuery()* utiliza *Where.equal* para seleccionar los objetos que cumplan la condición especificada. Para ordenar la salida ascendentemente se usa el método *orderByAsc()*, entre paréntesis se ponen los atributos por los que se realiza la ordenación; para ordenar desdecientemente se usa *orderByDesc()*.


```

import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBuilderFactory;
import org.neodatis.odb.Objects;

import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class ConsultarJugadoresTenis {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        ODB odb = ODBuilderFactory.open("neodatis.test");// Abrir BD

        //ICriterion crit = Where.equal("deporte", "tenis");
        //IQuery query = new CriteriaQuery(Jugador.class, crit);

        //Consulta sobre los jugadores que practican el deporte tenis
        IQuery query = new CriteriaQuery(Jugadores.class, Where.equal("deporte", "tenis"));

        //ordena ascendentemente el resultado de la consulta por nombre y edad
        query.orderByAsc("nombre,edad");

        //recuperar todos los jugadores de la consulta
        Objects <Jugadores> jugadores = odb.getObjects(query);
        System.out.println(jugadores.size() + " jugadores");

        // visualizar los jugadores (objetos)
        int i = 1;
        while(jugadores.hasNext()){
            Jugadores jug = jugadores.next();
            System.out.println((i++) + "\t: " + jug.getNombre() + "*" +
                               jug.getDeporte() + "*" + jug.getCiudad() + "*" + jug.getEdad());
        }

        odb.close(); // Cerrar BD
    }
}

```

La clase `CriteriaQuery` devuelve una instancia de la interfaz **IQuery**, que opcionalmente permite ordenar el resultado de la consulta con uno de los siguientes métodos:

- **orderByAsc(atributos)**. Ordena el resultado de la consulta de forma ascendente según los atributos especificados (en una cadena de texto y separados por comas).
- **orderByDesc(atributos)**. Ordena el resultado de la consulta de forma descendente según los atributos especificados (en una cadena de texto y separados por comas).

Con `CriteriaQuery()` se puede usar la interfaz `ICriterion` para construir el criterio de la consulta, para usarlo será necesario importar otros paquetes:

```
import org.neodatis.odb.core.query.criteria.ICriterion;
```

Por ejemplo:

```
ICriterion criterio = Where.equal("deporte", "tenis");
IQuery query = new CriteriaQuery(Jugador.class, criterio);
```

Para modificar un objeto, primero es necesario cargarlo, después lo modificamos usando los métodos `set` del objeto y a continuación los actualizamos con el método `store()`. Los cambios que se realicen en la base de datos se validarán con el método `commit()`, aunque también se validara cerrando la base de datos con el método `close()`.

El siguiente programa Java modifica el deporte de la jugadora que se llama María:

```

import org.neodatis.odb.ODB;
import org.neodatis.odb.ODFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class ModificarDeporteMaria {

    public static void main(String[] args) {

        ODB odb = ODFactory.open("neodatis.test");// Abrir BD
        //Consulta sobre los jugadores cuyo nombre es Maria
        IQuery query = new CriteriaQuery(Jugadores.class, Where.equal("nombre", "Maria"));
        //recuperar todos los jugadores de la consulta
        Objects <Jugadores> jugadores = odb.getObjects(query);
        //obtener solamente el primer jugador encontrado en la consulta
        Jugadores jugador = (Jugadores) jugadores.getFirst();
        //actualizar el jugador
        jugador.setDeporte("voley-playa");
        odb.store(jugador);
        odb.commit();
        //validar los cambios y cerrar la base de datos
        odb.close(); // Cerrar BD

    }

}

```

Para eliminar un objeto, primero lo localizamos como hemos hecho anteriormente, y luego usamos el método *delete()*:

```
odb.delete(jugador);
```

Con *CriteriaQuery* se puede usar la interfaz *ICriterion* para construir el criterio de la consulta, para usarlo será necesario importar otros paquetes:

```
import org.neodatis.odb.core.query.criteria.ICriterion;
```

Por ejemplo, para obtener los jugadores cuya edad es 14 utilizamos el criterio *Where.equal()*:

```
ICriterion criterio = Where.equal("edad", 14);
IQuery query = new CriteriaQuery(Jugador.class, criterio);
```

Para obtener los jugadores cuyo nombre empieza por la letra M usamos *Where.like()*:

```
// Jugadores que empiezan por M
ICriterion criterio = Where.like("nombre", "M%");

// Jugadores que no empiezan por M
ICriterion criterio = Where.not(Where.like("nombre", "M%"));
```

Para obtener los jugadores cuya edad es > que 14 usamos *Where.gt()*:

```
// Jugadores cuya edad es menor que 14
ICriterion criterio = Where.gt("edad", 14);
```

Para mayor o igual que usamos: **Where.ge("edad", 14);**

Para menor que usamos: **Where.lt("edad", 14);**

Para menor o igual usamos: **Where.le("edad", 14);**

Los siguientes son ejemplos de criterios de búsqueda con operadores de comparación:

```
import org.neodatis.odbc.core.query.criteria.ICriterion;
import org.neodatis.odbc.core.query.criteria.Where;

// Jugadores cuya edad es 14
ICriterion criterio = Where.equal("edad", 14);
// Jugadores que empiezan por M
ICriterion criterio = Where.like("nombre", "M%");
// Jugadores que no empiezan por M
ICriterion criterio = Where.not(Where.like("nombre", "M%"));
// Jugadores cuya edad es mayor que 14
ICriterion criterio = Where.gt("edad", 14);
// Jugadores cuya edad es mayor o igual que 14
ICriterion criterio = Where.ge("edad", 14);
// Jugadores cuya edad es menor que 14
ICriterion criterio = Where.lt("edad", 14);
// Jugadores cuya edad es menor o igual que 14
ICriterion criterio = Where.le("edad", 14);

IQuery query = new CriteriaQuery(Jugador.class, criterio);
//Consulta sobre los jugadores cuyo nombre es Maria
IQuery query = new CriteriaQuery(Jugadores.class, Where.equal("nombre", "Maria"));
//recuperar todos los jugadores de la consulta
Objects <Jugadores> jugadores = odb.getObjects(query);
```

Para comprobar si un array o una colección contiene un valor determinado usamos `Where.contain()`:

```
ICriterion criterio = Where.contain("nombreakarray", valor);
```

Para comprobar si un atributo es nulo usamos `Where.isNull()`:

```
ICriterion criterio = Where.isNull("atributo");
```

Para comprobar si un atributo no es nulo usamos `Where.isNotNull()`:

```
ICriterion criterio = Where.isNotNull("atributo");
```

Se puede añadir complejidad al criterio de la consulta mediante expresiones lógicas. Las clases **And**, **Or** y **Not** proporcionan mecanismos de construcción de expresiones compuestas. Necesitaremos añadir los paquetes:

```
import org.neodatis.odbc.core.query.criteria.And;
import org.neodatis.odbc.core.query.criteria.Or;
import org.neodatis.odbc.core.query.criteria.Not;
```

Para consultas dobles, o de varios campos, tienes a tu disposición los métodos `and()` y `or()`, dependiendo de la necesidad de tu condición.

Podemos añadir mediante el método `add()` a los criterios de búsqueda. Por ejemplo, para obtener los jugadores de Madrid y edad 15 escribimos el siguiente criterio **AND**:

```
// Jugadores de 15 años de Madrid
ICriterion criterio = new And().add(Where.equal("ciudad", "Madrid"))
    .add(Where.equal("edad", 15));
```

Para obtener los jugadores cuya ciudad sea Madrid o la edad sea \geq que 15 construimos el criterio **OR**:

```
// Jugadores de 15 años o de Madrid
ICriterion criterio = new Or().add(Where.equal("ciudad", "Madrid"))
                        .add(Where.equal("edad", 15));
```

En el caso de querer obtener los jugadores cuyo nombre no empiece por la letra M usamos **Where.not()**:

```
// Jugadores que no empiezan por M
ICriterion criterio = Where.not(Where.like("nombre", "M%"));
```

O también se puede poner:

```
ICriterion criterio1 = Where.like("nombre", "M%");
ICriterion criterio = Where.not(criterio1);
```

Más información:

<http://neodatis.wikidot.com/criteria-queries>

Los siguientes son ejemplos de criterios de búsqueda más complejos con expresiones lógicas:

```
// Jugadores de 15 años de Madrid
ICriterion criterio = new And().add(Where.equal("ciudad", "Madrid")).add(Where.equal("edad", 15));

// Jugadores >= de 15 años y de Madrid
ICriterion criterio2 = new And().add(Where.equal("ciudad", "Madrid")).add(Where.ge("edad", 15));

// Jugadores de 15 años o de Madrid
ICriterion criterio = new Or().add(Where.equal("ciudad", "Madrid"))
                              .add(Where.equal("edad", 15));

// Jugadores que no empiezan por M
ICriterion criterio = Where.not(Where.like("nombre", "M%"));
```

Ejemplos

Dadas la clase *País* y la clase modificada *Jugador*:

Vamos a crear una nueva clase, la clase *Países* con dos atributos y sus *getter* y *setter*.

Los atributos son: `private int id;` `private String nombrepais;`

En la clase anterior de *Jugadores* debemos añadir el siguiente atributo con sus *getter* y *setter*:

```
private Países pais;
```

Nuestra nueva base de datos se llamará `EQUIPOS.DB` en donde insertaremos países y jugadores de esos países; finalmente los visualizaremos.

```
public class Países {
    private int id;
    private String nombrepais;

    public int getId() {return id;}

    public void setId(int id) {this.id = id;}

    public Países() { }

    public Países(String nombrepais, int id){
        this.nombrepais = nombrepais;
        this.id = id;
    }

    public String getNombrepais() {return this.nombrepais;}

    public void setNombrepais(String nombrepais) {
        this.nombrepais = nombrepais;
    }
    public String toString() { return nombrepais; }
}

public class Jugadores {
    private String nombre;
    private String deporte;
    private String ciudad;
    private int edad;
    private Países pais;

    public Jugadores() {
    }

    public Jugadores(String nombre, String deporte, String ciudad, int edad, Países pais) {
        this.nombre = nombre;
        this.deporte = deporte;
        this.ciudad = ciudad;
        this.edad = edad;
        this.pais = pais;
    }

    public Países getPais() {
        return pais;
    }

    public void setPais(Países pais) {
        this.pais = pais;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    public void setDeporte(String deporte) {
        this.deporte = deporte;
    }

    public String getDeporte() {
        return deporte;
    }

    public void setCiudad(String ciudad) {
        this.ciudad = ciudad;
    }

    public String getCiudad() {
        return ciudad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public int getEdad() {
        return edad;
    }
}
```

Método para visualizar los jugadores de 14 años de los países de IRLANDA, FRANCIA e ITALIA.

```
private static void jugadores14irlandafranciaitalia() {
    ODB odb = ODBFactory.open("EQUIPOS.DB");
    ICriterion criterio = new And().add(Where.equal("edad", 14))
        .add(new Or().add(Where.equal("pais.nombrepais", "IRLANDA"))
            .add(Where.equal("pais.nombrepais", "ITALIA")).add(Where.equal("pais.nombrepais", "FRANCIA")));
    IQuery query = new CriteriaQuery(Jugadores.class, criterio);
    Objects jugadores = odb.getObjects(query);
    if (jugadores.size() == 0) {
        System.out.println(" No existen jugadores de 14 años de  IRLANDA, ITALIA, FRANCIA.");
    } else {
        Jugadores jug;
        System.out.println("Jugadores de 14 años de IRLANDA, ITALIA, FRANCIA.");
        while (jugadores.hasNext()) {
            jug = (Jugadores) jugadores.next();
            System.out.printf("Nombre: %s, Edad: %d, Ciudad: %s, Pais: %s\n", jug.getNombre(), jug.getEdad(),
                jug.getCiudad(), jug.getPais().getNombrepais());
        }
    }
    odb.close();
}
```

Método que recibe el nombre de un país y actualiza las edades de los jugadores de ese país. Suma 2 a la edad. Si no hay jugadores del país visualiza un mensaje indicándolo.

```
private static void actualizaredadjugadoresdepais(String pais) {
    ODB odb = ODBFactory.open("EQUIPOS.DB");
    ICriterion crit = Where.equal("pais.nombrepais", pais);
    IQuery query = new CriteriaQuery(Jugadores.class, crit);
    Objects jugadores = odb.getObjects(query);
    if (jugadores.size() == 0) {
        System.out.printf(" No existen Jugadores de %s, no se actualiza la edad %n", pais);
    } else {
        Jugadores jug;
        System.out.printf("ACTUALIZAMOS LA EDAD DE LOS JUGADORES DE %s\n", pais);
        while (jugadores.hasNext()) {
            jug = (Jugadores) jugadores.next();
            int edad = jug.getEdad() + 2;
            System.out.printf("%s %d, NUEVA: %d %n", jug.getNombre(), jug.getEdad(), edad);
            jug.setEdad(edad);
            odb.store(jug);
        }
        odb.commit();
    }
    odb.close();
}
```

-Método que reciba el nombre de país y un deporte y visualice los jugadores que son de ese país y practican ese deporte. Si no hay ninguno visualizar que no hay jugadores.

-Método que reciba el nombre de país y lo borre de la BD de Neodatis. Compueba antes de borrar si tiene jugadores. En caso de tener jugadores asigna *null* al país de esos jugadores.

3.4. CONSULTAS MÁS COMPLEJAS CON FUNCIONES

La API **Object Values** de NeoDatis ODB rompe el paradigma de objetos para proporcionar acceso directo a los valores de los atributos de los objetos, utilizar funciones agregadas sobre colecciones y agrupar los resultados. Concretamente, esta API suministra:

- Acceso directo a los valores de los atributos de los objetos.
- Vistas dinámicas que permiten la navegación a través de relaciones.
- Funciones agregadas sobre grupos (SUM, AVG, MIN, MAX, COUNT).
- Funciones personalizadas por el usuario.
- Agrupamientos de resultados (GROUP BY).

Más información: <http://neodatis-odb.wikidot.com/object-values-api>

La API Object Values funciona como una capa de consulta, no cambia nada en el modelo de objetos ni impone restricciones sobre ellos. Tampoco requiere un mapeo específico.

Por lo tanto, para utilizar agrupamientos GROUP BY y las funciones de grupo SUM, AVG, MIN, MAX, COUNT usamos la API **Object Values** de Neodatis que provee acceso a los atributos de los objetos y también se puede usar para recuperar objetos.

La interfaz **ObjectValues** se utiliza para contener el resultado de una consulta sobre los atributos de un objeto y dispone de los siguientes métodos:

- **getByAlias(alias)**. Obtiene el valor de un atributo de un objeto recuperado de una consulta. El atributo se especifica mediante un alias que, por defecto, es su nombre. Se puede definir un alias, si se define un alias a un atributo con `field("nombre", "n")` ;, entonces el valor del atributo se puede recuperar con `getByAlias("n")` ;.
- **getByIndex(índice)**. Obtiene el valor de un atributo de un objeto recuperado de una consulta. El atributo se especifica mediante su posición dentro del objeto devuelto: 0, 1, 2...

El siguiente programa Java realiza una consulta sobre el nombre y la ciudad de todos los jugadores:

```
import java.math.BigDecimal;
import java.math.BigInteger;
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.Values;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;

public class EjemploObjectValues {
    public static void main(String[] args) {

        ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
        //Consulta sobre el nombre y la ciudad de todos los jugadores
        Values valores = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).field("nombre").field("ciudad"));
        //visualizar los valores recuperados de la consulta
        while (valores.hasNext()) {
            ObjectValues objectValues = (ObjectValues) valores.next();
            System.out.printf("%s, Ciudad : %s %n",
                            objectValues.getByAlias("nombre"),
                            objectValues.getByIndex(1));
        }

        //cerrar la base de datos
        odb.close();
    }
}
```

Cada objeto es un ObjectValues que da acceso a los campos para finalmente poder mostrar el nombre del jugador y el nombre de la ciudad. Recuperamos el nombre del jugador por alias y el nombre de la ciudad por índice.

Para trabajar con la API **Object Values** será necesario importar algunos paquetes:

```
import java.math.BigDecimal;
import java.math.BigInteger;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.Values;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;
```

Una **función agregada** es una función que realiza un cálculo sobre un conjunto de valores, en lugar de sobre un único valor. Se soportan las siguientes funciones:

- **sum(atributo)**. Calcula la suma de todos los valores del atributo que satisfacen la consulta.
- **avg(atributo)**. Calcula el promedio de todos los valores del atributo que satisfacen la consulta.
- **count(alias)**. Cuenta el número de objetos que satisfacen la consulta.

- **min(atributo).** Recupera el valor mínimo de todos los valores del atributo que satisfacen la consulta.
- **max(atributo).** Recupera el valor máximo de todos los valores del atributo que satisfacen la consulta.
- **groupBy(atributo).** Ejecuta un agrupamiento del resultado de la consulta por el atributo.
- **sublist(atributo, índice, tamaño).** Devuelve una sublista de un atributo de tipo lista.
- **size(atributo).** Recupera el tamaño de una colección de objetos, sin cargar dichos objetos en memoria. Solo es aplicable a atributos que son de tipo colección (lista, vector, conjunto, bolsa).

Los siguientes son ejemplos de uso de varias funciones agregadas sobre un conjunto de valores. Pondremos comentada la sentencia SQL a la que equivaldrían las líneas de código propuestas:

```
// SUMA - Obtiene la suma de las edades
// SELECT SUM(edad) FROM Jugadores
Values val = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).sum("edad"));
ObjectValues ov = val.nextValues();
BigDecimal value = (BigDecimal) ov.getByAlias("edad");
System.out.printf("Suma de edad : %d %n", value.longValue());

System.out.println("-----");
System.out.printf("Suma de edad : %.2f %n", ov.getByAlias("edad"));
```

Si no se opera con los valores, sino que solo se visualiza, no es necesario convertirlos a *BigDecimal*. Ejemplos:

```
// CUENTA - Obtiene el número de jugadores
// SELECT COUNT(nombre) FROM Jugadores
Values val2 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).count("nombre"));
ObjectValues ov2 = val2.nextValues();
BigInteger value2 = (BigInteger) ov2.getByAlias("nombre");
System.out.printf("Numero de jugadores : %d %n", value2.intValue());

System.out.println("-----");
// MEDIA - Obtiene la edad media de los jugadores
// SELECT AVG(edad) FROM Jugadores
Values val3 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).avg("edad"));
ObjectValues ov3 = val3.nextValues();
BigDecimal value3 = (BigDecimal) ov3.getByAlias("edad");
System.out.printf("Edad media : %.2f %n", value3.floatValue());

System.out.println("-----");
// MAXIMO Y MINIMO - Obtiene la edad máxima y la edad mínima
// SELECT MAX(edad) edad_max , MIN(edad) edad_min FROM Jugadores
Values val4 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).max("edad", "edad_max"));
ObjectValues ov4 = val4.nextValues();
BigDecimal maxima = (BigDecimal) ov4.getByAlias("edad_max");

Values val5 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).min("edad", "edad_min"));
ObjectValues ov5 = val5.nextValues();

BigDecimal minima = (BigDecimal) ov5.getByAlias("edad_min");
System.out.printf("Edad máxima: %d, Edad mínima: %d %n", maxima.intValue(), minima.intValue());
```


Si la media sale redondeada se ejecuta bien la función `avg`, pero si la media no sale redondeada hay que capturar la excepción **ArithmeticException**, y hacer la media con la suma y el contador. Este ejemplo realiza un método para calcular la media de edad capturando la excepción:

```
private static void visualizarmediadeedad() {
    ODB odb = ODBFactory.open("EQUIPOS.DB");
    Values val;
    ObjectValues ov;
    try {
        val = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).avg("edad"));
        ov = val.nextValues();
        System.out.printf("AVG-La media de edad es: %f %n", ov.getByIndex(0));
    } catch (ArithmeticException e) {
        System.out.println(e.getMessage());

        Values val2 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).sum("edad").count("edad"));
        ObjectValues ov2 = val2.nextValues();
        float media;
        BigDecimal sumaedad = (BigDecimal) ov2.getByIndex(0);
        BigInteger cuenta = (BigInteger) ov2.getByIndex(1);
        media = sumaedad.floatValue() / cuenta.floatValue();

        System.out.printf("La media de edad es: %.2f Contador = %d "
            + "suma = %.2f %n", media, cuenta, sumaedad);
    }
    odb.close();
}
```

El siguiente ejemplo muestra un **agrupamiento** (GROUP BY), obtenemos por cada ciudad el número de jugadores:

```
// GROUP BY
Values groupby = odb
    .getValues(new ValuesCriteriaQuery(Jugadores.class)
        .field("ciudad").count("nombre").groupBy("ciudad"));

while (groupby.hasNext()) {
    ObjectValues objetos = (ObjectValues) groupby.next();
    System.out.printf("%s, %d%n", objetos.getByAlias("ciudad"), objetos.getByIndex(1));
}
```

En SQL la consulta sería *SELECT ciudad, count(nombre) FROM Jugadores GROUP BY ciudad.*

Cuando un objeto está relacionado con otro, por ejemplo un jugador está relacionado con su país, podremos acceder directamente a la información del objeto relacionado para obtener la información necesaria gracias a la API Object Values.

La API Object Values de NeoDatis ODB admite la posibilidad de navegar directamente usando las relaciones entre los objetos para obtener la información requerida. Esta característica recibe el nombre de **vistas dinámicas** y proporciona la misma funcionalidad que las sentencias SQL con *join* entre tablas para las relaciones semánticas.

Para usar una vista dinámica, en lugar de especificar el nombre de un atributo, se precisa el nombre completo de la relación para alcanzar el atributo.

Dadas la clase *País* y la clase modificada *Jugador* que hemos utilizado anteriormente.

Los siguientes son ejemplos de uso de vistas dinámicas que permiten la navegación entre relaciones de objetos.

Vamos a obtener el nombre, edad y país del jugador; en SQL sería como realizar la siguiente consulta: *SELECT nombre, edad, nombrepais FROM Jugadores j, Paises p WHERE j.id = p.d;*

```
private static void JugadoresPaíses() {
    ODB odb = ODBFactory.open("EQUIPOS.DB");
    Values valores = odb.getValues(
        new ValuesCriteriaQuery(Jugadores.class)
            .field("nombre")
            .field("edad")
            .field("pais.nombrepais"));

    while (valores.hasNext()) {
        ObjectValues objectValues = (ObjectValues) valores.next();
        System.out.printf("Nombre: %s, Edad: %d, Pais: %s %n",
            objectValues.getByAlias("nombre"),
            objectValues.getByIndex(1),
            objectValues.getByIndex(2));
    }
    odb.close();
}
```

Para obtener el nombre del país en el método **field()** escribimos la relación completa, el atributo de la clase Jugadores, un punto y a continuación el atributo de la clase Países: **.field("pais.nombrepais")**;

Para obtener el nombre y la ciudad de los jugadores cuyo país es ESPAÑA y edad igual a 15 escribimos:

```
Values valores = odb.getValues(new ValuesCriteriaQuery(
    Jugadores.class,
    new And().add(Where.equal("pais.nombrepais", "ESPAÑA"))
        .add(Where.equal("edad", 15))
    )
    .field("nombre")
    .field("ciudad"));
```

En el siguiente método visualizamos el número de jugadores, la edad máxima y la edad media por cada país. Para calcular la edad media utilizaremos la suma y el contador, así evitamos el error de la media redondeada:

```
private static void contadorymediaporpais() {
    ODB odb = ODBFactory.open("EQUIPOS.DB");
    System.out.println("Numero de jugadores por país, "+
        " max de edad y media de edad: ");
    Values groupby = odb.getValues(new ValuesCriteriaQuery(
        Jugadores.class, Where.isNotNull("pais.nombrepais"))
        .field("pais.nombrepais").count("nombre")
        .max("edad").sum("edad").groupBy("pais.nombrepais") );
    if (groupby.size() == 0)
        System.out.println( " La consulta no devuelve datos. ");
    else
    {
        while(groupby.hasNext()) {
            ObjectValues objetos= (ObjectValues) groupby.next();
            float media = ((BigDecimal) objetos.getByIndex(3)).floatValue() /
                ((BigInteger) objetos.getByIndex(1)).floatValue();

            System.out.printf("Pais: %-8s Num jugadores: %d, Edad Máxima: %.0f, "
                + "Suma de Edad: %.0f, Edad media: %.2f %n",
                objetos.getByAlias("pais.nombrepais"),
                objetos.getByIndex(1),
                objetos.getByIndex(2),
                objetos.getByIndex(3), media );
        }
    }
    odb.close();
} //
```