SISTEMAS GESTORES DE BASES DE DATOS ORIENTADAS A OBJETOS

Juan Jambrina Martín

Héctor Mateos Oblanca

Departamento de Informática y Automática

Universidad de Salamanca

Resumen

Este documento pretende mostrar la forma en la que las bases de datos orientadas a objetos resuelven el problema de la persistencia en la programación OO.

Se presenta en primer lugar una serie de conceptos y principios generales que los Sistemas Gestores de Bases de Datos OO han de cumplir.

A continuación se introducen algunas referencias básicas sobre el estándar ODMG, incluyendo tipos de datos, lenguajes ODL, OML y OQL, etc, con sus ejemplos ilustrativos correspondientes.

Finalmente se elige *Matisse* como uno de los SGDBOO fundamentado en el estándar ODMG y se expone con la ayuda de ejemplos en C++ los procedimientos de definición, acceso, consulta...

Abstract

This document tries to show the way that object-oriented databases solve the persistence problem in OO programming.

At first time it presents a list of concepts and general principles OO DataBase Management Systems have to fulfil.

After that, it introduces some basic references about ODMG standard including data types, ODL, OML and OQL languages, etc, with their corresponding illustrative examples.

Finally *Matisse* is chosen as one of the SGDBOOs based on ODMG and also the definition, access, consult... procedures are presented with a few C++ examples help.

Tabla de Contenidos

Tabla de figuras

1. INTRODUCCIÓN

Hasta la aparición de las BD[1] Orientadas a Objetos (BDOO[2]), las BD tradicionales no estaban diseñadas para almacenar objetos, con lo que al guardar los datos de un programa OO se incrementaba significativamente la complejidad del programa, dando lugar a más código y más esfuerzos de programación.

Las BDOO están diseñadas para simplificar la POO[3]. Almacenan los objetos directamente en la BD, y emplean las mismas estructuras y relaciones que los lenguajes de POO. Las BDOO surgen de la combinación de las BD tradicionales y la POO.

Un SGBDOO[4] es un Sistema de Objetos y un SGBD[5]. Se puede decir que un SGBDOO es un SGBD que almacena objetos incorporando así todas las ventajas de la OO. Para los usuarios tradicionales de BD, esto quiere decir que pueden tratar directamente con objetos, no teniendo que hacer la traducción a tablas o registros. Para los programadores de aplicaciones, esto quiere decir que sus objetos se conservan, pueden ser gestionados aunque su tamaño sea muy grande, pueden ser compartidos entre múltiples usuarios, y se mantienen tanto su integridad como sus relaciones.

Una clase después de programada es transportada a la BD tal como es, al contrario de lo que sucede en los SGBD relacionales donde el modelo de datos se distribuye en tablas.

Sin las BDOO, hay que hacer un complicado proceso de traducción de los objetos a registros o tablas de BD tradicionales. El problema de la traducción a tablas implica:

- *Mayor tiempo de desarrollo*. El tiempo empleado en generar el código para la traducción de objetos a tablas y viceversa.
- *Errores* debidos precisamente a esa traducción.
- *Inconsistencias* debidas a que el ensamblaje / desensamblaje puede realizarse de forma diferente en las distintas aplicaciones.
- *Mayor tiempo de ejecución* empleado para el ensamblaje / desensamblaje.

La utilización de una BDOO simplifica la conceptualización ya que la utilización de objetos permite representar de una manera más natural la información que se quiere guardar. Mejora el flujo de comunicación entre los usuarios, los diseñadores y los analistas.

Las BDOO permiten implementar los tres componentes de un modelo de datos:

1. Propiedades estáticas (objetos, atributos y relaciones)
2. Reglas de integridad de los objetos y operaciones
3. Propiedades dinámicas

1 BD: base de datos.

2 BDOO: base de datos orientada a objetos.

3 POO: programación orientada a objetos.

4 SGBDOO: sistema gestor de bases de datos orientadas a objetos.

5 SGBD: sistema gestor de bases de datos.

Antes de los SGBDOO, los dos primeros componentes de un modelo de datos eran implementados en la BD dejando las propiedades dinámicas a cargo de la aplicación. Los SGBDOO implementan las tres características en la BD permitiendo, por ejemplo, aplicar reglas uniformes a objetos sin necesidad de desarrollar otra aplicación nueva cuando surge un cambio en el modo de trabajar con los datos.

2. CARACTERÍSTICAS BÁSICAS DE UN SGBDOO

Un SGBDOO debe satisfacer dos criterios:

- Ser un **SGBD** lo que se traduce en 5 características principales: Persistencia, Concurrencia, Recuperación ante fallos, Gestión del almacenamiento secundario y facilidad de Consultas.
- Ser un **Sistema OO**[6]*:* todo sistema OO debe cumplir algunas características como son: Encapsulación, Identidad, Herencia y Polimorfismo. Además existen otras características deseables como Control de tipos y Persistencia.

Figura 1: Características básicas de los SGDBOO.

3. MANIFIESTO DE LAS BDOO

El conocido "Manifiesto de los sistemas de BDOO", que es resultado de la colaboración de un cierto número de expertos, ha tenido mucha influencia al definir los objetivos del movimiento de las BDOO. Los aspectos principales de dicho manifiesto son los siguientes:

6 Sistema OO: sistema orientado a objetos.

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

4. MODELO PROPUESTO POR ODMG

4.1. INTRODUCCIÓN

El ODMG[8] es un consorcio industrial de vendedores de SGBDOO que después de su creación se afilió al OMG[9]. El ODMG no es una organización de estándares acreditada en la forma en que lo es ISO o ANSI pero tiene mucha influencia en lo que a estándares sobre SGBDO se

7 DML: *Data Modification Language* (Lenguaje de modificación de datos).

8 ODMG: *Object Database Management Group* (Grupo de gestión de BDOO).

9 OMG: *Object Management Group* (Grupo de gestión de objetos).

refiere. En 1993 publicó su primer conjunto de estándares sobre el tema: el ODMG-93, que en 1997 ha evolucionado hacia el ODMG 2.0. En enero de 2000 se ha publicado el ODMG 3.0.

En el caso de las BDOO la carencia de un estándar es la mayor limitación para su uso generalizado. ODMG-93 fue un punto de partida muy importante para conseguir un lenguaje estándar de BDOO. Adopta una arquitectura que consta de un sistema de gestión que soporta un lenguaje de BDOO, con una sintaxis similar a un lenguaje de programación OO como puede ser C++ o Smalltalk.

La figura siguiente ilustra el estándar de ODMG para los SGBDOO. En esta arquitectura el programador escribe *declaraciones* para el *esquema* de la aplicación, y un *programa fuente* para la implementación. El programa fuente se escribe en un lenguaje de programación (PL) como C++ ampliado para proporcionar un lenguaje de manipulación de datos completo, incluyendo transacciones y consulta de objetos. Las declaraciones del esquema pueden escribirse mediante una extensión del lenguaje de programación (PL ODL en la figura) o en un lenguaje de programación independiente ODL.

Las declaraciones y el programa fuente son compilados con el SGBDOO para producir la aplicación ejecutable. La aplicación accede a BD nuevas o ya existentes, cuyos tipos deben estar de acuerdo con las declaraciones.

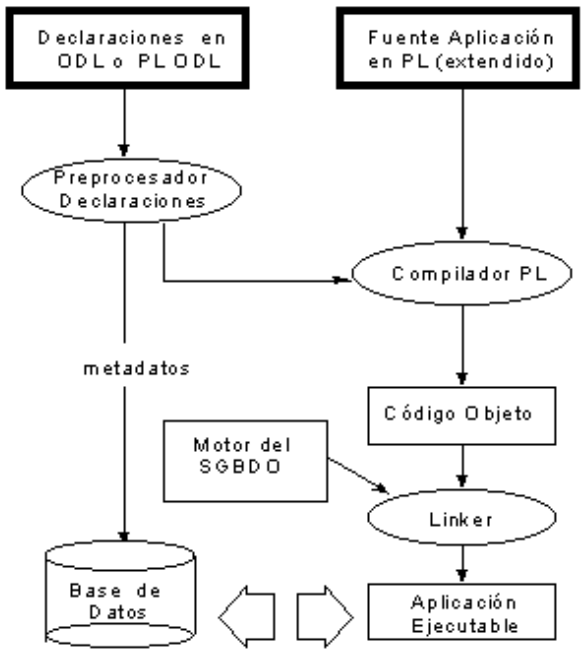

Figura 2: Arquitectura definida por ODMG.

El estándar ODMG define el **Modelo de Objetos** que debe ser soportado por el SGBDO. ODMG se basó en el Modelo de Objetos del OMG que sigue una arquitectura de "núcleo + componentes", como se verá a continuación.

Por otro lado, el lenguaje de BD es especificado mediante un **Lenguaje de Definición de Objetos** (**ODL**) que se corresponde con el DDL de los SGBD tradicionales, un **Lenguaje de Manipulación de Objetos** (**OML**) y un **Lenguaje de Consulta** (**OQL**).

La arquitectura propuesta por ODMG incluye además un método de conexión con lenguajes tan populares como Smalltalk, Java y C++.

4.2. EL MODELO DE OBJETOS

El modelo de objetos ODMG permite que tanto los diseños, como las implementaciones, sean portables entre los sistemas que lo soportan. Dispone de las siguientes primitivas de modelado:

Los componentes básicos de una base de datos orientada a objetos son los *objetos* y los *literales*.

Un **objeto** es una instancia autocontenida de una entidad de interés del mundo real. Los objetos tienen algún tipo de identificador único.

Un **literal** es un valor específico, como "Amparo" o 36. Los literales no tienen identificadores. Un literal no tiene que ser necesariamente un solo valor, puede ser una estructura o un conjunto de valores relacionados que se guardan bajo un solo nombre.

Los objetos se dividen en **tipos**. Los objetos de un mismo tipo tienen un mismo comportamiento y muestran un rango de estados común:

- El **comportamiento** se define por un conjunto de *operaciones* que pueden ser ejecutadas por un objeto del tipo.
- El **estado** de los objetos se define por los valores que tienen para un conjunto de **propiedades**. Las propiedades pueden ser:
  - *ATRIBUTOS* del objeto: Los atributos toman literales por valores y son accedidos por operaciones del tipo `get_value` y `set_value`.
  - *RELACIONES* entre el objeto y uno o más objetos: Son propiedades que se definen entre tipos de objetos, no entre instancias. Las relaciones pueden ser 1-a-1, 1-a-muchos o muchos-a-muchos.

Un tipo tiene una interfaz y una o más implementaciones.

La **interfaz** define las propiedades visibles externamente y las operaciones soportadas por todas las instancias del tipo.

La **implementación** define la representación física de las instancias del tipo y los métodos que implementan las operaciones definidas en la interfaz.

Los tipos pueden tener las siguientes propiedades:

- **Supertipos**. Los tipos se pueden jerarquizar. 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.

  Además de la herencia simple, se admite la herencia múltiple, y en el caso de que 2 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 a 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 realizada para todos los tipos.

- **Claves:** propiedad o conjunto de propiedades que identifican de forma única las instancias de un tipo. Las claves simples están constituidas por una única propiedad. Las claves compuestas están constituidas por un conjunto de propiedades. Las claves pueden estar constituidas no sólo por atributos, sino también por relaciones.

Un tipo puede tener una o más implementaciones. A cada una de estas implementaciones se le da un nombre, que además debe ser único dentro del ámbito definido por un tipo. Las implementaciones asociadas a un tipo son separadas léxicamente en el Lenguaje de Definición de Objetos (ODL). Una **clase**, en este modelo, es la combinación de la interfaz del tipo y una de las implementaciones definidas.

El hecho de permitir varias implementaciones presenta varias ventajas. La primera de ellas es que soporta fácilmente BD que están sobre redes, donde puede haber máquinas con arquitecturas diferentes (soportando mezcla de lenguajes y compiladores). La segunda es que facilita al programador su tarea al poder comparar fácilmente cuál de las implementaciones se comporta mejor. La implementación que emplee un objeto se especifica en tiempo de creación.

4.3. ODL

ODL es un Lenguaje de Definición de Datos para sistemas compatibles con ODMG. ODL es el equivalente del DDL (lenguaje de definición de datos) de los SGBD tradicionales. Define los atributos y las relaciones entre tipos, y especifica la signatura de las operaciones.

El ODL se utiliza para expresar la estructura y condiciones de integridad sobre el esquema de la BD. En una BD relacional define las tablas, los atributos en la tabla, el dominio de los atributos y las restricciones sobre un atributo o una tabla. El ODL además debe poder definir también métodos, jerarquías, herencia, etc.

4.3.1 Tipos en el lenguaje ODL

El lenguaje ODL ofrece al diseñador de BD un sistema de tipos semejantes a los de otros lenguajes de programación comunes. Se pueden crear tipos complejos a partir de otros más simples. Los *tipos* permitidos son:

1. **Tipos básicos** que son:
    a. *Tipos atómicos:* enteros, de punto flotante, caracteres, cadenas de caracteres y booleanos.
    b. *Enumeraciones:* los valores de una enumeración se declaran en ODL.
2. **Tipos de interfaz o estructurados**: son tipos complejos obtenidos al combinar tipos básicos por medio de los siguientes **constructores de tipos**:
    a. *Conjunto* (**`Set<tipo>`**) denota el tipo cuyos valores son todos los conjuntos finitos de elementos del tipo `tipo`.
    b. *Bolsa* (**`Bag<tipo>`**) denota el tipo cuyos valores son bolsas o *multiconjuntos* de elementos del tipo *tipo*. Una bolsa permite a un elemento aparecer más de una vez.

        Ej: `{1, 2, 1}` es una bolsa pero no un conjunto

    c. *Lista* (**`List<tipo>`**) denota el tipo cuyos valores son listas ordenadas finitas conteniendo 0 o más elementos del tipo `tipo`. Un caso especial lo constituye el tipo `string` que es una abreviatura del tipo `List<char>`.

Una lista admite más de una ocurrencia de un elemento pero, a diferencia de las bolsas, las ocurrencias están ordenadas.

Ej: `{1, 2, 1}` y `{2, 1, 1}` son la misma bolsa, pero no son la misma lista.

d. *Array* (**`Array<tipo,i>`**) denota el tipo cuyos elementos son `arrays` de i elementos del tipo *`tipo`*.

Ej: `Array<char, 10>` denota cadenas de caracteres de longitud 10

e. *Estructura* (**`Struct N{tipo1 F1, tipo2 F2, ..., tipop Fp}`**) denota el `tipo N` cuyos elementos son estructuras con `p` campos. El i-ésimo campo se llama `Fi` y tiene el tipo `tipoi`

A los primeros cuatro tipos (`conjunto, bolsa, lista` y `array`) se les denomina **tipos de colección**.

Hay reglas sobre qué tipos pueden asociarse a atributos y cuáles a relaciones.

- El tipo de un atributo se construye partiendo de un tipo básico o de una estructura cuyos campos sean básicos.
- El tipo de una relación es un tipo de interfaz o un tipo de colección que se aplica a un tipo de interfaz.

Por lo tanto, los tipos de interfaz no pueden aparecer en el tipo de un atributo y los tipos básicos no aparecen en el tipo de una relación.

4.3.2 Un ejemplo en ODL

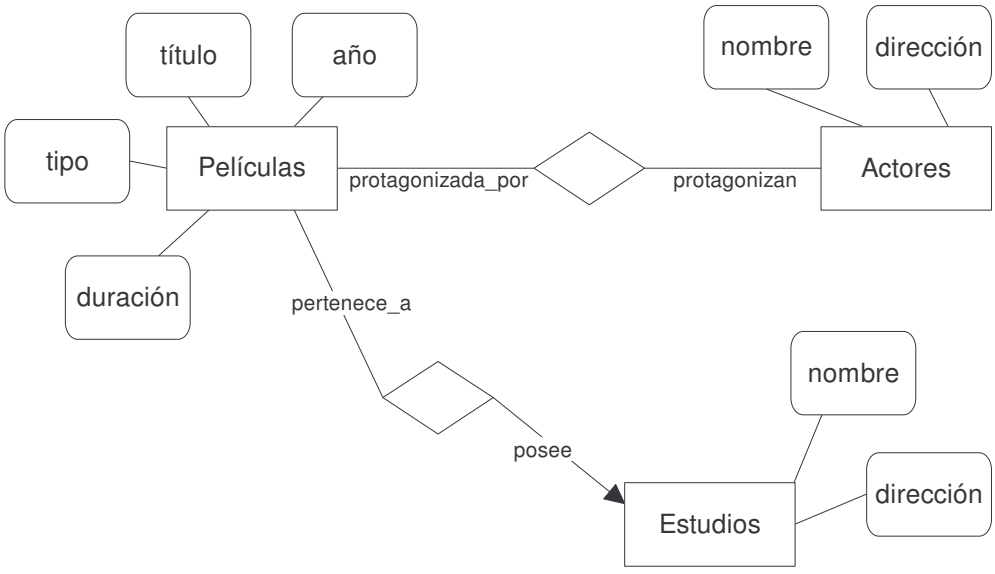

Figura 3: Esquema de la BDD del ejemplo.

```
interface Películas
{
      /* Definición de atributos */
```

```
      attribute string título;
      attribute integer año;
      attribute integer duración;
      attribute enum PosiblesTipos (Color,BlancoNegro) tipo;

      /* Definición de relaciones */
      relationship Estudios pertenece_a
            inverse Estudios::posee;
      relationship Set<Actores> protagonizada_por
            inverse Actores::protagonizan;
}

interface Estudios
{
      /* Definición de atributos */
      attribute string nombre;
      attribute string dirección;

      /* Definición de relaciones */
      relationship Set<Películas> posee
            inverse Películas::pertenece_a;
}

interface Actores
{
      /* Definición de atributos */
      attribute string nombre;
      attribute   Struct   Addr{string   calle,string   ciudad}
dirección;

      /* Definición de relaciones */
      relationship Set<Películas> protagonizan
            inverse Películas::protagonizada_por;
}
```

4.4. OML

El Lenguaje de Manipulación de Objetos (OML) es empleado para la elaboración de programas que permitan crear, modificar y borrar datos que constituyen la BD. El ODMG no propone un OML estándar, simplemente sugiere que este lenguaje sea la extensión de un lenguaje de programación, de forma que se puedan realizar, entre otras, las siguientes operaciones sobre la BD:

- creación de un objeto
- borrado de un objeto
- modificación de un objeto
- identificación de un objeto

4.5. OQL

OQL es un lenguaje declarativo del tipo de SQL que permite realizar consultas de modo eficiente sobre bases de datos orientadas a objetos, incluyendo primitivas de alto nivel para conjuntos de objetos y estructuras. Está basado en SQL-92, proporcionando un superconjunto de la sintaxis de la sentencia `SELECT`.

OQL no posee primitivas para modificar el estado de los objetos ya que las modificaciones se pueden realizar mediante los métodos que estos poseen.

La sintaxis básica de OQL es una estructura `SELECT... FROM... WHERE...`, como en SQL.

Por ejemplo, la siguiente expresión obtiene los títulos de las películas del año 1988:

```
SELECT p.título
 FROM p in películas
 WHERE p.año = 1988;
```

En las consultas se necesita un punto de entrada, que suele ser el nombre de una clase. Para cada objeto de la colección (sólo la forman objetos persistentes) que cumple la condición, se muestra el valor del atributo `título`. El resultado es de tipo `bag<string>`. Cuando se utiliza `SELECT DISTINCT`... el resultado es de tipo set ya que se eliminan los duplicados.

Una vez se establece un punto de entrada, se pueden utilizar expresiones de caminos para especificar un camino a atributos y objetos relacionados. Una expresión de camino empieza normalmente con un nombre de objeto persistente o una variable iterador, seguida de ninguno o varios nombres de relaciones o de atributos conectados mediante un punto. Si se da nombre a un objeto concreto, por ejemplo a una película se le llama `Película1988`:

```
Película1988.pertenece_a;
Película1988.pertenece_a.nombre;
Película1988.protagonizada_por;
```

La primera expresión devuelve una referencia a un objeto `Estudios`, aquel al que pertenece la película. La segunda expresión obtiene el nombre del estudio al que pertenece la

película (el resultado es de tipo `string`). La tercera expresión devuelve un objeto de tipo `set<Actores>`. Esta colección contiene referencias a todos los objetos `Actores` que protagonizan la película. Si se quiere obtener el nombre de todos estos actores, *no* se puede escribir la expresión:

```
Película1988.protagonizada_por.nombre;
```

El no poder escribir la expresión de este modo es porque no está claro si el objeto que se devuelve es de tipo `set<string>` o `bag<string>`. Debido a este problema de ambigüedad, OQL no permite expresiones de este tipo. En su lugar, es preciso utilizar variables iterador:

```
SELECT a.nombre
 FROM a in Película1988.protagonizada_por;


SELECT DISTINCT a.nombre
 FROM a in Película1988.protagonizada_por;
```

En general, una consulta OQL puede devolver un resultado con una estructura compleja especificada en la misma consulta utilizando struct. La siguiente expresión:

```
Película1988.protagonizada_por;
```

devuelve un objeto de tipo `set<Actores>`. Si lo que se necesita son los nombres y direcciónes de estos actores, se puede escribir la siguiente consulta:

```
SELECT struct(
nombre: a.nombre, dirección: a.dirección)
FROM a in Película1988.protagonizada_por;
```

OQL tiene además otras características:

- especificación de vistas dando nombres a consultas.
- Obtención como resultado de un solo elemento (además de las colecciones: `set, bag, list`).
- Uso de operadores de colecciones: funciones de agregados (`max, min, count, sum, avg`) y cuantificadores (`for all, exists`).
- Uso de `group by`.

5. MATISSE: UN EJEMPLO DE UN SGDBOO

5.1. INTRODUCCIÓN

Matisse, de ADB Inc., es un SGBDOO con soporte para C, C++, Eiffel y Java.

Matisse es un diseño atrevido con muchas ideas no convencionales. Está especialmente orientado a grandes bases de datos con una rica estructura semántica, y puede manipular objetos muy grandes como imágenes películas y sonidos. Aunque admite los conceptos básicos de la orientación a objetos, tales como la herencia múltiple, Matisse se abstiene de imponer demasiadas restricciones como lo tocante al modelo de datos y sirve en su lugar como un potente motor de base de datos orientadas a objetos.

Algunas de sus ventajas principales son:

- Una técnica de representación original que hace posible fragmentar un objeto, especialmente un objeto grande, en varios discos, para optimizar así el tiempo de acceso.
- Una ubicación optimizada de los objetos en los discos.
- Un mecanismo automático de duplicación que proporciona una solución software a los fallos de tolerancia del hardware: los objetos (en lugar de los discos en sí) se pueden duplicar especularmente en varios discos, con recuperación automática en caso de fallo del disco.
- Un mecanismo de versiones de objetos incorporado.
- Soporte para las transacciones.
- Soporte para una arquitectura cliente-servidor en la cual un servidor central gestiona los datos para un número posiblemente elevado de clientes, que mantienen una "reserva" de objetos a los que se haya accedido recientemente.

Matisse proporciona mecanismos interesantes para gestionar las relaciones. Si una clase como `Empleado` posee como atributo `Supervisor:Gerente`, Matisse mantendrá, si así se solicita los enlaces inversos de forma automática de modo que será posible no sólo acceder al supervisor de un cierto empleado sino también a todos los empleados que sean administrados por un cierto supervisor. Además, las facilidades de consulta pueden recuperar objetos a través de palabras reservadas asociadas.

5.2. EJEMPLO: DEFINICIÓN DEL ESQUEMA DE LA BD

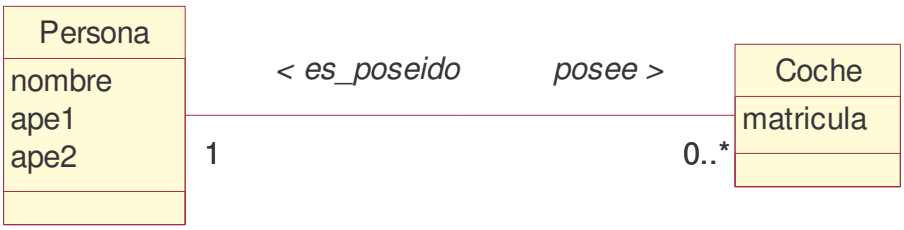

Figura 4: Esquema conceptual de ejemplo.

Si bien Matisse proporciona un entorno gráfico que permite la definición del esquema de la base de datos sin necesidad de conocer la sintaxis ODL (ver figura a continuación), en este ejemplo se opta por la creación de un simple fichero `*.odl` con cualquier editor de texto (se muestra a continuación `ejemploPOO.odl`) en el que se define directamente el esquema mediante dicho lenguaje estándar. Tras ello se importaría este fichero a la base de datos.

```
interface Persona : persistent {
      attribute String nombre;
      attribute String ape1;
      attribute String ape2;
      relationship List<Coche> posee[0, -1]
      inverse Coche::es_poseido;
};

interface Coche : persistent {
      attribute String matricula;
      relationship Persona es_poseido
      inverse Persona::posee;
};
```

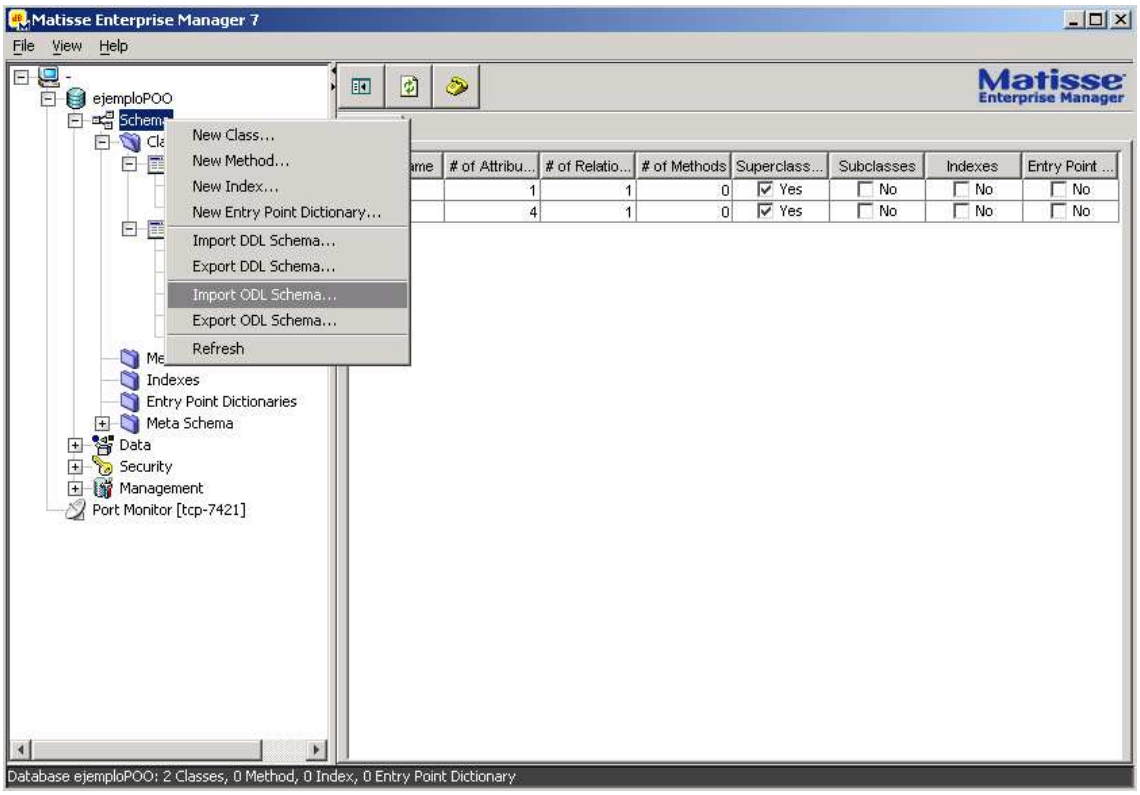

Figura 5: Definición de esquemas en Matisse.

5.3. EJEMPLO: GENERACIÓN DE FICHEROS FUENTE

Una vez que se dispone del esquema se podrá obtener un conjunto de ficheros fuente del lenguaje que se desee (en este caso C++, pero igualmente Java o Eiffel).

Para ello se recurre a un comando del tipo

```
mt_sdl stubgen -[cxx|eiffel|java] [-n paquete] fichero.odl
```

Estos ficheros contienen todas las declaraciones correspondientes a las clases del esquema (definiciones, constructores, destructores...), incluyendo además un completo conjunto de

operaciones para dar y recuperar los valores de cada uno de los atributos y para establecer, destruir y obtener las relaciones entre los objetos de múltiples maneras.

Todo esto sirve para lograr la interacción entre cualquier base de datos y cualquier programa que requiera de su contenido.

Véase a continuación uno de los ficheros generados en el ejemplo: `Persona.h`

```
#ifndef Persona_H
#define Persona_H 1

#include <matisseCXX.h>
#include <matisse/reflect/MtObject.h>
class Coche;
using namespace matisse;
using namespace matisse::reflect;

class Persona : public virtual MtObject
{
  public:
    static MtObject *newStub(const MtDatabase &db, ::MtOid oid);
  private:
    static const MtStub stub;
  protected:
    Persona(const MtDatabase &db, ::MtOid oid);
    virtual ~Persona();
  private:
    static const unsigned int CID;
  public:
    static MtClass &getClass(const MtDatabase &db);
    static MtObjectIterator<Persona> instanceIterator(const MtDatabase &db);
    static unsigned int getInstanceNumber(const MtDatabase &db);

/* Attribute 'nombre' */
  private:
    static const unsigned int nombreCID;

  public:
    static MtAttribute &getNombreAttribute(const MtDatabase &db);
    std::string getNombre() const;
    void setNombre(const std::string & val) const;
    void removeNombre() const;

/* Relationship 'posee' */
  private:
    static const unsigned int poseeCID;
  public:
```

```
    static MtRelationship &getPoseeRelationship(const MtDatabase &db);
    MtObjectArray<Coche> getPosee() const;
    unsigned int getPoseeSize() const;
    MtObjectIterator<Coche> poseeIterator() const;
    void setPosee(const MtObjectArray<Coche> &succs) const;
    void prependPosee(const Coche &succ) const;
    void appendPosee(const Coche &succ) const;
    void appendPosee(const MtObjectArray<Coche> &succs) const;
    void removePosee(const MtObjectArray<Coche> &succs) const;
    void removePosee(const Coche &succ) const;
    void clearPosee() const;

// END of mt_odl generated code
     // You may add your own code here...
/*COMO INDICA EL COMENTARIO ANTERIOR AQUÍ SE PUEDE AÑADIR NUEVOS MÉTODOS*/
   void imprime_ncompleto(){
          std::cout<<getApe1()<<" "<<getApe2()<<", "<<getNombre()<<std::endl;
   }
   static Persona &create(const MtDatabase &db);
};
MT_INLINE std::ostream &operator<<(std::ostream &o, const Persona &obj);
#endif /* Persona_H */
```

5.4. EJEMPLO: ACCESO A LA BASE DE DATOS DESDE UN FICHERO FUENTE C++

En estos dos ejemplos se muestra como acceder a una base de datos desde un fuente. En el primero se crean objetos y se establecen relaciones entre ellos y en el segundo se accede a los mismos.

Primer ejemplo:

```
#include <matisseCXX.h>
#include "Coche.h"
#include "Persona.h"

int main(int argc, char *argv[])
{
   if (argc < 3)
   {
          std::cerr<<"sintaxis: "<<argv[0]<<"<host> <base_de_datos>"<< std::endl;
          return -1;
   }
   try
   {
          MtDatabase ejemploPOO(argv[1], argv[2]);
          /*conexión*/
          ejemploPOO.open();
          ejemploPOO.startTransaction();
          std::cout << "conexión realizada" << std::endl;

          /*crea objetos Coche y da valor a sus atributos*/
          Coche& c1 = Coche::create(ejemploPOO);
```

```
            c1.setMatricula("6363 PJT");
            Coche& c2 = Coche::create(ejemploPOO);
            c2.setMatricula("3598 OBJ");

            /*crea un objeto Persona y da valor a sus atributos*/
            Persona& p = Persona::create(ejemploPOO);
            p.setNombre("Benito");
            p.setApe1("Floro");
            p.setApe2("Sanz");

            /*establece relaciones entre Coches y Persona anteriormente creados*/
            /*Al establecer una relación no hace falta establecer la inversa*/
            p.appendPosee(c1);
            //c1.setEs_poseido(p);

            //p.appendPosee(c2);
            c2.setEs_poseido(p);

            /*desconexión*/
            ejemploPOO.commit();
            ejemploPOO.close();
    }

    catch (MtException &e)
    {
            std::cerr << e << std::endl;
    }
    return 0;
}
```

Segundo ejemplo:

```
#include <matisseCXX.h>
#include "Coche.h"
#include "Persona.h"

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
            std::cerr<<"sintaxis: "<<argv[0]<<"<host> <base_de_datos>"<< std::endl;
            return -1;
    }

    try
    {
            MtDatabase ejemploPOO(argv[1], argv[2]);

            /*conexión*/
            ejemploPOO.open();
            ejemploPOO.startTransaction();
            std::cout << "conexión realizada "<< std::endl;

            /*objeto iterador que obtiene todas las personas que haya en la bdd*/
            MtObjectIterator<Persona> piter = Persona::instanceIterator(ejemploPOO);

            while (piter.hasNext())
            {
                    Persona &x = piter.next();

                    /*objeto iterador que obtiene todos los coches que tenga cada
                persona*/

                    MtObjectIterator<Coche> citer=x.poseeIterator();
                    while (citer.hasNext())
                    {
                            Coche &y = citer.next();
                            std::cou<<y.getMatricula()<<" --> ";x.imprime_ncompleto();
                    }
            }
            /*desconexión*/
            ejemploPOO.commit();
```

```
        ejemploPOO.close();
    }
    catch (MtException &e)
    {
        std::cerr << e << std::endl;
    }
    return 0;
}
```

Para compilar cada uno de estos programas en Visual C++ se hará uso de un fichero de tipo `Makefile` adecuado. Del mismo modo si se usa otro compilador se tendrá que emplear el `Makefile` correspondiente al mismo que Matisse proporciona.

En este caso el `makefile` es el siguiente:

```
RM=del
CP=copy
MATISSE_LIBS=matisse.lib
MATISSE_HOME=C:\Archiv~1\Matisse
CL_OPTS=/nologo /W3 /O1 /GR /GX
CPP_OPTS=/I$(MATISSE_HOME)\include
LDFLAGS=/nologo /libpath:$(MATISSE_HOME)\lib
CPPFLAGS=$(CL_OPTS) $(CPP_OPTS)
LINK=$(CXX)

ODL_SRCS = Coche.cpp Persona.cpp
##Colección de ficheros fuentes de las clases persistentes

## ficheros necesarios para un ejecutable concreto
EJEMPO_SRCS=\
      $(ODL_SRCS)\
      matisseCXX.cpp\
      main.cpp
## ...

EJEMPO_OBJS=$(EJEMPO_SRCS:.cpp=.obj)
EJEMPO_EXEC=EjemploPOO.exe

ALL_OBJS = $(EJEMPO_OBJS)
ALL_EXECS=$(EJEMPO_EXEC)

default: $(ALL_EXECS)

matisseCXX.cpp: $(MATISSE_HOME)\include\matisseCXX.cpp
      $(CP) $** .

$(EJEMPLO_EXEC) : $(EJEMPLO_OBJS)
      $(LINK) /Fe$@ $** /link $(LDFLAGS) $(MATISSE_LIBS)

clean:
      $(RM) $(ALL_OBJS) $(ALL_EXECS)
```

A continuación se ejecuta el siguiente comando para ejecutar dicho `makefile`:

```
nmake fMakefile.win32
```

5.5. EJEMPLO: ACCESO A LA BASE DE DATOS MEDIANTE CONSULTAS SQL

En este ejemplo que se presenta se muestra la forma de recuperar información de una base de datos utilizando sentencias OQL (Matisse lo sigue denominando SQL).

Si bien en el ejemplo se hace desde un programa editado en C++, Matisse proporciona además la posibilidad de hacer las consultas desde su propio entorno gráfico:

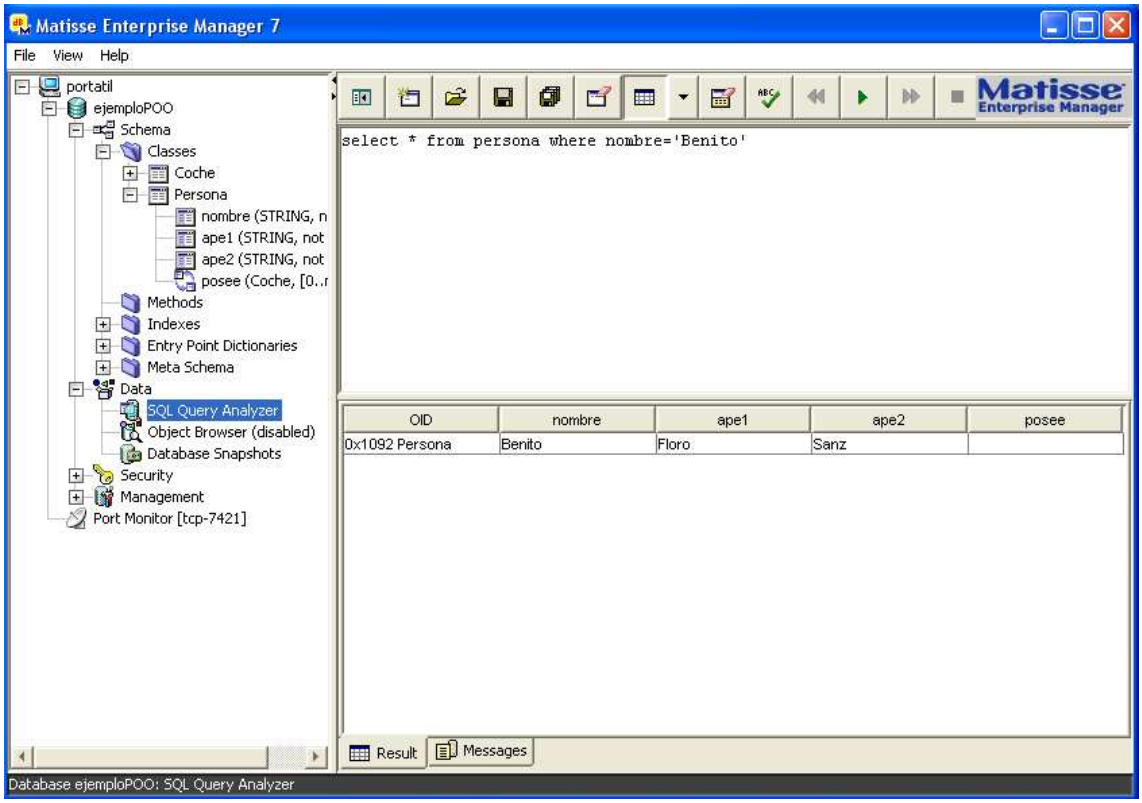

Figura 6: OQL en Matisse.

Véase el siguiente fragmento:

```
MtStatement stmt(ejemploPOO);

        /*Se ejecuta la sentencia SQL y se almacena en un objeto de tipo
        MtResultSet que funciona de forma similar a un iterador*/

MtResultSet res = stmt.executeQuery("Select Ref(Persona) from Persona where
nombre='Benito';");


while (res.next())

{

        /*Se recupera un objeto en cada iteración*/

        Persona *x = dynamic_cast<Persona *>(res.getMtObject(1));

        MtObjectIterator<Coche> citer=x->poseeIterator();

        while (citer.hasNext())

        {

                Coche &y = citer.next();
```

```
                std::cout<<y.getMatricula()<<" --> ";x->imprime_ncompleto();
        }
}
```

6. CONCLUSIONES

- Hay muchas aplicaciones que necesitarán el soporte de una BD y que tendrán que ofrecer un acceso concurrente a sus clientes.
- Para obtener todas las ventajas de la tecnología OO y evitar la falta de correspondencia entre el desarrollo y el modelo de datos se pueden utilizar BDOO.
- Se dispone del estándar ODMG que elabora procedimientos que pretenden ser comunes para los diferentes SGDBOO y obtener así las numerosas ventajas de esa unificación.
- Matisse es uno de los SGBDOO existentes en el mercado que se asientan en las bases de ODMG y proporciona soporte para los lenguajes OO más extendidos como C++, Java y Eiffel.

7. BIBLIOGRAFÍA

- "*The Object Data Standard ODMG 3.0*".Cattell y Barry 2000.
- "Introducción a los sistemas de Bases de Datos". Ullman y Widom.
- "Construcción de software orientado a objetos". Meyer 1998.
- "*Matisse ODL Programmer's guide*" 7th edition February 2004.
- "*Matisse C++ Programmer's guide*" 3rd edition February 2003.
- "*The Object-Oriented Database System Manifiesto*". Atkinson et al 1989.
- http://www.fresher.com
- http://www.odmg.com

8. LECTURAS COMPLEMENTARIAS

- "*C++ object databases : programming with the ODMG standard*". Jordan.
- "*Introduction to object-oriented databases*". Kim.
- "*Object-oriented database clearly explained*". Harrington.
- "*Object-oriented modeling and design for database*". Blaha.
- http://www.cetus-links.org/oo_db_systems_1.html

APÉNDICE 1: OBTENCIÓN E INSTALACIÓN DE MATISSE

Matisse se consigue de forma sencilla a través de su página web oficial (www.fresher.com) donde, previo registro gratuito, se permite la descarga de una versión libre (la nº7 es la última hasta el momento) del programa y se ofrece también una serie de manuales en inglés, además de otros servicios convencionales.

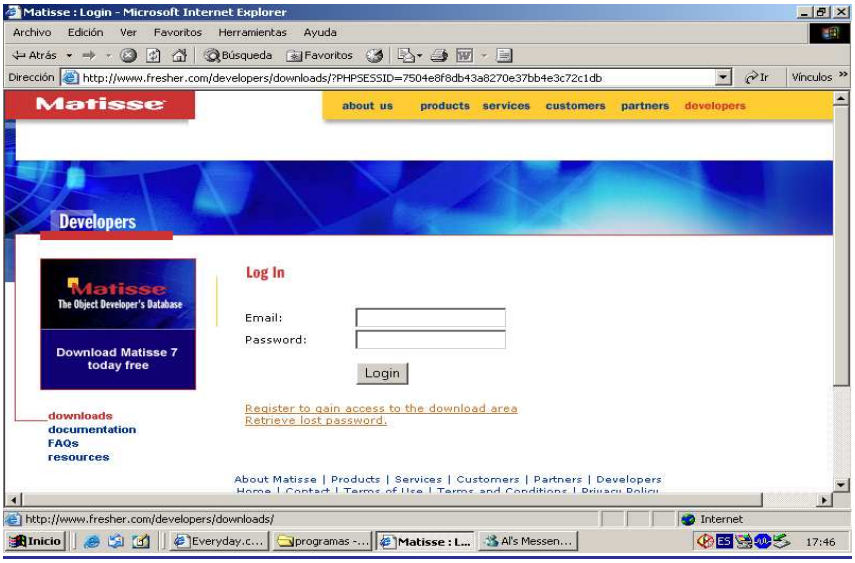

Figura 7: Sitio web de Matisse

La instalación es sencilla a través de un sencillo y conocido asistente.

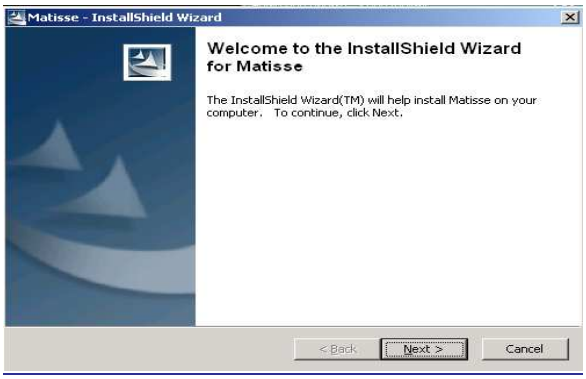

Figura 8: Instalador de Matisse

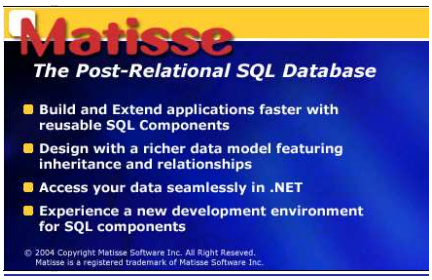

Figura 9: ¡Matisse Listo!