Tema 2. Bases de datos orientadas a objetos

Diseño de Sistemas de Bases de Datos
Merche Marqués

12 de abril de 2002

Índice

1. Introducción

Los modelos de bases de datos tradicionales (relacional, red y jerárquico) han sido capaces de satisfacer con éxito las necesidades, en cuanto a bases de datos, de las aplicaciones de gestión tradicionales. Sin embargo, presentan algunas deficiencias cuando se trata de aplicaciones más complejas o sofisticadas como, por ejemplo, el diseño y fabricación en ingeniería (CAD/CAM, CIM), los experimentos científicos, los sistemas de información geográfica o los sistemas multimedia. Los requerimientos y las características de estas nuevas aplicaciones difieren en gran medida de las típicas aplicaciones de gestión: la estructura de los objetos es más compleja, las transacciones son de larga duración, se necesitan nuevos tipos de datos para almacenar imágenes y textos, y hace falta definir operaciones no estándar, específicas para cada aplicación.

Las bases de datos orientadas a objetos se crearon para tratar de satisfacer las necesidades de estas nuevas aplicaciones. La orientación a objetos ofrece flexibilidad para manejar algunos de estos requisitos y no está limitada por los tipos de datos y los lenguajes de consulta de los sistemas de bases de datos tradicionales. Una característica clave de las bases de datos orientadas a objetos es la potencia que proporcionan al diseñador al permitirle especificar tanto la estructura de objetos complejos, como las operaciones que se pueden aplicar sobre dichos objetos.

Otro motivo para la creación de las bases de datos orientadas a objetos es el creciente uso de los lenguajes orientados a objetos para desarrollar aplicaciones. Las bases de datos se han convertido en piezas fundamentales de muchos sistemas de información y las bases de datos tradicionales son difíciles de utilizar cuando las aplicaciones que acceden a ellas están escritas en un lenguaje de programación orientado a objetos como C++, Smalltalk o Java. Las bases de datos orientadas a objetos se han diseñado para que se puedan integrar directamente con aplicaciones desarrolladas con lenguajes orientados a objetos, habiendo adoptado muchos de los conceptos de estos lenguajes.

Los fabricantes de los SGBD relacionales también se han dado cuenta de las nuevas necesidades en el modelado de datos, por lo que las nuevas versiones de sus sistemas incorporan muchos de los rasgos propuestos para las bases de datos orientadas a objetos, como ha ocurrido con Informix y Oracle. Esto ha dado lugar al modelo relacional extendido y a los sistemas que lo implementan se les denomina sistemas objeto–relacionales. La nueva versión de SQL, SQL:1999[1], incluye algunas de las características de la orientación a objetos.

Durante los últimos años se han creado muchos prototipos experimentales de sistemas de bases de datos orientadas a objetos y también muchos sistemas comerciales. Conforme éstos fueron apareciendo, surgió la necesidad de establecer un modelo estándar y un lenguaje. Para ello, los fabricantes de los SGBD orientadas a objetos formaron un grupo denominado

[1]Este es el nombre que recibe el estándar. En ocasiones se cita como SQL3 porque así se llamaba el proyecto que lo desarrolló. También se cita como SQL99, por ser un nombre similar al de la versión anterior, SQL92; sin embargo, este último nombre no se ha utilizado en esta ocasión porque se quiere evitar el efecto 2000 en el nombre de futuras versiones.

ODMG (*Object Database Management Group*), que propuso el estándar ODMG–93 y que ha ido evolucionando hasta el ODMG 3.0, su última versión. El uso de estándares proporciona portabilidad, permitiendo que una aplicación se pueda ejecutar sobre sistemas distintos con mínimas modificaciones. Los estándares también proporcionan interoperabilidad, permitiendo que una aplicación pueda acceder a varios sistemas diferentes. Y una tercera ventaja de los estándares es que permiten que los usuarios puedan comparar entre distintos sistemas comerciales, dependiendo de qué partes del estándar proporcionan.

2. Conceptos de orientación a objetos

El desarrollo del paradigma orientado a objetos aporta un gran cambio en el modo en que vemos los datos y los procedimientos que actúan sobre ellos. Tradicionalemente, los datos y los procedimientos se han almacenado separadamente: los datos y sus relaciones en la base de datos y los procedimientos en los programas de aplicación. La orientación a objetos, sin embargo, combina los procedimientos de una entidad con sus datos.

Esta combinación se considera como un paso adelante en la gestión de datos. Las entidades son unidades autocontenidas que se pueden reutilizar con relativa facilidad. En lugar de ligar el comportamiento de una entidad a un progama de aplicación, el comportamiento es parte de la entidad en sí, por lo en cualquier lugar en el que se utilice la entidad, se comporta de un modo predecible y conocido.

El modelo orientado a objetos también soporta relaciones de muchos a muchos, siendo el primer modelo que lo permite. Aún así se debe ser muy cuidadoso cuando se diseñan estas relaciones para evitar pérdidas de información.

Por otra parte, las bases de datos orientadas a objetos son navegacionales: el acceso a los datos es a través de las relaciones, que se almacenan con los mismos datos. Esto se considera un paso atrás. Las bases de datos orientadas a objetos no son apropiadas para realizar consultas *ad hoc*, al contrario que las bases de datos relacionales, aunque normalmente las soportan. La naturaleza navegacional de las bases de datos orientadas a objetos implica que las consultas deben seguir relaciones predefinidas y que no pueden insertarse nuevas relaciones "al vuelo".

No parece que las bases de datos orientadas a objetos vayan a reemplazar a las bases de datos relacionales en todas las aplicaciones del mismo modo en que éstas reemplazaron a sus predecesoras.

Los objetos han entrado en el mundo de las bases de datos de formas:

- SGBD orientados a objetos puros: son SGBD basados completamente en el modelo orientado a objetos.
- SGBD híbridos u objeto–relacionales: son SGBD relacionales que permiten almacenar

objetos en sus relaciones (tablas).

A continuación se definen los conceptos del paradigma orientado a objetos en programación, ya que el modelo de datos orientado a objetos es una extensión del mismo.

Objeto. Es un elemento autocontenido utilizado por el programa. Los valores que almacena un objeto se denominan atributos, variables o propiedades. Los objetos pueden realizar acciones, que se denominan métodos, servicios, funciones, procedimientos u operaciones. Los objetos tienen un gran sentido de la privacidad, por lo que sólo dan información sobre sí mismos a través de los métodos que poseen para compartir su infomación. También ocultan la implementación de sus procedimientos, aunque es muy sencillo pedirles que los ejecuten. Los usuarios y los programas de aplicación no pueden ver qué hay dentro de los métodos, sólo pueden ver los resultados de ejecutarlos. A esto es a lo que se denomina ocultación de información o encapsulamiento de datos. Cada objeto presenta una interface pública al resto de objetos que pueden utilizarlo. Una de las mayores ventajas del encapsulamiento es que mientras que la interface pública sea la misma, se puede cambiar la implementación de los métodos sin que sea necesario informar al resto de objetos que los utilizan. Para pedir datos a un objeto o que éste realice una acción se le debe enviar un mensaje. Un programa orientado a objetos es un conjunto de objetos que tienen atributos y métodos. Los objetos interactúan enviándose mensajes. La clave, por supuesto, es averiguar qué objetos necesita el programa y cuáles deben ser sus atributos y sus métodos.

Clase. Es un patrón o plantilla en la que se basan objetos que son similares. Cuando un programa crea un objeto de una clase, proporciona datos para sus variables y el objeto puede entonces utilizar los métodos que se han escrito para la clase. Todos los objetos creados a partir de la misma clase comparten los mismos procedimientos para sus métodos, también tienen los mismos tipos para sus datos, pero los valores pueden diferir. Una clase también es un tipo de datos. De hecho una clase es una implementación de lo que se conoce como un tipo abstracto de datos. El que una clase sea también un tipo de datos significa que una clase se puede utilizar como tipo de datos de un atributo.

Tipos de clases. En los programas orientados a objetos hay tres tipos de clases: clases de control, clases entidad y clases interface.

- Las clases de control gestionan el flujo de operación de un programa (por ejemplo, el programa que se ejecuta es un objeto de esta clase).
- Las clases entidad son las que se utilizan para crear objetos que manejan datos (por ejemplo, clases para personas, objetos tangibles o eventos).
- Las clases interface son las que manejan la entrada y la salida de información (por ejemplo, las ventanas gráficas y los menús utilizados por un programa).

En los programas orientados a objetos, las clases entidad no hacen su propia entrada/salida. El teclado es manejado por objetos interface que recogen los datos y los

envían a los objetos entidad para que los almacenen y los procesen. La salida impresa y por pantalla la formatea un objeto interface para obtener los datos a visualizar de los objetos entidad. Cuando los objetos entidad forman parte de la base de datos, es el SGBD el que se encarga de la entrada/salida a ficheros. El resto de la entrada/salida la manejan los programas de aplicación o las utilidades del SGBD. Muchos programas orientados a objetos tienen un cuarto tipo de clase: la clase contenedor. Estas clases contienen, o manejan, múltiples objetos creados a partir del mismo tipo de clase. También se conocen como agregaciones. Las clases contenedor mantienen los objetos en algún orden, los listan e incluso pueden permitir búsquedas en ellos. Muchos SGBD orientados a objetos llaman a sus clases contenedor *extents* (extensiones) y su objetivo es permitir el acceso a todos los objetos creados a partir de la misma clase.

Tipos de métodos. Hay varios tipos de métodos que son comunes a la mayoría de las clases:

- Constructores. Un contructor es un método que tiene el mismo nombre que la clase. Se ejecuta cuando se crea un objeto de una clase. Por lo tanto, un constructor contiene instrucciones para inicializar las variables de un objeto.
- Destructores. Un destructor es un método que se utiliza para destruir un objeto. No todos los lenguajes orientados a objetos poseen destructores.
- Accesores. Un accesor es un método que devuelve el valor de un atributo privado de otro objeto. Así es cómo los objetos externos pueden acceder a los datos encapsulados.
- Mutadores. Un mutador es un método que almacena un nuevo valor en un atributo. De este modo es cómo objetos externos pueden modificar los datos encapsulados.

Además, cada clase tendrá otros métodos dependiendo del comportamiento específico que deba poseer.

Sobrecarga de métodos. Una de las características de las clases es que pueden tener métodos sobrecargados, que son métodos que tienen el mismo nombre pero que necesitan distintos datos para operar. Ya que los datos son distintos, las interfaces públicas de los métodos serán diferentes. Por ejemplo, consideremos una clase contenedor, `TodosLosEmpleados`, que agrega todos los objetos creados de la clase `Empleado`. Para que la clase contenedor sea útil, debe proporcionar alguna forma de buscar objetos de empleados específicos. Se puede querer buscar por número de empleado, por el nombre y los apellidos o por el número de teléfono. La clase contenedor `TodosLosEmpleados` tendrá tres métodos llamados `encuentra`. Uno de los métodos requiere un entero como parámetro (el número de empleado), el segundo require dos cadenas (el nombre y los apellidos) y el tercero requiere una sola cadena (el número de teléfono). Aunque los tres métodos tienen el mismo nombre, poseen distintas interfaces públicas. La ventaja de la sobrecarga de los métodos es que presentan una interface consistente al programador: siempre que quiera localizar a un empleado, debe utilizar el método `encuentra`.

Nombres de clases, atributos y métodos. En el mundo de la orientación a objetos hay cierta uniformidad en el modo de dar nombres a clases, atributos y métodos.

- Los nombres de las clases empiezan por una letra mayúscula seguida de minúsculas. Si el nombre tiene más de una palabra, se puede usar el carácter subrayado para separar palabras o bien empezar cada una con una letra mayúscula (`Materia_prima` o `MateriaPrima`).
- Los nombres de los atributos y de los métodos empiezan por minúscula y si tienen más de una palabra, utilizan el subrayado o la mayúscula (`num_empleado` o `numEmpleado`).
- Los métodos accesores empiezan por la palabra `get` seguida del nombre del atributo al que acceden (`getNumEmpleado`).
- Los métodos mutadores empiezan por la palabra `set` seguida del nombre del atributo cuyo valor modifican (`setNumEmpleado`).

Herencia de atributos. En ocasiones se necesita trabajar con clases que son similares pero no idénticas. Para ello es muy útil una de las características del paradigma orientado a objetos: la herencia. Una clase puede tener varias subclases que representan ocurrencias más específicas de la súperclase. Por ejemplo, podemos tener la clase (súperclase) `Animal` con sus atributos (nombre común, nombre científico, fecha de nacimiento y género) y las subclases `Mamífero`, `Reptil` y `Pez`, cada una con unos atributos específicos (`Mamífero`: peso, altura del hombro, raza y color; `Reptil`: longitud actual y longitud máxima; `Pez`: color). Por el hecho de ser subclases de `Animal`, heredan sus atributos. La relación que matienen las subclases con la súperclase es del tipo "es un": un mamífero es un animal, un reptil es un animal y un pez es un animal. No todas las clases de una jerarquía se utilizan para crear objetos. Por ejemplo, nunca se crean objetos de la clase `Animal`, sino que se crean objetos de las clases `Mamífero`, `Reptil` o `Pez`. La clase `Animal` sólo se utiliza para recoger los atributos y métodos que son comunes a las tres subclases. Se dice que estas clases son abstractas o virtuales. Las clases que se utilizan para crear objetos se denominan clases concretas.

Herencia múltiple. Cuando una clase hereda de más de una súperclase se tiene herencia múltiple.

Interfaces. Algunos lenguajes orientados a objetos no soportan la herencia múltiple. En lugar de eso permiten que una clase se derive de una sola clase pero permiten que la clase implemente múltiples interfaces. Una interface es una especificación para una clase sin instrucciones en los métodos. Cada clase que implemente la interface proporcionará las instrucciones para cada método de la misma. Una interface puede contener atributos y métodos, o bien sólo atributos, o bien sólo métodos.

Polimorfismo. En general, las subclases heredan los métodos de sus súperclases y los utilizan como si fueran suyos. Sin embargo, en algunas ocasiones no es posible escribir un método genérico que pueda ser usado por todas las subclases. La clase `ObjetoGeométrico` posee un método `área` que deberá tener distinta implementación

para sus subclases `Círculo`, `Rectángulo` y `Triángulo`. La súperclase contendrá un prototipo para el método que calcula el área, indicando sólo su interface pública. Cada subclase redefine el método, añadiendo las instrucciones necesarias para calcular su área. Nótese que polimorfismo no es lo mismo que sobrecarga: la sobrecarga se aplica a métodos de la misma clase que tienen el mismo nombre y distintas signaturas, mientras que el polimorfismo se aplica a varias subclases de la misma súperclase que tienen métodos con la misma signatura y con distintas implementaciones.

A continuación se citan las ventajas de la orientación a objetos en programación:

- Un programa orientado a objetos consta de módulos independientes, por lo que se pueden reutilizar en distintos programas, ahorrando tiempo de desarrollo.
- El interior de una clase se puede modificar como sea necesario siempre que su interface pública no cambie, de modo que estas modificaciones no afectarán a los programas que utilizan la clase.
- Los programas orientados a objetos separan la interface de usuario de la gestión de los datos, haciendo posible la modificación de una independientemente de la otra.
- La herencia añade una estructura lógica al programa relacionando clases desde lo general a lo más específico, haciendo que el programa sea más fácil de entender y, por lo tanto, más fácil de mantener.

3. El modelo de datos orientado a objetos

El modelo de datos orientado a objetos es una extensión del paradigma de programación orientado a objetos. Los objetos entidad que se utilizan en los programas orientados a objetos son análogos a las entidades que se utilizan en las bases de datos orientadas a objetos puras, pero con una gran diferencia: los objetos del programa desaparecen cuando el programa termina su ejecución, mientras que los objetos de la base de datos permanecen. A esto se le denomina persistencia.

3.1. Relaciones

Las bases de datos relacionales representan las relaciones mediante las claves ajenas. No tienen estructuras de datos que formen parte de la base de datos y que representen estos enlaces entre tablas. Las relaciones se utilizan para hacer concatenaciones (*join*) de tablas. Por el contrario, las bases de datos orientadas a objetos implementan sus relaciones incluyendo en cada objeto los identificadores de los objetos con los que se relaciona.

Un identificador de objeto es un atributo interno que posee cada objeto. Ni los programadores, ni los usuarios que realizan consultas de forma interactiva, ven o manipulan estos identificadores directamente. Los identificadores de los objetos los asigna el SGBD y es él el único que los utiliza.

El identificador puede ser un valor arbitrario o puede incluir la información necesaria para localizar el objeto en el fichero donde se almacena la base de datos. Por ejemplo, el identificador puede contener el número de la página del fichero donde se encuentra almacenado el objeto, junto con el desplazamiento desde el principio de la página.

Hay dos aspectos importantes a destacar sobre este método de representar las relaciones entre datos:

- Para que el mecanismo funcione, el identificador del objeto no debe cambiar mientras éste forme parte de la base de datos.
- Las únicas relaciones que se pueden utilizar para consultar la base de datos son aquellas que se han predefinido almacenando en atributos los identificadores de los objetos relacionados. Por lo tanto, una base de datos orientada a objetos pura es navegacional, como los modelos prerrelacionales (el modelo jerárquico y el modelo de red). De este modo se limita la flexibilidad del programador/usuario a aquellas relaciones predefinidas, pero los accesos que siguen estas relaciones presentan mejores prestaciones que en las bases de datos relacionales porque es más rápido seguir los identificadores de los objetos que hacer operaciones de concatenación (*join*).

El modelo orientado a objetos permite los atributos multivaluados, agregaciones a las que se denomina conjuntos (*sets*) o bolsas (*bags*). Para crear una relación de uno a muchos, se define un atributo en la parte del uno que será de la clase del objeto con el que se relaciona. Este atributo contendrá el identificador de objeto del padre. La clase del objeto padre contendrá un atributo que almacenará un conjunto de valores: los identificadores de los objetos hijo con los que se relaciona. Cuando el SGBD ve que un atributo tiene como tipo de datos una clase, ya sabe que el atributo contendrá un indentificador de objeto.

Las relaciones de muchos a muchos se pueden representar directamente en las bases de datos orientadas a objetos, sin necesidad de crear entidades intermedias. Para representar la relación, cada clase que participa en ella define un aributo que contendrá un conjunto de valores de la otra clase con la que se relacionará. Aunque el hecho de poder representar relaciones de muchos a muchos parece aportar muchas ventajas, hay que tener mucho cuidado cuando se utilizan. En primer lugar, si la relación tiene datos, será necesario crear una entidad intermedia que contenga estos datos. Por ejemplo, en la relación de los artículos con los proveedores, en donde cada proveedor puede tener un precio distinto para un mismo artículo. En este caso, la relación de muchos a muchos se sustituye por dos relaciones de uno a muchos, como se haría en una base de datos relacional. En segundo lugar, se puede diseñar una base de datos que contiene relaciones de muchos a muchos en donde o bien se pierde

información, o bien se hace imposible determinar las relaciones con precisión. También en estos casos la solución es incluir una entidad intermedia que represente la relación.

Ya que el paradigma orientado a objetos soporta la herencia, una base de datos orientada a objetos también puede utilizar la relación "es un" entre objetos. Por ejemplo, en una base de datos para un departamento de recursos humanos habrá una clase genérica `Empleado` con diversos atributos: nombre, dirección, número de la seguridad social, fecha de contrato y departamento en el que trabaja. Sin embargo, para registrar el modo de pago de cada empleado hay un dilema. No a todos los empleados se les paga del mismo modo: a algunos se les paga por horas, mientras que otros tienen un salario mensual. La clase de los empleados que trabajan por horas necesita unos atributos distintos que la clase de los otros empleados. En una base de datos orientada a objetos se deben crear las dos subclases de empleados. Aunque el SGBD nunca creará objetos de la clase `Empleado`, su presencia en el diseño clarifica el diseño lógico de la base de datos y ayuda a los programadores de aplicaciones permitiéndoles escribir sólo una vez los métodos que tienen en común las dos subclases (serán los métodos que se sitúan en la clase `Empleado`).

En teoría, una base de datos orientada a objetos debe soportar dos tipos de herencia: la relación "es un" y la relación "extiende". La relación "es un", que también se conoce como *generalización–especialización*, crea una jerarquía donde las subclases son tipos específicos de las súperclases. Con la relación "extiende", sin embargo, una clase expande su súperclase en lugar de estrecharla en un tipo más específico. Por ejemplo, en la jerarquía de la clase `Empleado`, al igual que son necesarias clases para los empleados que realizan cada trabajo específico, hace falta guardar información adicional sobre los directores, que son empleados pero que también tienen unas características específicas. La base de datos incluirá una clase `Director` con un atributo para los empleados a los que dirige. En este sentido un director no es un empleado más específico sino un empleado con información adicional.

Una de las cosas que es difícil de manejar en las bases de datos relacionales es la idea de las partes de un todo, como en una base de datos de fabricación, en la que hace falta saber qué piezas y qué componentes se utilizan para fabricar un determinado producto. Sin embargo, una base de datos orientada a objetos puede aprovechar la relación denominada "todo–parte" en la que los objetos de una clase se relacionan con objetos de otras clases que forman parte de él. En el caso de la base de datos de fabricación, la clase `Producto` se relacionará con las clases `Pieza` y `Componente` utilizando una relación "todo–parte". Este tipo de relación es una relación de muchos a muchos con un significado especial. Un producto puede estar hecho de muchas piezas y muchos componentes. Además, una misma pieza o un mismo componente se puede utilizar para fabricar distintos productos. El identificar esta relación como "todo–parte" permite que el diseño sea más fácil de entender.

3.2. Integridad de las relaciones

Para que las relaciones funcionen en una base de datos orientada a objetos pura, los identificadores de los objetos deben corresponderse en ambos extremos de la relación. Por

ejemplo, si los aparejadores de una empresa de control de calidad se deben relacionar con las obras de construcción que supervisan, debe haber algún modo de garantizar que, cuando un identificador de un objeto `Obra` se incluye en un objeto `Aparejador`, el identificador de este mismo objeto `Aparejador` se debe incluir en el objeto `Obra` anterior. Este tipo de integridad de relaciones, que es de algún modo análogo a la integridad referencial en las bases de datos relacionales, se gestiona especificando relaciones inversas.

La clase `Aparejador` tiene un atributo de tipo conjunto llamado `supervisa`. Del mismo modo, la clase `Obra` tiene un atributo llamado `es_supervisada`. Para garantizar la integridad de esta relación, un SGBD orientado a objetos puro deberá permitir que el diseñador de la base de datos pueda especificar dónde debe aparecer el identificador del objeto inverso, como por ejemplo:

```
relationship set<Obra> supervisa
        inverse Obra::es_supervisada
```

en la clase `Aparejador` y:

```
relationship Aparejador es_supervisada
        inverse Aparejador::supervisa
```

en la clase `Obra`.

Siempre que un usuario o un programa de aplicación inserta o elimina un identificador de objeto de la relación `supervisa` en un objeto `Aparejador`, el SGBD actualizará automáticamente la relación `es_supervisada` en el objeto `Obra` relacionado. Cuando se hace una modificación en el objeto `Obra`, el SGBD lo propagará automáticamente al objeto `Aparejador`.

Del mismo modo que en las bases de datos relacionales es el diseñador de la base de datos el que debe especificar las reglas de integridad referencial, en las bases de datos orientadas a objetos es también el diseñador el que debe especificar las relaciones inversas cuando crea el esquema de la base de datos.

3.3. UML

Existen distintas notaciones o modelos para diseñar esquemas conceptuales de bases de datos orientadas a objetos: la notación de Coad/Yourdon, la Shlaer/Mellor, la OMT (Rombaugh) o la de Booch. Cada modelo presenta distintas deficiencias, por lo que algunos de sus autores han desarrollado conjuntamente un lenguaje, denominado UML (*Unified Modeling Language*), que las evita.

"La notación UML (no hay que confundir con las metodologías que utilizan dicha notación), se ha convertido desde finales de los 90 en un estándar para modelar con tecnología orientada a objetos todos aquellos elementos que configuran la arquitectura de un sistema de información y, por extensión, de los procesos de negocio de una organización. De la misma manera que los planos de un arquitecto disponen el esquema director a partir del cual levantamos un edificio, los diagramas UML suministran un modelo de referencia para formalizar los procesos, reglas de negocio, objetos y componentes de una organización. La interacción de todos estos elementos es una representación de nuestra realidad." Extraído de <http://www.vico.org/UMLguiavisual/>).

El estudio y uso de este lenguaje se realiza en la asignatura obligatoria *Ingeniería del Software*, del segundo ciclo de Ingeniería Informática.

4. El modelo estándar ODMG

Un grupo de representantes de la industria de las bases de datos formaron el ODMG (*Object Database Management Group*) con el propósito de definir estándares para los SGBD orientados a objetos. Este grupo propuso un modelo estándar para la semántica de los objetos de una base de datos. Su última versión, ODMG 3.0, apareció en enero de 2000. Los principales componentes de la arquitectura ODMG para un SGBD orientado a objetos son los siguientes:

- Modelo de objetos.
- Lenguaje de definición de objetos (ODL).
- Lenguaje de consulta de objetos (OQL).
- Conexión con los lenguajes C++, Smalltalk y Java.

4.1. 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 y los literales se categorizan en *tipos*. Cada tipo tiene un dominio específico compartido por todos los objetos y literales de ese tipo. Los tipos también pueden tener comportamientos. Cuando un tipo tiene comportamientos, todos los objetos de ese tipo comparten los mismos comportamientos. En el sentido práctico, un tipo puede ser una clase de la que se crea un objeto, una interface o un tipo de datos para un literal (por ejemplo, *integer*). Un objeto se puede pensar como una *instancia* de un tipo.
- Lo que un objeto sabe hacer son sus *operaciones*. Cada operación puede requerir datos de entrada (*parámetros de entrada*) y puede devolver algún valor de un tipo conocido.
- Los objetos tienen *propiedades*, que incluyen sus *atributos* y las *relaciones* que tienen con otros objetos. El *estado* actual de un objeto viene dado por los valores actuales de sus propiedades.
- Una *base de datos* es un conjunto de objetos almacenados que se gestionan de modo que puedan ser accedidos por múltiples usuarios y aplicaciones.
- La definición de una base de datos está contenida en un *esquema* que se ha creado mediante el lenguaje de definición de objetos ODL (*Object Definition Language*) que es el lenguaje de manejo de datos que se ha definido como parte del estándar propuesto para las bases de datos orientadas a objetos.

4.1.1. Objetos

Los tipos de objetos se descomponenen en atómicos, colecciones y tipos estructurados. Los tipos *colección*, que se derivan de la interface `Collection`, son la propuesta del estándar para las clases contenedor. Los objetos colección identificados por el estándar son los siguientes:

`Set<tipo>` : es un grupo desordenado de objetos del mismo tipo. No se permiten duplicados.

`Bag<tipo>` : es un grupo desordenado de objetos del mismo tipo. Se permiten duplicados.

`List<tipo>` : es un grupo ordenado de objetos del mismo tipo. Se permiten duplicados.

`Array<tipo>` : es un grupo ordenado de objetos del mismo tipo que se pueden acceder por su posición. Su tamaño es dinámico y los elementos se pueden insertar y borrar de cualquier posición.

`Dictionary<clave,valor>` : es como un índice. Esta formado por claves ordenadas, cada una de ellas emparejada con un solo valor.

Los tipos *estructurados* son los siguientes:

`Date` : es una fecha del calendario (día, mes y año).

`Time` : es una hora (hora, minutos y segundos).

`Timestamp` : es una hora de una fecha (con precisión de microsegundos).

`Interval` : es un período de tiempo.

Estos tipos tienen la misma definición que los tipos con el mismo nombre del estándar de SLQ.

Los objetos se crean utilizando el método `new()`. Además, todos heredan la interface que se muestra a continuación:

```
interface Object {
    enum      Lock_Type{read,write,upgrade};
    void      lock(in Lock_Type mode) raises(LockNotGranted);
    boolean   try_lock(in Lock_Type mode);
    boolean   same_as(in Object anObject);
    Object    copy();
    void      delete();
};
```

Cada objeto tiene un *identificador de objeto* único generado por el SGBD, que no cambia y que no se reutiliza cuando el objeto se borra. Cada SGBD genera los identificadores siguiendo sus propios criterios.

Los objetos pueden ser *transitorios* o *persistentes*. Los objetos transitorios existen mientras vive el programa de aplicación que los ha creado. Estos objetos se usan tanto como almacenamiento temporal como para dar apoyo al programa de aplicación que se está ejecutando. Los objetos persistentes son aquellos que se almacenan en la base de datos.

4.1.2. Literales

Los tipos literales se descomponen en atómicos, colecciones, estructurados o nulos. Los literales no tienen identificadores y no pueden aparecer solos como objetos, sino que están embebidos en objetos y no pueden referenciarse de modo individual. Los literales atómicos son los siguientes:

`boolean` : un valor que es verdadero o falso.

`short` : un entero con signo, normalmente de 8 o 16 bits.

`long` : un entero con signo, normalmente de 32 o 64 bits.

`unsigned short` : un entero sin signo, normalmente de 8 o 16 bits.

`unsigned long` : un entero sin signo, normalmente de 32 o 64 bits.

`float` : un valor real en coma flotante de simple precisión.

`double` : un valor real en coma flotante de doble precisión.

`octet` : un almacén de 8 bits.

`char` : un carácter ASCII o UNICODE.

`string` : una cadena de caracteres.

`enum` : un tipo enumerado donde los valores se especifican explícitamente cuando se declara el tipo.

Los literales estructurados contienen un número fijo de elementos heterogéneos. Cada elemento es un par `<nombre, valor>` donde `valor` puede ser cualquier tipo literal. Los tipos estructurados son: `date`, `time`, `timestamp`, `interval` y `struct`. Y los tipos colección son: `set<tipo>`, `bag<tipo>`, `list<tipo>`, `array<tipo>` y `dictionary<clave,valor>`.

4.1.3. Tipos

Una de las características más importantes del paradigma orientado a objetos es la distinción entre la interface pública de una clase y sus elementos privados (encapsulación). El estándar propuesto hace esta distinción hablando de la *especificación externa* de un tipo y de sus *implementaciones*.

Una *interface* es una especificación del comportamiento abstracto de un tipo de objeto y contiene las signaturas de las operaciones. Aunque una interface puede tener propiedades (atributos y relaciones) como parte de su especificación, éstas no pueden ser heredadas desde la interface. Además, una interface no es instanciable por lo que no se pueden crear objetos a partir de ella (es el equivalente de una *clase abstracta* en la mayoría de los lenguajes de programación).

Una *clase* es una especificación del comportamiento abstracto y del estado abstracto de un tipo de objeto. Las clases son instanciables, por lo que a partir de ellas se pueden crear instancias de objetos individuales (es el equivalente a una *clase concreta* en los lenguajes de programación).

El estándar propuesto soporta la herencia simple y la herencia múltiple mediante las interfaces. Ya que las interfaces no son instanciables, se suelen utilizar para especificar operaciones abstractas que pueden ser heredadas por clases o por otras interfaces. A esto se le denomina *herencia de comportamiento* y se especifica mediante el símbolo ":". La herencia de comportamiento requiere que el súpertipo sea una interface, mientras que el subtipo puede ser una clase o una interface.

La herencia es una relación "es un":

```
interface ArticuloVenta ...;
interface Mueble : ArticuloVenta ...;
class Silla : Mueble ...;
class Mesa : Mueble ...;
class Sofá : Mueble ...;
```

La interface o clase más baja de la jerarquía es el *tipo más específico*. Ya que hereda los comportamientos de todos los tipos que tiene por encima en la jerarquía, es la interface o clase más completa. En el ejemplo anterior, los tipos más específicos son `Silla`, `Mesa` y `Sofá`.

Uno de los beneficios prácticos de la herencia es que se puede hacer referencia a los subtipos como su súpertipo. Por ejemplo, un programa de aplicación puede hacer referencia a sillas, mesas y sofás como muebles o incluso como artículos de venta. Esto hace que sea más sencillo tratar los subtipos como un grupo cuando sea necesario.

Los subtipos se pueden especializar como sea necesario añadiéndoles comportamientos. Los subtipos de un subtipo especializado heredan también los comportamientos añadidos.

El modelo orientado a objetos utiliza la relación *extiende* (`extends`) para indicar la herencia de estado y de comportamiento. En este tipo de herencia tanto el subtipo como el súpertipo deben ser clases. Las clases que extienden a otra clase ganan acceso a todos los estados y comportamientos del súpertipo, incluyendo cualquier cosa que el súpertipo haya adquirido a través de la herencia de otras interfaces.

Una clase puede extender, como máximo, a otra clase. Sin embargo, si se construye una jerarquía de extensiones, las clases de más abajo en la jerarquía heredan todo lo que sus súpertipos heredan de las clases que tienen por encima.

El modelo permite al diseñador que declare una extensión (`extent`) para cada tipo de objeto definido como una clase. La extensión de un tipo tiene un nombre e incluye todas las instancias de objetos persistentes creadas a partir de dicho tipo. Declarar una extensión denominada `empleados` para el tipo de objeto `Empleado` es similar a crear un objeto de tipo `Set<Empleado>` denominado `empleados`. Una extensión se puede indexar para que el acceso a su contenido sea más rápido.

Una clase con una extensión puede tener una o más *claves* (`key`). Una clave es un identificador único. Cuando una clave está formada por una sola propiedad, es una *clave simple*; si está formada por varias propiedades, es una *clave compuesta*. A diferencia del modelo relacional, las claves únicas no son un requisito.

Una implementación de un tipo consta de dos partes: la representación y los métodos. La *representación* es una estructura de datos dependiente de un lenguaje de programación que contiene las propiedades del tipo. Las especificaciones de la implementación vienen de una conexión con un lenguaje (*language binding*). Esto quiere decir que la representación interna de un tipo será diferente dependiendo del lenguaje de programación que se utilice y

que un mismo tipo puede tener más de una representación.

Los detalles de las operaciones de un tipo se especifican mediante un conjunto de *métodos*. En la especificación externa de cada operación debe haber al menos un método. Sin embargo, un tipo puede incluir métodos que nunca se ven desde fuera del tipo. Estos métodos son los que realizan algunas funciones necesarias para otros métodos del tipo.

Los métodos se escribirán en el mismo lenguaje de programación utilizado para expresar la representación del tipo. Si una base de datos soporta aplicaciones programadas en C++, Java y Smalltalk, entonces será necesario tener tres implementaciones para cada tipo, una para cada lenguaje, aunque cada programa de aplicación utilizará sólo la implementación que le corresponda.

4.1.4. Propiedades

El modelo de objetos ODMG define dos tipos de propiedades: *atributos* y *relaciones*. Un atributo se define del tipo de un objeto. Un atributo no es un objeto de "primera clase", por lo tanto no tiene identificador, pero toma como valor un literal o el identificador de un objeto.

Las relaciones se definen entre tipos. El modelo actual sólo soporta relaciones binarias con cardinalidad 1:1, 1:n y n:m. Una relación no tiene nombre y tampoco es un objeto de "primera clase", pero define caminos transversales en la interface de cada dirección. En el lado del muchos de la relación, los objetos pueden estar desordenados (`set` o `bag`) u ordenados (`list`). La integridad de las relaciones la mantiene automáticamente el SGBD y se genera una excepción cuando se intenta atravesar una relación en la que uno de los objetos participantes se ha borrado. El modelo aporta operaciones para formar (`form`) y eliminar (`drop`) miembros de una relación.

4.1.5. Transacciones

El modelo estándar soporta el concepto de transacciones, que son unidades lógicas de trabajo que llevan a la base de datos de un estado consistente a otro estado consistente. El modelo asume una secuencia lineal de transacciones que se ejecutan de modo controlado. La concurrencia se basa en bloqueos estándar de lectura/escritura con un protcolo pesimista de control de concurrencia. Todos los accesos, creación, modificación y borrado de objetos persistentes se deben realizar dentro de una transacción. El modelo especifica operaciones para iniciar, terminar (*commit*) y abortar transacciones, así como la operación de *checkpoint*. Esta última operación hace permanentes los cambios realizados por la transacción en curso sin liberar ninguno de los bloqueos adquiridos.

4.2. Lenguaje de definición de objetos ODL

ODL es un lenguaje de especificación para definir tipos de objetos 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. La sintaxis de ODL extiende el lenguaje de definición de interfaces (IDL) de la arquitectura CORBA (*Common Object Request Broker Architecture*). El uso de ODL se muestra mediante un ejemplo:

```
class Persona
    (extent personas key dni)
{
/* Definición de atributos */
    attribute struct Nom_Persona {string nombre_pila, string apellido1,
              string apellido2} nombre;
    attribute string dni;
    attribute date fecha_nacim;
    attribute enum Genero{F,M} sexo;
    attribute struct Direccion {string calle, string cp, string ciudad}
              direccion;
/* Definición de operaciones */
    float edad();
}


class Profesor extends Persona
    (extent profesores)
{
/* Definición de atributos */
    attribute string categoria;
    attribute float salario;
    attribute string despacho;
    attributo string telefono;
/* Definición de relaciones */
    relationship Departamento trabaja_en
                 inverse Departamento::tiene_profesores;
    relationship Set<EstudianteGrad> tutoriza
                 inverse EstudianteGrad::tutor;
    relationship Set<EstudianteGrad> en_comite
                 inverse EstudianteGrad::comite;
/* Definición de operaciones */
    void aumentar_salario(in float aumento);
    void promocionar(in string nueva_categoria);
}
```

```
class Estudiante extends Persona
    (extent estudiantes)
{
/* Definición de atributos */
    attribute string titulacion;
/* Definición de relaciones */
    relationship set<Calificacion> ediciones_cursadas
                inverse Calificacion::estudiante;
    relationship set<EdicionActual> matriculado
                inverse EdicionActual::estudiantes_matriculados;
/* Definición de operaciones */
    float nota_media();
    void matricularse(in short num_edic) raises(edicion_no_valida, edicion_llena);
    void calificar(in short num_edic; in float nota)
                raises(edicion_no_valida, nota_no_valida);
};


class Calificacion
    (extent calificaciones)
{
/* Definición de atributos */
    attribute float nota;
/* Definición de relaciones */
    relationship Edicion edicion inverse Edicion::estudiantes;
    relationship Estudiante estudiante
                inverse Estudiante::ediciones_cursadas;
};


class EstudianteGrad extends Estudiante
    (extent estudiantes_graduados)
{
/* Definición de atributos */
    attribute set<Titulo> titulos;
/* Definición de relaciones */
    relationship Profesor tutor inverse Profesor::tutoriza;
    relationship set<Profesor> comite inverse Profesor::en_comite;
/* Definición de operaciones */
    void asignar_tutor(in string apellido1; in string apellido2)
                raises(profesor_no_valido);
    void asignar_miembro_comite(in string apellido1; in string apellido2)
                raises(profesor_no_valido);
};
```

```
class Titulo
{
/* Definición de atributos */
    attribute string escuela;
    attribute string titulo;
    attribute string año;
};

class Departamento
    (extent departamentos key nombre)
{
/* Definición de atributos */
    attribute string nombre;
    attribute string telefono;
    attribute string despacho;
    attribute string escuela;
    attribute Profesor director;
/* Definición de relaciones */
    relationship set<Profesor> tiene_profesores
                inverse Profesor::trabaja_en;
    relationship set<Curso> oferta
                inverse Curso::ofertado_por;
};

class Curso
    (extent cursos key num_curso)
{
/* Definición de atributos */
    attribute string nombre;
    attribute string num_curso;
    attribute string descripcion;
/* Definición de relaciones */
    relationship set<Edicion> tiene_ediciones
                inverse Edicion::de_curso;
    relationship Departamento ofertado_por inverse Departamento::oferta;
};

class Edicion
    (extent ediciones)
{
/* Definición de atributos */
    attribute short num_edic
    attribute string año;
```

```
    attribute enum Semestre{Primero,Segundo} semestre;
/* Definición de relaciones */
    relationship set<Calificacion> estudiantes
                inverse Calificacion::edicion;
    relationship Curso de_curso inverse Curso::tiene_ediciones;
};


class EdicionActual extends Edicion
    (extent ediciones_actuales)
{
/* Definición de relaciones */
    relationship set<Estudiante> estudiantes_matriculados
                inverse Estudiante::matriculado;
/* Definición de operaciones */
    void matricular_estudiante(in string dni)
                raises(estudiante_no_valido,edicion_llena);
};
```

4.3. Lenguaje de consulta de objetos 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 súperconjunto 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 éstos poseen.

La sintaxis básica de OQL es una estructura `SELECT...FROM...WHERE...`, como en SQL. Por ejemplo, la siguiente expresión obtiene los nombres de los departamentos de la escuela de 'Ingeniería':

```
SELECT d.nombre
FROM d in departamentos
WHERE d.escuela = 'Ingeniería';
```

En las consultas se necesita un punto de entrada, que suele ser el nombre de un objeto persistente. Para muchas consultas, el punto de entrada es la extensión de una clase. En el ejemplo anterior, el punto de entrada es la extensión `departamentos`, que es un objeto colección de tipo `set<Departamento>`. Cuando se utiliza una extensión como punto de entrada es necesario utilizar una variable iteradora que vaya tomando valores en los objetos de la

colección. Para cada objeto de la colección (sólo la forman objetos persistentes) que cumple la condición (que es de la escuela de 'Ingeniería'), se muestra el valor del atributo `nombre`. El resultado es de tipo `bag<string>`. Cuando se utiliza `SELECT DISTINCT...` el resultado es de tipo `set` ya que se eliminan los duplicados.

Las variables iterador se pueden especificar de tres formas distintas:

```
d in departamentos
departamentos d
departamentos as d
```

El resultado de una consulta puede ser de cualquier tipo soportado por el modelo. Una consulta no debe seguir la estructura `SELECT` ya que el nombre de cualquier objeto persistente es una consulta de por sí. Por ejemplo, la consulta:

```
departamentos;
```

devuelve una referencia a la colección de todos los objetos `Departamento` persistentes. Del mismo modo, si se da nombre a un objeto concreto, por ejemplo a un departamento se le llama `departamentoinf` (el departamento de informática), la siguiente consulta:

```
departamentoinf;
```

devuelve una referencia a ese objeto individual de tipo `Departamento`. Una vez se establece un punto de entrada, se pueden utilizar expresiones de camninos 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. Por ejemplo:

```
departamentoinf.director;
departamentoinf.director.categoria;
departamentoinf.tiene_profesores;
```

La primera expresión devuelve una referencia a un objeto `Profesor`, aquel que dirige el departamento de informática. La segunda expresión obtiene la categoría del profesor que dirige este departamento (el resultado es de tipo `string`). La tercera expresión devuelve un objeto de tipo `set<Profesor>`. Esta colección contiene referencias a todos los objetos `Profesor` que se relacionan con el objeto cuyo nombre es `departamentoinf`. Si se quiere obtener la categoría de todos estos profesores, *no* podemos escribir la expresión:

```
departamentoinf.tiene_profesores.categoria;
```

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 p.categoria
FROM p in departamentoinf.tiene_profesores;

SELECT DISTINCT p.categoria
FROM p in departamentoinf.tiene_profesores;
```

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

```
departamentoinf.director.tutoriza;
```

devuelve un objeto de tipo `set<EstudianteGrad>`: una colección que contiene los estudiantes graduados que son tutorizados por el director del departamento de informática. Si lo que se necesita son los nombres y apellidos de estos estudiantes y los títulos que tiene cada uno, se puede escribir la siguiente consulta:

```
SELECT struct(nombre:struct(ape1: e.nombre.apellido1,
                    ape2: e.nombre.apellido2,
                    nom: e.nombre.nombre_pila),
                titulos:(SELECT struct(tit: t.titulo,
                            año: t.año,
                            esc: t.escuela)
                        FROM t in e.titulos)
FROM e in departamentoinf.director.tutoriza;
```

OQL es ortogonal respecto a la especificación de expresiones de caminos: atributos, relaciones y operaciones (métodos) pueden ser utilizados en estas expresiones, siempre que el sistema de tipos de OQL no se vea comprometido. Por ejemplo, para obtener los nombres y apellidos de los estudiantes que tutoriza la profesora 'Gloria Martínez', ordenados por su nota media, se podría utilizar la siguiente consulta (el resultado, por estar ordenado, será de tipo `list`):

```
SELECT struct(ape1: e.nombre.apellido1,
              ape2: e.nombre.apellido2,
              nom: e.nombre.nombre_pila,
              media: e.nota_media)
FROM e in estudiantes_graduados
WHERE e.tutor.nombre_pila='Gloria'
AND e.tutor.apellido1='Martínez'
ORDER BY media DESC, ape1 ASC, ape2 ASC;
```

OQL tiene además otras características que no se van a presentar aquí:

- Especificación de vistas dando nombres a consultas.
- Obtención como resultado de un solo elemento (hasta ahora hemos visto que se devuelven 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. Sistemas objeto–relacionales

El modo en que los objetos han entrado en el mundo de las bases de datos relacionales es en forma de dominios, actuando como el tipo de datos de una columna. Hay dos implicaciones muy importantes por el hecho de utilizar una clase como un dominio:

- Es posible almacenar múltiples valores en una columna de una misma fila ya que un objeto suele contener múltiples valores. Sin embargo, si se utiliza una clase como dominio de una columna, en cada fila esa columna sólo puede contener un objeto de la clase (se sigue manteniendo la restricción del modelo relacional de contener valores atómicos en la intersección de cada fila con cada columna).
- Es posible almacenar procedimientos en las relaciones porque un objeto está enlazado con el código de los procesos que sabe realizar (los métodos de su clase).

Otro modo de incoporar objetos en las bases de datos relacionales es construyendo tablas de objetos, donde cada fila es un objeto.

Ya que un sistema objeto–relacional es un sistema relacional que permite almacenar objetos en sus tablas, la base de datos sigue sujeta a las restricciones que se aplican a todas las bases de datos relacionales y conserva la capacidad de utilizar operaciones de concatenación (*join*) para implementar las relaciones "al vuelo".

5.1. Objetos en Oracle

Los tipos de objetos en Oracle son tipos de datos definidos por el usuario. La tecnología de objetos que proporciona es una capa de abstracción construida sobre su tecnología relacional, por lo que los datos se siguen almacenando en columnas y tablas. En los siguientes apartados se resume la orientación a objetos que soporta la versión 9i de Oracle.

5.1.1. Tipos de objetos y referencias

Para crear tipos de objetos se utiliza la sentencia `CREATE TYPE`. A continuación se muestran algunos ejemplos:

```
CREATE TYPE persona AS OBJECT
(
    nombre VARCHAR2(30),
    telefono VARCHAR2(20)
);

CREATE TYPE lineaped AS OBJECT
(
    nom_articulo VARCHAR2(30),
    cantidad NUMBER,
    precio_unidad NUMBER(12,2)
);

CREATE TYPE lineaped_tabla AS TABLE OF lineaped;

CREATE TYPE pedido AS OBJECT
(
    id NUMBER,
    contacto persona,
    lineasped lineaped_tabla,

    MEMBER FUNCTION obtener_valor RETURN NUMBER
);
```

`lineasped` es lo que se denomina una *tabla anidada* (*nested table*) que es un objeto de tipo colección. Una vez creados los objetos, éstos se pueden utilizar como un tipo de datos al igual que `NUMBER` o `VARCHAR2`. Por ejemplo, podemos definir una tabla relacional para guardar información de personas de contacto:

```
CREATE TABLE contactos
(
    contacto persona,
    fecha DATE
);
```

Esta es una tabla relacional que tiene una columna cuyo tipo es un objeto. Cuando los objetos se utilizan de este modo se les denomina *objetos columna*.

Cuando se declara una columna como un tipo de objeto o como una tabla anidada, se puede incluir una cláusula `DEFAULT` para asignar valores por defecto. Veamos un ejemplo:

```
CREATE TYPE persona AS OBJECT
(
    id NUMBER,
    nombre VARCHAR2(30),
    direccion VARCHAR2(30)
);

CREATE TYPE gente AS TABLE OF persona;

CREATE TABLE departamento
(
    num_dept VARCHAR2(5) PRIMARY KEY,
    nombre_dept VARCHAR2(20),
    director persona DEFAULT persona(1,'Pepe Pérez',NULL),
    empleados gente DEFAULT gente( persona(2,'Ana López','C/del Pez, 5'),
                                   persona(3,'Eva García',NULL) )
)
NESTED TABLE empleados STORE AS empleados_tab;
```

Las columnas que son tablas anidadas y los atributos que son tablas de objetos requieren una tabla a parte donde almacenar las filas de dichas tablas. Esta tabla de almacenamiento se especifica mediante la cláusula `NESTED TABLE...STORE AS...`. Para recorrer las filas de una tabla anidada se utilizan cursores anidados.

Sobre las tablas de objetos se pueden definir restricciones. En el siguiente ejemplo se muestra cómo definir una clave primaria sobre una tabla de objetos:

```
CREATE TYPE ubicacion AS OBJECT
(
    num_edificio NUMBER,
    ciudad VARCHAR2(30)
);

CREATE TYPE persona AS OBJECT
(
    id NUMBER,
    nombre VARCHAR2(30),
    direccion VARCHAR2(30),
    oficina ubicacion
);
CREATE TABLE empleados OF persona
(
    id PRIMARY KEY
);
```

El siguiente ejemplo define restricciones sobre atributos escalares de un objeto columna:

```
CREATE TABLE departamento
(
    num_dept VARCHAR2(5) PRIMARY KEY,
    nombre_dept VARCHAR2(20),
    director persona,
    despacho ubicacion,
    CONSTRAINT despacho_cons1
         UNIQUE (despacho.num_edificio,despacho.ciudad),
    CONSTRAINT despacho_cons2
         CHECK (despacho.ciudad IS NOT NULL)
);
```

Sobre las tablas de objetos también se pueden definir disparadores. Sobre las tablas de almacenamiento especificadas mediante `NESTED TABLE` no se pueden definir disparadores.

```
CREATE TABLE traslado
(
    id NUMBER,
    despacho_antiguo ubicacion,
    despacho_nuevo ubicacion
);
```

```
CREATE TRIGGER disparador
    AFTER UPDATE OF despacho ON empleados
    FOR EACH ROW
    WHEN new.despacho.ciudad='Castellon'
    BEGIN
        IF (:new.despacho.num_edificio=600) THEN
           INSERT INTO traslado (id, despacho_antiguo, despacho_nuevo)
              VALUES (:old.id, :old.despacho, :new.despacho);
        END IF;
    END;
```

Las relaciones se establecen mediante columnas o atributos `REF`. Estas relaciones pueden estar restringidas mediante la cláusula `SCOPE` o mediante una restricción de integridad referencial (`REFERENTIAL`). Cuando se restringe mediante `SCOPE`, todos lo valores almacenados en la columna `REF` apuntan a objetos de la tabla especificada en la cláusula. Sin embargo, puede ocurrir que haya valores que apunten a objetos que no existen. La restricción mediante `REFERENTIAL` es similar a la especificación de claves ajenas. La regla de integridad referencial se aplica a estas columnas, por lo que las referencias a objetos que se almacenen en estas columnas deben ser siempre de objetos que existen en la tabla referenciada.

Para evitar ambigüedades con los nombres de atributos y de métodos al utilizar la notación punto, Oracle obliga a utilizar alias para las tablas en la mayoría de las ocasiones (aunque recomienda hacerlo siempre, para evitar problemas). Por ejemplo, dadas las tablas:

```
CREATE TYPE persona AS OBJECT (dni VARCHAR2(9));
CREATE TABLE ptab1 OF persona;
CREATE TABLE ptab2 (c1 persona);
```

las siguientes consultas muestran modos correctos e incorrectos de referenciar el atributo `dni`:

```
SELECT dni FROM ptab1; -- Correcto
SELECT c1.dni FROM ptab2; -- Ilegal: notación punto sin alias de tabla
SELECT ptab2.c1.dni FROM ptab2; -- Ilegal: notación punto sin alias
SELECT p.c1.dni FROM ptab2 p; -- Correcto
```

5.1.2. Métodos

Los métodos son funciones o procedimientos que se pueden declarar en la definición de un tipo de objeto para implementar el comportamiento que se desea para dicho tipo de

objeto. Las aplicaciones llaman a los métodos para invocar su comportamiento. Para ello se utiliza también la notación punto: `objeto.metodo(lista_param)`. Aunque un método no tenga parámetros, Oracle obliga a utilizar los paréntesis en las llamadas `objeto.metodo()`. Los métodos escritos en PL/SQL o en Java, se almacenan en la base de datos. Los métodos escritos en otros lenguajes se almacenan externamente.

Hay dos clases de métodos: miembros y estáticos. Hay otro tercer tipo, los métodos constructores, que el propio sistema define para cada tipo de objeto.

Los métodos miembro son los que se utilizan para ganar acceso a los datos de una instancia de un objeto. Se debe definir un método para cada operación que se desea que haga el tipo de objeto. Estos métodos tienen un parámetro denominado `SELF` que denota a la instancia del objeto sobre la que se está invocando el método. Los métodos miembro puden hacer referencia a los atributos y a los métodos de `SELF` sin necesidad de utilizar el cualificador.

```
CREATE TYPE racional AS OBJECT
(
    num INTEGER,
    den INTEGER,
    MEMBER PROCEDURE normaliza,
    ...
);

CREATE TYPE BODY racional AS
    MEMBER PROCEDURE normaliza IS
        g INTEGER;
    BEGIN
        g := gcd(SELF.num, SELF.den);
        g := gcd(num, den); -- equivale a la línea anterior
        num := num / g;
        den := den / g;
    END normaliza;
    ...
END;
```

`SELF` no necesita declararse, aunque se puede declarar. Si no se declara, en las funciones se pasa como `IN` y en los procedimientos se pasa como `IN OUT`.

Los valores de los tipos de datos escalares siguen un orden y, por lo tanto, se pueden comparar. Sin embargo, con los tipos de objetos, que pueden tener múltiples atributos de distintos tipos, no hay un criterio predefinido de comparación. Para poder comparar objetos se debe establecer este criterio mediante métodos de mapeo o métodos de orden.

Un método de mapeo (`MAP`) permite comparar objetos mapeando instancias de objetos con tipos escalares `DATE`, `NUMBER`, `VARCHAR2` o cualquier tipo ANSI SQL como `CHARACTER` o `REAL`. Un método de mapeo es una función sin parámetros que devuelve uno de los tipos anteriores. Si un tipo de objeto define uno de estos métodos, el método se llama automáticamente para evaluar comparaciones del tipo `obj1 > obj2` y para evaluar las comparaciones que implican `DISTINCT`, `GROUP BY` y `ORDER BY`.

```
CREATE TYPE rectangulo AS OBJECT (
    alto NUMBER,
    ancho NUMBER,
    MAP MEMBER FUNCTION area RETURN NUMBER,
...
);

CREATE TYPE BODY rectangulo AS
    MAP MEMBER FUNCTION area RETURN NUMBER IS
    BEGIN
        RETURN alto*ancho;
    END area;
    ...
END;
```

Los métodos de orden `ORDER` hacen comparaciones directas objeto–objeto. Son funciones con un parámetro declarado para otro objeto del mismo tipo. El método se debe escribir para que devuelva un número negativo, cero o un número positivo, lo que significa que el objeto `SELF` es menor que, igual o mayor que el otro objeto que se pasa como parámetro. Los métodos de orden se utilizan cuando el criterio de comparación es muy complejo como para implementarlo con un método de mapeo.

Un tipo de objeto puede declarar sólo un método de mapeo o sólo un método de orden, de manera que cuando se comparan dos objetos, se llama automáticamente al método que se haya definido, sea de uno u otro tipo.

Los métodos estáticos son los que pueden ser invocados por el tipo de objeto y no por sus instancias. Estos métodos se utilizan para operaciones que son globales al tipo y que no necesitan referenciar datos de una instancia concreta. Los métodos estáticos no tienen el parámetro `SELF`. Para invocar estos métodos se utiliza la notación punto sobre el tipo del objeto: `tipo_objeto.método()`

Cada tipo de objeto tiene un método constructor implícito definido por el sistema. Este método crea un nuevo objeto (una instancia del tipo) y pone valores en sus atributos. El método constructor es una función y devuelve el nuevo objeto como su valor. El nombre del método constructor es precisamente el nombre del tipo de objeto. Sus parámetros tienen los nombres y los tipos de los atributos del tipo.

```
CREATE TABLE departamento (
    num_dept VARCHAR2(5) PRIMARY KEY,
    nombre_dept VARCHAR2(20),
    despacho ubicacion
);

INSERT INTO departamento
    VALUES ( '233', 'Ventas', ubicacion(200,'Borriol') );
```

5.1.3. Colecciones

Oracle soporta dos tipos de datos colección: las tablas anidadas y los *varray*. Un *varray* es una colección ordenada de elementos. La posición de cada elemento viene dada por un índice que permite acceder a los mismos. Cuando se define un *varray* se debe especificar el número máximo de elementos que puede contener (aunque este número se puede cambiar después). Los *varray* se almacenan como objetos opacos (`RAW` o `BLOB`). Una tabla anidada puede tener cualquier número de elementos: no se especifica ningún máximo cuando se define. Además, no se mantiene el orden de los elementos. En las tablas anidades se consultan y actualizan datos del mismo modo que se hace con las tablas relacionales. Los elementos de una tabla anidada se almacenan en una tabla a parte en la que hay una columna llamada `NESTED_TABLE_ID` que referencia a la tabla padre o al objeto al que pertenece.

```
CREATE TYPE precios AS VARRAY(10) OF NUMBER(12,2);

CREATE TYPE lineaped_tabla AS TABLE OF lineaped;
```

Cuando se utiliza una tabla anidada como una columna de una tabla o como un atributo de un objeto, es preciso especificar cuál será su tabla de almacenamiento mediante `NESTED TABLE...STORE AS....`

Se pueden crear tipos colección multinivel, que son tipos colección cuyos elementos son colecciones.

```
CREATE TYPE satelite AS OBJECT (
    nombre VARCHAR2(20),
    diametro NUMBER );

CREATE TYPE tab_satelite AS TABLE OF satelite;
```

```
CREATE TYPE planeta AS OBJECT (
    nombre VARCHAR2(20),
    masa NUMBER,
    satelites tab_satelite );

CREATE TYPE tab_planeta AS TABLE OF planeta;
```

En este caso, la especificación de las tablas de almacenamiento se debe hacer para todas y cada una de las tablas anidadas.

```
CREATE TABLE estrellas (
    nombre VARCHAR2(20),
    edad NUMBER,
    planetas tab_planeta )
NESTED TABLE planetas STORE AS tab_alm_planetas
    (NESTED TABLE satelites STORE AS tab_alm_satelites);
```

Para crear una instancia de cualquier tipo de colección también se utiliza el método constructor, tal y como se hace con los objetos.

```
INSERT INTO estrellas
VALUES('Sol',23,
    tab_planeta(
        planeta(
            'Neptuno',
            10,
            tab_satelite(
                satelite('Proteus',67),
                satelite('Triton',82)
            )
        ),
        planeta(
            'Jupiter',
            189,
            tab_satelite(
                satelite('Calisto',97),
                satelite('Ganimedes',22)
            )
        )
    )
);
```

Las colecciones se pueden consultar con los resultados anidados:

```
SELECT e.nombre,e.proyectos
FROM empleados e;

NOMBRE   PROYECTOS
------   ---------
'Pedro'  tab_proyecto(67,82)
'Juan'   tab_proyecto(22,67,97)
```

o con los resultados sin anidar:

```
SELECT e.nombre, p.*
FROM empleados e, TABLE(e.proyectos) p;

NOMBRE   PROYECTOS
------   ---------
'Pedro'  67
'Pedro'  82
'Juan'   22
'Juan'   67
'Juan'   97
```

La notación `TABLE` sustituye a la notación `THE subconsulta` de versiones anteriores. La expresión que aparece en `TABLE` puede ser tanto el nombre de una colección como una subconsulta de una colección. Las dos consultas que se muestran a continuación obtienen el mismo resultado.

```
SELECT p.*
FROM empleados e, TABLE(e.proyectos) p
WHERE e.numemp = '18';

SELECT *
FROM TABLE(SELECT e.proyectos
           FROM empleados e
           WHERE e.numemp = '18');
```

También es posible hacer consultas con resultados no anidados sobre colecciones multinivel.

```
SELECT s.nombre
FROM estrellas e, TABLE(e.planetas) p, TABLE(p.satelites) s;
```

5.1.4. Herencia de tipos

La versión 9i es la primera versión de Oracle que soporta herencia de tipos. Cuando se crea un subtipo a partir de un tipo, el subtipo hereda todos los atributos y los métodos del tipo padre. Cualquier cambio en los atributos o métodos del tipo padre se reflejan automáticamente en el subtipo. Un subtipo se convierte en una versión especializada del tipo padre cuando al subtipo se le añaden atributos o métodos, o cuando se redefinen métodos que ha heredado, de modo que el subtipo ejecuta el método "a su manera". A esto es a lo que se denomina *polimorfismo* ya que dependiendo del tipo del objeto sobre el que se invoca el método, se ejecuta uno u otro código. Cada tipo puede heredar de un solo tipo, no de varios a la vez (no soporta herencia múltiple), pero se pueden construir jerarquías de tipos y subtipos.

Cuando se define un tipo de objeto, se determina si de él se pueden derivar subtipos mediante la cláusula `NOT FINAL`. Si no se incluye esta cláusula, se considera que es `FINAL` (no puede tener subtipos). Del mismo modo, los métodos pueden ser `FINAL` o `NOT FINAL`. Si un método es final, los subtipos no pueden redefinirlo (*override*) con una nueva implementación. Por defecto, los métodos son no finales (es decir, redefinibles).

```
CREATE TYPE t AS OBJECT ( ...,
    MEMBER PROCEDURE imprime(),
    FINAL MEMBER FUNCTION fun(x NUMBER) ...
) NOT FINAL;
```

Para crear un subtipo se utiliza la cláusula `UNDER`.

```
CREATE TYPE estudiante UNDER persona ( ...,
    titulacion VARCHAR2(30),
    fecha_ingreso DATE
) NOT FINAL;
```

El nuevo tipo, además de heredar los atributos y métodos del tipo padre, define dos nuevos atributos. A partir del subtipo se pueden derivar otros subtipos y del tipo padre tam-

bién se pueden derivar más subtipos. Para redefinir un método, se debe utilizar la cláusula `OVERRIDING`.

Los tipos y los métodos se pueden declarar como no instanciables. Si un tipo es no instanciable, no tiene método constructor, por lo que no se pueden crear instancias a partir de él. Un método no instanciable se utiliza cuando no se le va a dar una implementación en el tipo en el que se declara sino que cada subtipo va a proporcionar una implementación distinta.

```
CREATE TYPE t AS OBJECT (
    x NUMBER,
    NOT INSTANTIABLE MEMBER FUNCTION fun() RETURN NUMBER
) NOT INSTANTIABLE NOT FINAL;
```

Un tipo puede definir varios métodos con el mismo nombre pero con distinta signatura. La signatura es la combinación del nombre de un método, el número de parámetros, los tipos de éstos y el orden formal. A esto se le denomina *sobrecarga* de métodos (*overloading*).

En una jerarquía de tipos, los subtipos son variantes de la raíz. Por ejemplo, en tipo `estudiante` y el tipo `empleado` son clases de `persona`. Normalmente, cuando se trabaja con jerarquías, a veces se quiere trabajar a un nivel más general (por ejemplo, seleccionar o actualizar todas las personas) y a veces se quiere trabajar sólo son los estudiantes o sólo con los que no son estudiantes. La habilidad de poder seleccionar todas las personas juntas, pertenezcan o no a algún subtipo, es lo que se denomina *sustituibilidad*. Un súpertipo es sustituible si uno de sus subtipos puede sustituirlo en una variable, columna, etc. declarada del tipo del súpertipo. En general, los tipos son sustituibles.

- Un atributo definido como `REF miTipo` puede contener una `REF` a una instancia de `miTipo` o a una instancia de cualquier subtipo de `miTipo`.
- Un atributo definido de tipo `miTipo` puede contener una instancia de `miTipo` o una instancia de cualquier subtipo de `miTipo`.
- Una colección de elementos de tipo `miTipo` puede contener instancias de `miTipo` o instancias de cualquier subtipo de `miTipo`.

Dado el tipo `libro`:

```
CREATE TYPE libro AS OBJECT (
    titulo VARCHAR2(30),
    autor persona );
```

se puede crear una instancia de `libro` especificando un título y un autor de tipo `persona` o de cualquiera de sus subtipos, `estudiante` o `empleado`:

```
libro('BD objeto-relacionales',
    estudiante(123,'María Gil','C/Mayor,3','II','10-OCT-99')
```

A continuación se muestra un ejemplo de la sustituibilidad en las tablas de objetos.

```
CREATE TYPE persona AS OBJECT
    (id NUMBER,
    nombre VARCHAR2(30),
    direccion VARCHAR2(30)) NOT FINAL;

CREATE TYPE estudiante UNDER persona
    (titulacion VARCHAR2(10),
    especialidad VARCHAR2(30)) NOT FINAL;

CREATE TYPE estudiante_doctorado UNDER estudiante
    (programa VARCHAR2(10));

CREATE TABLE personas_tab OF persona;

INSERT INTO personas_tab
    VALUES (persona(1234,'Ana','C/Mayor,23'));

INSERT INTO personas_tab
    VALUES (estudiante(2345,'Jose','C/Paz,3','ITDI','Mecánica'));

INSERT INTO personas_tab
    VALUES (estudiante_doctorado(3456,'Luisa','C/Mar,45','IInf',NULL,'CAA'));
```

5.1.5. Funciones y predicados útiles con objetos

`VALUE` : esta función toma como parámetro un alias de tabla (de una tabla de objetos) y devuelve instancias de objetos correspondientes a las filas de la tabla.

```
SELECT VALUE(p)
FROM personas_tab p
WHERE p.direccion LIKE 'C/Mayor%';
```

La consulta devuelve todas las personas que viven en la calle Mayor, sean o no de algún subtipo.

`REF` : es una función que toma como parámetro un alias de tabla y devuelve una referencia a una instancia de un objeto de dicha tabla.

`DEREF` : es una función que devuelve la instancia del objeto correspondiente a una referencia que se le pasa como parámetro.

`IS OF` : permite formar predicados para comprobar el nivel de especialización de instancias de objetos.

```
SELECT VALUE(p)
FROM personas_tab p
WHERE VALUE(p) IS OF (estudiante);
```

De este modo se obtienen las personas que son del subtipo `estudiante` o que son de alguno de sus subtipos. Para obtener solamente aquellas personas cuyo tipo más específico es `estudiante` se utiliza la cláusula `ONLY`:

```
SELECT VALUE(p)
FROM personas_tab p
WHERE VALUE(p) IS OF (ONLY estudiante);
```

`TREAT` : es una función que trata a una instancia de un súpertipo como una instancia de uno de sus subtipos:

```
SELECT TREAT(VALUE(p) AS estudiante)
FROM personas_tab p
WHERE VALUE(p) IS OF (ONLY estudiante);
```

Bibliografía

Los capítulos 11 y 12 del texto de Elmasri y Navathe tratan ampliamente la orientación a objetos en la tecnología de bases de datos, como también se hace en los capítulos 21 y 22 del texto de Connolly, Begg y Strachan o en el capítulo 11 de Atzeni et al.. Este es un tema que aparece en gran parte de los textos básicos sobre bases de datos, aunque sólo los más recientes tratan los sistemas objeto–relacionales. El texto de Elmasri y Navathe los analiza en el capítulo 13, mientras que el texto de Connolly, Begg y Strachan lo hace en el capítulo 23 (Atzeni et al. los trata en el mismo capítulo 11).