5.1. Bases de datos de objetos y objeto-relacionales, y correspondencia objeto-relacional

Con el auge de la programación orientada a objetos desde finales de los años ochenta se pusieron de manifiesto las dificultades, tanto conceptuales como técnicas, que planteaba el almacenamiento de objetos complejos en bases de datos relacionales. Para referirse a ellas en conjunto se acuñó el término *desfase objeto-relacional* (en inglés, *object-relational impedance mismatch*) o *desajuste de impedancia objeto-relacional*.

Como solución natural se plantearon bases de datos que permitieran almacenar directamente objetos, a las que se llamó BDO, o en inglés ODB *(object databases),* o también bases de datos orientadas a objetos (BDOO), o en inglés OODB *(object-oriented databases)*.

Se intentó repetir la receta del éxito de las bases de datos relacionales unos años antes: el estar basadas en un modelo formal y el temprano desarrollo de estándares, principalmente el lenguaje SQL. Así pues, los primeros esfuerzos se desarrollaron en estos dos ámbitos. En el primero, destaca el manifiesto de Atkinson y otros (1989), con una propuesta para los requisitos que debería cumplir una BDO. Para desarrollar estándares sobre los que basar las BDO, se creó en 1991 el grupo ODMG (Object Database Management Group, en un principio). Este grupo fue creado por R. Cattel (de Sun) y representantes de varios fabricantes de BDO, justo cuando Sun acababa de desarrollar el lenguaje Java. En 1998, ODMG cambió el significado de sus siglas a Object Data Management Group, para reflejar que su objetivo era la persistencia de objetos en general, no necesariamente en BDO.

A pesar del entusiasmo inicial, los resultados a medio y largo plazo no fueron los esperados. No llegó a desarrollarse ningún modelo formal ampliamente aceptado. ODMG desarrolló estándares como ODL para definición de datos, OQL para consulta de datos y, en lugar de un OML o lenguaje específico de manipulación de datos, *language bindings* o vinculaciones con lenguajes orientados a objetos de propósito general ya existentes. La última versión del estándar es ODMG 3.0, publicada en 2000. ODMG se disolvió en 2001, y las compañías que lo integraban centraron sus esfuerzos en JDO (Java *data objects*). En 2004 se cedieron los derechos para revisar la especificación ODMG 3.0 a OMG (Object Management Group).

Por otra parte, en SQL:99 se incluyeron tipos estructurados definidos por el usuario, entre ellos tipos de objetos, que se han implementado en algunas bases de datos relacionales como Postgre-SQL y Oracle, que se pueden considerar por ello BDOR.

Las BDO siguen teniendo un uso muy limitado a fecha de hoy. Existen unas cuantas BDO de *software* libre y privativas, entre ellas Matisse, Objectivity/DB y db4o. Otros planteamientos alternativos para persistencia de objetos han tenido más éxito. Oracle es una BDOR dominante en el segmento de las aplicaciones empresariales, que implementa los objetos en una capa de *software* sobre una base de datos relacional. La correspondencia objeto-relacional (ORM) ha tenido mucho éxito con productos como Hibernate (2001), y el estándar JPA para ORM es parte integrante de Java EE desde Java EE 6 (2009).

5.2. Características de las bases de datos de objetos

Una BDO es una base de datos que puede trabajar directamente con objetos. El manifiesto de Atkinson y otros (1989) propone una lista de características obligatorias que debería cumplir cualquier SGBDO (sistema gestor de BDO), o en inglés ODBMS *(object database management system)*. Es muy amplio, pero contiene las ideas fundamentales sobre las que se basan las BDO. En este resumen se omite, por conocido, lo que es similar en las bases de datos relacionales, y conceptos no estrictamente de bases de datos, sino en general de programación orientada a objetos.

1. *Objetos complejos.* Debe ser posible almacenar objetos complejos. Un objeto complejo contiene un conjunto de atributos, cada uno con un nombre y un valor, como una fila de una base de datos relacional. Pero en un objeto complejo los atributos pueden ser no solo de tipos elementales, como números y cadenas de caracteres, sino también:

   a) Referencias a otros objetos.
   b) Colecciones desordenadas (conjuntos) y colecciones ordenadas (listas).

2. *Identidad de objetos.* Los objetos tienen una existencia independiente de su valor. Cada objeto tiene un identificador único. Se distingue entre igualdad (dos objetos tienen el mismo valor, o mejor dicho, los mismos valores para sus atributos) e identidad (dos objetos son el mismo objeto, es decir, tienen el mismo identificador único). Identidad implica igualdad, pero no a la inversa.
3. *Tipos o clases*. Un tipo comprende un conjunto de objetos con características comunes, a saber, un conjunto de atributos y un conjunto de operaciones. Una clase es similar, pero es algo más que un conjunto de objetos. Una clase es a su vez un objeto sobre el que se pueden realizar algunas operaciones. Una clase podría mantener su extens*ión (exte*nt), es decir, el conjunto de todos los objetos de la clase, y proporcionar mecanismos para acceder a ellos y realizar operaciones sobre ellos. En adelante se hablará solo de clases, pero todo lo que se diga de ellas se puede hacer extensivo a los tipos.
4. *Extensibilidad*. El sistema de bases de datos tendrá un conjunto de tipos y clases predefinidos. Debe poderse definir nuevos tipos y clases a partir de otros ya existentes, y estos deben poderse utilizar para cualquier cosa para la que se puedan usar los predefinidos.
5. *Completitud computacional.* Debe poderse realizar cualquier posible cálculo utilizando el DML (lenguaje de manipulación de datos) de la base de datos. Esta es una diferencia fundamental con las bases de datos relacionales. El sublenguaje DML de SQL no es computacionalmente completo. Para realizar muchos o la mayoría de los cálculos con los datos hay que obtener los datos en el contexto de un lenguaje de programación de propósito general. Este requisito se puede satisfacer mediante *bindings* o vinculaciones con lenguajes orientados a objetos de propósito general ya existentes, como por ejemplo Java, en lugar de con un DML separado.
6. *Métodos de consulta sencillos*. Idealmente de alto nivel, eficientes e independientes de la aplicación, es decir, utilizables con cualquier posible base de datos y lenguaje de programación.

Recurso digital

En el anexo web 5.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás el manifiesto de Atkinson, Bancilhon, DeWitt, Dittrich, Maier y Zdonik (1989).

5.3. El estándar ODMG

El estándar ODMG es un estándar para persistencia de objetos en bases de datos, cuya última versión es ODMG 3.0, del año 2000. Los principales componentes de ODMG 3.0 son:

- Modelo de objetos. Basado en el modelo de objetos de OMG.

- ODL o lenguaje de definición de objetos.
- OQL o lenguaje de consulta de objetos. Inspirado en SQL-92.
- *Language bindings* o vinculaciones con los lenguajes C++, Smalltalk y Java.

A diferencia de SQL para bases de datos relacionales, que incluye un DML independiente del lenguaje de programación desde el que se usa, ODMG 3.0 no incluye un DML, sino *language bindings* o vinculaciones con lenguajes de propósito general existentes, a saber: C++, Smalltalk y Java. Este planteamiento se conoce como *persistencia transparente*. Desde el lenguaje de programación de propósito general se crean, modifican y borran los objetos persistentes de la misma forma que los no persistentes, y se utilizan los mecanismos del propio lenguaje. Es en el momento de confirmar los cambios realizados dentro de una transacción cuando se graban en la base de datos los cambios realizados sobre objetos persistentes. Un *languaje binding* también proporciona mecanismos para realizar consultas de OQL y obtener los resultados como objetos persistentes.

Recurso digital

En el anexo web 5.2, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás más información acerca de ODMG y del soporte del estándar ODMG 3.0 por parte de diversas BDO (2001).

5.4. ODL

ODL es un lenguaje formal para especificación de objetos. Un SGBDO almacena objetos de tipos definidos en un esquema mediante ODL. Se pueden establecer las analogías con el modelo relacional que se recogen en el cuadro 5.1.

CUADRO 5.1
Analogías entre el modelo ODMG y modelo relacional

Modelo de objetos de ODMG 3.0	Modelo relacional
ODL: lenguaje de definición de objetos	DDL: lenguaje de definición de datos, subconjunto de SQL
Esquema de objetos	Esquema relacional
Clase	Tabla
Instancia de clase	Fila de tabla

5.4.1. Modelo de objetos de ODL

ODL se basa sobre el modelo de objetos de OMG, cuyas principales características son:

1. El estado de un objeto está definido por los valores de sus propiedades. El valor de las propiedades, y por lo tanto el estado de un objeto, puede cambiar a lo largo del tiempo.

2. Las propiedades pueden ser atributos del objeto o relaciones con otros objetos. Las relaciones pueden ser de uno a uno, de uno a muchos o de muchos a muchos.
3. Un objeto es una instancia de una clase y tiene un OID (*object identifier*), que es un identificador único a nivel del sistema, y que no cambia a lo largo del tiempo. Se distingue entre identidad, determinada por el OID (dos objetos son idénticos si y solo si tienen el mismo OID), e igualdad (dos objetos son iguales si y solo si tienen el mismo estado, es decir, los mismos valores para sus atributos).
4. La conducta (*behavior*) de un objeto se define por el conjunto de operaciones que se pueden realizar sobre él. Cada una puede tener una lista de parámetros de entrada y de salida, cada uno de un tipo, y puede devolver un resultado de un tipo determinado.
5. Un tipo (que puede ser una clase o una interfaz) tiene una especificación externa y puede tener una o más implementaciones. La especificación incluye las características visibles externamente del tipo, a saber:

   a) Operaciones: que se pueden invocar en las instancias del tipo.
   b) Propiedades: o variables de estado, cuyo valor se puede consultar y cambiar.
   c) Excepciones: que pueden lanzar sus operaciones.

5.4.2. Clases e interfaces

Un tipo puede ser:

a) Una interfaz (`interface`). Una interfaz se define por su conducta abstracta, es decir, por sus operaciones.
b) Una clase (`class`). Una clase se define por su conducta abstracta y por su estado abstracto, es decir, por sus atributos.

La distinción entre interfaces y clases, y su relación con los objetos, es similar a la que hay en Java.

- No se pueden crear instancias de una interfaz, pero sí de una clase. Un objeto es una instancia de una clase.
- Clases e interfaces pueden heredar de interfaces (relaciones de tipo "es un/una").
- Las clases pueden extender otras clases (relaciones de tipo "extiende"). Una clase que extiende a otra, es decir, una subclase, hereda las operaciones y propiedades de la otra, es decir, de la superclase.

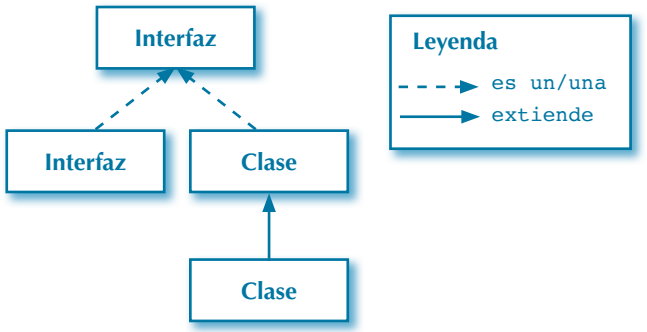


Figura 5.1
Relaciones "es un/una" y "extiende" entre interfaces y clases

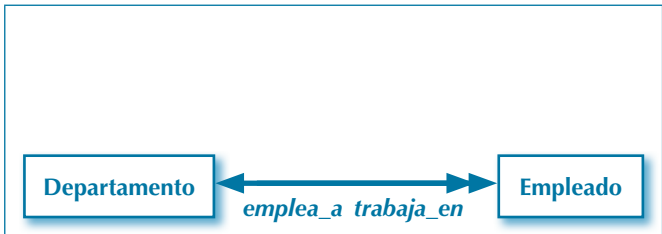
Para una clase se puede definir una extensión. Se define con la palabra clave `extent` y se le asigna un nombre. Una extensión proporciona acceso a todas las instancias de una clase. A una extensión (`extent`) se le

puede asociar una clave (`key`), formada por uno o varios atributos. No puede haber dos objetos distintos dentro de la extensión con idéntico valor para la clave. Esta es una restricción de integridad que debe mantener el SGBDO.

5.4.3. Relaciones

Las relaciones son propiedades de las clases. Una relación entre dos clases consiste en la asociación de cada instancia de una clase con una o más instancias de la otra. ODMG 3.0 solo contempla relaciones binarias, es decir, relaciones entre dos clases.

- Pueden ser de uno a uno (1-1), uno a muchos (1-N) y muchos a muchos (N-M).
- Se definen en ambos extremos, es decir, en cada una de las dos clases, como un *traversal path*. La relación entre `Departamento` y `Empleado`, por ejemplo, es de uno a muchos, y está definida por dos *traversal paths* recíprocos el uno del otro: un departamento *emplea_a* uno o varios empleados, y un empleado *trabaja_en* un departamento.




```
class Departamento {
   relationship set<Empleado>
      emplea_a
   inverse Empleado::trabaja_en;
};
class Empleado {
   relationship Departamento
      trabaja_en
   inverse
      Departamento::emplea;
};
```

Figura 5.2
Relaciones y *traversal paths* en ODL

- Según la cardinalidad máxima de la relación en un *traversal path,* este se puede implementar mediante un único objeto (cardinalidad máxima uno) o mediante una colección de objetos (cardinalidad máxima muchos). En este último caso, la colección puede ser ordenada (`list`) o desordenada (`set`).
- Las relaciones se gestionan mediante operaciones públicas de las clases involucradas. Si en un *traversal path* la cardinalidad máxima es 1, habrá operaciones para especificar el objeto relacionado, y en caso de que la mínima sea 0, para borrarlo. Si la cardinalidad máxima es mayor que 1, existirán operaciones para añadir y borrar objetos en el conjunto de objetos relacionados. En el ejemplo anterior, existirán operaciones para asignar a un empleado su departamento, así como para añadir un empleado a un departamento.

En ODMG 3.0 se usa una notación propia para diagramas de objetos que representan clases y relaciones entre ellas. Como ejemplo se incluye un diagrama E-R, que incluye los proyectos y los empleados de una empresa, y las relaciones entre ellos, y el correspondiente diagrama de objetos. La traducción de uno a otro es directa.

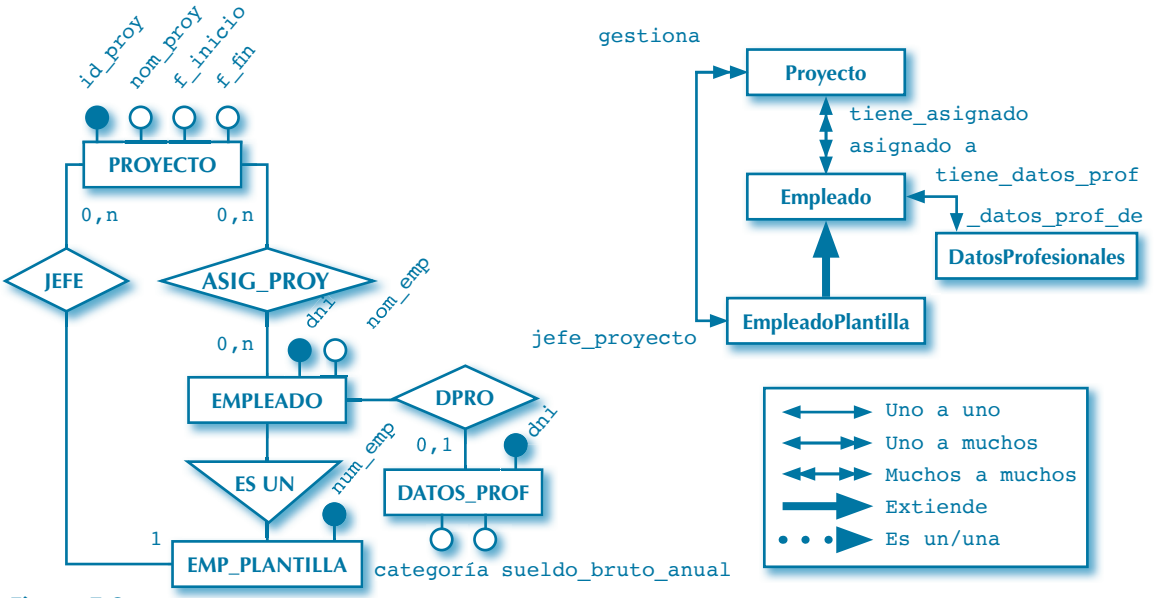


Figura 5.3
Diagrama E-R y de objetos correspondiente para proyectos y empleados de una empresa

Actividad propuesta 5.1

Crea un diagrama E-R, y el correspondiente diagrama de objetos, para representar las siguientes relaciones entre entidades. Una publicación puede ser un libro o una revista. En cualquier caso, tiene un nombre de publicación (`nom_pub`), que en el caso del libro representa el título y en el de la revista, el nombre. Un libro tiene un autor, del que interesa su nombre (`nom_autor`) y su nacionalidad (que se indica en un campo como texto libre). Un libro tiene como identificador un ISBN (`isbn`), y una revista, un ISSN (`issn`). El diagrama E-R debe incluir las entidades PUBLICACIÓN, LIBRO, REVISTA y AUTOR.

Recurso digital

En el anexo web 5.3, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás una explicación detallada de algunos aspectos del ODL de ODMG 3.0.

5.5. OQL

OQL es un lenguaje de consulta incluido en el estándar ODMG 3.0. Tiene una sintaxis inspirada en la sentencia `SELECT` de SQL, pero adaptada para su uso con objetos. El *language binding* permite realizar consultas en OQL y recuperar los resultados mediante un iterador, como objetos que se pueden utilizar con el lenguaje de programación. OQL no incluye sentencias similares a las sentencias `INSERT`, `UPDATE` o `DELETE` de SQL. En lugar de ello, y según el principio de persistencia transparente de ODMG 3.0, las operaciones de creación, borrado y modificación de objetos persis-

tentes se realizan con el lenguaje de programación, cuyo *language binding* garantiza que los cambios realizados sobre ellos dentro de una transacción se reflejen en la base de datos al confirmarse esta.

OQL tiene el equivalente de casi todas las opciones de la sentencia SELECT de SQL y es, por tanto, un lenguaje de consulta muy potente. No se explicará aquí en detalle, sino que se incluirán ejemplos sencillos para ilustrar su funcionamiento general y sus posibilidades.

Las consultas sobre bases de datos se realizan utilizando puntos de entrada. Un punto de entrada da acceso a un objeto o una colección de objetos almacenados. Como punto de entrada puede servir la extensión *(extent)* de una clase que, como ya se ha comentado, permite acceder a todas sus instancias. A continuación, se indica cómo se incluiría en la definición de las clases del esquema de ejemplo la definición de una extensión asociada y de una clave para ella.

```
class Proyecto(extent proyectos key id_proy)
class Empleado(extent empleados key dni)
class EmpleadoPlantilla(extent empleados_plantilla key numEmp) extends
    Empleado
```

La siguiente consulta obtiene los jefes de proyectos con fecha de inicio desde 2017.

```
SELECT p.jefe_proyecto
FROM proyectos p
WHERE p.f_inicio >= date '2017-01-01';
```

Se utiliza como punto de entrada a la base de datos la extensión `proyectos` de la clase `Proyecto`, con alias `p`. El resultado es un objeto de tipo `bag<EmpleadoPlantilla>`. En un `bag` puede haber elementos repetidos. Para evitarlo se puede utilizar `SELECT DISTINCT` en lugar de `SELECT`, y entonces se obtendría un objeto de tipo `set<EmpleadoPlantilla>`.

Se puede acceder a una propiedad de un objeto añadiendo un punto y el nombre de la propiedad. Una propiedad puede ser de un tipo atómico, o un objeto o una relación con otros objetos, en cuyo caso se puede a su vez acceder a sus propiedades de la misma forma. Utilizando este mecanismo repetidamente se puede construir un *path expression* para navegar por la estructura de objetos complejos recuperados a partir del punto de entrada. El siguiente ejemplo recupera el número de empleado de los jefes de proyecto recuperados en el ejemplo anterior, y devolvería un objeto de tipo `set<String>`.

```
SELECT DISTINCT p.jefe_proyecto.num_emp
FROM proyectos p
WHERE p.f_inicio >= date '2017-01-01';
```

Si se quisiera recuperar, además, el nombre de los jefes de proyecto, habría que recuperar ambas cosas en un tipo `struct(String, String)`.

```
SELECT DISTINCT struct(num: p.jefe_proyecto.num_emp, nom: p.jefe_proyecto.
    nom_emp)
FROM proyectos p
WHERE p.f_inicio >= date '2017-01-01';
```

En cada consulta anterior el tipo de objeto devuelto es distinto. En la última hay que utilizar incluso una palabra reservada del OQL (`struct`) para especificar el tipo de objeto devuelto. Todo esto dificulta la composición de consultas, es decir, la creación de nuevas consultas basadas en consultas existentes. En SQL, en cambio, las consultas operan con relaciones y obtienen relaciones. Esto simplifica también el desarrollo de programas de aplicación.

OQL incluye cláusulas análogas a `GROUP BY`, `HAVING` y `ORDER BY` de SQL, y los operadores de agregación `min`, `max`, `count`, `sum`, `avg`.

5.6. Consulta y manipulación de datos con el Java binding

Una implementación del Java *binding,* conforme al estándar ODMG 3.0, debe proporcionar una clase que implemente la interfaz `org.odmg.Implementation`. Esta tiene métodos para crear y recuperar bases de datos (`Database`) y transacciones (`Transaction`). Tiene también un método que permite crear consultas de OQL (`OQLQuery`).

La manipulación de objetos persistentes se realiza según el principio de persistencia transparente. Eso significa que no hay un lenguaje específico para manipulación de objetos persistentes (OML), sino que estos se manipulan desde el lenguaje de programación, y de igual manera que los objetos no persistentes. El *language binding* garantiza que, al realizar una operación *commit* para confirmar los cambios realizados en la transacción actual, se graban en la base de datos todos los cambios realizados sobre los objetos persistentes. También se graban los cambios realizados sobre objetos alcanzables siguiendo las referencias desde cualquier objeto persistente. Esto se conoce como *persistence by reachability* o *transitive persistence* (persistencia por alcanzabilidad o persistencia transitiva).

Las clases persistentes o *persistent-capable* (habilitadas para persistencia) se pueden generar de diversas formas. Una es mediante un preprocesador de ODL que genere su código a partir de definiciones en ODL, lo que no impide que se puedan modificar posteriormente.

Para ilustrar todos estos aspectos se desarrollarán varios ejemplos con el Java *binding* de Matisse, que es en gran medida conforme a ODMG 3.0.

5.7. La base de datos de objetos Matisse

Matisse es una de las BDO más veteranas y con mejor soporte para ODMG 3.0. Es una base de datos de pago pero se puede descargar, previo registro, para su evaluación. Cuenta con una magnífica y muy completa documentación, enlazada desde la ayuda del propio programa.

WWW

Recursos web

Los siguientes enlaces proporcionan acceso a la descarga de *software* y a toda la documentación para administradores de bases de datos y desarrolladores.

http://www.matisse.com/developers/downloads
http://www.matisse.com/developers/documentation

La empresa que desarrolla Matisse la define como base de datos posrelacional. Pero, ante todo, y es lo que interesa especialmente aquí, es una potente BDO que soporta en gran medida los diversos aspectos del estándar ODMG 3.0.

- ODL: su implementación es, en gran medida, conforme al planteamiento de ODMG 3.0, aunque usa una sintaxis distinta y presenta algunas discrepancias.
- OQL: Matisse no implementa el OQL de ODMG 3.0. Proporciona como alternativa un *driver* JDBC que permite recuperar colecciones de objetos de la base de datos mediante su lenguaje SQL. Matisse llama SQL a un lenguaje propio que es una adaptación de SQL para BDO, y que incluye no solo una sentencia SELECT como OQL, sino también sentencias `INSERT`, `UPDATE` y `DELETE`.
- *Language bindings:* Matisse proporciona *language bindings* no solo para los incluidos en ODMG 3.0 (C++, Smalltalk y Java, este último con soporte para Java 8 en su versión 9.1.1), sino también para C, C#, Eiffel, Perl, PHP, Python y Visual Basic.

En los siguientes apartados se muestra cómo crear un esquema de objetos en una base de datos de Matisse mediante el ODL de Matisse, y cómo utilizar su Java *binding* para crear programas que se conectan a ella para crear, modificar y borrar tanto objetos como relaciones entre ellos. Para ello será necesario instalar el Enterprise Manager.

5.7.1. Creación de una base de datos mediante ODL con Matisse

Primero, se crea la base de datos y, después, el esquema de objetos en ella mediante ODL. Para crear la base de datos desde Enterprise Manager, se pulsa con el botón derecho del ratón sobre el servidor y se selecciona "New Database". Como nombre se indica `AcDat_BDO`. Después, se arranca la base de datos pulsando sobre ella y con la opción "Start". También están disponibles estas opciones en una barra de iconos en la parte de arriba.

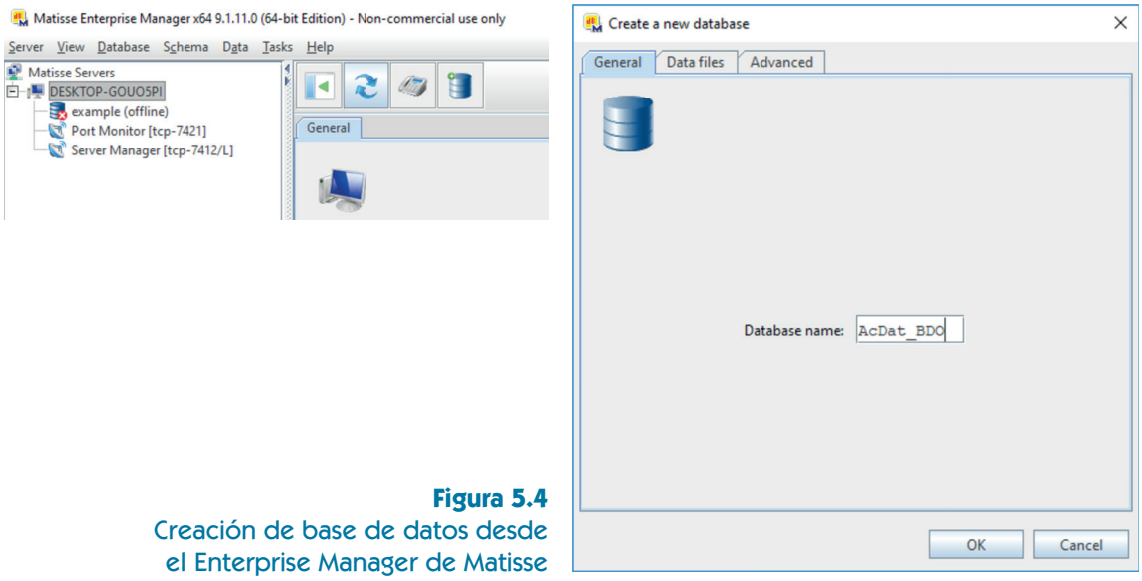


Figura 5.4
Creación de base de datos desde el Enterprise Manager de Matisse

Recurso web

En la siguiente dirección se puede encontrar un documento acerca del ODL de Matisse.

http://www.matisse.com/pdf/developers/matisse_odl.pdf

Las sentencias del ODL de Matisse que permiten crear el esquema de objetos para el modelo de objetos de ejemplo son las siguientes:

```
module gest_proyectos {
    interface Proyecto: persistent
    {
        attribute String<32> nom_proy;
        attribute Date f_inicio;
        attribute Date Nullable f_fin;
        relationship set<Empleado> tiene_asignado[0,-1]
            inverse Empleado::asignado_a;
        relationship EmpleadoPlantilla jefe_proyecto
            inverse EmpleadoPlantilla::gestiona;
    };
    interface Empleado: persistent {
        attribute String<9> dni;
        attribute String<60> nom_emp;
        relationship set<Proyecto> asignado_a[0,-1]
            inverse Proyecto::tiene_asignado;
        relationship DatosProfesionales tiene_datos_prof[0,1]
            inverse DatosProfesionales::datos_prof_de;
        mt_index Empleado_pk unique_key TRUE criteria { dni MT_ASCEND };
        mt_index Empleado_i_nom_emp criteria { nom_emp MT_ASCEND };
    };
    interface DatosProfesionales: persistent {
        attribute String<9> dni;
        attribute String<2> categoria;
        attribute Float sueldo_bruto_anual;
        relationship Empleado datos_prof_de
            inverse Empleado::tiene_datos_prof;
        mt_index Empleado_DatosProf_pk unique_key TRUE criteria { dni MT_ASCEND };
    };
    interface EmpleadoPlantilla: Empleado: persistent {
        attribute String<12> num_emp;
        relationship set<Proyecto> gestiona
            inverse Proyecto::jefe_proyecto;
        mt_index EmpleadoPlantilla_i_dni unique_key TRUE criteria { dni
            MT_ASCEND };
        mt_index EmpleadoPlantilla_i_nom_emp criteria { Empleado::nom_emp
            MT_ASCEND };
    };
};
```

Solo se admiten valores nulos para un atributo si se define con la opción `Nullable`. Por defecto, la cardinalidad mínima de una relación en cualquiera de sus dos extremos o *traversal paths* es 1, y no hay cardinalidad máxima si se define mediante un tipo de colección (`set` o `list`). Se puede cambiar cualquiera de las cardinalidades por defecto indicándolas con `[mín, máx]`. Un valor -1 para `máx` indica que no hay cardinalidad máxima.

Matisse proporciona un identificador único (OID) para cada objeto, único globalmente para todos los objetos existentes en la base de datos y que, por tanto, puede hacer las veces de clave primaria para cualquier clase. Matisse permite definir índices (`mt_index`), sobre un atributo o un conjunto de ellos, que aceleran las búsquedas basadas en sus valores. Los índices pueden ser únicos (de tipo `unique_key`), y entonces sirven como clave primaria, lo que impide que haya más de un objeto con los mismos valores para ellos. En la clase `Empleado` se ha definido un índice único sobre `dni` porque cada persona tiene un DNI distinto, y otro para acelerar búsquedas por nombre. No se ha definido ningún índice único para `Proyecto`. En una base de datos relacional hubiera sido necesario crear en una tabla `proyectos` un atributo para guardar un identificador único, y utilizarlo para la clave primaria.

El esquema de objetos se puede crear importando un fichero con las declaraciones en ODL. El nombre del módulo (en este caso, `gest_proyectos`) es el *namespace* o espacio de nombres, que se creará en la base de datos. El esquema se crea con el botón derecho del ratón sobre la base de datos recién creada, seleccionando "Schema", "Import ODL Schema…". Una vez creado el esquema, se puede utilizar el Data Modeller para obtener diagramas de clases. Este es un programa independiente del Enterprise Manager, y hay que instalarlo aparte.

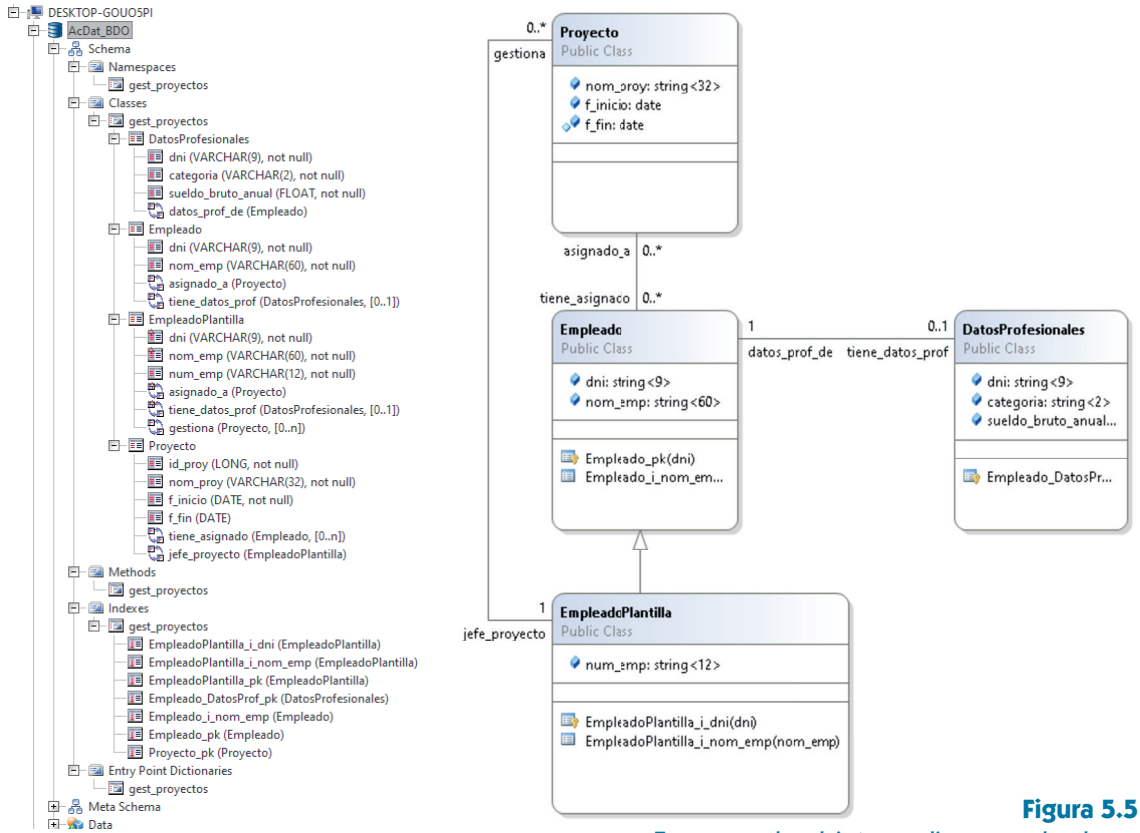


Figura 5.5
Esquema de objetos y diagrama de clases

Actividad propuesta 5.2

Define, utilizando el ODL de Matisse, el esquema de objetos para el modelo de objetos creado en la actividad anterior para publicaciones (libros y revistas), incluyendo todas las clases y los atributos indicados para ellas. Importa el esquema y visualízalo en el Enterprise Manager. Obtén el diagrama de clases con el Data Modeller.

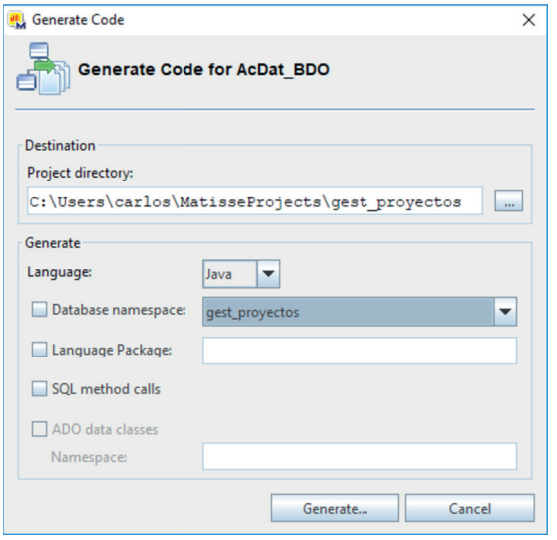
5.7.2. Utilización de la base de datos mediante el Java binding de Matisse

Matisse tiene un Java *binding* muy bien documentado, cuya documentación se puede encontrar en la sección para desarrolladores *(developers)* de su página web, cuyo enlace se ha proporcionado anteriormente.

Recursos web

El primero de los siguientes enlaces corresponde a una guía de programación en Java para Matisse con numerosos ejemplos. El segundo proporciona el código fuente completo de estos ejemplos. El tercero es una aplicación para Java EE que se puede desplegar en el servidor de aplicaciones Tomcat, y que permite diversos tipos de consulta sobre los contenidos de una base de datos de Matisse. Se hablará más acerca de Java EE, y del despliegue de aplicaciones en esta plataforma, en un capítulo posterior dedicado a componentes para acceso a datos.

http://www.matisse.com/pdf/developers/java_pg.pdf
http://www.matisse.com/pdf/developers/java_examples.zip
http://www.matisse.com/pdf/developers/jsp_demo.zip



Matisse genera clases de Java para objetos persistentes *(stub classes)* a partir sus definiciones en ODL. Se hace pulsando con el botón derecho sobre la base de datos y seleccionando "Schema", "Generate Code".

Las clases generadas contienen código que no hay que modificar entre comentarios `// BEGIN Matisse SDL Generated Code` y `// END of Matisse SDL Generated Code`. Este código se encarga de la persistencia de objetos. Fuera de ahí, se pueden añadir métodos adicionales a las clases y recompilarlas, y seguirán funcionando

Figura 5.6
Generación de código (*stub classes*)

perfectamente con la base de datos. A continuación, se indican los métodos más importantes disponibles en las *stub clases* generadas por Matisse para Java.

Matisse no proporciona una implementación de la interfaz `org.odmg.Implementation` conforme al estándar ODMG 3.0. En lugar de ello, proporciona una clase `com.matisse.MtDatabase` en `matisse.jar`.

CUADRO 5.2
Métodos más importantes de las *stub classes* generadas por Matisse para Java

	Método(s)	Funcionalidad
Para cada clase. Son de tipo `public static`. *`Clase`* es el nombre de la clase. Permiten crear objetos persistentes e iterar sobre los objetos persistentes de la clase	*`Clase`*`(MtDatabase db)`	Crea un objeto de la clase. *`Clase`* indica el nombre de la clase.
	`getInstanceNumber(MtDatabase db)` `getOwnInstanceNumber (MtDatabase db)`	Obtienen el número de instancias de la clase existente en la base de datos. Aquellos métodos en cuyo nombre aparece `Own` solo tienen en cuenta las instancias de la propia clase, pero no de las subclases.
	`instanceIterator(MtDatabase db)` `ownInstanceIterator(MtDatabase db)`	Obtienen iteradores para las instancias de la clase. El primero incluye instancias de subclases, el segundo no.
Para índices. Son de tipo `public static`. Permiten acceder directamente a objetos conociendo el valor de determinados atributos. *`Índice`* es el nombre de un índice	`Clase lookup`*`Índice`*`(MtDatabase db, `*`TipoAtrib1`*` valorAtrib1, `*`TipoAtrib2`*` valorAtrib2,...)` `Clase[] lookupObjects`*`Índice`*`(MtDatabase db, `*`TipoAtrib1`*` valorAtrib1, `*`TipoAtrib2`*` valorAtrib2,...)`	Recuperar objetos dados los valores para atributos incluidos en el índice. Los métodos del primer tipo permiten recuperar un objeto, y son especialmente útiles para índices únicos. Los del segundo tipo permiten recuperar varios objetos, y son de utilidad para índices no únicos.
	`Índice``Iterator(MtDatabase db, `*`TipoAtrib1`*` valorAtrib1, `*`TipoAtrib2`*` valorAtrib2,...)`	Devuelve un iterador sobre objetos. Existe otra función que permite especificar opciones adicionales, y que no se describe aquí.
Para cada atributo. *`Atr`* es el nombre del atributo	`getAtr()` `setAtr()`	Métodos *getter* y *setter*.
	`removeAtr()`	Elimina el valor del atributo y le asigna su valor por defecto.
	`isAtrNull()` `isAtrDefault()`	Comprueban si el valor del atributo es nulo (`null`) y si es el valor definido por defecto.

[.../...]

CUADRO 5.2 (CONT.)

Para todas las relaciones. *Rel* es el nombre de la relación	`clear`*`Rel`*`()`	Elimina la relación con todos los sucesores según la relación. Un sucesor de un objeto según una relación es un objeto con el que está relacionado de acuerdo a esa relación. No borra los sucesores, solo elimina la relación del objeto en cuestión con ellos.
Para relaciones con cardinalidad máxima 1. *TipoSuc* es el tipo de los sucesores según la relación	`get`*`Rel`*`()` `set`*`Rel`*`(`*`TipoSuc`*` succ)`	El primer método obtiene el sucesor según la relación. El segundo establece la relación con un objeto dado.
Para relaciones con cardinalidad máxima mayor que 1	*`TipoSuc`*`[] get`*`Rel`*`()`	Obtiene en un *array* sucesores según la relación *Rel*.
	`rel``Iterator()`	Obtiene iterador para sucesores según la relación *Rel*.
	`get`*`Rel`*`Size()`	Obtiene número de sucesores según la relación *Rel*.
	`set`*`Rel`*`(`*`TipoSuc`*`[] sucesores)`	Asigna los sucesores según la relación *Rel*.
	`prepend`*`Rel`*`(`*`TipoSuc`*` sucesor)`	Añade sucesor según relación *Rel*, en primera posición, es decir, antes que los ya existentes.
	`append`*`Rel`*`(`*`TipoSuc`*` sucesor)`	Añade sucesor según relación *Rel* en última posición, es decir, después de todos los ya existentes.
	`append`*`Rel`*`(`*`TipoSuc`*`[] sucesores)`	Añade múltiples sucesores según relación *Rel* después de todos los ya existentes.
	`remove`*`Rel`*`(`*`TipoSuc`*` sucesor)`	Elimina sucesor según la relación *Rel*.
	`remove`*`Rel`*`(`*`TipoSuc`*` [] sucesores)`	Elimina un sucesor según la relación *Rel*.

Para construir un programa que utilice el Java *binding* de Matisse hay que incluir en el proyecto `matisse.jar` y las *stub classes,* como se muestra en la figura.

A continuación, se proporcionan ejemplos del uso del Java *binding* de Matisse para abrir una conexión a una base de datos y, dentro de una transacción, y utilizando las funciones anteriores, crear, recuperar, modificar y borrar objetos, y gestionar las relaciones entre ellos.

El siguiente programa abre una conexión con la base de datos y crea un proyecto y varios empleados, tanto de plantilla como no de plantilla, que se asignan al proyecto. Como jefe de proyecto se asigna uno de los empleados de plantilla. Las cardinalidades de las relaciones se verifican al confirmar los cambios en la base de datos. Se produciría una excepción, por ejemplo, si no se asignara un jefe de proyecto a un proyecto. Se resaltan las sentencias para abrir la conexión a la base de datos y crear y confirmar una transacción, y también para gestionar las relaciones entre objetos.

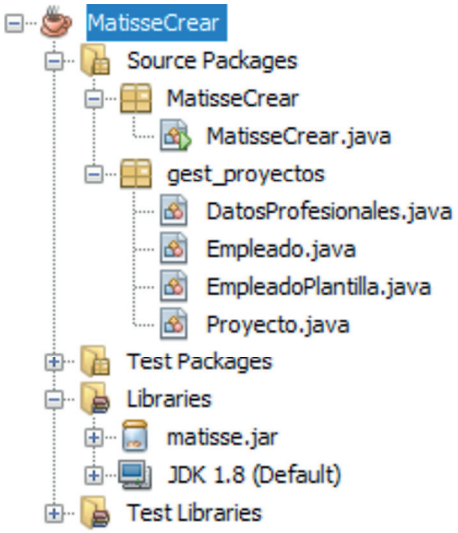


Figura 5.7
Proyecto para Matisse con *stub classes*.

```
// Creación de empleados y proyectos, y asignación de empleados a proyectos
package MatisseCrear;
import com.matisse.MtDatabase;
import com.matisse.MtException;
import gest_proyectos.*;
public class MatisseCrear {
    public static void main(String[] args) {
        try(MtDatabase db = new MtDatabase("localhost", "AcDat_BDO")) {
            db.open();
            db.startTransaction();
            Proyecto p1 = new Proyecto(db);
            p1.setNom_proy("PAPEL ELECTRÓNICO");
            p1.setF_inicio(new java.util.GregorianCalendar(2018,12,01));
            EmpleadoPlantilla jp1 = new EmpleadoPlantilla(db);
            jp1.setDni("78901234X");
            jp1.setNom_emp("NADALES");
            jp1.setNum_emp("604202");
            p1.setJefe_proyecto(jp1);
            Empleado e1=new Empleado(db);
            e1.setDni("56789012B");
            e1.setNom_emp("SAMPER");
            p1.appendTiene_asignado(e1);
            EmpleadoPlantilla e2=new EmpleadoPlantilla(db);
            e2.setDni("76543210S");
            e2.setNom_emp("SILVA");
            e2.setNum_emp("753014");
            DatosProfesionales dp2 = new DatosProfesionales(db);
            dp2.setDni("76543210S");
            dp2.setCategoria("B1");
            dp2.setSueldo_bruto_anual((float) 45200.00);
            e2.setTiene_datos_prof (dp2);
            p1.appendTiene_asignado(e2);
            Empleado e3=new Empleado(db);
```

```
            e3.setDni("89012345E");
            e3.setNom_emp("ROJAS");
            db.commit();
        }
        catch (MtException mte)
        {
            System.out.println("MtException: " + mte.getMessage());
        }
    }
}
```

Actividades propuestas

5.3. Añade un método `float subeSueldoBruto(float incr)` a la clase `Empleado`, al que se le pase el porcentaje de incremento. Si el sueldo bruto está definido, debe devolver el nuevo sueldo, y `null` en otro caso. Escribe y verifica un programa que actualice el sueldo bruto para un empleado utilizando el nuevo método.

5.4. Piensa/investiga/experimenta/verifica: todos los objetos creados por el programa de ejemplo anterior se crean como objetos persistentes. ¿Cómo se crearían objetos transitorios (es decir, no persistentes) de las mismas clases? ¿Es posible que un objeto creado como transitorio se acabe almacenando en la base de datos? ¿En qué casos? Justifica tus respuestas y verifícalas, en su caso, mediante un programa.

5.5. Genera las *stub classes* para el esquema de objetos creado en una actividad anterior para publicaciones. Crea un programa en Java que cree al menos tres libros, dos de ellos del mismo autor, y dos revistas en la base de datos y que, para terminar, muestre toda la información acerca de todos los objetos creados.

El Object Browser del Enterprise Manager de Matisse permite visualizar los objetos creados y las relaciones entre ellos. Se pueden seleccionar objetos mediante consultas en SQL, visualizar sus propiedades y acceder a los objetos relacionados. Como se puede ver, desde un objeto se puede acceder a los objetos relacionados siguiendo los *traversal paths*.

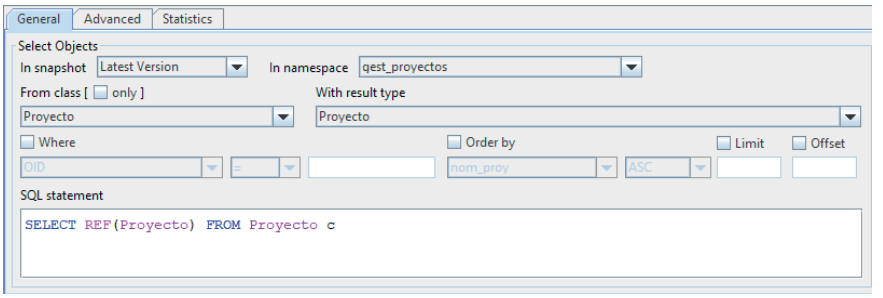
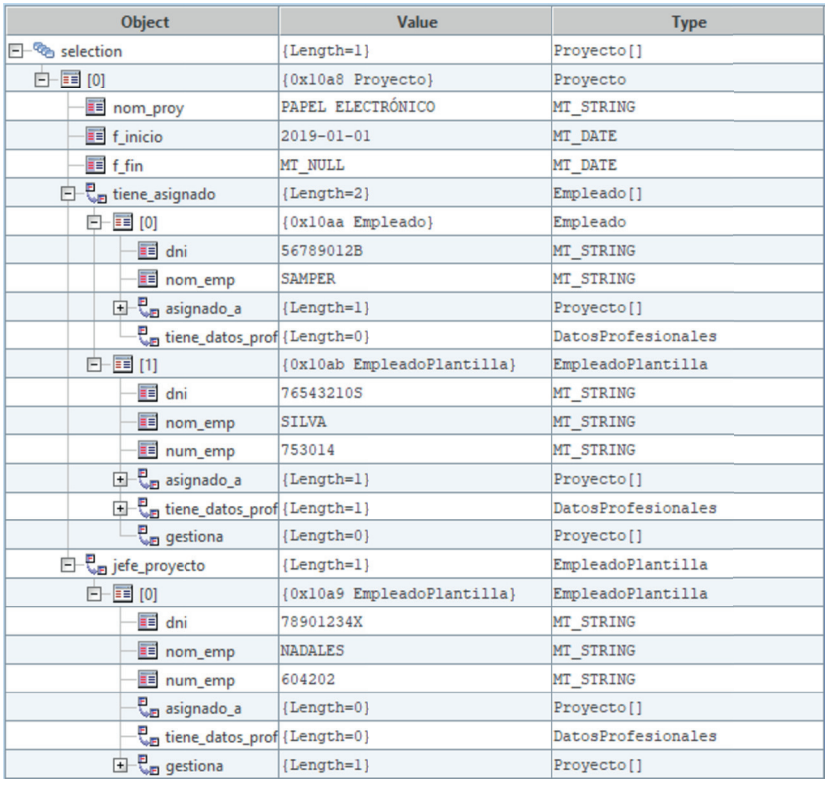


Figura 5.8 Consulta de objetos en el Object Browser de Matisse



Object	Value	Type
selection	{Length=1}	Proyecto[]
[0]	{0x10a8 Proyecto}	Proyecto
nom_proy	PAPEL ELECTRÓNICO	MT_STRING
f_inicio	2019-01-01	MT_DATE
f_fin	MT_NULL	MT_DATE
tiene_asignado	{Length=2}	Empleado[]
[0]	{0x10aa Empleado}	Empleado
dni	56789012B	MT_STRING
nom_emp	SAMPER	MT_STRING
asignado_a	{Length=1}	Proyecto[]
tiene_datos_prof	{Length=0}	DatosProfesionales
[1]	{0x10ab EmpleadoPlantilla}	EmpleadoPlantilla
dni	76543210S	MT_STRING
nom_emp	SILVA	MT_STRING
num_emp	753014	MT_STRING
asignado_a	{Length=1}	Proyecto[]
tiene_datos_prof	{Length=1}	DatosProfesionales
gestiona	{Length=0}	Proyecto[]
jefe_proyecto	{Length=1}	EmpleadoPlantilla
[0]	{0x10a9 EmpleadoPlantilla}	EmpleadoPlantilla
dni	78901234X	MT_STRING
nom_emp	NADALES	MT_STRING
num_emp	604202	MT_STRING
asignado_a	{Length=0}	Proyecto[]
tiene_datos_prof	{Length=0}	DatosProfesionales
gestiona	{Length=1}	Proyecto[]

Figura 5.9
Navegación por los objetos con el Object Browser de Matisse

El siguiente programa borra y modifica algunos objetos y elimina algunas relaciones entre objetos, y utiliza iteradores de varios tipos. Recupera empleados de plantilla a partir de su DNI con `lookupEmpleadoPlantilla_i_dni`, que se ha creado en `EmpleadoPlantilla` porque se ha definido el índice `EmpleadoPlantilla_i_dni`. Recupera empleados a partir de su DNI con `lookup_empleado_pk()`, que se ha creado porque DNI se ha definido como clave primaria de Empleado (`unique_key`). Antes de borrar un empleado, es necesario borrar, si existen, sus datos profesionales. Si no, se produce una excepción, dado que la cardinalidad mínima en el *traversal path* `datos_prof_de` es 1. También obtiene y utiliza iteradores. Primero, obtiene todos los proyectos con `instanceIterator()` de la clase `Proyecto`. Para cada proyecto obtiene los empleados asignados a él con un iterador obtenido con el método `tiene_asignadoIterator()` de la clase `Proyecto`, que se ha creado para el *traversal path* `tiene_asignado` de la clase `Proyecto`.

```
// Borrado objetos y relaciones entre ellos

import com.matisse.MtDatabase;
import com.matisse.MtException;
import com.matisse.MtObjectIterator;
import gest_proyectos.*;

public class MatisseModifBorrar {

    public static void muestraProyecto(Proyecto p) {

        System.out.println("Proyecto "+p.getNom_proy()+
            "[OID: "+p.getMtOidToHexString()+"]");
```

```
            System.out.println("---------");
            System.out.println("Jefe proyecto: DNI: "+p.getJefe_proyecto().
                getDni()+
                ", Nombre: "+p.getJefe_proyecto().getNom_emp());
            System.out.println("Empleados:");
            MtObjectIterator<Empleado> itEmp = p.tiene_asignadoIterator();
            while(itEmp.hasNext()) {
                Empleado e = itEmp.next();
                System.out.println("DNI: "+e.getDni()+", Nombre: "+e.getNom_emp());
            }
        }

        public static void main(String[] args) {
            try(MtDatabase db = new MtDatabase("localhost", "AcDat_BDO")) {
                    db.open();
                    db.startTransaction();
                    Proyecto p = new Proyecto(db);
                    p.setNom_proy("TINTA_HOLOGRÁFICA");
                    p.setF_inicio(new java.util.GregorianCalendar(2018,12,28));
                    EmpleadoPlantilla ep = // NADALES
                    EmpleadoPlantilla.lookupEmpleadoPlantilla_i_dni(db,
                        "78901234X");
                    p.setJefe_proyecto(ep);
                    Empleado e1 = Empleado.lookupEmpleado_pk(db, "89012345E"); //
                        ROJAS
                    e1.setNom_emp("ROSAS");
                    e1.appendAsignado_a(p);
                    Empleado e2 = Empleado.lookupEmpleado_pk(db, "76543210S"); //
                        SILVA
                    e2.getTiene_datos_prof().remove();
                    e2.remove();
                    Empleado e3 = Empleado.lookupEmpleado_pk(db, "56789012B"); //
                        SAMPER
                    e3.clearAsignado_a();
                    MtObjectIterator<Proyecto> itProy = Proyecto.
                        instanceIterator(db);
                    while(itProy.hasNext()) {
                        Proyecto unProy = itProy.next();
                        muestraProyecto(unProy);
                    }
                    db.commit();
                    }
            catch (MtException mte)
            {
                System.out.println("MtException: " + mte.getMessage());
            }
        }
    }
```

Actividades propuestas

5.6. Crea un programa que cree un nuevo libro y le asigne como autor uno de los autores ya existentes. El programa debe también crear un nuevo autor, que no tendrá ningún libro

asociado. El programa debe borrar uno de los dos libros del autor para el que había dos libros. Por último, como comprobación, debe mostrar toda la información acerca de todos los objetos existentes. Se sugiere mostrar autor a autor, y para cada autor sus libros.

5.7. Existe un método `deepRemove()` en cada clase que por defecto llama a `remove()`. Este se puede cambiar, cuando tenga sentido, para borrar objetos subordinados antes de llamar a `remove()` para borrar el propio objeto. Cambia `deepRemove()` en la clase `Empleado` para borrar antes los datos profesionales. Modifica el programa de ejemplo anterior para que utilice este método para borrar el empleado.

5.7.3. Consultas mediante el SQL de Matisse

Matisse no proporciona soporte para OQL conforme al estándar ODMG 3.0. En lugar de ello, tiene un lenguaje para consulta, definición y manejo de datos al que se llama SQL, similar al SQL estándar de las bases de datos relacionales, pero adaptado para BDO, y un *driver* JDBC. El lenguaje SQL de Matisse tiene sentencias `SELECT`, `INSERT`, `UPDATE` y `DELETE`. No hay que dejarse confundir por el nombre SQL ni por su similitud al SQL estándar para bases de datos relacionales, ni porque se denomine a Matisse base de datos posrelacional. Matisse es una BDO, y lo que se llama SQL de Matisse no es realmente SQL. En la parte para desarrolladores de la web de Matisse, para la que se ha proporcionado antes un enlace, se puede encontrar documentación acerca del SQL de Matisse, y acerca del Java *binding* de Matisse y programación en Java para Matisse en general, incluyendo el uso de su *driver* de JDBC.

Recursos web

En los siguientes enlaces se puede encontrar documentación detallada acerca del lenguaje SQL de Matisse y una guía de programación en Java para Matisse que dedica un capítulo específicamente a JDBC.

http://www.matisse.com/pdf/developers/sql_pg.pdf
http://www.matisse.com/pdf/developers/java_pg.pdf

La clase para el *driver* JDBC es `com.matisse.sql.MtDriver`. Se puede cargar el *driver* a la manera antigua con `Class.forName("com.matisse.sql.MtDriver")` o, lo más sencillo, obtener una conexión JDBC con `getJDBCConnection()` de `MtDatabase`. El siguiente programa crea un nuevo empleado y lo asigna a todos los proyectos de los que es jefe un determinado empleado, identificado por su DNI. Para obtener los proyectos como objetos hay que utilizar en la consulta `REF(p)`, que devuelve una referencia a cada objeto de clase `Proyecto`, que se obtiene con `getObject()`. El planteamiento con una base de datos estrictamente conforme a ODMG 3.0 sería análogo: recuperar los proyectos mediante una consulta en OQL, hacer los cambios pertinentes sobre ellos y confirmar los cambios.

```
// Recuperación de datos mediante JDBC y modificaciones sobre datos obtenidos

package MatisseConsultaConJDBCModifConJavaBinding;

import com.matisse.MtDatabase;
import com.matisse.MtException;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import gest_proyectos.*;

public class MatisseConsultaConJDBCModifConJavaBinding {

    public static void muestraErrorSQL(SQLException e) {
        System.err.println("SQL ERROR mensaje: " + e.getMessage());
        System.err.println("SQL Estado: " + e.getSQLState());
        System.err.println("SQL código específico: " + e.getErrorCode());
    }

    public static void main(String[] args) {
        try (
            MtDatabase db = new MtDatabase("localhost", "AcDat_BDO")) {
            db.open();
            db.startTransaction();
            Empleado e = new Empleado(db);
            e.setDni("65432109F");
            e.setNom_emp("LUQUE");
            try (
                Connection jdbcCon = db.getJDBCConnection();
                Statement stmt = jdbcCon.createStatement()) {
                    String commandText = "SELECT REF(p) FROM gest_proyectos.
                        Proyecto p WHERE p.jefe_proyecto.dni='78901234X';";
                    ResultSet rset = stmt.executeQuery(commandText);
                    Proyecto p;
                    while (rset.next()) {
                        p = (Proyecto) rset.getObject(1);
                        System.out.println("Proyecto: "+p.getNom_proy()+
                        ", jefe: ["+p.getJefe_proyecto().getDni()+"] "+
                        p.getJefe_proyecto().getNom_emp());
                        if(p.getJefe_proyecto().getDni().equals("78901234X")) {
                        p.appendTiene_asignado(e);
                        System.out.println("Asignado nuevo empleado.");
                    }
                }
            rset.close();
            stmt.close();
            db.commit();
        } catch (SQLException sqle) {
            muestraErrorSQL(sqle);
        }
        } catch (MtException mte) {
            System.out.println("MtException: " + mte.getMessage());
        }
    }
}
```