UD4. Parte 2. BBDD orientadas a objetos

Contenido

1. BASES DE DATOS ORIENTADAS A OBJETO

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 que almacena objetos.

2. 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 (*ObjectIdentifier*), 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) **Polimorfismo.** Los métodos se deben poder aplicar a diferentes tipos en tiempo de ejecución (ligadura dinámica). El método que se ejcute dependerá del tipo de objeto al que se aplique.
7) **Completitud 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 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, que se está convirtiendo en un estándar de facto.
- Competencia con los SGBD-R y los SGBD-OR, que tienen gran experiencia de uso.
- 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.

3. EL ESTÁNDAR ODMG

ODMG (Object Data Management Group) es un consorcio industrial de fabricantes de sistemas gestores de bases de datos orientadas a objetos y de herramientas de mapeo objeto-relacional. No es una organización de estándares acreditada, pero tiene mucha influencia con respecto a los estándares sobre los SGBD-OO.

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.

Entre 1993 y 2001, publicó cinco revisiones de su estándar. La última revisión fue **ODMG versión 3.0**, tras lo cual el grupo se disolvió. Los principales componentes de este estándar ODMG versión 3.0 son los siguientes:

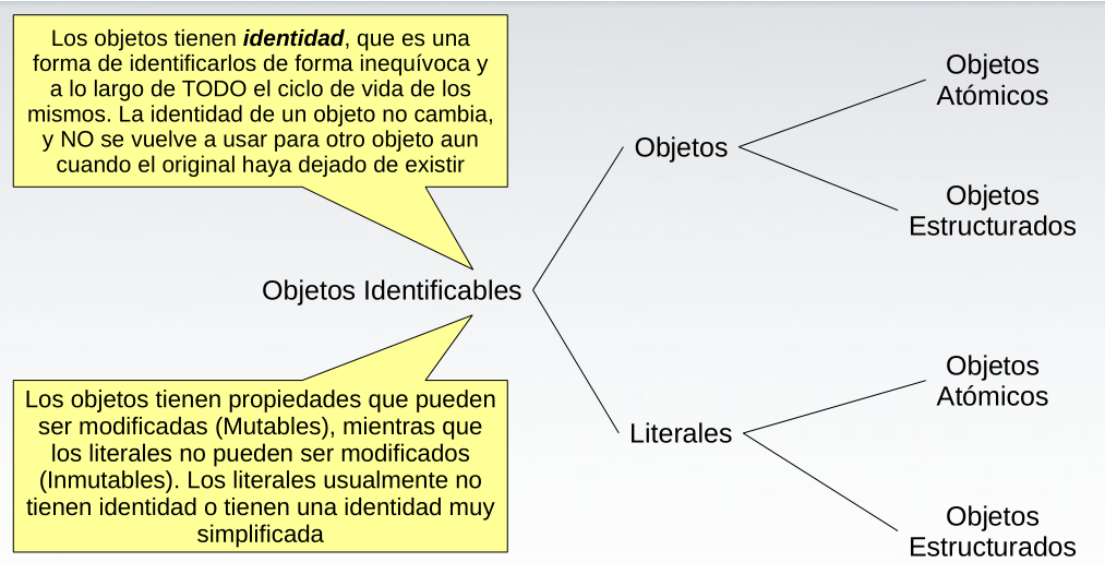
- Modelo de Objetos.
- Lenguaje de Definición de Objetos (ODL).
- Lenguaje de Consultas de Objetos (OQL).
- Conexión con el Lenguaje C++.
- Conexión con el Lenguaje Smalltalk.
- Conexión con el Lenguaje Java.

3.1. Modelo de Objetos

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

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 **desordenada** de objetos del mismo tipo que **no** admite **duplicados**.
  - *bag<tipo>.* Colección **desordenada** 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 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).

3.2. Lenguaje de Definición de Objetos (ODL)

Object Definition Language (ODL) es un lenguaje para definir la especificación de los tipos de objetos en sistemas compatibles con ODMG. ODL es el equivalente al DDL (lenguaje de definición de datos) de los SGBD-R. 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.

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*:

```
class Cliente (extent Clientes key NIF){
       /* definición de atributos */
       attribute string NIF;
       attribute struct NombrePersona {
              string apellidos,
              string nombrePila
       } nombre;
       attribute date fechaNacimiento;
       attribute enum Genero {
              Hombre,
              Mujer
       } sexo;
       Attribute struct DireccionCompleta {
              String calle,
              String población,
              String provincia
       } direccion;
       attribute set<string>telefonos;
       /* definición de operaciones */
       short calcularEdad();
}
class Producto (extent Productos key IDPRODUCTO)
{
       /* definición de atributos */
       attribute short IDPRODUCTO;
       attribute string descripcion;
       attribute float pvp;
       attribute short stockMinimo;
       attribute short stockActual;
}
class LineaVenta (extent LineasVentas)
{
       /* definición de atributos */
       attribute short numeroLinea;
       attribute Producto product;
       attribute short cantidad;
       /* definición de operaciones */
       float calcularImporte();
}
class Venta (extent Ventas key IDVENTA)
{
       /* definición de atributos */
       attribute short IDVENTA;
       attribute date fechaVenta;
       attribute set<LineaVenta>lineas;
       /* definición de relaciones */
       relationship Cliente perteneceACliente inverse
                    Cliente::tieneVenta;
       /* definición de operaciones */
       float calcularTotalVenta();
}
```

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.

3.3. Lenguaje de Consultas de Objetos (OQL)

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

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:

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

Para acceder a los atributos y objetos relacionados, se utilizan **expresiones de camino**, que empiezan 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.nombrePila FROM Clientes c WHERE c.sexo = "Mujer";
SELECT c.nombre.nombrePila FROM c IN Clientes WHERE c.sexo = "Mujer";
SELECT c.nombre.nombrePila FROM Clientes AS c WHERE c.sexo = "Mujer";
```

En general, si `v` es una variable cuyo tipo es *Venta*:

- `v.IDVENTA` es el identificador de venta del objeto v.
- `v.fechaVenta` es la fecha de venta del objeto v.
- `v.calcularTotalVenta()` obtiene el total de venta del objeto v.
- `v.perteneceACliente` es un puntero al cliente mencionado en v.
- `v.perteneceACliente.direccion` es la dirección del cliente mencionado en v.
- `v.lineas` es una colección de objetos del tipo *LineaVenta*. El uso de `v.lineas.numeroLinea` 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 ID de venta sea 1.

```
SELECT v.perteneceACliente.nombre, v.perteneceACliente.direccion,
v.fechaVenta, v.calcularTotalVenta()
FROM Ventas v
WHERE v.IDVENTA = 1;
```

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

```
SELECT lv.numeroLinea, lv.product.descripcion,
lv.cantidad, lv.calcularImporte()
FROM Ventas v, v.lineas lv
WHERE v.IDVENTA = 1;
```

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

Para obtener como resultado una colección que no admita duplicados, se usa DISTINCT en la sentencia SELECT:

```
SELECT DISTINCT c.nombre.nombrePila
FROM c IN clientes
WHERE c.sexo = "Mujer";
```

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

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

3.3.1. 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 siguientes operadores:

- **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.

3.3.2. 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 colección : condición`
- Alguno/Cualquier miembro: `colección comparación SOME/ANY condición`
  (comparación puede ser: <, <=, =, >=, >)
- Al menos 1 miembro: `EXISTS x IN colección : condición`
  `EXISTS x`
- Sólo 1 miembro: `UNIQUE x`

Los operadores **AVG**, **SUM**, **MIN**, **MAX** y **COUNT** se pueden aplicar a cualquier colección, siempre y cuando tengan sentido para el tipo de elemento.

Ejemplos:

1) Obtener los datos de las ventas de los clientes de la población de Toledo y cuyos apellidos empiecen por la letra A.

```
SELECT v.IDVENTA, v.fechaVenta,v.calcularTotalVenta()
FROM Ventas v
WHERE(v.tieneVenta.direccion.poblacion = "Toledo"
      AND v.tieneVenta.nombre.apellidos LIKE "A%");
```

2) Obtener para el ID de venta 1 aquellas líneas de venta cuya descripción del producto contenga el carácter P.

```
SELECT lv.numeroLinea, lv.product.descripcion,
lv.cantidad, lv.calcularImporte()
```

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

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

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

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

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

4. 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:http://www.ignitetech.com/solutions/for-information-technology/objectstore-standard-edition/
  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/

5. NEODATIS ODB

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

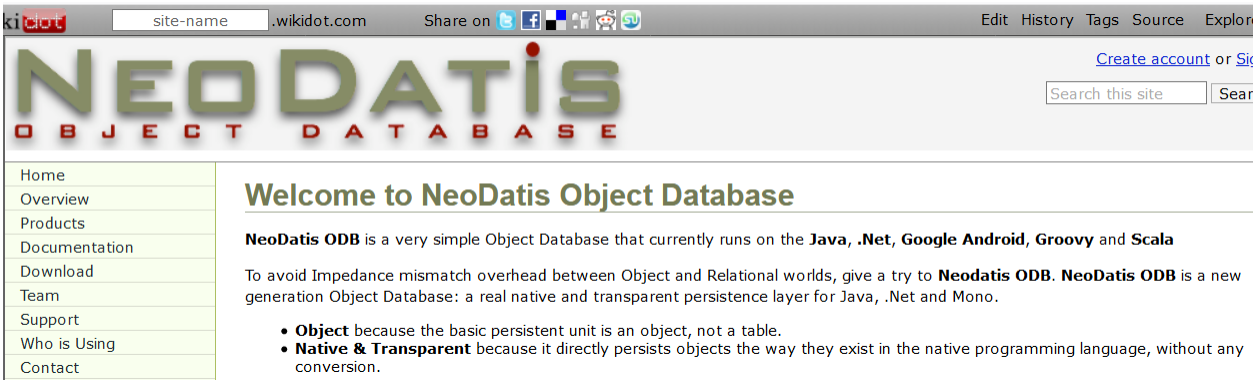
Con NeoDatis ODB, se evita la falta de impedancia entre los paradigmas orientado a objetos y relacional, ya que actúa como una capa de persistencia nativa y transparente para Java, .NET y Mono. Los objetos se pueden almacenar y recuperar con una sola línea de código, sin necesidad de tener que mapearlos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

Sus principales características son las siguientes:

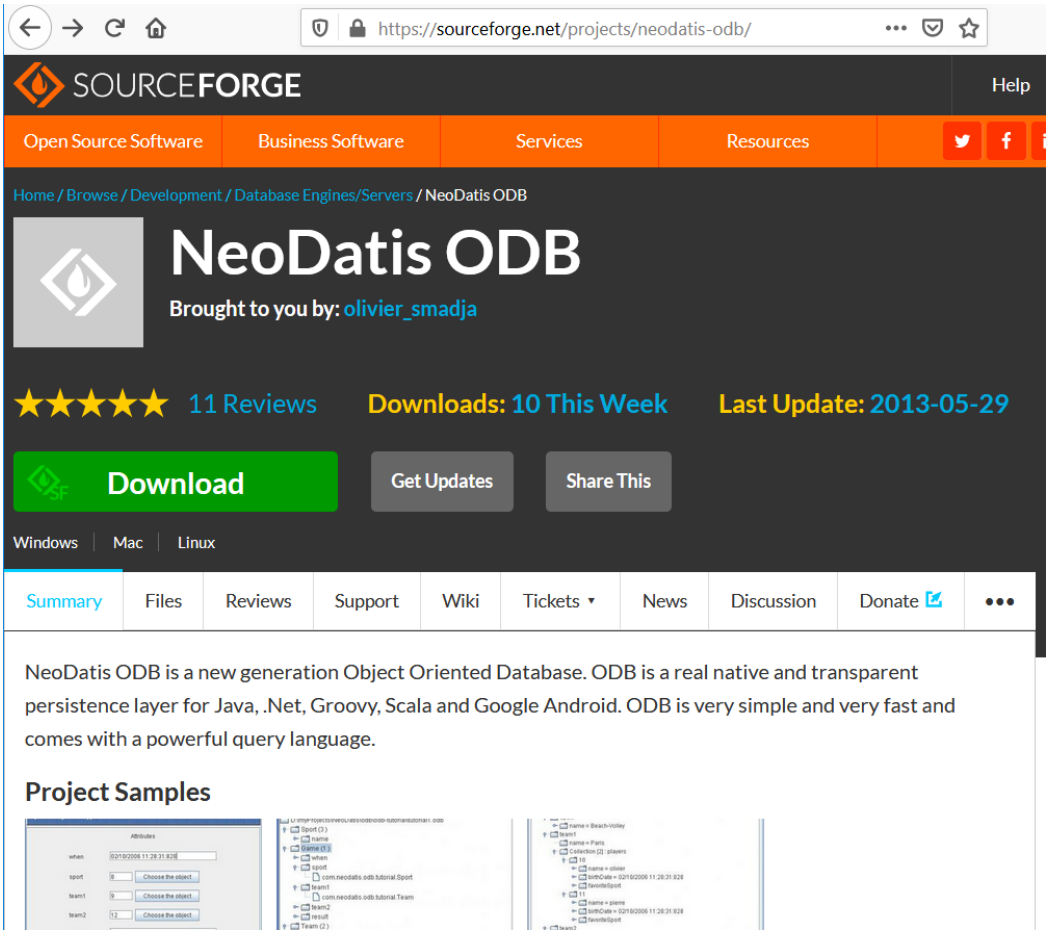
- Es muy simple e intuitiva, con un tiempo de aprendizaje mínimo.
- Su motor de bases de datos ocupa menos de 800 KB y se distribuye como un único fichero JAR/DLL que puede empaquetarse fácilmente en cualquier aplicación.
- Es rápida. Puede almacenar más de 30000 objetos en un segundo.
- Soporta transacciones 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, Scala y Android.
- 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.

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**).



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, la carpeta */doc/javadoc* contiene la documentación de la API de NeoDatis ODB.

5.1. 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.

```
package clases;

//Clase Jugador
public class Jugador {
	private String nombre;
	private String deporte;
	private String ciudad;
	private int edad;

	public Jugador() {
	}

	public Jugador(String nombre, String deporte, String ciudad, int edad) {
		this.nombre = nombre;
		this.deporte = deporte;
		this.ciudad = ciudad;
		this.edad = edad;

	}

	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.

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

import clases.Jugador;

public class EjemploNeodatis {

	public static void main(String[] args) {

		// Crear instancias de objetos de clase Jugador
		Jugador j1 = new Jugador("Maria", "voleibol", "Madrid", 14);
		Jugador j2 = new Jugador("Miguel", "tenis", "Madrid", 15);
		Jugador j3 = new Jugador("Mario", "baloncesto", "Guadalajara", 15);
		Jugador j4 = new Jugador("Alicia", "tenis", "Madrid", 14);
		// Abrir BD
		ODB odb = ODBFactory.open("neodatis.test");

		// Almacenamos los jugadores en la base de datos
		odb.store(j1);
		odb.store(j2);
		odb.store(j3);
		odb.store(j4);

		//recuperamos todos los jugadores
		// OJO. Objects no es la clase Object
		// Objects implementa la inteface Collection
		Objects<Jugador> objects = odb.getObjects(Jugador.class);
		System.out.printf("%d Jugadores: %n", objects.size());

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

}
```

La ejecución muestra:

```
4 Jugadores:
1: Maria, voleibol, Madrid
2: Miguel, tenis, Madrid
3: Mario, baloncesto, Guadalajara
4: Alicia, tenis, Madrid
```

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 (ObjectIdentifier). El siguiente programa Java muestra los datos del objeto cuyo OID es 3:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.OID;
import org.neodatis.odb.core.oid.OIDFactory;
import clases.Jugador;

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
      Jugador jug = (Jugador) odb.getObjectFromId(oid);
      System.out.printf("%s, %s, %s, %d %n",
         jug.getNombre(), jug.getDeporte(), jug.getCiudad(), jug.getEdad());
      odb.close(); // Cerrar BD
   }// main

}
```

La salida en la consola es:

```
Miguel, tenis, Madrid, 15
```

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 j, se puede hacer de dos maneras:

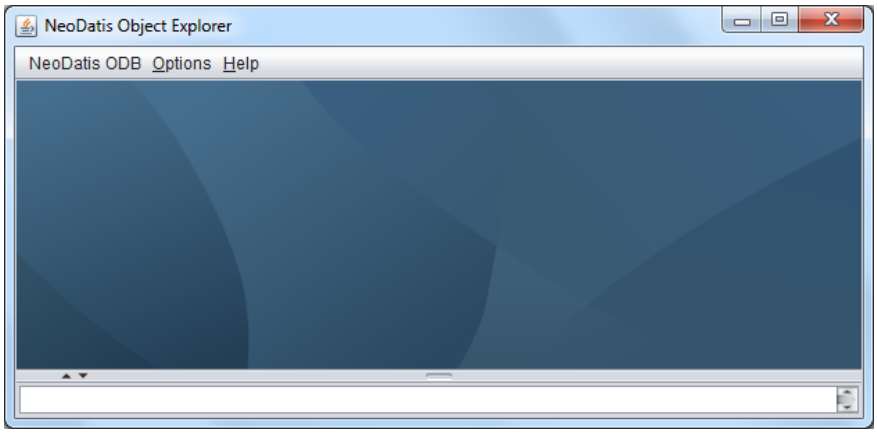
```
OID oid = odb.store(j);
OID oid = odb.getObjectId(j);
```

5.2. Explorador de objetos

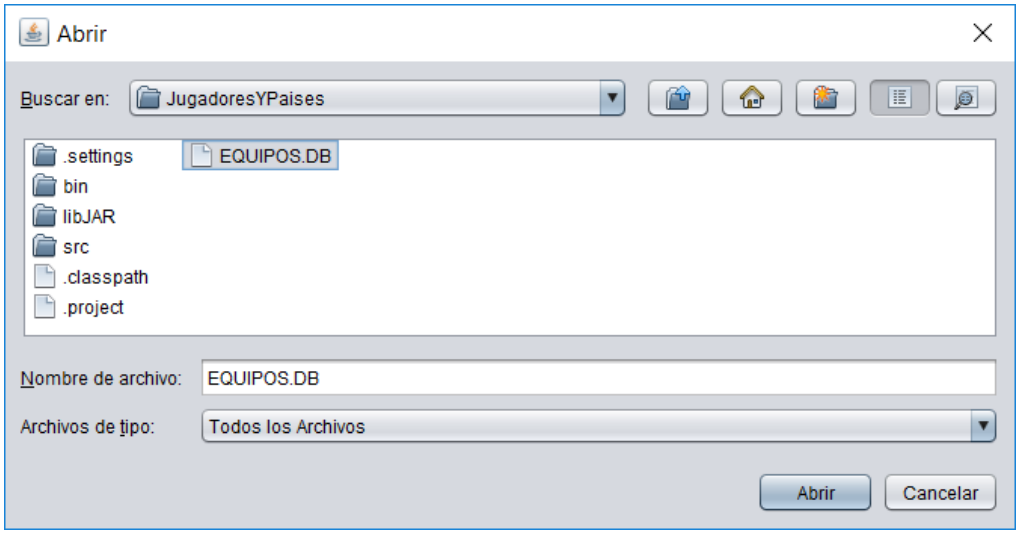
NeoDatisObject Explorer es una herramienta gráfica que viene con la base de datos NeoDatis para gestionar los datos. Esta herramienta permite:

- 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.

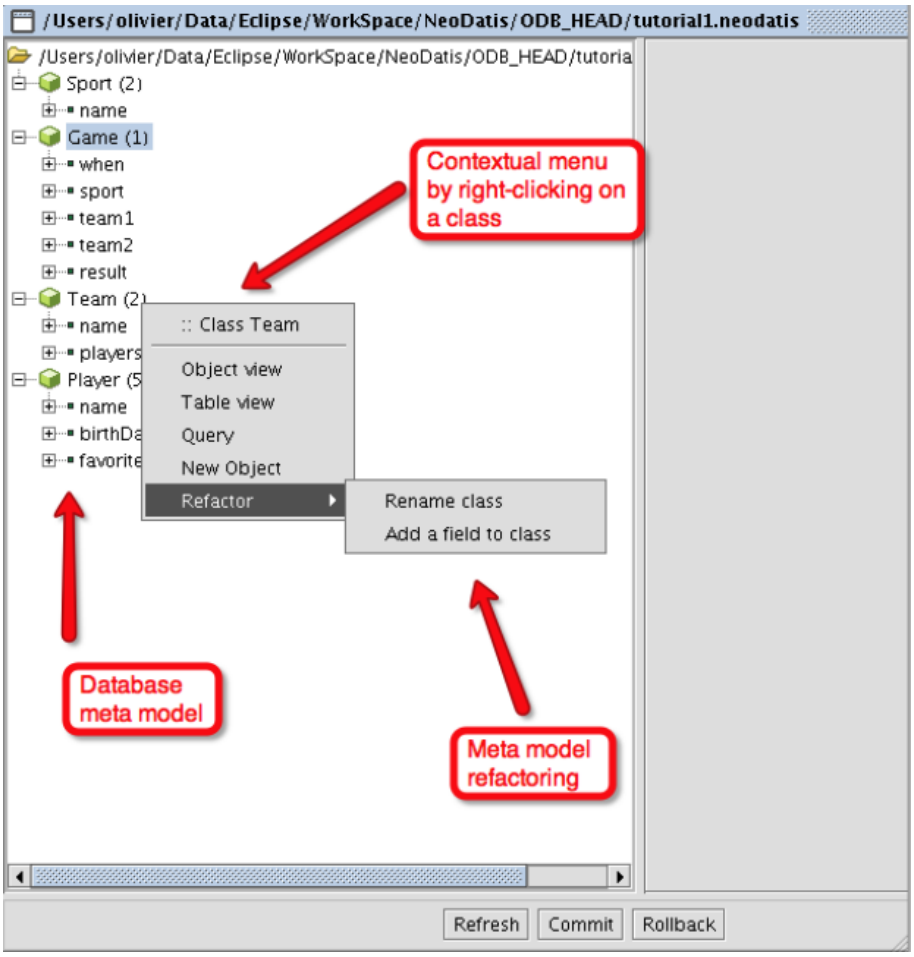
Para ejecutarla, se hace doble clic en el fichero **odb-explorer.bat** (sistemas Windows) u **odb-explorer.sh** (sistemas Linux).



A continuación, es necesario abrir una base de datos para poder navegar por ella (Menú **NeoDatis ODB** ->**Open Database**).



A continuación, el explorador de objetos muestra 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:

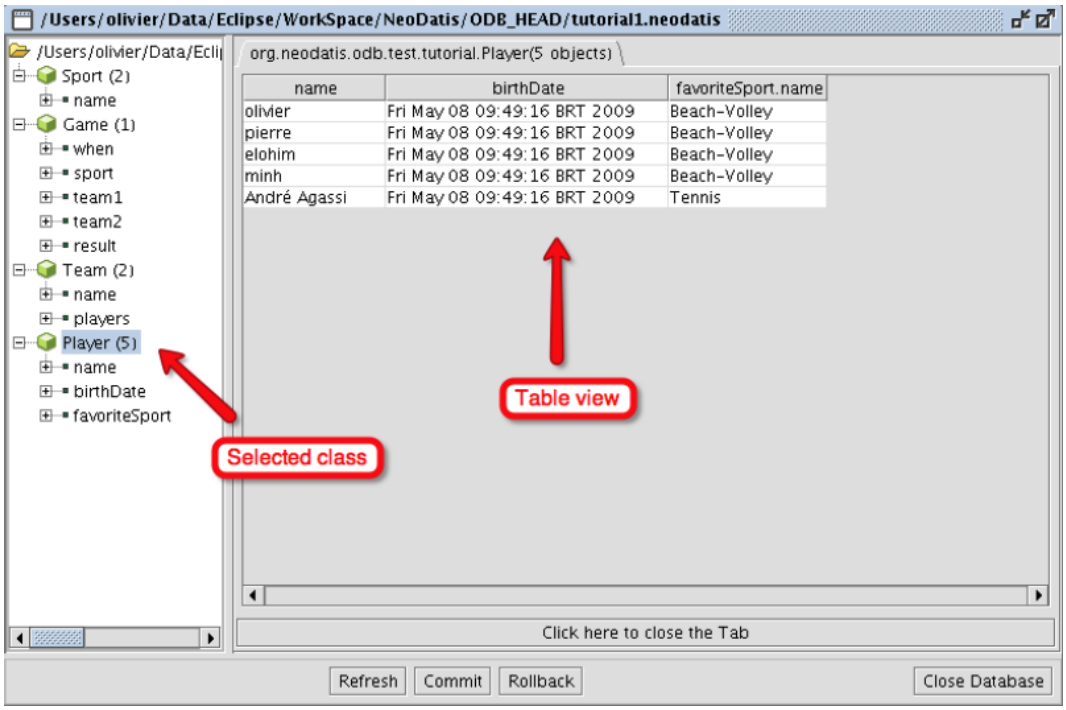


Pulsando sobre una clase aparece un menú contextual:

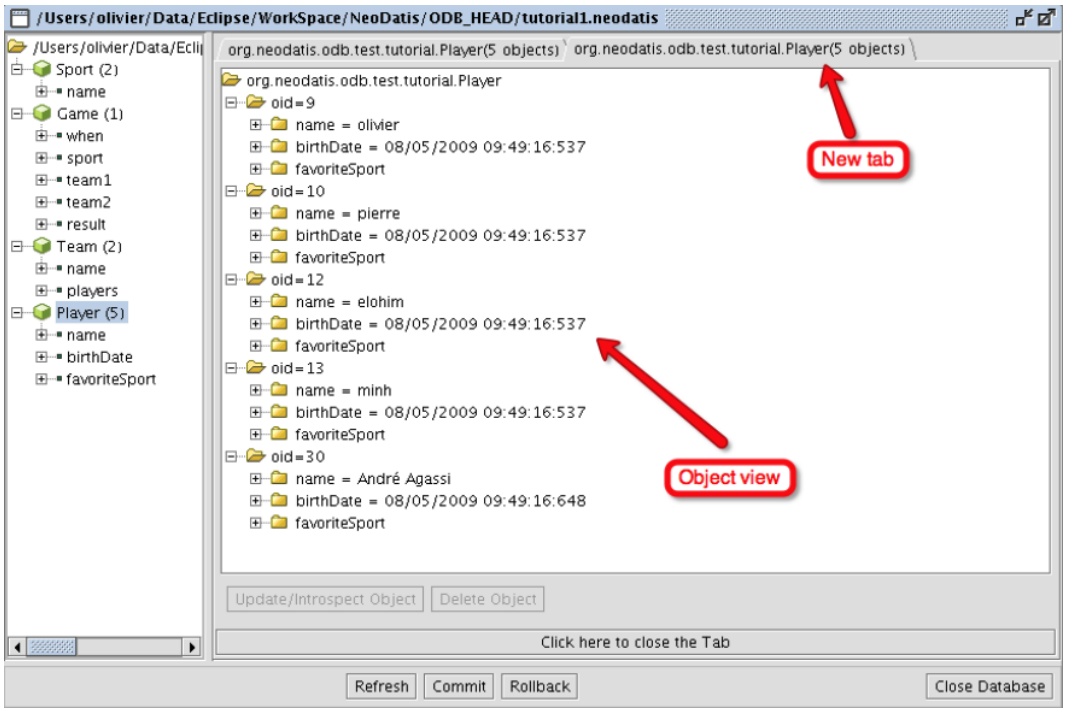
- La opción **Object View (Vista de Objetos)** muestra todos los objetos en un modo jerárquico.
- La opción **Table View (Vista de Tabla)** muestra los datos en un resultado de consulta parecida a SQL.
- La opción **Query (Consulta)** abre un asistente gráfico para realizar consultas con criterios.
- 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:

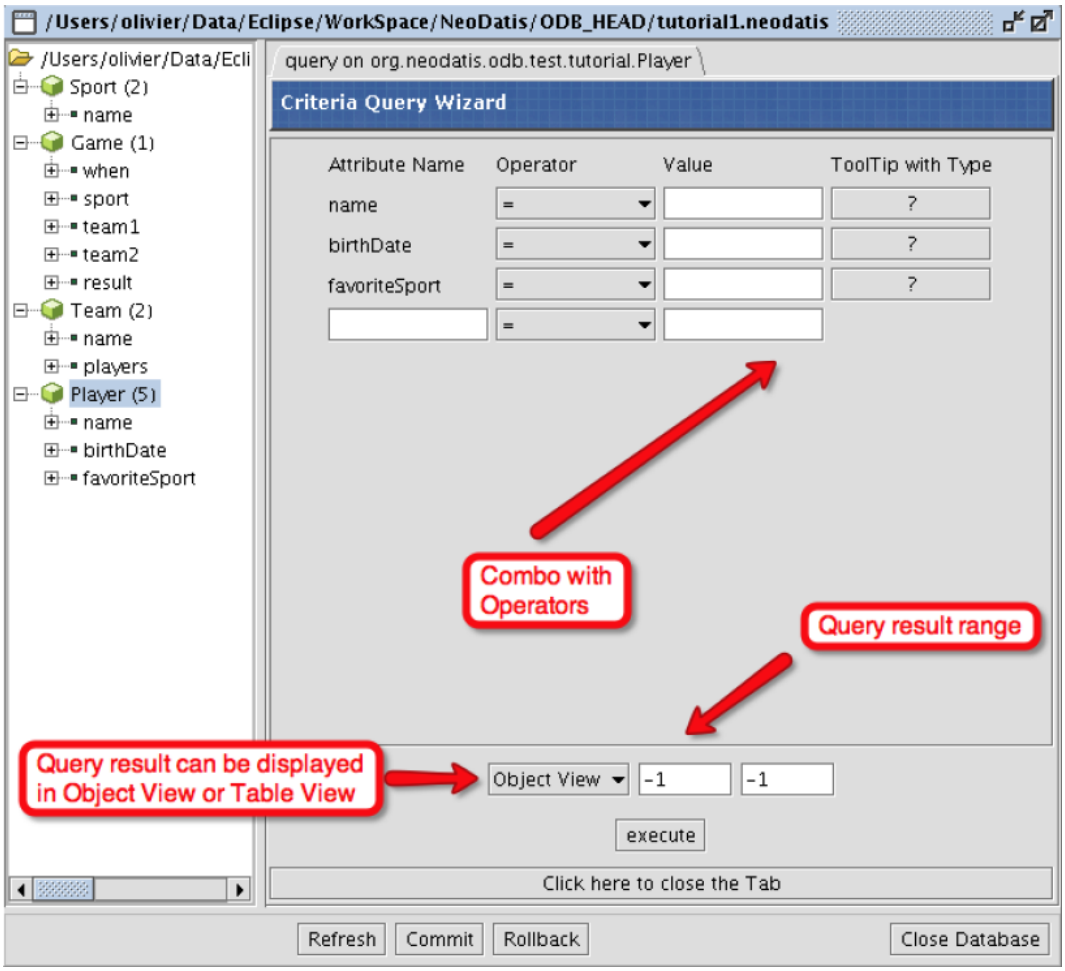
- **Table View (Vista de Tabla).** Muestra los resultados en un resultado de consulta parecida a SQL.



- **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:



5.3. 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.

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:

```
// CONSULTAS CON CRITERIOS EN NEODATIS
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.Objects;

import clases.Jugador;

public class ConsultarJugadoresTenis {

	public static void main(String[] args) {
		// abrir la base de datos
		ODB odb = ODBFactory.open("neodatis.test");
		// consulta sobre los jugadores que practican el deporte tenis
		IQuery query = new CriteriaQuery(Jugador.class, Where.equal("deporte", "tenis"));
		// ordenar ascendentemente el resultado de la consulta por nombre y edad
		query.orderByAsc("nombre,edad");
		// recuperar todos los jugadores de la consulta
		Objects<Jugador> jugadores = odb.getObjects(query);
		System.out.println(jugadores.size() + " jugadores:");
		// visualizar los jugadores recuperados
		int i = 1;
		while (jugadores.hasNext()) {
			Jugador j = jugadores.next();
			System.out.println(i + ") " +
				j.getNombre() + " *** " + j.getDeporte() + " *** " +
				j.getCiudad() + " *** " + j.getEdad());
			i++;
		}
		// cerrar la base de datos
		odb.close();
	}//main
}
```

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

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

import clases.Jugador;

public class ModificarDeporteMaria {
```

```
   public static void main(String[] args) {
      ODB odb = null;
      // abrir la base de datos
      try {
         odb = ODBFactory.open("neodatis.test");
         // consulta sobre los jugadores cuyo nombre es María
         // da error porque hemos puesto el nombre sin acento
         //IQuery query = new CriteriaQuery(Jugador.class, Where.equal("nombre", "María"));

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

         // obtener solamente el primer jugador encontrado en la consulta
         Jugador jugador = jugadores.getFirst();

         // actualizar el jugador
         jugador.setDeporte("voley-playa");
         odb.store(jugador);
      }catch(IndexOutOfBoundsException ex) {
         System.out.println("No hay ningun jugador con ese nombre");
      }catch(Exception e) {
         System.out.println("Error: " + e.getMessage());
      }finally {
         // validar los cambios y cerrar la base de datos
         if(odb!= null) {
            odb.close();
         }
      }
   }//main
}
```

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).

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

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

// jugadores cuya edad es 14
ICriterion criterio = Where.equal("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);
// 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 cuyo nombre empieza por la letra M
ICriterion criterio = Where.like("nombre", "M%");


// consulta sobre los jugadores
IQuery query = new CriteriaQuery(Jugador.class, criterio);
// recuperar todos los jugadores de la consulta
Objects<Jugador> jugadores = odb.getObjects(query);
```

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.

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

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

// jugadores cuya ciudad es Madrid y edad es 15
ICriterion criterio = new And().add(Where.equal("ciudad", "Madrid"))
                               .add(Where.equal("edad", 15));
// jugadores cuya ciudad es Madrid o edad es mayor o igual que 15
ICriterion criterio = new Or().add(Where.equal("ciudad", "Madrid"))
                              .add(Where.ge("edad", 15));
// jugadores cuyo nombre no empiece por la letra M
ICriterion criterio = Where.not(Where.like("nombre", "M%"));

// consulta sobre los jugadores
IQuery query = new CriteriaQuery(Jugador.class, criterio);
// recuperar todos los jugadores de la consulta
Objects<Jugador> jugadores = odb.getObjects(query);
```

5.4. Consultas con funciones

La API **ObjectValues** 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).

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.

El siguiente programa Java obtiene el nombre y la ciudad de todos los jugadores y los muestra en pantalla:

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. 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 org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Values;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.core.query.IValuesQuery;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;

import clases.Jugador;

public class ConsultaNombreYCiudad {

	public static void main(String[] args) {
		// abrir la base de datos
		ODB odb = ODBFactory.open("neodatis.test");
		// consulta sobre el nombre y la ciudad de todos los jugadores
		IValuesQuery valuesQuery = new ValuesCriteriaQuery(Jugador.class)
				.field("nombre").field("ciudad");
		Values values = odb.getValues(valuesQuery);
		// visualizar los valores recuperados de la consulta
		while (values.hasNext()) {
			ObjectValues objectValues = (ObjectValues) values.next();
			System.out.println("Nombre: " +
objectValues.getByAlias("nombre") +
" *** Ciudad: " + objectValues.getByIndex(1));
		}
		// cerrar la base de datos
		odb.close();

	}// main

}
```

Una **función agregada** es una función que realiza un cálculo sobre un conjunto de valores, en lugar de sobre un único valor. La interfaz **IValuesQuery** soporta 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:

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

import clases.Jugador;

public class EjemplosFuncionesAgregadas {

	public static void main(String[] args) {
		// abrir la base de datos
		ODB odb = ODBFactory.open("neodatis.test");

		// consulta del número de jugadores
		// SELECT count(nombre) FROM Jugador
		IValuesQuery valuesQuery = new ValuesCriteriaQuery(Jugador.class)
				.count("nombre");
		Values values = odb.getValues(valuesQuery);
		ObjectValues objectValues = values.nextValues();
		BigInteger count = (BigInteger) objectValues.getByAlias("nombre");
		System.out.println("Número de Jugadores: " + count.intValue());

		// consulta de la suma de las edades de los jugadores
		// SELECT sum(edad) FROM Jugador
		valuesQuery = new ValuesCriteriaQuery(Jugador.class)
				.sum("edad");
		values = odb.getValues(valuesQuery);
		objectValues = values.nextValues();
		BigDecimal sum = (BigDecimal) objectValues.getByAlias("edad");
		System.out.println("Suma de Edades: " + sum.longValue());

		// consulta del promedio de las edades de los jugadores
		// SELECT avg(edad) FROM Jugador
		valuesQuery = new ValuesCriteriaQuery(Jugador.class)
				.avg("edad");
		values = odb.getValues(valuesQuery);
		objectValues = values.nextValues();
		BigDecimal average = (BigDecimal) objectValues.getByAlias("edad");
		System.out.println("Promedio de Edades: " + average.floatValue());

		// consulta del mínimo y del máximo de las edades de los jugadores
		// SELECT min(edad), max(edad) FROM Jugador
		valuesQuery = new ValuesCriteriaQuery(Jugador.class)
				.min("edad", "edad_min")
				.max("edad", "edad_max");
		values = odb.getValues(valuesQuery);
		objectValues = values.nextValues();
		BigDecimal minimum = (BigDecimal) objectValues.getByAlias("edad_min");
		BigDecimal maximum = (BigDecimal) objectValues.getByAlias("edad_max");
		System.out.println("Mínimo de Edades: " + minimum.intValue());
		System.out.println("Máximo de Edades: " + maximum.intValue());
```

El siguiente ejemplo muestra un **agrupamiento** (GROUP BY) del resultado de una consulta:

```
		// consulta del número de jugadores en cada ciudad
		// SELECT ciudad, count(nombre) FROM Jugador GROUP BY ciudad
		IValuesQuery valuesQuery2 = new ValuesCriteriaQuery(Jugador.class)
				.field("ciudad")
```

```
                .count("nombre")
                .groupBy("ciudad");
Values values2 = odb.getValues(valuesQuery2);
while (values2.hasNext()) {
        ObjectValues objectValues2 = (ObjectValues) values2.next();
        System.out.println(objectValues2.getByAlias("ciudad") + " *** " +
                        objectValues2.getByIndex(1));
}
```

La API ObjectValues 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*:

```
package clases;

public class Pais {
	// atributos
	private int id;
	private String nombre;
	private String capital;
	// constructores
	public Pais() {}
	public Pais(int id, String nombre, String capital) {
		this.id = id;
		this.nombre = nombre;
		this.capital = capital;
	}
	// métodos de acceso
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getCapital() {
		return capital;
	}
	public void setCapital(String capital) {
		this.capital = capital;
	}
}
```

```
package clases;

//Clase Jugador
public class Jugador {
	// atributos
	private String nombre;
	private String deporte;
	private String ciudad;
	private int edad;
	private Pais pais;
	// constructores
	public Jugador() {}
	public Jugador(String nombre, String deporte, String ciudad, int edad, Pais
pais) {
		this.nombre = nombre;
		this.deporte = deporte;
		this.ciudad = ciudad;
		this.edad = edad;
		this.pais = pais;
	}
	// métodos de acceso
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getDeporte() {
```

```
		return deporte;
	}
	public void setDeporte(String deporte) {
		this.deporte = deporte;
	}
	public String getCiudad() {
		return ciudad;
	}
	public void setCiudad(String ciudad) {
		this.ciudad = ciudad;
	}
	public int getEdad() {
		return edad;
	}
	public void setEdad(int edad) {
		this.edad = edad;
	}
	public Pais getPais() {
		return pais;
	}
	public void setPais(Pais pais) {
		this.pais = pais;
	}
}
```

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

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Values;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.core.query.IValuesQuery;
import org.neodatis.odb.core.query.criteria.And;
import org.neodatis.odb.core.query.criteria.ICriterion;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;

import clases.*;

public class VistaRelacionesObjetos {

   public static void main(String[] args) {
      // abrir la base de datos
      ODB odb = ODBFactory.open("neodatisv2.test");
      // consulta del nombre, la edad y el país de todos los jugadores
      // SELECT j.nombre, j.edad, p.nombre FROM Jugador j, Pais p WHERE j.id = p.id
      IValuesQuery valuesQuery = new ValuesCriteriaQuery(Jugador.class)
            .field("nombre")
            .field("edad")
            .field("pais.nombre");
      Values values = odb.getValues(valuesQuery);
      while (values.hasNext()) {
         ObjectValues objectValues = (ObjectValues) values.next();
         System.out.println("Nombre: " + objectValues.getByAlias("nombre") +
               " *** Edad: " + objectValues.getByIndex(1) +
               " *** País: " + objectValues.getByIndex(2));
      }
      System.out.println("\nNombre de país y la ciudad de los jugadores "
            + "de FRANCIA mayores de 15 años\n");
      // consulta del nombre de país y la ciudad de los jugadores
```

```
      // cuyo nombre de país es FRANCIA y cuya edad es mayor que 15
      // SELECT p.nombre, j.ciudad FROM Jugador j, Pais p
      // WHERE j.id = p.id AND p.nombre = 'FRANCIA' AND j.edad > 15
      ICriterion criterio = new And().add(Where.equal("pais.nombre", "FRANCIA"))
            .add(Where.gt("edad", 15));
      valuesQuery = new ValuesCriteriaQuery(Jugador.class, criterio)
            .field("pais.nombre", "nombrepais")
            .field("ciudad");
      values = odb.getValues(valuesQuery);
      while (values.hasNext()) {
         ObjectValues objectValues = (ObjectValues) values.next();
         System.out.println("Nombre de País: " + objectValues.getByAlias("nombrepais") +
               " *** Ciudad: " + objectValues.getByAlias("ciudad"));
      }

   }//main

}
```