Programación Orientada a Objetos

OODB Y ODMG

Mayo, 2003

Bases de Datos Orientadas a Objeto y el estándar ODMG

Clara Martín Sastre

Enrique Medarde Caballero

Departamento de Informática y Automática

Universidad de Salamanca

Información de los autores:
Clara Martín Sastre
Facultad de Ciencias – Universidad de Salamanca
claroto11@hotmail.com

Enrique Medarde Caballero
Facultad de Ciencias – Universidad de Salamanca
tengounacasaenelcampo@yahoo.com

Resumen

En este informe técnico se aborda de una forma introductoria el tema de las bases de datos orientadas a objetos, haciendo un recorrido temporal sobre sus predecesoras, y analizando las características más significativas de las mismas. Así mismo, se realiza un breve estudio de uno de los estándares que, por el momento, parece ser más importante de acuerdo con sus perspectivas de futuro. Hablamos del estándar ODMG. Se afrontará dicho estándar desde el punto de vista del lenguaje de programación C++. El motivo que nos ha decidido a elegir este tema es, que teniendo en cuenta que el entorno de las bases de datos no nos resulta desconocido y nos interesa, adquirir cierta información acerca de lo que puede llegar a establecerse como el futuro de las mismas, y más, enfocándolo desde el punto de vista de C++, nos parecía atrayente. De ahí que este informe se dedique a este tema, aunque su tratamiento no sea demasiado profundo.

Abstract

In this technical report we make an introductory approach to object-oriented databases, a historical overview about their predecessors and a most-significant characteristics analysis. We also make a brief study of one of the most important standards, which seems to be one with great future perspectives. We are talking about the ODMG standard. We will aboard this standard from the C++ programming language side. We have chosen object-oriented databases because, as we have some notions of databases, getting some information about their future, and giving it a C++ focus seemed very important to us.

Tabla de Contenidos

Tabla de Figuras

1. Introducción

En este apartado trataremos de abordar la idea de lo que son las bases de datos orientadas a objetos (*Object-Oriented DataBases*, OODB). Para ello, realizaremos un recorrido a lo largo de la evolución de los sistemas de almacenamiento y manejo de información, puesto que quizá esto nos ayude a comprender mejor el papel de las OODB en la actualidad.

En un primer momento, los sistemas de manejo de datos se reducían exclusivamente a realizar operaciones comunes sobre ficheros de todo tipo, independientemente de la información que éstos almacenasen. El lenguaje de programación COBOL introdujo en este ámbito un conjunto de estructuras que distinguían una parte referida a los datos y ficheros que se iban a utilizar y otra, referida a las acciones que se podían realizar sobre ellos. En el momento que este sistema se quedó pequeño ante las necesidades de modelado de estructuras más complejas que requerían tratar datos de diferentes ficheros de forma conjunta, surgió el concepto de *base de datos.*

Aparecieron así un lenguaje de descripción de datos (*DDL*) y un lenguaje de manipulación de datos (*DML*), que sentaron las bases de los sistemas de bases de datos en red. El modelo de datos subyacente presentaba la base de datos como una red de nodos constituidos por tipos de registros y cuyos arcos representaban relaciones *uno-a-muchos* entre ellos.

Otro modelo, al igual que el de red, de tipo navegante, era el modelo jerárquico. En dicho modelo sólo se permitían dos roles entre los tipos de registros: el de "padre" de una relación *uno-a-muchos*, y el de "hijo". De esta forma, la información queda estructurada esta vez en forma de árbol y no de grafo como en el modelo en red. El hecho de que ambos modelos fueran de tipo navegante, significaba que un usuario interesado en acceder a un registro de la base de datos debía partir de un registro antecesor e ir recorriendo, a través de las relaciones del mismo, diversos registros hasta llegar al deseado. Es más, las relaciones que un determinado registro mantenía con otros, eran almacenadas de forma explícita en dicho registro, lo que en definitiva significaba que no existía independencia física y, por tanto, la visión que el usuario tenía de la base de datos reflejaba perfectamente la forma en que los datos estaban almacenados. Esta característica mermaba notablemente cualidades como la portabilidad, la extensibilidad y el mantenimiento de la propia base de datos como de las aplicaciones destinadas a la misma.

Ante los problemas que de por sí presentaban las bases de datos navegacionales y la dificultad de manejo de grandes bancos de datos, T. Codd presentó en 1970 el modelo de bases de datos relacional. Este modelo, *per sé*, presenta una serie de ventajas respecto a los anteriores, como por ejemplo que los lenguajes de consulta son más declarativos, de forma que se puede especificar a la base de datos lo que se requiere en un lenguaje de alto nivel, en lugar de hacerlo a través de los requerimientos de acceso de la misma. Por otra parte, el modelo se fundamenta en conceptos matemáticos, como el Álgebra Relacional y el Cálculo de Predicados. Además, resulta mucho más elegante, flexible y cómodo que los anteriores, lo que le proporcionó gran popularidad durante los años 80 y 90.

El origen de las bases de datos orientadas a objetos se encuentra en el hecho de que existen problemas para representar cierta información, puesto que los modelos clásicos permiten representar gran cantidad de datos, pero las operaciones que permiten realizar sobre ellos son bastante simples. Por ejemplo, consultas más complejas donde se especifican relaciones de tipo recursivo (ser padre de, ser jefe de), no se pueden desarrollar de forma natural en estos sistemas: requieren estructuras de representación más complejas que además hacen uso de elementos activos proporcionados por los lenguajes de programación. Otro de los problemas que dan origen a este tipo de bases de datos radica en el hecho de que se trate de representar información que no se adapte a los modelos de registro, sino dicha información se da mediante reglas lógicas que responden al conocimiento de un "experto".

Por tanto, las bases de datos orientadas a objetos surgen para tratar de paliar las deficiencias de los modelos anteriores y con el objetivo de proporcionar eficiencia y sencillez. Las clases utilizadas en un determinado LPOO son las clases que serán utilizadas en una OODB; esto se debe a la consistencia de los modelos, de tal forma, que no es necesaria una transformación del modelo de objetos para que pueda ser utilizado por un gestor de OODB. De forma contraria, el modelo relacional, requería una abstracción suficiente como para ser capaces de confinar objetos del mundo real en tablas.

Por tanto, antes de comenzar con algunos de los conceptos implícitos en el ámbito de las bases de datos orientadas a objeto es necesario que se pongan de manifiesto algunas de las razones por las cuales se justifica su necesidad. Generalmente, el uso de OODB es más ventajoso si se presenta en alguno de los siguientes escenarios:

- Un gran número de tipos de datos diferentes
- Un gran número de relaciones entre los objetos
- Objetos con comportamientos complejos

Las áreas de aplicación donde existe este tipo de complejidad acerca de tipos de datos, relaciones entre objetos y comportamiento de los objetos incluyen ingeniería, manufacturación, simulaciones, automatización de oficina y un elevado número de sistemas de información. Sin embrago, éstas no son las únicas áreas donde las OODB pueden utilizarse, puesto que al ofrecer la misma funcionalidad que su precursor relacional, esos campos tienen la posibilidad de aprovechar completamente la potencia que las OODB ofrecen para modelar situaciones del mundo real.

2. Evolución de las Bases de Datos Orientadas a Objeto

Las bases de datos orientadas a objetos (OODB) datan de los encuentran su origen en la década de los años 70, aunque no es hasta comienzo de los años 80 cuando empiezan a adquirir un gran significado. Es a finales de esta década el momento en el que aparecen los primeros productos comerciales.

Los OODBMS (*Object Oriented Data Base Management System*) se han establecido fuertemente en áreas como el *e-commerce*, la ingeniería de gestión de datos, y en bases de datos de propósito especial, como en seguridad y en medicina.

La fuerza del modelo de objetos radica en aplicaciones donde existe una necesidad imperante y subyacente de trabajar con relaciones complejas entre los objetos de datos. De esta manera, las OODB ya no suponen una “amenaza” para el modelo relacional, ya que el mercado se ha dividido, de tal manera que el modelo relacional se dedica a datos de poco volumen y no demasiada complejidad, mientras que el modelo orientado a objetos acoge grandes volúmenes y relaciones complejas entre los datos.

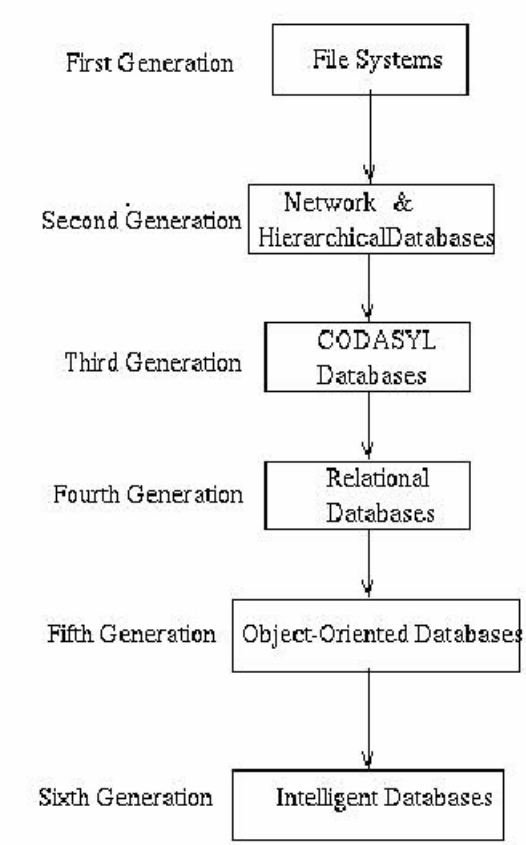

Figura 1: Evolución de las Bases de Datos

Se puede observar que el camino que han tomado las bases de datos orientadas a objetos es muy similar al seguido por las bases de datos relacionales: desde una funcionalidad mínima, hasta el manejo de grandes volúmenes de datos y relaciones de bastante complejidad entre ellos, pasando por un estadio intermedio que cubre casi todas las características de las bases de datos comunes, permitiendo el manejo de volúmenes y relaciones razonablemente complejos.

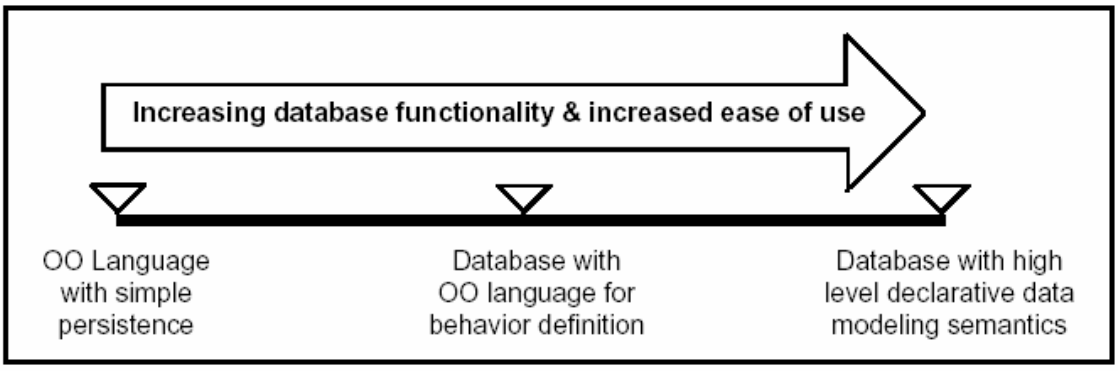

Figura 2: Relación complejidad – funcionalidad en las OODB

Los productos de la fase media-avanzada, son ya capaces de resolver el desarrollo de aplicaciones que gestionan datos de cierta complejidad. Los productos de semántica declarativa poseen la habilidad de reducir los esfuerzos de desarrollo, así como reforzar la uniformidad en la aplicación con esta semántica.

Hoy día, los OODBMS se encuentran en la fase media-avanzada. Existen productos que exhiben semántica declarativa, como restricciones, reglas de integridad referencial y capacidades de seguridad. El siguiente paso es más complejo; a medida que se avanza en la evolución, la base de datos ha de hacer más por el usuario, requiriendo menos esfuerzo para el desarrollo de aplicaciones. Por ejemplo, el desarrollo de interfaces de bajo nivel para optimizar el acceso a la base de datos.

Una forma general de establecer la madurez de la base de datos es mediante el grado en que funciones como la optimización del acceso, reglas de integridad, esquemas y potabilidad de la base de datos, archivos, copias de seguridad y operaciones de recuperación, pueden ser configuradas por el usuario utilizando comandos de declaración de alto nivel hacia el OODBMS. Otro signo de madurez es el establecimiento de grupos industriales para el establecimiento de estándares en diferentes aspectos de la tecnología. Hoy se aprecia un significativo interés en el desarrollo de estándares para las bases de datos orientadas a objetos. De esta forma, destacan OMG, ODMG, X3H7.

En la actualidad, los vendedores de OODBMS añaden más características a sus productos para proveer la funcionalidad que se podría esperar de un gestor de bases de datos maduro.

3. Características de las OODB

La tecnología utilizada en OODB es la consumación de la programación orientada a objetos y la tecnología de bases de datos. Muchos de los propósitos de las OODB son los mismos que los de las bases de datos tradicionales –gestión del almacenamiento en memoria secundaria (indexación, *clustering*, *buffering*, optimización), concurrencia y recuperación-, pero con la ventaja adicional de poder representar modelos de datos más complejos en un marco mucho más eficiente. Con este objetivo, lo que se ha hecho es aplicar el modelado de objetos a las bases de datos convencionales. El rasgo más característico de esta tecnología quizá sea precisamente su capacidad para combinar estos dos aspectos anteriormente citados, con el fin principal de proporcionar un sistema integrado de desarrollo de aplicaciones.

El hecho de incluir la definición de operaciones, característica propia de la orientación a objetos, con la definición de datos, lleva consigo una serie de ventajas. Una de ellas es que las operaciones definidas se aplican de forma ubicua y de forma independiente a la aplicación de bases de datos particular que se esté ejecutando en ese momento.

Otra de las ventajas a las que nos referimos radica en que los tipos de datos pueden ser extendidos para soportar datos complejos, como multimedia, por ejemplo, definiendo nuevas clases de objetos que tienen operaciones para soportar dichos nuevos tipos de datos.

Otros puntos fuertes del modelado orientado a objetos ya son bien conocidos. Por ejemplo, la **herencia**, permite desarrollar soluciones a problemas complejos de forma incremental, definiendo nuevos objetos a partir de los ya definidos. Proporciona a las subclases la habilidad de participar en la transmisión / manipulación de datos y métodos de acceso dentro del ámbito de su jerarquía. Mediante la herencia, problemas complejos son representados de una forma más fiel e intuitiva que facilita la reutilización de código y parte de las especificaciones de aplicaciones.

El **polimorfismo** y la **ligadura dinámica**, permiten definir operaciones para un objeto y posteriormente compartir esa especificación con otros objetos. Estos objetos pueden luego

extender esa operación para proporcionar comportamientos que son únicos para dichos objetos. El polimorfismo consta de sobrecarga y ocultamiento, con el objetivo de generar "una interfaz, múltiples métodos". Concretamente, la sobrecarga posibilita que un simple método sea implementado de diferentes maneras en diferentes momentos de tiempo, normalmente diferenciados por los parámetros que cada implementación requiere. El ocultamiento permite la redefinición de implementaciones que tiene lugar con la variación de los tipos. Junto a estos dos principios de la orientación a objeto, aparece la ligadura dinámica que determina, en tiempo de ejecución, cual de esas operaciones se ejecuta realmente, dependiendo de la clase del objeto requerido para llevar a cabo la operación. Estos tres rasgos constituyen características poderosas de la orientación a objetos, que permiten componer objetos para proporcionar soluciones sin tener que escribir código específico para cada objeto. Permiten que el código sea representativo de un determinado dominio sin quedar reducido al ámbito limitado de un objeto determinado.

Otra de las características básicas de las bases de datos orientadas a objetos es la **persistencia**. Desde el punto de vista de las bases de datos, sobre todo, del de la programación, es evidente que el modelo orientado a objetos es bastante extraño. Lo que la persistencia proporciona principalmente es el hecho de que los datos sobrevivan a la ejecución del objeto creado con el fin de ser reutilizados posteriormente en otro proceso. Para aliviar los problemas que surgen de los sistemas relacionales como son la dificultad para mantener la integridad y la barrera de comunicación, la persistencia posibilita que el lenguaje de programación y la base de datos posean el mismo modelo y además, que el programa tenga posibilidad de manipular datos transitorios y datos persistentes de la misma forma. De esta manera, el problema del aislamiento entre la aplicación y la base de datos se elimina directamente, puesto que se posibilita una interacción inmediata con la base de datos a través de la persistencia. Según Bancilhon, las tres reglas básicas de dicha interacción son las siguientes:

- El modelo de base de datos y es lenguaje de programación del sistema son el mismo.
- Los datos se dividen en datos transitorios y datos persistentes. El modelo de persistencia permite a los programas manipular datos persistentes y datos transitorios. Los datos persistentes se actualizan mediante transacciones una vez que se han comprometido (*commit*), así como los datos transitorios tienen una existencia de duración igual a la del programa.
- El programa puede operar de la misma manera tanto con datos transitorios como con datos persistentes.

En definitiva, el modelo de persistencia unifica la aplicación y su correspondiente base de datos insistiendo en operaciones globales que actúan indiferentemente sobre todos los datos, un lenguaje de interfaz común y una división reconocible entre datos transitorios y datos persistentes.

La **identificación de los objetos** es otra de las características esenciales en una base de datos orientada a objetos. Los objetos, por sí mismos, poseen una identidad independiente de su estado actual. Por ejemplo, si tenemos un objeto coche y remodelamos dicho coche y cambiamos su apariencia –el motor, la transmisión, las ruedas – de forma que parezca totalmente diferente, todavía será reconocido como el objeto que teníamos originalmente. De esta forma, dentro de una OODB, siempre se puede realizar la pregunta ¿este es el mismo objeto que tenía previamente?, asumiendo que uno recuerda la identidad del objeto. La identidad del objeto siempre permite a los objetos estar relacionados y compartidos dentro de una red distribuida de ordenadores. La identidad de los objetos establece relaciones entre objetos, además de una forma de navegar sin la estructura de la base de datos. A esto sigue que dos objetos pueden ser idénticos o que dos objetos pueden ser iguales. Como resultado, esto da lugar a dos implicaciones: la compartición de objetos y la actualización de objetos.

La compartición de objetos se basa en la idea de que dos objetos pueden compartir un componente o elemento de datos. En cuanto a la actualización de datos, se refiere a todos aquellos objetos que comparten componentes o elementos de datos, puesto que éstos requerirán una actualización consecuente. Las ventajas de mantenimiento en este aspecto se deben a la identificación de objetos.

En cuanto a la **encapsulación**, propia del paradigma orientado a objetos, podemos decir que en el ámbito de las bases de datos orientadas a objetos, provee de independencia a los datos donde la implementación de las clases pude ser alterada sin la necesidad de alterar otros métodos.

Además, es primordial que la base de datos sea extensible. Debe proveer de un soporte para la definición de nuevos objetos y métodos que operen con los objetos ya existentes.

Una diferencia significativa entre las OODB y las bases de datos relacionales es que las primeras representan las relaciones de forma explícita, soportando ambos modelos de bases de datos el acceso navegacional y asociativo a la información. A medida que la complejidad de las relaciones entre la información se va incrementando, se hacen evidentes las ventajas de representar las relaciones de forma explícita. Otro beneficio del uso de relaciones explícitas está en la mejora en la realización del acceso a los datos, sobre todo en comparación con el acceso que se realiza en el ámbito relacional.

4. El Sistema Gestor de OODB

En este apartado nos dedicaremos a exponer cuáles son las características que debe poseer un Sistema Gestor de Bases de Datos Orientadas a Objetos (OODBMS). Para ello, nos basaremos en "*The Object-Oriented Database System Manifesto*", propuesto por Malcolm Atkinson. En dicho manifiesto se hace una distinción de las características que de debe poseer todo sistema orientado a objetos, dividiéndolas en tres grupos: aquellas que el sistema está obligado a cumplir para ser cualificado como un sistemas de bases de datos orientado a objetos, aquellas que son opcionales y que pueden ser añadidas con el fin de hacer el sistema más eficiente y, por último, las opciones abiertas, en las cuales el diseñador puede hacer el número de cambios que desee. Detallaremos los tres grupos a continuación:

4.1 Características obligatorias: las reglas de Oro

Todo sistema de bases de datos orientado a objetos debe satisfacer dos criterios: ser un DBMS y ser un sistema orientado a objetos. El primero de los dos se reduce a cinco características: persistencia, gestión de almacenamiento secundario, concurrencia, recuperación y facilidad en la realización de consultas. El segundo se traduce en las siguientes características: objetos complejos, identificación de objetos, encapsulación, tipos o clases, herencia, ocultamiento combinado con ligadura tardía, extensibilidad y completitud computacional.

4.1.1 Complejidad de objetos

El OODBMS deberá soportar objetos complejos. Los objetos complejos se construyen a partir de los objetos simples como enteros, caracteres, cadenas de bytes, booleans o coma flotantes, aplicando constructores sobre ellos. Los constructores mínimos que todo sistema debe tener son conjuntos, listas y tuplas. Los conjuntos son esenciales porque ellos aportan una manera natural de representar colecciones del mundo real. Las tuplas son críticas porque

proporcionan un método natural para presentar propiedades de una entidad. Las listas o los arrays tienen la ventaja de capturan el orden, como ocurre en numerosas ocasiones en el mundo real. Los constructores de objetos deben ser ortogonales: un constructor debe aplicarse a algún objeto. En el modelo relacional, los constructores no eran ortogonales, puesto que el constructor de conjuntos sólo se podía aplicar a tuplas y el constructor de tuplas sólo se aplicaba a valores atómicos. Se ha de tener en cuenta que el manejo de objetos complejos requiere el uso de unos operadores adecuados. Dichos operadores deben propagar de forma transitiva las operaciones que efectúen sobre todos los componentes del objeto.

4.1.2 Identidad de los objetos

Sobre esta característica, ya comentada en apartados anteriores cabe destacar en esta ocasión que en todo modelo con identificación de objetos, un objeto tiene una existencia independiente de su valor. Existen dos nociones a tener en cuenta: dos objetos pueden se idénticos (son el mismo objeto) o dos objetos pueden ser iguales (tienen el mismo valor). Esto tiene dos implicaciones que ya hemos comentado: la compartición de objetos y la actualización de objetos.

4.1.3 Encapsulación

La idea de encapsulación tiene su origen en la necesidad de hacer una distinción clara entre la especificación y la implementación de las operaciones y la necesidad de modularidad. Existen dos puntos de vista de la encapsulación: el punto de vista del lenguaje de programación y la adaptación de la base de daros a dicho punto de vista. Desde el punto de vista de los lenguajes de programación, un objeto posee una interfaz y una implementación. La implementación tiene una parte de datos, que representa el estado del objeto y una parte procedural, que describe la implementación de las operaciones. El punto de vista de las bases de datos de este principio, mantiene que un objeto encapsula ambos, programa y datos.

La encapsulación proporciona independencia lógica de los datos: podemos cambiar la implementación de un tipo sin cambiar nada de las aplicaciones que hacen uso de ese tipo. Podemos decir que hemos alcanzado las posibilidades que nos ofrece la encapsulación si sólo son visibles las operaciones y su implementación y los datos se esconden en los objetos.

4.1.4 Tipos y clases

El OODBMS debe soportar tipos o clases. Un tipo en un sistema orientado a objetos, hace referencia a un conjunto de objetos con las mismas características. Se corresponde con la noción de tipo abstracto de datos. Se distinguirá una parte de implementación y una parte de interfaz. En los lenguajes de programación, los tipos son herramientas que incrementan la productividad de la programación, facilitando su corrección. La tipificación de la información facilita que los tipos sean chequeados en tiempo de compilación, y por tanto, la corrección de los programas.

La noción de clase es diferente. La especificación es igual que la de los tipos, pero aplicada más en tiempo de ejecución. Contiene dos aspectos: la clase

como fábrica de objetos y la clase como almacén. La fábrica de objetos se usa para crear nuevos objetos y el almacén se refiere a la clase y su extensión, es decir, el conjunto de objetos que son instancia de la clase. Las clases no se utilizan para chequear la corrección de los programas, sino para crear y manipular objetos.

Es evidente que existen fuertes similitudes entre clases y tipos. La cuestión es cual debe ser la elección en un sistema. Lo que sí es cierto es que todo sistema debe ofrecen un mecanismo de soporte para datos estructurados, sean clases o tipos.

4.1.5 Ocultamiento, sobrecarga y ligadura tardía

El OODBMS no debe permitir una ligadura prematura. Nosotros debemos poder redefinir la implementación de una determinada operación por cada uno de los tipos que la utilizan. A esto se le denomina ocultamiento. El resultado de esto es que un mismo nombre puede definir un número indefinido de implementaciones, a lo que se llama sobrecarga. El sistema será el encargado de determinar cual es la implementación apropiada en cada momento. La característica más importante es que esta elección se realizará en tiempo de ejecución. A esto es a lo que se llama ligadura tardía.

4.1.6 Completitud computacional

El OODBMS debe ser computacionalmente completo. Desde el punto de vista de los lenguajes de programación, esta propiedad es obvia: simplemente significa que se pude expresar cualquier función computable, usando el DML del sistema de bases de datos. Desde el punto de vista de las bases de datos es una novedad, desde que SQL no es completo. No es necesario que los diseñadores de lenguajes diseñen nuevos lenguajes; la completitud computacional se puede introducir a través de los de una conexión razonable con los lenguajes de programación existentes. Debe tenerse en cuenta que no es lo mismo la completitud en cuanto a recursos se refiere, puesto que esto significa tener acceso a todos lo recursos del sistema a través del lenguaje. Un sistema computacionalmente completo no tiene por qué expresar una aplicación completa, sin embargo, si proporciona más potencia que un sistema que sólo almacena y proporciona datos y permite computaciones simples sobre valores atómicos.

4.1.7 Extensibilidad

El OOBDMS debe ser extensible. El OODBMS posee una serie de tipos predefinidos, que pueden ser utilizados por los programadores para crear sus aplicaciones. Este conjunto de tipos debe ser extensible en los siguientes aspectos: debe permitir la definición de nuevos tipos y no debe haber distinción en el uso de los tipos predefinidos y aquellos definidos por el usuario, por lo menos desde el punto de vista de las aplicaciones y sus programadores, aunque el uso que el sistema hace de ellos si que implica ciertas diferencias.

4.1.8 Persistencia

El sistema debe recordar sus datos. Este requerimiento es evidente desde el punto de vista de las bases de datos, pero es una novedad desde el punto de vista de los lenguajes de programación. Ya se ha comentado anteriormente en este texto en qué consiste la persistencia, por ello sólo añadiremos ciertos aspectos como los que se citan a continuación. La persistencia debe ser ortogonal, esto es, cada objeto, independientemente de su tipo, puede ser persistente. Además debe estar implícito el hecho de que el usuario no tiene que hacer nada para que el dato sea persistente.

4.1.9 Gestión del almacenamiento secundario

El OODBMS debe gestionar extensas bases de datos. La gestión del almacenamiento en memoria secundaria es una característica clásica de los sistemas gestores de bases de datos. Esto se consigue a través de diversos mecanismos, que incluyen la gestión de índices, *clustering* de datos, *buffering* de datos, acceso a través de la selección del camino y la optimización de consultas. Ninguno de ellos es visible para el usuario, lo que significa que se establece una clara independencia entre el nivel lógico y el nivel físico del sistema.

4.1.10 Concurrencia

El OODBMS debe aceptar usuarios concurrentes. Ante una situación en la que el sistema tenga que gestionar múltiples usuarios interaccionando con él, debe proporcionar los mismos servicios que los sistemas de bases de datos concurrentes. Debe aportar una coexistencia armoniosa entre los usuarios que trabajen simultáneamente en la base de datos. El sistema debe soportar la noción de atomicidad de una secuencia de operaciones. De este modo la serialización de operaciones debe ofrecerse, así como otro tipo de alternativas.

4.1.11 Recuperación

El OODBMS debe ser capaz de recuperarse de fallos de hardware y software. El sistema debe proporcionar el mismo nivel de servicios que los sistemas de bases de datos concurrentes. En caso de fallos de hardware o software, el sistema debe recuperarse, es decir, volver a un estado coherente de los datos.

4.1.12 Facilidad de consultas *ad hoc*

El OODBMS debe proporcionar una manera sencilla de consulta de datos. El problema es proporcionar la funcionalidad de un lenguaje de consultas *ad hoc*. No se requiere que las consultas se realicen de la manera que lo hace un lenguaje de consulta, pero si que se consiga el mismo servicio. Dicho servicio consiste en permitir al usuario realizar simples consultas a la base de datos. La cuestión es tomar un número de consultas relacionales representativas y determinar cuales de ellas pueden comenzarse con la misma cantidad de trabajo. Esta facilidad puede llevarse a cabo mediante DML.

La facilidad de consulta debe satisfacer los tres siguientes criterios: (1) debe ser de alto nivel, es decir, en pocas palabras debe ser declarativa y

enfatizar el qué se quiere consultar y no el cómo. (2) Debe ser eficiente, lo que significa que la propia formulación de la consulta lleve implícita la optimización de la misma. (3) Debe ser independiente de la aplicación, lo que significa que debe poder llevarse a cabo en cualquier base de datos.

4.2 Características opcionales

Las características que se expondrán a continuación perfeccionan el sistema, pero no hacen a un sistema gestor orientado a objetos. Algunas de ellas tienen origen en el paradigma orientado a objetos y se incluyen en esta categoría porque contribuyen a que un sistema sea "más orientado a objetos". Otras características son simplemente características de las bases de datos que normalmente perfeccionan las funcionalidad del sistema y están orientadas a un aspecto más tecnológico, como facilitar la creación de nuevas aplicaciones tipo CAD / CAM, CASE y automatización de oficina.

4.2.1 Herencia Múltiple

Es opcional en los sistemas proporcionar herencia múltiple. Se considera opcional desde que no existe un acuerdo en la comunidad de la orientación a objeto sobre esta característica.

4.2.2 Comprobación de tipos e inferencia de tipos

El grado de comprobación de tipos que realiza el sistema en tiempo de compilación se deja abierto, pero cuanto más mejor. La situación óptima es aquella en la que un programa que ha sido aceptado por el compilador no produce errores de ejecución. En cuanto a la inferencia de tipos, está abierta al diseñador del sistema: la situación ideal es aquella en la que sólo los tipos base han de ser declarados y el sistema infiere los tipos temporales.

4.2.3 Distribución

Esta característica es ortogonal a la naturaleza orientada a objetos del sistema. De ahí, que el sistema de bases de datos sea distribuido o no.

4.2.4 Diseño de transacciones

En la mayoría de las nuevas aplicaciones, el modelo clásico de transacciones de negocio para sistemas de bases de datos orientadas a objeto no es satisfactorio, puesto que las transacciones tienden a ser muy extensas y el criterio de serialización actual no es adecuado. Por ello, algunos OODBMS dan soporte al diseño de transacciones

4.2.5 Versiones

Muchas de las nuevas aplicaciones (CAD / CAM y CASE) se envuelven en una actividad de diseño y requieren diferentes formas de versionado. Por ello, muchos OODBMS soportan versiones. De nuevo, proveer al sistema de

mecanismos de versionado no constituye una parte de los requerimientos del núcleo del sistema.

4.3 Opciones abiertas

Cualquier sistema que satisface las "reglas de oro", pude etiquetarse como un sistema de bases de datos orientado a objetos. A pesar de ello, quedan a la libre elección de los implementadores muchas opciones de diseño. Estas características se diferencian de las primeras en el sentido de que no existe un consenso alcanzado por la comunidad científica sobre ellas. Difieren de las características opcionales en que nosotros no sabemos cuales de ellas hacen al sistema más o menos orientado a objetos.

4.3.1 Paradigma de programación

No existe razón por la cual se deba imponer un paradigma de programación por encima de otro: el estilo lógico de programación, el estilo funcional de programación, o el estilo imperativo de programación pueden elegirse como paradigmas de programación. Otra solución es que el sistema se independiente del estilo de programación y soporte múltiples paradigmas. Por supuesto, la elección de la sintaxis es libre, y la gente puede discutir indefinidamente sobre cual elegir.

4.3.1 Sistema de representación

El sistema de representación se define mediante un conjunto de tipos y un conjunto de constructores. Desde el momento que se tenga un conjunto minimal de tipos atómicos y constructores (tipos elementales de los lenguajes de programación, conjuntos, tuplas y constructores de listas) suficientes para describir la representación de objetos, pueden extenderse de diferentes formas.

4.3.2 Sistema de tipos

Existe también libertad respecto a la formación de tipos. La única facilidad que se requiere en la formación de tipos es la encapsulación. Pueden existir otros formadores de tipos como los tipos genéricos o los generadores de tipos, restricciones, uniones y flechas (funciones).

Otra opción es que el sistema de tipos sea de segundo orden. Finalmente, el sistema de tipos para variables debe ser más rico que el sistema de tipos para objetos.

4.3.4 Uniformidad

Hay un debate en pie acerca de la uniformidad que se debe esperar en los sistemas: ¿es un tipo o es un objeto? ¿O es que estas nociones deben ser tratadas de forma diferente? Podemos analizar este problema desde tres niveles diferentes: el nivel de implementación, el nivel de lenguaje de programación y el nivel de interfaz.

En el nivel de implementación se debe decidir si la información debe ser clasificada como objetos o bien si debe implementarse un sistema *ad hoc*. La

decisión debe tomarse basándonos en las características y en la facilidad de la implementación, independientemente de la decisión tomada a otros niveles.

Al nivel de lenguaje de programación, la cuestión es la siguiente: ¿son los tipos entidades clases según la semántica del lenguaje? Hay diferentes estilos de uniformidad (sintáctica y semántica). Toda uniformidad a este nivel es inconsistente con el comprobación estática de tipos.

Finalmente, en el nivel de interfaz, se debe tomar otra decisión independiente. Se puede querer presentar la información al usuario con un punto de vista uniforme de los tipos, los objetos y los métodos, aunque en la semántica del lenguaje utilizado, sean nociones de naturaleza diferente. En consecuencia, se pueden presentar como entidades diferentes, siempre que las vistas que el lenguaje de programación tenga sobre ellos sean la misma cosa. Esta decisión se debe tomar basándonos en el criterio humano.

5. Diferencias entre RDBMS y OODBMS

Se analizarán a continuación algunas de las diferencias principales entre los Sistemas Gestores de Bases de Datos Relacionales (RDBMS) y los Sistemas Gestores de Bases de Datos Orientadas a Objeto (OODBMS). Para ello, se procederá a realizar un análisis comparativo de los puntos clave de los sistemas gestores de bases de datos.

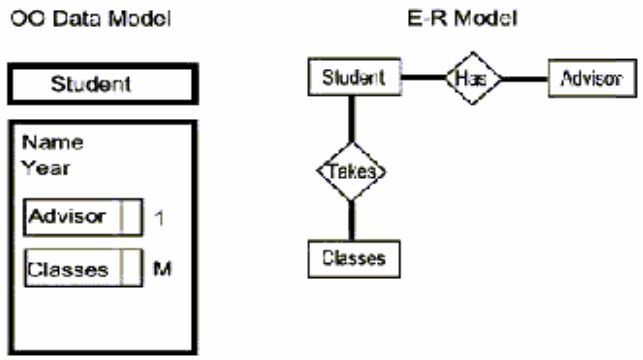

Figura 3: Comparación modelo OO y modelo ER

Como se puede apreciar en la figura, ambos modelos tratan de representar un estudiante que recibe clases y tiene un consejero. El modelo orientado a objetos es una clase que contiene dos atributos, Nombre y Año. Así mismo sirve como contenedor de un objeto Consejero y un objeto Clases. Frente a esto, el otro modelo, Entidad-relación, presenta tres entidades, Estudiante, Consejero y Clases y además, establece dos relaciones entre tablas para representar la información de forma adecuada. Como se puede apreciar, en el modelo relacional se necesitan dos entidades más que en el modelo orientado a objetos para establecer las relaciones que el segundo por si solo ya sugiere.

Los RDBMS buscan una representación de los datos independiente de las aplicaciones destinadas a explotar la base de datos. Mientras, los OODBMS persiguen la equivalencia de los objetos utilizados en la propia base de datos y los objetos utilizados en las aplicaciones que tratarán con la misma.

La interfaz para las diferentes arquitecturas de los RDBMS es siempre la misma: SQL. Sin embargo, los OODBMS requieren una API específica dependiente del lenguaje de programación orientada a objetos utilizado en cada caso.

En cuanto a lo que la representación se refiere, en los RDBMS, ésta se efectúa mediante tablas, previamente normalizadas y que cumplen unas determinadas reglas de integridad. Los OODBMS permiten el empleo de objetos complejos, capaces de albergar colecciones de objetos y de establecer referencias y relaciones con otros objetos.

Los RDBMS siguen un esquema conceptual correspondiente a bases de datos empresariales y sus aplicaciones son explotadas a través de dicho esquema externo. Los OODBMS se fundamentan en objetos persistentes, que mantienen la misma estructura que los objetos transitorios.

Por otra parte, los sistemas fundamentados en el modelo relacional permiten iniciar las consultas a partir de cualquier relación derivable de las relaciones representadas por las tablas de la base de datos. Frente a esta postura, en los sistemas basados en la orientación a objeto es estrictamente necesario que las aplicaciones conozcan el punto de entrada a la base de datos.

6. Ventajas e inconvenientes de las OODB

Los OODBMS aportan grandes beneficios sobre los modelos de bases de datos tradicionales, pero a pesar de ello, poseen puntos débiles que deben tenerse en cuenta y no pasar por encima. Por ello, a continuación analizaremos las ventajas y los inconvenientes de las bases de datos orientadas a objetos, así como sus limitaciones.

6.1 Ventajas

Las ventajas que ofrecen los OODBMS adquieren gran importancia debido a que éstas resuelven gran parte de los problemas que los sistemas tradicionales no podían resolver. En primer lugar, la cantidad de información que un OODBMS puede manejar es mucho mayor y además el modelado de dicha información se convierte en una tarea mucho más fácil.

Por otra parte, los OODBMS aportan objetos más complejos, capaces de soportar la integración de base de datos multimedia, CAD y otros tipos especializados.

Así mismo, poseen capacidades de modelado orientadas a la extensibilidad. Con un OODBMS, está permitido añadir capacidades de modelado con el objetivo de modelar sistemas más complejos. Esta extensibilidad proporciona una solución para incorporar y extender futuras bases de datos en el desarrollo.

Hay que sumar a las ventajas de modelado, las ventajas en cuanto al sistema que los OODBMS aportan. En un OODBMS, el versionado se posibilita, con el fin de ayudar en los cambios de modelado que se produzcan en el sistema. Mediante el versionado, está permitido volver a los juegos de datos previos, y comparar los juegos de datos actuales con ellos.

Otro de los aspectos ventajosos es la reutilización de clases, que juega un papel esencial en lo que a la rapidez en el desarrollo de aplicaciones y su mantenimiento se refiere. Las clases genéricas cuentan con una mayor potencia, pero lo más importante es que pueden ser reutilizadas. Desde que las clases pueden ser reutilizadas, el material redundante no necesita ser diseñado, lo que proporciona, como ya se ha dicho, una mayor facilidad en el desarrollo de aplicaciones y en el mantenimiento de las mismas.

6.2 Inconvenientes

A pesar de que los OODBMS aportan grandes ventajas, los inconvenientes implícitos también son importantes. Los tradicionales sistemas relacionales, debido al tiempo en que han estado en uso, están profundamente arraigados y un cambio puede suponer la pérdida de las ideas ya establecidas. Se requiere que las personas adopten una forma de pensar diferente y, en algunos casos, los usuarios de modelos relacionales no poseen los fundamentos necesarios de orientación a objetos necesarios para trabajar con OODBMS. Educar a las personas en el paradigma orientado a objetos requiere una cantidad de tiempo considerable, dinero y otro tipo de recursos.

Otra desventaja es que es estrictamente necesaria una forma de comunicación y una forma de trabajo conjunta entre los sistemas tradicionales y los OODBMS. Los sistemas tradiciones y los OODBMS deben entenderse entre ellos y entender las relaciones que representan. De nuevo, conseguir este objetivo, requiere un coste temporal, monetario y de recursos. Actualmente, no existe un sistema de comunicación y comprensión.

Además, no existe un lenguaje de consulta específico como SQL. Es curioso que realizar consultas complejas a un OODBMS sea más fácil y a pesar de ello, no existe un lenguaje de consulta para hacerlo. Además no hay ningún estándar lo suficientemente establecido para el diseño y la implementación. La realidad es que los OODBMS son capaces de resolver los problemas de los sistemas tradicionales, pero para ello es necesario el establecimiento de un estándar. Si bien es cierto que ODMG (*Object Database Management Group*) ha dado una ventana temporal de 2 años para establecer una serie completa de estándares desarrollados con la comunidad CPSC.

Por otra parte, los métodos definidos sin un sistema de bases de datos orientado a objetos para el diseño de clases establecen una restricción lógica sobre cómo pueden ser accedidos y manipulados los datos actuales, haciendo que las consultas realizadas "ad hoc" sean difíciles de ejecutar sin la OODB. Para ello, "*The Object Oriented Database System Manifiesto*" propone un *browser* gráfico que proporcione la funcionalidad necesaria para construir consultas simples para el usuario.

En cuanto a la gestión de transacciones, el problema surge debido a que los métodos de las OODB a menudo envuelven a una gran cantidad de datos de gran complejidad y es imperativo no perderlos a la hora de rechazar transacciones (*rollback*). Para ello, al igual que en el modelo relacional, se propone el establecimiento de *savepoints* en las transacciones de larga duración que permitan realizar *rollback*s parciales. El problema, sigue siendo determinar donde ponerlos y cuándo.

7. El estándar ODMG

Con el objetivo de definir estándares para los OODBMS se formó el grupo ODMG, compuesto por varios representantes de la industria de bases de datos. Este grupo propuso un modelo estándar para la semántica de los objetos de una base de datos. La última versión del estándar, ODMG 3.0, propone los siguientes componentes principales de la arquitectura ODMG para un OODBMS:

- Modelo de objetos
- Lenguaje de definición de objetos (ODL)
- Lenguaje de consulta de objetos (OQL)

- Conexión con los lenguajes C++, Smalltalk y Java (al menos)

7.1 Modelo de Objetos

El modelo de objetos ODMG permite que tanto los diseños como las implementaciones, sean portables entre los sistemas que lo soportan. Cuenta con 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 un identificador único. Un literal es un valor específico, como "Eustaquio" o 27. 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 interfaz 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.

7.2 Objetos

Los tipos de objetos de descomponen en atómicos, colecciones y tipos estructurados. Los tipos estructurados, que derivan de la clase interfaz *Collection*, son la propuesta del estándar para las clases contenedor. Entre los tipos colección admitidos por el estándar encontramos los siguientes:

`Set<tipo>, Bag<tipo>, List<tipo>, Array<tipo>, Dictionary<clave,valor>.`

En cuanto a los tipos estructurados encontramos los siguientes:

`Date`, como fecha del calendario.

`Time` , como hora (hora, minutos y segundos).

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

`Interval` ,es un periodo de tiempo.

Los objetos se crean utilizando el método `new()`. Además, todos heredarán la interfaz 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 dentro del dominio de almacenamiento del objeto, que es el ODMS, generado por el SGBD, que no cambia y que no se reutiliza cuando el objeto se borra. Debe notarse que la noción de identificador de objeto es diferente de la noción de clave primaria en el modelo relacional. Una tupla en el modelo relacional se identifica de forma única por el valor de la(s) columna(s) que forma(n) su clave primaria. Si el valor de la columna (o de alguna de ellas) cambia, la tupla cambia su identidad y se convierte en una tupla diferente. Los valores literales no tienen identificador, sino que están embebidos en objetos y no pueden ser referenciados individualmente.

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 y en el momento que el proceso termina, su espacio es liberado. Los objetos persistentes son aquellos que se almacenan en la base de datos y su memoria y almacenamiento son controlados por el sistema de tiempo de ejecución del ODMS. Estos objetos continúan existiendo aún después de que el procedimiento o proceso que los creó haya terminado.

Un aspecto importante del tiempo de vida de los objetos es que es independiente del tipo del objeto. Un objeto de un tipo puede tener instancias con un tiempo de vida transitorio y otras con un tiempo de vida persistente. En el modelo relacional, cualquier tipo conocido por el DBMS es persistente, y cualquiera no conocido (ej. Cualquier tipo no definido utilizando SQL) sólo tiene instancias transitorias.

7.3 Literales

Los tipos literales se descomponen en atómicos, colecciones, estructurados o nulos. No tienen identificadores y además no pueden aparecer solos como objetos, han de estar inmersos en objetos y no pueden referenciarse de forma individual. Entre los atómicos encontramos 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>`.

7.4 Tipos

Entre las características más importantes del paradigma orientado a objetos encontramos la capacidad para distinguir la interfaz pública de una clase de sus elementos privados, a lo que se conoce como encapsulación. El estándar al que nos referimos hace esta distinción hablando de especificación externa de un tipo y sus implementaciones.

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

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

El estándar de ODMG 3.0 soporta la herencia simple y la herencia múltiple mediante las interfaces, ya que al no ser instanciables, se pueden utilizar para especificar operaciones abstractas heredables por otras clases o por otras interfaces. A este hecho se le denomina *herencia de comportamiento*. Este tipo de herencia requiere que el supertipo sea una interfaz, mientras que el subtipo podrá ser una interfaz o una clase.

La herencia es una relación “es un”:

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

Uno de los beneficios prácticos de la herencia es que se puede hacer referencia a los subtipos como supertipos. Por ejemplo, un programa de aplicación puede hacer referencia a los subtipos como supertipo. Por ejemplo, un programa de aplicación puede hacer referencia a sillas, mesas o sofás como muebles o incluso como artículos de venta.

Los subtipos se pueden especializar como sea necesario añadiéndoles comportamientos, a lo que se llama *herencia de estado*. Los subtipos de un subtipo especializado heredan también los comportamientos añadidos. El modelo orientado a objetos utiliza las relación *extiende* (`extends`) para indicar la herencia de estado y de comportamiento. En este tipo de herencia, tanto el subtipo como el supertipo deben ser clases. Las clases que extienden a otra clase ganan acceso a todos lo estados y comportamientos del supertipo, incluyendo cualquier cosa que el supertipo haya adquirido vía 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 debajo de la jerarquía heredan todo lo que sus supertipos heredan de las clases que tienen por encima.

El modelo permite al diseñador que declare una extensión (`extend`) 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 extensión pude 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 una o más propiedades, es una clave compuesta. A diferencia del modelo relacional, las claves únicas no son un requisito.

Una representación de un tipo consta de dos partes: la representación y los métodos. La representación es una estructura de datos dependiente del lenguaje de programación que contiene las propiedades del tipo. Las especificaciones de la implementación vienen de una conexión con un lenguaje (*lenguaje binding*). Esto quiere decir que la representación interna de un tipo será diferente dependiendo del lenguaje de programación que se utilice y de 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 un tipo 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 representar el tipo. Si una aplicación soporta aplicaciones programadas en C++, Java y Smalltalk, entones será necesario tener tres implementaciones para cada tipo, una para cada lenguaje, aunque cada programa utilizará sólo la implementación que corresponda.

7.5 Propiedades

El modelo de objetos ODMG define dos tipos de propiedades: *atributos* y *relaciones*. Un atributo se define del tipo de un objeto, define el estado abstracto de sus instancias. Si un objeto participante en una relación es borrado, también deben borrarse todos los caminos transversales a ese 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 interfaz 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.

Es importante notar que el concepto de camino transversal de una relación no equivale al concepto de puntero en los lenguajes de programación, ya que este último no implica la connotación de que haya otro camino transversal inverso como en una relación. El modelo aporta operaciones para formar (*form*) y eliminar (*drop*) miembros de una relación.

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 protocolo pesimista de control de concurrencia. Todos los accesos, creación, modificación y borrado de objetos persistentes se deben hacer dentro de una transacción. El modelo especifica operaciones para iniciar, terminar (*commit*) y abortar transacciones, así como la operación *checkpoint*. Esta última operación hace permanentes los cambios realizados por la transacción en curso sin liberar ninguno de los bloqueos adquiridos.

7.6 Operaciones

Junto con los atributos y las propiedades de las relaciones, la otra característica de un tipo es su comportamiento, que se especifica mediante una serie de prototipos de operaciones. Cada uno de estos prototipos define el nombre de una operación, el nombre y tipo de cada uno de sus argumentos, el tipo de los valores devueltos y los nombres de cualquier excepción (condiciones de error) que la operación puede producir.

Una operación se define en un único tipo. No hay noción en el modelo de objeto de que una operación pueda existir fuera de un tipo o que una operación esté definida en dos o mas tipos. De esa forma, tipos diferentes pueden tener operaciones diferentes definidas con el mismo nombre (nombres sobrecargados). Cuando se invoca una operación utilizando un nombre sobrecargado debe especificarse una operación concreta para la ejecución, a esto se le llama *resolución de nombre de operación.*

7.7 El lenguaje de definición de objetos ODL

ODL es un lenguaje de especificación para definir tipos de objetos para sistemas complejos compatibles con ODMG. Es el equivalente de DDL (*Data Definition Languaje* o lenguaje de definición de datos) de los DBMS 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*).

En concreto. C++ ODL proporciona una descripción del *schema* de la base de datos como un conjunto de clases de objetos – incluyendo sus atributos, relaciones y operaciones – en un estilo sintáctico que es consistente con la parte de declaración de un programa en C++. Las instancias de las clases pueden ser manejadas mediante C++ OML.

Las declaraciones de atributos son sintácticamente idénticas a las declaraciones de miembros de datos en C++.

Se soportan la sintaxis y la semántica de C++. Sin embargo, las implementaciones no necesitan soportar los siguientes tipos de datos dentro de las clases persistentes como miembros:

- Uniones
- Campos de bits

- Referencias(&)

Las uniones y los campos de bits dan problemas al soportar diferentes entornos. La semántica de las referencias es que son inicializadas una vez en su creación, todas las operaciones siguientes se dirigen al objeto referenciado. Las referencias a los objetos persistentes no pueden reinicializarse al llevarlos de la base de datos a la memoria, y su valor de inicialización no será, en general, válido fuera de los límites del proceso.

Se muestra el uso de ODL mediante un ejemplo:

```
class City;
struct Address
{
  d_UShort      number;
  d_String      street;
  d_Ref_<City>  city;
  Address();
  Address(d_UShort, const char*,const d_Ref<City>&);
};

extern const char _spouse[], _parents[], _children[];

class Person: public d_Object
{
public:
   d_String     name;
   Address      address;
   d_Rel_Ref<Person, _spouse> spouse;
   d_Rel_List<Person, _parents> children;
   d_Rel_List<Person, _children> parents;
   Person(const char *pname);
   void birth(const d_Ref<Person> &child);
   void marriage(const d_Ref<Person> &to_whom);
   d_Ref<d_Set<d_Ref<Person>>> ancestors() const;
   void move(const Address &);
   static d_Ref<d_Set<d_Ref<Person>>> people;
   static cohnst char * const extent_name;
};
```

```
class City: public d_Object
{
public:
   d_Ulong                city_code;
   d_String               name;
   d_Ref<d_Set<d_Ref<Person>>>  population;
   City(int, const char*);
   static d_Ref<d_Set<d_Ref<City>>> cities;
   static const char * const    extent_name;
};

//Implementación del esquema
//Implementación de las clases en C++

#include"schema.hxx"

const char _spouse[]="spouse";
const char _parents[]="parents";
const char _children[]="children";

Address::Address(d_UShort pnum, const char *pstreet, const
             d_Ref<City> &pcity):
             number(pnumber),
             street(pstreet);
             city(pcity)
{}

Address::Address():    number(0),
             street(0),
             city(0)
{}

const char * const Person::extent_name="people";

Person::Person(const char * pname): name(pname)
{
```

```
    people->insert_element(this);
}

void Person::birth(const d_Ref<Person> &child)
{
    children.insert_element_last(child);
    if(spouse)
        spouse->children.insert_element_last(child);
}

void Person::marriage(const d_Ref<Person> &to_whom)
{
    spouse=with;
}

d_Ref<d_Set<d_Ref<Person>>> Person::ancestors()
{
    d_Ref<d_Set<d_Ref<Person>>> the_ancestors =
                        new d_Set<d_Ref<Person>>;
    int i;
    for(i=0;i<2;i++)
        if(parents[i])
        {
            the_ancestors->insert_element(parents[i]);
            d_Ref<d_Set<d_Ref<Person>>> grand_parents =
                        parents[i]->ancestors();
            the_ancestors->union_with(*grand_parents);
            grand_parents.delete_object();
        }
   return the_ancestors;
}

void Person::move(const Address &new_address)
{
    if(address.city)
        address.city->population->remove_element(this);
```

```
      new_address.city->population->insert_element(this);
      mark_modified();
      address=new_address;
   }

   const char * const City::extent_name="cities";

   City::City(d_ULong code, const char *cname): city_code(code),
                                 name(cname)
   {
      cities->insert_element(this);
   }

//Una aplicación

   #include<iostream.h>
   #include"schema.hxx"

   static d_Database dbobj;
   static d_Database * database=&dbobj;

   void Load()
   {
      d_Transaction load;
      load.begin();
      Person::people=new(database) d_Set<d_Ref<Person>>;
      City::cities=new(database) d_Set<d_Ref<City>>;
      Database->set_object_name(Person::people,
                                 Person::extent_name);
      d_Ref<Person> God, Adam, Eve;
      God = new(database,"Person")Person("God");
      Adam = new(database,"Person")Person("Adam");
      Eve = new(database,"Person")Person("Eve");
      Address Paradise(7,"Apple",
                     new(database,"City")City(0,"Garden"));
      Adam->move(Paradise);
```

```
      Eve->move(Paradise);

      God->birth(Adam);
      Adam->marriage(Eve);
      Adam->birth(new(database,"Person")Person("Cain"));
      Adam->birth(new(database,"Person")Person("Abel"));
      load.commit();
   }

   static void print_persons(const d_Collection<d_Ref<Person>>
&s);
   {
      d_Ref<Person> p;
      d_Iterator<d_Ref<Person>> it = s.create_iterator();
      while(it.next(p))
      {
         cout<<"---"<<p->name<<"lives in";
         if(p->address.city)
            cout<<p->address.city->name;
         else
            cout<<"Unknown";
         cout<<endl;
      }
   }

   void Consult()
   {
      d_Transaction          consult;
      d_List<d_Ref<Person>>     list;
      d_Bag<d_Ref<Person>>      bag;
      consult.begin();
      Person::people = database
                     ->lookup_object(Person::extent_name);
      City::cities = database
                     ->lookup_object(City::extent_name);
      cout<<"All the people…:"<<endl;
      print_persons(*Person::people);
```

```
        cout<<"All the people sorted by name…:"<<endl;
        d_oql_execute("select p from people order by name",
list);
        print_persons(list);
        cout<<"People having 2 children and living in
                                                Paradise…:"<<endl;
        d_oql_execute(list, "select p from p in people
                  where p.address.city.name = "Garden"
                  and count(p.children) = 2",bag);
        print_persons(bag);
        Address Earth(13,"Macadam",
                  new(database,"City")City(1,"St.Croix"));
        d_Ref<Person> Adam;
        d_oql_execute("element(select p from p in people
                  where p.name = "Adam")",Adam);
        Adam->move(Earth);
        Adam->spouse->move(Earth);
        Cout<<"Cain's ancestors…:"<<endl;
        d_Ref<Person> Cain = Adam-
>children.retrieve_element_at(0);
        print_persons(*(Cain->ancestors()));
        consult.commit();
    }

    void main(void)
    {
        database->open("family");
        Load();
        Consult();
        Database->close();
    }
```

7.8 El lenguaje de manipulación de objetos OML

Un principio básico para el diseño de OML es que la sintaxis utilizada para crear, borrar, identificar, referenciar, obtener / modificar los valores de las propiedades e invocar operaciones sobre objetos persistentes debe ser no muy diferente de la utilizada para los objetos con periodos de vida más cortos. Una única expresión debe poder mezclar libremente referencias a objetos

transitorios y persistentes. Sin embargo, en este estándar, las colas y la consistencia de transacciones se aplican sólo a los objetos persistentes.

Los objetos pueden ser creados, borrados y modificados. Los objetos se crean en OML utilizando el operador `new()`, que está sobrecargado para aceptar parámetros adicionales indicando el periodo de vida del objeto.

```
(1) void *operator new(size_t size);
(2) void *operator new(size_t size, const
    d_Ref_Any&clustering, const char* typename);
(3) void *operator new(size_t size, d_Database *database,
    const char* typename);
```

(1) se utiliza para la creación de objetos transitorios derivados de d_Object. (2) y (3) crean objetos persistentes.

7.8.1 Borrado de Objetos

Los objetos, una vez creados, pueden borrarse en OML para C++ utilizando la función miembro

```
d_Ref::delete_object();
```

La utilización del operador `delete()` sobre un puntero a un objeto persistente también borrará el objeto, al igual que en C++. El borrado de un objeto es permanente. El objeto se quita de la memoria, y si es un objeto persistente, también se elimina de la base de datos.

7.8.2 Modificación de Objetos

El estado de un objeto se modifica actualizando sus propiedades o invocando operaciones sobre él. Las actualizaciones realizadas a los objetos persistentes se hacen visibles al resto de usuarios de la base de datos cuando la transacción conteniendo la modificación se comete. El cambio en el objeto se comunica invocando la función miembro

```
d_Object::mark_modified();
```

De todas formas, cada vez que un objeto persistente es modificado por una función miembro de actualización proporcionada explícitamente por las clases de ODMG, la llamada a `mark_modified()` no es necesaria, ya que se hace automáticamente.

7.9 El 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 superconjunto de la sentencia SELECT.

OQL no posee primitivas para modificar el estado de los objetos, ya que éstas se deber realizar a través de los métodos que dichos objetos 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 del 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. Este tipo de variables 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 caminos para especificar un camino a atributos y objetos relacionados. Una expresión de camino empieza normalmente con un nombre de objeto persistente o una variable iterador, seguida de ninguno o varios nombres de relaciones o de atributos conectados mediante un punto. 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,
agno: t.agno,
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*.

8. Conclusiones

Tras el periodo de investigación y la elaboración del presente informe, en primer lugar, hemos podido comprobar que la información acerca del tema que se trata, no está al alcance de todo el mundo. La bibliografía existente no es fácil de conseguir y la fuente que constituye INTERNET tampoco es demasiado aceptable, bajo nuestro punto de vista. Esto nos hace recapacitar y pensar que el motivo, como ya sabemos, es que las OODB no están todavía lo suficientemente arraigadas en el entorno de la informática y la computación. Este hecho se ve confirmado por la falta de un estándar capaz de sobrellevar todo el peso de este tipo de bases de datos. En este informe, nos hemos tratado de acercar al estándar ODMG, puesto que éste parece ser uno de los que pueden tener unas perspectivas de futuro más fiable. A pesar de ello, existen otros estándares que tratan de abordar la estandarización de este tema, por ello, no queremos restringir este campo al estándar que hemos presentado.

En definitiva, la conclusión a la que hemos podido llegar, es que el mundo de las OODB es un mundo todavía en pleno desarrollo, pero que teniendo en cuenta las ventajas que ofrece respecto a otros modelos anteriores, constituye el futuro de las bases de datos.

9. Bibliografía

[1] **R. G. G. Cattell *et al.*** *"The Object Data Standard: ODMG 3.0"*. Morgan Kauffmann Publishers, 2000.

[2] **David Jordan.** *"C++ Object Databases: Programming With the ODMG Standard (Object Technology Series)"*. Prentice Hall, 2002.

[3] **Mario Piatinni Velthuis,** *"Bases de Datos Orientadas a Objetos"*. Web site: http://alarcos.inf-cr.uclm.es/doc/bbddavanzadas/mabd3 objetos.pdf

[4] Web site: http://www.service-architecture.com/object-oriented-databases/articles/c++_and_object_databases.html

[5] Web site: http://www.cetus-links.org/oo_data_bases.html

[6] Web site: http://www.eas.asu.edu/~cse494db/notes/oodbNotes/Fundamentals_LR_Jan_25.ppt

[7] **Manifesto de Atkinson,** *"Sistemas Manejadores de Bases de Datos Orientadas a Objetos"*. Web site: http://www2.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html

[8] Web site: http://www.upriss.org.uk/db/42009lecture_index.html

[9] Web site: http://sern.ucalgary.ca/courses/cpsc/547/f97/oodb/

[10] Web site: http://www.acm.org/crossroads/xrds7-3/objects.html

[11] Web site: http://frankiegulledge.com/MIS_619_Advanced_IS_Management/Object_Oriented_Database_Management_Systems.ppt