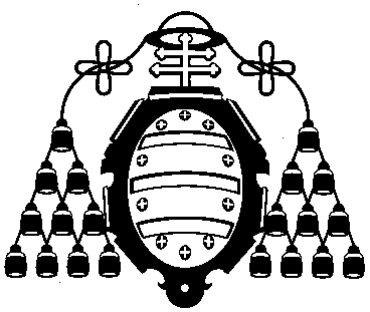
UNIVERSIDAD DE OVIEDO

Departamento de Informática



TESIS DOCTORAL

Un Sistema de Gestión de Bases de Datos Orientadas a Objetos sobre una Máquina Abstracta Persistente

Presentada por
Ana Belén Martínez Prieto
para la obtención del título de Doctora en Informática

Dirigida por el
Profesor Doctor D. Juan Manuel Cueva Lovelle

Oviedo, Mayo de 2001

Resumen

Esta tesis describe la forma en que se pueden utilizar las tecnologías orientadas a objetos para la construcción de un sistema de gestión de bases de datos orientadas a objetos flexible con un mecanismo de indexación extensible que lo convierte en una buena plataforma de experimentación.

Los sistemas de gestión de bases de datos orientadas a objetos son, conceptualmente la solución más completa para gestionar los objetos empleados en las aplicaciones. Sin embargo, en la mayoría de los sistemas existentes los mecanismos y técnicas definidos para proporcionar la funcionalidad están codificados de manera rígida, y no es posible adaptarlos a los requisitos de determinados patrones de aplicación. Además, estos sistemas suelen trabajar sobre sistemas operativos tradicionales e implementando de manera duplicada la mayoría de las abstracciones proporcionadas por éstos.

Una aproximación interesante para solucionar este problema es la que se presenta en este trabajo. La idea se basa en construir un SGBDOO sobre un sistema que ofrezca un soporte directo y común para el paradigma de orientación a objetos. Este soporte es proporcionado por un sistema integral, que está constituido por una máquina abstracta orientada a objetos que ofrece una funcionalidad mínima, y un sistema operativo orientado a objetos que extiende la funcionalidad de la misma. La máquina abstracta proporciona un modelo de objetos común compartido por el resto de elementos del sistema.

El tomar como base para la construcción del SGBDOO el sistema integral antes mencionado garantiza la flexibilidad, la uniformidad en cuanto a la orientación a objetos, y facilita la portabilidad del mismo. Además, evita la re-implementación de las abstracciones ofrecidas por el sistema operativo, ya que el SGBDOO las puede utilizar tal cual o extenderlas sin re-escribirlas en su totalidad.

Como ejemplo de estas ventajas, el mecanismo de indexación diseñado es extensible, permitiendo fácilmente la incorporación de nuevas técnicas de indexación. Estas técnicas se implementan como conjuntos de objetos con una interfaz determinada, que deberá ser respetada por cada nueva técnica de indexación a incorporar mediante herencia. Las clases empleadas pueden ser usadas y extendidas por cualquier usuario del sistema, reutilizándose así el propio código del SGBDOO.

Para comprobar la factibilidad del modelo se desarrolla un prototipo que es implementado sobre un sistema integral experimental, Oviedo3, basado en una versión de la máquina abstracta que proporciona persistencia. Como ejemplo de la plataforma de experimentación construida, en este prototipo se han incluido tres técnicas de indexación: *Single Class*, *Ch-Tree y Path Index*. Los resultados obtenidos, en comparación con otros gestores existentes, no son en absoluto desalentadores. Adicionalmente, se desarrolla también una herramienta que implementa completamente el estándar ODMG 2.0 en su *binding* para Java, generando código para diferentes gestores de bases de datos con el objetivo de compararlos entre si más fácilmente.

Palabras clave

Orientación a objetos, modelo de objetos, máquinas abstractas, sistemas integrales orientados a objetos, sistemas operativos orientados a objetos, sistemas de gestión de bases de datos, sistemas de gestión de bases de datos orientadas a objetos, técnicas de indexación, lenguajes de bases de datos, flexibilidad, uniformidad.

Abstract

This thesis describes the way in which object-oriented technologies can be used for the construction of a flexible object-oriented database management system with a extensible indexing mechanism, which transforms the system into a good platform for benchmarking, researching and experimenting.

The object-oriented database management systems are conceptually the most complete solution to manage the application objects. However, in the majority of the current systems the mechanisms and techniques for providing this capability are codified in an inflexible way. It is not possible to adapt them to concrete application needs. Moreover, these systems are used to work over traditional operating systems and to implement again its abstractions.

An interesting approach to solve this problem is presented in this work. The idea is based on building an OODBMS over a system which provides direct support for the object-oriented paradigm. This support is provided by an integral system. This integral system is supported by an object-oriented abstract machine that provides a set of minimal capabilities, and an object-oriented operating system that extends them. The abstract machine provides a common object model shared for the other system elements.

Building the OODBMS over the before mentioned integral system guarantees flexibility, OO uniformity, and facilitates its portability. Re-implementation of operating system abstractions is avoided, since these abstractions can be used by the OODBMS without changes, or can be extended without having to fully implement them.

As an example of these advantages, the designed indexing mechanism allows easily the incorporation of new indexing techniques. These techniques are implemented like objects sets with a particular interface. Every new indexing technique added to the system, by using inheritance, will have that interface. These classes can be used and extended by any user, thus reusing the OODBMS code itself.

As a proof of the feasibility of the model, a prototype was developed over an experimental integral system, named Oviedo3, based on a version of the abstract machine that provides persistence. In the benchmarking platform built over this prototype three indexing techniques have been included: *Single Class*, *CH-Tree* y *Path Index*. The obtained results, compared with other systems are encouraging. Additionally, a tool that completely implements the ODMG 2.0 Java binding was built. It generates code for different database management systems with the objective of easing the comparison between them.

Keywords

Object-orientation, object model, abstract machines, object-oriented integral systems, object-oriented operating systems, database management systems, object-oriented database management systems, indexing techniques, database languages, flexibility, uniformity.

Agradecimientos

A mis padres

Quisiera reflejar en estas líneas mi agradecimiento a todos mis compañeros, muy especialmente a los integrantes del grupo Oviedo3, por todo el apoyo tanto profesional como moral que me han proporcionado durante los años de elaboración de esta tesis doctoral, y que de una forma u otra han colaborado en la elaboración de la misma.

Gracias a Darío por sus valiosos intercambios de opinión, consejos y por haberse tomado la molestia de revisar este documento, y como no, a Marian, por su preocupación constante y su inestimable apoyo sobre todo en estos últimos meses.

Agradecimientos en especial para mi director de Tesis, Juan Manuel Cueva. Su dedicación y apoyo recibidos durante los años de elaboración de esta tesis, así como su constante estímulo me han permitido completarla.

Tabla de Contenidos

CAPÍTULO 1 OBJETIVOS Y ORGANIZACIÓN DE ESTA TESIS DOCTORAL 1

1.1 OBJETIVOS 1
1.2 EL PROYECTO DE INVESTIGACIÓN EN EL QUE ESTÁ ENMARCADA 1
1.3 FASES DE DESARROLLO 2
1.4 ORGANIZACIÓN DE LOS CONTENIDOS 3
1.4.1 Antecedentes 3
1.4.2 Necesidad y aportaciones de un sistema integral para la construcción del SGBDOO 4
1.4.3 Solución 4
1.4.4 Prototipo 5
1.4.5 Conclusiones 6

CAPÍTULO 2 NECESIDAD DE UN SGBDOO UNIFICADO CON UN SISTEMA INTEGRAL 7

2.1 POSIBILIDADES PARA LA GESTIÓN DE OBJETOS 7
2.2 GESTIÓN DE OBJETOS BASADA EN EL MODELO RELACIONAL 7
2.2.1 Bases de datos objeto-relacionales 7
2.2.2 Mediadores o envoltorios objeto-relacionales 8
2.3 GESTIÓN DE OBJETOS BASADA EN EL MODELO ORIENTADO A OBJETOS 9
2.3.1 Lenguajes persistentes orientados a objetos 10
2.3.2 Bases de datos orientadas a objetos 11
2.3.3 Gestores de objetos 12
2.4 PROBLEMÁTICA CON LAS BASES DE DATOS ORIENTADAS A OBJETOS 12
2.4.1 Código intruso para gestionar la base de datos 12
2.4.2 Carencia de interoperabilidad 13
2.4.3 Difícil portabilidad 14
2.4.4 Extensibilidad limitada 14
2.5 PROBLEMÁTICA CON LOS SISTEMAS OPERATIVOS 15
2.5.1 Desadaptación de impedancias 15
2.6 PROBLEMÁTICA CON EL MODELO DE OBJETOS 16
2.7 INCONVENIENTES DERIVADOS 16
2.8 SGBDOO UNIFICADO CON EL SISTEMA INTEGRAL 17

CAPÍTULO 3 CONCEPTOS BÁSICOS SOBRE SGBDOO 19

3.1 QUÉ ES UN SISTEMA DE GESTIÓN DE BASES DE DATOS ORIENTADAS A OBJETOS 19
3.1.1 Manifiesto de bases de datos orientadas a objetos 19
3.2 FUNCIONALIDAD DE BASE DE DATOS 21
3.2.1 Persistencia 21
3.2.2 Concurrencia y recuperación 22
3.2.3 Gestión del almacenamiento secundario 24
3.2.4 Protección 26
3.2.5 Lenguaje de consulta 27
3.2.6 Evolución 27
3.3 ALGUNAS CARACTERÍSTICAS DEL MODELO DE OBJETOS 29
3.3.1 Identidad de objetos 29
3.3.2 Atributos 30
3.3.3 Relaciones 30
3.3.4 Objetos compuestos 32
3.3.5 Almacenamiento y ejecución de métodos 32
3.4 ARQUITECTURA DE BASES DE DATOS ORIENTADAS A OBJETOS 33
3.4.1 Cliente-servidor 33
3.4.2 Bases de datos distribuidas 35
3.5 INTERFAZ DE USUARIO Y HERRAMIENTAS DE ALTO NIVEL 36
3.5.1 Herramientas de desarrollo 37
3.5.2 Herramientas de administración 37

CAPÍTULO 4 REQUISITOS DE UN SGBDOO 39

4.1 APLICACIÓN DE LAS TECNOLOGÍAS DE OBJETOS A LA CONSTRUCCIÓN DEL SGBDOO 39
4.1.1 Diseño del SGBDOO como un marco orientado a objetos 39
4.1.2 Objeto como abstracción única 40
4.1.3 Modelo de objetos uniforme 40
4.2 INTEROPERABILIDAD 40
4.3 HETEROGENEIDAD Y PORTABILIDAD 40
4.4 TRANSPARENCIA 40
4.5 INTEGRACIÓN TRANSPARENTE CON EL SISTEMA OPERATIVO 41
4.6 FLEXIBILIDAD, EXTENSIBILIDAD Y ADAPTABILIDAD 41
4.7 INDEPENDENCIA ENTRE MECANISMOS 41
4.8 INDEPENDENCIA DE MECANISMOS Y POLÍTICAS 42
4.9 RESUMEN DE LOS REQUISITOS DEL SGBDOO 42
4.9.1 Relativos a la construcción del SGBDOO 42
4.9.2 Relativos a la funcionalidad de base de datos 42
4.9.3 Relativos al soporte de un modelo de objetos 43
4.9.4 Relativos a la interfaz de usuario 43

CAPÍTULO 5 PANORAMA DE ALGUNOS SISTEMAS PARA EL ALMACENAMIENTO DE OBJETOS 45

5.1 STARBURST 45
5.1.1 Arquitectura 46
5.1.2 Mecanismos de extensión 47
5.1.3 Lenguaje de consulta Hydrogen 48
5.1.4 Crítica 48
5.1.5 Características interesantes 48
5.2 POSTGRES 49
5.2.1 Arquitectura 49
5.2.2 Características del modelo 50
5.2.3 Incorporación de nuevos métodos de acceso 50
5.2.4 Extensibilidad dirigida por catálogos 50
5.2.5 Interfaces para diferentes lenguajes de programación 51
5.2.6 Crítica 51
5.2.7 Características interesantes 51
5.3 OBJECT DRIVER 52
5.3.1 Arquitectura 53
5.3.2 Persistencia por alcance 53
5.3.3 Transacciones basadas en el servicio de transacciones del SGBDR subyacente 53
5.3.4 Lenguaje de consulta OQL 54
5.3.5 Crítica 54
5.3.6 Características interesantes 55
5.4 OBJECT FILE 56
5.4.1 Arquitectura 56
5.4.2 Características básicas 56
5.4.3 Crítica 56
5.4.4 Características interesantes 57
5.5 DARMSTADT 58
5.5.1 Arquitectura 58
5.5.2 Modelo de datos basado en registros complejos 59
5.5.3 Características del núcleo 59
5.5.4 Comunicación entre el front-end y el núcleo mediante object buffer 59
5.5.5 Crítica 59
5.5.6 Características interesantes 60
5.6 STREAMSTORE 61
5.6.1 Arquitectura 61
5.6.2 Persistencia por llamada explícita 62
5.6.3 Diferentes métodos para la recuperación de la información 62
5.6.4 Indexación basada en B-Tree 63
5.6.5 Concurrencia asociada a sesiones 63
5.6.6 Crítica 63

5.6.7 Características interesantes *64*
5.7 JEEVAN 65
5.7.1 Arquitectura *65*
5.7.2 Persistencia por llamada explícita *65*
5.7.3 Declaración explícita de las clases persistentes *66*
5.7.4 Diferentes métodos para la recuperación de información *66*
5.7.5 Indexación sobre campos de tipo simple *66*
5.7.6 Control de versiones basado en el mecanismo de serialización de Java *66*
5.7.7 Crítica *67*
5.7.8 Características interesantes *67*
5.8 OBJECT STORE PSE Y PSE PRO PARA JAVA 69
5.8.1 Arquitectura *69*
5.8.2 Persistencia por alcance y con postprocesamiento *69*
5.8.3 Eliminación con recolección *70*
5.8.4 Concurrencia asociada a sesiones *70*
5.8.5 Evolución de esquemas basada en la serialización de Java *70*
5.8.6 Consultas basadas en colecciones *70*
5.8.7 Indexación no extensible basada en B-Tree y tablas Hash *71*
5.8.8 Crítica *71*
5.8.9 Características interesantes *72*
5.9 JASMINE 73
5.9.1 Arquitectura *73*
5.9.2 Gestor en capas basado en la tecnología relacional *74*
5.9.3 Buffering de objetos y de páginas *75*
5.9.4 Indexación para soportar jerarquías de herencia y agregación *75*
5.9.5 Lenguaje de programación Jasmine/C *75*
5.9.6 Lenguaje de consulta ODQL *76*
5.9.7 Crítica *76*
5.9.8 Características interesantes *76*
5.10 POET 78
5.10.1 Arquitectura *78*
5.10.2 Persistencia por alcance y con preprocesamiento *78*
5.10.3 Eliminación con recolección *79*
5.10.4 Concurrencia y checkpoints *79*
5.10.5 Seguridad y Recuperación *79*
5.10.6 Evolución de esquemas con versiones *80*
5.10.7 Indexación básica sobre los identificadores y definición de índices adicionales *80*
5.10.8 Crítica *80*
5.10.9 Características interesantes *81*
5.11 EXODUS 82
5.11.1 Arquitectura *82*
5.11.2 Soporta tipos de datos y operadores definidos por el usuario *83*
5.11.3 Núcleo de EXODUS: el gestor de objetos de almacenamiento *83*
5.11.4 Lenguaje de programación E *83*
5.11.5 Permite la construcción de métodos de acceso genéricos *84*
5.11.6 Versiones en los objetos de almacenamiento *84*
5.11.7 Generador de optimizadores de consultas basado en reglas *84*
5.11.8 Crítica *85*
5.11.9 Características interesantes *85*
5.12 RESUMEN DE CARACTERÍSTICAS DE LOS SISTEMAS REVISADOS 86

CAPÍTULO 6 NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS PARA LA CONSTRUCCIÓN DEL SGBDOO 87

6.1 NECESIDAD DE UN SISTEMA INTEGRAL ORIENTADO A OBJETOS 87
6.2 CARACTERÍSTICAS DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS 87
6.3 APORTACIONES DEL SISTEMA INTEGRAL 88
6.3.1 Uniformidad conceptual en torno a la orientación a objetos *88*
6.3.2 Transparencia *89*
6.3.3 Heterogeneidad, portabilidad e Interoperabilidad *89*
6.3.4 Seguridad *89*

6.3.5 Concurrencia *89*
6.3.6 Flexibilidad *89*
6.3.7 Modelo de Objetos *90*

CAPÍTULO 7 ARQUITECTURA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS 91

7.1 SISTEMA INTEGRAL OO = MÁQUINA ABSTRACTA OO + SISTEMA OPERATIVO OO 91
7.2 PROPIEDADES FUNDAMENTALES DE LA ARQUITECTURA 91
7.2.1 Máquina abstracta orientada a objetos *91*
7.2.2 Sistema operativo orientado a objetos *91*
7.2.3 Orientación a objetos *92*
7.2.4 Espacio único de objetos sin separación usuario/sistema *92*
7.2.5 Identificador de objetos único, global e independiente *92*
7.3 MODELO DE OBJETOS ÚNICO PARA EL SISTEMA INTEGRAL 92
7.3.1 Parte del modelo en la máquina abstracta *93*
7.3.2 Parte del modelo en el sistema operativo *93*
7.4 ARQUITECTURA DE REFERENCIA DE LA MÁQUINA ABSTRACTA 93
7.4.1 Propiedades fundamentales de una máquina abstracta para un SIOO *94*
7.4.2 Principios de diseño de la máquina *94*
7.4.3 Estructura de referencia *94*
7.4.4 Juego de instrucciones *95*
7.4.5 Ventajas del uso de una máquina abstracta *96*
7.5 EXTENSIÓN DE LA MÁQUINA ABSTRACTA PARA PROPORCIONAR LA FUNCIONALIDAD DEL SISTEMA OPERATIVO. REFLECTIVIDAD 96
7.5.1 Modificación interna de la máquina *96*
7.5.2 Extensión de la funcionalidad de las clases básicas *96*
7.5.3 Colaboración con el funcionamiento de la máquina. Reflectividad *97*

CAPÍTULO 8 MODELO DE PERSISTENCIA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS 99

8.1 SUBSISTEMA DE PERSISTENCIA DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS 99
8.2 ELEMENTOS BÁSICOS DE DISEÑO DEL SISTEMA DE PERSISTENCIA 99
8.2.1 Persistencia completa *100*
8.2.2 Estabilidad y elasticidad *100*
8.2.3 Encapsulamiento de la computación *100*
8.2.4 Identificador uniforme único *100*
8.2.5 Eliminación explícita de objetos *100*
8.2.6 Ventajas *100*
8.3 IMPLEMENTACIÓN DE LA PERSISTENCIA EN EL SISTEMA INTEGRAL 101
8.3.1 Área de instancias virtual persistente *101*
8.3.2 Identificador uniforme de objetos igual al identificador de la máquina *102*
8.3.3 Mecanismo de envío de mensajes y activación del sistema operativo por reflexión explícita *102*
8.3.4 Objeto "paginador" *102*
8.3.5 Carga de objetos *102*
8.3.6 Reemplazamiento *103*
8.3.7 Manipulación interna *103*

CAPÍTULO 9 OTRAS APORTACIONES DEL SISTEMA INTEGRAL ORIENTADO A OBJETOS 105

9.1 SUBSISTEMA DE PROTECCIÓN 105
9.1.1 Modelo de protección *105*
9.1.2 Implantación en la máquina abstracta *107*
9.1.3 Modo de operación del mecanismo *107*
9.2 SUBSISTEMA DE DISTRIBUCIÓN 109
9.2.1 Características de la distribución *109*
9.2.2 Migración de objetos *109*
9.2.3 Invocación de métodos remota *109*
9.3 SUBSISTEMA DE CONCURRENCIA 109
9.3.1 Transacciones software *110*
9.3.2 Introducción de transacciones en el sistema integral *111*

CAPÍTULO 10 ARQUITECTURA DEL SGBDOO 113

10.1 ESTRUCTURA DE REFERENCIA DEL SGBDOO 113
10.1.1 Motor o núcleo del sistema 114
10.1.2 Lenguajes 114
10.1.3 Herramientas Visuales 115
10.2 PROPIEDADES BÁSICAS DEL MOTOR DERIVADAS DEL MODELO DE OBJETOS 115
10.2.1 Identidad única: identificador único de objetos 116
10.2.2 Abstracción y encapsulamiento. Clases 116
10.2.3 Relaciones 116
10.2.4 Polimorfismo 117
10.2.5 Concurrencia 117
10.2.6 Persistencia 118
10.2.7 Seguridad 119
10.2.8 Distribución 121
10.3 FUNCIONALIDAD BÁSICA PROPORCIONADA POR EL MOTOR 122
10.4 GESTOR DE EXTENSIONES 123
10.4.1 Información necesaria 124
10.4.2 Funciones a proporcionar 124
10.5 GESTOR DE ESQUEMAS 125
10.5.1 Información del esquema 126
10.5.2 Funciones a proporcionar 127
10.5.3 Evolución de esquemas 128
10.6 GESTOR DE INTEGRIDADES 128
10.6.1 Esquema de la base de datos 129
10.6.2 Validación de las referencias 129
10.6.3 Integridad de las relaciones 130
10.7 GESTOR DE CONSULTAS 131
10.8 GESTOR DE ÍNDICES 131
10.9 RESUMEN DE LAS CARACT ERÍSTICAS OFRECIDAS POR ESTA ARQUITECTURA 132

CAPÍTULO 11 PANORAMA DE LAS TÉCNICAS DE INDEXACIÓN EN BASES DE DATOS ORIENTADAS A OBJETOS 133

11.1 CARACTERÍSTICAS DE LOS LENGUAJES DE CONSULTA ORIENTADOS A OBJETOS QUE AFECTAN A LAS TÉCNICAS DE INDEXACIÓN 133
11.2 CLASIFICACIÓN DE LAS TÉCNICAS DE INDEXACIÓN EN ORIENTACIÓN A OBJETOS 134
11.3 TÉCNICAS BASADAS EN LA JERARQUÍA DE HERENCIA 134
11.3.1 Single Class (SC) 134
11.3.2 CH-Tree 135
11.3.3 H-Tree 136
11.3.4 hcC-Tree 138
11.4 TÉCNICAS BASADAS EN LA JERARQUÍA DE AGREGACIÓN 140
11.4.1 Path (PX) 141
11.4.2 Nested (NX) 142
11.4.3 Multiindex (MX) 143
11.4.4 Path Dictionary Index (PDI) 144
11.4.5 Join Index Hierarchy (JIH) 148
11.5 TÉCNICAS BASADAS EN LA JERARQUÍA DE HERENCIA Y DE AGREGACIÓN 152
11.5.1 Nested Inherited 152
11.5.2 Inherited MultiIndex 153
11.6 TÉCNICAS BASADAS EN LA INVOCACIÓN DE MÉTODOS 153
11.6.1 Materialización de métodos 154
11.7 RESUMEN DE CARACTERÍSTICAS DE LAS TÉCNICAS REVISADAS 156

CAPÍTULO 12 MECANISMO DE INDEXACIÓN 157

12.1 INTRODUCCIÓN 157
12.2 OBJETIVOS DEL MECANISMO 158
12.2.1 Extensibilidad 159
12.2.2 Plataforma de experimentación 159
12.3 CARACTERÍSTICAS FUNDAMENTALES DEL MECANISMO 159
12.3.1 Soporte para nuevos tipos de datos 159

12.3.2 Indexación aplicable a cualquier tipo de datos *160*
12.3.3 Incorporación de nuevas técnicas de indexación *160*
12.3.4 Integración con el optimizador de consultas *160*
12.3.5 Configurabilidad *160*
12.4 ALTERNATIVAS A CONSIDERAR PARA EL DISEÑO DEL MECANISMO 161
12.4.1 Núcleo fijo con extensiones *161*
12.4.2 Núcleo extensible *162*
12.5 CUESTIONES DE DISEÑO DEL MECANISMO 163
12.5.1 Marco orientado a objetos *163*
12.5.2 Núcleo extensible *164*
12.5.3 Recursos proporcionados por el sistema integral *164*
12.5.4 Representación de las técnicas de indexación *165*
12.5.5 Representación de los operadores *166*
12.5.6 Catálogo para la indexación *167*
12.6 CARACTERÍSTICAS POR DEFECTO DEL MECANISMO 168
12.6.1 Técnicas de indexación a incluir inicialmente en el mecanismo *168*
12.6.2 Operadores a incluir inicialmente en el mecanismo *169*
12.7 REQUISITOS PREVIOS PARA LA UTILIZACIÓN DEL MECANISMO DE INDEXACIÓN 169
12.8 APLICACIONES DEL MECANISMO DE INDEXACIÓN 170
12.9 RESUMEN DE LAS CARACTERÍSTICAS PROPORCIONADAS POR EL MECANISMO DE INDEXACIÓN 170

CAPÍTULO 13 ELECCIÓN DEL LENGUAJE PARA EL SGBDOO 173

13.1 LENGUAJES DE BASES DE DATOS DESDE EL PUNTO DE VISTA TRADICIONAL 173
13.1.1 Desadaptación de Impedancias *173*
13.2 CARACTERÍSTICAS DE UN LENGUAJE PARA UNA BASE DE DATOS ORIENTADA A OBJETOS 174
13.2.1 Lenguaje de programación orientado a objetos como lenguaje de definición y manipulación *174*
13.2.2 Necesidad de un lenguaje de consulta *174*
13.2.3 Requerimientos del lenguaje de consulta *175*
13.2.4 Persistencia ortogonal *177*
13.3 POSIBILIDADES PARA BDOVIEDO3 178
13.3.1 Diseñar y construir un nuevo lenguaje de base de datos *178*
13.3.2 Adopción de un lenguaje de base de datos ya existente *178*
13.3.3 Adopción de los lenguajes propuestos por el estándar ODMG *179*
13.4 LENGUAJE SELECCIONADO: PROPUESTO POR EL EST ÁNDAR ODMG 180
13.4.1 Motivaciones para adoptar el estándar en su binding para Java *181*
13.4.2 Deficiencias en el estándar *182*
13.4.3 Extensiones propuestas al estándar *184*
13.5 ADAPTACIÓN DEL MODELO DE OBJETOS DE ODMG AL SISTEMA INTEGRAL 184

CAPÍTULO 14 REQUISITOS DE LA INTERFAZ VISUAL PARA EL SGBDOO 185

14.1 INTRODUCCIÓN 185
14.2 CARACTERÍSTICAS DESEABLES PARA LA INTERFAZ DE UN SGBDOO 185
14.3 FUNCIONALIDAD BÁSICA DE LA INTERFAZ 186
14.3.1 Generador/Visualizador del esquema de la base de datos *186*
14.3.2 Navegación-inspección *188*
14.3.3 Entorno de consultas *188*

CAPÍTULO 15 SISTEMA INTEGRAL ORIENTADO A OBJETOS OVIEDO3 191

15.1 EL SISTEMA INTEGRAL ORIENTADO A OBJETOS OVIEDO3 191
15.2 LA MÁQUINA ABSTRACTA CARBAYONIA 192
15.2.1 Estructura *192*
15.2.2 El lenguaje Carbayón: juego de instrucciones *193*
15.2.3 Características de las clases Carbayonia *194*
15.2.4 Jerarquía de clases básicas *196*
15.2.5 Clase básica objeto *197*
15.3 EL SISTEMA OPERATIVO SO4 199
15.3.1 Persistencia *199*
15.3.2 Seguridad *200*
15.3.3 Concurrencia *200*

15.3.4 Distribución *201*

CAPÍTULO 16 DISEÑO E IMPLEMENTACIÓN DE UN PROTOTIPO DEL MOTOR PARA EL SGBDOO 203

16.1 INTRODUCCIÓN 203
16.2 CARACTERÍSTICAS PROPORCIONADAS POR EL SISTEMA INTEGRAL 203
16.2.1 Persistencia ortogonal no completa *203*
16.2.2 Declaración explícita de las clases persistentes *203*
16.2.3 Persistencia por llamada explícita *204*
16.2.4 Transparencia para el usuario en el trasiego disco/memoria *204*
16.2.5 Servidor de páginas *204*
16.2.6 Identificador con dirección estructurada *204*
16.2.7 Buffering *205*
16.2.8 Lenguaje Carbayón *205*
16.3 FUNCIONALIDAD PROPORCIONADA POR EL PROTOTIPO 205
16.3.1 Gestión de los esquemas *205*
16.3.2 Gestión de las extensiones *206*
16.3.3 Gestión de los índices *206*
16.3.4 Facilidad de consultas limitada *206*
16.4 CARACTERÍSTICAS DEL MECANISMO DE INDEXACIÓN IMPLEMENTADO 206
16.4.1 Incorporación de nuevas técnicas de indexación *206*
16.4.2 Indexación aplicable a cualquier tipo de dato *207*
16.4.3 Configurabilidad del mecanismo *207*
16.4.4 Gestión automática de los índices *207*
16.4.5 Propagación de los índices por la jerarquía *207*
16.4.6 Técnicas implementadas *207*
16.4.7 Flexibilidad estática *207*
16.4.8 Plataforma de experimentación *207*
16.5 ARQUITECTURA 207
16.6 DESCRIPCIÓN DE LA JERARQUÍA DE CLASES 208
16.6.1 Clase TOODB *209*
16.6.2 Clase TOODBItem *209*
16.6.3 Clase TClassStructure *213*
16.6.4 Clase TRepositoryItem *214*
16.6.5 Clase TObjectIndexList *214*
16.6.6 Clase TStructIndex *216*
16.6.7 Clases para la implementación de la técnica SC *217*
16.6.8 Clases para la implementación de la técnica CH-Tree *218*
16.6.9 Clases para la implementación de la técnica Path Index *218*
16.7 DESCRIPCIÓN DINÁMICA DEL PROTOTIPO 219
16.7.1 Gestión de instancias *219*
16.7.2 Gestión de clases *220*
16.7.3 Gestión de índices *220*
16.7.4 Realización de consultas elementales *221*
16.8 EXTENSIBILIDAD DEL MECANISMO 222
16.8.1 Incorporación de nuevas técnicas de indexación *223*
16.8.2 Registro y selección de la técnica de indexación *223*

CAPÍTULO 17 INTRODUCCIÓN DE LOS LENGUAJES EN EL SGBDOO 225

17.1 INCORPORACIÓN DE LOS LENGUAJES AL SGBDOO 225
17.1.1 Características básicas del traductor Java/Carbayón *226*
17.2 OBSERVACIONES SOBRE EL *BINDING* JAVA IMPLEMENTADO 226
17.2.1 Claves *226*
17.2.2 Estructuras *227*
17.2.3 Excepciones *227*
17.2.4 Extents *227*
17.2.5 Índices *227*
17.2.6 Modificadores *228*
17.2.7 Módulos *228*
17.2.8 Clase Database *228*

17.2.9 Clase Transaction *229*
17.2.10 Relaciones *230*
17.2.11 Tipos de datos *230*
17.3 HERRAMIENTA PARA LA EXPERIMENTACIÓN 230
17.3.1 Objetivos *231*
17.4 DISEÑO Y CONSTRUCCIÓN DE LA HERRAMIENTA 231
17.4.1 Diagrama de clases básicas *232*
17.4.2 Gestión de la tabla de símbolos *233*
17.4.3 Gestión del traductor *235*
17.4.4 Generación de código *237*
17.4.5 Gestión de la extensiones de las clases *239*
17.4.6 Gestión de las relaciones *240*
17.4.7 Paquete OML *241*
17.5 EJEMPLO SENCILLO DE UTILIZACIÓN DE LA HERRAMIENTA 243
17.5.1 En línea de comandos *243*
17.5.2 Mediante la interfaz WIMP *243*
17.6 UTILIDAD DE LA HERRAMIENTA 245

CAPÍTULO 18 EVALUACIÓN DEL PROTOTIPO 247

18.1 INTRODUCCIÓN 247
18.2 COMPARACIÓN DE LAS DIFERENTES TÉCNICAS DE INDEXACIÓN IMPLEMENTADAS 247
18.2.1 Condiciones de la comparativa *247*
18.2.2 Rendimiento de las técnicas de indexación *248*
18.2.3 Conclusiones *251*
18.3 COMPARACIÓN CON OTROS GESTORES 252
18.3.1 Consideraciones previas *252*
18.3.2 Condiciones de la comparativa *252*
18.3.3 Rendimientos sin considerar la indexación en BDOViedo3 ..... *254*
18.3.4 Rendimientos considerando la indexación en BDOViedo3 ..... *254*
18.3.5 Conclusiones *256*

CAPÍTULO 19 TRABAJO RELACIONADO 259

19.1 GOA++ 259
19.1.1 Arquitectura *259*
19.1.2 Soporte para distribución basado en CORBA *261*
19.1.3 Procesamiento paralelo *261*
19.1.4 Experimentación con la indexación *261*
19.1.5 Relación *261*
19.2 TIGUKAT 262
19.2.1 Modelo de objetos *262*
19.2.2 Lenguajes *263*
19.2.3 Arquitectura *263*
19.2.4 Relación *264*
19.3 MOOD 265
19.3.1 Modelo de objetos *265*
19.3.2 Arquitectura *265*
19.3.3 Relación *267*

CAPÍTULO 20 CONCLUSIONES 269

20.1 RESULTADOS DESTACABLES 270
20.1.1 Uniformidad *270*
20.1.2 Integración transparente con el sistema operativo *271*
20.1.3 Portabilidad *271*
20.1.4 Interoperabilidad *271*
20.1.5 Flexibilidad *271*
20.1.6 Extensibilidad *271*
20.1.7 Adaptabilidad *271*
20.1.8 Portabilidad de las aplicaciones *271*
20.1.9 Implementación completa de los lenguajes propuestos por ODMG 2.0 en su binding para Java *272*

20.2 VENTAJAS DEL SISTEMA DISEÑADO 272
20.2.1 Base de datos como sistema de almacenamiento global 272
20.2.2 Plataforma de experimentación 272
20.2.3 Desarrollo de sistemas de alta productividad 273
20.3 TRABAJO Y LÍNEAS DE INVESTIGACIÓN FUTURAS 273
20.3.1 Incorporación del resto de funcionalidades del gestor 273
20.3.2 Desarrollo de una interfaz de usuario orientada a objetos 273
20.3.3 Implementación de los lenguajes propuestos por el estándar ODMG 3.0 274
20.3.4 Mejora en la implementación de la máquina 274
20.3.5 Aprovechamiento de la reflectividad para la construcción del SGBDOO 274
20.3.6 Subsistema de persistencia apoyado en el motor de la base de datos 274

APÉNDICE A ESTÁNDAR ODMG 275

A.1 ARQUITECTURA 275
A.1.1 Arquitectura inicial: ODMG-93 y ODMG 2.0 275
A.1.2 Arquitectura Actual : ODMG 3.0 276
A.2 CARACTERÍSTICAS DEL MODELO DE OBJETOS 276
A.2.1 Objetos 277
A.2.2 Literales 278
A.2.3 Tipos 278
A.2.4 Propiedades 280
A.2.5 Operaciones 280
A.2.6 Características de bases de datos 281
A.3 LENGUAJES DE ESPECIFICACIÓN DE OBJETOS 282
A.3.1 Lenguaje de definición de objetos 283
A.3.2 Lenguaje de intercambio de objetos 284
A.4 LENGUAJE DE CONSULTA 285
A.5 BINDING PARA JAVA 286
A.5.1 Principios de diseño 286
A.5.2 Binding del lenguaje 286
A.5.3 Traducción del modelo de objetos ODMG a Java 287
A.5.4 Java ODL 287
A.5.5 Java OML 288
A.5.6 Java OQL 290
A.6 OTROS BINDINGS 290

APÉNDICE B OBJECTDRIVER Y PSE PRO 291

B.1 ELECCIÓN DE LOS GESTORES PARA LA EVALUACIÓN 291
B.1.1 ODMG y PSE Pro 291
B.1.2 ODMG y ObjectDRIVER 292
B.2 INCORPORACIÓN DE LOS GESTORES EN LA HERRAMIENTA 292
B.2.1 Generación de código para ObjectDRIVER 292
B.2.2 Generación de código para PSE Pro 292
B.3 GENERADOR DE ESQUEMAS PARA OBJECT DRIVER 293
B.3.1 Diagrama de clases 294
B.3.2 Utilización 298

APÉNDICE C GRAMÁTICAS ODL - OML Y OQL PARA JAVA 301

C.1 GRAMÁTICA ODL + OML 301
C.2 GRAMÁTICA OQL 306

GLOSARIO DE TRADUCCIONES 309

REFERENCIAS 311

Capítulo 1 Objetivos y organización de esta tesis doctoral

1.1 Objetivos

El objetivo principal de esta tesis es la descripción de un SGBDOO (Sistema de Gestión de Bases de Datos Orientadas a Objetos) flexible construido sobre un Sistema Integral Orientado a Objetos (SIOO) basado en una máquina abstracta persistente, lo que garantizará su portabilidad. La flexibilidad del SGBD le convertirá en una buena plataforma de experimentación que permitirá conseguir un funcionamiento eficiente del sistema permitiendo su adaptación para diferentes patrones de aplicación.

En concreto en esta tesis se describen las características que debe tener este sistema, así como la estructura de una arquitectura software que lo soporta. Se justifican ciertas decisiones de diseño profundizando sobre todo en un elemento del sistema: *el mecanismo de indexación*.

Los pilares básicos que se pretende que estén presentes en el diseño de este sistema de gestión de bases de datos OO son cuatro:

- **Uniformidad.** El único elemento conceptual a emplear por el sistema serán los objetos**.** La implementación del sistema se realizará empleando un paradigma de orientación a objetos, y el sistema únicamente trabajará con objetos.
- **Flexibilidad.** El sistema deberá ser flexible permitiendo añadir o reemplazar las técnicas que rigen los mecanismos de forma sencilla. Esta característica es clave en el sistema cara a convertirse en una buena plataforma de experimentación.
- **Integración con el sistema integral.** El constructor del SGBDOO aprovechará algunas de las abstracciones proporcionadas por el sistema operativo, como por ejemplo la persistencia, sin necesidad de tener que implementarlas de nuevo.
- **Interoperabilidad.** El diseño facilitará la interoperabilidad, por ejemplo, para diferentes lenguajes de manipulación de la base de datos.

Además de estos objetivos, como objetivo final se busca la evaluación del modelo desarrollado mediante la implantación y valoración de los resultados obtenidos.

1.2 El proyecto de investigación en el que está enmarcada

Esta tesis se desarrolla dentro del Proyecto de Investigación Oviedo3, a cargo del Grupo de Investigación de Tecnologías Orientadas a Objetos, en el Departamento de Informática de la Universidad de Oviedo [LOO01]. En él se trata de diseñar un entorno de computación que utilice las tecnologías orientadas a objetos en toda su extensión, eliminando las desadaptaciones de impedancias producidas por los habituales cambios de paradigma.

1.3 Fases de desarrollo

En este apartado se describen las fases de trabajo en las que se divide el desarrollo de la tesis. En él no se hace mención al contenido desarrollado en cada fase, sino a los aspectos funcionales de cada una.

1. En primer lugar, se trata de estudiar qué SGBDOO existen actualmente tanto comerciales como de investigación, y como abordan éstos el problema de la interoperabilidad y la flexibilidad, prestando especial atención a los mecanismos de indexación. En este estudio se conseguirá un conocimiento profundo de los modos y maneras de abordar dichas características. Asimismo, la visión de conjunto permitirá realizar comparativas que lleven a la extracción explícita de críticas respecto a los sistemas existentes, y de los fallos existentes, se podrán extraer conclusiones que resulten de utilidad en nuevos planteamientos.
2. En segundo lugar, se realiza un análisis de los requisitos deseables para el diseño de un Sistema de Gestión de Bases de Datos Orientadas a Objetos construido sobre un sistema integral OO. Los aspectos generales de un sistema de gestión de base de datos y los aspectos relacionados con la arquitectura del sistema integral tendrán mucho que ver con la elección de los requisitos de partida.
3. Como consecuencia de los requisitos impuestos y los conocimientos adquiridos en la primera fase se elabora el núcleo, centrado en el mecanismo de indexación, para un SGBDOO construido sobre el sistema integral. Esta fase se concibe como resultado del proceso de crítica, reflexión y selección de posibilidades iniciado tras la primera fase. A ello hay que añadir un proceso de síntesis y consecución de un nuevo sistema de gestión de bases de datos OO que se ajusta mejor a los requisitos de partida y que produce mayores ventajas que los existentes. Finalmente, dentro de esta fase, se formalizará este sistema, analizando sistemáticamente todos los aspectos que intervienen en él.
4. Una vez formalizado el sistema, se procede a la descripción de sus virtudes, que dan crédito a la bondad de éste y mayor adecuación respecto a otros ya existentes. Además de ello, se apoya la demostración en una implantación real sobre un sistema integral, el sistema Oviedo3, y con la máquina abstracta Carbayonia, substrato del mismo.

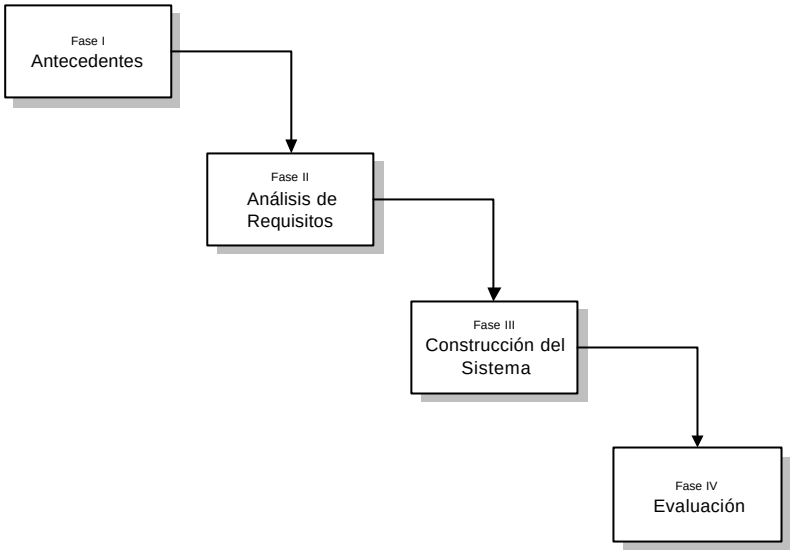


Figura1.1. Fases de desarrollo

1.4 Organización de los contenidos

La memoria de esta tesis se ha organizado en cinco secciones: antecedentes, aportaciones del sistema integral a la construcción del SGBDOO, construcción del sistema, desarrollo del prototipo y conclusiones.

1.4.1 Antecedentes

Esta sección comprende los capítulos 2, 3, 4 y 5. En estos capítulos se identifica la problemática existente con los gestores actuales, así como los requisitos exigibles para conseguir un SGBDOO ideal. Se analizan los sistemas existentes con el fin de extraer características interesantes con relación a los requisitos antes planteados.

El capítulo 2, *Necesidad de un SGBDOO unificado con un sistema integral*, plantea las diferentes posibilidades existentes para la gestión de objetos: las basadas en el modelo relacional y las basadas en el modelo de objetos. Tras analizarlas, se concluye que las más apropiadas son aquellas basadas en el modelo de objetos, pero las soluciones existentes no están exentas de problemas. Tras exponer los inconvenientes derivados de esta situación se justifica la necesidad de un SGBDOO unificado con un sistema integral orientado a objetos.

El capítulo 3, *Conceptos básicos sobre SGBDOO*, supone una introducción a los sistemas de gestión de bases de datos orientadas a objetos. En él se define un SGBD orientado a objetos, y se describen brevemente todos los aspectos a considerar a la hora de desarrollar uno de dichos sistemas: desde la funcionalidad de base de datos que deben proporcionar, hasta las diferentes posibilidades para su arquitectura.

En el capítulo 4, *Requisitos de un SGBDOO*, se enuncian explícitamente aquellos requisitos que se consideran básicos para la consecución de un sistema de gestión de objetos ideal. Los requisitos planteados en este capítulo afectan principalmente al diseño y a la construcción de dicho SGBDOO, y su consecución es, en definitiva, el objetivo de esta tesis doctoral.

En el capítulo 5, *Panorama de algunos sistemas para el almacenamiento de objetos*, se hace una exhaustiva revisión, con relación a los objetivos enunciados en el capítulo

anterior, de algunos sistemas de almacenamiento de objetos, con el fin de localizar características interesantes de los mismos. Los sistemas seleccionados para la revisión son representativos de las diferentes filosofías existentes para la gestión de objetos: gestores objeto-relacionales, *wrappers*, gestores de objetos, y sistemas de gestión de bases de datos orientadas a objetos.

1.4.2 Necesidad y aportaciones de un sistema integral para la construcción del SGBDOO

Esta sección comprende los capítulos 6 al 9, y en ella se pretende justificar la necesidad de un sistema integral orientado a objetos para la construcción de un SGBDOO en base a una serie de aportaciones del SIOO (sistema integral orientado a objetos). Se describen también las principales características de dicho SIOO: arquitectura y subsistemas de persistencia, protección, concurrencia y distribución.

El capítulo 6, *Necesidad de un sistema integral orientado a objetos para la construcción del SGBDOO*, pretende justificar la necesidad de un sistema integral orientado a objetos para solucionar problemas comunes en los sistemas existentes como son la falta de interoperabilidad entre modelos de objetos y la desadaptación de impedancias. Se enuncian aquí las características básicas que debe proporcionar este sistema integral, así como las aportaciones que éste puede proporcionar al desarrollo del SGBDOO.

El capítulo 7, *Arquitectura del sistema integral orientado a objetos*, describe una arquitectura para un SIOO que proporciona las características básicas enunciadas en el capítulo anterior. Dicha arquitectura está basada en una máquina abstracta reflectiva orientada a objetos, que es el substrato del sistema, sobre la que se construye el sistema operativo orientado a objetos. Se muestran también en este capítulo las alternativas disponibles para extender la funcionalidad del sistema.

El capítulo 8, *Modelo de persistencia del sistema integral orientado a objetos*, describe el modelo de persistencia que proporciona el sistema integral. Dicho modelo considera la persistencia como una propiedad fundamental del propio sistema haciendo que esté siempre disponible. El modelo además opta por una persistencia completa que conserva los datos y la computación.

El capítulo 9, *Otras aportaciones del sistema integral orientado a objetos*, describe otras características proporcionadas por el sistema integral, como son la distribución, la protección y la concurrencia. El mecanismo de protección proporcionado por la máquina abstracta permite disponer de capacidades con un número de permisos variable y proporciona la base para la implantación de diferentes modelos de autorización en el sistema. El sistema de distribución planteado permite la migración de objetos, así como la invocación de métodos remota. Y por último, el subsistema de concurrencia del SIOO es basado en un servicio de transacciones.

1.4.3 Solución

La sección dedicada a la solución es la más amplia de esta Tesis, y comprende desde el capítulo 10 al 14. En ella se propone la arquitectura del SGBDOO a construir sobre el sistema integral orientado a objetos, profundizando en la consecución de un mecanismo de indexación extensible que permita adaptar fácilmente el sistema a los diferentes patrones de aplicación con los que se puede encontrar. Esto le permite constituirse en una buena plataforma de *benchmarking*.

El capítulo 10, *Arquitectura del SGBDOO*, describe la arquitectura de referencia para un SGBDOO construido sobre un SIOO, comentando brevemente cada uno de los módulos de dicha arquitectura: motor, lenguajes y herramientas visuales. Con posterioridad, se analiza más en profundidad la arquitectura del motor del sistema, así como las unidades conceptuales en las que es estructurado, y sus propiedades fundamentales.

El capítulo 11, *Panorama de las técnicas de indexación en bases de datos orientadas a objetos*, representa una exhaustiva clasificación de las diferentes técnicas de indexación existentes para bases de datos orientadas a objetos, atendiendo a las características impuestas por el modelo de objetos en el lenguaje de consulta. Para cada una de esas técnicas se analiza su estructura y su funcionamiento, así como el tipo de operaciones para las que resultan más adecuadas, con el fin de seleccionar las más apropiadas para el SGBDOO que nos ocupa.

En el capítulo 12, *Mecanismo de indexación*, se describen detalladamente los objetivos a conseguir con el mecanismo de indexación para el sistema objeto de esta tesis, del que se destaca especialmente la extensibilidad, y se especifican las características que debe proporcionar dicho mecanismo para conseguir tales objetivos. Se justifican también las principales decisiones de diseño a tomar para la construcción del mismo.

En el capítulo 13, *Elección del lenguaje para el SGBDOO*, se describen las principales características que debe tener un lenguaje para una base de datos orientada a objetos, y se comentan las posibles opciones a la hora de seleccionar un lenguaje para el SGBDOO objeto de esta tesis. De entre esas posibilidades, finalmente se seleccionan los lenguajes propuestos por el estándar ODMG: *ODL*, *OML* y *OQL*, ligados en este caso al lenguaje de programación Java, y se justifica dicha elección.

En el capítulo 14, *Requisitos de la interfaz visual para el SGBDOO*, se describen las características deseables para la interfaz del SGBDOO, así como la funcionalidad básica que dicha interfaz debe proporcionar.

1.4.4 Prototipo

La sección dedicada al prototipo comprende los capítulos 15, 16, 17 y 18, y trata de mostrar una implementación real de lo descrito. Además, se ofrecen unos resultados de la comparación del sistema aquí propuesto con otros existentes en el mercado.

El capítulo 15, *Sistema integral orientado a objetos Oviedo3*, describe el sistema integral orientado a objetos Oviedo3 que se constituye en un objetivo de investigación de gran envergadura que incluye la presente Tesis y otra serie de ellas, dedicadas a diferentes aspectos del mismo como la persistencia, la seguridad, los lenguajes de programación, etc. Se procede en este capítulo a una descripción de la arquitectura básica del mismo, que será tomada como base para la construcción del SGBDOO.

El capítulo 16, *Diseño e implementación de un prototipo del motor para el SGBDOO*, describe en términos generales el diseño e implementación de un prototipo inicial del motor atendiendo a las características especificadas en la sección anterior. Este prototipo inicial es realizado sobre una versión básica de la máquina abstracta que únicamente proporciona persistencia.

El capítulo 17, *Introducción de los lenguajes en el SGBDOO*, describe algunas puntualizaciones realizadas sobre el *binding* Java de ODMG a emplear por el SGBDOO. Describe también una herramienta que genera código para diferentes

motores que sean compatibles con el *binding Java*, lo que facilitará por tanto la evaluación de diferentes productos.

En el capítulo 18, *Evaluación del prototipo*, se realiza una evaluación del prototipo construido, en dos direcciones. Por un lado, se evalúa su comportamiento con relación a las técnicas de indexación con las que ha sido dotado inicialmente el mecanismo de indexación diseñado. Y por otro lado, se evalúa el prototipo comparándolo con gestores comerciales como son PSE PRO para Java (orientado a objetos), y ObjectDRIVER (objeto-relacional). Dichas comparaciones permiten la extracción de ciertas conclusiones que en general no son desalentadoras.

1.4.5 Conclusiones

La última sección de esta tesis, dedicada a las conclusiones, comprende los capítulos 19 y 20.

El capítulo 19, *Trabajo Relacionado*, no trata exactamente de las conclusiones, sino de los trabajos contemporáneos al descrito en este documento, y que están relacionados con los temas tratados en él de una manera directa.

En el capítulo 20, *Conclusiones*, se resumen los resultados más destacables obtenidos como consecuencia de la elaboración de este trabajo, así como las principales ventajas aportadas por el sistema diseñado. Finalmente, se perfilan algunas líneas de investigación futuras que marcan el camino a seguir en la investigación empezada con esta tesis.

Capítulo 2 Necesidad de un SGBDOO unificado con un sistema integral

En la actualidad la adopción del paradigma de orientación a objetos es una realidad consolidada y supone además, la base de la mayoría de los sistemas con los que se está trabajando en este momento. Sin embargo, este paradigma no es adoptado de una forma uniforme por todos los componentes del sistema lo que se traduce en un problema de desadaptación de impedancias e interoperabilidad entre modelos de objetos. Por supuesto, para la gestión de objetos existen muchas posibilidades que no se escapan tampoco de los inconvenientes antes mencionados. En este capítulo se revisarán algunas de ellas, y se justificará la necesidad de un sistema de gestión unificado en un sistema integral orientado a objetos.

2.1 Posibilidades para la gestión de objetos

Los sistemas de gestión de bases de datos fueron concebidos como una alternativa a los sistemas de ficheros tradicionales para el almacenamiento y recuperación de la información [SKS97]. Con la llegada de la orientación a objetos la información debe seguir siendo almacenada y recuperada y para ello se plantearon diferentes alternativas:

1. **Basadas en el modelo relacional**
   - Bases de Datos Objeto-Relacionales
   - Wrappers o Envoltorios Objeto-Relacionales
2. **Basadas en el modelo de objetos**
   - Lenguajes persistentes
   - Gestores de Objetos
   - Bases de Datos Orientadas a Objetos

2.2 Gestión de objetos basada en el modelo relacional

La bases de datos relacionales tienen y han tenido una gran presencia y relevancia en el mercado, por eso es lógico que ante el surgimiento del paradigma de orientación a objetos, éstas intentasen dar soluciones para la gestión de los objetos.

2.2.1 Bases de datos objeto-relacionales

El origen de estos sistemas fue pronosticado en el Manifiesto del Sistema de Bases de Datos de Tercera Generación [SRL+90], y su esencia está centrada en la extensión del modelo relacional [SM96], de hecho antes eran denominados *sistemas de gestión de bases de datos relacionales extendidos.* Realmente, existen muchas formas de extender un modelo relacional, sin embargo, la elegida por los vendedores de bases de datos objeto-relacionales está muy centrada en las relaciones, de forma que una base de datos todavía consta de un conjunto de tablas, y las características de orientación a objetos

que soportan son proporcionadas como una extensión al núcleo del modelo relacional (tablas multi-filas, referencias entre filas, herencia entre tablas, etc.).

Ampliación artificial del modelo de datos relacional

El modelo relacional es generalmente extendido para soportar algunas características del modelo de objetos como por ejemplo, tipos y funciones definidas por el usuario, pero no todos los SGBD (sistema de gestión de bases de datos) objeto-relacionales soportan conceptos básicos de orientación a objetos como la herencia (aunque sí es contemplada en el estándar SQL:1999), y el polimorfismo. En definitiva, estos sistemas se basan en un modelo relacional al que se le imponen artificialmente algunas características del modelo de objetos.

Falta de uniformidad en los mecanismos para la extensión del modelo relacional

Cada SGBD objeto-relacional emplea su propio mecanismo para conseguir ampliar el modelo relacional, lo que hace que las diferencias entre ellos sean muy significativas [Ren97]. La aparición del estándar SQL:1999 [EM99] intenta aliviar estas diferencias aunque las características de orientación a objetos que incorpora son una mínima parte del estándar. En cualquier caso, en todos estos sistemas la tabla permanece siempre como elemento fundamental, lo que hace que el tratamiento de objetos desde estos sistemas no sea en absoluto transparente para el usuario.

Falta de claridad conceptual

Los sistemas objeto-relacionales son populares porque pueden beneficiarse mucho de lo que se ha aprendido acerca de los sistemas relacionales, y lo que es más importante, permiten la migración de usuarios desde productos de bases de datos relacionales existentes a sistemas que empleen esta aproximación. Sin embargo, lo expuesto anteriormente convierte a estos sistemas en inmensos y complejos, que deben su existencia a la prioridad que se da a las ganancias comerciales sobre la claridad conceptual o los principios de ingeniería del software.

Desadaptación de impedancias entre los lenguajes

Estos sistemas generalmente proporcionan al programador dos entornos separados: *el lenguaje de programación de la aplicación* y el *lenguaje de consulta de base de datos extendido*. Cada lenguaje puede invocar al otro, pero los dos tienen diferentes sistemas de tipos y entornos de ejecución. Esto plantea un problema de desadaptación de impedancias, ya que aunque normalmente ambos lenguajes son computacionalmente completos (contienen construcciones de control procedurales), los enunciados del lenguaje de consulta deben insertarse dentro del lenguaje de programación con el fin de transferir datos entre ambos entornos.

2.2.2 Mediadores o envoltorios objeto-relacionales

Los mediadores objeto-relacionales, envoltorios o *wrappers* [Dev97] son productos (Persistence [AK95], ObjectDriver [Obj00b], JavaBlend [Sun00], etc.) que se basan en poner capas de software de orientación a objetos sobre una base de datos relacional. Estos sistemas a nivel de programación proporcionan la sensación de estar trabajando con un SGBDOO (Sistema de Gestión de Bases de Datos Orientadas a Objetos), sin embargo, internamente, las clases son traducidas en tablas relacionales, y los objetos son traducidos en tuplas. El envoltorio generalmente emplea SQL para interactuar con la base de datos relacional.

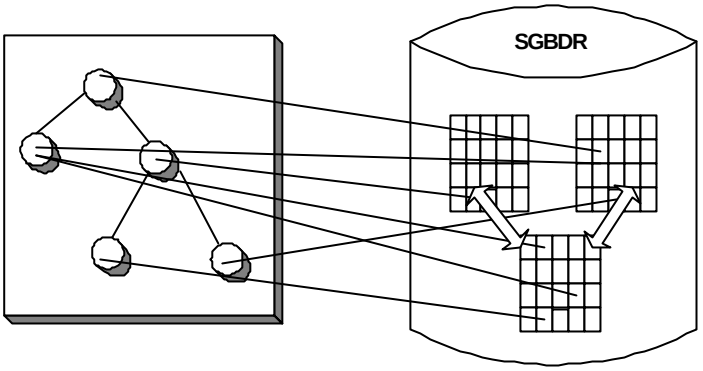


Figura 2.1. Funcionamiento de un *wrapper*

Descomposición del esquema conceptual orientado a objetos en tablas

Los *wrappers* proporcionan al programador la ilusión de estar trabajando con un SGBDOO puro. Esto incrementa la productividad al evitar que sea el propio programador el que tenga que hacer la conversión de clases y objetos a tablas, lo que se traduciría evidentemente en una gran cantidad de código (entre un 25% y un 40% más [Exc97]) y en una mayor probabilidad de fallos. Pero esto realmente no deja de ser una capa de software añadida que hay que atravesar cada vez que se necesita almacenar o recuperar un objeto, por el simple hecho de emplear un medio de almacenamiento cuyo modelo de datos no encaja en absoluto con el empleado en la aplicación.

Restricciones en el modelo de datos de la aplicación

Los *wrappers* al confiar en un SGBDR como medio de almacenamiento presentan ciertos problemas de rendimiento para la gestión de datos complejos, ya que hay ciertos tipos de datos que son muy difíciles de modelar en un SGBDR, con lo que los *wrappers* optan por imponer restricciones severas sobre el modelo de datos de la aplicación.

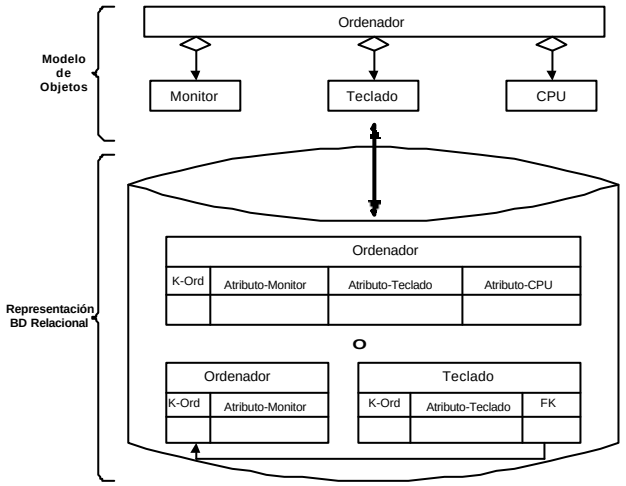


Figura 2.2. Traducción de clases agregadas en un *wrapper*

2.3 Gestión de objetos basada en el modelo orientado a objetos

Una perspectiva muy prometedora para los programadores ha sido desde siempre la idea de que el propio lenguaje de programación con el que están trabajando les proporcione la posibilidad de que sus datos persistan en el tiempo. Esta posibilidad es

proporcionada por los lenguajes persistentes. Dichos lenguajes pueden ser, por lo tanto, una posibilidad interesante para conseguir la persistencia de los objetos, dejando de lado otras prestaciones más ambiciosas que nos proporcionan los SGBDOO.

La otras posibilidades basadas en el modelo de objetos son los SGBDOO que ofrecen toda la funcionalidad de un SGBD pero ligado ya a un modelo de objetos, y los gestores de objetos que ofrecen algunas de las prestaciones de un SGBDOO, pero no todas.

2.3.1 Lenguajes persistentes orientados a objetos

Atkinson [ABC+83] propone la gestión de datos persistentes integrada a nivel de lenguaje, de forma que un lenguaje persistente se puede definir como un *lenguaje que proporciona la habilidad de preservar datos entre las sucesivas ejecuciones de un programa e incluso permite que tales datos sean empleados por programas muy diferentes* [Cla91]. Los datos en un lenguaje de programación persistente son independientes del programa y capaces de existir más allá de la ejecución y tiempo de vida del código que los creó.

Acercamiento de los lenguajes de programación a las bases de datos

Los lenguajes persistentes orientados a objetos parten del sistema de tipos y el modelo de programación de un lenguaje de programación orientado a objetos concreto, tal como Smalltalk, C++ o Java, y le añaden las características necesarias para hacer sus datos persistentes y sus ejecuciones de programa atómicas. Esto permite trabajar a los programadores directamente con los datos desde el lenguaje, proporcionando por tanto, un acercamiento entre los lenguajes de programación y las bases de datos.

Eliminación de la desadaptación de impedancias entre los lenguajes

En los lenguajes persistentes el lenguaje de consulta se halla totalmente integrado con el lenguaje anfitrión, y ambos comparten el mismo sistema de tipos, de forma que los objetos se pueden crear y guardar en la base de datos sin ningún tipo explícito ni cambio de formato. Además, con estos lenguajes los programadores pueden trabajar con datos persistentes sin tener que escribir de manera explícita código para buscarlos en la memoria o para volver a guardarlos a disco. Esta integración elimina la desadaptación de impedancias mencionada con anterioridad para los lenguajes de los SGBD objeto-relacionales.

Dificultad para conseguir interoperabilidad entre diferentes lenguajes de programación

Sin embargo, en los lenguajes persistentes se parte de un modelo de objetos único, el del lenguaje de programación orientado a objetos tomado como base, no facilitando así la interoperabilidad entre diferentes modelos de objetos y/o lenguajes de programación.

Los lenguajes persistentes a pesar de no haber tenido el éxito esperado, entre otras cosas porque hay poca gente y pocas compañías preparadas para aceptar un nuevo lenguaje de programación, es un área en la que se está trabajando [ADJ+96], y además, los resultados de las investigaciones realizadas sobre ellos en temas como modelos de persistencia, esquemas de *pointer swizzling*, y esquemas de recolección de basura para datos persistentes, han sido directamente transferidos a bases de datos orientadas a objetos.

2.3.2 Bases de datos orientadas a objetos

A la hora de desarrollar una aplicación en la que el modelo de objetos recoge bien su semántica, parece lógico emplear un método de gestión de la información en el que no sea necesario hacer una descomposición artificial de la misma. Para ello la posibilidad más lógica sería la de un SGBD que permitiese gestionar directamente los objetos.

Según Kim [Kim91] *una* **base de datos orientada a objetos** *es una colección de objetos en los que su estado, comportamiento y relaciones son definidas de acuerdo con un modelo de datos orientado a objetos, y un* **sistema de base de datos orientado a objetos** *es un sistema de base de datos que permite la definición y manipulación de una base de datos orientada a objetos.* En el Manifiesto del Sistema de Base de Datos Orientado al Objeto [ABD+89] (*The Object-Oriented Database System Manifesto*), se especificaron por primera vez las características que debía presentar un SGBDOO, sin embargo, durante los primeros años de existencia de estos sistemas no había en absoluto portabilidad ni interoperabilidad entre ellos. Con el fin de solucionar estos problemas surge el primer intento de estandarización [CAD+94] para los SGBDOO llevado a cabo por ODMG[1] (*Object Data Management Group*)

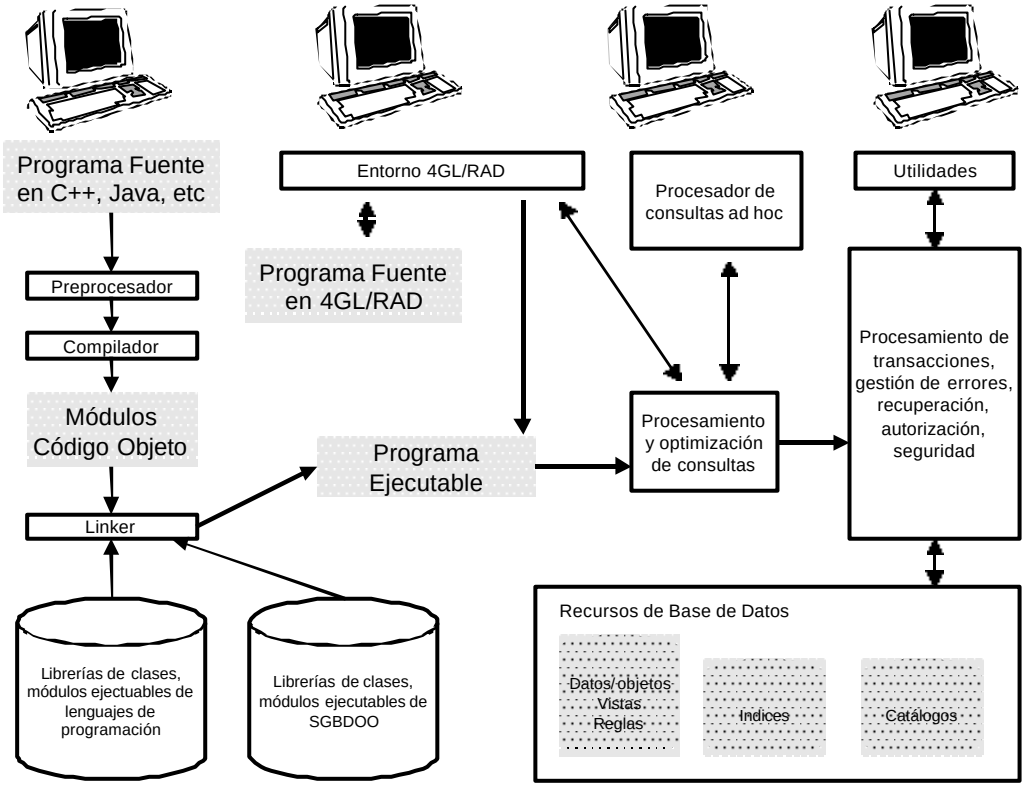


Figura 2.3. Arquitectura de un SGBDOO

Los SGBD orientados de objetos, o *lenguajes de programación de base de datos orientados a objetos* según Cattell [Cat94], combinan características de orientación a objetos y lenguajes de programación orientados a objetos con capacidades de bases de datos. Estos sistemas están basados en la arquitectura de un lenguaje de programación de bases de datos. Las aplicaciones son escritas en una extensión de un lenguaje de programación existente, y el lenguaje y su implementación (compilador, preprocesador, entorno de ejecución) han sido extendidos para incorporar funcionalidad de base de datos. El objetivo de estos sistemas es llegar a integrarse con múltiples lenguajes,

[1] Inicialmente Object Database Management Group

aunque esto puede suponer un problema debido a lo cercano de la asociación que se requiere entre el sistema de tipos de la base de datos y el sistema de tipos del lenguaje de programación.

Actualmente existen una gran cantidad de productos en el mercado etiquetados como SGBDOO (ObjectStore [Obj00a], POET [Poe00], Jasmine [CAI98], GemStone [Gem00], etc.), y aunque la mayoría de ellos se proclaman conformes con el estándar, sigue existiendo una falta de interoperabilidad entre sus modelos de objetos.

2.3.3 Gestores de objetos

Los gestores de objetos, también llamados *almacenes de objetos persistentes* proporcionan menos funcionalidad que los SGBDOO (podríamos decir que son sus hermanos pequeños). Están diseñados para gestionar el almacenamiento persistente de un número de objetos relativamente pequeño y con bajos requerimientos de concurrencia. Son por tanto, poco apropiados para muchos usuarios. Normalmente, ofrecen una distribución de datos limitada, un bloqueo de grano muy grueso, no proporcionan evolución de esquemas y carecen de un lenguaje de consulta o de un lenguaje de programación. Pueden asociarse con una extensión de un sistema de ficheros.

La mayoría de estos gestores vienen en forma de API que será utilizada desde el lenguaje de programación correspondiente (C++ o Java principalmente), y actualmente existe una gran cantidad de ellos, sobre todo para Java (PSE Pro [PSE99], Jeevan [Jee99], StreamStore [Str99], etc.).

2.4 Problemática con las bases de datos orientadas a objetos

De todas las opciones presentadas en los apartados anteriores para la gestión de objetos, conceptualmente las más adecuadas son aquellas basadas en el modelo de objetos, y que por tanto, soportan de una forma natural las características de dicho modelo sin necesidad de acudir a una ampliación artificial del modelo relacional o a una descomposición de las clases en tablas.

Dentro de las posibilidades basadas en el modelo de objetos la más completa es obviamente la proporcionada por los SGBDOO o los gestores de objetos[2], ya que los lenguajes persistentes nos proporcionan una gestión de objetos muy reducida, y ligada exclusivamente a un determinado lenguaje de programación. Sin embargo, a pesar de la adecuada filosofía de los SGBDOO, los sistemas existentes actualmente no están en absoluto exentos de problemas.

2.4.1 Código intruso para gestionar la base de datos

La mayoría de los gestores persistentes y SGBDOO vienen en forma de API (véase Figura 2.4), que será utilizada desde las aplicaciones que emplean un lenguaje de programación orientado a objetos como C++ o Java. El principal inconveniente es que cada gestor suele emplear su propia API (véase Código 2.1), lo que va a dificultar la legibilidad y el mantenimiento del código, va a restringir la flexibilidad y la portabilidad de la aplicación, y desde el punto de vista del usuario, va a incrementar la complejidad

[2] Suponemos la misma filosofía conceptual en los SGBDOO que en los gestores de objetos, pero debido a la mayor completitud proporcionada por los primeros en el resto del documento se hará referencia únicamente a éstos

de su trabajo, ya que dicha API variará en función de la herramienta o lenguaje empleado.

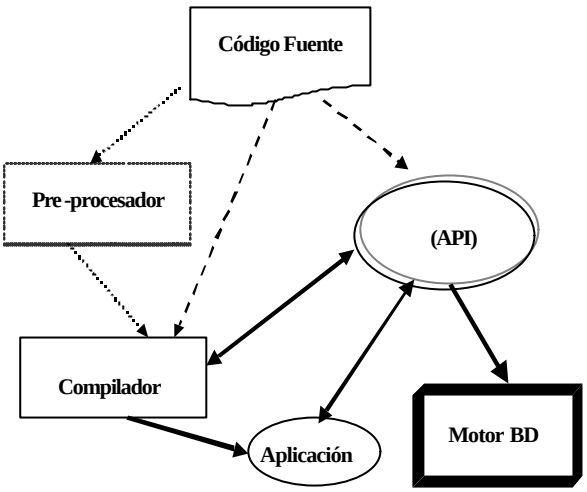


Figura 2.4. Inclusión de funcionalidades de bases de datos en un lenguaje de programación

Como estos problemas eran evidentes, y además una de las principales lacras con las que se encontraban los SGBDOO, ODMG [ODM99] intenta proporcionar un API estándar para que proporcione la funcionalidad de base de datos en un lenguaje de programación. El principal problema es que, aunque, inicialmente la mayoría de los vendedores de BDOO se comprometieron a adoptar dicho estándar, hoy en día cada uno emplea el API a conveniencia.

```
void main()

{

OS_PSE_ESTABLISH_FAULT_HANDLER //activar tratamiento de errores

objectstore::initialize();  //Inicializar la rutina PSE
os_database *db=os_database::open("DocDB");  //Abrir la base de datos
Documento * a_document=(Documento)(db->find_root("doc_root")->get_value()); // punto de
entrada
Persona * a_person=a_document->autor;  //Navegar por autor
cout<<"Documento Id: "<<a_document->doc_id<<endl; // Imprimir valores
...
db->close();  //Cerrar la bd

OS_PSE_END_FAULT_HANDLER  //Cerrar tratamiento de errores

}
```

Código 2.1. Ejemplo de código para gestionar la base de datos en PSE PRO C++

2.4.2 Carencia de interoperabilidad

Otro de los objetivos ideales perseguidos por los SGBDOO es la integración con múltiples lenguajes de programación, de forma que objetos que hayan sido creados en una base de datos que emplea como lenguaje de programación C++ puedan ser recuperados desde otra base de datos con otro lenguaje de programación orientado a objetos diferente (ej. Java). Es decir, cualquier base de datos ha de poder recuperar los objetos almacenados independientemente del lenguaje de programación y de la base de datos con la que hayan sido creados.

El estándar ODMG también intenta dar solución a este problema. Propone, por un lado, un lenguaje de definición de objetos (ODL) basado en el IDL de CORBA, que facilita la portabilidad de esquemas entre SGBD, y siempre que los sistemas sean compatibles con el estándar (condición ésta última bastante difícil de conseguir en su totalidad). Además, las ligaduras del ODL a lenguajes como C++, Smalltalk y Java están diseñadas para ser introducidas fácilmente en la sintaxis declarativa de su lenguaje de programación, sin embargo, debido a las diferencias inherentes en los modelos de objetos nativos a estos lenguajes de programación, no siempre es posible mantener una semántica consistente entre lenguajes de programación (versiones específicas de ODL).

Por otro lado, ODMG, propone un formato para el intercambio de objetos (*OIF*), que permite el intercambio de objetos entre bases de datos (véase Apéndice A). Este lenguaje se emplea para volcar y cargar el estado actual de una base de datos de objetos desde un fichero o conjunto de ficheros. Sin embargo, esto implica que sería necesario un proceso independiente para la importación y exportación de los datos.

Algunos sistemas como POET [Poe00], por ejemplo, soportan cierta interoperabilidad entre los *bindings* de ODMG para Java y C++ pero siempre dentro del propio producto e imponiendo ciertas restricciones. Así, el *binding* de C++ para conseguir interoperabilidad con el de Java no debe soportar, por ejemplo, herencia múltiple.

2.4.3 Difícil portabilidad

La mayoría de los SGBDOO existentes son construidos para una determinada plataforma, lo que dificulta la migración tanto del propio gestor como de los datos hacia otras plataformas. Esta es una característica especialmente relevante teniendo en cuenta la importancia de conseguir productos multiplataforma en la actualidad.

2.4.4 Extensibilidad limitada

La extensibilidad es un problema que empieza a preocupar a los investigadores de bases de datos a mediados de los 80, ya que los SGBD surgen para dar cabida a un amplio espectro de aplicaciones con requerimientos muy diferentes, sin embargo, hay muy poca probabilidad de que un mismo SGBD sea válido y eficiente para satisfacer dichos requerimientos. Por ello, sería ideal disponer de un SGBDOO que pudiese adaptarse al tipo de datos a gestionar, seleccionando por ejemplo las técnicas de indexación adecuadas. Esto exigiría un sistema flexible que permitiese la extensibilidad y adaptabilidad del sistema.

Aunque son muchos los proyectos (EXODUS [CDF+86], DASDBS [PSS+87], PostGRES [SR86], etc.) que se planteaban estos objetivos, los resultados finales son muy diferentes. Dichas aproximaciones van por ejemplo desde POSTGRES que es un SGBD completo, que pretende hacer el menor número de cambios posibles al modelo relacional, y en el que la extensibilidad viene proporcionada por la incorporación de nuevos tipos datos, nuevos operadores y métodos de acceso, y un mecanismo de recuperación simplificado, hasta EXODUS, que no es un SGBD, si no un generador de bases de datos basado en un núcleo básico más unas herramientas que facilitan la generación semi-automática de SGBD para nuevas aplicaciones. Sin embargo, la forma de extender el sistema tiene ciertas limitaciones, obligando, por ejemplo, a conocer los nuevos tipos a soportar por el sistema antes de generar el nuevo SGBD.

Lo ideal sería contar con un SGBDOO que soporte cualquier tipo de datos y con unas determinadas características por defecto (lenguaje de programación, técnica de

indexación, etc.), pero fácilmente modificables, lo que facilitaría la extensibilidad del sistema.

2.5 Problemática con los sistemas operativos

Los sistemas de almacenamiento de objetos, y en concreto, los SGBD necesitan un sistema operativo subyacente. La construcción de SGBD sobre los sistemas operativos convencionales siempre ha ocasionado problemas sobradamente reconocidos por la comunidad de bases de datos [ÖDV94]. La base de estos problemas está en que un sistema operativo (SO) soporta un sencillo conjunto de abstracciones con una implementación simple para cada una de ellas, de forma que cuando se requiere una implementación diferente para una de ellas (ej. sistema de ficheros), la implementación tiene que ser modificada (si hay acceso al código fuente) o bien el constructor del SGBD tiene que re-implementar la abstracción (o incluso el conjunto completo). Para la construcción de SGBDOO sobre SO convencionales estos problemas se acentúan, debido a los propios problemas que tienen los SO convencionales para adaptarse al paradigma de orientación a objetos.

2.5.1 Desadaptación de impedancias

Los sistemas operativos ofrecen abstracciones basadas en versiones evolucionadas de la máquina de Von Neumann que son más adecuadas para el paradigma procedimental que para el orientado a objetos, en el que las aplicaciones se estructuran como un conjunto de objetos que colaboran entre sí mediante la invocación de métodos.

2.5.1.1 Abstracciones inadecuadas de los sistemas operativos

Los objetos de los lenguajes en que se desarrollan las aplicaciones suelen ser de grano fino. Sin embargo, el elemento más pequeño que manejan los sistemas operativos es el de un proceso asociado a un espacio de direcciones, de un tamaño mucho mayor. Esto obliga a que sea el compilador del lenguaje el que estructure los objetos internamente dentro de un espacio de direcciones. Pero por ejemplo, al desarrollar aplicaciones distribuidas esto ya no funciona y el programador se ve obligado a abandonar el paradigma de orientación a objetos y a encajar de manera antinatural la invocación de métodos sobre un sistema operativo orientado a comunicar procesos. Para solucionar este problema se acaba recurriendo a la interposición de capas adicionales de software que adapten la gran diferencia existente entre el paradigma de orientación a objetos y los sistemas actuales. Un ejemplo de este software es COM (*Component Object Model*, Modelo de objetos de componentes) [Rog96] y *CORBA (Common Object Request Broker Architecture*, Arquitectura común de intermediarios entre objetos) [OMG97].

2.5.1.2 Desadaptación de interfaces

La interfaz de los sistemas operativos tradicionales está enfocada al paradigma procedimental, lo que hace que el programador deba utilizar dos paradigmas diferentes en el desarrollo de sistemas. Al igual que en el caso anterior se pueden utilizar capas de adaptación, por ejemplo encapsulando la interfaz del sistema operativo mediante una librería de clases. Así, las llamadas al sistema operativo se hacen indirectamente a través de objetos de la librería de clases de manera orientada a objetos. La librería, a su vez, se encarga de llamar al sistema operativo. Un ejemplo de estas capas de adaptación es la librería MFC (*Microsoft Foundation Classes*), que entre otras cosas, encapsula ciertos servicios del sistema Windows en un marco de aplicación.

2.6 Problemática con el modelo de objetos

Incluso aunque diferentes elementos del sistema usen el paradigma de la orientación a objetos, puede haber problemas de desadaptación que dificultan la interoperabilidad entre ellos. Es común la existencia de diferentes lenguajes de programación orientados a objetos, bases de datos orientadas a objetos, etc. Sin embargo, el modelo de objetos que utiliza cada uno de ellos suele ser diferente. Aunque sean orientados a objetos, las propiedades de los objetos de cada sistema pueden diferir, por ejemplo un modelo puede tener constructores y otros no, etc.

Para solucionar este problema, de nuevo se recurre a la introducción de capas de software de adaptación. CORBA es nuevamente un ejemplo de este tipo de software, ya que define un modelo de objetos propio y especifica únicamente la interfaz de un objeto y nunca la implementación. ODMG propone el empleo de un modelo de objetos común, pero obviamente sería una solución parcial únicamente aplicable para los SGBDOO, no para el resto de elementos del sistema.

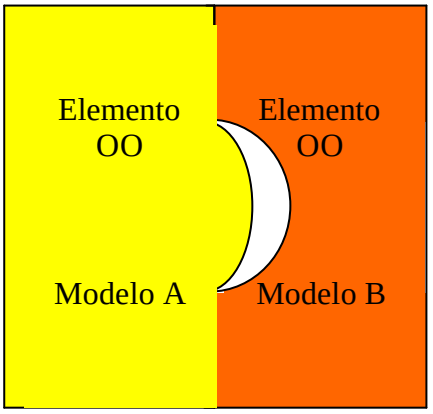


Figura 2.5. Desadaptación entre modelos de objetos diferentes

2.7 Inconvenientes derivados

En primer lugar, un SGBDOO parece la alternativa conceptual más correcta para la gestión de objetos, sin embargo, es muy difícil que un único SGBD se adapte eficientemente a las necesidades de todas las aplicaciones a las que va a dar servicio. En general los SGBDOO actuales adolecen de tres problemas principales

- **Falta de transparencia**. Las propuestas existentes para conseguir interoperabilidad de los datos almacenados entre diferentes gestores no son transparentes. Es necesario especificar, por un lado, el esquema en un lenguaje común determinado, y por otro es necesario un formato específico para el intercambio y un proceso encargado de la carga y descarga de los datos.
- **Falta de portabilidad**. Los sistemas existentes están generalmente ligados a una determinada plataforma, lo que dificulta la migración tanto de los gestores como de los propios datos a otras plataformas diferentes.
- **Dificultad para la adaptabilidad (extensibilidad) del sistema**. Los sistemas existentes son generalmente auténticas cajas negras, construidos en torno a mecanismos muy rígidos que no contemplan la incorporación de nuevas técnicas.

Por otro lado, los SGBDOO son construidos sobre sistemas operativos orientados a ficheros y procesos, abstracciones que distan mucho del concepto de objeto y método, lo que obliga a estos SGBD a tener que reimplementar muchas de dichas abstracciones o al menos a tener que introducir capas de adaptación entre los mismos. Es decir, el hecho de no adoptar el paradigma de OO de una forma integral en el sistema genera una serie de inconvenientes que afectan por supuesto a la construcción del SGBDOO:

- **Disminución del rendimiento global del sistema**, ocasionada por la necesidad de atravesar las capas de software para adaptar los paradigmas.
- **Aumento de la complejidad del sistema**. La adición de capas introduce más complejidad en los sistemas. Por un lado aumentan los problemas de integración entre las diferentes capas, errores en la misma, posibilidades de fallos, etc. Por otro lado, los sistemas se hacen más difíciles de comprender al intervenir tantos elementos y tan dispares.
- **Dificultad para la extensibilidad del sistema.** Los sistemas construidos sobre sistemas convencionales son bastante rígidos al no aprovechar el paradigma de orientación a objetos para permitir la adaptación de las abstracciones ofrecidas por el propio sistema operativo.
- **Falta de uniformidad y transparencia**. En la práctica, estas soluciones basadas en la superposición de capas no son todo lo transparentes que deberían ser cara al usuario. Por ejemplo, la escritura de objetos que vayan a funcionar como servidores suele realizarse de manera diferente del resto de los objetos, como en el caso de CORBA.
- **Pérdida de portabilidad y flexibilidad**. La falta de uniformidad produce una pérdida de portabilidad y flexibilidad. El programador se ve forzado a utilizar ciertas convenciones, formas especiales de programar, etc. que impone el uso de la propia capa que reduce la portabilidad de los objetos.

2.8 SGBDOO unificado con el sistema integral

Una manera de resolver estos problemas es crear un sistema homogéneo en el que se utilice en todos sus elementos el mismo paradigma de la orientación a objetos y se dé soporte nativo a los mismos: un sistema integral orientado a objetos.

Como parte integrante del entorno de computación de este sistema se encuentra un sistema gestor de objetos que aprovechará algunas características que proporciona el sistema operativo para su construcción. Este gestor será construido con el paradigma de orientación a objetos del propio sistema integral, con lo que no existe desadaptación, y al dar soporte directo a objetos, es decir, al gestionar directamente tanto la representación de los objetos como su utilización se soluciona el problema de la interoperabilidad. Los objetos, al ser gestionados por el propio sistema pueden “verse” unos a otros independientemente del gestor con el que hayan sido creados.

El sistema integral facilitará la extensibilidad del SGBD basándose en las propias características ofrecidas por el paradigma de orientación a objetos empleado, y en la reflectividad proporcionada por el sistema operativo.

Capítulo 3 Conceptos básicos sobre SGBDOO

En este capítulo se describen las características más comunes exigibles a un sistema de gestión de bases de datos orientadas a objetos (SGBDOO), así como algunos de los aspectos de implementación más relevantes que deben tenerse en cuenta en la confección de dichos sistemas.

3.1 Qué es un sistema de gestión de bases de datos orientadas a objetos

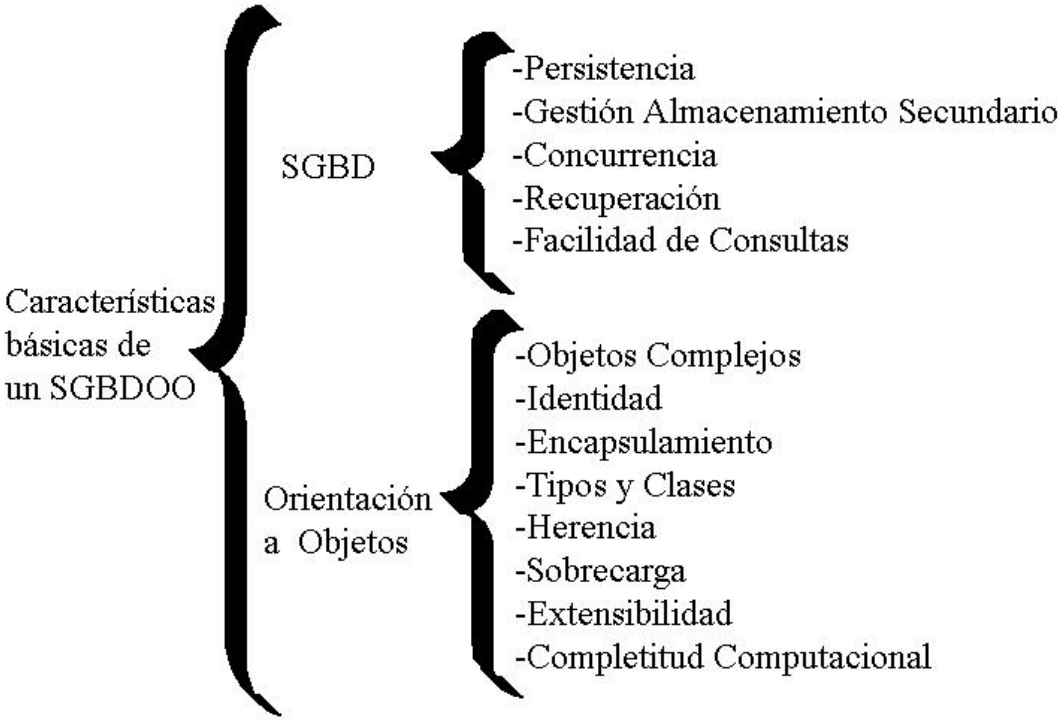
Un SGBDOO como se ha mencionado en el capítulo anterior es un sistema que combina características de orientación a objetos y lenguajes de programación OO con capacidades de bases de datos [Mar96]. Sin embargo, ésta es una definición vaga en la que quedan sin concretar las características que se consideran exigibles a un SGBDOO para considerarlo como tal. Los apartados siguientes tienen como objetivo evidenciar dichas características.

3.1.1 Manifiesto de bases de datos orientadas a objetos

La primera descripción realizada formalmente sobre los rasgos y características principales que debe tener un SGBDOO para poder calificarlo como tal fueron especificadas en el *Object-Oriented Database System Manifesto* [ABD+89]. En este manifiesto se clasifican las características en tres categorías: obligatorias, optativas y abiertas.

3.1.1.1 Obligatorias

Son aquellas características que se consideran esenciales en un SGBDOO. Un SGBDOO debe satisfacer dos criterios: debe *ser un SGBD,* y debe *ser un sistema orientado al objeto*; es decir, en la medida de lo posible deberá ser consistente con el actual conjunto de lenguajes de programación orientados a objetos. El primer criterio se traduce en cinco características: persistencia, gestión del almacenamiento secundario, concurrencia, recuperación y facilidad de consultas. El segundo criterio se traduce en ocho características: objetos complejos, identidad de objetos, encapsulamiento, tipos y clases, herencia, sobrecarga combinada con ligadura tardía, extensibilidad y completitud computacional.



3.1.1.2 Optativas

Son características que mejoran el sistema claramente pero que no son obligatorias para hacer de él un sistema de bases de datos orientado a objetos. Algunas de estas características son de naturaleza orientada a objetos (ej. herencia múltiple) y se incluyen en esta categoría porque, aunque hacen que el sistema esté más orientado a objetos, no pertenecen a los requisitos primordiales. Otras características son simplemente características de la base de datos (ej. gestión de las transacciones de diseño) que mejoran la funcionalidad del sistema pero que no pertenecen a los requisitos primordiales.

Estas características son:

- Herencia múltiple
- Chequeo e inferencia de tipos
- Distribución
- Transacciones de Diseño
- Versiones

3.1.1.3 Abiertas

Estas características ofrecen grados de libertad para los desarrolladores del sistema de base de datos orientado a objetos. Difieren de las obligatorias en el sentido de que todavía no se ha llegado a consenso alguno respecto a ellas en la comunidad científica. Difieren también de las opcionales en que no se conoce cuáles de sus alternativas están más o menos orientadas al objeto.

Estas características son:

- Paradigma de programación
- Sistemas de representación
- Sistema de tipos
- Uniformidad

A pesar de las especificaciones realizadas en este manifiesto los SGBDOO presentaban muy pocas características en común (carencia de un modelo de objetos común) que facilitasen la portabilidad entre las aplicaciones. En 1991, Rick Cattell encabezó la formación del grupo ODMG (Object Data Management Group) comenzándose a trabajar en la elaboración de un estándar. Un primer borrador fue publicado en 1992, y la primera versión del mismo en 1993 [CAD+94]. La última versión (3.0) ha sido publicada en septiembre de 1999 [CBB+99].

Este estándar se construye sobre estándares existentes tanto de lenguajes de programación como de bases de datos, y trata de simplificar el almacenamiento de objetos, y garantizar la portabilidad de la aplicación. Una descripción más detallada sobre dicho estándar puede encontrarse en el Apéndice A.

3.2 Funcionalidad de base de datos

En un sistema de base de datos la persistencia de los datos es la característica más relevante, no obstante, existen otra serie de características también básicas que debe proporcionar, como son la gestión del almacenamiento secundario, las transacciones u otro mecanismo similar para el control de la concurrencia y la recuperación, la seguridad y la posibilidad de hacer consultas. Se debe considerar en este bloque también la capacidad para cambiar el esquema de la base de datos.

3.2.1 Persistencia

La persistencia se identifica con *la cualidad de algunos objetos de mantener su identidad y relaciones con otros objetos con independencia del sistema o proceso que los creó*. En la práctica, se implementa proveyendo a los objetos de un mecanismo cuyo objetivo básico consiste tanto en el almacenamiento de objetos existentes en la memoria como en la recuperación posterior a éste de los mismos.

3.2.1.1 Tipos de persistencia

Un aspecto importante de la persistencia son las modalidades bajo las que los objetos se hacen persistentes. Para ello existen diferentes posibilidades [Cat94]:

- **Implícita.** La creación del objeto automáticamente implica su persistencia. Esta modalidad es la más simple, y es típicamente utilizada en sistemas en los que las clases tienen también una función extensional. (ej. ORION).
- **Por clase.** Un objeto se hace persistente cuando se crea basándose en su clase (se distingue entre clases persistentes y transitorias). La clase debe identificarse como persistente en la declaración o bien ser una subclase de una clase persistente proporcionada por el sistema. (ej. Objectivity/DB y ONTOS).

- **Por llamada explícita.** El usuario puede especificar explícitamente la persistencia para un objeto cuando se crea. Algunos sistemas permiten especificarlo en cualquier momento. (ej. ObjectStore).
- **Por referencia o alcance.** La persistencia de los objetos es determinada por alcance desde ciertas *raíces* persistentes conocidas globalmente. Este tipo de persistencia es la más potente, pero suele implicar una sobrecarga significativa en el sistema.(ej. GemStone).

3.2.1.2 Eliminación de objetos persistentes

En cuanto a la eliminación de objetos persistentes existen principalmente dos modalidades [BM93]:

- **Borrado explícito.** Consiste en tener una operación explícita para borrar. Como esto puede crear problemas de integridad referencial (si se elimina un objeto al que otros estaban apuntando, las referencias de éstos se invalidan), se acude a soluciones como disponer de un *contador de referencias* para determinar si un objeto está siendo referenciado por otros objetos antes de borrarlo.
- **Borrado libre.** Un objeto persistente se cancelará cuando se eliminen todas las referencias y nombres externos asociados con él. De esta forma las referencias a objetos borrados causarán excepciones, y los identificadores de los objetos (IDOs) de los objetos borrados no podrán ser reutilizados. La operación de borrado es eficiente (se asegura la integridad de referencias), pero se requiere código adicional tanto en las aplicaciones como en los métodos para tratar las excepciones.

3.2.2 Concurrencia y recuperación

Los mecanismos de control de concurrencia y recuperación son una característica muy importante de un SGBD porque ayudan a mantener la integridad lógica y física respectivamente de la base de datos. Las técnicas tradicionales de concurrencia y recuperación tienen cabida también en los SGBDOO. Sin embargo, son necesarias nuevas técnicas para afrontar nuevos requerimientos. En el caso de las transacciones, por ejemplo, se necesitan nuevos modelos de transacciones y unos criterios de corrección más flexibles.

3.2.2.1 Bloqueo

En un SGBDOO existen diferentes tipos de bloqueo y diferentes granularidades:

- **a nivel de objetos:** individuales, complejos, todos los de una clase.
- **a nivel de unidades físicas**: páginas o segmentos

Los protocolos basados en bloqueo son el mecanismo más común empleado por los SGBDOO para evitar conflictos. Sin embargo, no sería aceptable para un usuario que ha iniciado una transacción de larga duración descubrir que la transacción ha sido abortada debido a conflictos de bloqueo y se ha perdido el trabajo. Se proponen para ello otras soluciones: transacciones evolucionadas y versiones.

3.2.2.2 Transacciones

Una transacción es *una unidad lógica de trabajo que siempre debería transformar la base de datos de un estado consistente a otro*. Una transacción debe cumplir cuatro

propiedades (ACID) [GUW00]: *Atomicidad, Consistencia, aislamiento (Isolation) y Durabilidad.*

En las aplicaciones tradicionales las transacciones eran generalmente de corta duración, mientras que en las aplicaciones que intervienen objetos complejos las transacciones pueden durar horas e incluso días (*transacciones de larga duración*). Mantener la atomicidad y la durabilidad en este tipo de transacciones es complicado, y los esquemas para la gestión de transacciones simples no solucionan estos problemas. En [Ozs94] se exponen unos modelos de transacciones más avanzados como: *transacciones anidadas abiertas* y *cerradas*, y *modelos de workflow*.

En realidad, la mayoría de los sistemas comerciales implementan simplemente el bloqueo a nivel de página y sin permitir siquiera las transacciones anidadas.

3.2.2.3 Versiones

Las versiones suponen una alternativa al control de concurrencia mediante transacciones anidadas y conversacionales [Cat94]. El empleo de múltiples versiones de objetos permite la coordinación entre múltiples usuarios. Este mecanismo ha sido empleado en muchos prototipos y productos, pero con una semántica muy diferente entre ellos.

La funcionalidad básica requerida para un SGBDOO en relación con las versiones implica

- **Creación de objetos versión.** Un programa o un usuario puede crear nuevas versiones explícitamente o bajo ciertas circunstancias.
- **Borrado de una versión.** Es necesario poder destruir o archivar viejas versiones de objetos cuando no se necesiten.

Un SGBDOO que permita la gestión de versiones debe controlar que las referencias a un objeto siempre apunten a la versión correcta de ese objeto. Para ello surge el concepto de *configuración.*

Configuraciones

Una configuración (Figura 3.1) es una *colección de versiones de objetos de la base de datos que son mutuamente consistentes,* de tal forma que las referencias entre objetos en una configuración deben estar siempre actualizadas a las versiones apropiadas de los objetos.

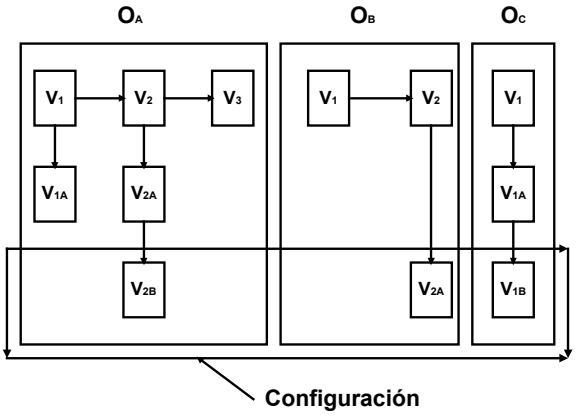


Figura 3.1. Configuración de versiones

Por lo general, para la gestión de las configuraciones los SGBDOO optan por una de estas dos opciones:

a) Proporcionar un mecanismo, pero no una política, para la gestión de las configuraciones, de forma que sea el usuario el que solucione la gestión de las configuraciones. Ej. Objectivity/DB.

b) El SGBDOO puede implementar las configuraciones automáticamente, con algunas opciones de la política especificadas por el usuario. Ej. ObjectStore.

3.2.3 Gestión del almacenamiento secundario

Un SGBD debe proporcionar la capacidad de tratar eficientemente con grandes volúmenes de datos ubicados en almacenamiento secundario. Para ello recurre a un conjunto de mecanismos que no son visibles al usuario, tales como la gestión de índices o la agrupación de datos.

3.2.3.1 Indexación

El objetivo de la indexación es mejorar el tiempo de acceso a la base de datos. Las técnicas de indexación o métodos de acceso tradicionales han tenido que evolucionar en el caso de los SGBDOO para albergar las características derivadas del nuevo modelo de datos. Esto ha producido una proliferación de técnicas de indexación para bases de datos orientadas a objetos (véase capítulo 11) que intentan abarcar dichas características.

De estas técnicas no se puede concluir que una de ellas proporcione un mejor rendimiento que las demás para todas las situaciones. Esto es básicamente debido a que determinados métodos de acceso se adaptan mejor a determinadas peticiones de información, y además, dada la gran variedad de aplicaciones a las que los SGBDOO pueden dar soporte, es difícil anticipar las necesidades de los mismos. Es por eso que sería ideal que el SGBDOO permitiese añadir nuevos métodos de acceso que cubriesen mejor las necesidades de aplicaciones particulares. Conseguir un sistema que facilite esta tarea es precisamente uno de los objetivos de esta tesis doctoral y será, por tanto, tratado en detalle en capítulos posteriores (véase capítulo 12).

3.2.3.2 Agrupamiento ('clustering')

El objetivo del agrupamiento es reducir el número de operaciones de entrada/salida que es necesario hacer sobre disco para la recuperación de los objetos. Las técnicas de agrupación para los SGBDOO, comparadas con las de los SGBDR, deben tener en cuenta la existencia de objetos complejos, de herencia (múltiple o simple), y de métodos. Un agrupamiento adecuado puede incrementar considerablemente el rendimiento de las aplicaciones.

Criterios de agrupación

En un SGBDOO el *agrupamiento* puede realizarse atendiendo a diferentes criterios:

- **Objetos compuestos.** Consiste en agrupar los objetos de acuerdo con sus relaciones de agregación. La estructura de un objeto complejo se puede representar como una jerarquía o como un grafo acíclico dirigido, y una buena estrategia en estos casos puede ser organizar los nodos del grafo en una secuencia de agrupamiento lineal [BKK+88].
- **Referencias.** Consiste en agrupar los objetos de acuerdo a sus relaciones con otros objetos. El anterior es por tanto un caso especial de éste.

- **Clases.** Consiste en agrupar los objetos por su clase. La mayoría de los SGBDOO agrupan por clase, sin embargo, esta forma de agrupamiento no es útil a menos que se espere un acceso repetido a objetos de una clase.
- **Índices.** Consiste en agrupar los objetos en función de un índice definido sobre sus atributos. Es eficiente cuando los objetos son accedidos frecuentemente por ese atributo.
- **A petición (custom).** Algunos SGBDOO permiten especificar el agrupamiento por ejemplo, en el momento de la creación.

La unidad de agrupación más usual suele ser la *página*, generalmente la unidad física más pequeña leída desde disco, aunque a veces se emplea el *segmento*. Las páginas son más adecuadas para el agrupamiento por índices, referencia u objetos compuestos. Los segmentos son más adecuados para el agrupamiento por clase. No obstante, los mejores rendimientos suelen obtenerse mediante el agrupamiento por páginas, ya que las páginas son la unidad de acceso desde el disco.

Tipos de agrupamiento

El agrupamiento puede ser estático o dinámico. El **agrupamiento estático** implica que una vez que se ha definido un agrupamiento éste no puede cambiar en tiempo de ejecución. Esto implica que no se tiene en cuenta la evolución dinámica de los objetos (creación y eliminación), con lo que con la actualización de los objetos se puede destruir la estructura de agrupamiento inicialmente creada, y además, en entornos con múltiples relaciones entre objetos, diferentes aplicaciones pueden requerir diferentes patrones de acceso, con lo que el rendimiento bajará si se emplea un agrupamiento simple.

El agrupamiento estático al no reagrupar los objetos una vez que han sido creados, puede incrementar mucho el costo de acceso a los objetos ya que éstos pueden estar esparcidos entre múltiples páginas. El **agrupamiento dinámico** debería intentar reorganizar las páginas esparcidas cuando el coste de acceso ya sea demasiado alto. Por supuesto la reorganización puede suponer un coste elevado, por lo que se han propuesto modelos para evaluar el beneficio y sobrecarga que ocasiona el reagrupamiento [CH91].

3.2.3.3 Caché

La caché de objetos (*object cache*) es un espacio de memoria que funciona a modo de buffer tradicional entre el almacenamiento secundario y la aplicación. Los objetos traídos desde el disco son almacenados en la caché de objetos lo que permite que la aplicación pueda acceder a los objetos directamente como si fueran objetos transitorios del lenguaje de programación. Sistemas como ObjectStore, POET y O_2 emplean esta técnica.

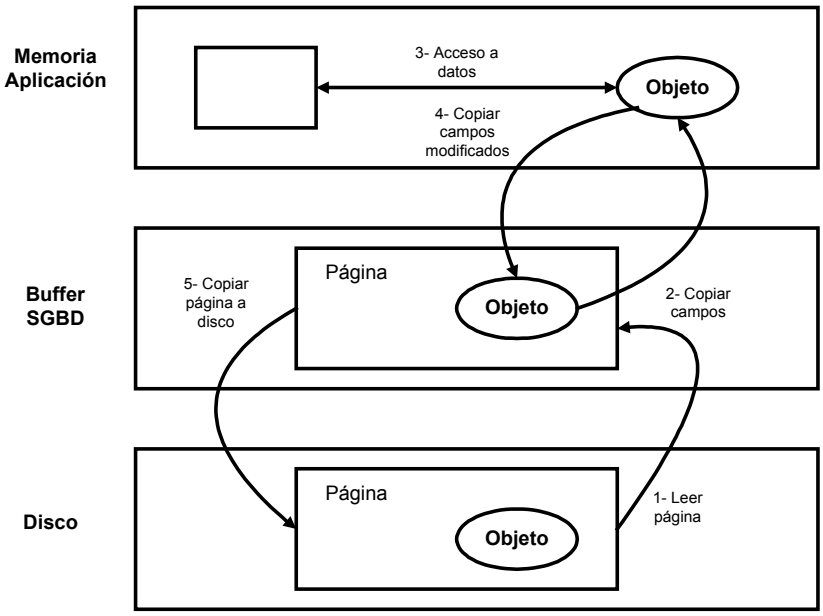


Figura 3.2. *Caching*

La caché no es una técnica para gestionar el almacenamiento secundario, pero se incluye aquí porque tiene como objetivo minimizar el número de accesos a disco almacenando el conjunto de datos que la aplicación emplea en un periodo (*working set*), de forma que si éstos cambian lentamente se reducen dichos accesos.

Esta técnica presenta dos dificultades: conocer por adelantado los objetos que empleará la aplicación, y hacer la caché lo suficientemente grande como para albergar a dichos objetos. Para ayudar a predecir los objetos algunos sistemas emplean *eager swizzling* [Mos90], que consiste en recorrer las referencias de los objetos que ya están *swizzled* para cargar todos los objetos alcanzables y realizar el *swizzling* a sus referencias.

El *caching* también puede reducir la necesidad de interacción entre cliente y servidor en arquitecturas de este tipo, reduciendo el tráfico de la red. Sin embargo, necesitan incorporar un protocolo de consistencia de la caché que puede añadir una sobrecarga al sistema [CFL+91].

3.2.4 Protección

Los datos almacenados en una base de datos deben estar protegidos contra los accesos no autorizados de la destrucción o alteración malintencionadas, y de la introducción accidental de inconsistencias. Esta protección tiene dos componentes:

- **Fiabilidad ('safeness')**. Un usuario no puede corromper estructuras de datos físicas mediante el lenguaje de programación, lenguaje de consulta o bien herramientas proporcionadas al usuario final.
- **Seguridad.** Un usuario no puede acceder a datos para los que no tiene autorización. La fiabilidad es por tanto un prerequisito necesario para la seguridad.

Los modelos de autorización para los SGBDOO tienen que tener en cuenta que la unidad de autorización puede variar: la clase, objetos compuestos, objetos individuales, etc. De esta forma se puede autorizar el acceso a los datos de un objeto pero también a los métodos. Otra cuestión a tener en cuenta es que en una jerarquía de clases el acceso

a una clase no implica autorización para acceder a sus subclases. Un modelo de autorización bastante completo es presentado en [RBK+91].

3.2.5 Lenguaje de consulta

Un SGBDOO se caracteriza por integrar un lenguaje de programación orientado a objetos con las capacidades de bases de datos. El hecho de emplear un lenguaje de programación como base garantiza un acceso navegacional a los datos. Sin embargo, la posibilidad de realizar consultas *ad hoc* (estilo SQL) es una característica de las consideradas como básica en los SGBDOO. Esto plantea entonces la necesidad de lenguajes de consulta, y por estar basados éstos en un modelo de objetos, se complica tanto la optimización como el procesamiento de las consultas.

3.2.5.1 Procesamiento y optimización de consultas

El procesamiento de consultas en bases de datos orientadas a objetos es más complicado que en el modelo relacional debido básicamente a una serie de características del propio modelo de objetos:

- **Tipos de datos diferentes.** Un lenguaje de consulta para una base de datos orientada a objetos puede emplear diferentes tipos (listas, arrays, etc.) frente al único tipo de las bases de datos relacionales (relación). Y además el tipo de entrada y de salida en una consulta puede ser muy distinto. Esto es una de las causas que dificultan el diseño de un álgebra de objetos.
- **Objetos complejos.** Un objeto puede tener referencias a otros objetos, que se implementan mediante IDOs. Algunas de las álgebras propuestas tienen operadores especiales para IDOs y objetos.
- **Jerarquía de clases.** Las jerarquías de clases ocasionan dos problemas principales: por un lado, la creación de índices que se acomoden a los diferentes accesos sobre la jerarquía (véase capítulo 11), y por otro lado, es necesaria una optimización en tiempo de ejecución, además de la de tiempo de compilación, ya que es posible que no se sepa que implementación del método se va a emplear en tiempo de compilación (*late-binding*).
- **Métodos.** Es muy difícil estimar el costo de ejecución de un método, ya que generalmente el código está oculto al optimizador de consultas.

A pesar de las diferencias entre el modelo relacional y el orientado a objetos las metodologías para el procesamiento de consultas relacionales pueden ser adoptadas para el procesamiento de consultas en bases de datos orientadas a objetos [YM98].

3.2.6 Evolución

En las bases de datos orientadas a objetos hay dos aspectos relativos a la evolución de objetos especialmente relevantes: evolución de esquemas y evolución de instancias.

3.2.6.1 Evolución de esquemas

Las aplicaciones hacia las que van dirigidos los SGBDOO suelen cambiar el esquema con bastante frecuencia, y esto representa un problema complejo ya que a medida que los modelos de datos son más avanzados, aumentan sus características y posibles transformaciones.

El hecho de hacer modificaciones en el esquema supone tres implicaciones: modificaciones a los programas que emplean el viejo esquema, modificación a las

instancias existentes de las clases modificadas, y efectos de los cambios sobre el resto del esquema. Con el fin de asegurar la consistencia, cada modelo de base de datos define unos *invariantes de esquema* (reglas) que deben ser verificados.

Estrategias de aplicación de modificaciones

Como se ha mencionado con anterioridad determinadas modificaciones en las clases (ej. eliminar un atributo) afectan a las instancias ya creadas de dichas clases. Para aplicar los cambios a las instancias existentes los sistemas emplean diferentes estrategias [BM93]:

- **Almacenar las clases una sola vez.** Esta aproximación es la más simple de todas y consiste en no permitir modificaciones en las clases una vez que las instancias ya han sido creadas. Si un usuario desea definir un nuevo atributo o cambiar una clase, debe definirse una nueva clase y copiar los datos antiguos a ésta.
- **Actualización perezosa.** Consiste en diferir las modificaciones sobre las instancias. Un mecanismo basado en esta estrategia consiste en que las instancias se modifiquen en el momento en el que las aplicaciones acceden a ellas. La modificación propuesta con esta estrategia no es costosa, pero el acceso a las instancias puede ser lento. Ej ORION.
- **Actualización inmediata**. Implica la modificación de todas las instancias tan pronto como la definición de la clase se modifica. Esta estrategia hace la modificación más costosa que la anterior.

Otro problema que puede ocasionar la modificación del esquema, es la *inconsistencia de métodos*. El hecho de eliminar un atributo de un objeto, hace que un método que emplee dicho atributo no sea consistente en el esquema actual. Para este problema no se ha encontrado todavía una solución que se pueda aplicar con carácter general, ya que los métodos a menudo se implementan en lenguajes de programación imperativos, y no es fácil determinar los efectos de una modificación de esquema en el conjunto de los métodos de las clases involucradas en la modificación.

Versiones y vistas

Los SGBDOO poseen otros mecanismos para permitir las modificaciones del esquema, entre los que se encuentran las *versiones de esquema* y *las vistas.* Estos enfoques difieren de los enfoques anteriores en que el viejo esquema no se reemplaza por el esquema modificado; se dispone de dos esquemas: el original, y el otro que refleja las modificaciones ejecutadas sobre el original.

Una diferencia esencial entre usar versiones de esquema y vistas es que, adoptando el primer enfoque, un objeto creado en una versión específica del esquema es visible sólo bajo esta versión específica del esquema, mientras que al adoptarse el último enfoque un objeto es visible en todas las vistas si las condiciones para pertenecer a éstas se cumplen. Una de las ventajas de emplear vistas en lugar de modificaciones directas del esquema es que si, por ejemplo, una modificación de esquema resulta ser no satisfactoria, podría retornarse al esquema original sin pérdida de información. Así, por ejemplo en los modelos de vistas planteados en [GBC+97] es posible simular la creación de nuevos atributos a las clases.

3.2.6.2 Evolución de instancias

Al igual que se pueden modificar los atributos de un objeto (evolución de estado), también se pueden modificar la estructura y funcionamiento de los objetos individuales mientras se mantenga constante su identidad. Esta modificación de la estructura y/o comportamiento puede conseguirse de tres formas:

- *Migración de objetos hacia clases diferentes*.
- *Adición dinámica de clases*, incluso aquellas no relacionadas por la herencia.
- *Especialización de instancias*.

La migración es diferente a añadir una nueva clase a un objeto, ya que en el primer caso se pierde la clase a la cual pertenecía la instancia previamente, mientras que en el segundo caso, esto no ocurre, pero para ello un objeto debe ser capaz de ser instancia de varias clases al mismo tiempo. Generalmente, estos tipos de evolución no son soportados por muchos sistemas debido a que crean problemas de implementación y de consistencia, y de hecho la mayoría de los sistemas que lo permiten se restringen a que los objetos migren hacia subclases de la clase a la que pertenecen.

3.3 Algunas características del modelo de objetos

3.3.1 Identidad de objetos

Los SGBDOO emplean identificadores de objetos (IDOs), independientes del estado del objeto e invariantes a lo largo de su vida, generados automáticamente por el sistema para identificar los objetos unívocamente e implementar referencias entre ellos. El hecho de emplear IDOs no restringe el empleo de *claves* al igual que en el modelo relacional (de hecho, muchos SGBDOO lo permiten), sin embargo, ya no son imprescindibles como en el modelo relacional.

3.3.1.1 Representación

La representación de los identificadores de objetos para un gestor de objetos puede ser un factor importante en el rendimiento del sistema, ya que un objeto se obtiene por medio de su IDO. Un IDO puede ser *físico* (contiene la dirección real del objeto) o *lógico* (es un índice a partir del que se obtiene la dirección real). Aunque se han propuesto diferentes enfoques para la representación de los IDOs [KC86], la mayoría coinciden en al menos cuatro tipos:

- **Dirección física.** El IDO es la dirección física del objeto. Es muy eficiente. Se utiliza en los lenguajes de programación pero raramente en los SGBDOO ya que si un objeto es borrado o trasladado hay que modificar todos los objetos que contienen ese IDO.
- **Dirección estructurada.** El IDO consta de dos partes. La primera contiene el número de página y el número de segmento, y la segunda contiene un número lógico de desplazamiento para obtener el objeto dentro de la página. Esta representación permite recuperar el objeto con un único acceso a página, y facilita su reubicación. Ejemplos de sistemas con esta representación para los identificadores son Ontos y Objectivity/DB.
- **Subrogado.** Se basa en generar el IDO mediante un algoritmo que garantiza su unicidad. Estos IDOs se transforman en direcciones físicas empleando un índice. No son muy eficientes para la obtención de objetos. Generalmente hay que

transformarlos en una dirección mediante funciones de dispersión (hash), pero si se emplea una función de dispersión bien equilibrada se pueden obtener los objetos con un único acceso a disco. Ejemplos de sistemas con esta representación son GemStone y PostGres.

- **Subrogado con tipo.** Es una variante del anterior, en la que se incluye una parte de identificador de clase, y una parte de identificador del objeto. Para cada clase se utiliza un contador diferente para generar la porción de identificador del objeto. El identificador de clase en el IDO permite determinar la clase del objeto sin necesidad de traer el objeto desde el disco. Proporcionan un rendimiento similar a los anteriores, pero la presencia del identificador de clase en el IDO dificulta la evolución de instancias. Ejemplos de sistemas con esta representación son Itasca, Orion y Jasmine.

3.3.1.2 Tamaño

Otro factor de los IDOs que afecta al rendimiento es la longitud, ya que puede tener un efecto significativo sobre el tamaño global de la base de datos. Una longitud de 32 bits permite manejar hasta 4 millones de objetos, pero los IDOs de 64 bits son más adecuados para entornos distribuidos o entornos en los que éstos no se reutilizan.

3.3.2 Atributos

Un SGBDOO debe proporcionar la posibilidad de que los atributos de los objetos tratados sean simples o complejos. Los atributos simples son aquellos que toman valores literales (entero, cadena, bitmap, etc.), mientras que los complejos pueden ser de tres tipos:

- **Referencias,** que se emplean para representar relaciones entre objetos.
- **Colecciones,** que pueden ser listas, conjuntos o arrays de valores (simples o referencias). Estas colecciones suelen tener una gran cantidad de elementos, y por ello la mayoría de los SGDBOO recurren a *iteradores,* que proporcionan al programa los elementos solicitados y permiten además bloquear los datos en uso actualmente, y no la colección completa.
- **Derivados,** que pueden ser definidos proceduralmente en vez de ser almacenados explícitamente. Estos atributos no suelen emplearse en los lenguajes donde la sintaxis para referirse a un atributo es diferente de la empleada para invocar un método o función. POSTGRES y O2 los proporcionan.

En algunos sistemas los literales son considerados como objetos y pueden ser tratados sintáctica y semánticamente como ellos.

3.3.3 Relaciones

Un SGBDOO debe proporcionar un mecanismo para representar las relaciones entre los objetos, y además este mecanismo debe mantener la integridad referencial evitando anomalías al borrar objetos o al redefinir la relación.

- **Relaciones Binarias.** En un modelo orientado a objetos se representan mediante referencias y colecciones de referencias. La representación depende de la multiplicidad y del grado de la relación. Por ejemplo, una relación muchos-a-muchos entre dos objetos de clases A y B se representan añadiendo un atributo conjunto de referencias a cada uno.

- **Relaciones Ternarias.** Se representan mediante una nueva clase al igual que ocurre con la representación relacional.
- **Atributos Inversos.** Se emplean para representar las relaciones y garantizar la integridad referencial de las mismas. Deben llevar un nombre asociado, al igual que ocurre en el modelo relacional. Es por ello que en cada clase y para cada relación se especifica el atributo inverso de la otra clase con la que está establecida la relación.

3.3.3.1 Integridad referencial

Los niveles de integridad referencial que puede proporcionar un SGBDOO son variados [Cat94]:

- **Sin comprobación de integridad.** Un sistema puede no proporcionar chequeos sobre las referencias (puede haber referencias a objetos inexistentes o del tipo equivocado). El programador debe mantener la integridad de las referencias.
- **Validación de la referencia.** Este nivel asegura que las referencias son a objetos que existen y son del tipo correcto. Esto puede realizarse de dos formas:
    a) El sistema puede borrar objetos automáticamente cuando no son más accesibles por el usuario (Gemstone), y para ello emplea algún algoritmo de recolección de basura (ej. contador de referencias). No suele permitir borrado explícito.
    b) El sistema requiere borrado explícito pero puede detectar referencias inválidas automáticamente (ej. VERSANT).
- **Integridad de las relaciones.** El sistema permite el borrado y la modificación explícita de objetos y relaciones, y garantiza la corrección de las relaciones. Para ello emplea los atributos inversos. La mayoría de los SGBDOO proporcionan integridad referencial (Objectivity/DB, ObjectStore, ONTOS), y algunos sistemas permiten a los usuarios elegir si desean mantener la integridad por la aplicación para evitar la sobrecarga (Objectivity y ONTOS).
- **Semántica de las referencias configurable.** Los sistemas más sofisticados permiten al diseñador especificar la semántica de la integridad referencial para cada clase o relación. Normalmente, los SGBDOO existentes no proporcionan esta posibilidad excepto para restricciones de dependencia por existencia limitadas. Una forma de conseguir esta semántica sería codificándola en métodos asociados con los objetos. Sin embargo, es mucho mejor que la integridad referencial sea controlada por el SGBD lo que hace la semántica explícita en el esquema, donde es visible a programas y usuarios, y es menos vulnerable a errores de código.

Algunos SGBDOO no proporcionan pares de atributos inversos con lo que no proporcionan integridad referencial. Otros SGBDOO proporcionan pares de atributos inversos pero permiten además definir atributos sin inversos. Esos modelos orientados a objetos más limitados no garantizan el mismo nivel de integridad referencial que el proporcionado por el modelo relacional.

3.3.3.2 Swizzling

El *swizzling* es una técnica que se emplea para aumentar la velocidad a la que se puede "navegar" por las referencias entre los objetos (IDOs), y se basa en convertir los

IDOs en direcciones de memoria cuando los objetos son traídos de disco a memoria principal. Sin embargo, transformar los IDOs en direcciones de memoria y viceversa es un proceso costoso, y las ventajas de una rápida navegación entre los objetos pueden no justificarse por el costo generado. Así, si los objetos tienen una alta probabilidad de intercambiarse entre disco y memoria, o si las referencias no se usan un número significativo de veces es más recomendable emplear otros mecanismos (ej. tablas que asocian los IDOs a las direcciones de los objetos en memoria [Mos90] como es el caso de Objectivity).

Cuando un SGBDOO emplea *swizzling* tiene diferentes alternativas donde ninguna de ellas es claramente superior a las otras, con lo que puede emplearse más de una. Se puede realizar el *swizzling* la primera vez que una aplicación trae el objeto desde el disco, la primera vez que se tiene que seguir una referencia, bajo demanda de la aplicación (lo que requiere programación adicional) o bien, se puede conseguir que las referencias estén siempre traducidas (swizzled), como hace ObjectStore, lo que limita el número total de objetos en la base de datos al tamaño máximo de la memoria virtual.

3.3.4 Objetos compuestos

Un objeto compuesto se obtiene agregrando o agrupando objetos. La conveniencia de los objetos compuestos viene impuesta por dos razones principales: su empleo como la base para el *clustering* físico y, su uso para realizar operaciones basándose en la abstracción del objeto compuesto. Cuando un objeto compuesto se borra o copia el SGBDOO puede automáticamente borrar o copiar todos los objetos componentes.

3.3.5 Almacenamiento y ejecución de métodos

Una de las características de la orientación a objetos es la encapsulación dentro de una clase de atributos y de métodos. Los SGBDOO deben entonces, a diferencia de los SGBDR, proporcionar un mecanismo para el almacenamiento y la ejecución de métodos.

En los SGBDOO hay dos opciones para el manejo de métodos:

- **Almacenar los métodos en ficheros externos**. Los procedimientos son almacenados empleando los ficheros de un lenguaje de programación convencionales; están asociados simplemente con las clases por las declaraciones en los programas de los usuarios. Esta opción es similar a las bibliotecas de funciones encontradas en los SGBD tradicionales, en las que una aplicación interactúa con un SGBD mediante el enlazado con funciones proporcionadas por el SGBD.
- **Almacenar los métodos en la base de datos.** Con esta opción los métodos (el código fuente, y opcionalmente una forma compilada binaria) son almacenados en la base de datos y ligados dinámicamente a la aplicación en tiempo de ejecución. Esta opción elimina código redundante y simplifica las modificaciones al existir una única copia del método. Facilita la implementación de la integridad y permite también aplicar la seguridad y la concurrencia del SGBDOO a los métodos, al encontrarse éstos almacenados en la propia base de datos. GemStone, Itasca, O2 y Postgres permiten que los métodos sean almacenados y activados desde la base de datos.

Ambas alternativas ofrecen problemas especiales si los procedimientos contienen código en múltiples lenguajes de programación. Es necesario construir un entorno e

interfaz para la invocación de procedimientos que soportan llamadas para lenguajes cruzados.

3.4 Arquitectura de bases de datos orientadas a objetos

La distribución es un requerimiento esencial en un SGBDOO ya que las aplicaciones que requieren estos sistemas son empleadas generalmente en un entorno de red. Los primeros SGBDOO (ej. GemStone) empleaban una arquitectura cliente-servidor en la que múltiples estaciones accedían a una base de datos centralizada sobre el servidor. Sin embargo, como resultado de los avances en hardware el poder de procesamiento se fue concentrando en los clientes más que en el servidor. De la misma forma, con el aumento de la capacidad y velocidad de los discos duros de las estaciones, fueron tomándose más en consideración las bases de datos distribuidas (Ej. ONTOS y ORION distribuido).

3.4.1 Cliente-servidor

La arquitectura tradicional cliente-servidor es de dos capas (múltiples clientes y un servidor), pero actualmente existen versiones más evolucionadas de está arquitectura que facilitan la escalabilidad de las aplicaciones al disponer de tres (o más capas) [Kir99]: una capa se encarga de la presentación, otra de la lógica de aplicación, y otra del acceso a los datos.

Sin embargo, la mayoría de los SGBDOO están implementados en un cliente-servidor de dos capas pero con variaciones ocasionadas por la funcionalidad asignada a cada componente [DFM+90]. En los apartados siguientes se analizan brevemente dichas variaciones.

3.4.1.1 Servidor de objetos

Esta arquitectura intenta distribuir el procesamiento entre los dos componentes (Figura 3.3). Esta arquitectura es empleada en sistemas como ORION y 02. El servidor es el responsable de gestionar el almacenamiento (*clustering*, métodos de acceso,...), los bloqueos, el *logging* y la recuperación, forzar la seguridad y la integridad, optimizar las consultas y ejecutar los procedimientos almacenados. El cliente es el responsable de la gestión de las transacciones y de la interfaz al lenguaje de programación.

- La unidad de transferencia entre el servidor y el cliente es el objeto. En esta arquitectura el servidor entiende el concepto de objeto y es capaz de aplicar métodos a los objetos. El hecho de emplear como unidad de transferencia el objeto puede dificultar el bloqueo a nivel de página ya que ni el cliente ni el servidor tratan realmente con páginas.
- En esta arquitectura la mayoría de la funcionalidad del SGBDOO es replicada en el cliente y en el servidor. Cuando el cliente necesita un objeto primero busca en su *caché* de objetos (*object cache*), si no encuentra el objeto lo solicita al servidor. Si el objeto no está en la caché del servidor, el servidor lo recupera de disco y se lo envía al cliente. Por otro lado, la replicación del gestor de objetos tanto en el servidor como en el cliente permite que los métodos sean ejecutados también en ambos, ya que este gestor entre otras funciones proporciona un contexto para la ejecución de métodos.

- Esta arquitectura es relativamente insensible al *clustering*. Para que sea viable debe incorporar algún mecanismo para pasar múltiples objetos o valores en cada mensaje entre el cliente y el servidor.

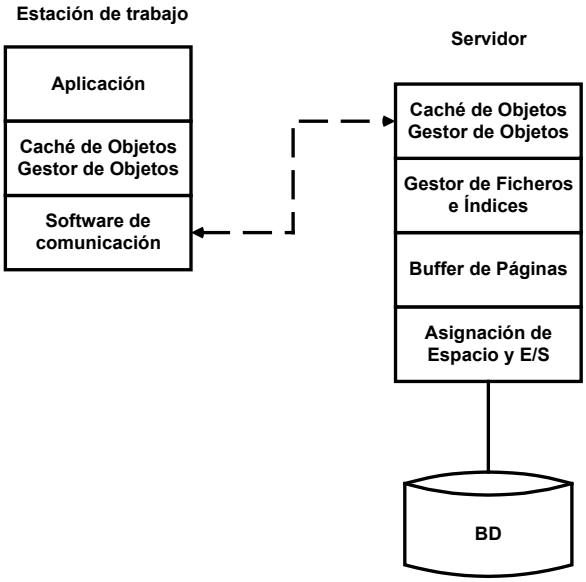


Figura 3.3. Arquitectura de un servidor de objetos

3.4.1.2 Servidor de páginas

En esta opción el servidor únicamente trata con páginas y no entiende por tanto, la semántica de los objetos. Además de proporcionar el almacenamiento y la recuperación de las páginas, el servidor proporciona control de concurrencia y servicios de recuperación para el software de base de datos que se ejecuta sobre los clientes (Figura 3.4). Ejemplos con esta arquitectura son Exodus y Observer.

- En esta arquitectura la unidad de transferencia es la página de disco. Cuando el servidor recibe una petición de una página, primero asigna el bloqueo sobre la página, y si la página no está en el buffer, el servidor recupera la página de disco y la envía al cliente.
- En esta arquitectura el *buffering* sobre el cliente puede ser hecho en términos de páginas, de objetos o de ambas. La ventaja de la caché de objetos es que el espacio del buffer no se desperdicia conteniendo objetos que no han sido referenciados por la aplicación recientemente, sin embargo esto acarrea el costo de copiar cada objeto desde la página a la caché local. Además, si se actualiza un objeto en la caché, puede que haya que volver a cargar la página desde el servidor (nueva operación de entrada/salida). En esta arquitectura puede ser difícil de implantar el bloqueo a nivel de objetos, especialmente cuando coinciden cambios a la misma página por dos o más clientes.
- Esta arquitectura sí se beneficia del *clustering*.

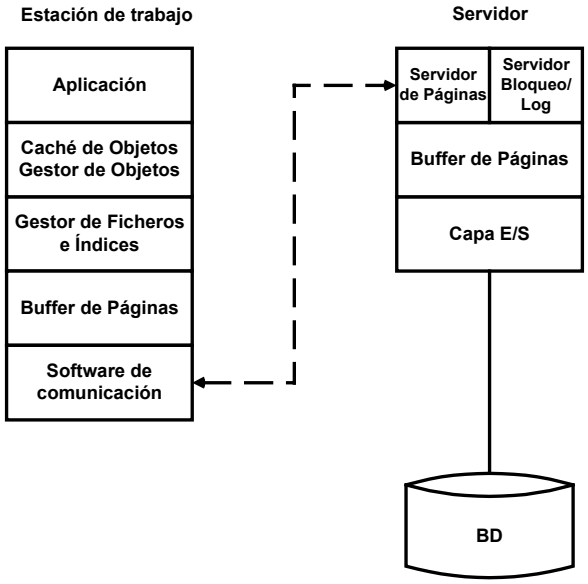


Figura 3.4. Arquitectura de un servidor de páginas

3.4.1.3 Servidor de consultas

En esta opción la mayor parte del procesamiento es realizado por el servidor. El cliente simplemente envía peticiones al servidor, recibe los resultados y se los envía a la aplicación. Esta es la opción tomada por muchos SGBDR.

3.4.2 Bases de datos distribuidas

Un SGBDOO distribuido permite la gestión transparente de los objetos que están distribuidos en diferentes máquinas [ÖDV94], es decir, tiene que proporcionar: *transparencia de localización* (oculta la existencia de la red y la distribución de los datos), *transparencia de replicación* (enmascara la existencia de copias físicas de datos) y *transparencia de fragmentación* (evita que el usuario tenga que tratar con las particiones de las bases de datos de objetos). Esta distribución transparente lleva implícita una serie de beneficios de entre los que se destacan:

- **Duplicidad de objetos.** Los objetos pueden aparecer duplicados en más de un sitio (de forma totalmente transparente para el usuario), lo que incrementa la confiabilidad del sistema (ante fallos de un nodo), y el rendimiento al poder colocar los datos cerca del punto de su utilización reduciendo el tráfico de la red.
- **Escalabilidad.** Los sistemas distribuidos permiten variar su tamaño de un modo sencillo. Se pueden agregar ordenadores adicionales a la red conforme aumentan por ejemplo el número de usuarios.

La distribución añade más complicación si cabe a cada una de las características de un SGBD como por ejemplo al procesamiento de consultas. Los objetos se pueden encontrar en varios nodos, y a su vez estos objetos pueden estar formados por otros objetos que se encuentran distribuidos en otros nodos, y así sucesivamente, lo que puede repercutir en un rendimiento más bajo del sistema. Por ello, conviene hacer un estudio cuidadoso antes de dividir la base de datos.

3.4.2.1 Fragmentación para BDOO

En el caso de las bases de datos orientadas a objetos hay que tener en cuenta que los objetos encapsulan estado y comportamiento, siendo este comportamiento definido para

la clase, y compartido por todas las instancias de la misma. Por lo tanto, conviene destacar las diferentes alternativas para el almacenamiento de las clases con relación a los objetos:

- Almacenar los objetos con sus clases. De esta forma las diferentes clases se almacenan en sitios diferentes, y objetos de una determinada clase pueden localizarse en sitios diferentes (*horizontal class partitioning*). En este caso las clases se duplican en cada sitio donde hay una instancia de esa clase, o bien la clase se almacena en un sitio y se soporta el procesamiento remoto.
- Particionar la especificación de la clase dividiendo su comportamiento (propiedades y métodos) en diferentes fragmentos (*vertical class partitioning*) y distribuyendo los objetos de esa forma. De nuevo es necesario el procesamiento de objetos remoto.

3.4.2.2 Estrategias para distribuir un SGBDOO

Tradicionalmente, un SGBDOO puede adoptar dos posiciones a la hora de distribuir una base de datos, que difieren en el grado en el que las bases de datos participantes son interdependientes:

- **Esquema Local.** En la distribución con esquema-local cada base de datos tiene su propio esquema. En este caso es necesario un esquema de traducción entre bases de datos clase a clase, de forma que los esquemas puedan ser referenciados en otras bases de datos. Si los tipos de las dos bases de datos no son idénticos la traducción entre los esquemas locales debe definir como se traducen los tipos de un esquema a otro. Otra posibilidad es proporcionar una distribución de esquema local reducida, permitiendo referencias a tipos en otra base de datos solamente cuando son idénticos a los tipos locales.
- **Esquema Global.** En una distribución con este esquema, uno o más de los recursos de bases de datos son centralizados. Un esquema de datos compartido une todas las bases de datos en un dominio global simple. Las clases y atributos en el esquema compartido son válidos sobre todas las bases de datos en el dominio.

El esquema global garantiza que los tipos son compatibles entre bases de datos en una organización, pero requiere que las organizaciones se coordinen en la definición del esquema. La distribución con esquema local es más flexible, ya que cada esquema de base de datos puede evolucionar independientemente, pero siempre se debe hacer una traducción de esquemas para poder acceder a otra base de datos.

3.5 Interfaz de usuario y herramientas de alto nivel

Las herramientas de alto nivel, y en definitiva la interfaz de usuario ofrecida por un SGBDOO, son un factor también determinante para la productividad del usuario con el sistema. Básicamente, un SGBDOO debe proporcionar una herramienta de alto nivel que permita el acceso a los datos y al esquema, así como asistir al programador en la generación de aplicaciones. Sin embargo, la mayoría de las interfaces existentes en los sistemas actuales están muy ligados a la implementación del mismo, dejando de lado la consecución de una herramienta usable para los diferentes tipos de usuarios de los que será objeto.

Las herramientas deseables para un SGBDOO se pueden clasificar en dos bloques: herramientas para el desarrollo y herramientas para la administración.

3.5.1 Herramientas de desarrollo

El objetivo de estas herramientas es ayudar a describir la aplicación en términos de objetos, construyendo el mejor modelo y permitiendo después transferir ese modelo al esquema para posteriormente poder manipular la información en él almacenada. La mayoría de los sistemas existentes actualmente incluyen dos tipos de herramientas básicas:

- **Editores de Esquema Gráficos.** Son herramientas que permiten visualizar y modificar el esquema de la base de datos. Estos editores generalmente producen declaraciones en el lenguaje de definición de datos subyacente y también son capaces de aceptar esquemas existentes como entradas.
- **Navegadores (***Object Browsers***).** Son herramientas que permiten examinar y modificar objetos en la base de datos. Puede permitir al usuario seguir las conexiones a otros objetos seleccionando los objetos sobre la pantalla.

También son deseables otro tipo de herramientas que permitan la confección de programas en el lenguaje de manipulación de datos seleccionado, así como consultar la base de datos indicando de forma visual la información que se desea obtener, y generando la especificación en el lenguaje de consulta seleccionado.

3.5.2 Herramientas de administración

El objetivo de estas herramientas es facilitar las tareas relacionadas con la administración tanto de la base de datos como de los usuarios.

- **Administración de la base de datos.** En este caso las herramientas podrían ayudar al usuario en tareas tales como back-up (completo, incremental, etc.), recuperación o configuración del sistema.
- **Administración de los usuarios.** Estas herramientas tienen como objetivo ayudar al administrador de la base de datos en tareas tales como la protección de la base de datos de los usuarios, o la restricción de los accesos de lectura, escritura y borrado a los objetos de una clase.

Es aconsejable también que las herramientas de administración dispongan de una versión en línea de comandos.

Capítulo 4 Requisitos de un SGBDOO

En el capítulo anterior se ha definido un SGBDOO como un SGBD con soporte para un modelo de objetos. Es por eso que entre los requisitos fundamentales de todo SGBDOO está el hecho de proporcionar las funcionalidades básicas de un SGBD, como son el soporte para la persistencia, la gestión de la concurrencia, la gestión del almacenamiento secundario y la posibilidad de consultas.

Por otro lado en el capítulo 2 se concluyó que los SGBDOO son la mejor solución para la gestión de objetos, sin embargo, se pudo observar que estos sistemas no están en absoluto exentos de problemas. Parte de estos problemas, son inherentes a los propios sistemas, y otros son generados por los sistemas operativos sobre los que son construidos. Se plantea entonces el diseño y la construcción de un SGBDOO que intente aliviar dichos problemas.

En este capítulo se identifican los requisitos que debe cumplir dicho SGBDOO. Estos requisitos afectan básicamente al diseño y a la construcción del mismo.

4.1 Aplicación de las tecnologías de objetos a la construcción del SGBDOO

En la actualidad, ingeniería del software y metodologías orientadas a objetos son casi sinónimos. Los objetos están presentes en todas las fases del desarrollo de software: análisis, diseño e implementación. Existen gran cantidad de libros y artículos que hablan de los beneficios de la utilización de las tecnologías de objetos en el desarrollo de software, de los que destacan [Boo94, Mey97].

4.1.1 Diseño del SGBDOO como un marco orientado a objetos

Se trata en este caso de **aplicar los principios de diseño e implantación del paradigma de la orientación a objetos** al propio SGBDOO, de forma que quede organizado como un marco orientado a objetos. El sistema ha de ser diseñado como una jerarquía de clases, cada una de las cuales ofrece determinada funcionalidad.

Los beneficios de esta aproximación son dobles: por un lado, el sistema tendrá una estructura modular, en la cual cada uno de sus componentes podrá ser identificado y reutilizado; por otro lado, con el uso de mecanismos como la herencia, el sistema podrá ser configurado para unas necesidades específicas o para cumplir determinadas restricciones.

Los objetos se presentan como una herramienta muy útil para estructurar los SGBDOO por varias razones:

- Los objetos proporcionan modularidad y encapsulación, dado que separan claramente la interfaz de la implementación.
- Los sistemas estructurados en términos de objetos evolucionan fácilmente (más fáciles de extender y mantener), por reemplazo de sus partes. Además los subsistemas y objetos individuales podrán ser reutilizados con facilidad y

adaptados a nuevas condiciones de operación o entorno vía el mecanismo de herencia.

4.1.2 Objeto como abstracción única

La utilización del objeto como abstracción única es el objetivo inicial de este trabajo. No se quiere realizar, por tanto, una aplicación parcial de las tecnologías orientadas a objetos para la construcción del sistema de gestión de bases de datos, sino que se quiere llevar hasta sus ultimas consecuencias.

El SGBDOO ha de estructurarse internamente como un conjunto de objetos que implementan la funcionalidad del sistema. Además, en tiempo de ejecución, se instanciarán objetos reales de las distintas clases en que se oferta la funcionalidad del sistema, con la semántica que ello conlleva.

4.1.3 Modelo de objetos uniforme

Todos los servicios del gestor se realizarán en forma de objetos que seguirán el mismo modelo que los objetos del sistema operativo subyacente. Esto eliminará los problemas de desadaptación de impedancias provocados por los sistemas operativos tradicionales (véase capítulo 2).

El modelo de objetos común (para el sistema operativo y el SGBDOO) ha de ser por tanto lo suficientemente potente como para poder representar los modelos de los lenguajes de programación orientados a objetos más utilizados. Deberá incorporar los conceptos del paradigma de la orientación a objetos más aceptados, es decir, retener toda la semántica del modelo general de objetos implícito en las obras de referencia de la tecnología de orientación a objetos.

4.2 Interoperabilidad

El sistema ha de facilitar la interoperabilidad entre diferentes SGBDOO, permitiendo la recuperación desde una base de datos, de objetos que han sido almacenados con otra base de datos diferente. De la misma forma, ha de facilitar también la integración con múltiples lenguajes de programación.

Ambos requisitos se verán favorecidos por el empleo de un modelo de objetos uniforme como el mencionado en el apartado anterior.

4.3 Heterogeneidad y Portabilidad

El SGBDOO no debe obligar a la utilización de un determinado modelo de máquina para su funcionamiento. Debe tenerse en cuenta la existencia de numerosos tipos de máquinas incluso dentro de la misma red de trabajo, que posiblemente sean incompatibles entre sí.

Por otro lado, para llegar al mayor número de máquinas posible, interesa que el esfuerzo para portar el propio sistema de una máquina a otra sea el menor posible.

4.4 Transparencia

El SGBDOO debe garantizar la transparencia tanto de la distribución como de la persistencia.

- **Distribución.** Evitando, como en cualquier SGBD distribuido que el usuario perciba que los objetos se encuentran distribuidos en diferentes máquinas en la red.
- **Persistencia.** El sistema se debe de ocupar de almacenar y recuperar los objetos transparentemente de la memoria secundaria, haciendo que el usuario perciba un único espacio de objetos.

4.5 Integración transparente con el sistema operativo

Tradicionalmente, y como ya se comentó en el capítulo 2, las implementaciones de las abstracciones ofrecidas por los sistemas operativos no encajan con las que necesitan los SGBD, o resultan insuficientes. Es por ello, que la mayoría de los SGBD vuelven a re-implementar dichas abstracciones, limitándose por tanto a ser meras aplicaciones ejecutándose sobre el sistema operativo, pero sin aprovechar mucha funcionalidad que éste ya ofrece.

Sería por ello conveniente que el SGBDOO aprovechase al máximo algunas de las características que ofrece el sistema operativo como por ejemplo la persistencia. Pero también hay que ser consciente de que un sistema operativo no puede proporcionar toda la funcionalidad que le hace falta a un SGBD (ya que no sólo existen SGBD sobre los sistemas operativos), por la sobrecarga innecesaria que eso conllevaría para el resto de aplicaciones. Lo ideal sería entonces contar con un sistema operativo que proporcione los servicios esenciales del propio sistema operativo, más las funcionalidades más básicas de un SGBDOO que puede proporcionar eficientemente.

4.6 Flexibilidad, extensibilidad y adaptabilidad

En la mayoría de los sistemas de gestión de bases de datos tradicionales, el conjunto de servicios que proporcionan están implantados rígidamente sobre un núcleo, con lo que no pueden ser modificados para adaptarlos a un entorno de ejecución concreto. El sistema de gestión de bases de datos orientado a objetos a diseñar tiene que superar estos inconvenientes.

En primer lugar tiene que ser flexible, de tal manera que se diseñe un núcleo que permita construir sobre él mecanismos de más alto nivel y diferentes políticas. En segundo lugar, tiene que ser extensible, en el sentido que sea sencilla y no traumática la incorporación de nuevos servicios en el sistema. Y finalmente, debe ser adaptable, de tal forma que se puedan escoger aquellas facilidades que el SGBDOO vaya a ofrecer para un entorno determinado, y sin que suponga un excesivo trastorno debido a las facilidades que no se ofrezcan.

Para un sistema experimental y de investigación como éste, la flexibilidad es muy importante, ya que muy a menudo se debe experimentar reemplazando o añadiendo nuevos servicios para comprobar el comportamiento del sistema, etc.

4.7 Independencia entre mecanismos

La independencia entre mecanismos es fundamental para maximizar la adaptabilidad del sistema. Los mecanismos utilizados por las distintas facilidades del SGBDOO deben ser lo más interindependientes posibles, de tal forma que su presencia o ausencia no afecte al resto.

4.8 Independencia de mecanismos y políticas

Tal y como ocurre entre los mecanismos, es deseable conseguir la máxima independencia entre los mecanismos y las políticas (o técnicas) que siguen éstos. Dicha independencia permite mejorar también la adaptabilidad del sistema, de tal forma que para un entorno de ejecución concreto se puedan escoger las técnicas más adecuadas para cada uno de los mecanismos del SGBDOO.

4.9 Resumen de los requisitos del SGBDOO

El objetivo de este apartado es reflejar resumidamente los requisitos exigibles a un SGBDOO cara a convertirse en el sistema de gestión de objetos ideal.

4.9.1 Relativos a la construcción del SGBDOO

Los requisitos resumidos en este apartado son aquellos que deben ser especialmente considerados en el diseño y la construcción del sistema.

- Aplicación de las tecnologías orientadas a objetos. El SGBDOO ha de ser diseñado como un **marco orientado a objetos**, en el que la **única abstracción sea el objeto**. El núcleo del sistema estará formado por objetos encargados de realizar las diferentes funciones del gestor.
- Empleo de un **modelo de objetos uniforme**, coincidente con el empleado en el sistema operativo subyacente. El objetivo de este requisito será eliminar la desadaptación de impedancias existente respecto a los sistemas operativos tradicionales, y facilitar la **interoperabilidad** entre sistemas.
- **Integración transparente con el sistema operativo**, cuyo principal objetivo será evitar la re-implementación completa de las abstracciones ofrecidas por el sistema operativo.
- **Diseño de un núcleo básico flexible,** que permita la inclusión o eliminación de funcionalidad al sistema de modo sencillo. Esto se verá facilitado por un diseño en el que se garantice tanto la **independencia entre mecanismos** como entre **mecanismos y políticas.**
- El diseño y la construcción del SGBDOO han de garantizar también la **portabilidad** del mismo, así como la **transparencia** en la utilización de los recursos del sistema.

4.9.2 Relativos a la funcionalidad de base de datos

Los requisitos mencionados en el apartado anterior no eximen al sistema de otros requisitos a tener también en cuenta durante el diseño, pero que van más encaminados a proporcionar la funcionalidad propiamente dicha del SGBDOO, y que han sido ampliamente tratados en el capítulo anterior. Esta funcionalidad incluye una serie de mecanismos que garanticen como mínimo las siguientes características:

- **Persistencia**
- **Concurrencia y recuperación**
- **Gestión almacenamiento secundario**
- **Procesamiento y optimización de consultas**

- **Protección**

Estos requisitos básicos pueden verse incrementados con otros que mejoren las prestaciones del gestor, como la **evolución de esquemas**, la **distribución** o la posibilidad de **versionado**.

De todos estos mecanismos en esta tesis doctoral se tratará especialmente la consecución de un mecanismo de indexación extensible que permita fácilmente la incorporación de nuevas técnicas de indexación, y en el que pueda apoyarse el optimizador de consultas para mejorar su rendimiento.

4.9.3 Relativos al soporte de un modelo de objetos

Un SGBDOO debe ser un sistema orientado al objeto y dar soporte, por tanto, a las principales características del paradigma de orientación a objetos, tal y como se evidenció en el capítulo anterior. Luego el modelo de objetos soportado por el SGBDOO deberá incorporar al menos las siguientes características:

- **Identidad de objetos**
- **Abstracción y Encapsulamiento**
- **Jerarquías de herencia y agregación**
- **Tipos y Clases**
- **Polimorfismo**
- **Relaciones de asociación**

El objetivo del modelo de objetos es que pueda dar soporte al mayor número de lenguajes de programación orientados a objetos posible, de forma que el SGBDOO pueda ser **multi-lenguaje**, es decir, puedan emplearse diferentes lenguajes de programación para su manipulación.

4.9.4 Relativos a la interfaz de usuario

Para que el usuario pueda acceder a toda esta funcionalidad proporcionada por el SGBDOO es necesario un conjunto de herramientas que faciliten el trabajo con dicho sistema. Es decir, es necesario dotar al sistema de una **interfaz de usuario** usable y completa que facilite al usuario la manipulación del mismo. Para ello, la interfaz de usuario ha de incorporar al menos las siguientes herramientas:

- **Editor de esquemas**, que permita tanto la generación como la visualización y modificación del esquema de la base de datos. Debe tener en cuenta también la confección de las aplicaciones.
- **Navegador**, que permita la inspección (*browsing*) del contenido de la base de datos sin necesidad de acudir a la implementación de un programa o a una consulta.
- **Entorno para las consultas**, que facilite la formulación de las consultas de forma visual.

Capítulo 5 Panorama de algunos sistemas para el almacenamiento de objetos

En este capítulo se revisan diferentes posibilidades para el almacenamiento de objetos que comparten algunas de las características deseables para un SGBDOO. Se trata de detectar características comunes y estrategias que sean de utilidad para el diseño de una arquitectura para el SGBDOO.

Los sistemas revisados intentan ser representativos de las diferentes tendencias actuales y son una selección de los sistemas examinados. Los aspectos relevantes de éstos y otros sistemas para apartados específicos del SGBD se examinarán posteriormente al tratar estos apartados.

5.1 Starburst

Starburst es un sistema de gestión de bases de datos relacional extendido. El objetivo del proyecto Starburst desarrollado en el centro de investigación de Almaden de IBM (1984-1992), y que además constituye la causa de su inclusión en esta revisión, era la construcción de un prototipo completo de un SGBD extensible. La extensibilidad tenía como objetivo permitir una adaptación más fácil del sistema a las necesidades de los usuarios sin que el rendimiento del sistema se redujera. Starburst tiene dos componentes principales [LLP+91]:

- **Corona**, el procesador del lenguaje de consulta. Es el responsable de analizar sintáctica y semánticamente la consulta, elegir una estrategia de ejecución y producir el conjunto de operaciones para implementar la estrategia de ejecución. Durante cada una de estas actividades invoca al gestor de datos cuando lo necesita.
- **Core**, el gestor de datos. Entre los servicios proporcionados por este gestor se encuentra la gestión de registros, la gestión del buffer, la gestión de los caminos de acceso y el control de la concurrencia y recuperación. Este gestor ha sido dotado con una arquitectura que permite añadir entre otras cosas, nuevos métodos de almacenamiento para tablas, nuevos tipos de métodos de acceso, operaciones definidas por el usuario sobre tablas o columnas de esas tablas, nuevas estrategias de optimización de consultas, y acciones definidas por el usuario en respuesta a cambios en la base de datos.

Abstracciones de Starburst

Starburst trata las extensiones como implementaciones alternativas de ciertas abstracciones genéricas que tienen interfaces genéricas. Starburst define dos abstracciones genéricas distintas:

- **Métodos de almacenamiento de relaciones**. Existen diferentes métodos de almacenamiento (ej. en relaciones recuperables se pueden almacenar los registros secuencialmente o en las hojas de un B-Tree), y todos ellos deben soportar un conjunto de operaciones bien definidas sobre relaciones como son

borrar, insertar, destruir la relación o estimar los costos de acceso. También deben definir la noción de clave y soportar diferentes accesos en función de la clave.

- **Caminos de acceso, restricciones de integridad o triggers** *que son asociados con instancias de relaciones*. Reciben el nombre de *attachments* y son ejemplos de éstos los índices B-Tree, tablas hash, índices *join*, restricciones de integridad referencial y restricciones de integridad a nivel de registro. En principio, cualquier tipo de *attachment* puede ser aplicado a cualquier método de almacenamiento aunque algunas combinaciones pueden no tener sentido. Una relación puede tener múltiples *attachments* asociados del mismo o diferente tipo. A diferencia de las operaciones sobre los métodos de almacenamiento, las operaciones sobre los caminos de acceso no son invocadas directamente por el usuario sino que son invocadas cuando se realizan operaciones sobre las relaciones.

5.1.1 Arquitectura

La arquitectura que permite extender la gestión de datos en Starburst consta de tres componentes principales [LMP87]: Operaciones directas sobre métodos de almacenamiento y *attachments*, operaciones indirectas sobre *attachments,* y servicios comunes del entorno.

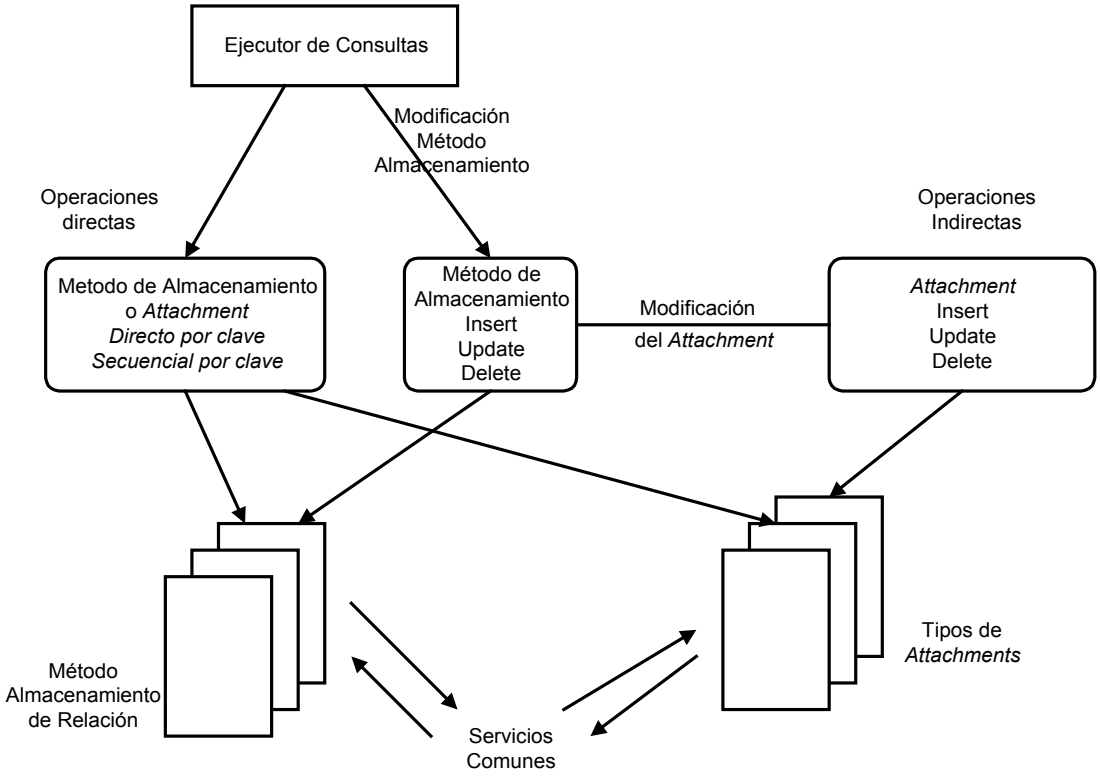


Figura 5.1. Interfaces genéricas para la gestión de datos en Starburst

Operaciones genéricas directas

Son operaciones internas del SGBD sobre relaciones y caminos de acceso. Estas operaciones son seleccionadas dinámicamente en función de los métodos de

almacenamiento o tipos de *attachments* que estén siendo manipulados. Estas operaciones son divididas en dos clases:

- Operaciones de gestión y modificación de la relación. Incluyen la inserción, borrado, y actualización de registros, así como la creación y eliminación de relaciones o instancias de *attachments*. Para especificar el método de almacenamiento o el tipo de *attachment* a emplear se ha extendido el lenguaje de definición de datos del SGBD.
- Accesos a campos de registros o claves. Estas operaciones genéricas soportan el acceso directo a relaciones y caminos de acceso ya almacenados, tanto por clave como secuencial. Además de estas operaciones de acceso, están otras operaciones para ayudar al planificador de consultas en la estimación de costos.

Operaciones indirectas

Se basa en la definición de *attachments* procedurales a las instancias relación. Estos procedimientos son invocados indirectamente como consecuencia de las operaciones que modifican las relaciones almacenadas. Siempre que es insertado, borrado o actualizado un registro en una relación se invoca el correspondiente procedimiento asociado para cada tipo de *attachment* que tiene instancias que son definidas en la relación que está siendo modificada.

Servicios comunes

Las extensiones a los métodos de almacenamiento y a los *attachments* están incrustadas en el entorno de ejecución del SGBD y por lo tanto deben obedecer ciertas convenciones y hacer uso de ciertos servicios comunes. Entre estos servicios comunes se encuentran el control de la concurrencia, la gestión de transacciones, la recuperación y la evaluación de predicados.

5.1.2 Mecanismos de extensión

Con el fin de que el rendimiento del SGBD no decaiga al dotarlo de extensibilidad es necesario al añadir nuevos métodos de almacenamiento y/o *attachments* enlazarlos con el SGBD y recompilar ciertas rutinas del servicio común, lo que permite que las extensiones puedan llamar a servicios del sistema directamente[1].

Para cada operación genérica sobre relaciones almacenadas hay un vector de procedimientos con una entrada para cada método de almacenamiento. Para operaciones genéricas sobre *attachments* hay un vector de procedimientos con una entrada para cada tipo *de attachment*. Y finalmente, para las operaciones de inserción, actualización y borrado sobre relaciones hay vectores de procedimientos asociados con entradas para cada tipo de *attachment*.

El SGBD mantiene en tiempo de ejecución los descriptores para la relación y los caminos de acceso. El descriptor de la relación está compuesto de un descriptor para el método de almacenamiento y descriptores para los *attachments* definidos sobre la relación. A través de la información almacenada en el descriptor (identificador del método de almacenamiento) se llama a la rutina del método de almacenamiento correspondiente accediendo al vector asociado. A continuación las extensiones asociadas a la relación son invocadas accediendo al vector de procedimientos asociados.

[1] La persona encargada de realizarlo no podrá ser un usuario casual

Starburst permite incluso diferir una acción hasta que ocurran ciertos eventos. Core proporciona un número de colas de eventos estándar, pero los usuarios pueden añadir colas para cualquier evento.

5.1.3 Lenguaje de consulta Hydrogen

En Starburst las operaciones sobre tipos definidos por el usuario deben realizarse en el lenguaje de consulta, sin embargo, la aplicación es escrita en un lenguaje que es una combinación del lenguaje de programación y del lenguaje de consulta. Un preprocesador separa ambos.

El lenguaje de consulta de Starburst, Hydrogen, expande SQL en dos aspectos importantes [HFL+89]: generaliza la gramática de SQL para hacer el lenguaje más ortogonal, y permite añadir nueva funcionalidad al lenguaje (definiendo nuevas funciones sobre columnas, nuevas operaciones sobre tablas y nuevos tipos de datos para columnas).

5.1.4 Crítica

5.1.4.1 Mecanismo de extensión basado en el paradigma procedimental

El mecanismo adoptado para la selección del procedimiento a emplear en función del método de almacenamiento o del camino de acceso correspondiente se vería simplificado si el desarrollo del sistema se hubiera realizado empleando herencia y polimorfismo, en definitiva orientación a objetos.

5.1.5 Características interesantes

5.1.5.1 Extensibilidad desde el diseño

La característica a destacar de este sistema, y por eso ha sido examinado, es que el objetivo que se perseguía desde el principio de su construcción era la extensibilidad y que ésta abarcase tanto a los datos del usuario como a la gestión que se realiza de los mismos. Es destacable el hecho de que la extensibilidad de la gestión de los datos sea obtenida mediante un descriptor de relaciones extensible así como mediante unas interfaces genéricas que deben ser proporcionadas por todos los métodos de almacenamiento o caminos de acceso que deseen incorporarse al sistema. También conviene destacar la idea de un conjunto de servicios comunes que podrán ser empleados por todas las extensiones, de forma que cada extensión no tenga que implementar dichos servicios.

5.2 POSTGRES

Es un SGBD objeto-relacional desarrollado bajo la dirección de M. Stonebraker en la Universidad de California (Berkeley), y que proporciona extensibilidad para tipos de datos, operadores y métodos de acceso. En 1996 cambia su nombre por el de PostgresSQL[2] tras haber sustituido el lenguaje de consulta inicial (POSTQUEL) por SQL. Es el sucesor del SGBD relacional INGRES (POST inGRES) y el predecesor del sistema Illustra.

5.2.1 Arquitectura

La arquitectura de POSTGRES [SR86] emplea un simple modelo cliente-servidor "proceso-por-usuario". Una sesión POSTGRES consta de tres procesos UNIX cooperantes: el supervisor (*postmaster*), la aplicación del usuario (*front-end*) y uno o más servidores de bases de datos (*back-end*).

El supervisor gestiona una colección de bases de datos sobre un simple *host* (instalación). Las aplicaciones (*front-end*) que desean acceder a una base de datos dentro de una instalación, realizan mediante la librería *libpq* su petición al supervisor, que es el que se encarga de crear un nuevo proceso servidor (*back-end*) y conectarlo con la aplicación. A partir de este momento ambos se comunicarán sin intervención del supervisor. La librería *libpq* permite a un simple *front-end* hacer múltiples conexiones a procesos *back-end*. Sin embargo, el *front-end* es un proceso que no soporta múltiples hilos, con lo que no son soportadas conexiones multihilo entre ambos. Esto implica que el *back-end* y el supervisor siempre se ejecutan en la misma máquina, mientras que el *front-end* puede ejecutarse en cualquier lugar.

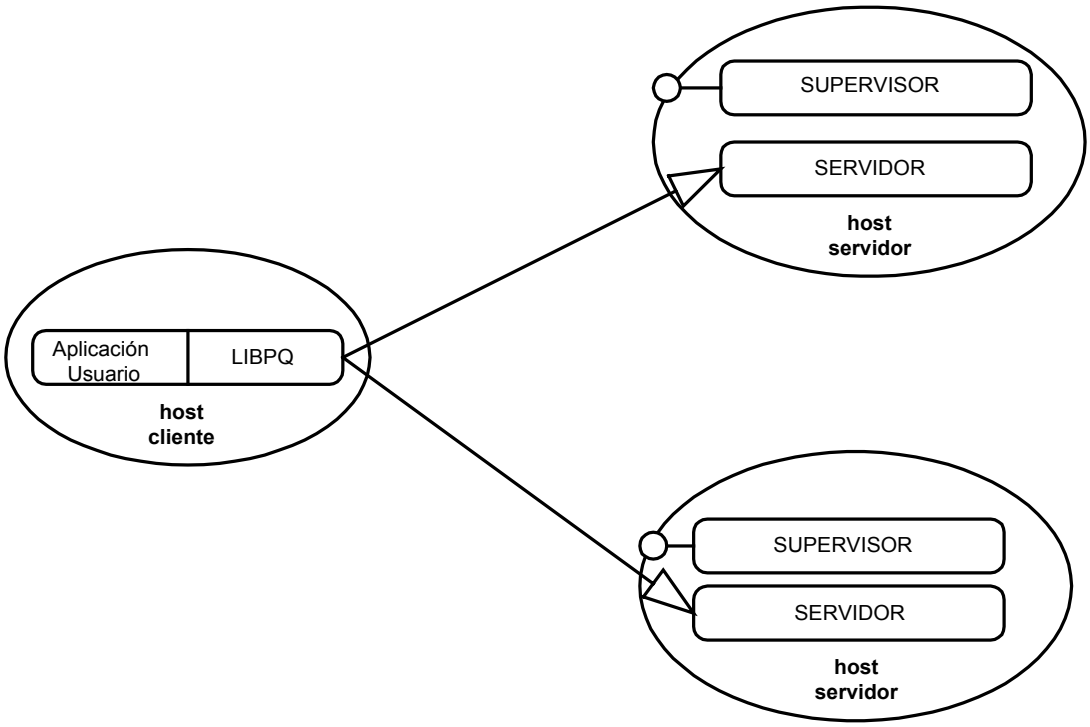


Figura 5.2. Arquitectura de POSTGRES

[2] En este documento se seguirá utilizando el nombre de POSTGRES

5.2.2 Características del modelo

El modelo de datos de POSTGRES está basado en el modelo relacional, pero proporciona objetos (una clase[3] es una tabla, una fila es una instancia y una columna es un atributo), identificadores de objetos (único para cada instancia), objetos compuestos, herencia múltiple, sobrecarga de funciones o métodos, y versiones. Incluye triggers y un potente sistema de reglas.

Clases

La noción fundamental en POSTGRES es la de *clase*. Una clase es una colección con nombre de instancias de objetos, donde cada instancia tiene una colección de atributos con nombre y tipo específico. El modelo distingue tres categorías de clases: *reales* (instancias que se almacenan realmente), *derivadas* y *versiones*. Las clases serán agrupadas en bases de datos.

Tipos

Los tipos son clasificados en tipos base (tipos abstractos de datos) y tipos compuestos. Los tipos base pueden ser predefinidos (compilados en el sistema) y definidos por el usuario (incluso en tiempo de ejecución). Un tipo compuesto es creado siempre que el usuario crea una clase.

Funciones

POSTGRES soporta operadores además de funciones en C y funciones en SQL (antes POSTQUEL). Los argumentos de las funciones pueden ser tipos base o compuestos o una combinación de ambos, al igual que el valor devuelto. Las funciones C pueden definirse a POSTGRES mientras el sistema se está ejecutando y son cargadas dinámicamente cuando son requeridas durante la ejecución de la consulta.

5.2.3 Incorporación de nuevos métodos de acceso

Básicamente un método de acceso POSTGRES es una colección de trece funciones C que hacen operaciones a nivel de registro, tales como traer el siguiente registro, insertar un nuevo registro, eliminarlo, etc. Para añadir un nuevo método de acceso, lo que un usuario debe hacer es proporcionar implementaciones para cada una de estas funciones y crear una colección de entradas en el catálogo del sistema [Sto86].

5.2.4 Extensibilidad dirigida por catálogos

En el catálogo de POSTGRES además de guardar la información sobre las tablas, y columnas que constituyen la base de datos como hacen los SGBDR tradicionales, se almacena mucha más información como son los tipos, las funciones, los operadores y los métodos de acceso [Aok91]. Esta información, que aparece a los usuarios en forma de clases, puede ser modificada por el usuario permitiendo así la extensibilidad del sistema. El optimizador y el procesador de consultas son conducidos por tablas de forma que todas las extensiones definidas por el usuario y empleadas en una consulta pueden ser cargadas en un proceso servidor de POSTGRES en tiempo de ejecución. El usuario puede especificar un fichero de código objeto que implementa un nuevo tipo o

[3] De hecho en la nomenclatura POSTGRES tabla y clase, fila e instancia, columna y atributo se emplean indistintamente

función y POSTGRES lo cargará dinámicamente. El mecanismo de carga dinámica empleada será el del sistema operativo subyacente.

5.2.5 Interfaces para diferentes lenguajes de programación

POSTGRES proporciona APIs para lenguajes de programación como C (*libpq*) y C++ (*libpq++*), así como interfaces ODBC y JDBC. Permite también la ejecución de SQL inmerso en la aplicación.

5.2.6 Crítica

5.2.6.1 Ampliación artificial del modelo relacional

El modelo de datos empleado es una ampliación del modelo relacional para dotar a éste de determinadas características de la orientación a objetos como la herencia. Sin embargo, no introduce otras características como la encapsulación.

5.2.6.2 Extensibilidad basada en el modelo relacional

POSTGRES es un SGBD extensible, pero está extensibilidad está basada en la construcción del SGBD sobre un modelo relacional, de forma que cada vez que se quiere incorporar por ejemplo un método de acceso, es necesario acceder a una cantidad considerable de tablas cuyas relaciones no son muy evidentes.

5.2.7 Características interesantes

5.2.7.1 Extensibilidad

La característica a destacar de POSTGRES es por supuesto su extensibilidad, que facilita la incorporación de nuevos métodos de acceso, operadores y tipos de datos en tiempo de ejecución.

5.2.7.2 Procesador y optimizador de consultas dirigidos por tablas

El hecho de que tanto el optimizador como el procesador de consultas sean dirigidos por tablas y por tanto no sea necesario para el sistema parar y recompilar cada vez que se añade, por ejemplo, un nuevo método de acceso, convierte a POSTGRES en un importante proyecto cara a la investigación en bases de datos.

5.3 ObjectDRIVER[4]

ObjectDRIVER [Obj00b] es una herramienta *middleware* que permite crear un SGBDOO compatible ODMG sobre un SGBDR. Cualquier acceso a la base de datos subyacente se realiza por medio de la capa de ODBC, lo que hace que ObjectDRIVER trabaje sobre una gran cantidad de SGBDR comerciales.

Para construir una aplicación con ObjectDRIVER son necesarios tres ficheros:

- El fichero relacional que contiene físicamente las tablas. Es un fichero de base de datos de Microsoft Access que contiene las tablas necesarias.
- El fichero de texto con la descripción del esquema relacional.
- El fichero de texto con la descripción del esquema de objetos. Este fichero contiene las correspondencias entre las clases y las tablas del modelo relacional (*mapping*).

El concepto clave de ObjectDRIVER es la noción de *mapping*, que se basa en definir una jerarquía de clases sobre tablas relacionales empleando un lenguaje de correspondencias genérico. Una vez que una jerarquía de clases ha sido definida, los "aspectos relacionales" de los datos están ocultos al usuario. Se pueden ver los datos como si estuvieran en un SGBDOO compatible ODMG.

Ofrece una interfaz *OQL Query* para las consultas, y *bindings* para Java y C++, que proporcionan una manera de crear nuevas aplicaciones orientadas a objetos compatibles ODMG sin tener que modificar el SGBD subyacente.

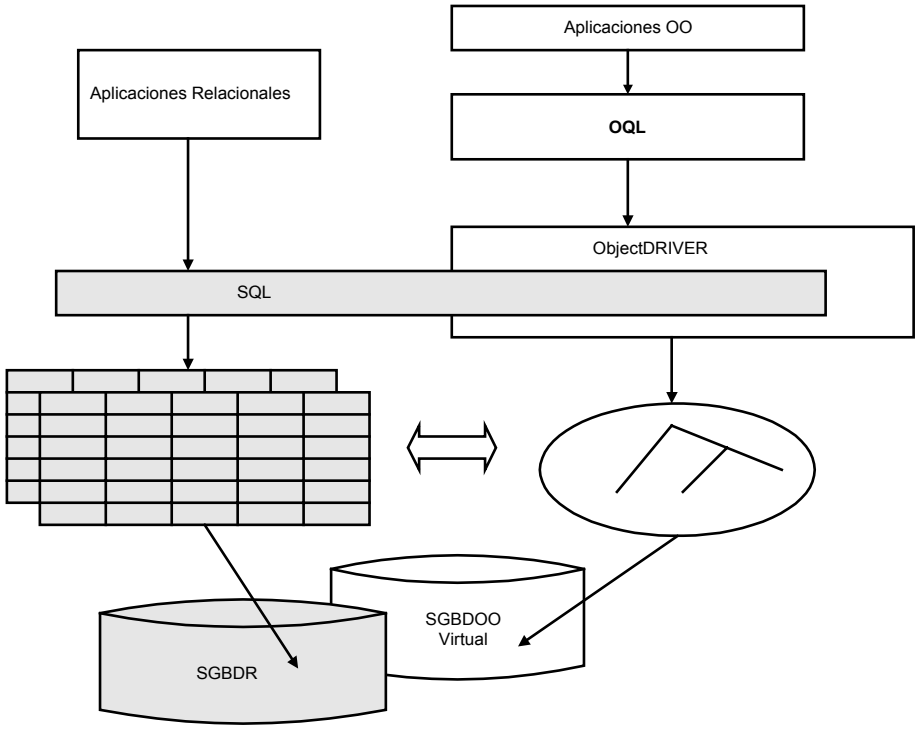


Figura 5.3. Esquema de ObjectDRIVER

[4] Versión 1.1

ObjectDRIVER proporciona un API ObjectDRIVER y un API ODMG para trabajar desde un lenguaje de programación. Si lo desea el usuario podrá trabajar también en modo consola pero con limitaciones en las funcionalidades.

5.3.1 Arquitectura

El sistema necesita cuatro ficheros (para almacenar parámetros del gestor) que deben estar situados en un directorio concreto, y que deben ser protegidos por el sistema operativo para prevenir accesos no autorizados a la base de datos.

- **Fichero de opciones del sistema.** Este fichero contiene tres partes: *opciones generales del sistema* (ej. lenguaje a emplear en los mensajes), *opciones de objetos* (ej. considerar o no las modificaciones realizadas en memoria sin finalizar una transacción a la hora de realizar una consulta) y *opciones relacionales* (ej. tamaño del buffer dónde se construyen las consultas)
- **Fichero de mensajes.** Contiene la descripción de todos los mensajes que pueden ser devueltos al usuario.
- **Fichero de passwords.** Contiene las definiciones de los usuarios que pueden conectarse a ObjectDRIVER. Si este fichero no existe nadie puede conectarse.
- **Fichero de Esquema**. Contiene las definiciones de la base de datos: nombre de la base de datos de objetos, nombre del fichero de texto donde se define esa base de datos, nombre de la base de datos relacional y fichero de texto donde se define la base de datos relacional. Si la base de datos relacional ya existe, ignora el contenido especificado en el fichero que contiene la definición de la base de datos relacional. Si no, la crea. A continuación hace lo mismo con la base de datos de objetos. De esta forma una misma base de datos relacional puede emplearse para traducir varias bases de datos de objetos, por ejemplo, cuando de una misma base de datos relacional se necesitan diferentes vistas de objetos.

5.3.2 Persistencia por alcance

Las clases cuyos objetos se almacenen en la base de datos deben ser descritas en el esquema de objetos asociado con la aplicación, y además deben ser identificadas como capaces de ser persistentes. Una clase es capaz de ser persistente cuando extiende la clase *fr.cermics.ObjectDRIVER.ObjectDRIVERObject* o implementa la interfaz *fr.cermics.ObjectDRIVER.persistentObject*.

En el *binding* para Java la persistencia es por alcance, con lo que un objeto Java transitorio referenciado por un objeto Java persistente será persistente si es una instancia de una clase capaz de ser persistente, y los dos objetos no están ligados por un campo transitorio. Y tal y como propone el estándar, dentro de una clase persistente puede haber campos transitorios.

Por otro lado, la recuperación de un objeto de la base de datos no implica la recuperación de todos los objetos asociados a éste. Cuando se recupera un objeto, todos los objetos relacionados con éste son representados en la aplicación cliente con *proxies*. Cuando se accede a un *proxy* el objeto correspondiente es cargado en la memoria local.

5.3.3 Transacciones basadas en el servicio de transacciones del SGBDR subyacente

Las bases de datos de ObjectDRIVER sólo pueden ser accedidas mediante transacciones, pero el modelo transaccional no soporta transacciones anidadas. Las

transacciones pueden emplearse tanto en modo consola como con cualquiera de los dos APIs antes mencionados.

Es conveniente tener en cuenta que el sistema transaccional de ObjectDRIVER es un envoltorio sobre el sistema de transacciones del gestor relacional base, con lo que la mayoría de los algoritmos que permiten verificar las propiedades ACID, son ejecutados por el sistema de transacciones subyacente. Esto implica que el comportamiento de las transacciones del ObjectDRIVER puede ser diferente dependiendo del SGBDR seleccionado.

5.3.4 Lenguaje de consulta OQL

Soporta el OQL de ODMG pero con algunas limitaciones. Sólo pueden realizarse consultas sobre las extensiones de las clases (no sobre expresiones que incluyen otras consultas u objetos nombrados), y no soporta las cláusulas *Group By* y *Having*, ni funciones predicado como *count*, *min* o *max*.

Las consultas al igual que las transacciones pueden realizarse en modo consola y en un lenguaje de programación.

5.3.5 Crítica

5.3.5.1 Traducción explícita del esquema por el programador

Como se ha mencionado anteriormente un envoltorio pretende dar la ilusión al usuario de que está trabajando con una base de datos orientada a objetos pero internamente las clases son traducidas al modelo relacional. En este gestor dicha traducción debe ser realizada explícitamente por el programador. El programador, debe diseñar la base de datos relacional e indicar para cada atributo de la clase el campo de la tabla en la que se almacenará. Esto se convierte en una tarea laboriosa y tediosa. Y aunque sería lógico que el programador pudiese modificar/crear un esquema óptimo, la herramienta podría proporcionar uno por defecto, ya que existen ciertas asignaciones muy obvias.

5.3.5.2 Diferente sistema de tipos

El lenguaje Java y los esquemas ObjectDRIVER tienen diferentes sistemas de tipos, de forma que si un usuario quiere traducir una clase Java a la declaración de un esquema de objetos en ObjectDRIVER debe escribir el esquema reemplazando los tipos Java por los tipos ObjectDRIVER adecuados.

5.3.5.3 Carencia de flexibilidad/ extensibilidad

Este gestor no considera la posibilidad de extender su funcionalidad, a pesar de ser relacional, ni siquiera permitiendo la incorporación de nuevos métodos de acceso.

5.3.5.4 No uniformidad en el tratamiento de objetos

ObjectDRIVER distingue dos clases de objetos: objetos locales (eventualmente persistentes) y *proxies* de objetos persistentes. Estos últimos deben ser cargados explícitamente (método *getObject*) en memoria local para recibir mensajes o ser inspeccionados. Esto obliga a implementar métodos que lo primero que hagan sea recuperar el objeto con el fin de poder modificar sus atributos, lo que dista mucho de ser una recuperación de objetos transparente, y obscurece bastante el código.

5.3.5.5 Falta de seguridad en el sistema

El acceso al sistema está limitado a los usuarios definidos, pero sin embargo, la descripción de los usuarios aparece definida en un fichero que debe ser protegido de forma externa y manual en el momento de la instalación desde el sistema operativo, si no cualquier usuario podría modificar esa información.

5.3.6 Características interesantes

5.3.6.1 Compatible con el estándar ODMG

ObjectDRIVER proporciona un API ODMG que permite construir aplicaciones compatibles con el estándar, facilitando así la portabilidad de la aplicación hacia otros gestores compatibles ODMG.

5.3.6.2 Cierta configurabilidad

No proporciona extensibilidad, pero si proporciona ciertas opciones para configurar el sistema. Permite así por ejemplo, especificar el tamaño del buffer donde se construyen las consultas, o indicar al sistema si en las consultas se deben considerar los objetos actualizados en la transacción en curso o no.

5.4 ObjectFile

Es un motor comercial [Obj99], suministrado en forma de código fuente, que es empleado para proporcionar persistencia de objetos a cualquier aplicación C++ mediante el almacenamiento de los mismos en ficheros tradicionales. La entrada y la salida se realizan empleando funciones estándar de C o funciones del sistema operativo nativo. Está escrito en C++ estándar y emplea la *Standard Template Library*, por lo que se exige la existencia de un compilador que soporte estas características.

5.4.1 Arquitectura

La arquitectura está constituida por tres clases:

- **Opersist**, superclase que da a los objetos las características de persistencia. Estas características pueden ser modificadas sobreescribiendo las funciones virtuales. Sin embargo, se pueden emplear objetos que se pueden hacer persistentes sin necesidad de que deriven de esta clase, siempre que estén incrustados en otros objetos y no necesite accederse a ellos directamente.
- **Ofile**, proporciona las funcionalidades para la gestión de ficheros. Se pueden guardar o recuperar objetos *Opersist* desde el *Ofile*.
- **Oiterator**, que permite iterar sobre los objetos de cualquier clase y sus subclases, en un fichero.

5.4.2 Características básicas

La principal característica soportada por este sistema es la persistencia. Sin embargo, ObjectFile al no utilizar un precompilador para convertir el esquema de la base de datos en código C++, obliga al programador a escribir el código para leer y guardar los datos. Para ello los objetos derivados de *Opersist* tienen un constructor que toma como parámetro de entrada un *stream*, cuyas funciones serán llamadas para leer los datos. Para escribir los datos hay una función virtual que toma un *stream* como salida.

En cuanto a la indexación, realiza una indexación automática por el identificador de los objetos, pero no permite definir nuevos índices. De hecho, sostiene que es mucho más fácil que cada usuario se construya sus propios índices con sus propios requerimientos de espacio, memoria, etc.

Soporta también un mantenimiento opcional de una caché de objetos, siendo el programador el encargado de decir cuantos objetos pueden estar en memoria a la vez.

5.4.3 Crítica

5.4.3.1 Carencia de funcionalidades básicas

Este motor está más cercano a un mecanismo para proporcionar persistencia a un lenguaje de programación, como es C++, que a un gestor de objetos, ya que carece del resto de funcionalidades básicas de un gestor: gestión de concurrencia, posibilidad de consultas, etc.

5.4.3.2 Ligado al lenguaje de programación C++

ObjectFile está pensado única y exclusivamente para objetos creados con el lenguaje de programación C++, no facilitando la interoperabilidad con objetos creados desde otros lenguajes orientados a objetos como, por ejemplo, Java.

5.4.3.3 Persistencia no transparente

Desde el punto de vista de la persistencia, la forma en la que ObjectFile la proporciona no es en absoluto transparente, al ser el propio usuario el que tiene que implementar los métodos que permitan el almacenamiento y la recuperación de los objetos.

5.4.4 Características interesantes

5.4.4.1 Fomenta el empleo de técnicas de indexación adecuadas a las necesidades de la aplicación

Realmente este sistema aporta poco al estudio que se está realizando, aunque es aquí incluido por la insistencia de los desarrolladores en la idea de que las técnicas de indexación deben poder incorporarse en función de las necesidades de la aplicación.

5.5 DarmStadt

El proyecto DarmStadt [SPS+90], desarrollado en la Universidad de Darmstadt (Alemania), representa un gestor de objetos también conocido como DASDBS (*DarmStadt Data Base System*) que se basa en un núcleo común sobre el que se superponen capas de soporte para aplicaciones específicas, de forma que un SGBDespecífico para una aplicación se obtiene enlazando el núcleo con el *front-end* correspondiente.

5.5.1 Arquitectura

Las funcionalidades del núcleo fueron asignadas partiendo de la idea de que el núcleo debería proporcionar requerimientos comunes que son necesarios para aplicaciones avanzadas, y por otro lado, acudiendo a un criterio de rendimiento, debería soportar características que pueden ser implementadas eficientemente sólo dentro del núcleo.

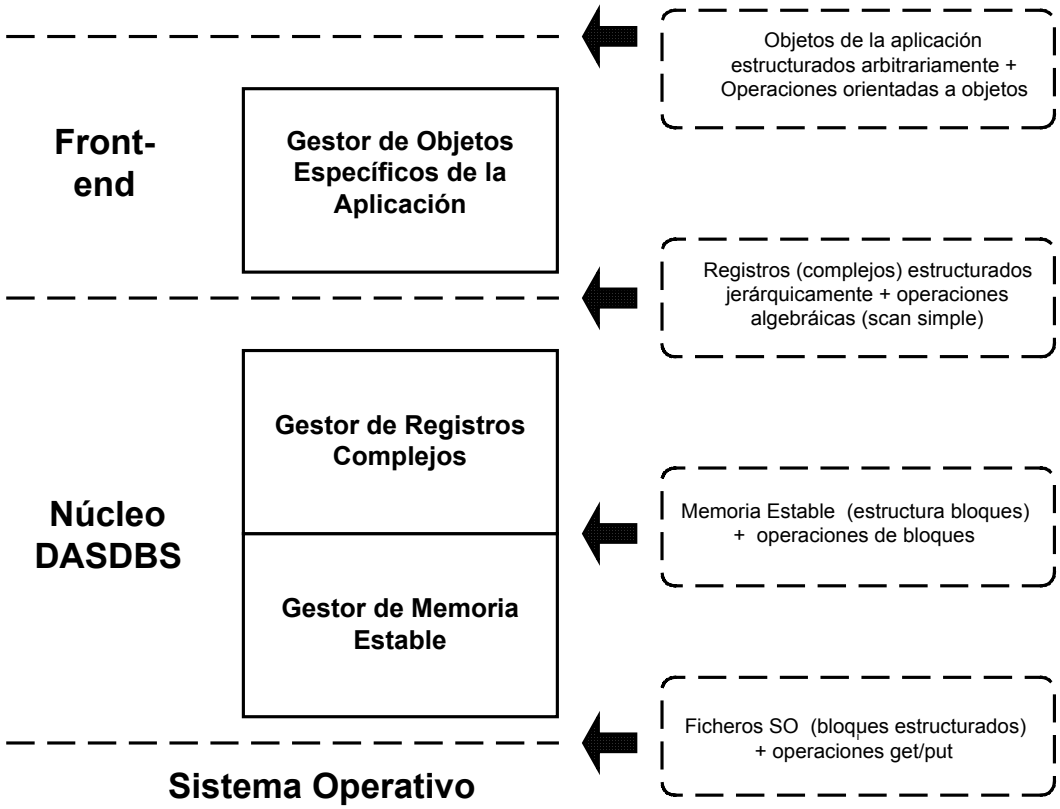


Figura 5.4. Arquitectura de tres niveles de DASDBS

Como se aprecia en el dibujo (Figura 5.4) para un tipo de aplicación concreto la arquitectura de DarmStadt distingue tres niveles [PSS+87]:

- **Gestor de objetos específicos de la aplicación,** que opera sobre conjuntos de "registros complejos" estructurados jerárquicamente.
- **Gestor de registros complejos,** manipula los "registros complejos" como "objetos complejos" primitivos a nivel del núcleo.

- **Gestor de memoria estable,** que es una capa orientada a conjuntos de páginas, que incluye el *buffering* y soporta la gestión de transacciones clásica de un SGBD.

5.5.2 Modelo de datos basado en registros complejos

El modelo de datos está basado en "registros complejos" que son la implementación interna de las tuplas en el modelo relacional con atributos cuyos valores pueden ser relaciones. Este modelo recibe el nombre de modelo relacional anidado NF^2 (no primera forma normal) [SS86]. La estructura de datos es obtenida por una secuencia alternativa de constructores tupla y conjunto. El único tipo de datos permitido es la relación. No hay listas, ni vectores ni ninguna otra estructura de datos.

5.5.3 Características del núcleo

El concepto clave para todas las operaciones del núcleo es la orientación a conjuntos y la propiedad *single scan*, que permite que el procesamiento de todas las operaciones del núcleo se realice accediendo a todas las tuplas implicadas sólo una vez.

El núcleo ofrece operaciones orientadas a conjuntos, es decir, cada operación recupera o manipula un conjunto de registros complejos, en vez de un registro cada vez. Además, el *clustering* físico es expresado en términos de registros complejos que resultan desde el diseño físico de la base de datos, y el servidor de páginas fue diseñado también con una interfaz orientada a conjuntos. Si el sistema operativo proporcionará una interfaz de entrada-salida orientada a conjuntos se optimizarían los accesos a disco lo que supondría un beneficio adicional para el sistema.

El núcleo no ofrece soporte para tipos de datos definidos por el usuario ni tampoco para los tipos de datos estándar soportados por la mayoría de los lenguajes de programación. Únicamente soporta el tipo *byte string*.

Las funciones del núcleo si incluyen la gestión de transacciones, pero no así la gestión de referencias entre objetos ni el control de la información redundante.

5.5.4 Comunicación entre el *front-end* y el núcleo mediante *object buffer*

Los resultados de las consultas, así como las entradas para actualizaciones de los datos requieren la transferencia de registros complejos entre el núcleo y la siguiente capa (*front-end*). Con el fin de evitar un cuello de botella en este traslado de información (que no sea registro a registro) se necesitaría una combinación arbitraria de constructores conjunto y registro. Como la mayoría de los lenguajes no permiten este tipo de construcciones, se incorpora un tipo abstracto de datos *object buffer* en el núcleo. Cada *object buffer* puede contener conjuntos de registros complejos del mismo tipo. Estos objetos no son compartidos entre transacciones.

5.5.5 Crítica

5.5.5.1 No facilita la portabilidad

A diferencia de la tendencia actual a la realización de sistemas portables, en este sistema entre sus objetivos no se plantea el facilitar la portabilidad del sistema a diferentes entornos. Y esto queda patente al ser el lenguaje inicial de implementación el Pascal y con posterioridad el lenguaje C.

5.5.5.2 Desadaptación de impedancias entre los diferentes niveles de la arquitectura

Una de las principales ventajas atribuibles a esta arquitectura es la construcción de un núcleo que ofrece operaciones orientadas a conjuntos. Sin embargo, se observa que al no emplear ni el sistema operativo ni los lenguajes de programación ese mismo paradigma se producen problemas de desadaptación de impedancias entre ellos, lo que obliga a construir artefactos software como el *object buffer* para solucionarlos.

5.5.5.3 Modelo relacional anidado

El modelo empleado para representar objetos complejos es el modelo relacional anidado NF^2 que además de suponer una descomposición artificial de los datos dificulta entre otras cosas las relaciones de cardinalidad n-m.

5.5.5.4 Descarga demasiada funcionalidad en el *front-end*

El hecho de que el núcleo no reconozca tipos definidos por el usuario y ni siquiera tipos básicos estándar, hace que las operaciones sobre estos tipos tengan que ser implementadas a nivel de *front-end*, al igual que los mecanismos de indexación que no son en absoluto considerados en el núcleo. Esto ocasiona una transformación para adaptar las cadenas almacenadas en el núcleo a los tipos de datos correspondientes, lo que se traduce obviamente en una disminución del rendimiento del sistema.

5.5.6 Características interesantes

5.5.6.1 Núcleo básico

Cara a conseguir un sistema medianamente flexible ante diferentes requerimientos de las aplicaciones el hecho de contar con un núcleo básico que proporcione la funcionalidad mínima común a las diferentes clases de aplicaciones representa una posible solución que evita disponer de un SGBD enorme.

5.5.6.2 Núcleo orientado a conjuntos de registros complejos

La aproximación llevada a cabo por DarmStadt para aumentar el rendimiento del sistema al diseñar el núcleo de forma que pueda trabajar sobre conjuntos de registros complejos, es decir, sobre las estructuras en las que el modelo relacional implementa las tuplas, es buena, aunque se vería beneficiada por la existencia de un sistema operativo que empleara ese mismo paradigma orientado a conjuntos.

5.6 StreamStore

StreamStore [Str99] es un gestor de almacenamiento de bajo nivel escrito en Java que proporciona clases Java para almacenar, recuperar e indexar cualquier clase de objetos. Consta de dos subsistemas que pueden emplearse independientemente: *subsistema de almacenamiento* y *subsistema B-Tree* (para la indexación).

5.6.1 Arquitectura

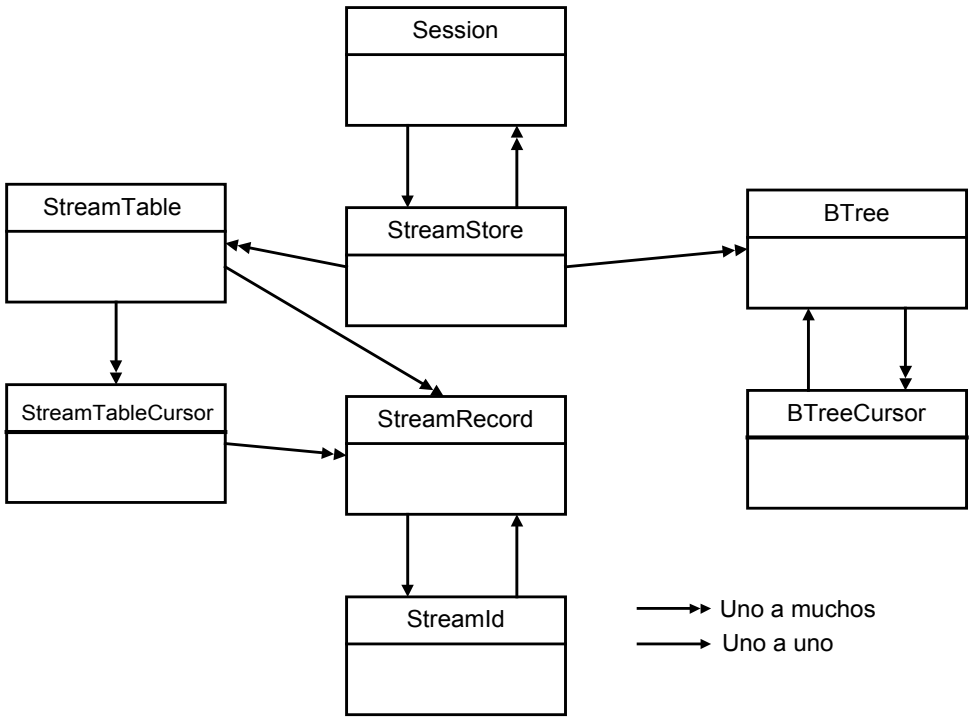


Figura 5.5. Estructura de las clases de StreamStore

Subsistema de almacenamiento

El subsistema de almacenamiento está constituido por las clases siguientes: *StreamStore*, *StreamTable*, *StreamRecord*, *StreamTableCursor* y *StreamId*.

El almacenamiento de objetos en StreamStore se realiza guardando los objetos en un *StreamRecord*. Cada objeto *StreamRecord* tiene asociado un identificador, no reutilizable, que es una instancia de la clase *StreamId* (aunque los identificadores también pueden asignarse empleando un Btree). Este identificador puede emplearse para acceder al *StreamRecord* correspondiente, y también se puede emplear para asociar un registro con otro. Los *StreamRecords* serán almacenados en *StreamTables*. Los *StreamTables* además, permiten la creación de unos objetos transitorios (*StreamTableCursor*) que nos permiten recorrer todos los *StreamRecords* de una *StreamTable.* El objeto *StreamStore* representa y controla la base de datos completa. Mantiene una colección de todas las *StreamTables* y de todos los *Btrees* existentes.

Subsistema Btree

El subsistema Btree está constituido por las clases *Btree* y *BtreeCursor*. Un objeto *Btree* se emplea para indexar un StreamRecord o algún otro objeto. Cuando se crea un *Btree* es posible configurar tanto el tamaño de los nodos como el número de nodos que se desea almacenar en la caché. Un *BtreeCursor* es un objeto transitorio que permite el

recorrido del índice completo de forma ascendente o descendente, entre un rango de claves, etc.

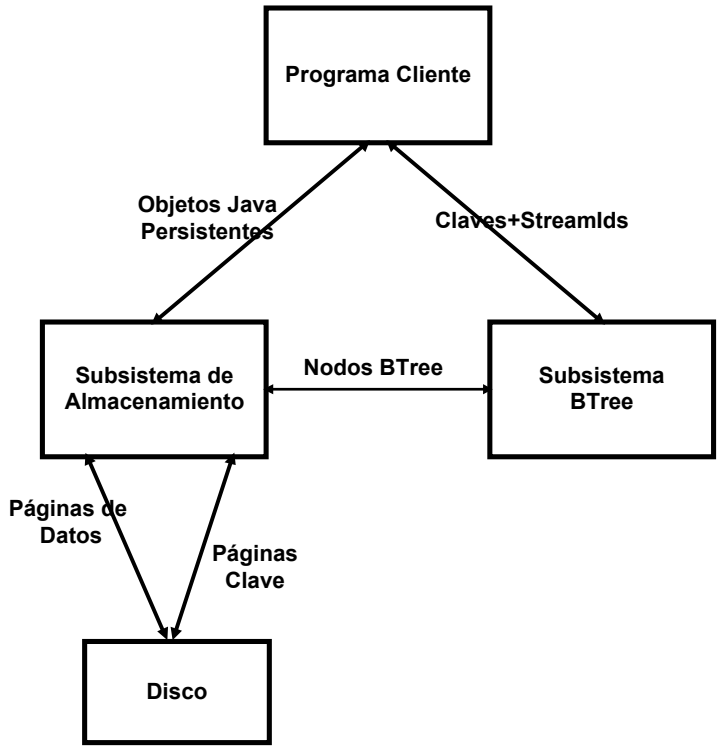


Figura 5.6. Subsistemas de StreamStore

5.6.2 Persistencia por llamada explícita

La persistencia en StreamStore se consigue mediante invocación explícita del método *save* de la interfaz *RecordStreamable*[5]. Es por tanto necesario que todas aquellas clases cuyas instancias puedan ser persistentes implementen dicha interfaz. StreamStore permite también, si se desea, emplear la serialización de Java (*StreamRecord::getOutputStream()*). La eliminación de las instancias es también por borrado explícito.

5.6.3 Diferentes métodos para la recuperación de la información

StreamStore no permite asociar nombres a los objetos. La recuperación de los objetos almacenados en la base de datos se puede realizar mediante:

- Un *StreamTableCursor* que permite recorrer todos los *StreamRecords* de una *StreamTable.*
- El identificador (*StreamId*) del *StreamRecord.*
- Una clave. Se puede acceder a un *StreamRecord* buscando la clave en el Btree y empleando entonces el *StreamId* asociado para recuperar el *StreamRecord*.

[5] Aunque incluye un paquete de utilidades (*com.bluestream.ssutil*) que facilita ligeramente esta tarea

5.6.4 Indexación basada en B-Tree

Los subsistemas de StreamStore pueden emplearse conjuntamente (los objetos son almacenados en el subsistema de almacenamiento y son indexados empleando Btree), o independientemente (si no interesa la indexación se puede emplear el almacenamiento de objetos en el subsistema de almacenamiento o viceversa).

Otra posibilidad es usar el sistema de almacenamiento para los datos de usuario, pero construyendo unos índices propios que pueden persistir empleando el sistema de almacenamiento.

5.6.5 Concurrencia asociada a sesiones

En StreamStore aparece también el concepto de sesión. El objetivo de la sesión es crear un contexto de transacciones para asociar operaciones de bases de datos con él. Así, se debe crear una sesión para cada StreamStore sobre el que se desee trabajar, y también para cada vista concurrente que se desee del mismo. Para tener esta vista concurrente será necesario tener un hilo separado. Por lo tanto, una aplicación puede tener muchos hilos que acceden concurrentemente al mismo StreamStore, y cada hilo, por medio del espacio de transacciones de la sesión, puede acceder a los mismos datos concurrentemente de acuerdo a las reglas de conflicto de las transacciones.

Las transacciones en StreamStore son ACID, proporcionan un aislamiento nivel 3 (*Lectura Repetible*) que garantiza que una transacción no puede ver los cambios hechos por otra transacción en valores que ya haya leído, y además, permite una estrategia de concurrencia optimista basada en el número de versión asociado con cada *StreamRecord*.

5.6.6 Crítica

5.6.6.1 Falta de transparencia para conseguir la persistencia

Para conseguir la persistencia es necesario añadir a cada clase dos métodos que realizan el almacenamiento y posterior carga de los objetos, no siendo en absoluto un proceso transparente para el usuario. Se necesita en primer lugar crear un *StreamStore*, después una *StreamTable* y posteriormente, hay que crear un *StreamRecord* para cada objeto que desea hacerse persistente, y por supuesto, invocar el método correspondiente para guardar el objeto.

5.6.6.2 Mantenimiento manual de los índices

El mantenimiento de los índices es totalmente responsabilidad del programador, es decir, es el programador el encargado de insertar, borrar y actualizar en la estructura índice.

5.6.6.3 No soporta consultas declarativas

No proporciona soporte para interrogaciones declarativas a la base de datos mediante un lenguaje estilo OQL.

5.6.6.4 Filosofía relacional

StreamStore no es un adaptador objeto-relacional propiamente, ya que finalmente los objetos no se almacenan en tablas, sin embargo si se adecua muy bien al modelo relacional. Así, las clases Java se traducen en *StreamTables* de forma que todos los objetos de esa clase sean almacenados en el mismo *StreamTable*, las relaciones muchas

a muchas se traducen en una *StreamTable* de intersección, los *StreamIds* serían las claves primarias, etc.

5.6.7 Características interesantes

5.6.7.1 Facilita la portabilidad

Es un gestor escrito en Java y que por tanto, facilita su utilización en diferentes plataformas.

5.6.7.2 Indexación extensible y sobre cualquier objeto

Este gestor permite que la indexación se realice sobre cualquier objeto definido por el usuario sin más que implementar los métodos de una determinada interfaz (*interface Key)*. También contempla la posibilidad de incorporar nuevas técnicas de indexación que se harían persistentes acudiendo al subsistema de almacenamiento.

5.6.7.3 Módulos que pueden emplearse independientemente

La estructuración del sistema en dos módulos que pueden actuar conjunta o independientemente es realmente una buena idea. Sin embargo, sería necesario un nivel de abstracción mayor en el caso, por ejemplo de la indexación, para permitir nuevos subsistemas con otras técnicas de indexación diferentes, no solamente basadas en B-Tree.

5.7 Jeevan

Es un sencillo gestor de almacenamiento persistente [Jee99] para un único usuario sobre la plataforma Java, desarrollado por *W³APPS*. Este gestor es un API, que permite al usuario desarrollar aplicaciones empleando únicamente el *Java Development Kit*, sin que sean por tanto necesarios ni precompiladores ni *third party libraries*.

5.7.1 Arquitectura

Jeevan[6] consta de un motor de base de datos, cuatro clases y dos interfaces. Las clases son: *JeevanServer*, *Storable*, *RecordSet* y *OQLQuery*. Las interfaces son: *Database* y *ObjectSet*.

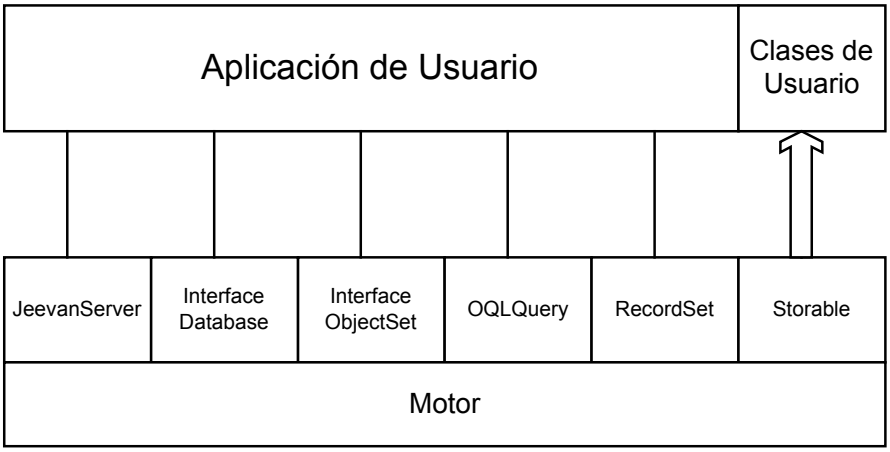


Figura 5.7. Arquitectura de Jeevan

La clase *JeevanServer* gestiona una o más bases de datos en el sistema; su principal misión es crear bases de datos o abrir bases de datos existentes. Para que una clase definida por el usuario sea persistente debe extender la clase *Storable* o bien implementar la interfaz *java.io.Serializable.* Cuando en una consulta no se recuperan todos los campos del objeto se crea un *RecordSet*, que contiene uno o más registros, y cada registro se forma con uno o más campos de la consulta solicitada. Esta clase proporciona métodos para recuperar un campo individual del registro y para moverse entre la colección de registros. Para representar consultas dinámicas se emplea la clase *OQLQuery*.

La interfaz *Database* define métodos para mantener (definir, manipular y de utilidad) la base de datos. Cuando se ejecuta una consulta se crea un *ObjectSet* que contiene los objetos que satisfacen el criterio de selección. Este proporciona métodos para manipular los objetos resultado de la consulta.

5.7.2 Persistencia por llamada explícita

La persistencia de un objeto se consigue en Jeevan invocando explícitamente el método *insert* de la interfaz *Database,* y siempre que la clase a la que pertenece ese objeto cumpla una de las dos condiciones siguientes:

- Extiende la clase *Storable*, con lo que se reserva un área de almacenamiento dónde serán almacenados los objetos de esa clase, y además se les asociará un identificador único. Este identificador es un entero largo y no reutilizable.

[6] La versión examinada ha sido la 2.0

- Implementa la interfaz *java.io.Serializable*. En este caso no se asigna un área de almacenamiento separada y tampoco se le asocia un identificador único. Los objetos de las clases que implementan esta *interfaz* son empleados como objetos incrustados, es decir, existen dentro del ámbito de su objeto contenedor.

5.7.3 Declaración explícita de las clases persistentes

El sistema de tipos de Jeevan es el del lenguaje Java, pero para la definición del esquema de la base de datos emplea el método *defineClass* de la interfaz *Database*, a diferencia de lo propuesto por el estándar ODMG (la definición de las clases es la del propio lenguaje).

Con relación a la definición del esquema se pueden destacar las siguientes características:

- Las clases cuyos objetos van a estar almacenados en la base de datos, así como los atributos empleados en las consultas o en los índices, deben ser definidos como públicos.
- No se permite la especificación de claves ni de relaciones.
- Las relaciones deben ser mantenidas por el propio programador. Y, el establecimiento de las relaciones puede realizarse de dos formas: mediante un atributo (o vector) que contiene el identificador del objeto de la otra clase, o bien sin hacer ninguna modificación en las clases a relacionar, mediante el método *relate* (interfaz *Database).*

5.7.4 Diferentes métodos para la recuperación de información

Este gestor para la recuperación de los datos no dispone de un lenguaje de consulta potente estilo OQL, si no que acude a una serie de métodos (de la interfaz *Database* mayoritariamente) que proporcionan las siguientes posibilidades:

- Obtener un objeto por su identificador (método *getObject*).
- Obtener un conjunto de objetos: bien todos los objetos de una clase, o bien todos los que cumplen una determinada condición (método *selectObjects*).
- Obtener todos los objetos de una jerarquía de clases. También se puede especificar un criterio de selección (método *selectObjectOfType*).
- Obtener determinados atributos del objeto (método *selectFields)*.

Para emplear criterios de selección dinámicos se debe emplear la clase OQLQuery.

5.7.5 Indexación sobre campos de tipo simple

Jeevan permite la creación y el borrado explícito de índices, pero siempre sobre campos de tipo simple (*int, long, float, double o string*). Permite también indexar sobre campos de un objeto incrustado. El índice definido en Jeevan para una clase es heredado por las clases hijas.

5.7.6 Control de versiones basado en el mecanismo de serialización de Java

Permite la evolución de esquemas (añadir atributos y métodos) basándose en el mecanismo de control de versiones proporcionado por el mecanismo de serialización de Java, para lo que hay que utilizar el programa *serialver* que viene con el JDK.

5.7.7 Crítica

5.7.7.1 Carencia de flexibilidad/extensibilidad

Este gestor proporciona un conjunto mínimo de servicios rígidamente establecidos, no permitiendo la incorporación de nuevas técnicas para ninguno de esos servicios, ni por supuesto la incorporación de los servicios de los que carece.

5.7.7.2 Indexación mínima no extensible

La indexación soportada por este gestor es básica, ya que únicamente permite indexar por campos de tipo primitivo. Tampoco proporciona ningún mecanismo para incorporar nuevas técnicas de indexación.

5.7.7.3 No es compatible con el estándar ODMG

Como se puede observar no es en ningún grado compatible con el API propuesto por el estándar ODMG para Java, lo que dificulta la portabilidad de la aplicación hacia otros gestores diferentes de Jeevan.

5.7.7.4 No uniformidad en el tratamiento de los objetos

Jeevan no realiza un tratamiento uniforme de los objetos, ya que distingue claramente dos tipos de objetos: los contenedores (extienden *Storable*) y los incrustados (implementan *java.io.Serializable*). Además, no comprueba que ambos tipos se empleen del modo adecuado, con lo que puede ocurrir que una clase que extiende la clase *Storable* tenga un atributo que además es de tipo *Storable*, lo que puede ocasionar que tengamos dos copias del objeto, una en su propia área de almacenamiento y otra incrustada en el objeto contenedor, con las posibles inconsistencias que esto puede acarrear.

5.7.7.5 Persistencia mediante serialización Java

El hecho de emplear la serialización como método para conseguir la persistencia impone lentitud si el volumen de datos es grande y además, no proporciona un medio confiable de almacenamiento, ya que si se produce un fallo en el sistema mientras se está escribiendo el objeto a disco se pierde el contenido del fichero.

5.7.7.6 No soporta concurrencia ni recuperación

Este gestor (al menos la versión revisada) no soporta el concepto de transacción, ni la posibilidad de recuperación ante cualquier fallo del sistema.

5.7.8 Características interesantes

5.7.8.1 Portabilidad

La principal característica a destacar de este gestor es que está construido en Java y que por lo tanto cualquier aplicación escrita en Java que emplee este gestor de almacenamiento puede portarse sin dificultad a otras plataformas.

5.7.8.2 Núcleo con funcionalidad básica

Este gestor presenta una gran cantidad de carencias, sin embargo, puede destacarse la idea del diseñador de construir un gestor que no ocupe demasiado en memoria y que proporcione una funcionalidad básica (en este caso demasiado) para una aplicación en Java.

5.7.8.3 Gestión automática de la extensión

Jeevan mantiene la extensión de las clases automáticamente, sin necesidad de que el usuario declare colecciones que contengan las instancias de las clases y establezca puntos de entrada (*roots*).

5.8 ObjectStore PSE y PSE Pro para Java

Son gestores de almacenamiento que proporcionan un medio de almacenamiento persistente para objetos Java (aunque también existen para objetos C++[7]) en aplicaciones de usuario simple.

PSE (*Persistent Storage Engine*) [PSE99] es una librería escrita en Java diseñada para el acceso a un volumen limitado de datos por usuario, como máximo 50MB. Soporta transacciones, y el acceso a objetos sin necesidad de leer la base de datos completa.

PSE Pro[8], además de las características del PSE, proporciona una mayor capacidad de almacenamiento, y soporta mejor concurrencia, múltiples sesiones, *interfaces collection* y clases que son compatibles con el Jdk 1.2, consultas e índices, y recuperación antes fallos del sistema. Incluye además un recolector de basura, y utilidades para mostrar información de los objetos en la base de datos y chequear las referencias en la misma.

5.8.1 Arquitectura

PSE Pro es una librería de Java que se ejecuta enteramente dentro del proceso de la máquina virtual de Java, y que no permite una distribución cliente-servidor.

Una base de datos PSE Pro consta de tres ficheros con extensiones: .odb, .odt y .odf.

5.8.2 Persistencia por alcance y con postprocesamiento

La persistencia proporcionada por este gestor es una persistencia *por alcance*, que necesita *roots* o puntos de entrada para acceder a los objetos almacenados, y que está implementada basándose en la serialización de Java, aunque mejorada, ya que permite la recuperación ante una caída del sistema.

Cara a conseguir la persistencia con PSE Pro hay que tener en cuenta las siguientes consideraciones:

- En primer lugar es necesario identificar, tanto las clases cuyas instancias pueden ser persistentes (*persistence capable classes*), como aquellas que aunque no lo pueden ser, sus métodos si pueden acceder y manipular instancias que si lo son (*persistence aware classes*). La capacidad de ser persistente no se hereda.
- En segundo lugar, las clases enclavadas en una de las categorías anteriores, serán compiladas normalmente, y a continuación serán postprocesadas todas juntas. El postprocesador hace una copia de los ficheros[9] de estas clases, y anota el código que necesitan para la persistencia (añade instrucciones al *bytecode*). Las anotaciones realizadas por el postprocesador permiten asegurar que los campos siempre son cargados antes de ser accedidos, y que las instancias modificadas son siempre actualizadas en la base de datos cuando se realiza el *commit* de la transacción en curso. Las anotaciones también pueden realizarse manualmente.

[7] Aquí veremos únicamente las características del gestor para Java

[8] De ahora en adelante haremos referencia siempre a PSE Pro por sus mejores prestaciones

[9] Los nuevos ficheros pueden sobreescribir a los originales o bien ser situados en otro directorio

- Las extensiones[10] de las clases en PSE Pro son gestionadas mediante colecciones, de hecho, cada colección Java tiene su equivalente en PSE Pro. Para recorrer las colecciones PSE Pro proporciona *iteradores* que permiten entre otras operaciones la inserción y el borrado.
- Con el fin de acceder a los objetos almacenados en la base de datos se necesitan establecer *roots* o puntos de entrada. Si el objeto que se especifica como *root* no es persistente PSE Pro lo hace persistente automáticamente.

5.8.3 Eliminación con recolección

PSE Pro dispone de un recolector de basura (GC, Garbage Collector) que libera el espacio asociado con los objetos Java que no son alcanzables. Trabaja en dos fases: *marcado*, que identifica los objetos que no son alcanzables, y *barrido*, que libera el espacio usado por los objetos no alcanzables.

El recolector evita tener que escribir código para borrar los objetos desde la base de datos, pero es responsabilidad del programador desconectar los objetos de sus relaciones y asociaciones.

5.8.4 Concurrencia asociada a sesiones

PSE Pro permite múltiples sesiones concurrentes para un mismo proceso de la JVM (*Java Virtual Machine),* entendiéndose por sesión un contexto en el que se puede crear una transacción, acceder a la base de datos y manipular objetos persistentes. Cada sesión puede ejecutar sólo una transacción a la vez y tener abierta una única base de datos (aunque la misma base de datos puede estar abierta simultáneamente por varias sesiones). En una misma sesión pueden participar cualquier número de hilos Java.

5.8.5 Evolución de esquemas basada en la serialización de Java

La evolución de esquemas contemplada por PSE Pro permite añadir o eliminar un campo de instancia, cambiar el tipo de un campo o cambiar el orden de los campos. Sin embargo, no se puede emplear evolución de esquemas para cambiar la jerarquía de herencia de una clase añadiendo, borrando o cambiando una superclase. La técnica que emplea se basa en la serialización para la carga y descarga.

5.8.6 Consultas basadas en colecciones

PSE Pro proporciona un paquete (*com.odi.util.query)* para consultar colecciones de objetos siempre que éstas implementen la interfaz *java.util.Collection*.

En las consultas en PSE Pro se pueden destacar las siguientes particularidades:

- La consulta puede contener métodos que contengan argumentos (literales o variables ligadas), siempre que devuelvan algún valor (no pueden devolver *void)*. Una consulta puede ser reutilizada contra la misma colección, quizás con diferentes *bindings* para las variables libres, o contra diferentes colecciones.
- Las consultas sobre una clase devuelven instancias específicas de esa clase, pero no de sus subclases. No permite, por tanto, especificar si las consultas son sobre la clase o sobre la jerarquía.

[10] La extensión de una clase la constituyen el conjunto de instancias de la misma

- La clase especificada en la consulta así como cualquier miembro (campo o método) especificado en la consulta deben ser públicamente accesibles.
- PSE Pro permite una optimización manual y automática de las consultas. Si se optimizan explícitamente las consultas (*Query.optimize()*), cualquier índice especificado en la optimización debe estar disponible sobre esta colección, y si no lo está se generará un error. Si no se optimiza explícitamente una consulta, PSE Pro lo realiza automáticamente cuando aplica la consulta a la colección y para todos los índices disponibles en la colección consultada. Si una consulta optimizada se aplica de nuevo a la misma colección o a otra con los mismos índices, sigue siendo válida esta optimización, pero si no estuvieran todos los índices que fueron presentados en la optimización se reoptimiza la consulta automáticamente (a diferencia de la optimización manual).

5.8.7 Indexación no extensible basada en B-Tree y tablas Hash

En PSE Pro se pueden añadir índices a cualquier colección que implemente la interfaz *com.odi.util.IndexedCollection.* Esta interfaz proporciona métodos para añadir y eliminar índices, y para actualizarlos cuando los datos indexados hayan cambiado. La colección más apropiada para almacenar grandes cantidades de objetos persistentemente es *OSTreeSet* (o una subclase de ella), ya que es la única que tiene ya implementada la interfaz que permite la indexación. Sin embargo, no permite crear instancias transitorias.

Sobre la gestión de índices en PSE Pro se pueden resaltar las siguientes características:

- Las técnicas empleadas para la indexación son las clásicas: árboles B (B-Tree) y tablas hash. No permite la incorporación de nuevas técnicas de indexación.
- En la creación de un índice se puede especificar si debe estar ordenado y si admite duplicados. Por defecto, es no ordenado y permite duplicados.
- PSE Pro mantiene automáticamente el índice cuando se añaden o se borran elementos a la colección. Sin embargo, es responsabilidad del programador actualizar el índice cuando los miembros indexados cambien en instancias que ya son elementos de una colección indexada.
- No permite la indexación sobre más de un campo o método.

5.8.8 Crítica

5.8.8.1 Carencia de flexibilidad/extensibilidad

Este gestor proporciona un conjunto de servicios rígidamente establecidos, no permitiendo la incorporación de nuevas técnicas para ninguno de esos servicios.

5.8.8.2 Indexación básica y sin extensibilidad

Permite la indexación de cualquier colección, pero esa indexación sólo puede realizarse sobre claves simples. Además, las únicas técnicas de indexación que soporta son las tradicionales hash y B-Tree, sin permitir la incorporación de otras nuevas.

5.8.8.3 No permite ningún tipo de distribución

PSE Pro por su arquitectura antes mencionada no proporciona ningún mecanismo que permita trabajar con objetos distribuidos, ni siquiera con una arquitectura cliente

servidor. Una posibilidad para trabajar con objetos distribuidos pasaría por emplear CORBA como soporte para objetos distribuidos, y emplear PSE Pro a modo de repositorio local en cada nodo.

5.8.8.4 Soporte básico de consultas

PSE Pro proporciona un soporte básico para las consultas, sin embargo, no proporciona ningún entorno para la interrogación de la base de datos con un lenguaje de consulta estilo OQL- SQL.

5.8.9 Características interesantes

5.8.9.1 Basado en el modelo de objetos

Es un gestor basado en el modelo de objetos, y por tanto, soporta directamente las características de un modelo de objetos (en este caso de Java) sin necesidad de acudir a descomposiciones artificiales.

5.8.9.2 Portabilidad

Es un gestor implementado completamente en Java lo que garantiza que en todas las plataformas para las que se disponga de un emulador de Java este gestor será válido, lo que facilita su portabilidad.

5.8.9.3 Soporta el *binding* Java de ODMG

Esto es realmente interesante, ya que si el *binding* para Java de ODMG pretende ser el API estándar, PSE Pro da ya el primer paso para la portabilidad de las aplicaciones entre bases de datos orientadas a objetos.

5.9 Jasmine

Jasmine es una base de datos que soporta estructuras de datos complejas, herencia múltiple, propiedades y métodos a nivel de instancia y a nivel de clase, colecciones y métodos a nivel de colección, especificación de integridad referencial, y a diferencia de otros gestores la clase contiene su propia extensión. Proporciona dos posibilidades para el desarrollo de aplicaciones:

- **API Jasmine C** que es un conjunto de llamadas a funciones que manipulan la base de datos Jasmine, a través de C, C++ y otros lenguajes de programación. Cuando se programa con este API, ODQL proporciona acceso a la base de datos, e incluye funciones para definiciones, consultas y manipulación de objetos. El lenguaje anfitrión permite realizar las operaciones no relacionadas con la base de datos, y operaciones con datos externos.
- **JADE (Jasmine Application Development Environment)** es un entorno de desarrollo gráfico que permite trabajar directamente con las clases e incluso con las instancias definidas en una base de datos. Permite diseñar, hacer prototipos y desarrollar aplicaciones con Jasmine con una interfaz de arrastrar y soltar.

Además de con C y C++, Jasmine proporciona un *binding* para Java creando una clase Java para cada clase Jasmine. Proporciona también una interfaz para cualquier entorno que soporte ActiveX, y mediante un conjunto de herramientas (*WebLink*), proporciona una forma de acceso a la base de datos mediante Internet empleando páginas HTML.

5.9.1 Arquitectura

La arquitectura de Jasmine [CAI98] está compuesta por cuatro módulos principales: el servidor de la base de datos, el cliente, los almacenes, y las librerías de clases.

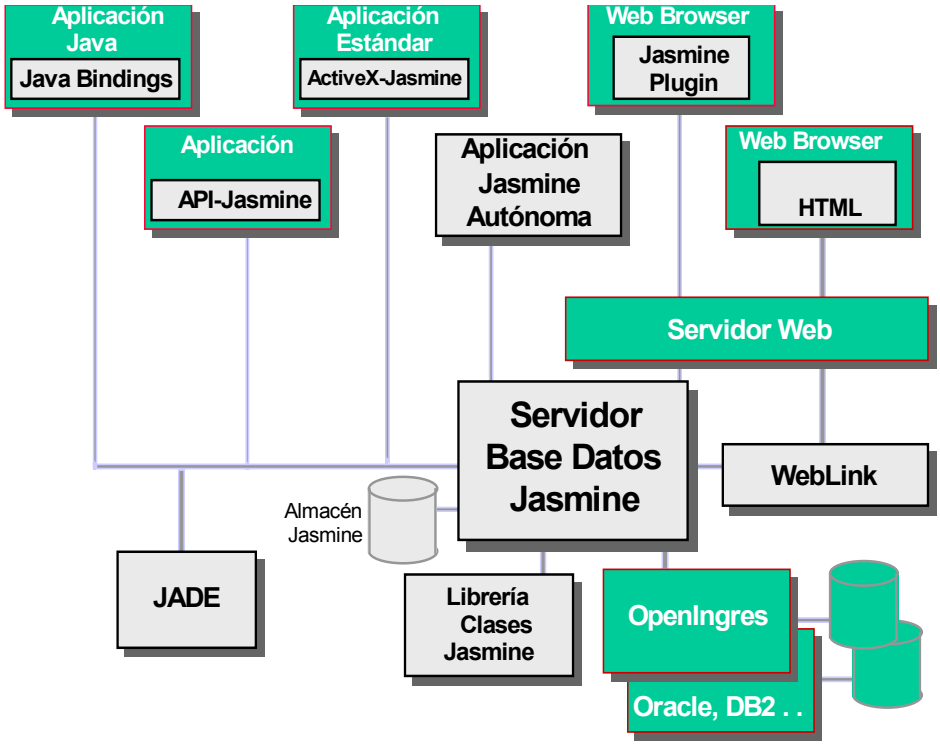


Figura 5.8. Arquitectura de Jasmine

- **Almacenes**, uno o más (unos definidos por el sistema y otros por el usuario). Los almacenes son los contenedores físicos de datos dentro de la base y contienen tanto metadatos como datos de usuario. Cada almacén contiene una o más familias de clases, dónde una familia de clases es una colección de clases relacionadas entre sí a las que se les da un nombre. Físicamente un almacén es un conjunto de ficheros, cuyo tamaño es definido por el usuario cuando lo declara.
- **Servidor de Base de Datos,** es el servidor que controla todo tipo de operaciones sobre los datos almacenados en los almacenes. Jasmine emplea la tecnología *thin client,* de forma que los métodos complejos son ejecutados en el servidor, encontrándose así bajo su sistema de transacciones y seguridad.
- **Clientes***,* representan el entorno donde se ejecutan las aplicaciones que acceden a la base de datos.
- **Librería de clases***,* son grupos de familias de clases que están predefinidas y distribuidas con Jasmine. Entre ellas destacan la librería de clases multimedia (para el almacenamiento de diferentes datos multimedia), y la familia de clases SQL que permiten el acceso y actualización de datos de otras bases de datos, incluso relacionales (Oracle, Sybase, Informix, etc.).

5.9.2 Gestor en capas basado en la tecnología relacional

La implementación del gestor de base de datos está organizada en capas [II96] con dos partes principales (Figura 5.9), la capa del tratamiento de datos, y la capa de tratamiento de objetos. La capa de tratamiento de datos está basada en una ampliación de la tecnología relacional.

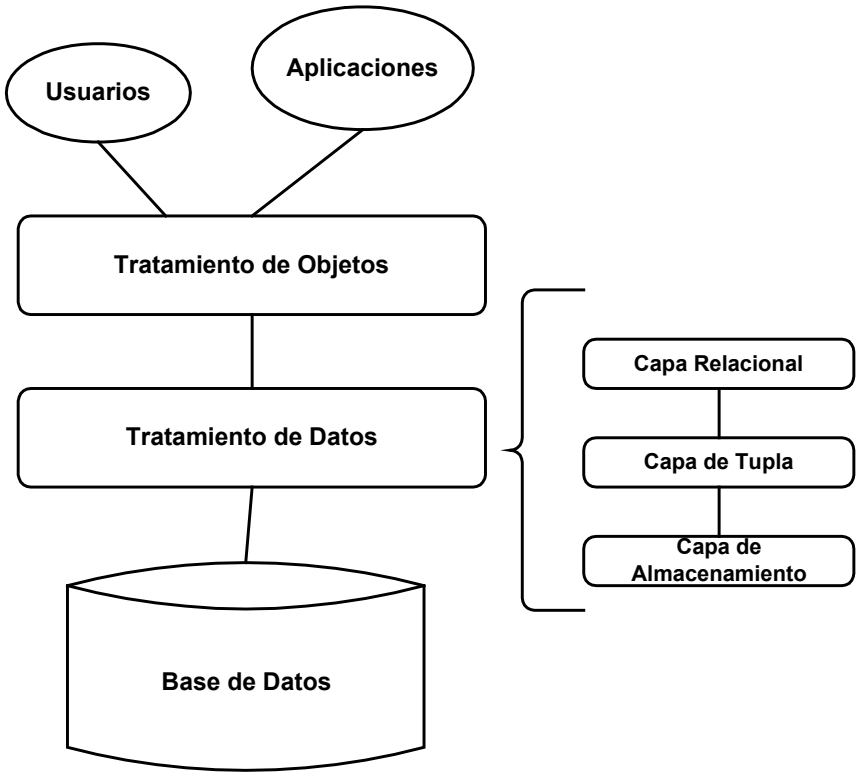


Figura 5.9. Modelo en capas del gestor de base de datos de Jasmine

El **subsistema de manejo de datos** implementa los índices y las relaciones entre objetos soportando únicamente cuatro tipos de relaciones: secuenciales, B-Tree, hash e

internas, siendo la estructura de la tupla independiente del tipo de relación. La *capa relacional* proporciona funciones para ejecutar operaciones sobre conjuntos, como un álgebra relacional extendida. La *capa de tupla* proporciona funciones a nivel de tuplas (*scan, raster, delete, insert*, etc.), y la *capa de almacenamiento* proporciona funciones que incluyen E/S de disco, *buffering* de páginas, transacciones y control de concurrencia y recuperación.

En el **subsistema de manejo de objetos** el sistema realiza automáticamente la traducción de objetos en relaciones; la información sobre esta traducción, se almacena en las clases. Todas las instancias intrínsecas a una clase son almacenadas en una única relación, haciendo corresponder una instancia a una tupla, y un atributo a un campo. Las instancias intrínsecas a una superclase, y las intrínsecas a subclases, son almacenadas en relaciones separadas, de tal forma que al crear o destruir instancias intrínsecas a una clase dada, no se tiene que propagar ninguna modificación a las clases o subclases asociadas. El código fuente y compilado de los métodos y los *demons*[11] es almacenado también en relaciones de las que se recuperan y utilizan durante la optimización de las consultas. Los métodos polimórficos son traducidos en sus correspondientes funciones de implementación.

5.9.3 Buffering de objetos y de páginas

El subsistema de manejo de objetos requiere *buffering de objetos* además del tradicional *buffering de páginas*. El buffer de páginas tiene estructuras adecuadas para acceder a diferentes instancias de la misma clase, mientras que el buffer de objetos tiene estructuras adecuadas para acceder a instancias correlativas de diferentes clases. A pesar de los dos buffers, la evaluación de las consultas Jasmine la realiza sobre un único buffer (el de páginas), previo traspaso de las instancias del buffer de objetos al de páginas.

5.9.4 Indexación para soportar jerarquías de herencia y agregación

Jasmine para la optimización de consultas emplea índices de clase simple (SC), apropiados para soportar consultas basadas en las jerarquías de herencia. Sin embargo, también proporciona índices sobre caminos para consultas sobre la jerarquía de agregación. Sin embargo, los índices se aplican solamente para la clase dónde son creados y no a las clases subordinadas.

5.9.5 Lenguaje de programación Jasmine/C

Jasmine incorpora como lenguaje de programación para la construcción de aplicaciones avanzadas un lenguaje (*Jasmine/C*) que integra un lenguaje de propósito general C, y un lenguaje de base de datos en un contexto orientado a objetos. El compilador de Jasmine, está implementado para que utilice un compilador de C. Los programas de aplicación, escritos en *Jasmine/C,* son precompilados a programas de C, los cuales son compilados y enlazados con la *runtime support library.* El preprocesamiento, se utiliza para obtener el máximo rendimiento de la portabilidad y la optimización de código del compilador de C.

[11] Funciones especializadas que pueden ser asociadas a los atributos y sirven para definir condiciones o acciones a llevar a cabo antes de que ciertos valores sean asociados a los mismos

5.9.6 Lenguaje de consulta ODQL

El lenguaje de consulta de Jasmine es ODQL (*Object Database Query Language*). Es un lenguaje utilizado para hacer consultas y manipular los objetos de la base de datos. Expresa la lógica de las operaciones a través de métodos definidos en la propia base de datos, que son ejecutados en el servidor. ODQL puede emplearse además de con API C, interactivamente en un terminal, utilizando el intérprete de ODQL.

Con relación a las consultas hay que tener en cuenta las siguientes consideraciones:

- Este lenguaje permite consultas dinámicas.
- Una consulta en Jasmine devuelve todas las *instancias intrínsecas* de la clase, y las intrínsecas a sus subclases que cumplan las condiciones necesarias.
- Cuando en una consulta se hace una llamada a un método definido en una superclase, va a ser invocado automáticamente ese método sobre instancias de la propia clase, y sobre las de cada subclase definida, siendo posible en algunos casos, que el tratamiento de las instancias sea distinto en unas clases o en otras, debido a la posibilidad de redefinición de métodos heredados.
- Es posible introducir métodos en el cuerpo de las consultas, e inversamente también es posible que los métodos puedan llevar especificaciones de consultas (*atributos virtuales).* Además, el usuario puede invocar consultas directamente desde los programas de aplicación. Para ello, se introducen *variables conjunto*, que pueden contener los conjuntos de objetos devueltos por la consulta.

5.9.7 Crítica

5.9.7.1 Desadaptación de impedancias con el lenguaje de consulta

La inserción de ODQL en un programa C supone una desadaptación impedancias al estilo de las mencionadas en el capítulo 2, y para adaptar ambos lenguajes Jasmine acude al concepto de variables de instancia y variables de conjunto.

5.9.7.2 Gestor relacional

Aunque Jasmine sea tildada de base de datos pura, se ha observado por el modelo en capas del gestor que lo que realmente hace es una traducción de los objetos a relaciones, eso si, almacenando también los métodos dentro de las mismas. Sin embargo, no deja de ser una traducción adicional e innecesaria ya que realmente y a diferencia de en un SGBDR en el que las relaciones existen para el usuario, aquí únicamente se utilizan como intermediarias.

5.9.7.3 Carencia de flexibilidad/extensibilidad

El mecanismo de indexación en Jasmine es compacto y no permite su ampliación con nuevas técnicas. De hecho, este gestor no considera la posibilidad de extender el sistema en ningún sentido, excepto, la librería multimedia que ha sido diseñada para que pueda soportar fácilmente nuevos tipos de datos.

5.9.8 Características interesantes

5.9.8.1 Soporte para múltiples lenguajes

Jasmine es una base de datos muy potente, pero la forma que tiene de proporcionar enlaces para diferentes lenguajes varía. En unos casos es mediante ActiveX, mientras

que en otros es definiendo nuevas clases (Java). Para el lenguaje Java, soporta el Java *binding* de ODMG.

5.9.8.2 Almacenamiento de métodos en la base de datos

El hecho de almacenar los métodos en la base de datos, hace la integración de consultas y facilidades de programación más elegante que el almacenamiento en ficheros ordinarios, haciendo además que estén sujetos a los mismos criterios, por ejemplo de seguridad, que el resto de datos almacenados en la base de datos.

5.9.8.3 Técnicas de indexación apropiadas para el modelo de objetos

Jasmine incluye un mecanismo de indexación, que a pesar de no ser extensible, si está adaptado al modelo de objetos al incluir técnicas apropiadas para consultas basadas tanto en la jerarquía de herencia como de agregación.

5.9.8.4 Potente interfaz de usuario

Este gestor proporciona una interfaz de usuario (*JADE*) basada en 'arrastrar y soltar' y con diferentes opciones para manejar y visualizar la información: *Class Browser* (permite trabajar con una base de datos), *Class Property Inspector* (para acceder a las propiedades de una clase), *Object Property Inspector* (para acceder a las propiedades de los objetos), y *Method Editor* (permite la creación de métodos asociados a las clases). Incorpora también un *Query Editor* para construir o recuperar consultas.

5.10 POET[12]

Poet Java [Poe00] es un SGBDOO que permite a un programa Java conectarse a cualquier Servidor de Objetos de Poet, posibilitando de esta forma acceder a los mismos objetos (base de datos) desde cualquier aplicación Poet (independiente del *binding* con el que haya sido desarrollada), pudiéndose incluso copiar la base de datos a máquinas que se ejecutan sobre sistemas operativos diferentes. El formato físico es binario compatible entre plataformas y lenguajes de programación.

5.10.1 Arquitectura

Poet en su versión para Java trabaja con cuatro elementos imprescindibles: la base de datos propiamente dicha, el diccionario, un preprocesador y un fichero de configuración.

La **base de datos** almacena todos los objetos que crea y usa la aplicación. El **diccionario**, llamado también *esquema de clases*, almacena para cada clase que puede ser persistente, el nombre y el identificador de la clase, descripciones de los miembros de cada clase y descripciones de los índices definidos por la clase (si los hay). Además, conoce también las superclases, subclases y las interfaces implementadas. Puede ser compartido por cualquier número de bases de datos, de forma que si la aplicación abre varias bases de datos que comparten un diccionario, éste es cargado una única vez. El **fichero de configuración**, en el que se especifican entre otras cosas las clases susceptibles de ser persistentes, es la entrada para el **preprocesador**. Tanto el diccionario como la base de datos son generados automáticamente por el preprocesador *ptj*.

La base de datos puede estar formada por un único fichero (por defecto), o por varios: *objects.dat*, que es el fichero de datos que contiene los objetos almacenados y, *objects.idx*, que contiene la información de los índices para los objetos en la base de datos.

5.10.2 Persistencia por alcance y con preprocesamiento

En relación con la persistencia en POET hay que tener en cuenta las siguientes consideraciones:

- Es necesario especificar todas aquellas clases que son susceptibles de ser persistentes en el fichero de configuración (*ptj.opt*). Este fichero de configuración es leído por el procesador, y éste registra la información en el diccionario y añade el código, métodos y clases auxiliares a los ficheros de clases Java, para convertirlas en clases capaces de ser persistentes.
- Una vez que una clase ya es susceptible de contener instancias persistentes, para convertirse realmente en persistentes éstas deberán ser referenciadas por otros objetos persistentes.
- Las extensiones de las clases en POET son creadas por el programador a conveniencia pero son mantenidas por el propio gestor, por lo que no disponen de métodos para añadir o eliminar elementos, pero si para navegar en la extensión, recuperar un elemento, avanzar al siguiente, etc. Para poder crear un

[12] Versión 6.1 de Poet para Java

objeto de tipo *Extent* asociado a una clase es necesario haber especificado en el fichero de configuración que dicha clase va a disponer de extensión.

- Es posible asociar un nombre a un objeto para su posterior recuperación, pero también pueden existir objetos sin nombre que pueden ser recuperados empleando la extensión de la clase.

5.10.3 Eliminación con recolección

POET dispone de un recolector de basura (*ptgc*) que se encarga de borrar todo objeto que no es alcanzable desde cualquier objeto ligado a la base de datos. El principal problema con el recolector es que elimina cualquier objeto que no tenga nombre asociado. Es por eso que para evitar este problema el recolector acude a un esquema simple que se basa en no borrar el objeto si está bajo una de estas cuatro condiciones:

- El objeto tiene asociado un nombre.
- La clase del objeto es marcada como no débil en la declaración de la clase en el fichero de configuración de POET. Por defecto todas las clases se consideran débiles, lo que quiere decir que el recolector puede borrar sus instancias. Cuando se marca una clase como no débil todas las extensiones de sus subclases son consideradas también como no débiles.
- La clase de los objetos hereda 'no debilidad' recursivamente desde una de sus clases base.
- El objeto es alcanzable desde otro objeto que cumple una o más de las tres condiciones precedentes.

5.10.4 Concurrencia y *checkpoints*

En POET todas las operaciones de base de datos deben realizarse dentro de una transacción. Las transacciones en POET soportan *checkpoints* que son puntos de ejecución internos que no finalizan la transacción pero que escriben en la base de datos las modificaciones realizadas en los objetos hasta ese momento, aunque manteniendo sus bloqueos. Estos *checkpoints* son útiles sobre todo cuando hay un gran número de objetos modificados en una transacción y se quiere evitar una gran operación *commit* al finalizar la transacción, o bien cuando se quiere hacer una consulta basada en los datos de los objetos modificados sin necesidad de salir de la transacción. POET también soporta transacciones anidadas (pero no *checkpoints* dentro de éstas).

En cuanto al bloqueo, POET permite un bloqueo implícito por defecto (de lectura cuando se recupera un objeto, o de escritura cuando se intenta modificar) o un bloqueo explícito con cuatro niveles (*read, upgrade, write y delete*).

POET Java soporta aplicaciones, con un único hilo que emplea una transacción en cada momento, con múltiples hilos y cada hilo asociado con una transacción, y con múltiples hilos que comparten una transacción (en este caso el programador debe añadir sus propios controles de concurrencia).

5.10.5 Seguridad y Recuperación

POET proporciona una opción de configuración para usar ficheros de recuperación de forma que si se produce un fallo mientras se actualiza la base de datos se puede usar este fichero para reescribir la actualización. Además, proporciona una serie de funciones

de administración que incluyen entre otras lo que POET llama *'External Backups'*. Esta utilidad permite a un cliente (en un entorno cliente-servidor) suspender la base de datos con el fin de hacer una copia sin afectar a otros clientes que estén usándola. Los cambios hechos por los otros clientes son almacenados hasta que la base de datos esté de nuevo en línea.

5.10.6 Evolución de esquemas con versiones

Al cambiar las definiciones de algunas de las clases (añadir atributos, cambiar el tipo de un atributo, etc.) es necesario ejecutar de nuevo el preprocesador para reflejar los cambios en el diccionario, de forma que en el diccionario se mantienen las dos versiones. Así, cuando se lee un objeto de la versión antigua automáticamente es convertida (se le aplican los cambios) para reflejar las características de la nueva versión. Si el objeto es entonces almacenado de nuevo en la base de datos, lo hará con el formato actualizado. Sin embargo, POET Java, a diferencia de POET C++, no soporta el acceso explícito a diferentes versiones de una clase. Siempre emplea la más reciente.

Incorpora también una utilidad que cambia la versión a todos los objetos siempre que la base de datos esté cerrada.

5.10.7 Indexación básica sobre los identificadores y definición de índices adicionales

Una clase siempre tiene al menos un índice basado en los identificadores asociados internamente a los objetos, y que determina el orden de los objetos cuando se recuperan desde la extensión. Pero POET permite definir índices adicionales para la extensión de la clase y proporciona métodos para determinar y asignar que índice es usado. Estos índices deben ser especificados en el fichero de configuración, asociados con la extensión de la clase correspondiente, y pueden contener más de un campo como clave. El mantenimiento de los índices es realizado automáticamente por el sistema.

5.10.8 Crítica

5.10.8.1 Limitaciones en la interoperabilidad

Si bien es cierto que es posible conseguir cierta interoperabilidad entre los diferentes *bindings* que soporta actualmente POET, también es cierto que está limitada únicamente a gestores POET, no pudiendo por ejemplo acceder desde otros gestores a los datos almacenados desde POET.

También hay que considerar que la interoperabilidad entre diferentes *bindings* en POET impone ciertas limitaciones derivadas en gran parte de la no adopción de un modelo de objetos común. Así por ejemplo, C++ soporta herencia múltiple y Java no, con lo cual cara a conseguir plena interoperabilidad no debería emplearse. Lo mismo ocurre con los *templates* de C++, y otra serie de características.

5.10.8.2 Carencia de flexibilidad/extensibilidad

A diferencia de otros sistemas comerciales existentes en el mercado, POET no contempla la extensibilidad del sistema no permitiendo ni siquiera la incorporación de nuevos métodos de acceso.

5.10.9 Características interesantes

5.10.9.1 Compatible con el estándar ODMG

Este SGBDOO soporta el Java *binding* de ODMG y emplea como lenguaje de consulta OQL lo que supone un paso hacia la portabilidad de las aplicaciones entre bases de datos.

5.10.9.2 Portabilidad de la base de datos

Este gestor emplea un formato canónico para el almacenamiento de la base de datos en disco. Esto permite que la base de datos sea accedida por programas clientes POET que se ejecutan sobre diferentes plataformas físicas. Las descripciones de las clases son también almacenadas en el diccionario o esquema cuyo formato canónico permite que los datos sean accedidos por aplicaciones escritas empleando diferentes *bindings* Poet (actualmente Java y C++).

5.10.9.3 Permite cambiar el comportamiento del bloqueo

POET sigue la recomendación del Java *binding* de ODMG en el que se proponen cuatro modos de bloqueo (*read, write, upgrade y delete*). No obstante, estos cuatro modos de bloqueo POET Java los traduce a un modo interno más apropiado que proporciona máxima concurrencia mientras proporciona aislamiento completo, por lo que sí el programador lo desea puede emplear directamente estos modos (mediante las propiedades de las transacciones) cambiando así el comportamiento del bloqueo, para conseguir por ejemplo un bloqueo verdaderamente optimista.

5.10.9.4 Patrones de acceso

Para minimizar el tráfico en la red y el consumo de memoria el *Servidor de Objetos* de POET difiere la transmisión de sub-objetos hasta que éstos son accedidos por la aplicación. Esta transmisión aplazada proporciona muchas ventajas, pero puede incurrir en una llamada adicional para cada uno de los sub-objetos necesitados. Entonces, como unas veces es conveniente la recuperación de todos los objetos y otras veces no, POET permite un acceso basado en patrones, de forma que antes de solicitar un objeto raíz se le puede decir que objetos asociados son necesarios, permitiendo así al servidor transferir los objetos especificados con el raíz.

Los patrones de acceso son definidos en el fichero de configuración del servidor y se aplican sólo a arquitecturas cliente-servidor; usarlos en modo local no tiene efecto.

5.11 EXODUS

EXODUS (Extensible Object-Oriented Database System) es un sistema de base de datos extensible que a diferencia de otros sistemas no pretende ser un SGBD completo. Su objetivo es proporcionar un núcleo elemental de utilidades básicas de gestión de bases de datos, más una serie de herramientas que faciliten la generación semi-automática de SGBD específicos para nuevas aplicaciones [HHR91]. Con EXODUS aparece un nuevo usuario de base de datos, además de los usuarios en el sentido tradicional de la palabra y los administradores de bases de datos, que es el *implementador de la base de datos* (IBD). El objetivo de EXODUS es que no sea necesario un gran esfuerzo por parte del IBD para generar nuevos SGBD.

5.11.1 Arquitectura

En la Figura 5.10 se presenta la estructura de un SGBD específico sobre EXODUS para una aplicación. Las herramientas [CDF+86] con las que cuenta un IBD a la hora de desarrollar un sistema se describen brevemente a continuación.

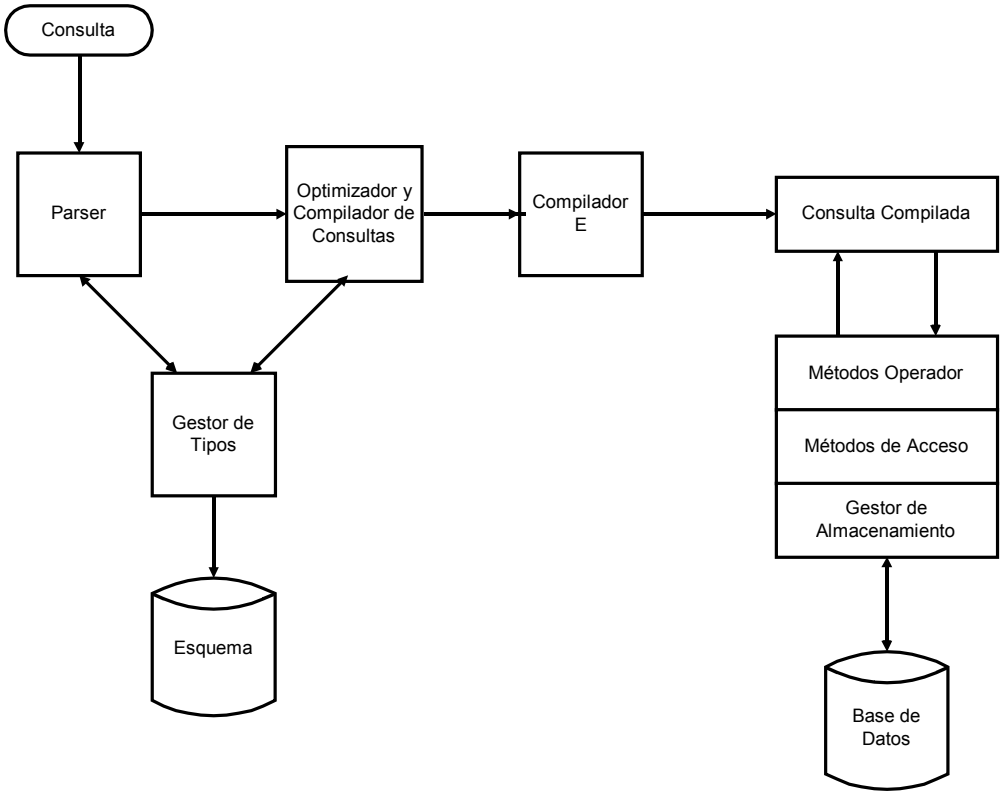


Figura 5.10. Arquitectura de un SGBD basado en EXODUS

Propósito general

En el nivel inferior del sistema se encuentran elementos que proporcionan funcionalidades de propósito general y que serán empleados por cada SGBD construido sobre EXODUS.

- **Gestor de objetos de almacenamiento.** Este gestor tiene como unidad básica el objeto de almacenamiento, una secuencia de bytes de tamaño arbitrario de longitud variable no interpretados y sin tipo.

- **Lenguaje de programación E y su compilador para escribir software de bases de datos.** E es un lenguaje de implementación para todos los componentes del sistema. Su objetivo es simplificar el desarrollo de sistemas de software para un SGBD.
- **Una librería de métodos de acceso independientes del tipo, que pueden ser usadas para el acceso a objetos de almacenamiento.** Para los métodos de acceso EXODUS proporciona una librería de estructuras de índice independientes del tipo (soportan el parámetro tipo) que incluyen B-Tree, ficheros Grid y hash lineal.

Métodos operador

La siguiente capa en la arquitectura la constituyen los **métodos operador**, que son una colección de métodos que pueden ser combinados con otros con el fin de operar sobre objetos de almacenamiento. El IBD necesita implementar uno o más métodos para cada operador en el lenguaje de consulta asociado con la aplicación. Esto lo hará con E.

Gestor de tipos

En el centro de la arquitectura de EXODUS está el **gestor de tipos** que pretende proporcionar soporte para las necesidades de un amplio rango de SGBD específicos. Este gestor mantiene información sobre dos categorías relacionadas del sistema:

- **Jerarquía de clases.** La información acerca de la jerarquía de clases es almacenada en un conjunto de objetos de una meta-clase (Class).
- **Un catálogo.** Mantiene un catálogo que contiene información acerca de la clase de cada fichero, e información estadística sobre el fichero (tal como las cardinalidades de cada subclase dentro del fichero) para usar durante la optimización de consultas.

5.11.2 Soporta tipos de datos y operadores definidos por el usuario

EXODUS soporta tipos base, tipos construidos y nuevos tipos base añadidos por el usuario. EL IBD puede definir nuevos tipos y operadores sobre ellos empleando el lenguaje E. Estos tipos son compilados con el gestor de tipos cuando se generan los nuevos SGBD específicos.

5.11.3 Núcleo de EXODUS: el gestor de objetos de almacenamiento

El gestor de objetos es el núcleo de EXODUS. Se accede a él mediante una interfaz procedural que permite la creación y destrucción de ficheros que contienen conjuntos de objetos, así como la iteración en los contenidos del fichero. Los objetos pueden ser insertados y eliminados en el fichero, y también permite la especificación explícita del *clustering* de objetos sobre disco. Además de la función básica de almacenamiento, proporciona gestión de buffer, control de concurrencia, mecanismos de recuperación, y un mecanismo de versionado que puede ser empleado para implementar versiones en aplicaciones específicas.

5.11.4 Lenguaje de programación E

El lenguaje de programación E [RCS93] es un lenguaje para el IBD no un lenguaje de bases de datos. E extiende C con variables objeto persistentes, tipos de datos IDO, y tipos parametrizados entre otras características. El traductor de E genera código fuente C más las llamadas apropiadas al gestor de almacenamiento. El traductor es también el

encargado de añadir las llamadas apropiadas para *buffering*, bloqueo de objetos, recuperación, etc. De esta forma el IBD no se tiene que preocupar de la estructura interna de los objetos persistentes.

5.11.5 Permite la construcción de métodos de acceso genéricos

EXODUS permite la construcción de métodos de acceso genéricos acudiendo al empleo de clases generadoras soportadas por el lenguaje de programación E [RC87]. Una clase puede ser instanciada con funciones como parámetros además de tipos. E además, libera al IBD de traducir las estructuras de datos entre memoria y disco, y permite una gestión de transacciones transparente.

EXODUS para facilitar la coordinación entre los métodos de acceso y la gestión de transacciones proporciona una serie de *protocol stub*[13], que son abstracciones de un protocolo de bloqueo particular implementadas como una colección de fragmentos de código, que el traductor de E inserta en puntos apropiados durante la compilación de un programa E, más estructuras de datos relacionadas. De esta forma un IBD para escribir un nuevo método de acceso únicamente necesitará: a) seleccionar el protocolo deseado[14] en tiempo de compilación por medio de indicaciones declarativas en E, b) encerrar cada operación del método de acceso entre las palabras reservadas *begin* y *end* de E; y entonces puede c) incluir una o más llamadas a rutinas específicas del *stub* en el código de la operación.

5.11.6 Versiones en los objetos de almacenamiento

Los objetos de almacenamiento pueden ser pequeños (residen en una página de disco) o grandes (pueden ocupar varias páginas de disco), pero en cualquier caso su IDO[15] es una dirección de la forma (página, slot).

EXODUS para cada objeto de almacenamiento sólo retiene una versión como la actual, siendo todas las versiones precedentes marcadas como viejas. Las versiones de los grandes objetos se mantienen copiando y actualizando las páginas que difieren de una versión a otra, con lo que cuando se elimina una versión hay que descartar cualquier página que sea compartida por otras versiones del mismo objeto.

5.11.7 Generador de optimizadores de consultas basado en reglas

EXODUS incluye un generador de optimizadores que produce un optimizador de consultas para una aplicación específica desde una especificación de entrada. Este generador toma como entradas un conjunto de operadores, un conjunto de métodos que implementan los operadores, reglas de transformación que describen las transformaciones que conservan la equivalencia de árboles de consultas, y la implementación de las reglas que describen como reemplazar el operador con un método específico. Además, el optimizador requiere que para cada método se proporcione una función que calcule el costo del método en función de las características de las entradas del mismo. El costo de un plan de acceso es definido como la suma de los costos de los métodos implicados.

---

[13] No se consideran parte extensible del sistema

[14] EXODUS proporciona *protocol stub* para el bloqueo en dos fases y para el bloqueo jerárquico

[15] El IDO de un objeto pequeño apunta al objeto sobre el disco, mientras que el de un objeto grande apunta a una cabecera que contiene punteros a otras páginas implicadas en la representación del objeto

5.11.8 Crítica

5.11.8.1 Definición restringida de nuevos tipos

Los nuevos tipos base únicamente pueden ser definidos por el IBD en el momento en que se genera el SGBD específico, lo que restringe quien y cuando puede definir nuevos tipos.

5.11.8.2 Generador de SGBD no SGBD

EXODUS es un generador de base de datos, y como tal proporciona herramientas para generar SGBD específicos pero realmente no es un SGBD, ya que carece, por ejemplo, de un lenguaje de programación o consulta propio.

5.11.9 Características interesantes

5.11.9.1 Extensibilidad

EXODUS es extensible. Es decir, EXODUS permite añadir nuevos métodos de acceso, nuevos tipos base e incluso nuevos optimizadores de consulta, pero todo en tiempo de diseño, es decir, mientras se construye el SGBD específico, lo que supone una extensibilidad limitada al momento de la construcción del SGBD.

5.11.9.2 Incorporación de nuevos métodos de acceso

La librería de métodos de acceso antes mencionada se puede ampliar por supuesto para proporcionar un mejor rendimiento para nuevos tipos de aplicaciones. Para realizar dichas ampliaciones debe emplearse el lenguaje E, que facilita la implementación al no tener que preocuparse el IDB de cuestiones como la transferencia de datos entre disco y memoria, o el control de la concurrencia.

5.11.9.3 Configurabilidad

EXODUS proporciona cierto grado de configurabilidad, permitiendo especificar algunas características que pueden influir en el rendimiento del sistema. Así, a la hora de crear objetos admite pistas sobre donde colocar un objeto, su tamaño aproximado, si el objeto debería estar sólo sobre una página de disco, etc. El gestor de *buffer* también admite indicaciones como el tamaño y número de *buffers* a usar, así como la política de reemplazamiento a emplear.

5.12 Resumen de características de los sistemas revisados

	Modelo	Indexación	Porta-bilidad	Extensi-bilidad	Compble ODMG	Característica Destacable
Starburst	SGBDOR	B-Tree Hash Extensible	No	Si	No	SGBD concebido para la extensibilidad
POSTGRES	SGBDOR	Extensible	No	Si	No	Extensibilidad dirigida por catálogos
ObjectDriver	*Wrapper*	No extensible	Si	No	Si	Trabaja con diferentes SGBDR comerciales
ObjectFile	Gestor Objetos	Únicamente por IDOs	No	No	No	Fomenta extensibilidad en la indexación
StreamStore	Gestor Objetos	B- Tree Extensible	Si	No	No	Módulos indexación y almacenamiento independientes
DarmStadt	Gestor Objetos	No en el núcleo	No	Si	No	Núcleo orientado a conjuntos
Jeevan	Gestor Objetos	No extensible	Si	No	No	Gestión automática de las extensiones
ObjectStore PSE Pro	Gestor Objetos	Hash B-Tree No extensible	Si	No	Si	Consultas basadas en colecciones
Jasmine	SGBDOO	SC J. agregación No extensible	No	No[16]	Si	Interfaz de usuario potente
POET	SGBDOO	No extensible	Si	No	Si	Interoperabilidad entre los *bindings* de ODMG
EXODUS	Generador BD	Hash B-tree Grid Extensible	No	Si	No	Generador de SGBD no un SGBD

Tabla 5.1. Resumen de algunas de las características de los sistemas revisados

[16] Únicamente es extensible la librería multimedia que incorpora

Capítulo 6 Necesidad de un sistema integral orientado a objetos para la construcción del SGBDOO

En los capítulos anteriores se ha planteado la problemática existente con los SGBDOO (carencia de flexibilidad/extensibilidad, dificultad para la interoperabilidad entre SGBD, difícil portabilidad, y reimplementación de las abstracciones ofrecidas por los sistemas operativos tradicionales), para la que algunos de los sistemas revisados en el capítulo 5 proponen soluciones parciales, pero ninguno de ellos propone una solución integral.

Los problemas anteriormente mencionados pueden beneficiarse de la adopción integral por todo el sistema del paradigma de orientación a objetos. En este capítulo se justifica la existencia de un sistema integral orientado a objetos, y se comentan las características que éste aportará al resto de elementos del sistema (y por tanto al SGBDOO).

6.1 Necesidad de un sistema integral orientado a objetos

Como ya se comentó en el capítulo 2 el problema que se plantea con el paradigma de orientación a objetos es que no es adoptado de una forma integral por todos los componentes del sistema, lo que provoca:

- **Desadaptación de impedancias o salto semántico** ocasionado cada vez que un elemento del sistema (ej. sistema operativo) debe interactuar con otro (ej. aplicación orientada a objetos), y que es debida básicamente a una desadaptación de interfaces y a unas abstracciones inadecuadas del sistema operativo.
- **Falta de interoperabilidad entre modelos de objetos** de forma que una aplicación desarrollada con el modelo de objetos de C++ no tiene ningún problema de comunicación con sus propios objetos, pero si con los de otros lenguajes de programación o bases de datos. Existe una falta de compatibilidad entre los modelos de objetos.

Una solución a este problema es la construcción de un sistema homogéneo en el que se utilice en todos sus elementos el mismo paradigma de orientación a objetos y se dé soporte nativo a los mismos: **un sistema integral orientado a objetos** [CIA+96]**.**

6.2 Características del sistema integral orientado a objetos

La idea de crear un sistema que dé soporte directo y utilice exclusivamente el paradigma de la orientación a objetos nos conduce a la siguiente definición de sistema integral orientado a objetos:

Un sistema integral orientado a objetos ofrece al usuario un entorno de computación que crea un mundo de objetos: un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando

mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados.

En un sistema integral como éste se crea un entorno de computación en el que todos los elementos: lenguajes, aplicaciones, compiladores, bases de datos, comparten el mismo paradigma de orientación a objetos. El sistema comprende el concepto de objeto y es capaz de gestionar directamente los objetos.

Al usar el mismo paradigma no existe desadaptación. Desde una aplicación OO tanto el acceso a los servicios del sistema como la interacción con otros objetos se realizan utilizando los mismos términos de orientación a objetos.

No son necesarias capas de adaptación con lo que se reduce la complejidad conceptual y técnica del sistema. Los usuarios/programadores se ven liberados de preocuparse por detalles técnicos y contraproductivos cambios de paradigma, y pueden concentrarse en el aspecto más importante que es la resolución de los problemas usando la orientación a objetos.

6.3 Aportaciones del sistema integral

El sistema integral permite aprovechar al máximo las ventajas de la orientación a objetos [Boo94] como son la reutilización de código, la mejora de la portabilidad, un mantenimiento más sencillo, una extensión incremental, etc. en todos los elementos del sistema y no sólo en cada aplicación orientada a objetos como en los sistemas convencionales.

Este sistema integral aporta al resto de elementos del sistema y por tanto, al SGBDOO, las características que se comentan a continuación.

6.3.1 Uniformidad conceptual en torno a la orientación a objetos

El único elemento conceptual que utiliza el sistema es un mismo paradigma de orientación a objetos. El sistema proporciona una única perspectiva a los usuarios/programadores: la de objetos, que permite comprender más fácilmente sistemas cada vez más complicados y con más funcionalidad.

Modo de trabajo exclusivamente orientado a objetos

La única abstracción es, por tanto, el objeto (de cualquier granularidad), que encapsula toda su semántica. Lo único que puede hacer un objeto es crear nuevas clases que hereden de otras, crear objetos de una clase y enviar mensajes a otros objetos. Toda la semántica de un objeto se encuentra encapsulada dentro del mismo.

Homogeneidad de objetos

Todos los objetos tienen la misma categoría. No existen objetos especiales. Los propios objetos que dan soporte al sistema no deben ser diferentes del resto de los objetos.

Esta simplicidad conceptual hace que todo el sistema en su conjunto sea fácil de entender, y se elimine la desadaptación de impedancias al trabajar con un único paradigma.

6.3.2 Transparencia

El sistema hace transparente la utilización de recursos del entorno, y en general todas las características del sistema, especialmente la distribución y la persistencia, de las que se beneficiará particularmente el SGBDOO.

Distribución

El sistema es inherentemente distribuido y oculta al usuario los detalles de la existencia de numerosas máquinas dentro de una red. En términos de objetos, los objetos pueden residir en cualquier máquina del sistema y ser utilizados de manera transparente independientemente de cual sea ésta.

Persistencia

El usuario no debe preocuparse de almacenar ni recuperar los objetos en memoria secundaria explícitamente.

6.3.3 Heterogeneidad, portabilidad e Interoperabilidad

El sistema tiene en cuenta la posible variedad de máquinas en una red, y su arquitectura es especialmente concebida para facilitar la portabilidad y la heterogeneidad.

Por otro lado, al dar soporte directo a objetos, es decir, al gestionar directamente tanto la representación de los objetos como su utilización, soluciona el problema de la interoperabilidad. Los objetos no son conceptos que sólo existen dentro de los procesos del sistema operativo y que sólo son conocidos por el compilador que creó el proceso. Al ser gestionados por el propio sistema pueden "verse" unos a otros, independientemente del lenguaje y del compilador que se usó para crearlos.

6.3.4 Seguridad

El sistema dispone de un mecanismo de protección que permite controlar el acceso no autorizado a los objetos. Por supuesto, este mecanismo se integra de manera uniforme dentro del paradigma OO, y representa la base sobre la que se asentará la seguridad del SGBDOO.

6.3.5 Concurrencia

El sistema presenta un modelo de concurrencia que permite la utilización de los objetos aprovechando el paralelismo. Dentro de una misma máquina aprovecha la concurrencia aparente, y en sistemas distribuidos o multiprocesador la concurrencia real. Este modelo puede ser aprovechado por el SGBDOO para gestionar la concurrencia.

6.3.6 Flexibilidad

El sistema integral proporciona **flexibilidad estática**, que permite que el sistema sea personalizado para una aplicación o conjunto de aplicaciones en tiempo de compilación, enlace o carga, y **flexibilidad dinámica**, que supone que el sistema puede ser personalizado para cubrir o adaptarse a las necesidades de las aplicaciones en tiempo de ejecución.

6.3.7 Modelo de Objetos

El modelo de objetos adoptado por el sistema integral es intencionadamente estándar y recoge las características más utilizadas en los lenguajes de programación más populares, y en general más aceptadas entre la comunidad OO:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía de herencia y agregación
- Tipos
- Concurrencia
- Persistencia
- Distribución

Capítulo 7 Arquitectura del sistema integral orientado a objetos

En este capítulo se describe una arquitectura para un sistema integral orientado a objetos (SIOO) que proporciona las características mencionadas en el capítulo anterior. El sistema aquí presentado opta por un modelo de objetos único proporcionado por una máquina abstracta con arquitectura reflectiva que representa el corazón del sistema integral, y en la que se basa el sistema de gestión de bases de datos orientadas a objetos.

7.1 Sistema integral OO = máquina abstracta OO + sistema operativo OO

La estructura del SIOO está basada en una máquina abstracta orientada a objetos que proporciona soporte para un modelo único de objetos para todo el sistema. Una serie de objetos que forman el sistema operativo extienden la funcionalidad de la máquina proporcionando a los objetos diferentes facilidades de manera transparente.

7.2 Propiedades fundamentales de la arquitectura

Las propiedades más importantes que incorpora la arquitectura anterior en cada uno de sus dos elementos, la máquina abstracta y el sistema operativo, se reseñan a continuación.

7.2.1 Máquina abstracta orientada a objetos

Dotada de una arquitectura reflectiva, proporciona un modelo único de objetos para el sistema.

7.2.1.1 Modelo único de objetos

La máquina proporcionará al resto del sistema el soporte para objetos necesario. Los objetos se estructuran usando el modelo de objetos de la máquina, que será el único modelo de objetos que se utilice dentro del sistema.

7.2.1.2 Reflectividad

La máquina dispondrá de una arquitectura reflectiva [Mae87], que permita que los propios objetos constituyentes de la máquina puedan usarse dentro del sistema como cualquier otro objeto dentro del mismo.

7.2.2 Sistema operativo orientado a objetos

Estará formado por un conjunto de objetos que se encargarán de conseguir las propiedades que más comúnmente se asocian con funcionalidad propia de los sistemas operativos.

7.2.2.1 Transparencia: persistencia, distribución, concurrencia y seguridad

Estos objetos serán objetos normales del sistema, aunque proporcionarán de manera transparente al resto de los objetos las propiedades de **persistencia**, **distribución**, **concurrencia** y **seguridad**.

7.2.3 Orientación a objetos

Se utiliza en el sistema operativo al organizarse sus objetos en una jerarquía de clases, lo que permite la reusabilidad, extensibilidad, etc. del sistema operativo. La propia estructura interna de la máquina abstracta también se describirá mediante la orientación a objetos.

7.2.4 Espacio único de objetos sin separación usuario/sistema

La combinación de la máquina abstracta con el sistema operativo produce un único espacio de objetos en el que residen los objetos. No existe una división entre los objetos del sistema y los del usuario. Todos están al mismo nivel, independientemente de que se puedan considerar objetos de aplicaciones normales de usuario u objetos que proporcionen funcionalidad del sistema.

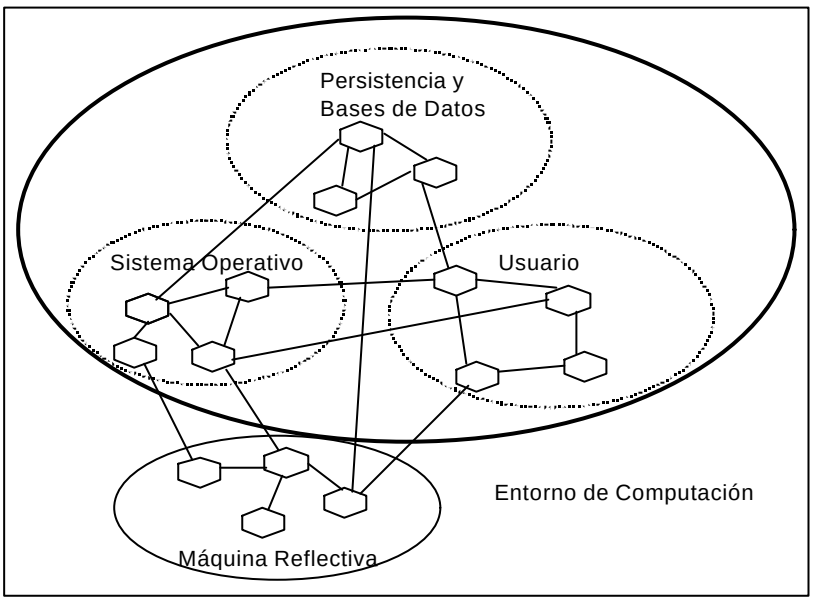


Figura 7.1. Entorno de computación compuesto por un conjunto de objetos homogéneos.

7.2.5 Identificador de objetos único, global e independiente

Los objetos dispondrán de un identificador único dentro del sistema. Este identificador será válido dentro de todo el sistema y será el único medio por el que se pueda acceder a un objeto. Además, el identificador será independiente de la localización del objeto, para hacer transparente la localización de un objeto al resto del sistema.

7.3 Modelo de objetos único para el sistema integral

El modelo de objetos que ha sido definido para el sistema sigue, intencionadamente, las especificaciones de los modelos de objetos de las metodologías de orientación a objetos más populares, como la de Booch [Boo94]. El objetivo es aprovechar las ventajas de los conceptos totalmente establecidos, así como soportar toda la semántica

del modelo de objetos que se usa desde la fase de análisis y diseño hasta la fase de implementación.

No todas estas características tienen que estar implementadas por el mismo elemento del sistema integral. De hecho, algunas de ellas, como la persistencia, es más interesante introducirlas de manera que (aunque transparente) su uso pueda ser opcional. Por tanto, se pueden dividir las características en grupos en función del elemento del sistema que se ocupe de implementarlas.

7.3.1 Parte del modelo en la máquina abstracta

La máquina abstracta se encargará de proporcionar las características fundamentales de este modelo de objetos, especialmente las más cercanas a los elementos normalmente encontrados en los lenguajes orientados a objetos.

- **Identidad única de los objetos**, basada en el uso de referencias.
- **Encapsulación**. Acceso a los objetos únicamente a través de sus métodos.
- **Clases**. Utilizadas para derivar tipos.
- **Relaciones de herencia múltiple** (es-un).
- **Relaciones de agregación** (es-parte-de).
- **Relaciones generales de asociación.**
- **Polimorfismo y comprobación de tipos en tiempo de ejecución.**
- **Manejo de excepciones.**

7.3.2 Parte del modelo en el sistema operativo

Se encargará de conseguir las propiedades que más comúnmente se asocian con funcionalidad propia de los sistemas operativos:

- **Concurrencia.**
- **Persistencia.**
- **Distribución.**
- **Seguridad.**

Hay que reseñar que bajo el epígrafe de "sistema operativo" se agrupa todo lo referente a la manera de conseguir esta funcionalidad. En general, esta se facilitará mediante un conjunto de objetos normales que extiendan la máquina abstracta para proporcionar a todos los objetos del sistema estas propiedades de manera transparente.

Sin embargo, en el diseño del sistema operativo, se prevé que pueda ser necesario por determinadas razones, incluir elementos propios de funcionalidad anterior como parte de la máquina abstracta, o al menos repartir la misma entre los objetos del SO y responsabilidades introducidas en la máquina.

7.4 Arquitectura de referencia de la máquina abstracta

En este apartado se describe una arquitectura de referencia de una máquina abstracta orientada a objetos con las propiedades necesarias para dar soporte a un sistema integral orientado a objetos.

7.4.1 Propiedades fundamentales de una máquina abstracta para un SIOO

Las características fundamentales de la máquina se muestran a continuación:

- **Modelo único de objetos**, que implementará la máquina.
- **Identificador único de objetos.**
- **Uniformidad en la OO**, único nivel de abstracción.
- **Interfaz de alto nivel**, independiente de estructuras de bajo nivel.
- **Juego reducido de instrucciones.**
- **Flexibilidad** (extensión).

7.4.2 Principios de diseño de la máquina

A continuación se describen brevemente los principios de diseño de la máquina abstracta [DAG99]:

- **Elevación del nivel de abstracción**, con una interfaz de instrucciones de alto nivel orientadas a objetos. Se eleva el nivel de forma homogénea a un nivel de orientación a objetos.
- **Conjunto reducido de instrucciones**. Contrariamente a lo que ocurre en la máquina virtual de Java [LY97], y más en la línea del espíritu inicial de la máquina Smalltalk [GR83], el conjunto de instrucciones debe reducirse al mínimo necesario para expresar los conceptos de la orientación a objetos. El conjunto de instrucciones proporcionará instrucciones declarativas que describen clases, referencias agregadas, y referencias asociadas; e instrucciones de comportamiento para el cuerpo de los métodos.
- **Sin mezcla de niveles de abstracción**. Existirá sólo un tipo de objetos, sin tipos primitivos especiales como en Java. La interfaz no utilizará estructuras de implementación de bajo nivel como una pila.

7.4.3 Estructura de referencia

A continuación se describen brevemente los elementos básicos en que se puede dividir conceptualmente la arquitectura (Figura 7.2). El objetivo de esta estructura es facilitar la comprensión del funcionamiento de la máquina, no necesariamente indica que estos elementos existan como tales entidades internamente en una implementación de la máquina.

Los elementos básicos que debe gestionar la arquitectura son:

- **Clases**, que se pueden ver como agrupadas en un **área de clases**. Las clases contienen toda la información descriptiva acerca de las mismas.
- **Instancias**, agrupadas en un **área de instancias**. Donde se encuentran las instancias (objetos) de las clases definidas en el sistema. Cada objeto será instancia de una clase determinada.
- **Referencias**, en un **área de referencias**. Se utilizan para realizar la invocación de métodos sobre los objetos y acceder a los objetos. Son la única manera de acceder a un objeto (no se utilizan direcciones físicas). Las referencias contienen el identificador del objeto al que apuntan (al que hacen referencia). Para la

comprobación de tipos, cada referencia será de un tipo determinado (de una clase).

- **Referencias del sistema**. Un conjunto de referencias especiales que pueden ser usadas por la máquina.
- **Jerarquía de clases básicas.** Existirán una serie de clases básicas en el sistema que siempre estarán disponibles, siendo la raíz de la jerarquía una clase denominada comúnmente *Object*. Serán las clases básicas del modelo único de objetos del sistema. Cada distribución del sistema establecerá los mecanismos necesarios para proporcionar la existencia de estas clases básicas. Normalmente serán implementadas directamente de manera primitiva por la propia máquina, pero en ciertos casos podrían estar implementadas con el lenguaje de programación del usuario.

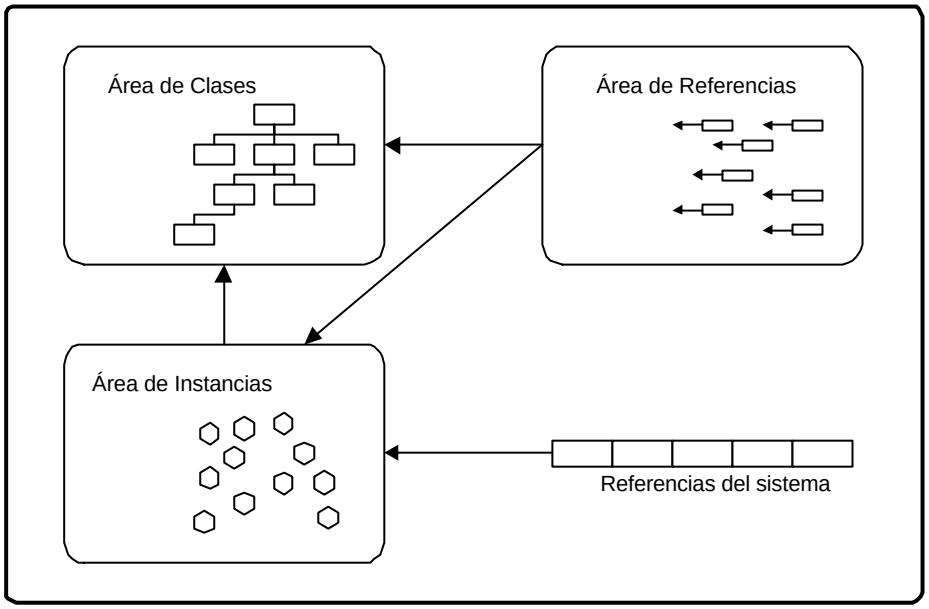


Figura 7.2. Estructura de referencia de una máquina abstracta para el sistema integral

7.4.4 Juego de instrucciones

El juego de instrucciones orientadas a objetos de alto nivel deberá ser independiente de estructuras internas de implementación. Para ello se describirá en términos de un lenguaje ensamblador.

7.4.4.1 Instrucciones declarativas

La unidad de descripción en este sistema son las **clases** y por ello, la arquitectura de la máquina abstracta almacena la descripción de las mismas en el **área de clases**.

Para la definición de clases, el lenguaje de la máquina abstracta ofrece un conjunto de instrucciones declarativas, cuyo resultado es la descripción de la información de una clase.

Estas instrucciones sirven para definir completamente y sin ambigüedad una clase, lo que supone identificar la clase de forma unívoca mediante un **nombre** que no esté asociado ya a alguna clase, especificar las **relaciones** entre distintas clases, tanto de **herencia**, como de **agregación** o de **asociación**, y, por último, definir la **interfaz** de la clase, o conjunto de métodos que la clase ofrece al exterior.

7.4.4.2 Instrucciones de comportamiento

Permiten definir el comportamiento de la clase, es decir, el comportamiento de sus métodos. La descripción del código que compone el cuerpo de un método supone la enumeración ordenada de un conjunto de instrucciones de entre las que se indican a continuación.

- Instrucciones para manipulación de objetos a través de las referencias
- Invocación de un método a través de una referencia. Son, en esencia, la única operación que se puede realizar en un objeto y suponen la ejecución del método invocado.
- Control de flujo. Instrucciones para controlar el flujo de ejecución de un método entre las que se encuentran la **finalización de un método**, los **saltos** y las **instrucciones de control de flujo para gestionar las excepciones**.

7.4.5 Ventajas del uso de una máquina abstracta

Las ventajas del uso de una máquina abstracta como ésta son básicamente la portabilidad y la heterogeneidad, y la facilidad de comprensión y desarrollo, que la hacen muy adecuada como plataforma de investigación en las tecnologías OO. El inconveniente de la pérdida de rendimiento por el uso de una máquina abstracta se ve contrarrestado por la disposición de los usuarios a aceptar esa pérdida de rendimiento a cambio de los beneficios que ofrece una máquina abstracta. Por otro lado, mejoras en el hardware y optimizaciones en la implementación de las máquinas minimizan aún más este problema.

7.5 Extensión de la máquina abstracta para proporcionar la funcionalidad del sistema operativo. Reflectividad

En general, la funcionalidad asignada al sistema operativo será proporcionada mediante objetos normales de usuario. Sin embargo, en algunos casos será necesario modificar de ciertas maneras la máquina, incluyendo el soporte que sea necesario para la implementación de esta funcionalidad. En general existen tres maneras de introducir esta funcionalidad en el sistema, que podrán usarse por separado o, lo que es más probable, en combinación. Además, casi siempre será junto con objetos normales de usuario que proporcionen la funcionalidad restante.

7.5.1 Modificación interna de la máquina

Se trata de realizar cambios internos en la máquina para incorporar la funcionalidad que se necesite. Estos cambios involucrarán una modificación en la semántica de las operaciones internas, o bien un cambio en la interfaz de la máquina (como por ejemplo la adición de una nueva instrucción).

7.5.2 Extensión de la funcionalidad de las clases básicas

En este caso se modifica la funcionalidad que proporcionan las clases básicas en el sistema. Se pueden añadir atributos y métodos, y también cambiar la semántica de métodos existentes (aunque no debería ser habitual). Por ejemplo, al añadirse un nuevo método a la clase raíz de la jerarquía de clases del sistema, automáticamente todos los objetos pasan a tener ese método a su disposición.

7.5.3 Colaboración con el funcionamiento de la máquina. Reflectividad

Permitir que objetos del sistema operativo (objetos normales) colaboren con la máquina cuando ésta no pueda realizar una tarea por sí misma, mediante una arquitectura reflectiva [Mae87]. La máquina tendrá una arquitectura en la que los objetos componentes de la máquina se exponen como si fueran objetos normales.

De esta manera, los objetos de la máquina pueden usarse como objetos normales, y a su vez pueden usar estos objetos. Cuando un objeto de la máquina no puede continuar por alguna razón, llama a un objeto del sistema operativo (mediante una excepción o por llamada directa). Este objeto colaborará con la máquina y posiblemente con otros objetos para resolver en conjunto el problema. El sistema puede verse como un conjunto de objetos que interactúan, todos con las mismas características. Algunos de ellos serán objetos de la máquina, otros serán objetos normales de usuario y algunos de estos últimos implementarán funcionalidad del sistema operativo. Sin embargo, el origen de un objeto es transparente para todos los demás, e incluso para sí mismo. Como ejemplo, la persistencia puede ser proporcionada con objetos de la máquina colaborando con objetos del sistema operativo para guardar instancias en el disco.

Capítulo 8 Modelo de persistencia del sistema integral orientado a objetos

Si hay una característica que distingue con diferencia a un SGBDOO es la persistencia. Podríamos decir que es una característica que nunca puede faltar en un gestor de base de datos, pero lo que si puede es cambiar la forma en la que el sistema proporciona esa persistencia. En este capítulo se analiza el modelo de persistencia proporcionado por el sistema integral subyacente. Con posterioridad (véase capítulo 10) se verá como este modelo condiciona el desarrollo del SGBDOO que nos ocupa.

8.1 Subsistema de persistencia del sistema integral orientado a objetos

La mayoría de los sistemas persistentes se han construido por encima de sistemas operativos ya existentes. Estos sistemas operativos convencionales no tuvieron entre sus objetivos de diseño el soporte para la persistencia. De hecho, la mayoría de los sistemas operativos únicamente tienen los ficheros como abstracción de la memoria a largo plazo [DRH+92]. El resultado es que normalmente se implementa el sistema de persistencia como una capa completa adicional al sistema operativo, con poca integración con el mismo y la pérdida de rendimiento consiguiente. Por otro lado, muchos sistemas se enfocan a dar soporte de persistencia a un determinado lenguaje [ACC81] específicamente. De esta manera se van añadiendo capas adicionales, una por cada lenguaje, con los problemas de la proliferación de capas de software (véase el capítulo 2).

Por todo ello parece claro que el camino a seguir es implementar la persistencia como propiedad fundamental de un sistema integral orientado a objetos [ADA+96]. Por supuesto, se deben soportar los objetos con toda la semántica del modelo de objetos del sistema: no sólo los datos estrictamente, si no con una semántica más amplia, que incluya como parte del estado de un objeto la computación, las relaciones con otros objetos, etc. De esta manera el soporte de persistencia estará disponible para todos los lenguajes, sin necesidad de un soporte adicional redundante para los mismos.

A continuación se describe la forma en que se introduce la persistencia en el sistema integral. Una descripción completa se puede encontrar en [Álv98].

8.2 Elementos básicos de diseño del sistema de persistencia

En esencia se trata de proporcionar la abstracción de un **espacio de objetos potencialmente infinito** en el que coexisten simultáneamente e indefinidamente todos los objetos del sistema, hasta que ya no son necesitados. Podría considerarse como una memoria virtual persistente para objetos. El usuario simplemente trabaja con los objetos en ese espacio virtual.

El sistema de persistencia deberá proporcionar con los recursos existentes en el entorno (a partir de la máquina abstracta orientada a objetos) la abstracción anterior.

Los puntos básicos del sistema de persistencia del sistema integral son referenciados a continuación.

8.2.1 Persistencia completa

Absolutamente todos los objetos creados en el sistema son persistentes por definición. Esto incluye también a los objetos del sistema operativo, que deben tener la misma categoría que los objetos de usuario. No es necesario almacenar y recuperar explícitamente los objetos, el sistema lo realiza de manera transparente. Sólo existe un único conjunto de operaciones para manipular los objetos.

8.2.2 Estabilidad y elasticidad

El espacio virtual persistente de objetos debe proporcionar las propiedades de estabilidad y elasticidad. La **estabilidad** es la capacidad de un sistema para registrar su estado de manera consistente en un medio seguro, de tal forma que el funcionamiento podría reanudarse a partir de ese punto en el futuro. Un sistema es **elástico** si puede reanudarse el funcionamiento con seguridad después de una caída inesperada del mismo, como por ejemplo, un fallo de alimentación.

8.2.3 Encapsulamiento de la computación

Se propone un modelo de objetos que además encapsule como parte de su estado la computación, lo que tendrá que ser perfilado por la parte de concurrencia del sistema operativo.

8.2.4 Identificador uniforme único

Se utilizará el identificador único de objetos de la máquina para referenciar los objetos, independientemente de su localización física. Este identificador es usado tanto por el usuario como por el sistema internamente para referenciar el objeto en memoria principal y en almacenamiento persistente. Esto tiene mucha similitud con el modelo de sistema operativo de espacio de direcciones virtual único [VRH93, SMF96], jugando el papel de las direcciones virtuales los identificadores de los objetos. Este espacio único se extiende a la memoria secundaria, formando un sistema de almacenamiento con sólo un nivel (*single level store* [KEL+62]).

8.2.5 Eliminación explícita de objetos

A la hora de eliminar los objetos se puede optar por borrado explícito o bien por recolección de basura. La recolección de basura siempre implica una sobrecarga adicional en el sistema que se complica aún más si los objetos pueden residir además de en el área de instancias en el almacenamiento persistente. Es por ello que en este sistema integral inicialmente se ha optado por borrado explícito [Álv98].

8.2.6 Ventajas

La utilización de un esquema como el que se ha propuesto conlleva unas ventajas, de las cuales destacan las siguientes:

Permanencia automática

Una vez que se crea un objeto en el sistema, este permanece el tiempo necesario hasta que no sea necesitado más. No es necesario preocuparse de almacenar el objeto en almacenamiento secundario si se desea que no se pierda su información. Esto facilita la programación.

Abstracción única de memoria. Uniformidad

La única abstracción necesaria para la memoria es el objeto. El sistema de manera transparente conserva la información del objeto siempre. Al eliminar la dualidad de la abstracción entre memoria de largo plazo y de corto plazo, y sustituirlas por una única abstracción de espacio virtual persistente de objetos se facilita la labor de los programadores. Abstracciones como fichero ya no son necesarias (aunque podrían implementarse como objetos normales, sí se desea). Simplemente se trabaja con un único paradigma, que es el de la orientación a objetos.

Permanencia de la computación

Al encapsular el objeto también el procesamiento, no sólo se conservan los datos, sino también la computación [KV92, TYT92]. Se mantiene totalmente el estado: datos y proceso. Esta característica se suele considerar ya parte integrante de un modelo de objetos [Boo94].

Memoria virtual persistente distribuida

Se crea fácilmente una verdadera memoria virtual distribuida de objetos al utilizar un identificador único de objetos. Simplemente basta con utilizar un mecanismo que haga que este identificador sea diferente para todos los objetos, independientemente de la máquina en que se crearon. Se utilizará este identificador único en conjunción con el mecanismo de distribución para localizar en todo el sistema distribuido un objeto dado.

8.3 Implementación de la persistencia en el sistema integral

Las referencias pueden considerarse como punteros a otros objetos, aunque se diferencian de los punteros convencionales por el hecho de que siempre son válidos, al utilizarse el identificador único del objeto.

El elemento fundamental de la máquina es el área de instancias, que almacena los objetos. Este área es el equivalente a la memoria principal de los sistemas convencionales.

8.3.1 Área de instancias virtual persistente

En esencia, se implementa la persistencia como una extensión del área de instancias. Es decir, lograr la ilusión de un **área de instancias virtual**[1] (**memoria virtual**), utilizando para ello un almacenamiento secundario para hacer persistir los objetos, guardándolos cuando no estén (o no quepan) en el área de instancias. Todo ello de manera totalmente transparente para el resto del sistema.

[1] Hay que recalcar que, dentro de la semántica de un objeto, un elemento fundamental es la clase a la que pertenece. Por tanto, aunque se haga referencia únicamente a los objetos, se entiende implícitamente que los mismos mecanismos se aplican para hacer persistir las clases de los objetos. Es decir, la persistencia se aplica de manera análoga al área de clases.

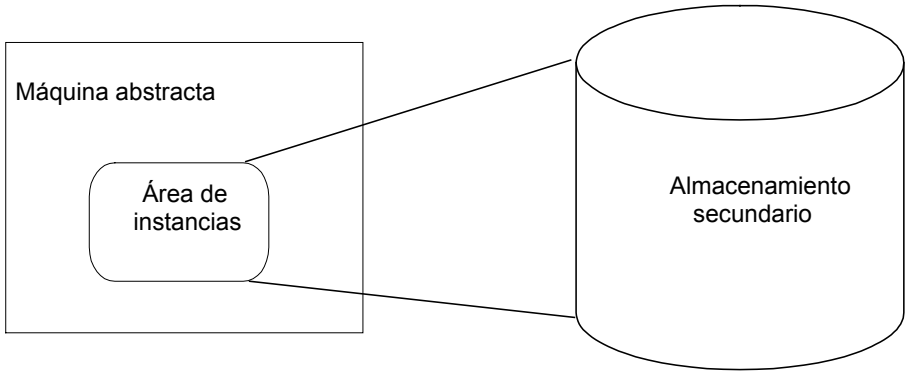


Figura 8.1. Área de instancias virtual

Se unifican los principios de memoria virtual tradicionales de los sistemas operativos con los de persistencia. El sistema proporciona un espacio de memoria único en el que residen todos los objetos (persisten), virtualmente infinito.

8.3.2 Identificador uniforme de objetos igual al identificador de la máquina

Como identificador del objeto en almacenamiento persistente se utilizará de manera uniforme el propio identificador del objeto dentro de la máquina. Es decir, se usa en todo momento un único identificador del objeto, que siempre es válido en cualquier situación en la que se encuentre el objeto: tanto en almacenamiento persistente como en el área de instancias.

8.3.3 Mecanismo de envío de mensajes y activación del sistema operativo por reflexión explícita

Cuando se envía un mensaje a un objeto (en general cuando se necesite acceder a un objeto por alguna razón), se proporciona a la máquina una referencia al mismo (identificador del objeto). La máquina usa este identificador para localizar el objeto en el área de instancias. Si el objeto no está en esta área debe generarse una excepción o mensaje que permita intervenir al sistema operativo. Mediante la reflectividad se hace una reflexión explícita que cede el control a un objeto del sistema operativo.

8.3.4 Objeto “paginador”

Se activará un objeto colocado al efecto por el sistema operativo. La función de este objeto es precisamente ocuparse del trasiego de los objetos entre el área de instancias y el almacenamiento persistente. Este objeto y el mecanismo en general son similares al concepto del paginador externo para memoria virtual de sistemas operativos como Mach [ABB+86] y, por tanto, se le denominará objeto “**paginador**”.

8.3.5 Carga de objetos

Para localizar el objeto en almacenamiento persistente se utilizará el identificador del objeto proporcionado por la referencia usada en el envío del mensaje.

Una vez localizado el objeto, se debe colocar en el área de instancias, reconstruyendo en efecto el estado completo del mismo. Para ello se invocarán métodos adecuados en la meta-interfaz del objeto que representa reflectivamente el área de instancias.

Al finalizar el trabajo del objeto "paginador", debe regresar el control a la máquina para que continúe con su funcionamiento normal, al estar ya el objeto en el área de instancias.

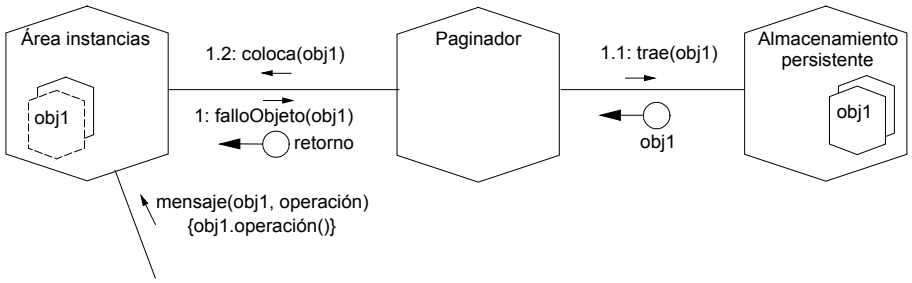


Figura 8.2. Carga de un objeto en el área de instancias.

8.3.6 Reemplazamiento

En caso de no existir sitio en el área de instancias para situar un objeto, bien en la creación de un nuevo objeto, bien por necesitarse la carga del mismo por el motivo anterior, debe liberarse espacio en la misma. Por un mecanismo similar al anterior se activará el objeto paginador y seleccionará uno o varios objetos que llevará al almacenamiento persistente, registrando su estado y liberando así el espacio necesario. Para seleccionar los objetos a reemplazar puede utilizarse otro objeto que proporcione la estrategia de reemplazo.

Hay que destacar que dado que el objeto encapsula la computación, también se conserva el estado de la misma.

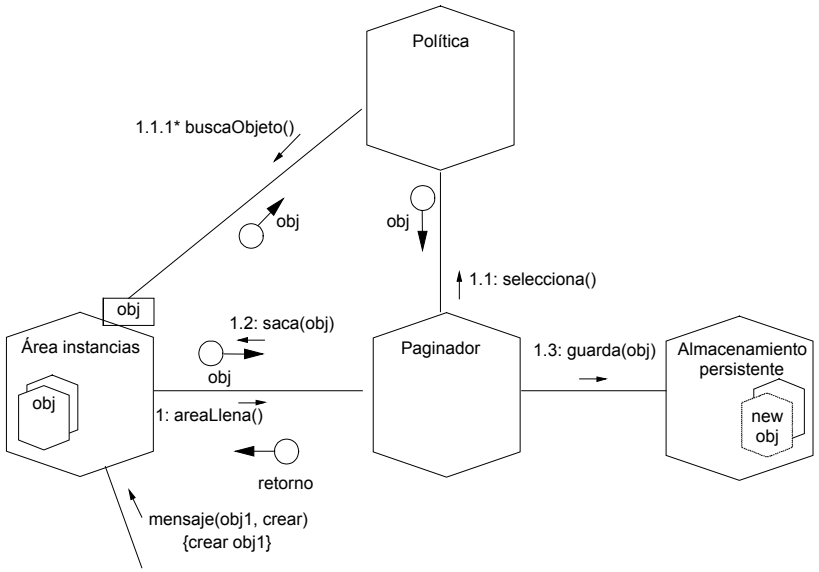


Figura 8.3. Reemplazamiento de un objeto en el área de instancias

8.3.7 Manipulación interna

Es necesario habilitar un mecanismo que permita a este objeto del sistema manipular el área de instancias, para lo cual se aprovecha la arquitectura reflectiva de la máquina. Algo similar debe hacerse para poder registrar el estado de los objetos, al tener que

acceder a su representación interna, bien por el método anterior, bien definiendo unas operaciones primitivas disponibles en todo objeto que lo permitan.

Capítulo 9 Otras aportaciones del sistema integral orientado a objetos

La construcción de un SGBDOO sobre un sistema integral como el descrito en los capítulos anteriores se ve favorecida por las características ofrecidas por el sistema operativo y/o máquina abstracta del sistema. De entre esas características destacaremos cuatro: persistencia, concurrencia, distribución y seguridad. En el capítulo anterior se analizó el modelo de persistencia del sistema. En este capítulo se analizan las características del resto de subsistemas, y con posterioridad (véase capítulo 10) se puntualizarán las repercusiones de las mismas para la construcción del SGBDOO.

9.1 Subsistema de protección

El subsistema de protección se convierte en el núcleo de seguridad para el sistema integral. Un buen mecanismo de protección deberá ser lo suficientemente flexible para que puedan implementarse sobre él diversas políticas de seguridad. Para una información completa sobre el mecanismo de protección y las políticas de seguridad véase [Día00].

9.1.1 Modelo de protección

El modelo básico de protección del sistema integral está basado en capacidades. En concreto, se ha diseñado un tipo de capacidades denominadas **capacidades orientadas a objetos**, que aprovechan parte de las ventajas que proporciona el uso de una máquina abstracta orientada a objetos con su lenguaje ensamblador orientado a objetos. El propio diseño del repertorio de instrucciones del lenguaje es seguro: garantiza que la única manera de manipular los objetos (datos, incluidas las propias capacidades) es a través de las instrucciones definidas al efecto. Es imposible, por tanto, acceder a la representación física de los objetos y cambiar de manera arbitraria éstos. Es decir la propia máquina abstracta evita la posibilidad de falsificación de las capacidades.

La idea consiste en **implementar las capacidades integrando la información de protección con la referencia al objeto** que existe en la máquina abstracta, añadiendo a ésta los permisos que el poseedor de la referencia tendrá sobre el objeto al que se hace referencia. A partir de este momento, las referencias de la máquina siempre incorporan la información de protección sobre el objeto al que apuntan. Es decir, las referencias se convierten en capacidades[1]. En este sentido, se tiene un sistema basado en capacidades puras, en las que las capacidades combinan denominación (referencia a un objeto) y protección, y ésta es la única manera de acceder a los objetos.

[1] En lo sucesivo se utilizarán las denominaciones de referencia o capacidad indistintamente, ya que describen el mismo concepto

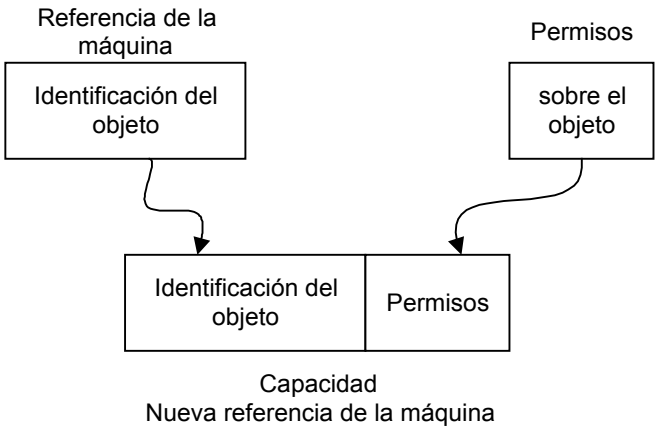


Figura 9.1. Estructura interna de una capacidad

9.1.1.1 Capacidades con número de permisos variable

En un SIOO los objetos tienen la semántica y el nivel de abstracción tradicional de las metodologías. Por tanto pueden tener un conjunto arbitrario de operaciones, tanto en número como en tipo. La homogeneidad y la protección de granularidad fina que necesita un SIOO requiere que todos los métodos de cualquier objeto sean protegidos de forma individual.

Parece evidente que un mecanismo de protección flexible y completo consiste en un tipo de capacidades que dé soporte a un número variable de permisos (tantos como métodos tenga el objeto). Una ventaja del SIOO es que las referencias son abstracciones de alto nivel del sistema, y como tales son manejadas por las aplicaciones. Estas desconocen la implementación física interna de las capacidades y no se hace ninguna suposición sobre cómo puede ser ésta (por ejemplo, que tenga un tamaño determinado). Esto permite que sea posible implementar esta abstracción con un número variable de permisos sin afectar en absoluto al resto del sistema.

Se utiliza, por tanto, un mecanismo de protección al nivel de todas las operaciones de cualquier objeto. El número de permisos almacenados en la capacidad será variable en función de los métodos que tenga el objeto al que apunte la referencia/capacidad.

9.1.1.2 Salto de la protección

En un SIOO como el planteado, con objetos de cualquier granularidad, muchos de estos objetos van a formar parte de objetos locales a otros de mayor granularidad. Estos objetos normalmente sólo serán utilizados por sus propietarios (objeto que los creó), y no van a ser accedidos por otros objetos que su creador. Es decir, estos objetos no van a ser compartidos. En estos casos la necesidad de protección de los métodos de estos objetos no suele ser necesaria, y puede dejarse que se tenga permiso de acceso a todos sus métodos.

En los casos en los que la capacidad tiene todos los permisos activos, y por tanto el mecanismo de protección permitirá realizar cualquier operación, se utiliza un permiso especial que indica “Salto de protección”. En estos casos el mecanismo de protección no comprobaría los permisos, si no que directamente pasaría a la ejecución del método en cuestión.

9.1.2 Implantación en la máquina abstracta

La comprobación de la protección se implanta como parte integrante de la propia máquina abstracta del sistema integral.

Dado que la máquina ya tiene integrado un mecanismo para realizar el paso de mensajes entre los objetos a través de referencias, es sencillo introducir la comprobación de permisos en el propio paso de mensajes. Así, cuando se envía un mensaje a un objeto, la máquina simplemente debe hacer un paso adicional: comprobar que dentro de los permisos que están en la capacidad se encuentra el necesario para invocar el método solicitado. Si es así se procede normalmente, en caso contrario se devuelve una excepción de protección al objeto que intentó la llamada.

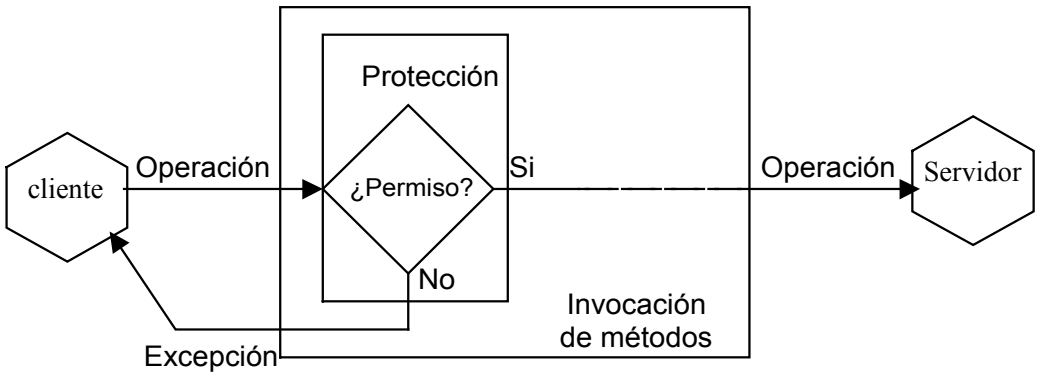


Figura 9.2. Integración del mecanismo de protección junto con la invocación de métodos, dentro de la máquina abstracta

Con esto se consigue mantener la uniformidad del modelo de objetos existente, puesto que externamente no se cambia la funcionalidad anterior del sistema. Por otro lado se consigue de una manera sencilla e integrada en el corazón del sistema: todas las operaciones en el sistema (pasos de mensajes) están sujetas al mecanismo de protección y además es imposible saltarse el mecanismo ya que es parte del nivel más bajo del sistema.

9.1.3 Modo de operación del mecanismo

En este apartado, se resume cómo funciona el modelo de protección elegido y se integra dentro de la máquina de forma natural proporcionando una protección no esquivable, a la vez que totalmente flexible.

El mecanismo de protección funcionará de la siguiente manera:

1. Para poder realizar una operación sobre un objeto es necesario tener su capacidad. La consecución de ésta es transparente al modelo de objetos puesto que está incluida en la referencia. En un lenguaje de programación OO, para poder operar sobre un objeto es necesario contar con una referencia a éste.
2. Existen dos modos de obtener una capacidad: obteniéndola al crear un objeto o recibiéndola como parámetro en una invocación del método. Este mecanismo es totalmente transparente, puesto que coincide con la forma de obtener referencias en un modelo de objetos.
3. En una capacidad aparece información acerca de los permisos existentes sobre cada uno de los métodos del objeto. Si la capacidad otorga permisos sobre todos

los métodos, entonces el permiso especial de "salto de protección" estará activo, en caso contrario está inactivo. Esta información forma parte de los atributos de la referencia al objeto.

4. Cuando se realiza una invocación a un método, la máquina la resuelve, localizando el objeto, la clase a la que pertenece y el método invocado. Antes de transferir la ejecución al método del objeto invocado, la máquina comprueba el estado del permiso de "salto de protección" localizado en la referencia/capacidad. Si éste está activo se pasará a realizar la invocación. Si no lo está, la máquina comprueba el estado del permiso correspondiente al método invocado, situado en la referencia/capacidad. Si éste está activo, se efectúa la invocación. En caso contrario se lanza una excepción de protección.
5. Un objeto A que posee una capacidad sobre otro B puede pasársela a un tercero C, para que C tenga también capacidad sobre B. El modo de pasar capacidades será como parámetro en una invocación a uno de sus métodos. Este mecanismo es transparente puesto que funciona como las referencias en un modelo de objetos.
6. Un objeto nunca podrá modificar ni aumentar el número de permisos de una capacidad por sí mismo (falsificación). Puesto que los permisos son atributos privados de las capacidades, no existe modo alguno de que sean manipulados, excepto a través de las operaciones definidas al efecto por la máquina abstracta.
7. Un objeto deberá poder disminuir el número de permisos de una capacidad. Esta facilidad podrá aprovecharse para enviar capacidades restringidas a otros objetos, de los que sólo interesa que realicen un número de operaciones más reducido y de esta manera poder cumplir el principio de mínimo privilegio. Esto se consigue mediante la instrucción que permite restringir el acceso a un método en una capacidad (Restringir <capacidad>, <método>).
8. Cuando se crea un objeto, el objeto creador recibe una capacidad con permisos para todos sus métodos. Esto implica que estará activo el permiso de "salto de protección".
9. Cuando un objeto restringe un permiso de una capacidad cuyo permiso de "salto de protección" está activo, entonces éste se desactiva y se crean permisos explícitos para todos los métodos, estando todos activos salvo el que se pretende restringir con dicha operación. Posteriores restricciones de otros permisos implicarán únicamente la desactivación del permiso susodicho.
10. La fusión de capacidades y referencias implica la modificación interna de algunas operaciones del modelo de objetos relacionadas con las referencias. El uso de referencias debe tener en cuenta ahora que también existe la información de protección. Así por ejemplo, instrucciones como la asignación, deberán copiar también los permisos. La creación de un objeto debe retornar una capacidad con permisos de acceso inicial, etc. Sin embargo estas modificaciones internas son transparentes para las aplicaciones, puesto que su semántica externa no cambia.

9.2 Subsistema de distribución

Para proporcionar distribución al sistema integral se ha optado por aprovechar la reflectividad de la máquina de forma que el paradigma de objetos no se rompe y los servicios pueden adaptarse fácilmente por modificación o reemplazamiento de los objetos.

El sistema operativo distribuido del sistema integral está construido como un conjunto de objetos de forma que la solicitud tradicional de servicios al sistema operativo se realiza invocando métodos de objetos que proporcionan esas funciones. El sistema de distribución permite que la invocación de esos métodos se realice de una forma transparente e independiente de la ubicación de los objetos (local o remota). Esto soluciona todos los problemas que se ocasionan debido a las diferentes situaciones en que se puedan encontrar los objetos, sin necesidad de que el programador haya tenido que escribir código para su tratamiento. Además, proporciona mecanismos para la migración de objetos entre los distintos nodos del sistema distribuido [ATD+00].

9.2.1 Características de la distribución

Las interacciones entre objetos están basadas en las invocaciones de métodos. Cada objeto posee un identificador único dentro del sistema e independiente de la ubicación del mismo.

Se emplea un modelo de objetos activo, de forma que cada objeto encapsula el estado de su computación. Esto facilita la migración de objetos ya que cuando se mueve un objeto no sólo se mueve su estado si no también su computación.

9.2.2 Migración de objetos

En este sistema la unidad de migración es el objeto. El movimiento del objeto se realiza de forma transparente, lo que se traduce en que el objeto continuará su computación en el nodo de destino como si el movimiento no hubiese tenido lugar, y las invocaciones hechas cuando el objeto se está moviendo se redireccionarán a la nueva localización.

La migración de objetos se soluciona en el meta-nivel de una forma transparente. Existe un conjunto de meta-objetos que son responsables de interrumpir cualquier actividad interna en el objeto, obteniendo su estado encapsulado en una cadena, enviando la cadena en un mensaje al destino elegido y reactivando el objeto allí. Permite emplear diferentes políticas para decidir cuándo, dónde y cómo moverlos [Álv00].

9.2.3 Invocación de métodos remota

El mecanismo de invocación de métodos proporcionado por la máquina abstracta es reescrito al meta-nivel para hacer posible la invocación de métodos remota. Un conjunto de meta-objetos implementan un nuevo mecanismo de invocación de métodos que reduce las dificultades introducidas por la posible dispersión de objetos, clases y referencias implicadas.

9.3 Subsistema de concurrencia

La concurrencia en el sistema integral es obtenida mediante el empleo de transacciones. Un servicio de transacciones debe registrar los efectos de todas las

transacciones entregadas y ninguno de los efectos de las transacciones que han abortado. El servicio de transacciones debe, por tanto, estar al corriente de aquellas transacciones que abortan, tomando las precauciones necesarias para que no afecten a otras transacciones que se estén ejecutando concurrentemente.

En un mundo ideal, todas las aplicaciones estarían construidas en base a transacciones. Todos los programas tendrían que seguir una férrea disciplina transaccional, dado que con que un solo programa fallase, podría corromperse todo el sistema. Una transacción que trabaja inconscientemente sobre datos corrompidos, producidos por un programa no transaccional, está construida sobre unos fundamentos a su vez corrompidos.

9.3.1 Transacciones software

El coste de un servicio de transacciones que funcione constantemente para todas las aplicaciones de un sistema distribuido es muy elevado, dado que se tienen que garantizar las propiedades ACID (*Atomicidad*, *Consistencia*, *Aislamiento*, *Durabilidad*) para todas ellas. En los sistemas reales se tiende a marcar de alguna manera cuáles son las fronteras de toda transacción, es decir, dónde empieza y termina el conjunto de operaciones que definen una transacción en particular.

Adicionalmente, todo servicio de transacciones tiene que tener soluciones para las siguientes cuestiones:

- Cuándo se hacen accesibles los efectos de una transacción.
- Qué unidades de recuperación se utilizarán en el caso de fallos.

Existen diferentes modelos de transacciones que de una u otra manera solucionan los problemas planteados: transacciones planas, transacciones encadenadas y transacciones anidadas. En general, todas ellas tienen una variante distribuida. Con el fin de no hacer de esta sección un estudio en profundidad de las transacciones, se describirán las transacciones planas, que servirán como base para un estudio posterior de su incorporación al sistema integral.

Una **transacción plana** es aquella en la que las fronteras que marcan el inicio y fin de las operaciones están al mismo nivel.

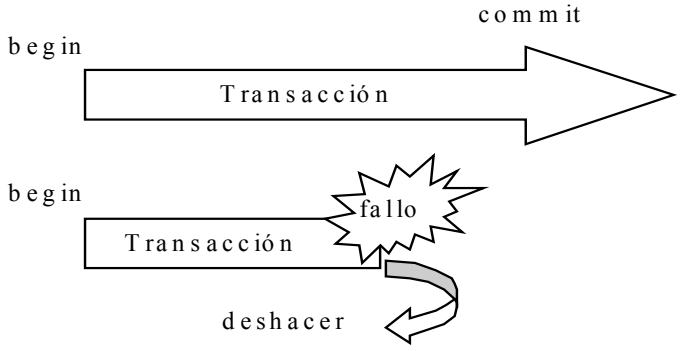


Figura 9.3. Transacción plana

La transacción empieza con un *begin_transaction* y finaliza con un *commit_transaction* o con un *abort_transaction*. Todas las acciones entre el inicio y el final se realizarán de manera indivisible.

Los programas que se construyen en base a transacciones planas están divididos en un conjunto relativamente amplio de transacciones con un pequeño número de operaciones, que se ejecutan una tras otra.

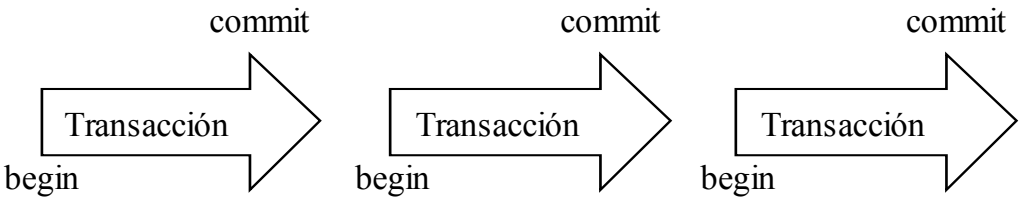


Figura 9.4. Construcción de un programa como conjunto de transacciones

Una **transacción plana distribuida** es aquella que permite que alguna o varias de las operaciones que la componen utilice recursos de diferentes nodos del sistema distribuido. El servicio de transacciones se hace notablemente más complejo, siempre con el objetivo de hacer este incremento de la complejidad transparente al programador.

El servicio de transacciones deberá expandirse por todos los nodos del sistema distribuido para hacer posibles las transacciones distribuidas y en uno de ellos deberá coordinarse el conjunto de actividades para asegurar las propiedades ACID de la transacción. Todo esto se consigue con la utilización de un **protocolo de entrega de dos fases** (*two-phase commit protocol*) [CDK94], que coordina la entrega o la cancelación de una transacción que involucra varios nodos.

9.3.2 Introducción de transacciones en el sistema integral

Un servicio de transacciones para el sistema integral orientado a objetos debe **mantener las propiedades ACID para conjuntos de invocaciones** a objetos (las operaciones posibles en el sistema integral). El código de los objetos cliente y objetos servidor no debe verse alterado por la introducción de las transacciones más allá del etiquetado de inicio y fin de transacción (los objetos cliente serán los que demandarán la realización de transacciones, que involucrarán a uno o más objetos servidor). Por lo tanto, los objetos cliente contendrán únicamente la lógica del problema a resolver. La reflectividad del sistema permitirá introducir la funcionalidad necesaria para implantar el sistema de transacciones de manera transparente para los objetos de usuario.

El servicio de transacciones será el encargado de invocar a los objetos servidores en el momento adecuado y en la secuencia adecuada en base a las solicitudes realizadas por los objetos cliente. Para ello, interceptará todas las invocaciones a objetos que se realicen entre los límites de inicio y fin de transacción. Dicha interceptación es posible en dos puntos: en el meta-objeto emisor, del objeto cliente y en el meta-objeto receptor, del objeto servidor.

En la figura siguiente se muestra una posible implantación del sistema de transacciones en el sistema integral. El servicio de transacciones es invocado de manera transparente desde los meta-objetos emisor ② que obtienen el control en cada invocación a método ①. Está compuesto por un conjunto de objetos encargados de mantener las propiedades ACID para todas las invocaciones que tienen lugar entre el inicio y el fin de la transacción y que involucran a un conjunto de objetos servidores. Con ese fin, pueden apoyarse en el servicio de persistencia del sistema integral ③. Finalmente, con el fin de hacer los cambios definitivos en los objetos servidores, el

servicio de transacciones dialoga con los meta-objetos receptores ④ responsables de sus invocaciones entrantes ⑤.

El servicio de transacciones aquí mostrado proporciona un grado más de indirección en la invocación a métodos. El objeto de usuario es consciente únicamente de una invocación como la señalada en la figura con una **A**. Pero lo que realmente ocurre es que el meta-objeto emisor intercepta la invocación en primera instancia, (podría ser, incluso, una invocación remota) y el servicio de transacciones lo hace en segunda instancia, para garantizar la correcta realización de las operaciones sujetas a la transacción.

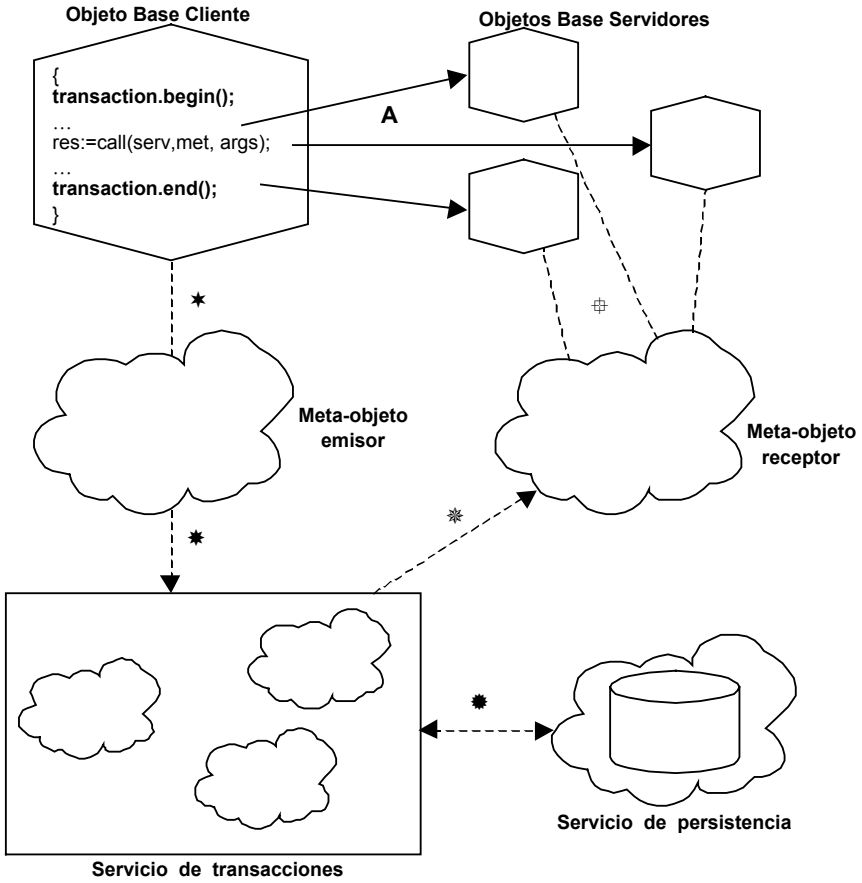


Figura 9.5. Diseño de un sistema de transacciones para el sistema integral

Capítulo 10 Arquitectura del SGBDOO

En el capítulo 4 se especificaron los requisitos que debía cumplir un SGBDOO ideal, y en el capítulo 5 se revisaron diferentes sistemas de gestión de objetos con el fin de descubrir si alguno de ellos cumplía dichos requisitos. Sin embargo, lo único que se encontraron fueron soluciones parciales a dichos planteamientos.

Con el fin de construir un SGBDOO que cumpla los requisitos anteriormente mencionados, en este capítulo se propone una arquitectura de referencia para dicho sistema. Una vez mostrada la arquitectura de dicho SGBDOO, se analizarán sus propiedades fundamentales, muchas de ellas directamente relacionadas con el modelo de objetos subyacente, y los principales componentes del mismo.

10.1 Estructura de referencia del SGBDOO

BDOviedo3 (Base de Datos para Oviedo3) [MCA96], es el nombre que recibe el SGBDOO a construir sobre el sistema integral. La figura siguiente (Figura 10.1) muestra una posible estructura de referencia para el SGBDOO BDOviedo3. En esta estructura de referencia se identifican a muy alto nivel los elementos en los que se puede dividir conceptualmente la arquitectura del sistema [MC98], y que serán estudiados con más detalle con posterioridad: el **motor** o **núcleo** de la base de datos, los **lenguajes** que permiten su manipulación y consulta, y por último, un conjunto de **herramientas visuales** que faciliten el trabajo con dicho sistema.

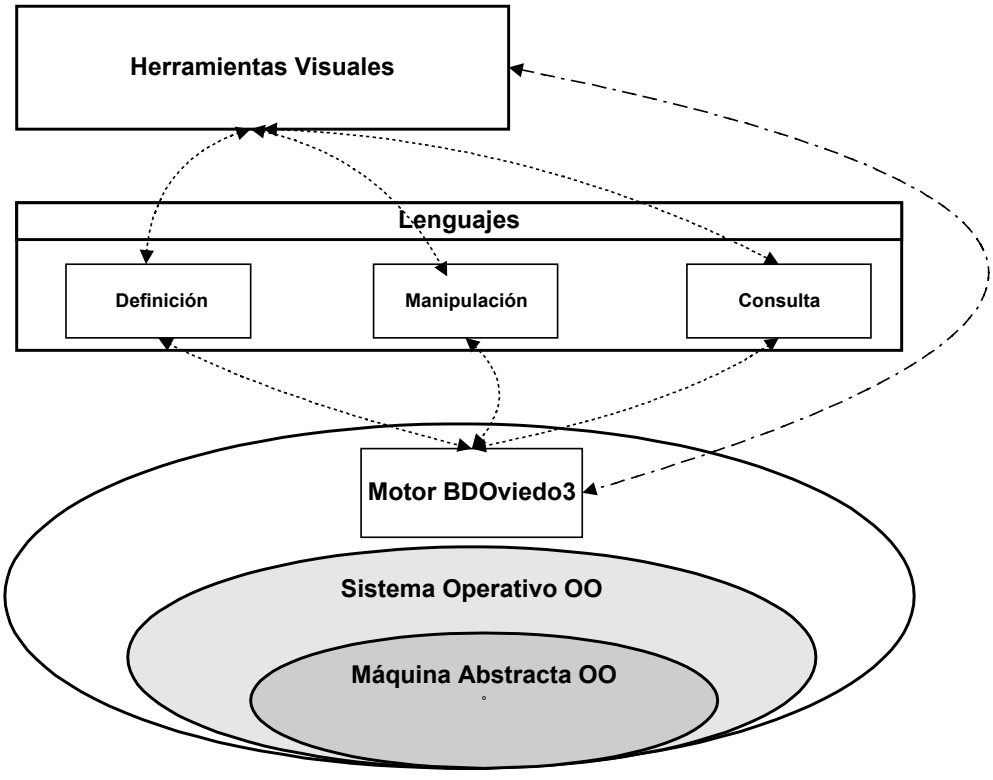


Figura 10.1. Estructura de referencia de BDOviedo3

10.1.1 Motor o núcleo del sistema

Representa la base del SGBDOO. Es el componente en contacto con el sistema operativo y la máquina abstracta, y comparte por tanto el mismo modelo de objetos. Será también el elemento del SGBDOO que más se beneficie y aproveche la flexibilidad ofrecida por el sistema integral.

Este núcleo debe proporcionar módulos, constituidos realmente por conjuntos de objetos, que completen las abstracciones ofrecidas por el sistema operativo cara a facilitar la funcionalidad de un gestor de bases de datos, además de otros módulos encargados por ejemplo de agilizar la recuperación de la información cara a conseguir un procesamiento de consultas óptimo [MAC+98a].

Una descripción más completa de este componente, que incluye los bloques funcionales básicos, será planteada en el apartado 10.3.

10.1.2 Lenguajes

Clásicamente, es básico en cualquier SGBD la incorporación de un lenguaje o lenguajes que permitan tanto la definición como la manipulación/consulta de los datos almacenados.

A la hora de seleccionar lenguajes para un SGBDOO existen diversas posibilidades a considerar. Sin embargo, la elección de los lenguajes iniciales a soportar por el sistema se ha visto condicionada por el afán de conseguir facilitar la portabilidad de las aplicaciones hacia otros gestores. Es por eso que se han seleccionado inicialmente los lenguajes propuestos por el estándar ODMG 2.0 en su *binding* para Java[CBB+97]: lenguaje de definición, lenguaje de manipulación y lenguaje de consulta.

La elección inicial de los lenguajes propuestos por el estándar (justificada en el capítulo 13), no implica que no puedan considerarse otras opciones con posterioridad. De hecho, la incorporación de nuevos lenguajes al SGBDOO, se traduciría en la incorporación al sistema de un traductor encargado de transformar el lenguaje empleado para la construcción de la aplicación, en el código entendido por la máquina abstracta más las funcionalidades ofrecidas por los objetos que constituyen el motor de la base de datos (Figura 10.2).

La inclusión de nuevos lenguajes, y en definitiva el proceso de traducción, se ve facilitada por la adopción por parte del sistema de un modelo de objetos que incluye las principales características de la orientación a objetos y comunes, por tanto, a los principales lenguajes de programación orientados a objetos, tal y como se comentó en el capítulo 7.

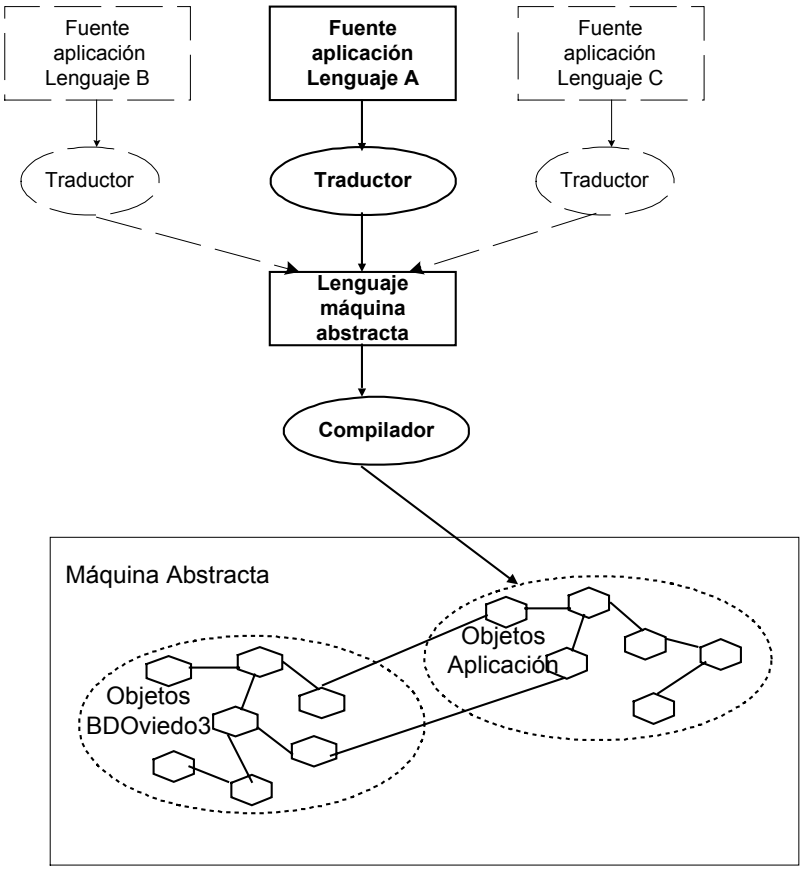


Figura 10.2. Arquitectura del SGBDOO BDOviedo3

10.1.3 Herramientas Visuales

Representan la capa de más alto nivel de la arquitectura y pretenden proporcionar un entorno de trabajo que permita una interacción eficiente, efectiva y segura para la mayor cantidad posible de usuarios, manteniendo siempre la consistencia con el paradigma de orientación a objetos. Para más información sobre los requisitos de la interfaz visual, así como las herramientas que se consideran imprescindibles para este SGBDOO véase el capítulo 14.

10.2 Propiedades básicas del motor derivadas del modelo de objetos

BDOviedo3, tal y como se aprecia en la Figura 10.1, es construido sobre un sistema integral orientado a objetos constituido por una máquina abstracta y un sistema operativo que proporcionan un modelo de objetos común al resto de elementos del sistema.

Las características de dicho modelo de objetos (véase capítulo 7), y que son soportadas por tanto por el SGBDOO BDOviedo3, no difieren de las que son consideradas de manera tácita como las más representativas de la orientación a objetos, y que fundamentalmente se encuentran recogidas en la metodología de Booch [Boo94]. Dichas características, que serán examinadas a continuación, se convierten en propiedades básicas del motor [MAC+98b].

10.2.1 Identidad única: identificador único de objetos

El sistema integral dispone de un mecanismo para distinguir un objeto de otros dentro del sistema. Dicho mecanismo se basa en asociar a cada objeto un identificador único y uniforme generado por la máquina abstracta, es decir, como identificador del objeto emplea el identificador del objeto dentro de la máquina. El identificador no será reutilizado una vez que el objeto sea eliminado.

Por otro lado, tradicionalmente los SGBDOO necesitan y emplean **identificadores de objetos** para identificar unívocamente a los objetos, y para establecer referencias entre ellos, siendo la elección del tipo de identificador a emplear un factor importante cara a su rendimiento (véase apartado 3.3.1).

En BDOviedo3 se conjugan ambas cuestiones, de forma que el identificador generado por la máquina abstracta va a ser empleado directamente como el identificador de los objetos de la base de datos, lo que hace que al SGBDOO no se le tenga que dotar de ningún mecanismo artificial, como ocurre en la mayoría de los existentes [CAI98, Jee99], para generar dichos identificadores. En este sistema todos los objetos tienen un identificador independientemente de que sean de la base de datos o no.

10.2.1.1 Identificador hardware

El identificador generado por la máquina abstracta es un identificador hardware con lo que no es necesario el paso intermedio producido por un identificador software, lo que se traduce a su vez en una mayor eficiencia.

10.2.1.2 No es necesario *swizzling*

El objeto siempre tiene el mismo identificador en memoria principal y en memoria secundaria, es decir, es uniforme, con lo que se consigue que ni el sistema integral ni el motor tengan que realizar el *pointer swizzling*.

10.2.2 Abstracción y encapsulamiento. Clases

El sistema integral emplea como unidad de **abstracción** y representación del **encapsulamiento** la **clase**, de forma que todos los objetos que tienen las mismas características se agrupan en una misma clase. La clase describe las características que tendrán todos los objetos de la misma: la representación del estado interno y su interfaz. De esta forma cuando se crea un nuevo objeto, se crea a partir de una clase de la que toma su estructura.

10.2.2.1 Modelo intensional

Las clases empleadas en el sistema integral no tienen una función extensional, es decir, la clase no gestiona su extensión o conjunto de instancias. Únicamente especifica las características para todos los objetos de la misma clase.

Esto exigirá que sea el propio motor del SGBDOO el que controle las extensiones de las clases.

10.2.3 Relaciones

El modelo de objetos del sistema, y por tanto el motor del SGBDOO, soporta el concepto de **jerarquía**, representada por las relaciones de **herencia** (es-un) y **agregación** (es-parte-de). Además, soporta también relaciones **generales de asociación**.

10.2.3.1 Herencia simple y múltiple

El modelo de objetos soporta el concepto de **herencia** (relación que se establece entre las diferentes clases de un sistema). Esta relación indica que una clase (subclase) comparte la estructura de comportamiento (las propiedades) definidas en otra clase (superclase), pudiéndose formar una jerarquía de clases, de forma que un objeto de una clase también es-un objeto de la superclase. Permite también que una clase herede de más de una superclase (**herencia múltiple).**

10.2.3.2 Agregación

El modelo de objetos soporta jerarquías de agregación o jerarquías es-parte-de que describen relaciones de agregación entre objetos para formar otros objetos de un nivel superior. Permiten definir un objeto en términos de agregación de sus partes.

En este sistema, los objetos agregados se crean cuando se crea el objeto que los contiene, y se destruyen cuando éste se destruye.

10.2.3.3 Asociación

El modelo de objetos soporta un tercer tipo de relación: la **relación de asociación** "asociado-con", que representa en general la existencia de otras relaciones diferentes a las de herencia o agregación entre objetos. La especificación de asociación se realiza en la definición de una clase, mediante la declaración de los objetos que participan de una relación de asociación con la clase que se está definiendo. A diferencia de los anteriores (agregados), estos objetos no se crean automáticamente al crear el objeto ni tampoco se destruyen automáticamente al borrarlo, sino que todo ello es responsabilidad del programador.

Esta relación de asociación permite establecer una relación de cardinalidad uno, de forma que, para expresar otro tipo de cardinalidades en las relaciones es necesario un mecanismo que lo permita. Es por tanto, responsabilidad del motor dotar al sistema de un mecanismo que permita tanto establecer relaciones de diferentes cardinalidades como comprobar su integridad.

10.2.4 Polimorfismo

En este modelo un **tipo** denota simplemente una estructura común de comportamiento de un grupo de objetos. Se identifica el concepto de tipo con el de clase, haciendo que la clase sea la única manera de definir el comportamiento común de los objetos. La existencia de tipos permite aplicar las normas de comprobación de tipos a los programas, y dicha comprobación permite controlar que las operaciones que se invocan sobre un objeto formen parte de las que realmente tiene.

El modelo de objetos soporta el **enlace dinámico,** de forma que el tipo del objeto se determina en tiempo de ejecución, y el sistema determinará cuál es la implementación de la operación que se invocará en función de cual sea el objeto utilizado. Esto es, el sistema soporta **polimorfismo** puesto que podemos aplicar el mismo nombre de operación a un objeto y en función de cual sea este objeto concreto el sistema elegirá la implementación adecuada de la operación.

10.2.5 Concurrencia

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo. Por tanto, entre las propiedades de un objeto está la de la **concurrencia**, es decir, la posibilidad de tener actividad independientemente del resto de los objetos.

La concurrencia en el sistema integral es obtenida mediante un servicio de transacciones que se encarga de mantener las propiedades ACID para conjuntos de invocaciones a objetos.

10.2.6 Persistencia

El modelo de persistencia con el que ha sido dotado el sistema integral proporciona una serie de características basadas en la abstracción única del objeto, que permiten que el motor pueda ser construido más fácilmente apoyándose en ellas.

El sistema integral proporciona persistencia, y además completa, para todos los objetos del sistema con lo que no es necesaria ninguna transformación adicional por parte del constructor del motor para conseguir la persistencia de sus objetos.

Mejoras exigibles por el sistema de gestión de base de datos a este subsistema de persistencia cara a la consecución de un buen rendimiento, tales como el agrupamiento de los objetos o algún método de acceso que acelere el acceso sobre la memoria secundaria serán proporcionados por objetos que se apoyen en dicho mecanismo básico.

10.2.6.1 Persistencia del comportamiento

El tratamiento de los métodos y en especial el almacenamiento de los mismos ha sido una cuestión que ha sido tratada de forma diferente por los distintos sistemas de gestión de objetos [Cat94]. Así, algunos sistemas como POET almacenan los métodos en ficheros externos binarios ajenos a la base de datos, que son enlazados con los objetos cuando se accede a éstos. Por el contrario, otros sistemas como por ejemplo O_2 o Jasmine [CAI98], almacenan el código binario dentro de la propia base de datos.

En BDOviedo3, el almacenamiento de los métodos se realiza dentro del propio sistema de persistencia del sistema integral, ya que éste proporciona persistencia no sólo para los datos si no también para los métodos o comportamiento del objeto, estando de esta manera sujetos a los criterios de seguridad y concurrencia del propio sistema. Es también el propio sistema de persistencia el que se encarga de enlazar el código asociado a cada objeto, sin necesidad de que el motor implemente ningún mecanismo para dicho proceso, al estilo del enlazador dinámico de funciones de MOOD [DAO+95].

10.2.6.2 Eliminación con borrado explícito

El sistema de persistencia, tal y como se comentó en el capítulo 8, ha optado inicialmente por borrado explícito en lugar de recolección de basura, sin embargo no se descarta la inclusión de un recolector de basura con probada eficacia.

10.2.6.3 Agrupamiento

El objetivo del agrupamiento (véase capítulo 3) es reducir el número de operaciones de entrada/salida sobre disco, que en definitiva son las que ralentizan el acceso a la información almacenada, mediante la agrupación de objetos que se encuentran relacionados de alguna manera. Es por esto que éste puede ser también un factor clave a la hora de conseguir un buen rendimiento para el sistema.

Inicialmente el subsistema de persistencia no proporciona posibilidad de agrupamiento. Sin embargo, podría proporcionarlo realizando algunas modificaciones en el objeto ‘paginador’, mencionado en el capítulo 8, encargado del trasiego de los objetos entre el área de instancias y el almacenamiento persistente.

Una posible adaptación del objeto 'paginador' con este fin puede ocasionar dos modificaciones en el objeto. Por un lado, en el diseño planteado del sistema de persistencia, el objeto 'paginador' trasladaba un único objeto independientemente de su tamaño entre el almacenamiento persistente y el área de instancias. Entonces, es posible, que el objeto 'paginador' se modifique para realizar el proceso con un conjunto de objetos.

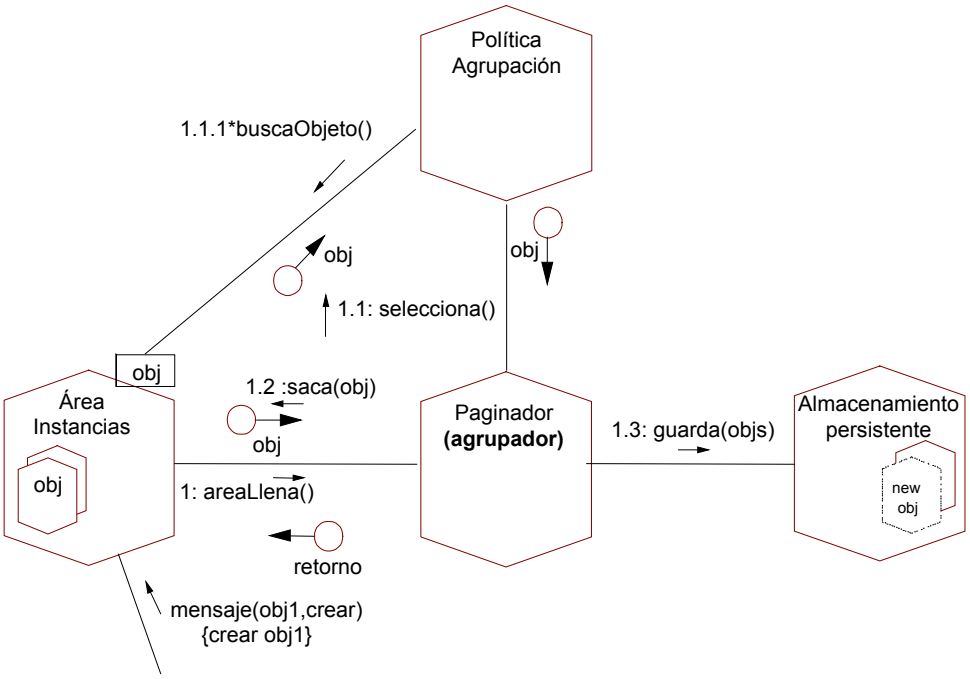


Figura 10.3. Reemplazamiento de un objeto en el área de instancias teniendo en cuenta criterios de agrupación

Por otro lado, será necesario conocer cual es el criterio o política de agrupación (por clase, por referencia, por índices, etc.) a emplear para que el 'paginador' se encargue de agrupar los objetos que ha de trasladar entre el área de instancias y la memoria persistente.

10.2.7 Seguridad

El mecanismo de protección del sistema integral, descrito brevemente en el capítulo 9, es de muy bajo nivel, pero sentará la base sobre la que el motor del SGBDOO pueda construir su propio modelo de autorización. Esta construcción se verá beneficiada sobre todo de la granularidad a nivel de método que proporciona el mecanismo, así como por la posibilidad de disponer de capacidades con el número de permisos variable (tantos como métodos tenga el objeto). Por otro lado, la transparencia del mecanismo al garantizar automáticamente la comprobación de permisos en la invocación de un método (generando la excepción correspondiente si la invocación no es pertinente) evita tener que recurrir a una autorización procedimental para realizar dichas comprobaciones.

Este modelo de protección aporta una ventaja fundamental a BDOviedo3 con respecto a la mayoría de los SGBDOO, ya que generalmente éstos consideran los objetos a muy bajo nivel (segmentos de memoria), sin considerar su semántica, con lo que dan una serie de permisos como lectura o ejecución pero sobre todos los métodos que tenga el objeto. Por ejemplo, en el caso se GemStone la asignación de permisos se hace por segmentos, de forma que a todos los objetos de un segmento se les asignan los

mismos permisos; entonces para cambiar los permisos de los objetos el usuario puede transferir objetos de un segmento a otro. Esos permisos además son únicamente o de lectura o de modificación.

Con el modelo de protección de BDOviedo3 se especificará la capacidad para la ejecución de determinados métodos del objeto, con lo que la autorización se realizará a nivel de las operaciones definidas sobre el objeto.

10.2.7.1 Consideraciones básicas sobre la autorización

En cualquier caso, independientemente del modelo existen una serie de cuestiones a considerar que son básicas [BM93], como la gestión de usuarios, y las repercusiones de las propias características del modelo de objetos.

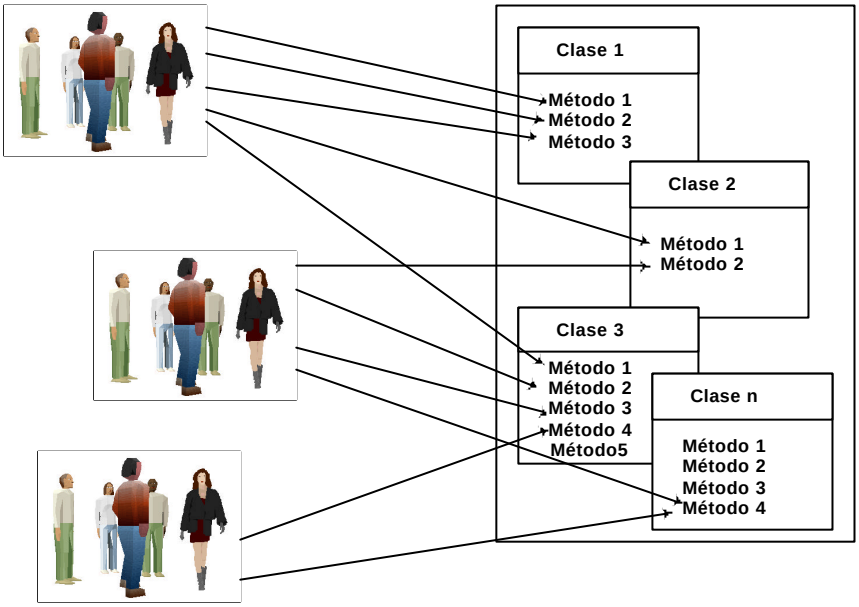


Figura 10.4. Autorización con granularidad a nivel de método

Gestión de los usuarios

Hace referencia a la gestión de los usuarios que pueden acceder al sistema gestor. Esto puede implicar la creación de grupos de usuarios a los que se les asignan los permisos, y de perfiles, de forma que un usuario puede añadir un grupo a su perfil y obtener implícitamente todas las autorizaciones otorgadas a dicho grupo. Entre estos grupos de usuarios tendrá que haber uno dotado con los permisos necesarios para la modificación de las definiciones de las clases. Todo esto además demanda la existencia de un lenguaje de control que proporcione una serie de operaciones como son otorgar y revocar permisos a los usuarios sobre las clases.

En vez de emplear un sistema de gestión de usuarios propio para el SGBDOO, otra posibilidad sería crear un sistema de gestión de usuarios general para todo el sistema integral y compartido con el sistema operativo, de forma que se asignen permisos a los usuarios sobre los objetos independientemente de que pertenezcan a la base de datos o no.

Características del modelo de objetos

De entre todas las características de la orientación a objetos se va a destacar la relevancia que puede tener en el modelo la existencia de jerarquías de herencia y de jerarquías de agregación.

- **Jerarquías de herencia**. Es decir, hay que decidir si las autorizaciones otorgadas a un usuario sobre una clase se deben heredar por todas sus subclases, o por el contrario cada clase es responsable de la privacidad de sus propios datos (ORION). Esta segunda opción parece la más lógica aunque requiera más verificaciones, por ejemplo, en una consulta sobre toda la jerarquía de clases.
- **Objetos complejos.** Hay que tener en cuenta que la autorización sobre una clase compuesta C implica la misma autorización sobre todas las instancias de C y sobre todos los objetos que sean componentes de C. Sin embargo, hay que considerar que el sujeto recibe la autorización sobre unas determinadas instancias de las clases componentes, no sobre todas las instancias de las clases componentes.

10.2.8 Distribución

El sistema operativo del sistema integral, como se comentó en el capítulo 9, incorpora un servicio de distribución que permite la invocación de métodos remota, y facilita la migración de objetos, siendo la unidad de migración el objeto. Este servicio puede ser la base para la conversión del SGBDOO en un sistema distribuido [MAO+00], al favorecer la consecución de algunas de las funcionalidades de los SGBD distribuidos.

10.2.8.1 Procesamiento distribuido

El procesamiento de consultas en bases de datos distribuidas se basa en generar y ejecutar estrategias para descomponer la consulta en operaciones que son enviadas a varios nodos, en la ordenación de estas operaciones, y en el ensamblaje de los resultados parciales para construir el resultado final de la consulta, y siempre teniendo en cuenta que los objetos pueden estar formados por otros objetos que se encuentran distribuidos sobre diferentes nodos.

El procesamiento distribuido sería posible en el SGBDOO gracias a la posibilidad de invocación remota de métodos que nos proporciona ya el sistema de distribución del sistema operativo, así como a la transparencia de localización de los objetos favorecida en gran medida por la existencia de un identificador de objetos independiente de la ubicación física de los mismos.

10.2.8.2 Duplicidad de los objetos

Una de las características de un SGBDOO distribuido es la posibilidad de que los objetos aparezcan duplicados en más de un sitio pero de forma totalmente transparente para el usuario. Esta posibilidad incrementa la confiabilidad del sistema de forma que si los ordenadores de un determinado nodo se quedan fuera de servicio el resto de la red puede seguir funcionando. Los usuarios no dependen de la disponibilidad de una sola fuente para sus datos. Esto además permite colocar los datos cerca del punto de su utilización, de forma que el tiempo de comunicación sea más corto reduciéndose el tráfico de la red.

En el SGBDOO sería posible duplicar los objetos, incluyendo no sólo el estado de los objetos sino también la computación. Para ello se empleará un mecanismo similar al empleado para permitir la migración de objetos, pero a diferencia de éste no elimina la copia original del objeto.

10.2.8.3 Escalabilidad

Los sistemas distribuidos permiten variar su tamaño de un modo sencillo. Se pueden agregar ordenadores adicionales a la red conforme aumentan por ejemplo, el número de usuarios y la carga de procesamiento.

La escalabilidad que nos podría proporcionar el SGBDOO en función de las posibilidades ofrecidas por el sistema operativo distribuido son:

- **Estática.** Proporciona un procesamiento distribuido de los objetos, basándose en una configuración inicial escalada en función del número de usuarios, el volumen de información y la futura carga inicial.
- **Dinámica.** Posibilita la migración de la información de forma dinámica a otros gestores de bases de datos, acudiendo a la reflectividad del sistema, en función de un meta-objeto que indica la carga del sistema.

10.3 Funcionalidad básica proporcionada por el motor

El núcleo del SGBDOO BDOviedo3 lo constituye el motor, que será el encargado de proporcionar la funcionalidad básica del SGBDOO (véase capítulo 3). Y tal y como se ha visto en el apartado anterior, el sistema integral proporciona unas propiedades básicas al sistema, que serán la base de la que partirá el motor para proporcionar el resto de funcionalidad del gestor.

Los módulos básicos identificados para el motor de BDOviedo3, que proporcionarán los servicios correspondientes a un SGBDOO son los siguientes:

- **Gestor de extensiones**
- **Gestor de esquemas**
- **Gestor de integridades**
- **Gestor de consultas**
- **Gestor de índices**

Conviene destacar que estos módulos o gestores representan en realidad un conjunto de objetos que están relacionados y que van a proporcionar unos servicios al resto del sistema. Se les agrupa en unidades conceptuales para comprender más fácilmente su papel en esta arquitectura.

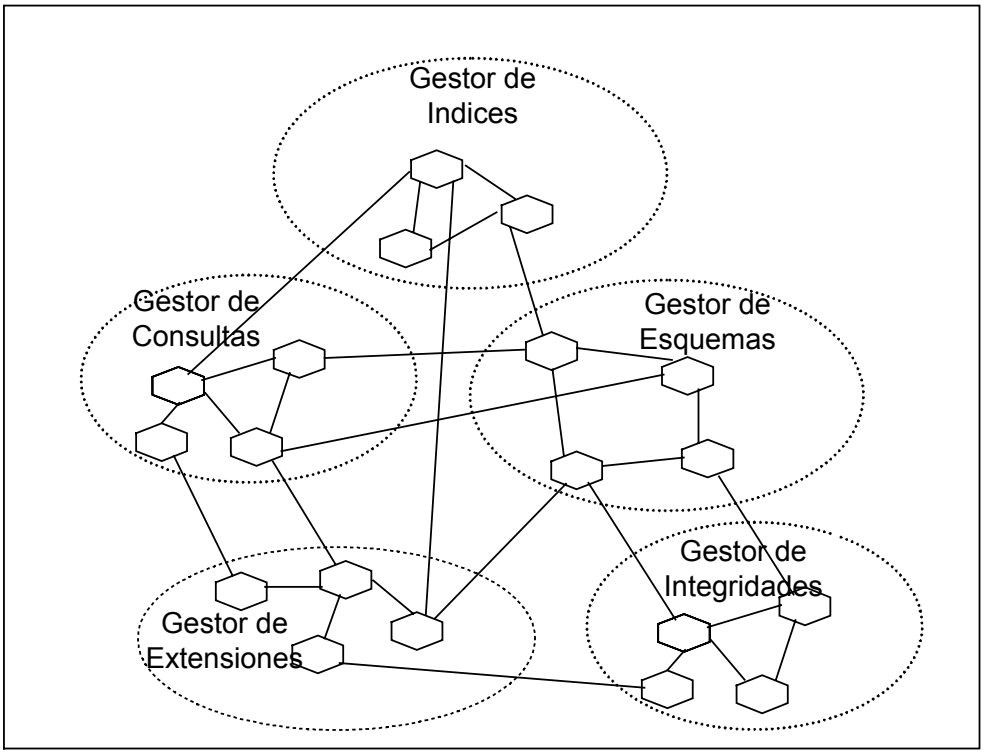


Figura 10.5. Parte de los módulos constituyentes del motor de BDOviedo3

En la Figura 10.5 se representan algunos de los módulos que constituyen el motor. No obstante, existen otros módulos que no han sido incluidos inicialmente, porque no van a ser tratados explícitamente en este trabajo, pero cuya misión es de igual forma extremadamente importante dentro del sistema. Estos gestores son:

- **Gestor de transacciones.** Este módulo será el encargado de gestionar la concurrencia del SGBDOO basándose en el servicio de transacciones proporcionado por el sistema integral.
- **Gestor de autorización.** Será el encargado de gestionar las autorizaciones de los usuarios para acceder a los datos. Este gestor se basará en el subsistema de protección proporcionado por el sistema integral, y deberá tener en cuenta las consideraciones especificadas en el apartado 10.2.7.1
- **Gestor de distribución.** Este gestor, es extremadamente complejo, y sería el encargado de garantizar la distribución en el SGBDOO. Estaría basado en los servicios de distribución proporcionados por el sistema operativo y que fueron comentados brevemente en el apartado 10.2.8

10.4 Gestor de extensiones

En un SGBD relacional son muy corrientes las peticiones de información en las que se necesita acceder a todos los registros de una tabla. De la misma manera, en un SGBDOO es muy corriente la necesidad de acceder a todas las instancias de una determinada clase, en definitiva es necesario acceder a la **extensión de la clase**.

BDOviedo3 ha de facilitar y gestionar, por tanto, el acceso a las extensiones de las clases. Para ello, hay que tener en cuenta que en el sistema integral subyacente todas las instancias de las clases registradas en el sistema se encuentran almacenadas en el área de instancias, y cada una de ellas tiene una referencia a la clase a la que pertenece (que

está en el área de clases), con lo que en cada momento cada instancia sabe a que clase pertenece.

Sin embargo, cuando se necesite acceder a todas las instancias de una determinada clase, puede ser extremadamente tedioso recorrer toda el área de instancias interrogando a cada una de las instancias para saber si realmente pertenece a la clase objetivo o no.

Es necesario entonces, que el motor proporcione un mecanismo que permita controlar las extensiones de las clases. De esta forma, las instancias seguirán almacenadas en el área de instancias, sin ningún tipo de ordenación, pero el gestor de extensiones permitirá almacenar para cada clase, los identificadores de las instancias que constituyen su extensión.

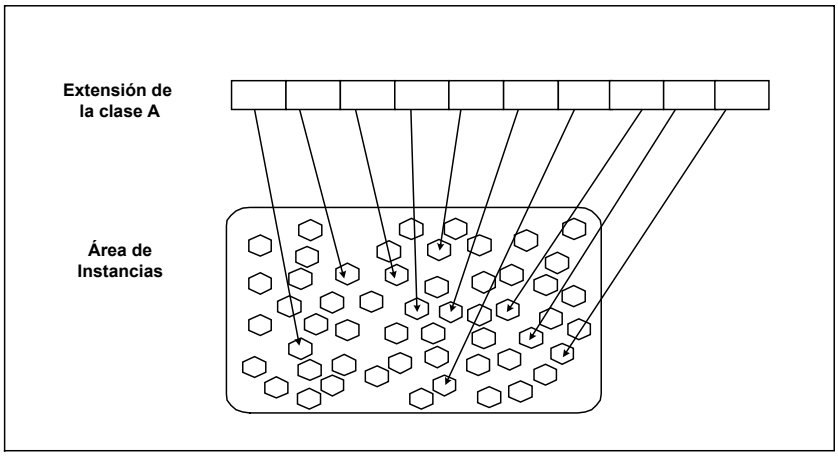


Figura 10.6. Posible representación de la extensión de una clase

Como se puede observar, se está haciendo referencia a la extensión de la clase, sin tener en cuenta que una clase, puede ser a su vez la raíz de una jerarquía de clases, y que las instancias de las subclases son también instancias de la superclase. El modelo que se propone está basado en la clase, pero siempre habrá posibilidad de acceder a las instancias de las subclases. Esto es posible, ya que el catálogo o gestor de esquemas, que se verá a continuación, siempre para cada clase almacena sus clases padres e hijas. Esta información permitirá, situados en una determinada clase, acceder a las extensiones de todas sus subclases.

10.4.1 Información necesaria

La información que necesita el gestor de extensiones es muy simple. Debe disponer únicamente del conjunto de identificadores de los objetos almacenados en el área de instancias, y que corresponden a la clase que se está tratando (Figura 10.6). Esta información se almacenará para cada una de las clases especificadas en los esquemas.

La estructura empleada para almacenar la extensión de la clase puede ser desde una simple matriz o tabla hash, hasta una estructura más compleja como un árbol B^+.

10.4.2 Funciones a proporcionar

El gestor de extensiones ha de proporcionar una funcionalidad muy elemental, cuyas operaciones básicas e imprescindibles son las siguientes:

- Inserción del identificador de un objeto, instancia de una clase, en la estructura que representa la extensión de la misma.

- Eliminación del identificador de un objeto de la estructura que represente la extensión de su clase.
- Comprobación si un determinado objeto pertenece a la extensión de la clase.
- Recuperación de todos los objetos de la extensión. Para ello recorre la estructura que representa la extensión de la clase y devuelve todos los objetos almacenados en la misma.
- Indicación del número de instancias de la clase.

El gestor podrá además estar dotado de otras operaciones o métodos que permitan la interrogación más especializada de la extensión de la clase.

Con el fin de proporcionar esta gestión de extensiones existen otras posibilidades que son igualmente factibles. Una de ellas implica la modificación de la máquina abstracta, de forma que automáticamente cuando una instancia sea almacenada en el área de instancias, a la vez se vayan gestionando las extensiones de las clases. La máquina debería entonces proporcionar métodos para interrogar dichas extensiones. Esta opción ha sido desestimada porque supone una modificación interna del funcionamiento de la máquina.

Otra posibilidad cara a gestionar la extensión de la clase, es la consideración de la reflectividad de la máquina, que aunque ahora no está siendo tenida en cuenta, proporcionaría una buena solución a este problema. Dicha posibilidad implicaría que, por ejemplo, cada vez que un nuevo objeto es creado, mediante reflectividad se activa un objeto cuya misión es precisamente almacenar dicho identificador en la extensión de la clase.

10.5 Gestor de esquemas

El gestor de esquemas, o también denominado **gestor de catálogo** [DAO+95], tiene como misión registrar el esquema de una base de datos. Cuando se crea una base de datos es imprescindible que persista su esquema, por tanto, en el caso de una base de datos orientada a objetos es imprescindible que persistan las declaraciones de las clases, de los índices definidos, de las integridades referenciales, etc.

La información a almacenar y a tratar por el gestor de esquemas es obtenida, principalmente a partir del lenguaje de definición de objetos empleado.

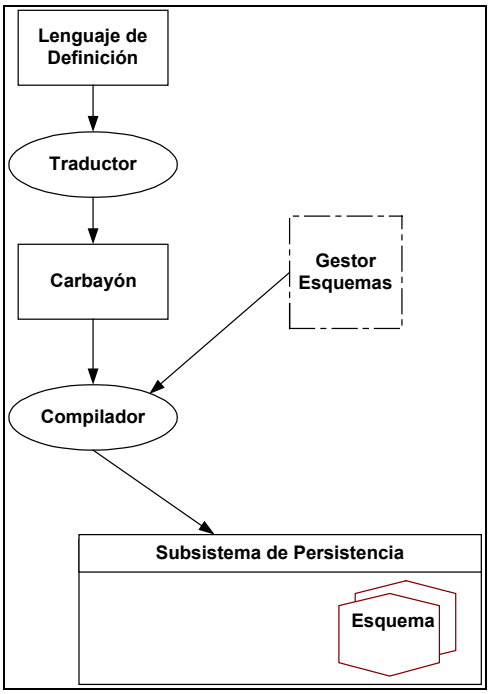


Figura 10.7. Generación del esquema de la base de datos

10.5.1 Información del esquema

El gestor de esquemas ha de posibilitar el acceso a toda la información de la base de datos que es necesitada por el resto de elementos del SGBD para su correcto funcionamiento. Por ejemplo, los índices definidos para cada clase y que necesita conocer el optimizador de consultas, o los atributos de cada clase y sus tipos, para la verificación sintáctica de las consultas.

Por lo tanto el esquema o catálogo de BDOviedo3 debe albergar como mínimo la información que se comenta a continuación. Esto no quiere decir que la estructura que represente el esquema contenga físicamente toda esta información, en muchos casos puede contener únicamente referencias a otras estructuras que son las que realmente la contienen.

Información sobre las clases

Es necesario almacenar toda la información relativa a las clases especificadas mediante el lenguaje de definición:

- Clases de las que hereda
- Clases que heredan de ella
- Nombre y tipo de los atributos
- Nombre de los métodos y argumentos con sus tipos. También debe indicar el valor que devuelve (si es que devuelve alguno).

Si se tiene en cuenta la arquitectura de referencia de la máquina abstracta (véase capítulo 7) se puede observar que en el área de clases se almacena información relativa a las clases del sistema (atributos, métodos, clases de las que hereda, etc.). Es por ello, que en el esquema no sería necesario volver a almacenar toda esta información de manera duplicada, simplemente serviría con almacenar una referencia al objeto del área de clases que contiene dicha información.

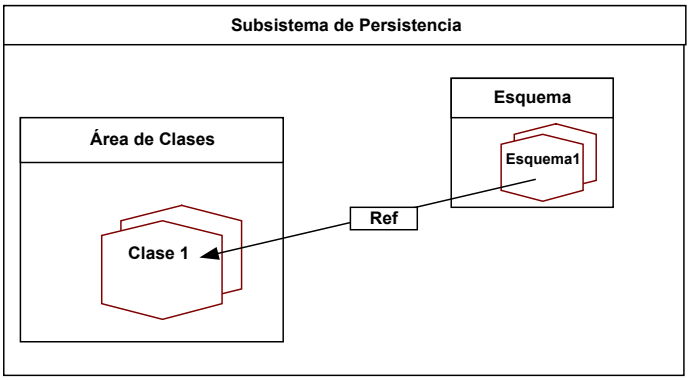


Figura 10.8. Almacenamiento en el esquema de una referencia a una clase almacenada en el área de clases

Información sobre los índices

Es necesario almacenar información sobre los índices especificados para cada clase. Relativo a los índices, en BDOviedo3 se almacenará bastante información, al igual que ocurre con otros sistemas como Oracle8i [Ora99b] o POSTGRES [SR86] en los que el mecanismo de indexación es también potente y extensible.

BDOviedo3 necesita almacenar los atributos que constituyen la clave de indexación para cada uno de los índices especificados para la clase, la técnica empleada para cada índice definido, etc. Parte de esta información podría ser especificada en este catálogo general, sin embargo, se ha optado por crear un catálogo específico para la indexación en el que además de estos datos se almacena más información necesaria para el mecanismo (véase capítulo 12). Entonces aquí, únicamente habrá una referencia a la información almacenada sobre la clase tratada en ese catálogo de índices.

Información sobre la extensión de la clase

El catálogo almacena una referencia al objeto que representa la extensión de la clase y que tiene las características descritas en el apartado 10.4.

Otra Información

Además de la información anterior, existe mucha más información que es necesario almacenar en el catálogo. De entre toda esa información se destaca la siguiente:

- Información que permite gestionar la integridad referencial para las relaciones establecidas con otras clases (detallada en el apartado 10.6).
- Información para la optimización de consultas, relativa a las clases.

10.5.2 Funciones a proporcionar

Las funciones a realizar por este gestor de esquemas están encaminadas básicamente a proporcionar al resto de componentes del sistema la información que le solicitan, en función de la información almacenada en el esquema, y de la operación que éstos deseen realizar.

10.5.2.1 Métodos para la manipulación del esquema

El gestor de esquemas ha de proporcionar en primer lugar toda una serie de métodos que permitan la manipulación de la estructura empleada para albergar dicho esquema. Entre esas operaciones estarían:

- Insertar y eliminar referencias al área de clases, dónde se encuentra almacenada la información de las clases.
- Insertar y eliminar referencias al catálogo de índices dónde se encuentra la información sobre los índices asociados a la clases.
- Insertar y eliminar referencias a las extensiones de las clases.
- Insertar y eliminar referencias a las clases con las que se mantiene integridad referencial, y en definitiva métodos para insertar y eliminar toda la información que necesita el gestor de integridades.

10.5.2.2 Métodos para la interrogación del esquema

El gestor de esquemas debe proporcionar una colección de métodos que permitan interrogar el contenido del esquema con diversos fines. El número de operaciones a proporcionar aquí sería bastante elevado, de forma que únicamente se pondrán algunas de las más significativas:

- Comprobar la existencia de una clase a partir de su nombre. De igual modo comprobar la existencia de atributos y métodos de una clase, sus tipos, etc.
- Indicar que índices hay definidos para una clase, de que tipo son, cuales son los atributos indexados, etc.
- Indicar si la clase mantiene integridad referencial con alguna otra. Obtener las clases con las que mantiene integridad referencial, los caminos de recorrido, etc.
- Indicar si la clase tiene instancias, etc.

10.5.3 Evolución de esquemas

Al hablar de esquemas de bases de datos, es necesario comentar la posible evolución de los mismos. La evolución de esquemas en los SGBDOO es un complejo problema a abordar, tal y como se pudo observar en el capítulo 3. De hecho, la mayoría de los gestores revisados que permiten la evolución de esquemas lo hacen con bastantes restricciones.

La solución tradicional y a la que se puede acudir siempre es la ‘manual’. Es decir, cuando se desea hacer cambios en un esquema, que sea el propio usuario el que confeccione el programa que realice la transformación del antiguo esquema hacia el nuevo. Además, esta opción ha de estar siempre presente ya que siempre hay cambios en el esquema que no pueden ser considerados ni previstos por el SGBDOO. De ahí, que la mayoría de los SGBDOO no se molesten en exceso por proporcionar un mecanismo de evolución de esquemas completo (tal y como se puede observar en el capítulo 5).

BDOviedo3 ha de estar dotado por tanto, de algún mecanismo para la evolución de esquemas, pero dicho mecanismo no es considerado inicialmente en este trabajo.

10.6 Gestor de integridades

Este gestor, estará formado por un conjunto de objetos que se encargarán de mantener la integridad de la información almacenada en la base de datos. Esta integridad va dirigida en dos direcciones:

- **Esquema de la base de datos.** Es necesario garantizar que cualquier cambio realizado en el esquema sólo sea permitido en caso de que el estado de la base de datos siga siendo consistente. Así, por ejemplo, el sistema no debería permitir eliminar una clase que tiene instancias.
- **Relaciones y referencias.** Es necesario garantizar la integridad de las asociaciones y relaciones establecidas entre las clases. Inicialmente únicamente se consideraran las relaciones binarias.

10.6.1 Esquema de la base de datos

El esquema de una base de datos está sujeto a variación, como ya se ha visto en el apartado 10.5.3, pero esta variación ha de estar controlada con el fin de evitar inconsistencias entre la información del esquema y los datos almacenados en la base de datos.

Inicialmente el gestor de integridades de BDOviedo3 ha de controlar, entre otras, que no se realicen las siguientes operaciones:

- Eliminación de clases que tienen instancias
- Eliminación de clases que sean superclases de otras

La eliminación de una clase implicaría su eliminación física del área de clases de la máquina, más la eliminación de toda la información asociada a la clase que aparezca reflejada en el catálogo (índices, extensiones, etc.).

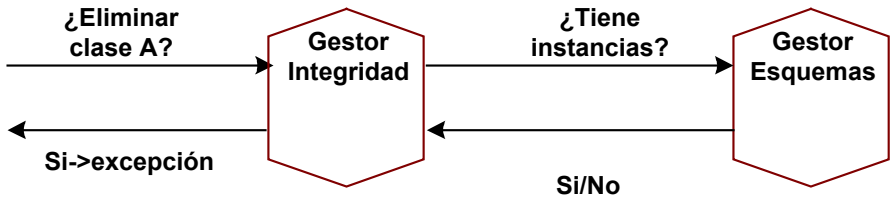


Figura 10.9. Comunicación entre los gestores para saber si una clase tiene instancias

Los servicios de este gestor serán invocados siempre previamente a la llamada encargada de realizar la eliminación física de la clase, de tal forma que si alguna de las condiciones anteriores se cumple, se generará una excepción que impedirá su ejecución.

10.6.2 Validación de las referencias

Es necesario garantizar que cuando se elimina un objeto todos los objetos que lo referenciaban seguirán teniendo un valor válido. En caso de que en el sistema se emplee un borrado explícito es necesario que el sistema sea también el encargado de validar dichas referencias.

Como se comentó en el capítulo 8, el modelo de persistencia del sistema integral ha optado por un borrado explícito en vez de recolección de basura (al menos inicialmente). Entonces es responsabilidad del gestor de objetos asegurar que las referencias sean siempre a objetos que existen y que además son del tipo adecuado.

Sin embargo, en este caso es el propio sistema integral el que facilita esta tarea, ya que, los identificadores no se reutilizan con lo que no es necesario localizar todos los objetos que referencian al objeto que se va a eliminar. Estos objetos pueden ser

detectados con posterioridad. El sistema integral cuando se llama a un objeto que ya no existe, automáticamente genera una excepción.

10.6.3 Integridad de las relaciones

El gestor de integridades se debe encargar también de mantener la sincronización en las relaciones entre clases en las que se ha establecido integridad referencial. Esto quiere decir que si se elimina un objeto de una clase, debe actualizarse automáticamente la clase con la que está relacionada. Así, en el ejemplo de la Figura 10.10, si se añade un nuevo *artículo* perteneciente a una determinada *revista*, el gestor de integridad referencial garantizará que en la clase *revista* se hace la modificación pertinente para que refleje la nueva publicación. Esta será precisamente la tarea del gestor.

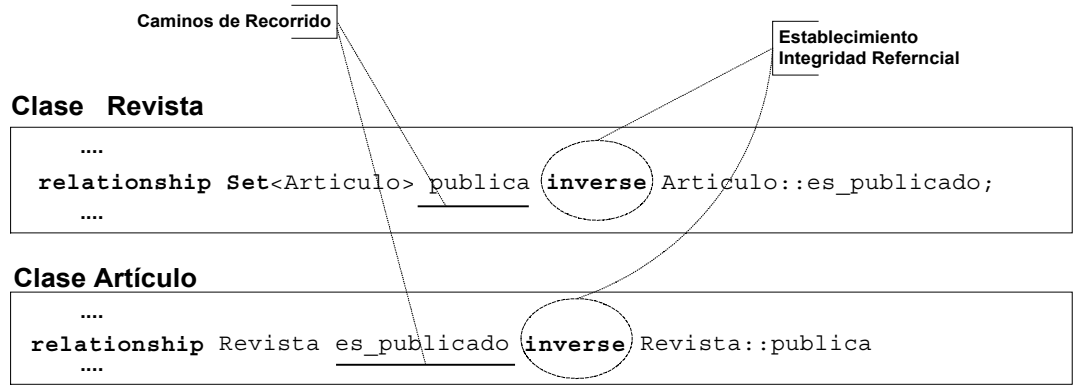


Figura 10.10. Ejemplo de especificación de integridad referencial en un esquema

Para gestionar la integridad referencial en BDOviedo3 es necesaria la intervención de dos elementos del sistema:

- **Traductor del lenguaje empleado**, que a partir de una declaración de relaciones con integridad referencial, en un lenguaje de definición, estilo a la mostrada en la Figura 10.10, generará el código correspondiente a las siguientes operaciones en el lenguaje de la máquina abstracta del sistema integral:
  - Representación de los caminos de recorrido como atributos de la clase, y atendiendo a la cardinalidad de la relación (referencias para cardinalidad uno, y colecciones de referencias para cardinalidad muchos).
  - Incorporación de unos métodos para cada clase, que serán los que permitan tratar con los atributos anteriormente declarados (y que representan los caminos de recorrido), y se encarguen de la sincronización entre las relaciones.
  - Incorporación en el catálogo de la información acerca de las relaciones entre las que hay que mantener la integridad. Sería recomendable almacenar para cada una de las integridades referenciales definidas sobre una clase: el nombre del atributo que representa el camino de recorrido en la clase que nos ocupa, la clase referenciada con la que se tiene que mantener la sincronización, y el nombre del atributo que representa el camino de recorrido en dicha clase. Por ejemplo, para la clase *revista*, la información a almacenar sería: *publica*, *artículo* y *es_publicado* respectivamente. Esta información es similar a la que propone almacenar el estándar ODMG 3.0 en su fichero de propiedades para el *binding* Java.[CBB+99].

- **Motor,** que por medio del gestor de integridades será el encargado de velar por el mantenimiento de la integridad referencial. Dicho gestor será el responsable de invocar los métodos definidos, a tal efecto, sobre cada una de las clases, cada vez que se realice una operación (inserción, eliminación, modificación) sobre los objetos de la misma.

10.7 Gestor de consultas

En un SGBDOO es fundamental, tal y como se vio en el capítulo 3, contar con un acceso declarativo a los datos, en forma de lenguaje de consulta. La presencia de un lenguaje de este tipo, requiere la incorporación al sistema de un procesador de consultas que permita obtener un rendimiento óptimo. De esto se deduce que, el procesamiento de consultas está estrechamente relacionado con el lenguaje de consulta seleccionado.

El objetivo a perseguir es que el módulo que se encargue del procesamiento de consultas ha de ser flexible de forma que si se decide incorporar otro álgebra de objetos o bien otro lenguaje de consulta, sea sencilla la adaptación de BDOviedo3.

10.7.1.1 Funcionalidad básica

A pesar de las diferencias existentes entre el modelo relacional y el modelo orientado a objetos, se pueden adoptar las metodologías de procesamiento de consultas de bases de datos relacionales para las bases de datos orientadas a objetos. Es por esto que la funcionalidad básica que debe proporcionar este procesador de consultas es la ofrecida por la mayoría de los procesadores tradicionales, y se puede agrupar en torno a tres elementos:

- **Analizador y traductor.** Este módulo es el encargado de verificar la sintaxis de las consultas y construir un árbol para el análisis de la consulta que se transformará en una expresión del álgebra de objetos. Cada nodo del árbol representará un operador del álgebra de objetos a ser aplicado a sus entradas. Cada consulta puede ser representada por uno o más árboles de operadores que son equivalentes. Este componente tiene que interactuar con el catálogo o esquema para conocer la información que le permite la verificación sintáctica.
- **Optimizador**. Será el encargado de seleccionar el plan de evaluación o ejecución de la consulta más eficiente. Estos planes de evaluación se obtienen cuando se sustituyen los operadores en el árbol, por los algoritmos empleados para implementarlos (ya que cada operador puede ser implementado por uno o más algoritmos). Para decidirse entre los distintos planes, suelen hacer uso de información estadística que se encuentra también almacenada en el catálogo.
- **Motor de ejecución.** Toma un plan de evaluación, lo ejecuta y devuelve su respuesta a la consulta.

10.8 Gestor de Índices

Con este nombre se recogen un conjunto de objetos que constituyen el mecanismo de indexación con el que se ha dotado al sistema. Uno de los aspectos más importantes en un sistema de gestión de base de datos, es el tiempo empleado en recuperar la información almacenada ante una petición de un usuario. Con el fin de proporcionar una respuesta eficiente, es necesario un buen optimizador de consultas, así como la

existencia de índices que puedan ser empleados por éste con el fin de acelerar dicha recuperación.

De todas las unidades funcionales presentadas, en esta tesis se profundizará en el mecanismo de indexación, por la importancia que para un SGBD tiene el tiempo empleado en acceder a la información ya almacenada. Es objetivo de esta tesis doctoral dotar al SGBDOO de un mecanismo de indexación al que se puedan incorporar de forma fácil y eficiente nuevas técnicas de indexación, que le permitan adaptarse a los diferentes tipos de datos y patrones de aplicación con los que debe de tratar. En los siguientes capítulos (11 y 12) se profundiza en los aspectos relacionados con el mecanismo de indexación.

10.9 Resumen de las características ofrecidas por esta arquitectura

La arquitectura aquí presentada proporciona una serie de características que son enumeradas a continuación a modo de resumen:

- **Uniformidad en la orientación a objetos.** El SGBDOO es construido en base a objetos. El sistema es diseñado como un marco orientado a objetos, de forma que todas las funcionalidades del gestor son proporcionadas por objetos.
- **Extensibilidad.** La organización del código de la base de datos como un conjunto de clases, posibilita que el usuario puede emplear dichas clases o extenderlas para nuevas condiciones de operación, reutilizándose así el propio código del SGBD.
- **Flexibilidad.** Garantizada inicialmente por la orientación a objetos. El sistema ha sido estructurado en diferentes mecanismos, lo más independientes posibles, de forma que la no existencia de uno de ellos no afecte al resto. Por ejemplo, la no existencia del mecanismo de indexación no influiría en el mecanismo de consultas (aunque obviamente si en su rendimiento).
- **Portabilidad.** Derivada de la construcción del sistema sobre una máquina abstracta que facilita la migración a cualquier plataforma.
- **Portabilidad de las aplicaciones.** Facilitada por la adopción del estándar ODMG en su *binding* para Java.
- **Interoperabilidad.** La adopción de un modelo de objetos con la mayoría de los conceptos ya establecidos de la orientación a objetos facilita la interoperabilidad entre sistemas, y para múltiples lenguajes.
- **Integración transparente con el sistema integral**. El SGBDOO emplea las abstracciones ofrecidas por el sistema operativo, tal como la persistencia o la protección. Así, por ejemplo, el sistema de seguridad del gestor se asentará en el subsistema de protección del sistema integral, sin tener que re-implementarlo, únicamente lo complementará.
- **Soporte para el modelo de objetos.** El SGBDOO soporta un modelo de objetos con las principales características de la orientación a objetos.
- **Funcionalidad de base de datos.** La arquitectura del SGBDOO aquí descrita contempla la funcionalidad básica de un SGBD, aunque no se hayan detallado todos los mecanismos que permitan su consecución.

Capítulo 11 Panorama de las técnicas de indexación en bases de datos orientadas a objetos

Un índice, en terminología de base de datos, *es una estructura de almacenamiento físico empleada para acelerar la velocidad de acceso a los datos*. Los índices juegan, por tanto, un papel fundamental en las bases de datos orientadas a objetos, de la misma manera que lo hacen en las relacionales. Son un soporte básico e indispensable [BC99] para acelerar el procesamiento de las consultas, y en definitiva el acceso a los datos, y es precisamente por esto por lo que se existe una gran proliferación de dichas técnicas.

En este capítulo se hace un análisis de aquellas técnicas de indexación consideradas más relevantes, y que abarcan las principales características del modelo de objetos, con el fin de encontrar las más adecuadas para incorporar al mecanismo de indexación con el que se va a dotar al SGBDOO objeto de esta tesis.

11.1 Características de los lenguajes de consulta orientados a objetos que afectan a las técnicas de indexación

La orientación a objetos afecta no sólo a la especificación del esquema de la base de datos si no también al lenguaje de consulta. Si nos centramos en los lenguajes de consulta orientados a objetos, conviene resaltar tres características que vienen directamente impuestas por el modelo de objetos (y que marcarán claramente las diferencias con los lenguajes de consulta relacionales):

- **El mecanismo de la herencia (Jerarquías de Herencia)**. La herencia provoca que una instancia de una clase sea también una instancia de su superclase. Esto implica que, el ámbito de acceso de una consulta sobre una clase en general incluye a todas sus subclases, a menos que se especifique lo contrario.
- **Predicados con atributos complejos anidados (Jerarquías de Agregación)**. Mientras que en el modelo relacional los valores de los atributos se restringen a tipos primitivos simples, en el modelo orientado a objetos el valor del atributo de un objeto puede ser un objeto o conjunto de objetos. Esto provoca que las condiciones de búsqueda en una consulta sobre una clase, se puedan seguir expresando de la forma <atributo operador valor>, al igual que en el modelo relacional, pero con la diferencia básica de que el atributo puede ser un atributo anidado de la clase.
- **Predicados con invocación de métodos.** En el modelo de objetos los métodos definen el comportamiento de los objetos, y al igual que en el predicado de una consulta puede aparecer un atributo, también puede aparecer la invocación de un método.

11.2 Clasificación de las técnicas de indexación en orientación a objetos

Las características anteriormente mencionadas exigen técnicas de indexación que permitan un procesamiento eficiente de las consultas bajo estas condiciones [MC99]. Así, son muchas las técnicas de indexación en orientación a objetos que se han propuesto y que clásicamente [BF95] se pueden clasificar en:

- **Estructurales.** Se basan en los atributos de los objetos. Estas técnicas son muy importantes porque la mayoría de los lenguajes de consulta orientados a objetos permiten consultar mediante predicados basados en atributos de objetos. A su vez se pueden clasificar en:
  - Técnicas que proporcionan soporte para consultas basadas en la **Jerarquía de Herencia.** Ejemplos de los esquemas investigados en esta categoría son *SC* [KKD89*], CH-Tree* [KKD89]*, H-Tree* [CCL92]*, Class Division* [RK95] y hcC-Tree [SS94].
  - Técnicas que proporcionan soporte para *predicados anidados*, es decir, que soportan la **Jerarquía de Agregación.** Ejemplos de los índices de esta categoría son, entre otros, *Nested*, *Path*, *Multiindex* [BK89] y *Path Dictionary Index* [LL98].
  - Técnicas que soportan tanto la **Jerarquía de Agregación** como la **Jerarquía de Herencia**. Nested Inherited e Inherited MultiIndex [BF95] son ejemplos de esta categoría.
- **De Comportamiento.** Proporcionan una ejecución eficiente para consultas que contienen invocación de métodos. La *materialización de métodos* (*method materialization* [KKM94]) es una de dichas técnicas. En este campo no existe una proliferación de técnicas tan grande como en los anteriores.

11.3 Técnicas basadas en la jerarquía de herencia

Estas técnicas se basan en la idea de que una instancia de una subclase es también una instancia de la superclase. Como resultado, el ámbito de acceso de una consulta contra una clase generalmente incluye, no sólo sus instancias sino también las de todas sus subclases. Con el fin de soportar las relaciones de subclase-superclase eficientemente, el índice debe cumplir dos objetivos: *la recuperación eficiente de instancias de una clase simple, y la recuperación eficiente de instancias de clases en una jerarquía de clases.*

11.3.1 Single Class (SC)

La técnica *Single Class* fue una de los primeras en emplearse [KKD89]. Se basa en la idea tradicional del modelo relacional de un índice para cada relación (en este caso clase) y atributo indexado. Esto quiere decir que para soportar la evaluación de una consulta cuyo ámbito es una jerarquía de clases, el sistema debe mantener un índice sobre el atributo para cada clase en la jerarquía de clases.

11.3.1.1 Estructura

La estructura de esta técnica de indexación se basa en los árboles B^+. En esta técnica, la creación de un índice para un atributo de un objeto requiere la construcción de un árbol B^+ para cada clase en la jerarquía indexada.

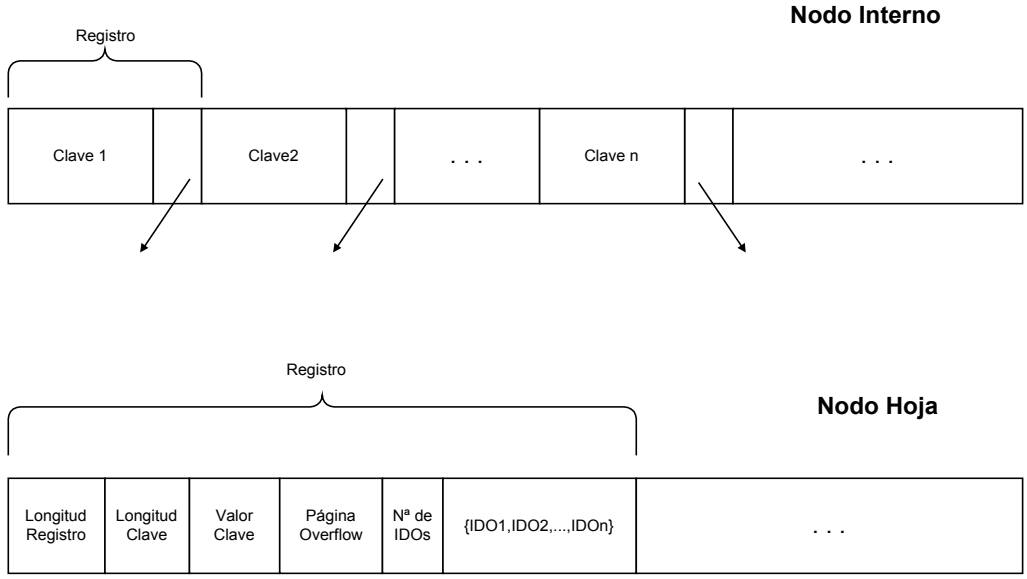


Figura 11.1. Nodo interno y nodo hoja de un árbol B+

11.3.1.2 Operaciones

Las operaciones de búsqueda y actualización siguen los mismos criterios que los árboles B^+ típicos [SKS97, Jan95].

11.3.1.3 Adecuado para consultas en una única clase de la jerarquía

Esta técnica de indexación parece apropiada cuando las búsquedas se realizan en una clase concreta dentro de una jerarquía, ya que esto exige únicamente el recorrido del árbol correspondiente a esa clase. Sin embargo, parece no favorecer las búsquedas que implican todas (o algunas) de las clases de la jerarquía indexada, ya que eso exigiría el recorrido indiscriminado de los árboles asociados a cada una de las clases.

11.3.2 CH-Tree

Kim et al. [KKD89] proponen un esquema alternativo al anterior llamado "*Árbol de Jerarquía de Clases" (Class Hierarchy Tree - CH Tree)* que se basa en mantener un único árbol índice para todas las clases de una jerarquía de clases. El CH-Tree indexa una jerarquía de clases sobre un atributo común, típicamente uno de los atributos de la superclase, sobre una estructura de un árbol B^+.

Esta técnica permitirá evaluar consultas realizadas contra una clase simple en la jerarquía de clases, así como contra cualquier sub-jerarquía de dicha jerarquía.

11.3.2.1 Estructura

La estructura del CH-Tree está basada en los árboles B^+, y de hecho, el nodo interno es similar al de éstos. La diferencia está en los nodos hoja. Éstos contienen (Figura 11.2):

- Para cada clase en la jerarquía, el número de elementos almacenados en la lista de identificadores de objetos (IDOs) que corresponden a los objetos que contienen el *valor-clave* en el atributo indexado, y la propia lista de IDOs.

- Un *directorio clave* que almacena el número de clases que contienen objetos con el *valor-clave* en el atributo indexado, y, para cada clase, el identificador de la misma y el desplazamiento en el registro índice, que indica donde encontrar la lista de IDOs de los objetos.

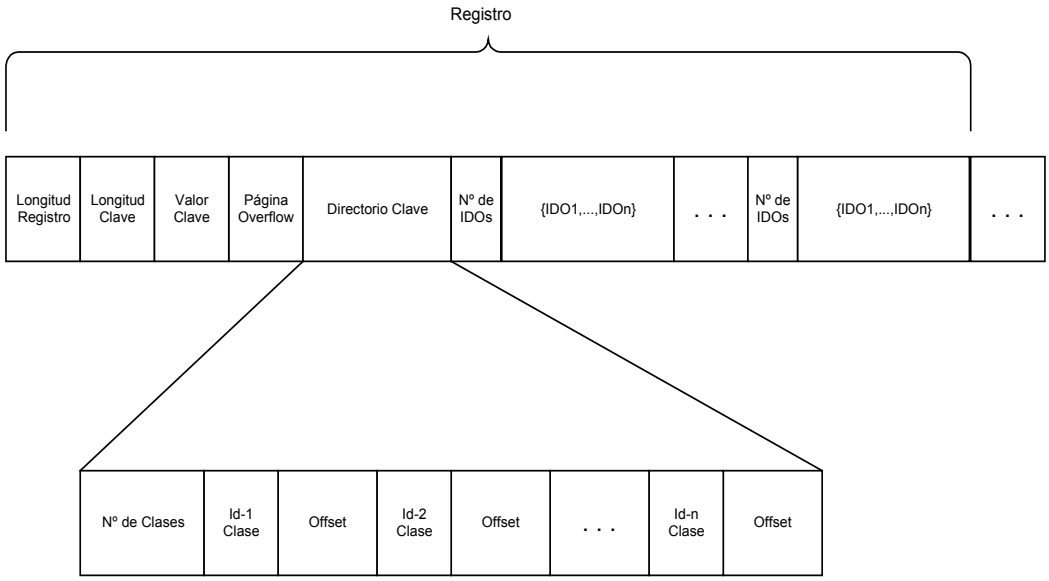


Figura 11.2. Nodo hoja de un CH-Tree

Es decir, el nodo hoja agrupa la lista de IDOs para un *valor-clave* en términos de las clases a las que pertenecen.

11.3.2.2 Operaciones

La búsqueda en un CH-Tree es similar a la de los árboles B^+, de tal manera que, cuando el nodo hoja es localizado, se comparan los valores clave almacenados en el registro índice con la clave buscada. Si coinciden los valores, entonces, si en la consulta se referencia la jerarquía de clases completa, se devuelven todos los IDOs que aparecen en el registro índice. Por el contrario, si en la consulta se referencia sólo una clase, se consulta el *directorio clave* del registro índice para localizar los IDOs asociados con la clase.

Esta técnica es empleada, por ejemplo, en el sistema ORION [KGB+90].

11.3.2.3 Adecuado para consultas que implican varias clases de la jerarquía

Lo que intuitivamente parece lógico los estudios realizados lo confirman [KKD89]. Si el predicado de una consulta implica únicamente una clase en la jerarquía de clases, el SC se comporta mejor que el CH-Tree. Por el contrario, si se implican todas las clases de la jerarquía (o al menos dos) en la consulta, el CH-Tree se comporta mejor que un conjunto de índices SC. Con relación al tamaño de los índices generados los experimentos realizados no muestran un claro ganador o perdedor entre el CH-Tree y el conjunto de índices SC.

11.3.3 H-Tree

Representan una alternativa al CH-Tree propuesta en [CCL92]. Se basan también en los árboles B^+, y pueden verse como los sucesores de los índices *SC*. Esta técnica, se

basa en mantener un H-Tree (*Hierarchy Tree*) para cada clase de una jerarquía de clases, y estos H-Trees se anidan de acuerdo a sus relaciones de superclase-subclase. Cuando se indexa un atributo, el H-Tree de la clase raíz de una jerarquía de clases, se anida con los H-Trees de todas sus subclases inmediatas, y los H-Trees de sus subclases se anidan con los H-Trees de sus respectivas subclases, y así sucesivamente. Indexando de esta manera se forma una jerarquía de árboles de índices.

11.3.3.1 Estructura

La estructura del H-Tree se basa en los árboles B+, pero presenta variaciones tanto en el nodo interno como en el nodo hoja (Figura 11.3):

- El nodo hoja contiene, un contador que índica el número de IDOs que hay en la lista de identificadores de objetos cuyo valor en el atributo indexado es el *valor-clave*, y la propia lista de IDOs.
- El nodo interno aparte de los valores clave discriminantes y los punteros a los nodos hijos, almacena punteros que apuntan a subárboles de H-Trees anidados. También almacena el puntero al subárbol anidado, así como los valores máximos y mínimos de dicho subárbol con el fin de evitar recorridos innecesarios del mismo.

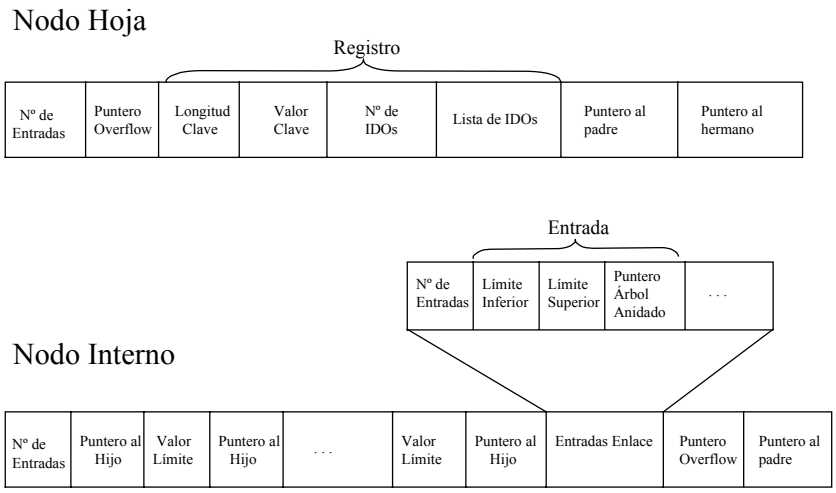


Figura 11.3. Nodo interno y nodo hoja de un H-Tree

11.3.3.2 Operaciones

El anidamiento de los H-Trees evita la búsqueda en cada H-Tree cuando se consultan un número de clases de la jerarquía [CHL+96]. Cuando se busca en los árboles de una clase y sus subclases, se hace una búsqueda completa en el H-Tree de la superclase, y una búsqueda parcial en los H-Trees de las subclases. Esta es la principal ventaja con relación a los índices SC. Además, un H-Tree puede ser accedido independientemente del H-Tree de su superclase, ya que la clase consultada no tiene porque ser la clase raíz de la jerarquía de clases, y por tanto, buscar instancias dentro de una sub-jerarquía de clases puede comenzar en cualquier clase siempre que esté

indexada por el mismo atributo. El mayor inconveniente de esta estructura es la dificultad para dinamizarla.

11.3.3.3 Adecuado para consultas de recuperación en un número pequeño de clases de la jerarquía

Estudios realizados demuestran también que el H-Tree se comporta mejor que el CH-Tree para consultas de recuperación, especialmente cuando en la consulta se referencian un número pequeño de clases de la jerarquía, ya que, el CH-Tree emplea la misma estrategia para la recuperación de datos de una clase simple que de una jerarquía de clases. Para las consultas de actualización no hay estudios suficientes para concluir cual de los dos se comporta mejor.

11.3.4 hcC-Tree

Los *hcC-trees* (*hierarchy class Chain tree*) representan una alternativa a los esquemas vistos anteriormente, y al igual que ellos permiten la recuperación eficiente tanto de instancias de una clase simple como de una jerarquía de clases, pero para ello almacena información en dos clases de cadenas: cadenas de jerarquía (*hierarchy chain*) y cadenas de clase (*class chain*), y al igual que el CH-Tree únicamente emplea un árbol para indexar una jerarquía de clases por un atributo. Este esquema [SS94] pretende la creación de una estructura que favorezca los siguientes tipos de consulta:

- **CHP** (*Class Hierarchy Point*): consultas puntuales[1] sobre todas las clases de una jerarquía de clases que comienza con la clase consultada.
- **SCP** (*Single Class Point*): consultas puntuales sobre una clase simple.
- **CHR** (*Class Hierarchy Range*): consultas de rango[2] sobre todas las clases de una jerarquía de clases que comienza con la clase consultada.
- **SCR** (*Single Class Range*): consultas de rango sobre una clase simple.

Atendiendo a esta clasificación se observa que tanto las consultas CHP como las CHR se favorecerían si los identificadores de los objetos de todas las clases de la jerarquía con un valor dado en el atributo indexado estuvieran agrupados. Y por otro lado, las consultas SCP y SCR se favorecerían si los identificadores de los objetos de una clase simple para un valor concreto del atributo indexado estuvieran agrupados juntos. A priori, ambos requerimientos como se aprecia entran en conflicto, y para evitarlo se diseña esta estructura que da solución a ambos.

11.3.4.1 Estructura

La estructura del hcC-Tree se basa en los árboles B^+, pero a diferencia de éstos distingue tres tipos de nodos:

- **Nodos Internos**. Constituyen los niveles superiores del árbol. La estructura del nodo interno está constituida por m claves y m+1 punteros (similar al nodo interno del árbol B^+), pero además, para cada uno de los m+1 intervalos se almacena un bitmap de n bits. Un bit (en el bitmap) correspondiente a una clase

[1] Consultas puntuales solicitan todas las instancias con un valor concreto para el atributo indexado

[2] Consultas de rango incluyen todas las instancias cuyos valores para el atributo indexado estén dentro de un cierto rango

para un intervalo es 0 si y solo si no existen objetos pertenecientes a esa clase que tengan un valor para el atributo indexado que esté dentro del intervalo.

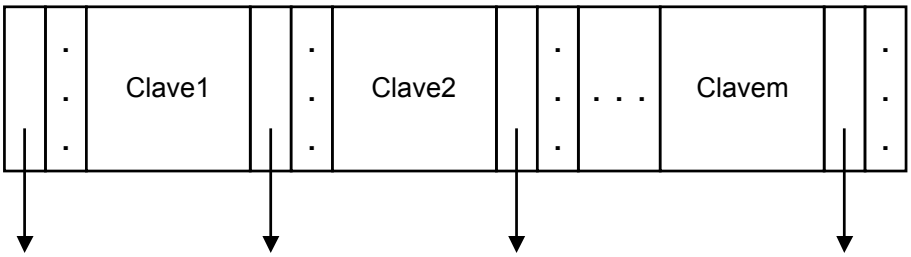


Figura 11.4. Nodo interno del hcC-Tree

- **Nodos Hoja.** Representan los penúltimos niveles del árbol. Cada entrada en el nodo hoja está constituido por un valor clave (k), un bitmap (de n bits, uno por clase) y un conjunto de punteros. Un bit correspondiente a una clase es asignado si y solo si existe al menos un objeto perteneciente a esa clase que tiene el valor clave k. Si suponemos que son asignados m bits en el bitmap para el valor k, entonces el conjunto de punteros consta de m+1 punteros, m de ellos apuntan a nodos-IDO de cadenas de clase y uno a un nodo-IDO de cadenas de jerarquía.

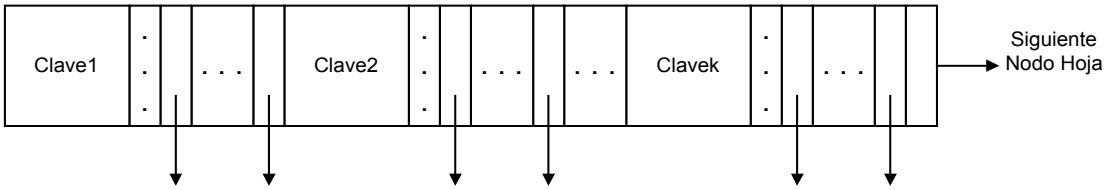


Figura 11.5. Nodo hoja de un hcC-Tree

- **Nodos IDO.** Representan el último nivel del árbol. Para cada clase en la jerarquía hay una cadena de nodos-IDO (cadena de clase), y hay sólo una cadena de nodos-IDO que corresponden a todas las clases (cadena de jerarquía). Los nodos-IDO de una cadena de clase almacenan el IDO de los objetos de esa clase solamente (cada entrada es de la forma <clave, lista IDOs>). Los nodos-IDO de una cadena de jerarquía contienen los IDO de los objetos pertenecientes a todas las clases (una entrada es de la forma <clave, conjunto de lista de IDOs>). Los nodos-IDO de cada una de las n+1 cadenas son enlazados para facilitar el recorrido de los nodos de izquierda a derecha.

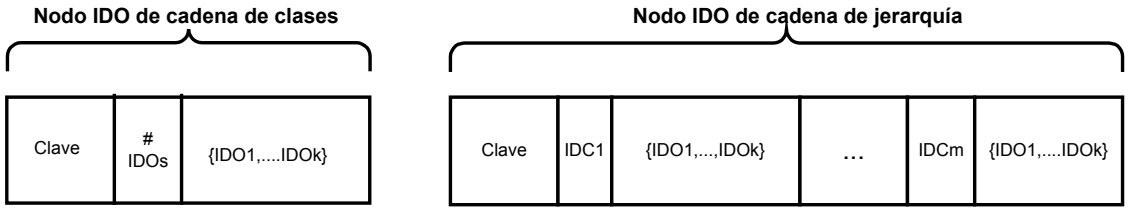


Figura 11.6. Entrada de nodos-IDO del hcC-Tree3

[3] IDC, identificador de la clase del objeto

11.3.4.2 Operaciones

La operación de búsqueda en un hcC-Tree es similar a la de los B^+, pero teniendo en cuenta que en función del tipo de consulta (SCP, SCR, CHR o CHP) se examinará o bien la cadena de jerarquía o bien la cadena de clase, y además se utilizarán los bit asignados en los bitmap para evitar recorridos innecesarios.

El algoritmo de inserción se divide en tres partes: la primera es similar al algoritmo de inserción de los B^+, la segunda consiste en añadir el IDO a la cadena de clase, y la tercera consiste en añadir el IDO a la cadena de jerarquía. Al analizar estas operaciones se observa que pueden ser necesarios como máximo tres recorridos extra para la inserción, pero en la mayoría de los casos no habrá sobrecarga extra. Por lo tanto, en el peor de los casos el costo del algoritmo de inserción está limitado por la altura del árbol.

11.3.4.3 Es menos dependiente de la distribución de los datos que CH-Tree y H-Tree

Los hcC-Trees son más fáciles de implementar, y de hecho más simples que los H-Trees ya que no hay anidamiento de árboles. Además, los experimentos realizados atendiendo a la distribución de los valores clave en la jerarquía de clases para controlar el número de páginas accedidas [SS94] muestran que H-Trees y CH-Trees son más dependientes de la distribución actual de los datos. Los H-Trees no tienen buen rendimiento para CHR y CHP, y los CH-Tree no tienen buen rendimiento para consultas SCR.

11.3.4.4 Se comporta mejor que el H-Tree cuando el número de clases en la jerarquía aumenta

Cuando el número de clases en una jerarquía crece el rendimiento del H-Tree se degrada para consultas CHP ya que hay más saltos desde los H-Trees de las superclases a los H-Trees de las subclases. A su vez, el CH-Tree se comporta ligeramente mejor que el hcC-Tree.

11.4 Técnicas basadas en la jerarquía de agregación

Estas técnicas se basan en la idea de que las consultas en orientación a objetos soportan predicados anidados. Para soportar dichos predicados, los lenguajes de consulta generalmente proporcionan alguna forma de expresiones de camino.

Previo al análisis de estas técnicas de indexación es necesario concretar el significado de algunos conceptos.

Camino

Un camino *es una rama en una jerarquía de agregación que comienza con una clase C y termina con un atributo anidado de C.* Formalmente, un camino P para una jerarquía de clases H se define como C(1).A(1).A(2)...A(n), donde

- C(1) es el nombre de una clase en H,
- A(1) es un atributo de la clase C(1*)*, y
- A(i) es un atributo de una clase C(i) en H, tal que C(i) es el dominio del atributo A(i-1) de la clase C(i-1), $1< i \leq n$.

Instanciación de un camino

La instanciación de un camino *representa la secuencia de objetos obtenida por la instanciación de las clases que pertenecen al camino.* Formalmente, dado un camino P = C(1).A(1).A(2)...A(n), una *instanciación* de P se define como una secuencia de n+1 objetos, denotados como O(1).O(2)...O(n+1), donde,

- O(1) es una instancia de la clase C(1), y
- O(i) es el valor de un atributo A(i-1) del objeto O(i-1), $1< i \leq n+1$.

Instanciación parcial

Una instanciación parcial *es una instanciación que comienza con un objeto que no es una instancia de la primera clase a lo largo del camino, sino de alguna otra clase del camino*. Formalmente, dado un camino P= C(1).A(1).A(2)...A(n), una *instanciación parcial* de P es definida como una secuencia de objetos O(1).O(2)...O(j), $j< n+1$, donde

- O(1) es una instancia de una clase C(k) en *Class* (P) para k=n-j+2,
- O(i) es el valor del atributo A(i-1) del objeto O(i-1), $1< i \leq j$, y siendo
- *Class*(P) = C(1) $\cup$ {C(i) tal que C(i) es el dominio del atributo A(i-1) de la clase C(i-1), $1 < i \leq n$}.

Instanciación parcial no redundante

Una instanciación de camino *es no redundante si no está contenida en otra instanciación más larga.* Formalmente, dada una instanciación parcial p= O(1).O(2)...O(j) de P, p es *no redundante,* si no existe una instanciación p'=O'(1).O'(2)...O'(k), k>j, tal que O(i) = O'(k-j+i), $1\leq i \leq j$.

11.4.1 Path (PX)

Un *path index (índice de camino)* es un índice que almacena las instanciaciones de un camino (secuencias de objetos), siendo la clave del índice el objeto al final del camino de instanciación. Formalmente, dado un camino P= C(1).A(1).A(2)...A(n), un índice *Path* (PX) sobre P es definido como un conjunto de pares (O,S), donde

- es el *valor clave*, y
- S={π<j-1>(p(i)) tal que, p(i)=O(1).O(2)...O(j) (1<=j<=n+1) es una instanciación no redundante (parcial o no) de P, y O(j) = O}. π<j-1>(p(i)) denota la proyección de p(i) sobre los primeros j-1 elementos.

De esto se deduce que, un índice PX almacena todos los objetos que finalizan con esa clave.

11.4.1.1 Estructura

La estructura de datos base para este índice puede ser un árbol B^+, al igual que para los Nested y Multiindex, y de hecho, el formato de los nodos internos es idéntico en los tres casos. La variación está en el nodo hoja.

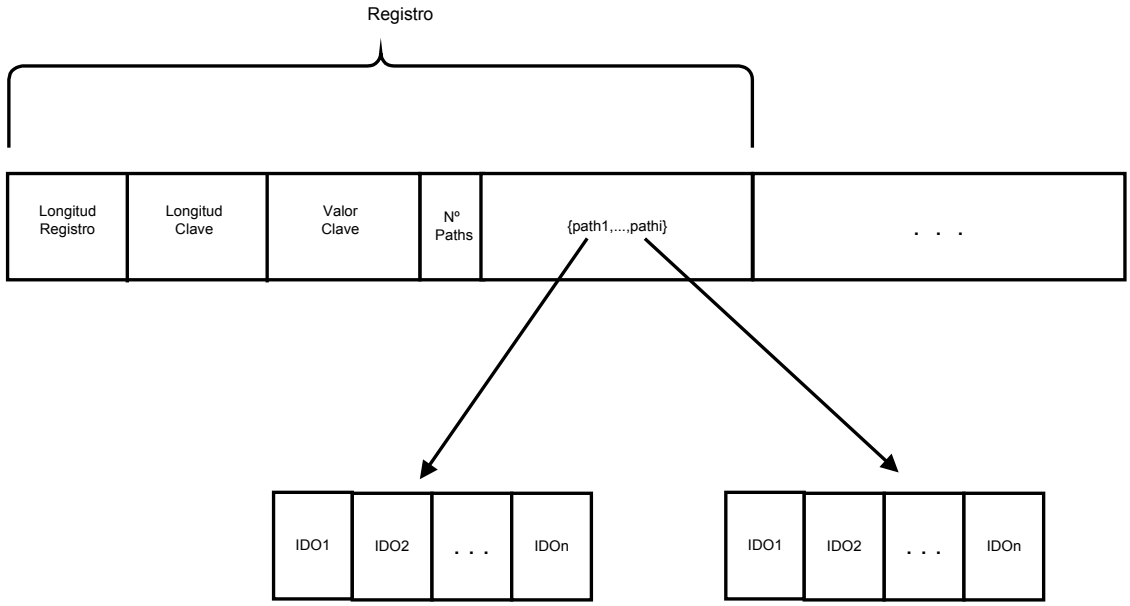


Figura 11.7. Nodo hoja para un índice Path

En el caso del índice PX, el nodo hoja (**Figura 11.7**) contiene entre otra información, el número de elementos en la lista de instanciaciones que tienen como objeto final el *valor clave*, y la propia lista de instanciaciones. Cada instanciación de camino se implementa por un array que tiene la misma longitud que el camino. El primer elemento del array es el IDO de una instancia de la clase C(1), el segundo elemento es el IDO de la instancia de la clase C(2) referida por el atributo A(1) de la instancia del primer IDO del array, y así sucesivamente.

11.4.1.2 Operaciones

En un PX para evaluar un predicado sobre un atributo A(n) de la clase C(i) (1<=i<=n) del camino P, sólo se necesita recorrer un índice. Una vez que se han determinado las instanciaciones de camino asociadas con el valor de la clave, se extraen los IDOs que ocupan la posición i-ésima de cada array.

Las actualizaciones sobre un PX son costosas, ya que se requieren recorridos hacia delante, sin embargo no es necesario un recorrido inverso ya que en los nodos hoja se almacenan las instanciaciones completas del camino. Esta organización se puede emplear aunque los objetos no contengan referencias inversas al no ser necesario el recorrido inverso.

11.4.1.3 Permite resolver predicados anidados sobre todas las clases del camino

Un PX almacena instanciaciones de caminos, y como se puede observar, puede ser empleado para evaluar predicados anidados sobre todas las clases a lo largo del camino.

11.4.2 Nested (NX)

Un *Nested Index* (Índice Anidado) establece una conexión directa entre el objeto al comienzo y el objeto al final de un camino de instanciación, siendo la clave del índice el objeto al final de dicho camino. Formalmente, dado un camino P= C(1).A(1).A(2)...A(n), un índice *Nested* (NX) sobre P se define como un conjunto de pares (O,S), donde

- O es el *valor clave*, y
- S es el conjunto de IDOs de objetos de C(1), tales que O y cada IDO en S, aparecen en la misma instancia del camino.

Un NX a diferencia del índice PX sólo almacena los objetos iniciales de las instanciaciones de los caminos. Como se puede observar cuando n=1, el NX y el PX son idénticos, y son los índices empleados en la mayoría de los SGBD relacionales.

11.4.2.1 Estructura

El nodo interno es similar al de los árboles B^+ típicos. El nodo hoja para el NX simplemente almacena la longitud del registro, la longitud de la clave, el valor clave, el número de elementos en la lista de IDOs de los objetos que contienen el valor clave en el atributo indexado, y la lista de IDOs. Se supone que la lista de IDOs está ordenada, y que cuando el tamaño de un registro excede el tamaño de la página, se mantiene un pequeño directorio al comienzo del registro. Este directorio contiene la dirección de cada página del registro y el valor mayor contenido en la página. Cuando se añade o se elimina un IDO de un registro se puede identificar de manera directa la página en la que hay que hacer la modificación.

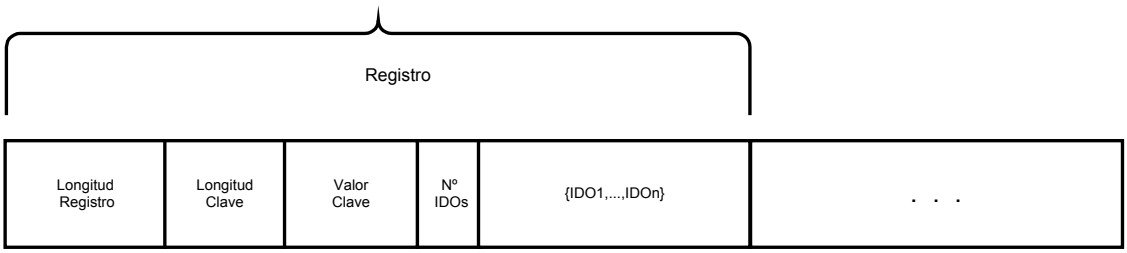


Figura 11.8. Nodo hoja de un índice Nested

11.4.2.2 Operaciones

La actualización con estos índices si es problemática ya que requieren el acceso a varios objetos, con vistas a determinar que entradas de índice se deben actualizar. Es necesario tanto del recorrido hacia delante como del recorrido inverso de los objetos. El recorrido hacia delante se necesita para determinar el valor del atributo indexado (valor del atributo final del camino) del objeto modificado. El recorrido inverso es necesario para determinar todas las instancias al comienzo del camino, y es muy costoso cuando no hay referencias inversas entre objetos.

11.4.2.3 Bajo costo en operaciones de recuperación

Con este índice la evaluación de un predicado para el camino P sobre un atributo anidado A(n) de la clase C(1) requiere el recorrido de un único índice, por tanto, el costo de resolver un predicado anidado es equivalente al costo de resolver el predicado sobre un atributo simple C(1).

11.4.3 Multiindex (MX)

Esta organización se basa en asignar un índice a cada una de las clases que constituyen el camino [MS86]. Formalmente, dado un camino P= C(1).A(1).A(2)...A(n), un *Multiindex* (MX) se define como un conjunto de n *índices*

Nested (NX), NX.1, NX.2,... NX.n, donde NX.i es un índice Nested definido sobre el camino C(i).A(i), 1≤ i ≤n. Esta técnica es empleada en el sistema GemStone.

11.4.3.1 Estructura

La estructura de estos índices se basa en un árbol B^+ al igual que los anteriores. Y tanto su nodo interno como sus nodos hoja tienen una estructura similar a la de los Nested (véase **Figura 11.8**)

11.4.3.2 Operaciones

Con esta organización la operación de recuperación se lleva a cabo recorriendo primero el último índice asignado sobre el camino. El resultado de este recorrido se usa como clave para la búsqueda sobre el índice que precede a este último, y así sucesivamente hasta que se recorra el primer índice.

La mayor ventaja de esta técnica es el bajo costo de actualización, sin embargo, el resolver un predicado anidado requiere recorrer una cantidad de índices igual a la longitud del camino.

11.4.3.3 Bajo costo en operaciones de actualización

Para consultas de actualización, el que mejor se comporta es el MX, seguido del PX, y finalmente el NX. De hecho, la razón de dividir un camino en subcaminos, es principalmente, para reducir los costos de actualización del NX o del PX, pero permitiendo a la vez una recuperación eficiente. En [Ber94] se propone un algoritmo que determina si un camino debe ser dividido en subcaminos, y también, a que subcaminos se les debe asignar un índice, así como la organización (NX, PX, etc.) que éste debería tener.

11.4.3.4 Alto costo en operaciones de recuperación

Si se examina el rendimiento con relación a consultas de recuperación, el NX es el que mejor se comporta, ya que el PX tiene un gran tamaño, pero éste último se comporta bastante mejor que el MX, ya que éste necesita acceder a varios índices.

11.4.3.5 PX es el mejor teniendo en cuenta todas las operaciones

Para concluir, se puede decir que teniendo en cuenta las diferentes clases de operaciones (recuperación, actualización, inserción y borrado) el PX se comporta mejor que las otras dos estructuras.

11.4.3.6 MX requerimientos de espacio intermedios entre NX y PX

Atendiendo a los experimentos realizados y expuestos en [BK89], se puede concluir que en cuanto a requerimientos de espacio, el NX siempre tiene los requerimientos más bajos. El PX requiere siempre más espacio que el MX, excepto cuando hay pocas referencias compartidas entre objetos, o bien no hay ninguna. Esto es debido a que el mismo IDO puede aparecer en muchas instancias de camino, ocasionando que ese IDO sea almacenado muchas veces.

11.4.4 Path Dictionary Index (PDI)

Un *Path Dictionary Index* (Índice Diccionario de Caminos) es una estructura de acceso [LL98] que reduce los costos de procesamiento de las consultas que soportan tanto búsquedas asociativas como recorridos de objetos. Para ello, consta de dos partes (**Figura 11.9**): el *path dictionary*, que soporta un recorrido eficiente de los objetos, y los

índices *identity* y *attribute* que soportan una búsqueda asociativa, y son construidos sobre el *path dictionary*.

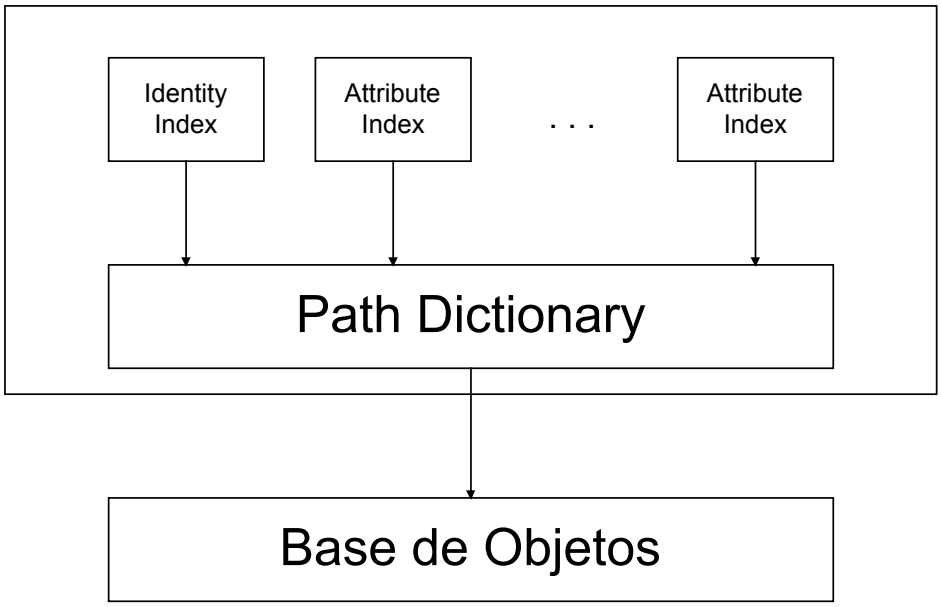


Figura 11.9. Índice diccionario de caminos

Path Dictionary (PD)

El *path dictionary* extrae los atributos complejos de la base de datos para representar las conexiones entre objetos. Evita, por tanto, un acceso innecesario a los objetos en la base de datos almacenando la información del camino entre los objetos en una estructura de acceso separada. Y, ya que los valores de los atributos primitivos no son almacenados, es mucho más rápido recorrer los nodos en el *path dictionary* que los objetos en la base de datos.

Attribute Index (AI)

El *path dictionary* soporta un recorrido rápido entre objetos, pero no ayuda en la evaluación de predicados que implican la búsqueda de objetos que cumplen ciertas condiciones especificadas sobre sus atributos. Para facilitar esta búsqueda el PDI proporciona los índices *attribute*, que asocian los valores de los atributos a los IDOs en el *path dictionary* que corresponden a esos valores. En lugar de traducir valores de atributos a objetos directamente (como *Nested* o *Path*), el índice *attribute* traduce valores de atributos a información del camino almacenado en el *path dictionary*. El *path dictionary* sirve como una estructura compartida para el recorrido de objetos, y como un nivel de indirección desde los valores de los atributos a los objetos físicos.

Esta separación entre el soporte para el recorrido y para las búsquedas asociativas permite construir tantos índices *attribute* sobre el *path dictionary* como sea necesario sin incurrir en un crecimiento extraordinario en la sobrecarga de almacenamiento. Esta separación se traduce también en un bajo costo de mantenimiento.

Identity Index (II)

Como los IDOs son empleados para describir la información del camino entre objetos, a menudo se necesita obtener información en el *path dictionary* sobre el camino asociado con un determinado IDO. Con el fin de soportar eficientemente esta posibilidad se crea un índice *identity* que proporciona una asociación de los IDOs con las localizaciones en el *path dictionary* donde pueden encontrarse dichos IDOs. Este

índice *identity* reduce significativamente el costo de las operaciones de recuperación y actualización, y al igual que ocurre con el índice *attribute*, este índice es organizado como un árbol de búsqueda separado sobre el *path dictionary*.

11.4.4.1 Estructura

Esta técnica de indexación como se ha visto con anterioridad consta de varias estructuras, una para cada tipo de índice empleado. A continuación se describen brevemente las estructuras de cada uno de ellos.

Path Dictionary

Una implementación del *path dictionary* es el esquema *s-expresión*, donde cada s-expresión en el *path dictionary* almacena la información del camino para un subárbol de objetos; es decir, codifica en una expresión todos los caminos que terminan con el mismo objeto en una clase hoja. La s-expresión para un camino C(1).C(2).C(n) se define como

- $S(1) = \varnothing_1$, donde $\varnothing_1$ es el IDO de un objeto en la clase C(1) o nulo.
- $S(i) = \varnothing_i(S(i-1)[,S(i-1)])$ $1<i<=n$, donde $\varnothing_i$ es el IDO de un objeto en la clase C(i) o nulo, y S(i-1) es una s-expresión para el camino C(1)C(2)...C(i-1)

El *path dictionary* para C(1).C(2).C(n) consta de una secuencia de n-niveles de s-expresiones. El objeto que va en cabeza en una s-expresión, que no tiene porque pertenecer necesariamente a C(n) es el objeto terminal de los caminos denotados por la s-expresión. Así, el número de s-expresiones que corresponden a un camino es igual al número de objetos a lo largo del camino que no tienen un objeto anidado sobre el camino.

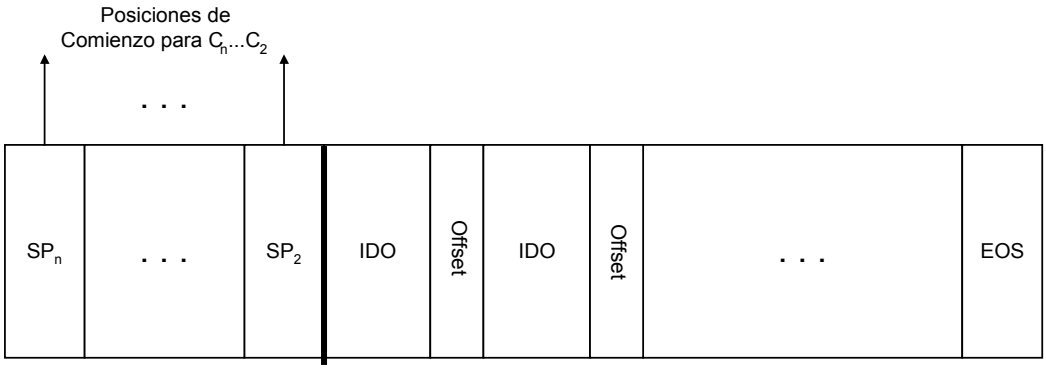


Figura 11.10. Estructura de datos de una s-expresión

Los punteros SP_i apuntan a la primera ocurrencia de $\varnothing_i$ en la expresión. Siguiendo los campos SP_i hay una serie de pares <IDO, Offset>, donde los Offsets asociados con un $\varnothing_{i\ (2<=i<=n)}$ apuntan a la siguiente ocurrencia de $\varnothing_i$ en la s-expresión. Al final de la s-expresión hay un símbolo especial de *end-of-s-expresión*(EOS).

Identity Index

Su estructura está basada también en los árboles B^+, pero tanto el nodo interno como el hoja sufren ligeras variaciones. En este índice los IDOs son empleados como valor clave (**Figura 11.11**). La *s-dirección* en el nodo hoja es la dirección de la s-expresión que corresponde al IDO que está en ese nodo hoja. Los punteros de página en un nodo interno están apuntando al siguiente nivel de nodos internos o a los nodos hoja.

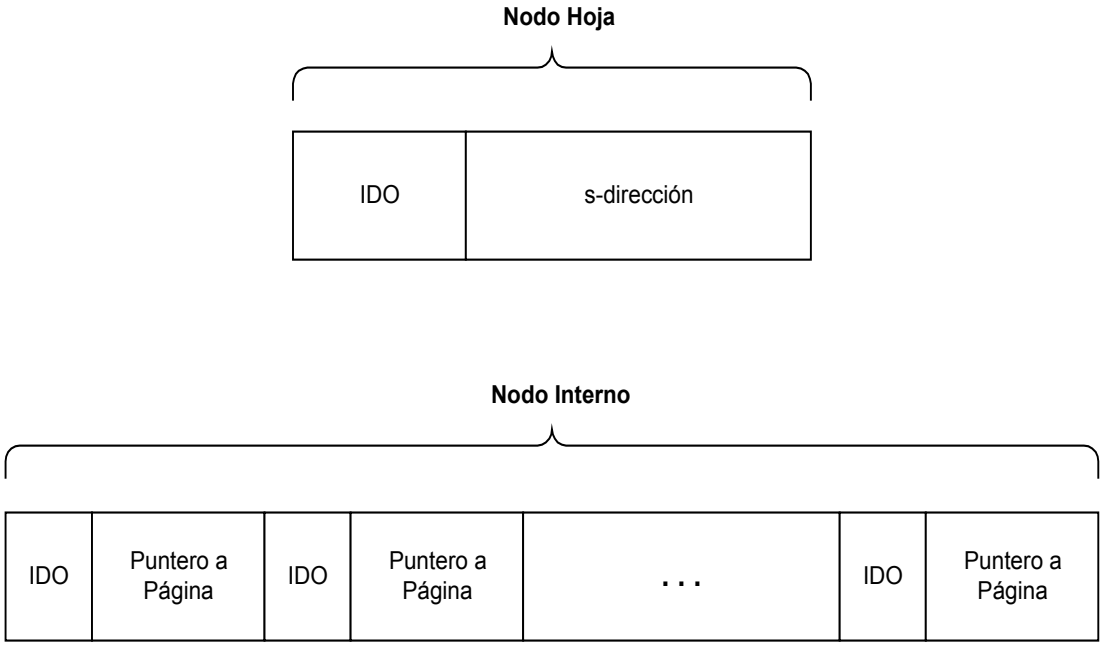


Figura 11.11. Nodo hoja e interno de un índice identity

Attribute Index

La estructura de este índice está basada también en los árboles B^+. Los IDOs y las direcciones de las s-expresiones (s-direcc) (**Figura 11.12**) son empleados para acceder a las s-expresiones de los correspondientes IDOs.

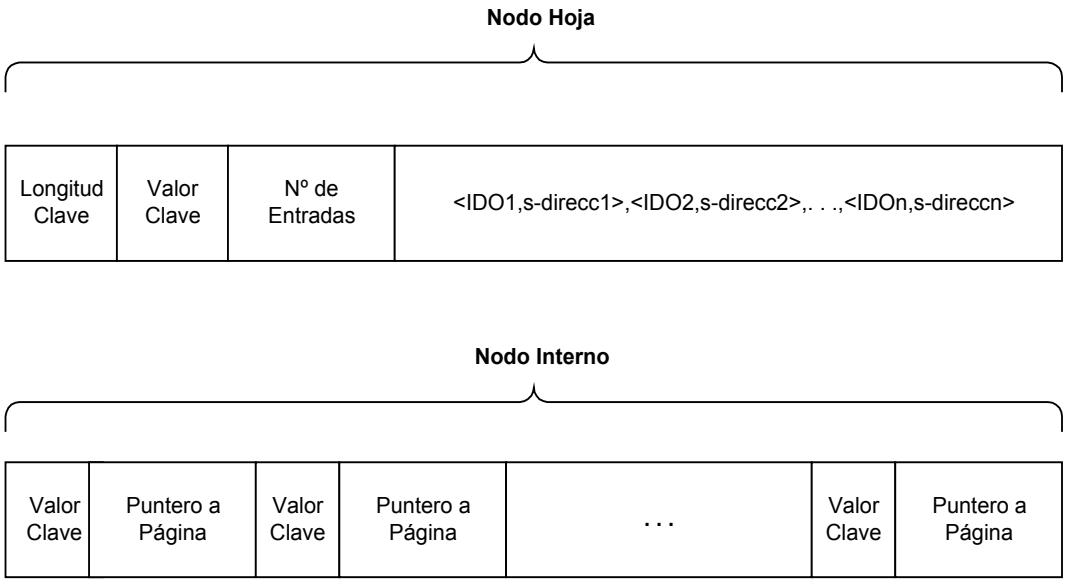


Figura 11.12. Nodos hoja e interno de un índice attribute

11.4.4.2 Operaciones

Los índices *attribute* proporcionan ventajas para el procesamiento de predicados simples con operaciones de rango e igualdad. Así, por ejemplo, en una operación de

igualdad se emplea el valor del atributo especificado en el predicado para buscar en el índice *attribute* las correspondientes direcciones de las s-expresiones, y a través de estas direcciones se pueden obtener las s-expresiones y derivar desde las s-expresiones los IDOs de la clase. Si la operación es de desigualdad, el índice *attribute* no puede emplearse, pero sí el índice *identity* y el *path dictionary*. Los índices *attribute* tampoco proporcionan beneficios para la evaluación de predicados complejos; estos requieren la búsqueda también en el índice *identity* o buscar secuencialmente en el *path dictionary*.

11.4.4.3 Posibilita la existencia de muchos índices sobre atributos

Los estudios realizados [LL98] muestran que cuando hay un atributo indexado el PD se comporta mejor en cuanto a sobrecarga de almacenamiento que el PDI y este mejor que el *Path* (PX), pero cuando hay más de un atributo indexado, la sobrecarga de almacenamiento del PX se incrementa mucho más que el PD y el PDI. Los requerimientos de almacenamiento para el PD son constantes, ya que es suficientemente general para soportar todos los tipos de consultas anidadas. Para el PDI las necesidades de almacenamiento extra para crear los índices *attribute* es baja en comparación con el PX.

11.4.4.4 El PDI tiene mejor rendimiento en la recuperación que el PX

El PDI presenta mejor rendimiento en operaciones de recuperación que el PX y además, el PX no puede emplearse para soportar consultas en las que la clase predicado es un ancestro de la clase objetivo. En general, el PDI y el PD se comportan mucho mejor que el PX cuando se considera una mezcla general de consultas anidadas.

11.4.4.5 El PX se comporta mejor en consultas de rango

El PX presenta un rendimiento en consultas de rango mejor que las otras dos opciones, y esto es debido a que los registros del nodo hoja del PX están ordenados basándose en los valores clave del atributo indexado, de forma que después de acceder al registro correspondiente al valor clave más bajo, se pueden devolver los IDOs de los objetos que interesan recorriendo los nodos hojas hasta que se alcance el valor del rango más alto. Sin embargo, el PDI tiene que acceder a las s-expresiones en el PD para devolver los IDOs de los objetos después de obtener las direcciones de las s-expresiones. El PD tiene el peor rendimiento, ya que la evaluación de predicados está basada en acceder a los objetos en la clase predicado.

11.4.4.6 El PD se comporta mejor en actualización que el PX

En operaciones de actualización el PD se comporta mejor que el PDI, y cualquiera de ellos se comporta mejor que el PX.

11.4.5 Join Index Hierarchy (JIH)

La técnica Join Index Hierarchy (*Jerarquía de Índices Join*) se basa en extender el índice *join* [Val87] de las bases de datos relacionales y sus variaciones en bases de datos espaciales [Rot91] para construir jerarquías de índices *join* que aceleren la navegación entre secuencias de objetos y clases.

En esta técnica [HXF99] un índice *join* almacena pares de identificadores de objetos de dos clases que están conectadas por medio de relaciones lógicas directas (*índices join base*) o indirectas (*índices join derivados*). JIH soporta el recorrido hacia delante y hacia atrás en una secuencia de objetos y clases, y además, soporta una propagación de actualizaciones eficiente comenzando en los índices *join* base y localizando la

propagación de actualizaciones en la jerarquía. Esta estructura es especialmente apropiada para navegaciones frecuentes y actualizaciones poco frecuentes.

Schema Path (*Camino de Esquema***)**

Un *esquema* de una base de datos es un grafo dirigido en el que los nodos corresponden a clases, y las flechas a relaciones entre clases. Así, si A(k) es un atributo de la clase C(i), y su rango es la clase C(j), entonces existe una flecha dirigida de C(i) a C(j) etiquetada como A(k). Además, si para i=0,1,...n-1 hay una flecha dirigida de C(i) a C(i+1), etiquetada como A(i+1), en el esquema de la base de datos, entonces <C(0),A(1),C(1),A(2),...,A(n),C(n)> es un *camino de esquema* (**Figura 11.13**).

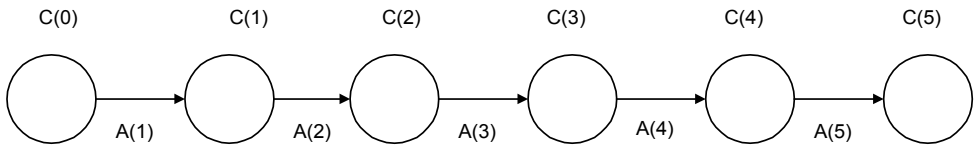


Figura 11.13. Camino de esquema de longitud 5

Join Index File *(Fichero de Índice Join)*

Dado un camino de esquema <C(0),A(1),C(1),A(2),...,A(n),C(n)> sobre un esquema de una base de datos, un *fichero de índice join,* JI(i,j) (1<=i<j<=n) consta de un conjunto de tuplas (IDO(o_i),IDO(o_j),m), dónde o_i y o_j son objetos de clases C(i) y C(j), respectivamente, y existe al menos un camino de objetos < o_i,o_{i+1},...,o_{j-1},o_j> tal que para k=0,1,...,j-i-1,el objeto o_{i+k+1} es referenciado por o_{i+k} por medio del atributo A(i+k+1), y m es el número de caminos de objetos distintos que conectan los objetos o_i y o_j.

Join Index Hierarchy *(Jerarquía de Índices Join)*

Una jerarquía de índices *join* JIH(C(0),A(1),C(1),A(2),...,A(n),C(n) ó JIH(0,n) está constituida por nodos de índices *join* que conectan diferentes objetos a lo largo de un camino de esquema. JI (0,n) representa la raíz de la jerarquía, y cada nodo JI(i,j), donde j-i>1, puede tener dos hijos directos JI (i,j-k) y JI(i+l,j) dónde 0<k<j-i y 0<l<j-i. Esta relación de padre-hijo representa una dependencia para actualización entre el nodo hijo y el padre, de forma que cuando el nodo hijo es actualizado, el padre debería ser actualizado también. En la parte inferior de la jerarquía están los nodos JI(i,i+1) para i=0,1,..,n-1, que constituyen los índices *join* base.

11.4.5.1 Estructura

La JIH presenta diferentes variaciones en función de la riqueza de las estructuras de los índices *join* derivados. Así, se presentan tres estructuras para JIH: completa, base y parcial, y en cualquiera de ellas los nodos que constituyen los JI suelen basarse en árboles B^+.

Jerarquía de Indices Join Completa (JIH-C)

Está formada por el conjunto completo de todos los posibles índices *join* base y derivados. Soporta, por tanto, la navegación entre dos objetos cualesquiera del camino de esquema conectados directa o indirectamente.

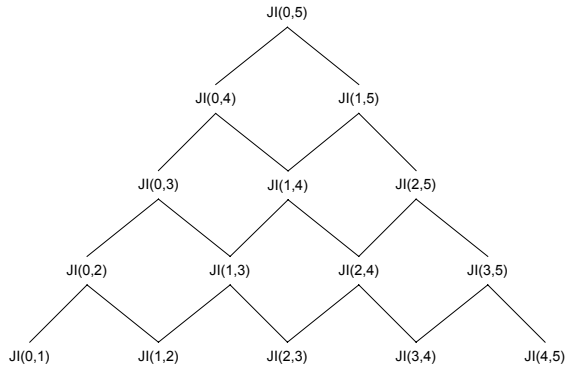


Figura 11.14. JIH completa para el camino de esquema de la Figura 11.13

Jerarquía de Indices Join Base (JIH-B)

Está formada únicamente por los índices *join* base, por lo que soporta una navegación directa solamente entre dos cualesquiera clases adyacentes. En este caso es una jerarquía degenerada en la que no aparecen los nodos JI de más alto nivel pero que pueden ser derivados desde los índices *join* base.

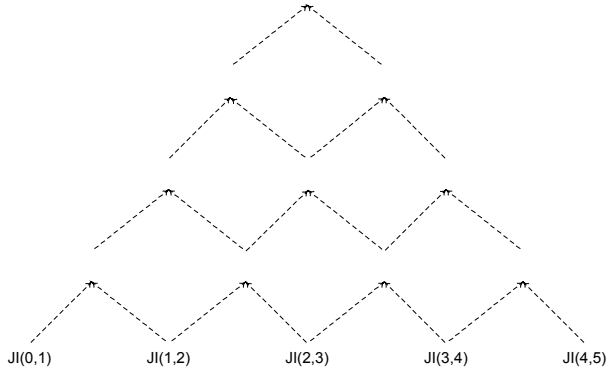


Figura 11.15. JIH base para el camino de esquema de la Figura 11.13

Jerarquía de Índices Join Parcial (JIH-P)

Está formada por un subconjunto apropiado de índices *join* base y derivados de una jerarquía de JI completa. Soporta navegaciones directas entre un conjunto de pares de clases preespecificado ya que materializa únicamente los JI correspondientes y sus índices join derivados (auxiliares) relacionados.

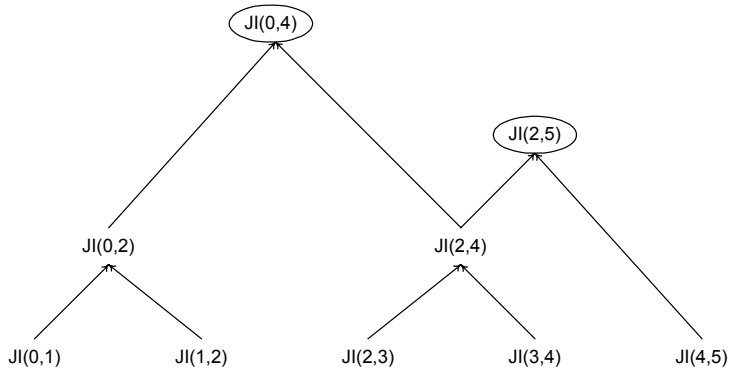


Figura 11.16. JIH parcial para el camino de esquema de la Figura 11.13

11.4.5.2 Operaciones

De las tres estructuras anteriores la parcial es la más sugerente ya que parece más beneficioso materializar únicamente los índices *join* de la jerarquía que soportan las navegaciones empleadas más frecuentemente, de ahí que en [HXF99] se especifiquen los pasos para la construcción de una jerarquía de índices *join* parcial (e indirectamente para el resto de jerarquías):

a) Encontrar el conjunto mínimo de JI auxiliares para un conjunto de índices objetivo, donde los índices objetivo son aquellos que deben ser soportados debido a referencias frecuentes.

b) Construir los JI base correspondientes, calculando JI(i,i+1) para i=0,1,...,n-1, y creando los correspondientes índices B^+ sobre i para cada JI base.

c) Construir los JI objetivo y auxiliares desde abajo hacia arriba. Esto se realiza calculando los JI auxiliares seleccionados y/o los JI objetivo desde el nivel inferior empleando la operación join ($\boxtimes_c$) y construyendo los árboles B^+ sobre i para cada JI derivado.

d) Construir los JI inversos para buscar en la dirección inversa. El JI(j,i), inverso de JI(i,j), se deriva de JI(i,j) ordenándolo sobre j en una copia de JI(i,j) y construyendo los índices B^+ sobre j.

Ante las operaciones de actualización (inserción, borrado y modificación) en los JI base es necesaria una propagación de esas actualizaciones a los JI derivados. La propagación de la actualización se realiza mediante el operador $\boxtimes_c$, que es similar al operador *join* en bases de datos relacionales. Así, JI(i,k) $\boxtimes_c$ JI(k,j) contiene una tupla $(IDO(o_i),IDO(o_j), m_1 \times m_2)$ si hay una tupla $(IDO(o_i),IDO(o_k),m_1)$ en JI(i,k) y una tupla $(IDO(o_k),IDO(o_j),m_2)$ en JI(k,j). Una actualización sobre un JI de un nivel bajo JI(C(i),C(k)) es propagado a un nivel JI de nivel superior, JI(C(i),C(j)), j>k, añadiendo δ[4]JI(i,k) $\boxtimes_c$ JI(k,j) a JI(i,j). Las tuplas idénticas se juntan en una única en la que se ha acumulado el número de caminos. En el caso del borrado si el contador es 0, se debe eliminar la tupla correspondiente. El algoritmo de actualización propuesto en detalle se encuentra en [HXF99].

11.4.5.3 JIH-C requiere más espacio y más costo de actualización

El algoritmo para la construcción de la jerarquía JIH-C es el mismo que el anterior pero sin necesidad de seleccionar los índices auxiliares, ya que aquí son materializados todos los índices. La recuperación podría ser más rápida si se emplea una jerarquía completa que si se emplea una parcial que requiere acceder a nodos virtuales que no han sido materializados directamente. Sin embargo, una jerarquía completa requiere más espacio de almacenamiento y más costo de actualización que una parcial (aunque los algoritmos son los mismos).

11.4.5.4 JIH-B se comporta peor que JIH-P y que JIH-C

Una JIH-B es construida y actualizada de forma más sencilla que las JIH-P, ya que JIH-B es una jerarquía degenerada, y por tanto, no necesita una propagación de las actualizaciones hacia arriba. Sin embargo, la navegación entre dos clases C(i) y C(i+l) en una JIH-B requiere el recorrido de una secuencia de l índices *join* base: JI(i,i+1),..., JI(i+l-1,i+l), a diferencia de un JIH-P que requiere el recorrido de uno o un número muy pequeño de JI.

[4] δ representa el conjunto de tuplas a añadir o a eliminar

11.4.5.5 División del esquema de camino para esquemas de caminos largos

A medida que se incrementa la longitud de un esquema de camino se incrementa también el tamaño de la JIH y por supuesto su costo de actualización, y es por ello necesario acudir a una partición del esquema de camino en otros más pequeños y fácilmente manejables.

11.5 Técnicas basadas en la jerarquía de herencia y de agregación

Estas técnicas de indexación proporcionan soporte integrado para consultas que implican atributos anidados de objetos y jerarquías de herencia. Es decir, son técnicas que pretenden mejorar el rendimiento en consultas que contienen un predicado sobre un atributo anidado e implican varias clases en una jerarquía de herencia dada.

El grado de complejidad de estas técnicas aumenta considerablemente al ser principalmente combinaciones de las anteriores, por lo que a continuación se describen muy brevemente. Para más información sobre los algoritmos y características ver [BF95].

11.5.1 Nested Inherited

Dado un camino P= C(1).A(1).A(2)...A(n), un índice *Nested Inherited* (NIX) asocia el valor v del atributo A(n) con los IDOs de las instancias de cada clase en el ámbito de P que tienen v como valor del atributo anidado A(n). Entendiéndose por ámbito de un camino, el conjunto de todas las clases y subclases del camino.

11.5.1.1 Estructura

El formato de un nodo no hoja tiene una estructura similar a los índices tradicionales basados en árboles B^+. Sin embargo, la organización del registro de un nodo hoja, llamado también registro primario, se basa en el del CH-Tree, en el sentido de que contiene un directorio que asocia a cada clase el desplazamiento, dentro del registro, donde son almacenados los IDOs de las instancias de la clase. Sin embargo, un NIX puede ser empleado para consultar todas las jerarquías de clases encontradas a lo largo de un camino, mientras que un CH-Tree sólo permite consultar una única jerarquía de clases. Para más detalle sobre el nodo hoja véase [BF95].

La organización del índice consta de dos índices. El primero, llamado *indice primario*, que es indexado sobre los valores del atributo A(n). Asocia con un valor v de A(n), el conjunto de IDOs de las instancias de todas las clases que tienen v como valor del atributo anidado A(n). El segundo índice, llamado *índice auxiliar*, tiene IDOs como claves de indexación. Asocia con el IDO de un objeto la lista de IDOs de sus padres. Los registros del nodo hoja en el índice primario contienen punteros a los registros del nodo hoja del índice auxiliar, y viceversa. El índice primario es utilizado para las operaciones de recuperación, mientras que el índice secundario es empleado, básicamente, para determinar todos los registros primarios donde son almacenados los IDOs de una instancia dada, con el fin de realizar eficientemente las operaciones de inserción y borrado.

11.5.1.2 Operaciones

Esta técnica es apropiada para evaluaciones rápidas de predicados sobre el atributo indexado en consultas centradas en una clase, o jerarquía de clases, para el ámbito del camino que finaliza con el atributo indexado. Primeramente, se ejecuta una búsqueda

sobre el índice primario con el valor clave igual al buscado. Entonces es accedido el registro primario, y se realiza una búsqueda en el directorio de clases para determinar el desplazamiento dónde están los IDOs de la clase consultada. Si la consulta se realiza no sobre la clase si no sobre toda la jerarquía cuya raíz es la clase consultada se realizan los mismo pasos pero con la diferencia de que la búsqueda en el directorio de clases se realiza para todas las clases de la jerarquía, con lo que hay que acceder a diferentes partes del registro (uno para cada desplazamiento obtenido desde la búsqueda en el directorio de clases).

11.5.1.3 Apropiado cuando todos los atributos del camino son de valor-simple

Los estudios realizados indican que esta técnica es apropiada cuando todos los atributos en el camino indexado son simples; o cuando únicamente las primeras clases en el camino tienen atributos multi-valuados, mientras que todas las demás clases tienen atributos simples. En el resto de situaciones esta técnica tiene un costo superior a IMX.

11.5.2 Inherited MultiIndex

Dado un camino P= C(1).A(1).A(2)...A(n) un *Inherited MultiIndex* (IMX) se basa en asociar un índice sobre cada clase C(i), del conjunto de clases que interviene en el camino, que asocia los valores del atributo A(i) con los IDOs de las instancias de C(i) y todas sus subclases.

11.5.2.1 Estructura

La estructura de esta técnica son los CH-Trees (basados a su vez en los B^+). Se dispondrá de un número de índices igual a la longitud del camino, ya que se empleará un índice CH-Tree para cada jerarquía de herencia que exista en el camino que se está indexando.

11.5.2.2 Operaciones

Para la recuperación el IMX necesita recorrer un número de índices (CH-Trees) igual al número de clases del camino, independientemente de que se consulte únicamente la clase raíz de una jerarquía o bien toda la jerarquía.

11.5.2.3 Apropiado cuando las consultas son principalmente sobre la última clase del camino indexado

El rendimiento del IMX es mejor que el del NIX cuando las consultas son principalmente sobre la última clase del camino indexado, por lo tanto, el NIX debe emplearse principalmente cuando el número de predicados anidados en la consulta es elevado.

11.6 Técnicas basadas en la invocación de métodos

El uso de métodos es importante porque define el comportamiento de los objetos, sin embargo, tiene dos efectos negativos desde el punto de vista del procesamiento y optimización de consultas[YM98]:

- Es caro invocar un método, ya que eso ocasiona la ejecución de un programa, y si esto tiene que realizarse sobre un conjunto grande de objetos ralentizará el procesamiento de la consulta.

- La implementación de cada método está oculta (por la encapsulación del modelo OO), por lo que es muy difícil estimar el coste de su invocación en una consulta.

11.6.1 Materialización de métodos

La *materialización de métodos* (MM) es una técnica que pretende aliviar los problemas anteriormente mencionados. La idea básica consiste en calcular los resultados de los métodos accedidos frecuentemente, *a priori,* y almacenar los resultados obtenidos para emplearlos más tarde.

11.6.1.1 Estructura

El resultado materializado de un método puede ser almacenado en una tabla con la estructura mostrada en la **Figura 11.17**. Como el tamaño de esta tabla puede llegar a ser muy grande, se pueden crear índices para acelerar la velocidad de acceso a la misma.

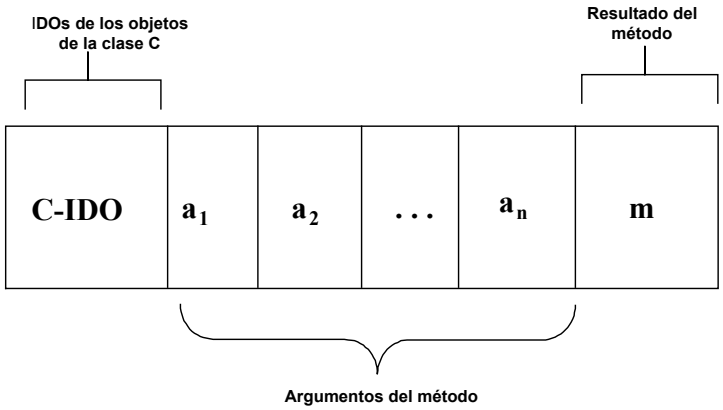


Figura 11.17. Formato de la tabla que almacena los métodos materializados

Cuando varios métodos para la misma clase tengan el mismo conjunto de argumentos, los resultados materializados de estos métodos pueden ser almacenados en la misma tabla, con lo que se necesita menos espacio, y si además estos métodos son referenciados en la misma consulta, sólo sería necesario acceder a una tabla para evaluar estos métodos, lo que se traduciría en un coste de procesamiento más bajo.

Si los resultados de cada método referenciado en una consulta han sido precalculados, será mucho más fácil para el optimizador de consultas generar un buen plan de ejecución para la consulta.

11.6.1.2 Operaciones

La materialización de métodos permite un procesamiento más eficiente de las consultas de recuperación, pero supone un costo más alto para las consultas de actualización, ya que, una operación de actualización puede ocasionar que el resultado precalculado llegue a ser inconsistente con los datos actualizados.

Una cuestión importante a tener en cuenta en la materialización de métodos es el manejo eficiente de este problema de inconsistencia. Así, cuando se borra un objeto, todas las entradas materializadas que usan ese objeto deben ser borradas, y cuando un objeto existente es modificado, todos los resultados materializados que usen el objeto modificado necesitan ser recalculados. Este proceso puede ser inmediato, o retrasado

hasta que el resultado de la instancia del método sea necesario para evaluar la consulta (*lazy materialization*). En [KKM94] se presentan además varias técnicas que permiten reducir el costo de mantenimiento de los métodos materializados.

11.7 Resumen de características de las técnicas revisadas

Técnica	Carácter. OO	Estructura	Apropiada para...	Característica Destacable
SC	JH	Árbol B^+	Consultas sobre una única clase en la jerarquía	Emplea un árbol para cada clase en la jerarquía.
CH-Tree	JH	Árbol B^+ nodo hoja modificado	Consultas sobre más de una clase en la jerarquía	Emplea un único árbol para todas las clases de la jerarquía.
H-Tree	JH	Árbol B^+ nodo hoja modificado nodo interno modificado	Consultas de recuperación que afectan a pocas clases en la jerarquía	Emplea un árbol para cada clase, anidándolos de acuerdo a la relación subclase-superclase.
HcC-Tree	JH	Árbol B^+ Cuatro tipos de nodos: -nodo interno y hoja -nodos IDO cadena -nodos IDO jerarquía	Consultas que implican que los valores para el atributo estén dentro de un rango	Emplea un único árbol para la jerarquía, con cadenas de clase y cadenas de jerarquía.
PX	JA	Árbol B^+ nodo hoja modificado	Consultas que impliquen la evaluación de predicados anidados sobre todas las clases del camino	Almacena las instanciaciones de un camino cuya clave es el objeto al final del camino de instanciación.
NX	JA	Árbol B^+ nodo hoja modificado	Consultas de recuperación con jerarquías de agregación	Establece una conexión directa entre el objeto del principio y el del final del camino de instanciación.
MX	JA	Árbol B^+ nodo hoja modificado	Consultas de actualización	Divide el camino en subcaminos, asignándoles a cada uno un índice NX.
PDI	JA	Tres estructuras: -Path Dictionary -Attribute Index (B+) -Identity Index (B+)	Consultas que implican búsquedas asociativas y recorrido entre objetos	Emplea una estructura (PD) para el recorrido entre objetos, y otra diferente (AI) para las búsquedas asociativas.
JIH	JA	Tres posibilidad. -JIH-C (B+) -JIH-B (B+) -JIH-P (B+)	Consultas con navegaciones frecuentes y actualizaciones poco frecuentes	Presenta variedades en función de la riqueza de las estructuras de los *join* derivados.
NIX	JH JA	Dos estructuras: -Indice primario (B+) -Indice auxiliar (B+)	Consultas en las que los atributos en el camino sean simples, y el número de predicados anidados elevado	El índice primario es empleado para operaciones de recuperación; el auxiliar para operaciones de actualización.
IMX	JH JA	CH-Tree	Consultas sobre la última clase del camino indexado	El número de índices es igual al número de clases del camino.
MM	IM	Tabla	Consultas de recuperación que implican invocación de métodos	Métodos de la misma clase con los mismos argumentos pueden almacenarse en la misma tabla.

JH = Jerarquía de herencia, JA = Jerarquía de agregación, IM = Invocación de métodos

Capítulo 12 Mecanismo de indexación

En el capítulo 10 se describió una arquitectura de referencia para el motor del SGBDOO BDOviedo3, identificándose un módulo encargado de la gestión de los índices en el sistema, y en el capítulo 11, se realizó una revisión exhaustiva sobre las técnicas de indexación en bases de datos orientadas a objetos con las que dotar al mecanismo de indexación del SGBDOO. En este capítulo se describen los objetivos del mecanismo de indexación, sus características fundamentales, y se justifican algunas de las decisiones más importantes tomadas en su diseño.

12.1 Introducción

Tradicionalmente, la indexación es empleada en los SGBD para conseguir un acceso eficiente a los datos almacenados en la base datos, facilitando el procesamiento de las consultas.

En el caso de las bases de datos orientadas a objetos, la riqueza del modelo de objetos complica las técnicas de indexación clásicas con el fin de proporcionar un rendimiento óptimo ante las diferentes posibilidades que ofrece dicho modelo. De hecho, y como se pudo observar en el capítulo anterior unas técnicas proporcionan mejores rendimientos que otras para determinadas características del modelo de objetos (véase Figura 12.1). Y además, con los estudios realizados no se puede concluir que una técnica de indexación sea mejor que las demás en todas las circunstancias, o al menos bajo las condiciones en las que han sido probadas.

Por otro lado, los SGBDOO surgen para dar cabida a un amplio rango de aplicaciones en las que los datos a tratar y por tanto a indexar, con el fin de proporcionar un procesamiento eficiente de las consultas, pueden ser muy variados. Esto obliga a tener en cuenta las siguientes consideraciones:

- Se puede desear indexar un atributo cuyo tipo ha sido definido por el usuario, y que por tanto no era conocido por la técnica de indexación cuando ésta fue implementada.
- Determinadas técnicas de indexación pueden tener un rendimiento probado con determinados tipos de datos, pero no se puede asegurar su comportamiento para otros nuevos tipos de datos. Así por ejemplo, los árboles B^+ se comportan bien con tipos de datos básicos (ej. enteros), pero no se puede asegurar su comportamiento para otros tipos de datos definidos por el usuario.

Esta flexibilidad en el desarrollo de aplicaciones y en la definición de tipos de datos hace muy difícil predecir o anticipar cuales serán las técnicas que mejor se comporten para las nuevas situaciones que se pueden presentar.

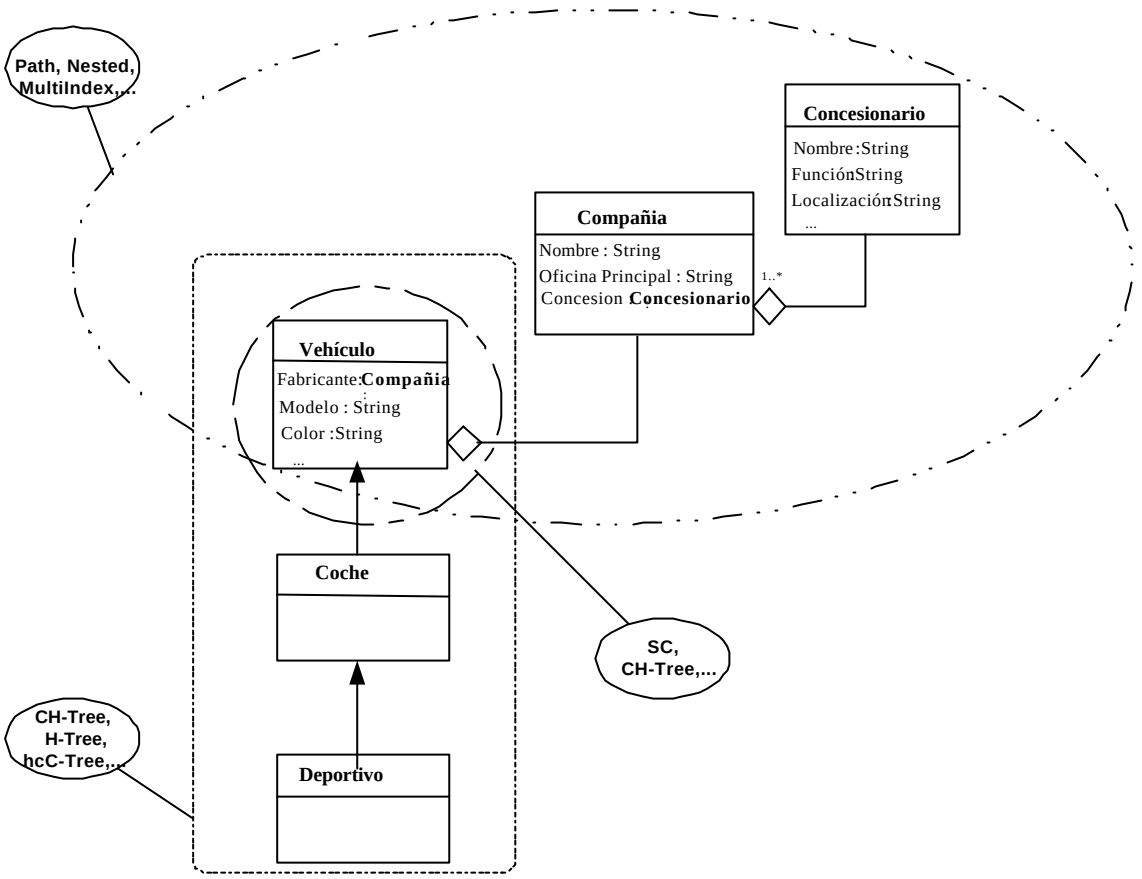


Figura 12.1. Posibles técnicas de indexación a aplicar en función del tipo de consulta

12.2 Objetivos del mecanismo

El mecanismo de indexación que se incorpore en BDOviedo3, ha de dar por tanto solución a las situaciones anteriores. Para ello hay inicialmente dos posibilidades:

- **Mecanismo de indexación rígido**[1]**,** basado en la incorporación al sistema del mayor número de técnicas posibles. Es decir, que el mecanismo de indexación del SGBDOO incluya el mayor número posible de técnicas de indexación, preparándolas para aceptar diferentes tipos de datos, de forma que se incrementen las posibilidades de encontrar la más adecuada ante los diferentes patrones de aplicación. Esta opción no es viable, debido entre otras cosas, a la sobrecarga que supondría incorporar al sistema un número elevado de técnicas de indexación, cuando es posible que no se utilicen nunca. Otro inconveniente claro, es que ante el surgimiento de una nueva técnica de indexación de eficacia probada para un determinado patrón de aplicación, sería imposible incorporarla.
- **Mecanismo de indexación extensible,** que permite la incorporación de nuevas técnicas en el mecanismo de indexación, adecuadas al tipo de dato o consulta más frecuente a realizar. Esta es la solución tomada por algunos sistemas como Starburst [LMP87], POSTGRES [Aok91] u Oracle [Ora99a], y por algunos generadores de bases de datos como EXODUS [CDF+86]. Esta parece la

[1] La mayoría de los SGBDOO existentes emplean este modelo para su mecanismo de indexación pero incluyen un número muy pequeño de técnicas de indexación (dos o tres a lo sumo)

solución más razonable para dar una solución completa a los problemas planteados anteriormente, y además desde el punto de vista de un proyecto de investigación como éste, sin duda es la más adecuada. En estos sistemas se intenta que la flexibilidad de la que se dota al mecanismo no repercuta en el rendimiento general del producto.

Se pueden entonces deducir dos objetivos fundamentales a perseguir por el mecanismo de indexación para BDOviedo3, y que están estrechamente relacionados entre si.

12.2.1 Extensibilidad

El objetivo principal perseguido por este mecanismo de indexación es la extensibilidad, entendiéndose por indexación extensible **la capacidad por la que un usuario puede incorporar nuevos índices a un SGBD** [Sto86]. Y tradicionalmente, la literatura remarca que esta capacidad es particularmente útil cuando es empleada en conjunción con la **incorporación de nuevos tipos de datos** y **nuevos operadores** al sistema.

Un mecanismo de indexación que facilitase la extensibilidad del sistema permitiría, por ejemplo, que ante la necesidad de almacenar un nuevo tipo de datos (ej. datos para un sistema de información geográfica), el sistema fuese capaz de aceptar y reconocer una nueva técnica de indexación que tratase eficientemente el nuevo tipo de dato introducido.

12.2.2 Plataforma de experimentación

Es objetivo de esta tesis convertir BDOviedo3, y en concreto el mecanismo de indexación, en una buena plataforma de experimentación, con el fin de examinar el comportamiento de las diferentes técnicas existentes u otras nuevas, atendiendo a diferentes criterios: número de clases de la jerarquía, población elevada de instancias, agregaciones múltiples, distribución de las claves, etc.

Este objetivo está ligado al anterior, ya que con un sistema rígidamente diseñado no se plantearía la posibilidad de obtener resultados experimentales, por la imposibilidad de incorporar modificaciones e incluir nuevas técnicas en el sistema.

12.3 Características fundamentales del mecanismo

El mecanismo de indexación de BDOviedo3 debe proporcionar las siguientes características con el fin de cumplir los objetivos anteriormente mencionados. La mayoría de estas características están directamente relacionadas con el concepto de indexación extensible.

12.3.1 Soporte para nuevos tipos de datos

Hace referencia a la posibilidad de incorporar al SGBD nuevos tipos de datos, de forma que el sistema no tenga que tratar exclusivamente con los tipos de datos básicos. Esta es una imposición heredada de la extensibilidad en bases de datos relacionales que fueron las pioneras en dar este tipo de soporte, ya que en las bases de datos orientadas a objetos esta características ya forma parte de su filosofía.

Las técnicas de indexación que se incorporan en los SGBD relacionales, tradicionalmente operan con tipos de datos básicos como pueden ser enteros, reales o cadenas, porque éstos son los tipos predefinidos del sistema. Es por ello que, la primera

exigencia que se hace cara a conseguir indexación extensible en un SGBD relacional u objeto-relacional es la posibilidad de incorporar al sistema nuevos tipos de datos, ya que para los primitivos se suponen que están incorporadas las técnicas con mejores resultados.

En el caso de una base de datos orientada a objetos, y concretamente en la que nos ocupa, esto no es una exigencia, es algo ya inherente al sistema. El SGBDOO será capaz de tratar tanto con los datos predefinidos en el sistema (*String*, *Integer*...) como con nuevos tipos de datos definidos por el usuario (*bloque*, *caja*, etc.).

12.3.2 Indexación aplicable a cualquier tipo de datos

Siguiendo con la tradición de los SGBD relacionales, algunos SGBDOO (o gestores de objetos) únicamente permiten indexar sobre atributos de tipos primitivos (*Integer*, *String*, etc.), como por ejemplo Jeevan [Jee99]. Sin embargo, el objetivo de un mecanismo de indexación extensible es permitir la indexación sobre cualquier atributo sea éste de un tipo primitivo o no.

12.3.3 Incorporación de nuevas técnicas de indexación

La incorporación de nuevas técnicas de indexación representa la esencia de un mecanismo de indexación extensible, y es objetivo de este sistema que esa incorporación se lleve a cabo de una forma sencilla.

La incorporación de técnicas de indexación en un sistema, como ya se comentó con anterioridad va a obedecer generalmente a una o varias de estas causas:

- La nueva técnica se adecua mejor al tipo de dato a indexar. Por ejemplo, si se construye una nueva técnica que se comporta mejor para la indexación sobre los enteros que los árboles B^+.
- La incorporación de un nuevo tipo de dato que necesite un método de acceso más apropiado.
- La obtención de un rendimiento adecuado en función del ámbito de acceso de las consultas más usuales, atendiendo al modelo de objetos (jerarquías de herencia, de agregación, etc.), tal y como se mencionó en el capítulo anterior.

12.3.4 Integración con el optimizador de consultas

Esta característica es muy importante, y además de obligado cumplimiento, para que la extensibilidad en el mecanismo de indexación tenga efectividad.

Las técnicas de indexación surgen principalmente para proporcionar una recuperación eficiente en las consultas. Cuando esas técnicas son fijas y están incorporadas en el núcleo del SGBD permanentemente, el optimizador de consultas tiene un perfecto conocimiento de ellas y puede emplearlas adecuadamente.

Sin embargo, cuando el sistema permite la incorporación de nuevas técnicas es necesario que el optimizador sea consciente de ellas para poder emplearlas apropiadamente. Eso implica también que, cuando se proporciona un mecanismo de indexación extensible, el optimizador de consultas ha de estar preparado para aceptar la posibilidad de considerar nuevas técnicas de indexación.

12.3.5 Configurabilidad

Una de las posibilidades que se comentó inicialmente para conseguir un mecanismo de indexación abierto a nuevos tipos de datos y solicitudes de información, era la de la

inclusión de la mayor cantidad posible de técnicas de indexación. Sin embargo, esta opción fue desestimada, entre otras cosas, por la sobrecarga que eso podía ocasionar para el sistema, ante la posibilidad de que hubiese un gran número de situaciones en las que esas técnicas fuesen innecesarias.

Partiendo de un mecanismo de indexación extensible, como el que se presenta aquí, también puede ser interesante proporcionar la posibilidad de configurar el sistema para incluir aquellas técnicas que se deseen que formen parte del mismo en una determinada situación, de forma que aquellas técnicas que no sean necesarias puedan no ser consideradas, evitando así una sobrecarga innecesaria al sistema.

En el caso de BDOviedo3 la configurabilidad del mecanismo es facilitada por el funcionamiento del sistema integral. Es decir, en este sistema, únicamente se cargan en memoria, aquellas clases a las que se referencia de manera dinámica. Para que no supongan una sobrecarga en disco, sería suficiente con no compilar las clases de los índices que no se vayan a emplear.

12.4 Alternativas a considerar para el diseño del mecanismo

De todas las características mencionadas en el apartado anterior se pueden destacar dos fundamentalmente, cara a conseguir la extensibilidad en el mecanismo de indexación: la posibilidad de incorporar nuevas técnicas de indexación, así como el hecho de que la indexación sea soportada sobre cualquier tipo de dato, no únicamente tipos primitivos.

Para dar solución a estas cuestiones, y tras una revisión exhaustiva de los mecanismos empleados por los sistemas extensibles existentes actualmente, se plantean al menos dos posibilidades.

12.4.1 Núcleo fijo con extensiones

Este modelo se basa en plantear el mecanismo de indexación como un núcleo fijo con el que trabajará el sistema por defecto, y que podrá ser incrementado con nuevas técnicas cuando se consideren necesarias. Por supuesto, las extensiones añadidas serán tenidas en cuenta para la evaluación de las consultas, de la misma manera que lo son las ya predefinidas en el sistema. Este es el modelo empleado, por ejemplo, en Oracle8i [Ora99b].

Mecanismo de indexación = núcleo fijo + extensiones

12.4.1.1 Núcleo

El **núcleo** está constituido por un conjunto de técnicas predefinidas preparadas para trabajar con los tipos primitivos que soporta el sistema. Por ejemplo, suelen incorporar árboles B^+ para trabajar sobre los tipos primitivos básicos como enteros o cadenas.

El núcleo representa la parte no modificable del mecanismo.

12.4.1.2 Extensiones

Las **extensiones** están constituidas por técnicas de indexación empleadas para un determinado tipo de dato. En muchos casos puede llevar asociado, incluso, la definición de nuevos operadores que pueden ser tratados por el lenguaje de consulta.

Por ejemplo, se puede incorporar un índice al sistema con el objetivo de indexar documentos completos, y un operador **contiene** que indica si una palabra está o no en ese documento. En estos casos, el lenguaje de consulta tiene que estar preparado para admitir nuevos tipos de operadores.

Las extensiones tienen que proporcionar una serie de funciones comunes (una interfaz común), tales como, insertar o borrar un elemento, que permitan incorporarlas a los servicios básicos del sistema, y ser consideradas por el resto de elementos del SGBD, como por ejemplo, el optimizador de consultas.

12.4.1.3 Inconveniente

Este modelo cumple los objetivos planteados para el mecanismo, por lo que sería perfectamente válida su aplicación. No obstante, en determinadas circunstancias puede ocasionar una duplicidad en el código fuente, así como una sobrecarga innecesaria en el sistema.

Supóngase que entre las técnicas que incluye el núcleo se encuentra implementada la técnica B^+. Y supóngase ahora que se desea emplear la técnica B^+ para indexar datos del tipo 'caja'. Con este modelo sería necesario volver a codificar la técnica B^+ pero ahora para tipos de datos 'caja', ya que la técnica implementada en el núcleo únicamente trabaja con datos de tipo primitivo.

12.4.2 Núcleo extensible

Este modelo se basa en diseñar el mecanismo de indexación basándose en un núcleo extensible que permite la incorporación de nuevas técnicas, pero planteándose desde el principio que dichas técnicas no tienen porque emplearse únicamente con datos de tipo primitivo. Este es el modelo empleado por ejemplo en POSTGRES [Aok91], y la filosofía empleada en EXODUS [CDF+86].

Esta posibilidad plantea dos consideraciones a tener en cuenta:

- En primer lugar, las técnicas de indexación han de ser implementadas para aceptar cualquier tipo de datos.
- En segundo lugar, y relacionado con lo anterior, será necesario especificar el comportamiento de cada operador para cada tipo o clase que se desee indexar.

12.4.2.1 Especificación del comportamiento de los operadores

De lo anterior se deduce que el mecanismo de indexación ha de permitir la aplicación de la técnica de indexación sobre cualquier tipo de dato de los empleados en el esquema de la base de datos. Pero entonces se plantean algunos problemas con los operadores empleados. Si, por ejemplo, se toma como base una técnica de indexación clásica como es el árbol B^+ (*SC*), y el índice se establece sobre el objeto 'caja', es necesario que el sistema conozca cuando un objeto 'caja' es menor o mayor que otro.

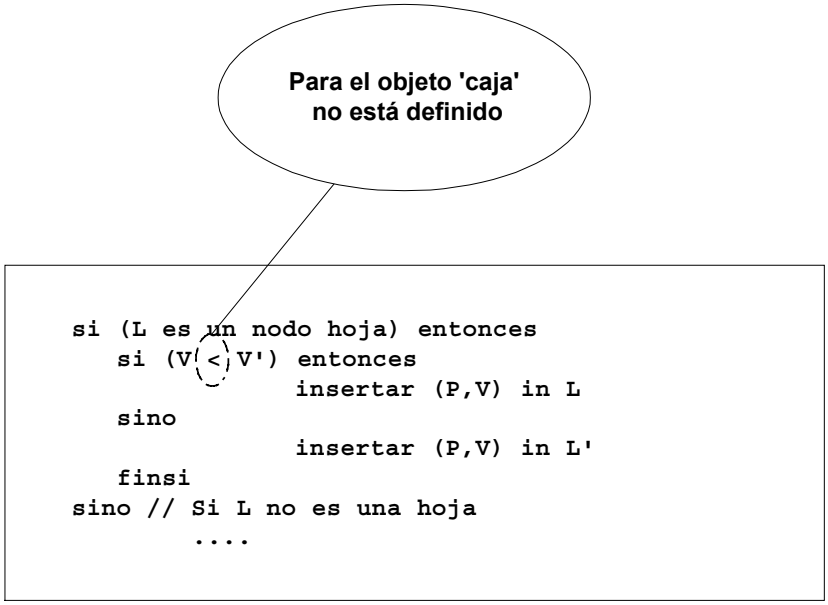


Figura 12.2. Empleo de los operadores en la inserción en un árbol B^+

En definitiva, el comportamiento de los operadores no está definido para los nuevos tipos incorporados, por lo que será necesario especificar cuál es el comportamiento de los operadores empleados por la técnica de indexación correspondiente para cada nuevo tipo de dato incorporado al sistema, y que se desee indexar.

Por otro lado, es muy probable que sea necesario tener que incorporar técnicas de indexación no basadas en árboles B+, como por ejemplo los R-Tree, y cuyos operadores (contenido-en, igual, contenido-en-o-igual) no sean por tanto ninguno de los anteriores. El comportamiento de esos operadores también debe ser especificado explícitamente.

12.5 Cuestiones de diseño del mecanismo

Una vez establecidos los objetivos que debe cumplir el mecanismo de indexación de BDOviedo3, y examinadas las posibles alternativas para proporcionar la extensibilidad en el mecanismo de indexación, se van describir algunas de las decisiones tomadas a la hora de diseñar dicho mecanismo.

12.5.1 Marco orientado a objetos

La mayoría de las soluciones existentes para la extensibilidad en los mecanismos de indexación están basado en tablas. Es así, por ejemplo, el caso de POSTGRES [SR86], que proporciona una extensibilidad conducida por tablas en la que se permiten incorporar nuevos tipos de datos y nuevos métodos de acceso para esos datos, pero la solución pasa por la modificación de una gran maraña de tablas (una para los métodos de acceso, otra para los operadores, otra para los procedimientos, y otras para establecer las relaciones entre las anteriores), cuyas relaciones a veces quedan poco claras, lo que se traduce en un proceso nada sencillo de realizar.

En el caso que nos ocupa, al estar diseñando un mecanismo de indexación para un SGBDOO construido sobre un sistema integral que facilita un determinado modelo de objetos común a todos los elementos del sistema, parece lógico que en su construcción se apliquen los principios del paradigma de la orientación a objetos, quedando el mecanismo organizado como un marco orientado a objetos, en el que cada una de las clases ofrece una determinada funcionalidad. De hecho, características como la herencia

y el polimorfismo simplificarán la construcción del mecanismo facilitando su extensibilidad.

El objetivo es explotar al máximo las características de la orientación a objetos. Para ello se van a definir clases abstractas con métodos virtuales. Estas clases inicialmente no asumen ningún conocimiento acerca de las implementaciones de las diferentes técnicas de indexación.

12.5.2 Núcleo extensible

El modelo empleado para este mecanismo será el del núcleo extensible. Se ha seleccionado este modelo ya que proporciona un mecanismo extensible en el que es posible aprovechar la codificación realizada de las técnicas de indexación independientemente del tipo de dato a indexar.

A cambio de estas prestaciones hay que considerar que la construcción del mecanismo exigirá:

- **Recursos para una implementación genérica de las técnicas de indexación**. Estos recursos serán suministrados por el sistema integral.
- **Especificación del comportamiento de los operadores.** Para ello se acude a una clase que permitirá especificar dicho comportamiento.

12.5.3 Recursos proporcionados por el sistema integral

La incorporación de nuevas técnicas de indexación, según el modelo anteriormente seleccionado, exige que la programación de dichas técnicas se realice de la forma más genérica posible teniendo en cuenta que ha de poder ser utilizada por diferentes tipos de datos (enteros, cadenas, caja, etc.). Esto implica que es necesario que el lenguaje empleado para la implementación de las técnicas proporcione algún mecanismo para facilitar dicha programación genérica.

Por otro lado, y pese a lo que pudiera parecer, la programación de las técnicas no es la parte más laboriosa de todo el proceso, representa el 30% del tiempo empleado en el mismo [CDF+86]. Existen otras tareas, como por ejemplo, la traducción de las estructuras empleadas por el método de acceso a los objetos primitivos soportados por el sistema de almacenamientos subyacente, que consumen más tiempo que la propia programación de la técnica [CDF+86].

Sin embargo, el sistema integral orientado a objetos subyacente al sistema BDOviedo3 facilita bastante dichas tareas.

12.5.3.1 No es necesaria traducción de las estructuras de almacenamiento

Al realizar la implementación de los métodos de acceso mediante el empleo de los objetos del propio sistema, y éstos tener la capacidad de persistir en el sistema subyacente, no es necesaria ninguna traducción explícita entre ambas representaciones porque en realidad son una única. Otros sistemas como EXODUS acuden al empleo de variables de objetos persistentes proporcionadas por el lenguaje E [CDF+86].

12.5.3.2 Facilita la implementación de técnicas de indexación genéricas

Para el sistema todos son objetos, sin distinguir cuáles son del sistema y cuales son definidos por el usuario, y todos son tratados uniformemente (todos descienden de la clase *Object*). Esto favorece la implementación de las distintas técnicas de indexación de una forma natural, sin necesidad de acudir a otros elementos artificiales como los

templates, empleados por ejemplo, en el lenguaje E para la implementación de los métodos de acceso [RC87].

12.5.4 Representación de las técnicas de indexación

Las técnicas de indexación serán representadas mediante una clase abstracta (TI), cuya misión es guiar, y facilitar la incorporación de nuevas técnicas de indexación en el sistema, especificando aquellos métodos que debe proporcionar obligatoriamente cada nuva técnica de indexación que se desee incorporar. Esta clase es básica y central en el diseño.

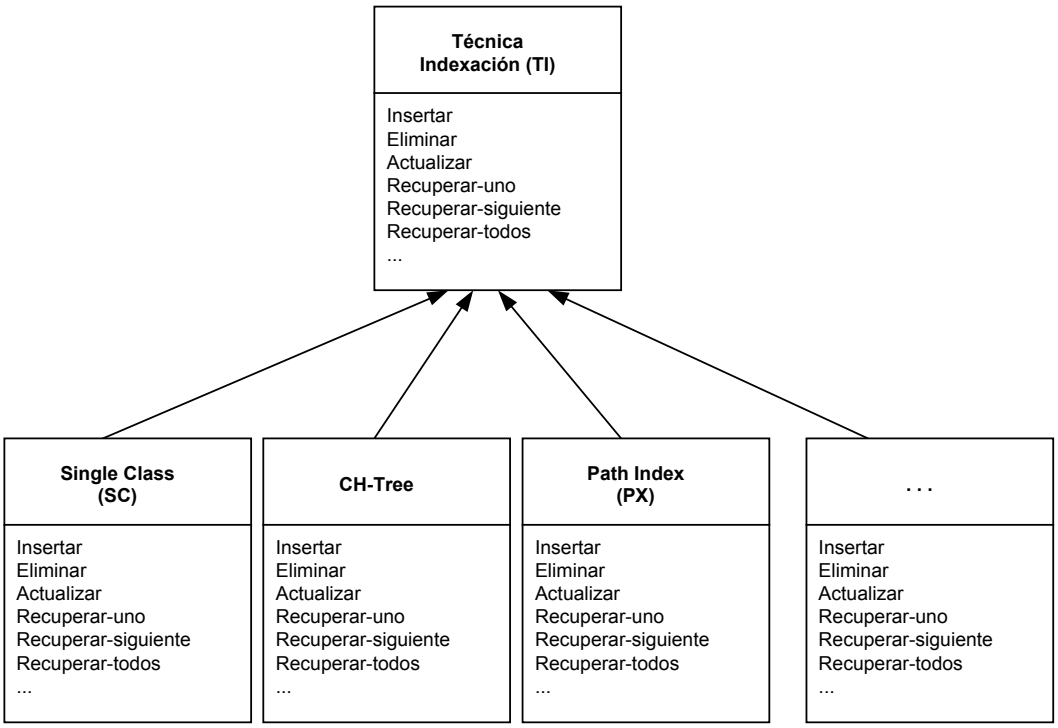


Figura 12.3. Jerarquía de clases para las técnicas de indexación

La funcionalidad a proporcionar por esta clase se puede clasificar en dos apartados: mantenimiento y recorrido de índices.

Métodos de Mantenimiento de índices

Son los métodos que permiten la manipulación del contenido de los índices propiamente:

- **Insertar.** Inserta un objeto en la estructura que representa al índice.
- **Eliminar.** Elimina un objeto de la estructura que representa al índice.
- **Actualización.** Refleja en el índice cambios realizados en un objeto ya indexado

Métodos de recorrido de índices

Son los métodos que permiten recuperar información almacenada en un índice previa evaluación de predicados que contienen operadores. Los métodos inicialmente podrían ser:

- **Recuperar-uno.** Recupera el primer elemento que cumple la condición que se le pasa como parámetro

- **Recuperar el siguiente.** Recupera el siguiente elemento que cumple la condición. Estos dos métodos permitirían ir obteniendo uno a uno todos los objetos que cumplen las condiciones especificadas.
- **Recuperar todos.** Recupera todas las instancias que cumplen unas determinadas condiciones.

12.5.4.1 Incorporación de nuevas técnicas al sistema

Cuando se desee añadir una nueva técnica de indexación al mecanismo, el primer paso será construir una nueva clase que herede de esta clase abstracta e implemente los métodos anteriormente especificados. Esta interfaz común facilitará el empleo de modo uniforme de este mecanismo, independientemente de las técnicas que estén implementadas, por el resto de subsistemas del motor.

12.5.5 Representación de los operadores

En un mecanismo de indexación como el descrito en este trabajo hay dos elementos fundamentales: las técnicas de indexación (cuya posible representación ha sido descrita en el apartado anterior) y los operadores.

Para representar los operadores en el sistema, una posibilidad puede ser emplear una clase abstracta (*operador*) con la interfaz común de todos los operadores especificados para el sistema.

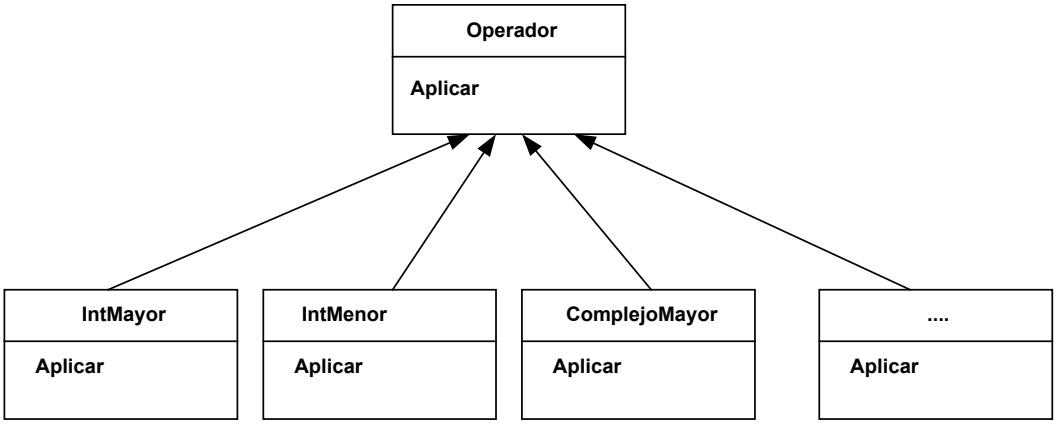


Figura 12.4. Representación de los operadores

Cada técnica de indexación necesita obligatoriamente una serie de operadores para su implementación. Por ejemplo, en el caso de la técnica B^+ y sus derivados (PX, NX, etc.), dichos operadores serán el mayor, menor, igual, mayor o igual y menor o igual. En el caso de la técnica R-Tree, por ejemplo, los operadores podrían ser contenido-en, igual y contenido-en-o-igual.

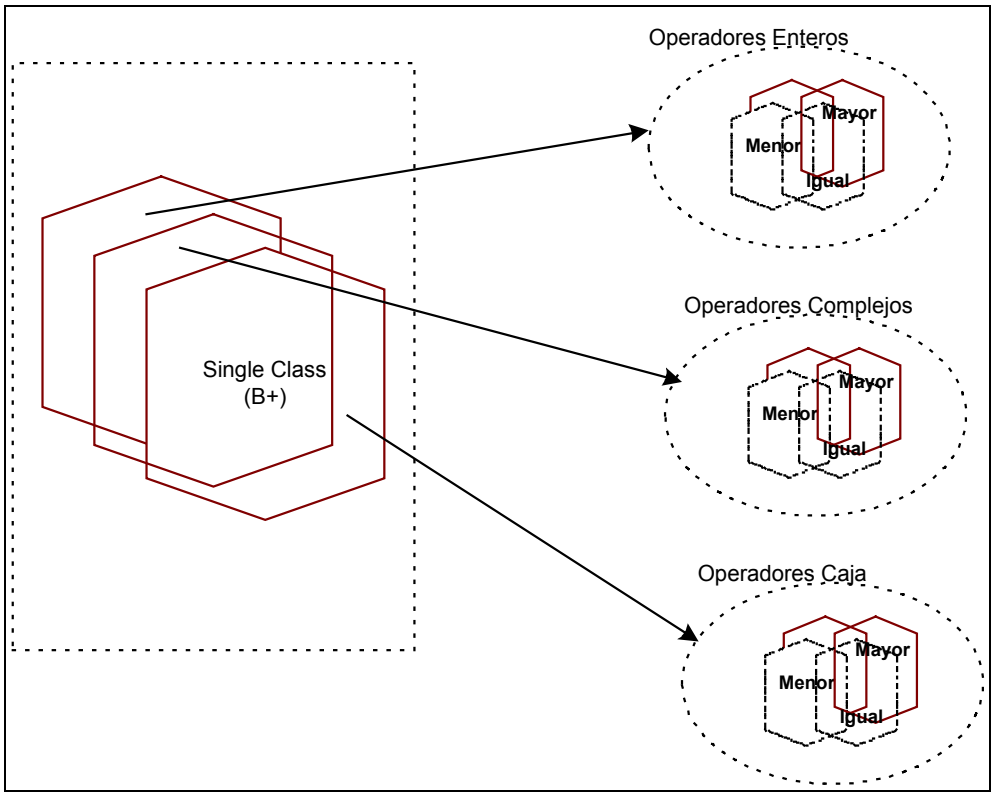


Figura 12.5. Cada técnica de indexación debe conocer el conjunto de objetos operador a emplear

Por otro lado, para cada clase a indexar será necesario especificar el comportamiento de los operadores correspondientes a la técnica de indexación a emplear. Por ejemplo, para indexar por una clase que sea 'complejos' (haciendo referencia a números complejos) mediante un árbol B^+, será necesario especificar el comportamiento de los operadores (mayor, menor, etc.) para dicha clase. Cada uno de estos operadores supondrá la creación de una clase, que herede de la clase operador y especifique dicho comportamiento.

12.5.6 Catálogo para la indexación

Una vez especificado el comportamiento de los operadores para las técnicas a emplear en la indexación en función de las clases a indexar, será necesario conocer siempre que operadores se deben emplear para cada índice creado a petición del usuario.

Será necesario, por tanto, una especie de catálogo para la indexación, que podría ser representado por una clase (CatálogoIndices), que almacene entre otras cosas, la siguiente información:

- **Atributo a indexar** (que será de una determinada clase)
- **Técnica de indexación a emplear**
- **Conjunto de operadores** correspondientes a la clase a la que pertenece el atributo indexado y que se van a emplear con esa técnica de indexación. Puede ocurrir que una misma clase tenga diferentes conjuntos de operadores.

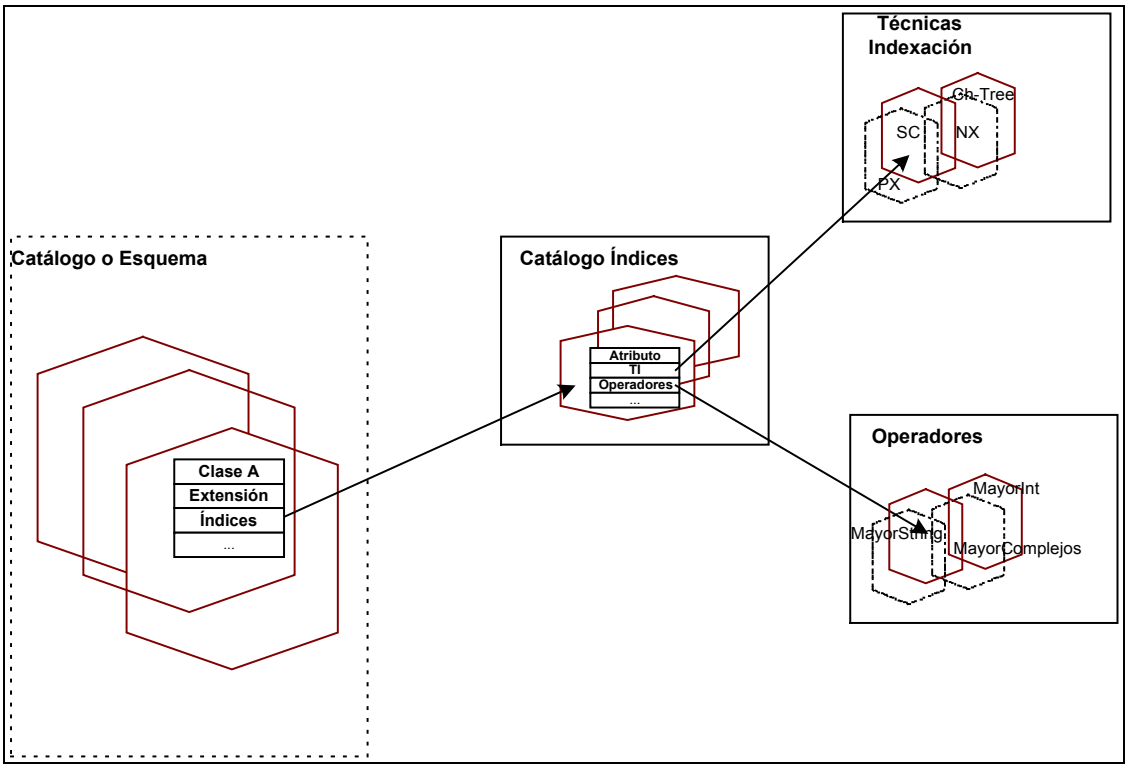


Figura 12.6. Información necesaria en el mecanismo de indexación

Esta información se almacenará para cada índice que sea construido sobre una determinada clase. Luego ha de estar accesible desde el catálogo o esquema de la base de datos descrito en el capítulo anterior.

12.6 Características por defecto del mecanismo

En los apartados anteriores se han expuesto los objetivos y características que debe proporcionar el mecanismo de indexación de BDOviedo3. Pero, cara a la construcción del sistema parece lógico dotar al mecanismo con algunas técnicas de indexación por defecto, así como con algunos operadores para los tipos básicos, también por defecto, que eviten trabajo innecesario por parte del usuario.

12.6.1 Técnicas de indexación a incluir inicialmente en el mecanismo

La elección de dichas técnicas en principio no sería crítica por la capacidad del sistema para incorporar fácilmente otras nuevas, sin embargo, generalmente siempre las opciones que se incluyen por defecto suelen ser las más utilizadas, aunque en muchos casos eso obedece a causas ajenas al rendimiento (como es por ejemplo el desconocimiento del usuario de la posibilidad para cambiarlas).

A la hora de seleccionar las técnicas de indexación con las que dotar inicialmente al mecanismo se han tenido en cuenta las siguientes consideraciones:

- Parece más adecuado incluir **técnicas de indexación estructurales**, ya que la mayoría de los lenguajes de consulta orientados a objetos permiten consultar mediante predicados basados en atributos de objetos. Las más relevantes de dichas técnicas fueron analizadas en el capítulo 11.
- Es conveniente incluir técnicas de indexación que se comporten bien tanto para consultas realizadas sobre **jerarquías de herencia** como sobre **jerarquías de**

agregación. Por eso, inicialmente se consideró incluir la técnica *CH-Tree*, que es una de las que mejor se comporta para soportar consultas basadas en jerarquías de herencia (véase capítulo 11), y la técnica *Path Index*, porque es una de las que mejores resultados presenta para consultas basadas en jerarquías de agregación (véase capítulo 11).

- Para consultas basadas en **una única clase** y sin jerarquías de agregación, la técnica *Single Class* (SC) es una de las que mejor se comporta globalmente.

Por otro lado, sería interesante conocer cuáles son las técnicas de indexación empleadas por los sistemas comerciales o de investigación, ya que eso sería un indicador de la adecuación de la elección aquí realizada. La Tabla 12.1 refleja los resultados obtenidos, y como se puede apreciar las técnicas de indexación más empleadas en bases de datos orientadas a objetos son la SC y la CH-Tree.

SGBDOO[2]	Técnicas de Indexación
Jasmine	SC
ORION	CH-Tree
O2	CH-Tree
GOA++[3]	SC, CH-Tree
GemStone	MultiIndex
ObjectStore	B-Tree, hash
EXODUS[4]	SC, Hash

Tabla 12.1. Técnicas de indexación empleadas por algunos sistemas

Teniendo en cuenta las consideraciones anteriores, BDOviedo3 inicialmente, incorporará las técnicas de indexación estructurales: *SC*, *CH-Tree* y *Path Index*.

12.6.2 Operadores a incluir inicialmente en el mecanismo

Los operadores a incluir inicialmente en el mecanismo deben de estar en consonancia con las técnicas de indexación seleccionadas por defecto. En este caso se han seleccionado tres técnicas basadas en árboles B+, y por tanto los operadores serán los clásicos: mayor, menor, igual, menor o igual y mayor o igual. Las clases para las que se proporciorá el funcionamiento de dichos operadores son las clases básicas del sistema (*String*, *Integer*, ...).

12.7 Requisitos previos para la utilización del mecanismo de indexación

En los apartados anteriores se han examinado las características del mecanismo de indexación con el que se desea dotar al SGBDOO BDOviedo3. Pero, para un

[2] Excepto EXODUS que es un generador de bases de datos no un SGBDOO

[3] Proporcionadas inicialmente por el gestor ya que permite la incorporación de nuevas técnicas basadas en árboles B^+.

[4] Proporcionados inicialmente por EXODUS, ya que permite la incorporación de nuevas técnicas.

aprovechamiento completo de dicho mecanismo es necesario contar con un lenguaje de control o una herramienta que permita especificar al menos la siguiente información (ya que el lenguaje de definición propuesto por el estándar ODMG no lo soporta):

- La clase del atributo a indexar
- El tipo de técnica de indexación a emplear para cada índice definido
- La definición de los operadores a emplear para cada clase e índice
- La asociación de los índices con sus operadores correspondientes para cada clase

Con el fin de evitar un trabajo adicional al programador, cuando las indexaciones a realizar sean sobre datos de tipo simple, es posible considerar una técnica de indexación por defecto, de forma que el trabajo requerido al usuario sería similar al de cualquier otro SGBDOO no extensible.

12.8 Aplicaciones del mecanismo de indexación

Hasta ahora se ha estado examinando el mecanismo de indexación desde el punto de vista clásico de un SGBDOO. Sin embargo, la concepción del mismo como un conjunto de objetos similares al resto de los objetos del sistema integral (incluidos por ejemplo los del sistema operativo), hace que este mecanismo pueda ser empleado por cualquier otro elemento del sistema integral, no únicamente por los objetos que constituyen el motor de la base de datos.

De la misma manera, no hay nada que impida que este mecanismo de indexación, se convierta en un mecanismo para indexar todas las instancias que se crean en el sistema integral, no únicamente aquellas que han sido creadas desde el entorno de la base de datos. Este mecanismo puede convertirse así en un mecanismo de indexación para todo el sistema integral, pudiendo, por ejemplo, crearse un índice B+ para indexar todos los objetos registrados en el sistema por su identificador. Cada vez que un objeto es creado, reflectivamente un objeto lo inserta en la estructura de indexación, y cuando se intenta acceder a un objeto, de nuevo también de forma reflectiva un objeto es el encargado de localizarlo en el índice. De la misma manera, podría por ejemplo emplearse para indexar las extensiones de las clases.

12.9 Resumen de las características proporcionadas por el mecanismo de indexación

Las características del mecanismo de indexación aquí presentado son expuestas a continuación de forma resumida:

- **Extensibilidad.** Es la principal característica de dicho mecanismo. Su diseño estructurado en torno a un marco orientado a objetos facilita la adaptación por parte del usuario a sus necesidades. Permite concretamente:
  - La incorporación de nuevas técnicas
  - La incorporación y definición del comportamiento de los operadores
  - La aplicación a cualquier tipo de dato, no únicamente datos simples
- **Configurabilidad.** El funcionamiento del sistema integral subyacente al mecanismo de indexación, facilita la configurabilidad del mismo, permitiendo que únicamente sean cargadas en memoria aquellas técnicas que se necesitan.

- **Integración con el sistema integral.** La integración del mecanismo de indexación con el sistema integral evita, por ejemplo, que el programador tenga que hacer la traducción entre las estructuras de almacenamiento empleadas en la técnica de indexación y las empleadas por el sistema de almacenamiento.
- **Plataforma de experimentación.** El diseño realizado del mecanismo, y las características anteriores lo convierten en una buena plataforma de experimentación, al permitir de manera sencilla la incorporación de nuevas técnicas al sistema.
- **Indexación para el almacenamiento global.** El mecanismo de indexación podría considerarse con un mecanismo de indexación para todo el sistema de almacenamiento global.

Capítulo 13 Elección del lenguaje para el SGBDOO

Todo SGBDOO debe estar dotado de uno o varios lenguajes que permitan su manipulación y consulta. En este capítulo se describen las características de un lenguaje para una base de datos orientada a objetos, y se plantean las diferentes posibilidades a la hora de seleccionar un lenguaje para BDOviedo3. Entre esas posibilidades se encuentra la de emplear los lenguajes propuestos por el estándar, que es por la que se ha optado finalmente.

13.1 Lenguajes de bases de datos desde el punto de vista tradicional

La visión tradicional de las bases de datos es que los datos persistentes son externos e independientes de las aplicaciones. Es por tanto tarea de los programadores distinguir cuando están trabajando con datos de la base de datos y cuando lo están con datos de la aplicación. La manipulación de la base de datos se realiza con un lenguaje proporcionado por el SGBD y que se emplea solo para el acceso a la base de datos.

La principal justificación a esta situación se deriva del hecho de que la aplicación siempre puede acceder a la base de datos independiente del lenguaje de programación en que esté codificada, ya que la interfaz con la base de datos es siempre la misma (SQL, en el caso de bases de datos relacionales). Sin embargo, esta separación entre el lenguaje de acceso a la base de datos y el de la aplicación provoca un serio problema de desadaptación de impedancias [Moo96].

13.1.1 Desadaptación de Impedancias

Los SGBDR proporcionan al usuario un lenguaje de programación y un lenguaje de consulta[1] separados, con diferentes sistemas de tipos y entornos de ejecución. Tradicionalmente, los SGBDR incluyen un lenguaje de consulta declarativo que permite el acceso y manipulación de la base de datos pero que no incluye construcciones de control, variables u otras características de programación. Por otro lado, los lenguajes de programación se emplean para construir la aplicación pero no permiten el acceso a la base de datos. Ambos entornos tienen sistemas de tipos diferentes. Para permitir el acceso a la base de datos desde un lenguaje de programación, se incrustan los enunciados del lenguaje de consulta en el lenguaje de programación con el fin de copiar los datos entre los dos entornos.

[1] Se emplea el término lenguaje de consulta por comodidad pero realmente incluye además del propio lenguaje de consulta, los lenguajes de definición y de manipulación de la base de datos

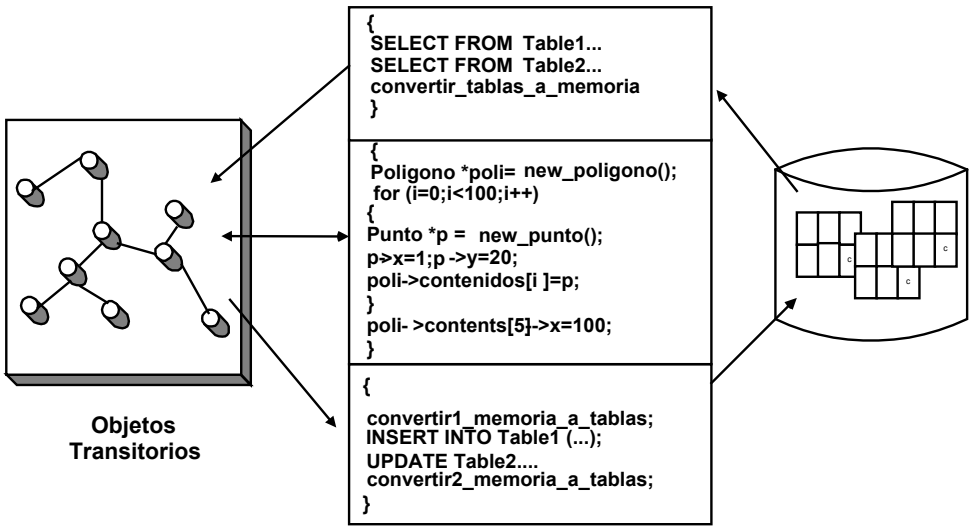


Figura 13.1. Ejemplo de traducción de datos entre los dos entornos

La existencia de dos entornos se traduce en un problema para el programador: tiene que traducir los datos entre las representaciones y por tanto conocer los dos lenguajes. Todo esto repercute en una disminución de la productividad del programador.

13.2 Características de un lenguaje para una base de datos orientada a objetos

Para aliviar el problema de la desadaptación de impedancias los SGBDOO preconizan el empleo de un lenguaje de programación de base de datos con un único entorno de ejecución, lenguaje procedural y sistema de tipos. La idea se basa en extender la sintaxis de un lenguaje de programación con la sintaxis de un lenguaje de consulta. Esto inicialmente puede parecer similar a incrustar el lenguaje de consulta en el lenguaje de programación, pero difiere porque el lenguaje de consulta tiene la misma estructura de tipos que el lenguaje de programación, es ejecutado en el mismo proceso y no contiene su propia construcción procedural.

A continuación se relacionan algunas de las características más relevantes de un lenguaje de programación de bases de datos para un SGBDOO [Loo95].

13.2.1 Lenguaje de programación orientado a objetos como lenguaje de definición y manipulación

La mayoría de los SGBDOO han adoptado la idea de emplear un lenguaje de programación orientado a objetos como lenguaje de definición y de manipulación del esquema, ya que el usuario de un SGBDOO espera tratar los objetos transitorios y persistentes con una sintaxis simple: la del lenguaje de programación. No es necesario entonces que el programador aprenda un lenguaje análogo a SQL para, por ejemplo, definir el esquema de la base de datos.

13.2.2 Necesidad de un lenguaje de consulta

En un SGBDR los lenguajes de consulta son la única forma de acceder a los datos. Un SGBDOO, por el contrario, proporciona por defecto una forma de acceso *navegacional* basada en los identificadores de los objetos y en las jerarquías de agregación. Sin embargo, esta forma de acceso no implica que no sea conveniente un acceso *declarativo* a los objetos, ya que hay muchas aplicaciones que necesitan acceder

a los objetos basándose en valores, sin necesidad de recorrer explícitamente las jerarquías de agregación. Por supuesto, los dos tipos de acceso se pueden emplear de manera complementaria. Una consulta convencional puede ser útil para seleccionar un conjunto de objetos que luego serán accedidos y procesados por las aplicaciones usando las capacidades navegacionales.

13.2.2.1 Alternativas para proporcionar consultas desde un lenguaje de programación

El hecho de que un SGBDOO proporcione un acceso declarativo a la base de datos se considera importante, y con el fin de eliminar la desadaptación de impedancias, es obvio que el lenguaje de consulta ha de estar integrado con el lenguaje de programación. Partiendo de esta situación, se han planteado diferentes posibilidades:

- *Consultas expresadas en un lenguaje de programación orientado a objetos estándar.* Al no ampliar la sintaxis del lenguaje de programación se puede emplear cualquier compilador/intérprete que soporte dicho lenguaje sin necesidad de preprocesadores. La mayoría de estos sistemas heredan de una clase base virtual que dispone de un método *select*, que contiene un argumento cuya cadena será la consulta.
- *Consultas expresadas en un lenguaje de programación orientado a objetos extendido.* La integración también puede ser juzgada como la familiaridad del lenguaje resultante para el programador. Esto se traduce en proporcionar al lenguaje extensiones para acomodar la semántica de las consultas, prestando especial atención a la consistencia con el estilo del lenguaje de programación orientado a objetos elegido. Requiere el empleo de un preprocesador que devolverá código que el compilador pueda entender.

13.2.2.2 Resultados de las consultas

Los SGBDOO también difieren en cuanto a los resultados obtenidos como consecuencia de la ejecución de consultas. A diferencia del modelo relacional, en el que el resultado de una consulta es siempre una relación, en el caso de un SGBDOO pueden distinguirse claramente dos situaciones:

- *Resultados Colección* (GemStone, ObjectStore, ORION, ONTOS). Las consultas en estos sistemas no operan sobre la base de datos completa; las consultas son empleadas para seleccionar objetos que satisfacen ciertas restricciones en una colección de objetos particular. Estas colecciones pueden recuperarse desde atributos de objetos o bien pueden ser la extensión de una clase.
- *Resultados Abiertos* (O_2, ODMG). Los resultados de las consultas en estos sistemas pueden ser cualquier tipo de dato (valor literal, un objeto, una lista, un conjunto, etc.). Un lenguaje de consulta con resultados abiertos se distingue por su simetría respecto a los tipos de datos.

13.2.3 Requerimientos del lenguaje de consulta

A continuación se enumeran los requerimientos a cumplir por un lenguaje de consulta para una base de datos orientada a objetos. Estos requerimientos son derivados de su condición de ser un lenguaje de consulta de base de datos, y de su condición de ser un lenguaje que soporta un modelo de objetos.

13.2.3.1 Derivados de bases de datos

A un lenguaje de consulta que soporte un modelo de orientación a objetos se le hacen una serie de exigencias heredadas de un lenguaje ya consagrado como es SQL:

- Posibilidad de incluir la cuantificación existencial o universal, así como operaciones sobre colecciones de objetos.
- Funciones booleanas que permiten que criterios de búsqueda simples sean combinados en criterios de búsqueda compuestos.
- Funciones agregadas que pueden ser incorporadas en los criterios de búsqueda, y en los objetivos.
- Funciones de agrupamiento y ordenación, empleadas para agrupar u ordenar objetos de acuerdo a alguna característica común.
- Consultas anidadas, o subconsultas, que permiten que el resultado de una consulta sea empleado como el ámbito para el criterio de búsqueda de otra consulta.

13.2.3.2 Derivados del empleo de un modelo de objetos

Idealmente, el lenguaje de consulta para un SGBDOO se espera que sea consistente con el modelo de objetos, y esto supone tres consideraciones básicas:

a) Una consulta debe poder especificar cualquier característica de la clase (atributos, relaciones y operaciones) tanto en el predicado como en los objetivos. Ha de poder especificar características del modelo tales como:
   - **Jerarquía de agregación.** El concepto de jerarquía de agregación implica ciertas extensiones al lenguaje. Las condiciones que aparecen en una consulta en forma de combinaciones de predicados se imponen sobre los atributos anidados en la jerarquía del objeto tratado, por lo que incluir expresiones de camino dentro del lenguaje hace que la consulta se pueda formular más fácilmente.
   - **Jerarquía de herencia.** Relacionado con la herencia, es conveniente destacar que la jerarquía de herencia nos permite teóricamente consultar una clase, o bien una clase y todas sus subclases. En la mayoría de los lenguajes existentes ambas opciones son posibles.
   - **Métodos.** Los métodos se pueden emplear en las consultas como *atributos derivados* (procedimiento que calcula un valor sobre la base de los valores de los atributos de un objeto o de otros objetos) o como *métodos predicados* (procedimiento que devuelve valores lógicos y se emplea para hacer comparaciones entre objetos).

b) Una consulta debería ser capaz de ejecutar operaciones polimórficas correctamente sobre objetos de clases y subclases.

c) Las consultas deberían abarcar objetos de cualquier tipo, independientemente de si son transitorios o persistentes.

Problemas con la encapsulación

En un lenguaje de programación orientado a objetos la encapsulación proporciona independencia de los datos mediante la implementación de métodos, que permiten que la parte privada de un objeto se cambie sin afectar a los programas que usan esa clase, y

según *The Object-Oriented Database System Manifesto* [ABD+89] una encapsulación real sólo se obtiene cuando las operaciones son visibles y el resto del objeto está oculto.

Por otro lado, el lenguaje de consulta proporciona una independencia lógica y física en una base de datos, pero generalmente las consultas se expresan en términos de predicados sobre los valores de los atributos. Por lo tanto, una aproximación estricta orientada a objetos obliga a que los atributos y las relaciones asociadas con los objetos sean invisibles al usuario; el lenguaje de consulta debe operar solamente sobre métodos. Ante eso se plantean diferentes alternativas:

- Violación del paradigma de orientación a objetos para el lenguaje de consulta, o al menos, para el uso del lenguaje de consulta por el usuario, permitiendo que los atributos y las relaciones sean visibles por medio del lenguaje de consulta como lo serían por medio de un método sobre el objeto. Estos sistemas proporcionan *ocultamiento parcial*: la implementación privada de un objeto está oculta desde los programas pero no desde las consultas.
- Definición de métodos para los atributos que se desea que sean visibles para el lenguaje de consulta (típicas operaciones *getAtributo()*). Esta definición puede ser realizada por los programadores, o bien una alternativa mucho mejor sería que fuese el propio sistema el que proporcionase dichos métodos implementados de forma eficiente (liberando por tanto al usuario de dicha tarea).

13.2.4 Persistencia ortogonal

Las bases de datos desde el punto de vista de un lenguaje de programación deben promover el hecho de que todas las capacidades sean igualmente aplicables a objetos persistentes y transitorios independientemente de su tipo. Promueve una separación completa entre persistencia y tipo: *ortogonalidad.* Esto implica que cada tipo puede tener instancias transitorias y persistentes, y que cualquier comportamiento que se aplique a un tipo es independiente de si sus instancias son transitorias o persistentes.

Aunque la mayoría de los SGBDOO consideran esta ortogonalidad entre persistencia y tipos deseable, ninguno de ellos implementa completamente esta separación.

13.2.4.1 Especificación de la persistencia

A la hora de especificar la persistencia en una aplicación para un SGBDOO existen dos tendencias claramente diferenciadas:

- Seguir la tradición de base de datos, con lo que debería haber una declaración explícita de las clases que pueden tener instancias persistentes, lo que implicaría que la capacidad de persistencia debería ser declarada explícitamente en las declaraciones de clase. Esta opción suele ser implementada mediante una clase de la que heredan todas las clases que sean susceptibles de poseer instancias persistentes y que proporciona el comportamiento persistente. Esta opción es empleada mayoritariamente (véase capítulo 5).
- Seguir la tendencia de los lenguajes de programación, con lo que todas las clases deberían ser capaces de ser persistentes, y no habría necesidad de declarar aquellas clases capaces de ser persistentes. Por ejemplo, el modelo de persistencia del sistema integral que nos ocupa, permitiría perfectamente esta posibilidad.

13.2.4.2 Consultas sobre objetos persistentes o transitorios

La ortogonalidad de tipo exigiría que las consultas se pudiesen aplicar tanto sobre objetos persistentes como transitorios, frente a la postura tradicional en la que únicamente se aplican sobre objetos persistentes. La mayoría de los SGBDOO, sin embargo, únicamente proporcionan consultas sobre objetos persistentes. A lo máximo que suelen llegar algunos sistemas es a permitir considerar en las consultas objetos que están siendo tratados (creados, modificados) en la transacción en curso (ej. ObjectDRIVER).

13.2.4.3 Transacciones sobre objetos persistentes o transitorios

La idea de la ortogonalidad entre tipo y persistencia también afecta a la gestión de las transacciones. Una cuestión a considerar es si las transacciones deberían afectar también a datos transitorios. Actualmente los SGBDOO no garantizan esta consistencia para datos transitorios acudiendo al hecho de que supondría una sobrecarga bastante elevada para el sistema y por tanto una disminución en su rendimiento. Una posibilidad podría ser que sea el propio usuario el que decida si desea o no este comportamiento por parte del sistema.

13.3 Posibilidades para BDOviedo3

En los apartados anteriores se han desglosado brevemente las características que debe tener un lenguaje de base de datos orientado a objetos. Una vez conocidas dichas características es necesario seleccionar un lenguaje para el SGBDOO BDOviedo3. Cara a realizar dicha selección se presentan tres opciones claramente diferenciadas.

13.3.1 Diseñar y construir un nuevo lenguaje de base de datos

Un lenguaje de base de datos ideal debería proporcionar las características mencionadas en el apartado anterior, sin embargo, en la mayoría de los lenguajes existentes algunas de esas características no son soportadas, o al menos en su totalidad (ej. ortogonalidad tipo-persistencia).

Es por eso, que una primera opción podría ser diseñar y construir un nuevo lenguaje en el que se traten de recoger los requerimientos especificados anteriormente, y que soporte directamente las características del sistema integral sobre el que subyace el SGBDOO BDOviedo3.

Esta opción, cuenta en principio con un inconveniente, y es la reticencia tanto de los usuarios como de las organizaciones de la comunidad de base de datos a aceptar un nuevo lenguaje de programación para bases de datos. Sin embargo, es una opción muy a tener en cuenta dentro del marco de investigación en el que se sitúa este proyecto, y además, es una posibilidad que no se descarta incorporar con posterioridad al SGBDOO. Actualmente, pueden encontrarse trabajos en la literatura sobre este tema, incluso en sentido inverso, es decir, extender un lenguaje de consulta para convertirlo en un lenguaje de programación [Sub96].

13.3.2 Adopción de un lenguaje de base de datos ya existente

La segunda posibilidad cara a dotar a BDOviedo3 de un lenguaje de base de datos puede ser la de tomar un lenguaje de programación de base de datos[2] de los que ya

[2] Al referirnos a un lenguaje de programación de base de datos se está considerando también el lenguaje de consulta

existen (O_2, GemStone, etc.). Pero ante esta situación, se plantea la cuestión de cual de ellos elegir.

Inicialmente, cada constructor de base de datos adaptaba el lenguaje de programación orientado a objetos tomado como base a conveniencia. Es decir, optaban por soluciones 'a medida' para sus sistemas. Esta situación creó un problema. No había portabilidad de datos ni de aplicaciones entre los diferentes SGBDOO existentes. Y de hecho, una de las mayores trabas de los SGBDOO fue la carencia de un estándar que les proporcionase la misma potencia que proporcionaba el SQL a las bases de datos relacionales.

Entonces, el hecho de adoptar uno de estos lenguajes ya existentes o simplemente tomar un lenguaje de programación y adaptarlo particularmente a nuestro sistema sigue siendo una solución 'a medida', y además haría que nuestro sistema que preconiza portabilidad, flexibilidad, e interoperabilidad se viese de alguna manera frenado por la imposición de un lenguaje particular.

13.3.3 Adopción de los lenguajes propuestos por el estándar ODMG

Representa una tercera opción cara a dotar a BDOviedo3 de un lenguaje para su manipulación. El estándar ODMG[3] (*Object Database Management Group*) surge inicialmente para solucionar precisamente los problemas anteriormente mencionados de falta de portabilidad entre sistemas.

ODMG, define un SGBDOO como un SGBD que integra las capacidades de bases de datos con las capacidades de los lenguajes de programación orientados a objetos, de forma que los objetos de la base de datos aparezcan como objetos del lenguaje de programación, en uno o más lenguajes de programación existentes. El SGBDOO extiende el lenguaje con persistencia, control de concurrencia, recuperación de datos, consultas asociativas y otras capacidades de bases de datos.

13.3.3.1 Componentes Principales

Los componentes funcionales que integran este estándar son un modelo de objetos, un lenguaje de especificación de objetos, un lenguaje de consulta y las ligaduras (*bindings)* para C++, Java y Smalltalk.

A continuación se hace un breve resumen de cada uno de estos componentes. No obstante, para más información sobre el estándar véase el apéndice A.

Modelo de Objetos

El modelo de objetos es el centro del estándar ODMG, y está basado en el modelo de objetos de OMG (*OMG Common Object Model*), que fue ideado como un denominador común para gestores de peticiones de objetos (*Object Request Brokers)*, sistemas de almacenamiento de objetos, lenguajes de programación y otras aplicaciones, y al que se le añadieron otros componentes como relaciones y transacciones para soportar las necesidades de almacenamiento de objetos.

Las características centrales del modelo de objetos son:

- Atributos y relaciones como propiedades de los objetos
- Operaciones de los objetos y excepciones

[3] Aunque recientemente ha cambiado su nombre por *Object Data Management Group*

- Herencia múltiple
- Extensiones y claves
- Denominación de objetos, tiempo de vida e identidad
- Tipos atómicos, estructurados y literales colección
- Clases colección *list*, *set*, *bag* y *array*
- Control de concurrencia y bloqueo de objetos
- Operaciones de base de datos

Lenguaje de Especificación de Objetos

Los lenguajes para la especificación de objetos son dos: el **lenguaje de definición de objetos** (*ODL, Object Definition Language*), que está basado en la sintaxis del IDL de OMG, y el lenguaje que especifica el **formato para el intercambio de objetos** (*OIF, Object Interchange Format*).

El ODL permite que un esquema generado con este lenguaje sea independiente del lenguaje de programación y de la base de datos, permitiendo que las aplicaciones sean movidas entre bases de datos compatibles.

Lenguaje de Consulta de Objetos

El lenguaje de consulta es OQL (*Object Query Language*), que es un lenguaje de consulta declarativo estilo SQL (superconjunto de la sintaxis SELECT de SQL-92) que permite la consulta de bases de datos de objetos, incluyendo primitivas de alto nivel para tratar con conjuntos de objetos, listas, etc. Soporta objetos complejos, expresiones de camino, invocación de operaciones y herencia. Las consultas OQL pueden invocar operaciones en el lenguaje empleado para el *binding,* e inversamente OQL puede ser incrustado en dicho lenguaje. OQL mantiene la integridad de los objetos empleando las operaciones definidas por el objeto, en vez de sus propios operadores de actualización.

Bindings

Los *bindings* del estándar ODMG para Java, C++ y Smalltalk definen el lenguaje de manipulación de objetos (*OML, Object Manipulation Language)* que extiende el lenguaje de programación para soportar objetos persistentes. Además, incluyen soporte para OQL, navegación y transacciones. De todos los *bindings* es especialmente relevante el *binding* Java.

El *binding* Java añade clases y otras construcciones al entorno Java para soportar el modelo de objetos de ODMG, incluyendo colecciones, transacciones y bases de datos, sin alterar la semántica del lenguaje. Las instancias de las clases existentes pueden hacerse persistentes sin cambiar el código fuente. La persistencia es por alcance.

13.4 Lenguaje seleccionado: propuesto por el estándar ODMG

De las tres posibilidades planteadas, inicialmente se ha seleccionado la tercera, es decir, dotar a BDOviedo3 con los lenguajes propuestos por el estándar ODMG 2.0 [CBB+97], y en su *binding* para Java.

13.4.1 Motivaciones para adoptar el estándar en su *binding* para Java

Las principales razones para adoptar el estándar son mencionadas a continuación, pero la mayoría de ellas obedecen a los propios objetivos perseguidos por dicho estándar [CAD+94,CBB+97].

13.4.1.1 Cumple la mayoría de los requisitos

En los apartados anteriores se han especificado las características principales para un lenguaje de una base de datos orientada a objetos. Los lenguajes propuestos por el estándar cumplen la mayoría de ellos:

a) *El estándar preconiza el empleo de un lenguaje de programación orientado a objetos como lenguaje de definición y manipulación de la base de datos.* Para la definición del esquema proporciona un lenguaje específico, ODL, pero este lenguaje es introducido fácilmente en la sintaxis de los lenguajes de programación orientados a objetos empleados para los diferentes *bindings* (en este caso concreto Java).

b) *Incorpora un lenguaje de consulta: OQL*, que permite un acceso declarativo a los objetos. Además, este lenguaje soporta los principales requerimientos especificados en el apartado 13.2.3, como se puede apreciar en la Tabla 13.1, siendo los resultados de las consultas abiertos, es decir, el resultado puede ser cualquier estructura de datos que pertenezca al modelo de objetos.

Requerimientos del lenguaje de consulta	**Descripción (en OQL)**
Base de Datos	
Cuantificación existencial...	*All, any,...*
Criterios de búsqueda compuestos	Si
Funciones agregadas	*Avg, max,min,...*
Funciones agrupamiento y ordenación	*Order by, group by,...*
Consultas anidadas	Si
Modelo de Objetos	
Jerarquía de agregación	Si *Select e.Nombre From Empleado e* *Where e.direccion.calle='Uria'*
Jerarquía de herencia	Si. Por defecto se incluyen todas las instancias de las subclases de la clase consultada. *Select e.Nombre* *From (Empleado except Funcionario) e*
Métodos	Si
Polimorfismo	Si

Tabla 13.1. Requerimientos cumplidos por el lenguaje de consulta OQL

13.4.1.2 Portabilidad de las aplicaciones

El empleo de un estándar como este pretende garantizar la portabilidad para la aplicación completa (y no únicamente para la porción de semántica de los enunciados SQL incrustados), del código fuente. Esto no quiere decir que todos los SGBDOO sean iguales; se podrán diferenciar unos productos de otros en cuanto a rendimiento, lenguajes soportados, posibilidad de versionado y gestión de la configuración, herramientas para la construcción de aplicaciones, disponibilidad de plataforma, etc.

La idea al seleccionar el estándar para BDOviedo3, es facilitar la portabilidad de las aplicaciones desde y hacia otros SGBDOO, y en definitiva hacia otros sistemas de gestión de objetos.

13.4.1.3 *Binding* Java: el más utilizado

Con la explosión de Internet se demandan nuevas aplicaciones dinámicas, para las que los programadores emplean arquitecturas multi-hilo y Java principalmente. En consecuencia, el almacenamiento de objetos, especialmente objetos Java, ha llegado a ser un problema crucial.

ODMG ante este problema se transforma en un estándar para el almacenamiento de objetos persistentes [Bar98], sin tener en cuenta el mecanismo final de almacenamiento. De esta forma ODMG amplia su carácter permitiendo cualquier mecanismo de almacenamiento y cambia su nombre por *Object Data Management Group*.

De esta forma el *binding* Java es soportado, como se puede observar en la Tabla 13.2, no solo sobre bases de datos orientadas a objetos, sino también sobre bases de datos y envoltorios (*wrappers*) objeto-relacionales.

Compañía	Producto	Descripción
Computer Associates	Jasmine	SGBDOO
Object Design	ObjectStore	SGBDOO
Object Design	PSE para Java	Gestor Objetos
Versant	Versant ODBMS	SGBDOO
POET	POET Object Server	SGBDOO
Objectivity	Objectivity/DB	SGBDOO
Ardent Software	O_2 System	SGBDOO
CERMICS	ObjectDRIVER	Wrapper

Tabla 13.2. Algunos sistemas que soportan el *binding Java*

El estándar ahora permite otras bases de datos además de las orientadas a objetos, aunque sigue considerando que la productividad sería mayor si el producto dónde se almacenan los objetos fuese orientado a objetos.

13.4.2 Deficiencias en el estándar

El estándar propuesto por ODMG ha acarreado una serie de contribuciones técnicas que han sido expuestas en los apartados anteriores, y que pueden verse con más detalle en el Apéndice A, que son las que han ocasionado su elección como lenguaje para BDOviedo3. Sin embargo, también posee una serie de deficiencias o inconsistencias [Ala97], algunas de las cuales son comentadas a continuación.

13.4.2.1 Deficiencias en el Modelo de Objetos

Primitivas de bloqueo en el modelo del usuario

Aunque es una cuestión criticada, algunos autores [Ala97] afirman que a diferencia de la posibilidad de que un objeto persista, que se supone que es una elección del usuario, el bloqueo es simplemente una técnica de implementación y no debería como tal ser colocada en el modelo del usuario (*interface Object*), al igual que no lo son las técnicas para la implementación de la persistencia o la especificación de la indexación para la extensión de una clase.

Tipos Parametrizados

El modelo de objetos de ODMG no ofrece soporte para clases paramétricas, y la razón más probable para esto es que el estándar define *bindings* además de para C++ (que si las permite) para Java y Smalltalk que no las permiten.

13.4.2.2 Deficiencias en los Lenguajes

Para los defensores del modelo relacional los lenguajes de bases de datos aquí planteados presentan una serie de deficiencias si se comparan a las prestaciones que ofrece un lenguaje de base de datos relacional como SQL, y algunas de ellas quizás deban ser consideradas por el estándar.

Lenguaje de Manipulación

El lenguaje de manipulación OML no permite insertar, actualizar o borrar basándose en unas condiciones de búsqueda, es decir, no ofrece la posibilidad de insertar más de un objeto basándose en el cumplimiento de unas condiciones de búsqueda, mientras que SQL si lo hace. Desde el punto de vista del estándar, estas operaciones explícitas no son incluidas para mantener la semántica del modelo de objetos, obligando así a acudir a los métodos definidos en los objetos para realizar dichas operaciones.

Lenguaje de Definición

El lenguaje de definición ODL no proporciona soporte para vistas, que en el modelo relacional suponen una unidad de autorización de acceso. En las bases de datos orientadas a objetos las vistas implican mucho más que en el modelo relacional ya que pueden formar parte de una jerarquía de herencia, pueden emplearse como el dominio de un atributo, pueden tener métodos al igual que atributos, e incluso suponen una forma de integrar bases de datos heterogéneas y dar soporte para la evolución de esquemas [GBC+97]. Sin embargo, el modelo ODMG no se pronuncia sobre las mismas.

El ODL tampoco proporciona facilidades para llevar a cabo cambios dinámicos en el esquema de la base de datos que impliquen algo más que añadir un nuevo subtipo a un tipo ya existente. No proporciona facilidades para añadir o borrar un método o un atributo de un tipo, añadir un nuevo supertipo a un tipo existente o bien eliminarlo, etc.

Lenguaje de Control

El lenguaje de control es el que permite gestionar las transacciones (commit, rollback), controlar el acceso a bases de datos desde múltiples usuarios (grant, revoke), gestionar los recursos como los índices (create index, drop index), reforzar la integridad de la base de datos basándose en condiciones especificadas por el usuario (triggers), etc. Además, en la autorización hay que tener en cuenta el modelo de orientación a objetos, contemplando la autorización por ejemplo para ejecutar métodos, para acceder no sólo a un tipo sino a una jerarquía de tipos, etc. En ODMG el lenguaje de control forma parte del lenguaje de manipulación, pero no incluye la mayoría de las funcionalidades anteriores.

En ODMG se considera un modelo de transacciones anidado para la gestión de transacciones (según Kim debería ser un modelo de transacciones no anidado por defecto), pero no se consideran detalles como la autorización para ejecutar métodos. También se indica explícitamente que en un mundo ideal no sería necesario especificar si una colección debe ser indexada o no, con lo que no se considera la indicación explícita de la indexación en el modelo del usuario.

Este estándar tampoco se pronuncia sobre los disparadores que, aunque la gran mayoría de la comunidad de las bases de datos orientadas a objetos no los considera entre otras cosas porque suponen una violación de la encapsulación, también tienen sus defensores [MS97].

13.4.3 Extensiones propuestas al estándar

Para subsanar algunos de los problemas anteriores, y como muestra de que el estándar está vivo, independientemente de la velocidad a la que los fabricantes lo estén incorporando a sus productos, constantemente se están proponiendo ampliaciones o extensiones al mismo por parte de la comunidad científica. Así, se proponen desde extensiones al modelo de objetos para soportar relaciones ternarias [MVS99] y objetos compuestos e integridad referencial [BG98], hasta extensiones para gestionar datos temporales [BFG+98]. También hay investigaciones recientes para proporcionar una extensión deductiva para bases de datos que son compatibles con el estándar (DOQL) [FP99].

13.5 Adaptación del modelo de objetos de ODMG al sistema integral

Ciertamente a la hora de seleccionar un lenguaje para el SGBDOO BDOviedo3, la solución del estándar es una posibilidad aceptable, pero hay que tener en cuenta que el modelo de objetos que soporta el sistema integral sobre el que es construido BDOviedo3 tiene ligeras diferencias con relación al modelo de objetos del estándar ODMG.

Las diferencias fundamentales entre ambos modelos provienen principalmente de los tipos de objetos soportados, ya que ODMG distingue entre objetos y literales (no tienen identificador), mientras que en el sistema integral no existe el concepto de literal; todos son objetos y todos tienen identificador. También hay diferencias en cuanto a las características de bases de datos que soporta el modelo de objetos de ODMG, como es por ejemplo el concepto de base de datos (Database). Este concepto como tal no forma parte del modelo de objetos del sistema integral.

Existen también otra serie de diferencias a considerar en función del modelo del lenguaje elegido para el *binding*, no obstante, el empleo por parte del sistema integral de un modelo de objetos que incorpora los conceptos de orientación a objetos más comúnmente aceptados (véase capítulo 7), hace este proceso de adaptación sencillo y en absoluto traumático.

Capítulo 14 Requisitos de la interfaz visual para el SGBDOO

En este capítulo se describen las características deseables para la interfaz del SGBDOO BDOviedo3, así como la funcionalidad básica que dicha interfaz debe proporcionar. Es decir, aunque como se ha comentado anteriormente las funcionalidades de un gestor de base de datos son muchas y la interfaz debe ser diseñada acorde a ello, existen unas funcionalidades que se consideran básicas para la interfaz y que son las que se exponen a continuación.

14.1 Introducción

La interfaz de usuario para un SGBDOO ha de permitir de forma sencilla e intuitiva la realización de cada una de las funcionalidades proporcionadas por dicho sistema. Las herramientas que constituyen dicha interfaz, tal y como se comentó en el capítulo 3, pueden clasificarse en dos grupos: **de administración,** que ayudan a la administración tanto de la base de datos como de los usuarios, y **de desarrollo**. Este capítulo se centrará únicamente en las herramientas para el desarrollo.

14.2 Características deseables para la interfaz de un SGBDOO

Como ya se ha mencionado en el capítulo 5 existen una gran cantidad de bases de datos y gestores orientados a objetos, pero no existe uniformidad en cuanto a las características que presentan sus interfaces. A continuación se enumeran una serie de características que se consideran esenciales para la interfaz de usuario de cualquier SGBDOO [Lop94] y en especial para el SGBDOO objeto de esta tesis [HMC+00]:

- **Transparencia y realimentación.** La interfaz para el SGBDOO ha de ser transparente para permitir al usuario saber en todo momento lo que ocurre en el sistema. Será para ello necesario dotar al sistema de un mecanismo de realimentación adecuado.
- **Concisión y calidad en la representación.** Una interfaz visual debe ser concisa en dos aspectos: en la presentación de opciones y de información al usuario, y en la cantidad de datos de entrada necesarios para que el usuario pueda expresar sus necesidades.
- **Adaptabilidad y tutorialidad.** La interfaz ha de poder adaptarse a las preferencias del usuario, y éste tiene que poder preguntar al sistema sobre el contexto en que se encuentra.
- **Coherencia e integridad.** Para conseguir coherencia la interfaz debe mostrar para idénticas acciones idénticos resultados en cualquier contexto. La integridad será referida sobre todo a los mecanismos de confirmación y a las rutinas de manejo de errores. Éstos son necesarios para preservar la integridad de los datos que se manejan mediante la interfaz.

- **Completitud funcional.** La interfaz debe proporcionar el acceso a todas las funciones disponibles en el SGBD.
- **Generación automática de vistas.** El sistema ha de permitir la representación de los objetos independientemente de la aplicación que los haya generado.
- **Soporte de diferentes niveles de abstracción.** La interfaz tiene que permitir que el usuario elija el nivel de detalle deseado en la visualización de la información.
- **Acceso a los datos a través del SGBD.** La interfaz de usuario debe ser capaz de acceder a los datos almacenados en el SGBD. Un módulo de la interfaz tiene que ser el responsable del envío y recepción de la información del SGBD. Está claro que este módulo depende del SGBD subyacente. Sin embargo, si la implementación de estas dependencias está bien diseñada, la interfaz puede ser transportada a otro SGBD, con ajustes mínimos realizados en un solo módulo
- **Independencia en las acciones.** La interfaz de usuario debe garantizar que cualquier acción realizada por un usuario produzca un resultado completo y predecible. A cada acción debe corresponder una respuesta o señal. El conjunto de funciones disponibles en cada contexto es siempre el mismo, y la semántica de las acciones es coherente en cualquier situación.
- **Integración y representación de bases de datos.** Las interfaces de usuario tienen que permitir la visualización y manipulación simultánea de los esquemas y los datos de diferentes bases de datos.

14.3 Funcionalidad básica de la interfaz

La completitud funcional mencionada en el apartado anterior implica que la interfaz permita el acceso a toda la funcionalidad del SGBD. No obstante, en este apartado nos centraremos en las funcionalidades consideradas básicas (o de obligada incorporación) desde el punto de vista de uso del SGBDOO.

14.3.1 Generador/Visualizador del esquema de la base de datos

La definición de las clases de las que consta el esquema de una base de datos clásicamente se realiza utilizando un programa de edición de texto, normalmente sin formato, en el que el diseñador - programador escribe la especificación de una clase en un cierto lenguaje de programación, en este caso el *binding* Java de ODMG.

Como se puede apreciar en la tabla 14.1 tanto ObjectStore, como Poet como Jasmine permiten la definición textual del esquema. Sin embargo, este entorno se considera muy poco amigable. Una solución a este problema pasa por la exigencia al sistema de las siguientes propiedades.

14.3.1.1 Editor dirigido por sintaxis

El entorno ha de permitir escribir - editar el código de la definición de las clases que componen la base de datos, y comprobar la corrección de este código. Para ello se considera muy apropiado un editor de texto dirigido por sintaxis, en el que además de la utilización de los colores para distinguir unos elementos sintácticos de otros, hay otra función valorable consistente en mostrar un pequeño cuadro que acompaña al cursor según se escribe el nombre de un objeto o una clase. En este cuadro se muestran los métodos disponibles ordenados alfabéticamente, permitiendo la selección de uno de

ellos, y evitando así errores de tipografía. Además de los nombres de los identificadores de los métodos, se pueden mostrar automáticamente los tipos de los parámetros de la invocación. Evidentemente, esta ayuda se proporcionará no únicamente para las clases predefinidas en el lenguaje, si no para cualquier clase de usuario definida en el sistema.

14.3.1.2 Definición visual de las clases

Es necesaria la existencia de un módulo de definición de las clases totalmente visual alternativo al editor de texto, y con el que se pudiese conmutar, de forma que una clase descrita visualmente tuviese su descripción textual en un correcto lenguaje (en este caso concreto sería el *binding* Java de ODMG), y viceversa.

Algunas herramientas permiten construir gráficamente las clases, especificando sus atributos, las relaciones entre ellas, ya sean jerárquicas o de composición y la enunciación de los métodos con sus parámetros. En el caso de ObjectStore Database Designer [Obj00c] proporciona además dos vistas del esquema de la base de datos (árbol jerárquico de clases y diagrama de clases). Sin embargo, la aplicación es en realidad una herramienta de modelado de clases, pero no ofrece ninguna posibilidad para traducir el modelo a las clases que utilizará el SGBDOO, ni siquiera para escribir los métodos completos en el lenguaje elegido, sino que esa es una labor que debe hacerse desde el tradicional editor de texto.

En la tabla siguiente se pueden observar a modo de resumen las posibilidades ofrecidas por tres de los entornos comerciales más empleados en este momento.

	Poet Developer	**ObjectStore Database Designer**	**JADE**
Definición Textual	Si	Si	Si
Definición Visual	No	• Herencia • Atributos • Relaciones Unidirecc. • Métodos (Cabeceras)	• Herencia • Atributos • Relaciones Unidirecc. • Métodos (Completos)
Generación ODL-Java	ODL Java/C++	No	No
Visualización	• Árbol	• Árbol • Diagrama de Clases	• Árbol

Tabla 14.1. Posibilidades ofrecidas por los Generadores/Visualizadores de esquemas de diferentes gestores

14.3.1.3 Resumen de características para el generador/visualizador de esquemas de BDOviedo3

Todo esto nos permite concluir que las características que debe proporcionar el Generador/Visualizador de esquemas en BDOviedo3 son:

- Definición textual del esquema mediante un editor dirigido por sintaxis.
- Definición visual completa, compatible con el *binding* Java de ODMG, de atributos, relaciones, métodos, extensión de clases, índices, etc.

- Visualización del árbol jerárquico de las clases de la base de datos, así como del diagrama de las clases que constituyen el esquema de la misma.
- Selección del detalle de las vistas a obtener.

14.3.2 Navegación-inspección

Es deseable que el entorno para nuestro SGBDOO permita la inspección de la información contenida en la base de datos sin necesidad de acudir a la implementación de un programa o a la formulación de una consulta en el lenguaje OQL. Esta inspección o navegación (*browsing*) es una tarea realizada frecuentemente y como tal debe ser facilitada.

14.3.2.1 Inspección sincronizada

Aparte de una visión general de los objetos almacenados en la base de datos, que la incorporan ya la mayoría de los sistemas existentes (Poet [Poe00] en forma tabular, Goodies en forma de lista, etc.), a nuestro sistema se le exige la capacidad de realizar una inspección sincronizada como la propuesta en OdeView [AGS90]. Los atributos de un objeto que no son tipos primitivos (referencias a otros objetos) pueden ser visualizados en una nueva ventana (se crea un enlace de sincronización). Este proceso puede realizarse recursivamente, con lo que se genera un árbol de sincronización. Al recorrer los objetos que son raíz de un cierto árbol de sincronización, en las ventanas que se encontraban abiertas se muestran de forma automática los sub-objetos correspondientes.

	Poet Developer	PSE Pro (Java Browser)	JADE	Goodies	OdeView
Extensión de la clase	Tabular	Tabular	Iconos	Lista	Lista
Objeto Simple	Lista	Lista	Lista	Formulario	Formulario
Inspección Sincronizada	No	No	No	Si	Si

Tabla 14.2. Posibilidades de navegación-inspección de diferentes gestores

14.3.2.2 Resumen de características para la navegación-inspección en BDOviedo3

Atendiendo a lo anteriormente comentado, en BDOviedo3 se permitirá a la hora de visualizar la extensión de una clase seleccionar entre varias opciones:

- Tabular
- Lista
- Iconos
- Inspección sincronizada

Por otro lado, a la hora de visualizar un objeto simple se podrá seleccionar entre la visualización en forma de lista o bien en modo de formulario.

14.3.3 Entorno de consultas

El lenguaje de consulta empleado para el SGBDOO BDOviedo3 es el OQL (*Object Query Language*) propuesto por el estándar ODMG 2.0. Este lenguaje, como puede

observarse en el Apéndice A tiene un estilo similar al SQL-92 en cuanto a formulación de las consultas *select-from-where*, con los conceptos propios del manejo de objetos, como objetos compuestos, identidad de objetos, expresiones de camino, polimorfismo, invocación de métodos, y ligadura tardía o dinámica.

La mayoría de los SGBDOO no se preocupan en exceso de la creación de una interfaz adecuada para el lenguaje de consulta, que facilite la realización de las mismas y minimice además el número de errores. Así, algunos entornos como GOODIES [LO93] y OdeView [AGS90] se limitan a seleccionar los objetos a visualizar mediante la formulación de predicados de selección según los valores de alguno de los atributos de los objetos. Otros, como POET 6.0, permiten la escritura de consultas en lenguaje OQL guardándolas y recuperándolas como un fichero de texto.

	Poet Developer	PSE Pro (Java Browser)	JADE	Goodies	Ode View	VOODOO
Formulación Textual	Si	No	Si	No	No	Si
Formulación Visual	No	No	No	No	No	Si
Generación OQL	OQL Java/C++	OQL Java	No	No	No	Si

Tabla 14.3. Posibilidades de diferentes entornos de consultas

14.3.3.1 Resumen de características del entorno de consultas para BDOviedo3

El entorno de consultas de BDOviedo3 proporcionará un soporte completo para el OQL de ODMG, permitiendo la formulación de las consultas de una forma visual (además de, por supuesto, textual). El modelo seguido para ello está basado en los trabajos mostrados en VOODOO [Feg99], y propone la representación visual de un árbol en el que cada nodo representa una clase del esquema de la base de datos. Para formular la consulta se establecen valores para algunos de los atributos simples de la clase, y los atributos complejos pueden expandirse creando un nuevo nodo del árbol. El resultado de la consulta se genera desde las hojas hacia la raíz, de forma que el valor de un nodo se eleva al nodo predecesor para calcular su valor y así llegar hasta el nodo origen. El resultado de la consulta es el valor de la raíz del árbol.

Capítulo 15 Sistema integral orientado a objetos Oviedo3

Con el fin de construir un prototipo para el SGBDOO con las características mencionadas en los capítulos anteriores, es necesario partir de un sistema integral orientado a objetos. En este capítulo se hace una descripción del sistema integral orientado a objetos Oviedo3, prototipo sobre en el que se asentará el SGBDOO mencionado.

15.1 El sistema integral orientado a objetos Oviedo3

Oviedo3 [CIA+96] es un proyecto investigación que está siendo desarrollado por un equipo de profesores y alumnos del grupo de investigación en Tecnologías Orientadas a Objetos de la Universidad de Oviedo, que pretende construir un sistema integral orientado a objetos con la arquitectura descrita en el capítulo 7.

Los elementos básicos que proporcionarán el espacio de objetos del sistema integral son la máquina abstracta Carbayonia y el sistema operativo SO4 [ATA+97] que extiende las capacidades de la misma.

La combinación de estos dos elementos forma el SIOO Oviedo3. Esta plataforma permite dar un soporte directo a las tecnologías orientadas a objetos, y desarrollar más rápidamente todos los demás elementos de un sistema de computación. Todos estos elementos, desde la máquina hasta la interfaz de usuario utilizarán el mismo paradigma de la orientación a objetos.

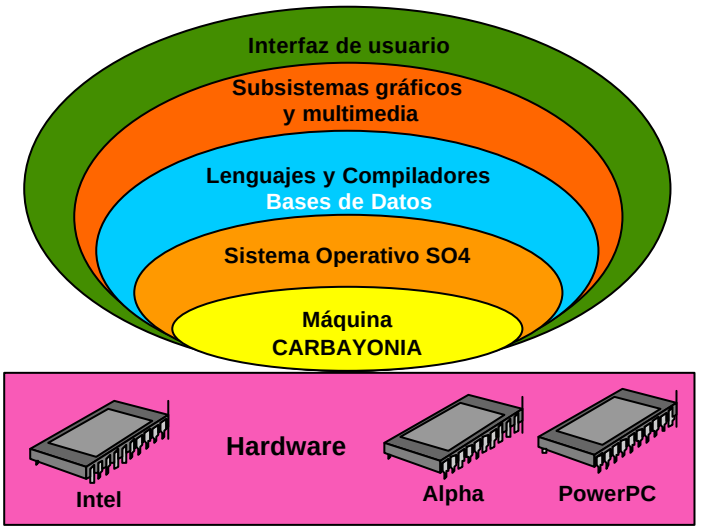


Figura 15.1. Esquema general del sistema integral orientado a objetos Oviedo3

Los fines del proyecto Oviedo3 son tanto didácticos como de investigación. En lo que a investigación se refiere, se pretende usar el sistema como plataforma de investigación para las diferentes áreas de las tecnologías de objetos en un sistema de

computación: la propia máquina abstracta y sistema operativo orientado a objetos, así como lenguajes y compiladores OO, bases de datos OO, sistemas gráficos y multimedia, interfaces de usuario, etc.

15.2 La máquina abstracta carbayonia

Carbayonia es una máquina abstracta orientada a objetos que sigue la estructura de referencia marcada en el capítulo 7. La descripción de la máquina está adaptada de la documentación del primer prototipo de la misma, desarrollado como proyecto fin de carrera de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Oviedo [Izq96].

La máquina es una máquina abstracta orientada a objetos pura que implementa el modelo único de objetos del sistema. Todos los objetos tienen un identificador que se usa para operar con ellos exclusivamente a través de referencias. Las referencias, como los propios objetos, tienen un tipo asociado y tienen que crearse y destruirse. En Carbayonia todo se hace a través de referencias: las instrucciones tienen referencias como operandos; los métodos de las clases tienen referencias como parámetros y el valor de retorno es una referencia.

15.2.1 Estructura

Para comprender la máquina Carbayonia se utilizan las tres áreas de la máquina de referencia. Un área se podría considerar como una zona o bloque de memoria de un microprocesador tradicional. Pero en Carbayonia no se trabaja nunca con direcciones físicas de memoria, si no que cada área se puede considerar a su vez como un objeto el cual se encarga de la gestión de sus datos, y al que se le envía mensajes para que los cree o los libere.

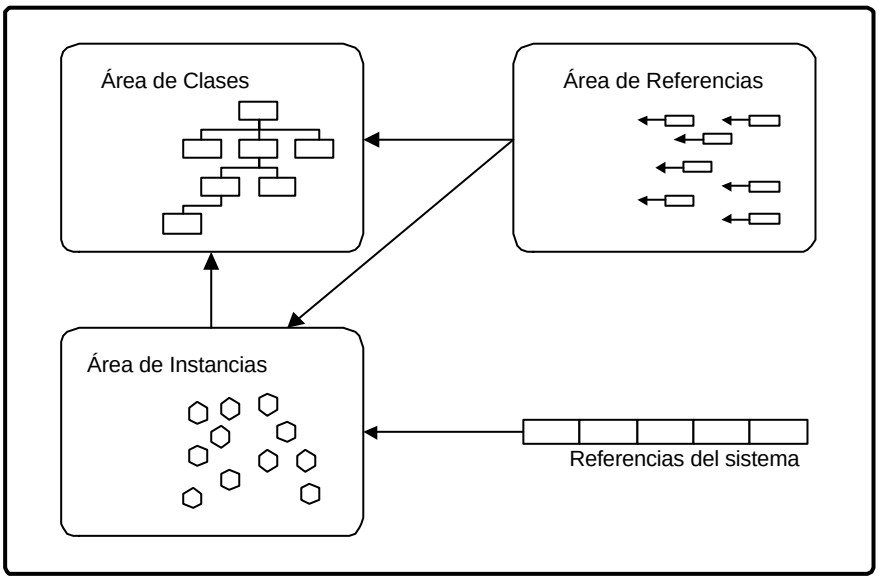


Figura 15.2. Estructura de referencia de una máquina abstracta para el sistema integral

A continuación se expone una breve descripción de las áreas que componen Carbayonia, comparándola para una mejor comprensión con arquitecturas convencionales como las de Intel x86. Sus funciones se verán mas en detalle en la descripción de las instrucciones.

15.2.1.1 Área de clases

En éste área se guarda la descripción de cada clase. Esta información está compuesta por los métodos que tiene, qué variables miembro la componen, de quién deriva, etc. Esta información es fundamental para conseguir propiedades como la comprobación de tipos en tiempo de ejecución (RTTI, *Run Time Type Information*), la invocación de métodos con polimorfismo, etc.

Aquí ya puede observarse una primera diferencia con los micros tradicionales, y es que en Carbayonia realmente se guarda la descripción de los datos. Por ejemplo, en un programa en ensamblador del 80x86 hay una clara separación entre instrucciones y directivas de declaración de datos: las primeras serán ejecutadas por el micro mientras que las segundas no. En cambio Carbayonia ejecuta (por así decirlo) las declaraciones de las clases y va guardando esa descripción en éste área.

15.2.1.2 Área de instancias

Aquí es donde realmente se almacenan los objetos (instancias de las clases). Cuando se crea un objeto se deposita en éste área, y cuando éste se destruye se elimina de aquí. Se relaciona con el área de clases puesto que cada objeto es instancia de una clase determinada. Así desde un objeto se puede acceder a la información de la clase a la que pertenece.

Las instancias son identificadas de forma única con un número que asignará la máquina en su creación. La forma única por la que se puede acceder a una instancia es mediante una referencia que posee como identificador el mismo que la instancia. Se mantiene el principio de encapsulamiento. La única forma de acceder a una instancia mediante una referencia es invocando los métodos de la instancia.

15.2.1.3 Área de referencias

Para operar sobre un objeto necesitamos antes una referencia al mismo. En éste área es donde se almacenan dichas referencias. El área de referencias se relaciona con el área de clases (ya que cada referencia tiene un tipo o clase asociado) y con el área de instancias (ya que apuntan a un objeto de la misma). Una referencia se dirá que está *libre* si no apunta a ningún objeto. Las referencias son la única manera de trabajar con los objetos[1].

15.2.1.4 Referencias del sistema

Son una serie de referencias que están de manera permanente en Carbayonia y que tienen funciones específicas dentro del sistema. Estas referencias son las siguientes:

- **this**: Apunta al objeto con el que se invocó el método en ejecución.
- **exc**: Apunta al objeto que se lanza en una excepción.
- **rr** (*return reference*): Referencia donde los métodos dejan el valor de retorno.

15.2.2 El lenguaje Carbayón: juego de instrucciones

El juego de instrucciones de la máquina se describe en términos del lenguaje ensamblador asociado al mismo. Este lenguaje se denomina Carbayón y será la interfaz

[1] Por tanto, aunque en algunos casos se mencionen los objetos directamente, como por ejemplo "se devuelve un objeto de tipo cadena" se entiende siempre que es una referencia al objeto.

de las aplicaciones con la máquina. En cualquier caso, existe la posibilidad de definir una representación compacta de bajo nivel (*bytecode*) de este lenguaje que sea la que realmente se entregue a la máquina.

A continuación se muestra un ejemplo de programación en lenguaje Carbayón (Código 15.1). Se trata de la definición de una clase *ServidorNombres* que proporciona cierta funcionalidad de un servidor de nombres. Dispone de dos métodos: *setTamanio*, para inicializar el servidor y *registrarRef*, para registrar un nombre simbólico a asociar a una referencia a objeto.

```
CLASS ServidorNombres
Aggregation
      Nelems:integer;
      Tabla:array;
Methods
SetTamanio()
Instances
      Cero:integer(0);
Code
      Tabla.setsize(cero);
      Nelems.set(cero);
      Exit;

Endcode

RegistrarRef(nom:string;ref:object)
Refs
      Tam:integer;
Instances
      Uno:integer(1);
      Dos:integer(2);
Code
      Tabla.getsize():tam;
      Tam.add(dos);
      Tabla.setsize(tam);
      Tabla.setref(nelems,nom);
      Nelems.add(uno);
      Tabla.setref(nelems,ref);
      Nelems.add(uno);
      Exit;
Endcode
ENDCLASS
```

Código 15.1. Ejemplo de programación en lenguaje Carbayón

15.2.3 Características de las clases Carbayonia

A continuación se describe brevemente las características propias de las clases Carbayonia.

15.2.3.1 Herencia virtual

Un aspecto a destacar es que toda derivación es virtual. Al contrario que en C++, no se copian simplemente en la clase derivada los datos de las superclases. Al retener toda la semántica del modelo de objetos en tiempo de ejecución, simplemente se mantiene la información de la herencia entre clases. Es decir, en la estructura de herencias tipo como la de la Figura 15.3, la clase *D* sólo tiene una instancia de la clase *A*. Se mantiene sincronizada respecto a los cambios que se puedan hacer desde *B* y desde *C*. A la hora de crear instancias de una clase, se repite la misma estructura.

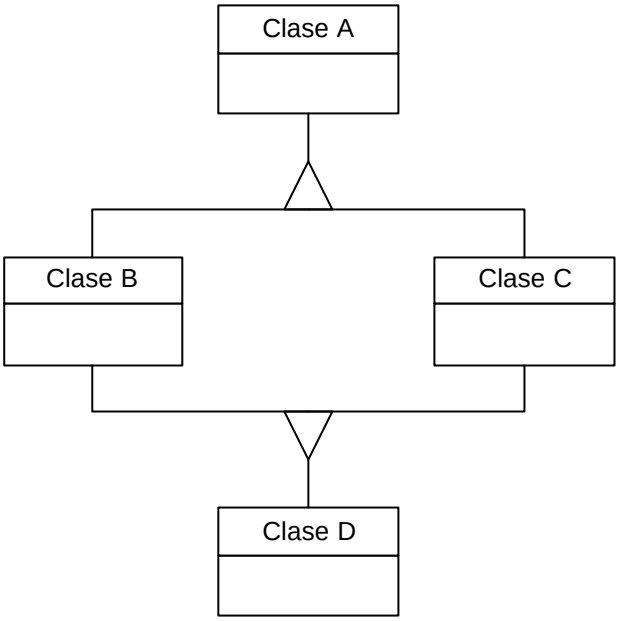


Figura 15.3. Jerarquía de herencia múltiple con ancestro compartido

15.2.3.2 Herencia múltiple. Calificación de métodos

El problema de la duplicidad de métodos y variables en la herencia múltiple se soluciona dando prioridad a la superclase que aparece antes en la declaración de la clase. Si en la figura anterior se supone que tanto la clase *B* como la clase *C* tienen un método llamado *M*, y desde *D* (o a través de una referencia *a* a *D*) se invoca a dicho método, se ejecutará el método de la clase que primero aparezca en la sección *Isa* de la clase *D*. En caso de que se desee acceder al otro método se deberá calificar el método, especificando antes del nombre del método la clase a la que pertenece, separada por dos puntos.

```
a.Metodo()          // método de la clase B
a.ClaseB:Metodo()   // método de la clase B
a.ClaseC:Metodo()   // método de la clase C
```

Código 15.2. Calificación de métodos en Carbayón

15.2.3.3 Uso exclusivo de métodos

No existen operadores con notación infija. Los operadores son construcciones de alto nivel que se implementarán como métodos. Esto es lo que hace al fin y al cabo el C++ a la hora de sobrecargarlos.

15.2.3.4 Uso exclusivo de enlace dinámico (sólo métodos virtuales)

No es necesario especificar si un método es virtual[2] o no, ya que todos los métodos son virtuales (polimórficos). Es decir, se utiliza únicamente el mecanismo de enlace dinámico, siguiendo la línea de una arquitectura OO más pura. El uso de enlace estático restringe en exceso la extensibilidad del código, perdiéndose una de las ventajas de la

[2] En el sentido de C++ de la palabra virtual: utilizar enlace dinámico con ese nombre de método.

OO. La posible sobrecarga de ejecución de los métodos virtuales se puede compensar con una implementación interna eficiente.

15.2.3.5 Ámbito único de los métodos

No se restringe el acceso a los métodos clasificándolos en ámbitos como los *private*, *public* o *protected* del C++. Estos accesos son de utilidad para los lenguajes de alto nivel pero para una máquina no tienen tanto sentido. Si un compilador de alto nivel no desea que se accedan a unos determinados métodos *private*, lo único que tiene que hacer es no generar código de llamada a dichos métodos. Un ejemplo parecido ocurre cuando se declara una variable *const* en C++. No es que la máquina subyacente sepa que no se puede modificar; es el compilador el que no permite sentencias que puedan modificarla.

En cualquier caso, en el sistema operativo existe un mecanismo de protección (véase el capítulo 9) que permitirá una restricción de acceso individualizada para cada método y cada objeto. Así se podrán crear ámbitos de protección particularizados para cada caso, en lugar de simplemente en grupos *private*, *public* y *protected*.

15.2.3.6 Inexistencia de constructores y destructores

No existen métodos especiales caracterizados como constructores ni destructores. Al igual que ocurre con algunos de los puntos anteriores, es el lenguaje de alto nivel el que, si así lo desea, debe generar llamadas a unos métodos que hagan dichas labores a continuación de las instrucciones de Carbayonia de creación y destrucción de objetos.

Sin embargo, no es necesario que se aporten métodos para la gestión de la semántica de los objetos agregados[3], que ya es conocida por la máquina y se realiza automáticamente.

15.2.3.7 Redefinición de métodos

Para que un método redefina (*overriding*) a otro de una superclase debe coincidir exactamente en número y tipo de parámetros y en el tipo del valor de retorno. No se permite la sobrecarga (*overloading*) de métodos (dos o más métodos con el mismo nombre y diferentes parámetros).

15.2.4 Jerarquía de clases básicas

Independientemente de las clases que defina el programador (y que se irán registrando en el área de clases), Carbayonia tiene una serie de clases básicas (también denominadas primitivas) que se pueden considerar como definidas en dicho área de manera permanente.

- Object
- Bool
- Integer
- Float
- String
- Array

[3] Como por ejemplo su eliminación al eliminarse el objeto que los contiene.

Estas clases básicas serán las clases fundamentales del modelo único que se utilizarán para crear el resto de las clases. Una aplicación Carbayonia es un conjunto de clases con sus métodos en los cuales se llaman a otros métodos. Siguiendo este proceso de descomposición, siempre llegamos a las clases básicas y a sus métodos.

En cualquier caso, estas clases básicas no se diferencian en nada de cualquier otra clase que cree el usuario. Desde el punto de vista de utilización son clases normales como otras cualesquiera. Cada implementación de la máquina establecerá los mecanismos necesarios para proporcionar la existencia de estas clases básicas.

Las clases básicas se organizan en una jerarquía, cuya raíz es la clase básica *Object*.

15.2.5 Clase básica objeto

Se define clase básica como aquella cuya implantación está codificada internamente en la propia máquina abstracta, utilizando para ello el lenguaje de desarrollo empleado para codificar la máquina.

Dentro de la jerarquía de clases básicas ofertadas por la máquina abstracta, se encuentra la clase **Object**, de la que derivan todos los objetos existentes en el sistema, bien sean de usuario o de otras capas superiores como el sistema operativo, bases de datos, etc.

15.2.5.1 Nivel de abstracción único: Uniformidad en torno al objeto

Existen dos características de la máquina abstracta que garantizan la uniformidad en torno a la abstracción de objeto, o lo que es lo mismo, impiden que la máquina proporcione ninguna otra abstracción en tiempo de ejecución, excepto el objeto.

Juego de instrucciones

Por un lado, el juego de instrucciones reducido que la máquina abstracta ofrece pocas posibilidades al usuario para añadir elementos nuevos en tiempo de ejecución.

Concretamente, las únicas instrucciones que la máquina ofrece para añadir elementos son la **instrucción declarativa** *Class* para añadir una clase o bien, la **instrucción** *new* para crear una instancia de una clase, o sea, un objeto.

Por tanto, las operaciones que la interfaz de la máquina abstracta permite realizar en tiempo de ejecución, se limitan a aquellas referidas a la definición, creación y manipulación de objetos.

Es decir, en tiempo de ejecución, el único elemento que existe es el objeto y, para resolver un problema del mundo real, la máquina abstracta permite crear objetos e invocar métodos en los mismos. No existe ninguna otra abstracción en tiempo de ejecución.

Clase básica

Por otro lado, la máquina abstracta define un objeto como una instancia en tiempo de ejecución de una clase, creada gracias a la instrucción *new*[4].

[4] Se puede considerar que la instrucción new es equivalente al envío del mensaje new al objeto clase concreto del que se pretende crear una instancia.

Toda clase pertenece a la jerarquía de clases de la máquina, que tiene como raíz la clase *Object*. Esta clase define el comportamiento básico de cualquier nuevo elemento que se cree en el sistema.

La definición de una clase básica de objetos de la que derivan todos los objetos existentes en el sistema, promueve que todos los objetos estén dotados de una serie de características básicas que, ineludiblemente, heredarán.

15.2.5.2 Identificador único de objetos

Todo objeto que existe en el sistema en tiempo de ejecución es una instancia de clase básica *Object* y se comportará según el comportamiento definido para él por la máquina abstracta, al tratarse esta de una clase básica.

Cuando se invoca la instrucción *new* sobre cualquier clase de la jerarquía de clases de la máquina (sean estas clases básicas o de usuario), la máquina abstracta crea una nueva instancia de un objeto al que proporciona un identificador único y global en el sistema [ATD+98a, ATD+98b].

15.2.5.3 Reflectividad: Aumento del soporte al objeto por parte de la máquina abstracta

El aumento en la semántica del objeto supone un soporte más sofisticado al objeto. Dicho de otra forma, la representación del objeto que la máquina soporta se ve modificada por la necesidad de mantener la relación entre los objetos y el conjunto de meta-objetos que completan su entorno de ejecución.

Soporte en el entorno de ejecución base al enlace base-meta

Se añade a la clase básica *Object*, la referencia *meta.* De esta forma, las instancias en tiempo de ejecución podrían acceder a su meta-espacio simplemente consultando a qué instancia apunta la referencia *meta.*

Así, pueden determinar a qué objetos pasarle el control cuando la máquina tenga lagunas en el proceso de ejecución. También se hace posible cambiar el meta-objeto simplemente cambiando el meta-objeto al que apunta la referencia *meta.*

Realmente, este enlace *meta* no será una única entrada sino un conjunto de entradas a los distintos meta-objetos que definen el entorno en tiempo de ejecución del objeto base, a saber, los meta-objetos encargados del paso de mensajes, los meta-objetos encargados del control de la concurrencia y aquel o aquellos encargados de la planificación.

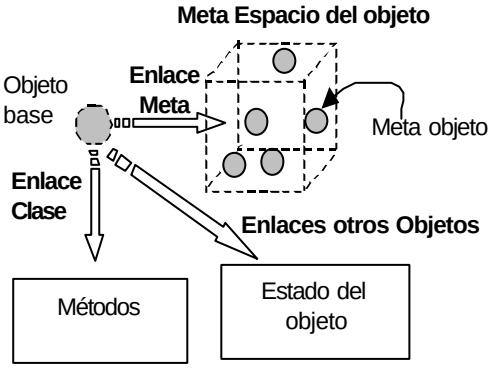


Figura 15.4. Enlace meta: asociación del objeto y su meta espacio

15.3 El sistema operativo SO4

El sistema operativo SO4 [ATA+96, ATA+97] es el encargado de extender la funcionalidad básica de la máquina abstracta Carbayonia. Los elementos más importantes de esta funcionalidad son los que permiten dotar a los objetos de manera transparente con las propiedades de persistencia, concurrencia, distribución y seguridad.

El objetivo fundamental a la hora de implantar estas propiedades es la transparencia**.** Los objetos adquirirán estas propiedades sin hacer nada especial. No tienen por qué ser conscientes ni intervenir en los mecanismos que se usan para lograrlas (excepto en el caso en que sea imprescindible, como los objetos del sistema operativo). Por otro lado es muy importante que la introducción de estas propiedades se integre de manera fluida con el modelo de objetos del sistema. Es decir, que no se necesiten cambios en la semántica del modelo, o que sean menores y no rompan la esencia del modelo.

Para alcanzar estos objetivos, el sistema operativo utilizará el mecanismo de reflectividad proporcionado por la máquina abstracta. La arquitectura del sistema, basado en la máquina abstracta, ciertas propiedades del modelo de objetos, la reflectividad y la extensión en el espacio del usuario facilitan la implementación del sistema operativo, para alcanzar el objetivo de un mundo de objetos: un único espacio de objetos, virtualmente infinito, en el que un conjunto de objetos homogéneos coopera intercambiando mensajes independientemente de su localización, y donde los objetos residen indefinidamente hasta que ya no son necesitados.

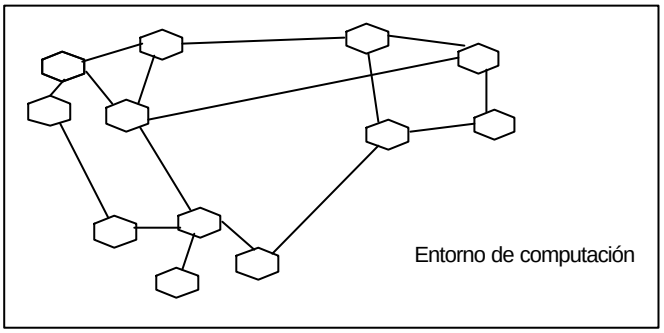


Figura 15.5. Entorno de computación compuesto por un conjunto de objetos homogéneos

El diseño en profundidad de las propiedades que debe implementar el sistema operativo, así como la implementación de las mismas está siendo investigado por otros miembros del proyecto Oviedo3. Los resultados de dichas investigaciones se pueden encontrar en [Álv98, Día00, Taj00]. A continuación se resumen los planteamientos de dichas investigaciones.

15.3.1 Persistencia

La idea básica para integrar la persistencia en el sistema es proporcionar la abstracción de un espacio de objetos virtualmente infinito en el que residen los objetos hasta que no se necesitan. Podría considerarse como una memoria virtual persistente para los objetos.

Para conseguir esto, se propone una implementación [Álv98] basada en la idea de conseguir en la máquina un área de instancias virtual, empleando para ello un almacenamiento secundario y un objeto paginador que entrará en funcionamiento

gracias a la reflectividad de la máquina, y que se encargará del trasiego de objetos entre el área de instancias y el almacenamiento persistente.

Las modificaciones para permitir un modelo de persistencia en la máquina abstracta [OAI+97] aparecen reflejadas en una versión de la máquina desarrollada como proyecto fin de carrera de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Oviedo [Ort97].

15.3.2 Seguridad

En un sistema de objetos homogéneos como el que nos ocupa el objetivo del sistema de seguridad es proporcionar un mecanismo de protección uniforme para controlar el acceso a los métodos de cualquier objeto del sistema. Es decir, que objetos tienen acceso a cada operación de un objeto determinado.

La protección de los objetos se consigue acudiendo al empleo de capacidades como referencias a los objetos y a un mecanismo de protección en el nivel más interno de la máquina [Día00]. Para implantar las capacidades se ha modificado la máquina abstracta:

- **Adición de atributos en las referencias.** Se amplia la arquitectura de la máquina para que las referencias pasen a ser capacidades, es decir, se amplia el contenido de una referencia con la información de protección.
- **Adición y modificación de operaciones con referencias.** Las operaciones actuales con referencias se mantienen en la máquina y su funcionamiento es similar, simplemente hay que tener en cuenta la existencia de la información de protección. Sin embargo, deben añadirse operaciones adicionales que tengan en cuenta la nueva categoría de las referencias como capacidades.
- **Modificación del mecanismo de envío de mensajes.** El mecanismo de envío de mensajes de la máquina se basa en tomar una referencia, localizar el objeto e invocar la operación deseada. Ahora, tras localizar el objeto mediante el identificador del mismo, se debe examinar la información de protección y si se tienen permisos para ello, invocar la operación deseada. En caso de que no se tengan los permisos necesarios, se lanza una excepción.

Estas modificaciones han sido realizadas en la máquina abstracta y aparecen reflejadas en una versión de la máquina desarrollada como proyecto fin de carrera de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Oviedo [Gar99b].

15.3.3 Concurrencia

El sistema de concurrencia deberá proporcionar un modelo de concurrencia que permita a los objetos la realización de tareas concurrentes. Se debe permitir la concurrencia entre objetos, entre métodos del mismo objeto y entre varias ocurrencias del mismo método de un objeto. Todo ello de la manera más segura posible, garantizando la corrección de las operaciones efectuadas y, por tanto, la consistencia del objeto [Taj00].

Parte de la funcionalidad del modelo de concurrencia se implementará en la máquina y parte como objetos normales de usuario. La capacidad de ejecución de métodos y la caracterización de métodos como concurrentes o exclusivos se dejará a cargo de la máquina. La máquina colaborará con objetos del sistema operativo, que implementarán las políticas de planificación.

Las modificaciones para conseguir la concurrencia en estas condiciones han sido realizadas en la máquina abstracta y aparecen reflejadas en una versión de la máquina desarrollada como proyecto fin de carrera de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Oviedo.

15.3.4 Distribución

En un entorno distribuido compuesto por varias máquinas Carbayonia[5] conectadas mediante una red de comunicación, el sistema de distribución permitirá básicamente la comunicación entre los objetos independientemente de la máquina en la que se encuentren [Álv00].

La extensión realizada por el sistema operativo para la distribución es llevada a cabo en dos sentidos: en primer lugar se posibilita la invocación de métodos de objetos de manera transparente con independencia de su ubicación; y en segundo lugar, los objetos pueden moverse entre las diferentes máquinas abstractas del sistema.

Con el fin de proporcionar las dos facilidades de distribución mencionadas, se definen un conjunto de objetos del sistema operativo (meta-objetos) asociados a cada objeto, que describen un comportamiento que sustituye al descrito de manera primitiva por la máquina abstracta.

Estas modificaciones han sido realizadas en la máquina abstracta y aparecen reflejadas en una versión de la máquina desarrollada también como proyecto fin de carrera de la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Oviedo.

[5] Todas las máquinas abstractas Carbayonia tienen la misma funcionalidad. Sin embargo, el hardware subyacente puede ser heterogéneo.

Capítulo 16 Diseño e implementación de un prototipo del motor para el SGBDOO

En el capítulo 10 se ha descrito una arquitectura para un SGBDOO construido sobre un sistema integral orientado a objetos, y se han especificado las propiedades básicas del motor o núcleo de dicho sistema. En dicho núcleo se han identificado una serie de módulos encargados de proporcionar las distintas funcionalidades del gestor, prestando especial atención al mecanismo de indexación con el que se ha de dotar al sistema, especificado en el capítulo 12.

En este capítulo se describen algunas características del diseño e implementación del prototipo de un motor para el SGBDOO BDOviedo3, construido sobre el sistema integral orientado a objetos Oviedo3, y con un mecanismo de indexación extensible.

16.1 Introducción

El prototipo aquí presentado fue desarrollado como Proyecto Fin de Carrera de la Escuela Superior de Ingenieros Industriales e Ingenieros Informáticos de la Universidad de Oviedo. La descripción completa del mismo se puede consultar en [Gar99].

El objetivo fundamental del prototipo era desarrollar en un tiempo reducido un producto que permitiera comprobar, en la práctica, el comportamiento de un motor de un sistema de gestión de bases de datos orientadas a objetos construido sobre un sistema integral orientado a objetos con un mecanismo de indexación extensible. Inicialmente la eficiencia con respecto al sistema global pasaba a un segundo plano.

16.2 Características proporcionadas por el sistema integral

La construcción de este primer prototipo está basada en una versión de la máquina abstracta que únicamente proporciona un sistema de persistencia [Ort97], y que no está dotada todavía de reflectividad, protección, concurrencia o distribución. Dicha máquina proporciona entre otras las características que se comentan a continuación [OAI+97].

16.2.1 Persistencia ortogonal no completa

La máquina permite que cualquier objeto que se desee pueda ser persistente pero indicándoselo explícitamente. Existirán, por tanto, objetos temporales y persistentes aunque no se diferenciarán en su utilización.

16.2.2 Declaración explícita de las clases persistentes

Los objetos que sean susceptibles de ser hechos persistentes tendrán que pertenecer a una clase persistente. Esto se realiza anteponiendo la palabra clave *Persistent* a la declaración de la clase. Para los agregados de una clase también se puede elegir cuáles de ellos serán persistentes. Para ello también se antepone la palabra *Persistent* delante de la declaración de estas referencias agregadas.

16.2.3 Persistencia por llamada explícita

La máquina incluye una clase primitiva *Persistence* que proporciona métodos que permiten añadir un objeto al sistema de persistencia y darle un nombre simbólico, eliminar un objeto, etc.

```
Class Persistence
Isa Object
Methods
  exists(String):Bool;
  add (String,Object);
  getObject(String):Object;
  remove(String);
EndClass
```

Código 16.1. Declaración de la clase Persistence en Carbayón

- **Exists(String):Bool.** Devuelve el valor booleano referente a la existencia del objeto persistente en el sistema de persistencia, cuyo nombre simbólico en el sistema de persistencia es la cadena pasada.
- **Add(String, Object).** Dándole un nombre asociado y el objeto, almacena el objeto en el sistema de persistencia. Se produce una excepción si el objeto con ese nombre ya existe.
- **GetObject(String):Object.** Devuelve el objeto asociado en el sistema de persistencia al nombre simbólico pasado. Se produce una excepción si el objeto con ese nombre no existe en el sistema de persistencia.
- **Remove(String)**. Elimina el objeto del sistema de persistencia, cuyo nombre sea la cadena de caracteres pasada como parámetro. Se produce una excepción si el objeto con ese nombre no existe en el sistema de persistencia.

Se añade también una referencia del sistema, *persistence*, que apunta a una instancia de esa clase, de forma que el usuario accederá al servicio de directorio de persistencia a través de esta referencia, invocando sus métodos.

16.2.4 Transparencia para el usuario en el trasiego disco/memoria

El sistema de persistencia crea un área de instancias virtual en la que pueden residir todas las instancias que se necesiten. En el caso de los objetos persistentes, el sistema de persistencia se ocupa de almacenarlos en disco y traerlos al área de instancias cuando sea necesario, sin intervención del usuario.

16.2.5 Servidor de páginas

El sistema de persistencia divide la memoria virtual en bloques de igual tamaño, dentro de los cuales se almacenan los segmentos que representan los objetos persistentes. Esto implica que cuando se necesite un objeto en realidad se traerá ese objeto y todos los demás que compartan su página. De la misma manera cuando se grabe ese objeto en almacenamiento secundario se grabará con todos los que formen su página. Para evitar inconsistencias realiza un bloqueo a nivel de página.

16.2.6 Identificador con dirección estructurada

El identificador asociado a cada objeto lleva información de la página en la que se encuentra el objeto y del desplazamiento que ocupa su segmento dentro de la página.

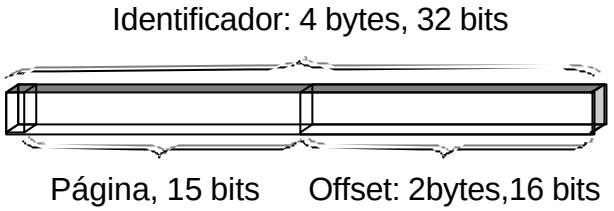


Figura 16.1. Formato del identificador de objetos persistentes

El primer bit del identificador se utiliza para etiquetarlo como identificador de un objeto persistente. El resto se pueden utilizar para representar la página y el desplazamiento dentro de la página. El número de bits que se dediquen a cada apartado determina el tamaño de página y el número total de páginas posibles en la memoria virtual.

16.2.7 Buffering

Para mejorar el rendimiento, en lugar de que la memoria virtual se represente directamente sobre el disco, el sistema de persistencia utiliza una memoria intermedia (caché) que almacena en memoria principal los últimos objetos persistentes utilizados. La funcionalidad de esta memoria intermedia es similar a la del *buffer caché* de entrada/salida de Unix.

El emparejamiento entre la representación normal y la persistente se realiza sobre la memoria intermedia. Dado que la memoria intermedia es limitada y en algún momento se agotará, es necesario gestionar el espacio mediante determinadas políticas de *emplazamiento* y *reemplazamiento.* Además, permite la utilización de diferentes políticas para ello, así como la variación del tamaño de página con fines experimentales.

16.2.8 Lenguaje Carbayón

El lenguaje de implementación de este prototipo de BDOviedo3 es el propio lenguaje ensamblador de la máquina Carbayonia. Este es un lenguaje completamente orientado a objetos cuya unidad declarativa básica es la clase, y será por tanto ésta la unidad mínima de trabajo que las aplicaciones comuniquen a la máquina.

El empleo de este lenguaje garantiza la utilización del modelo de objetos descrito en el capítulo 7, y la posibilidad de emplear la herencia y el polimorfismo en la construcción del motor, facilitando así la extensibilidad del mismo.

16.3 Funcionalidad proporcionada por el prototipo

El desarrollo de este primer prototipo se ha centrado en la construcción de un motor para el SGBDOO, cuya funcionalidad básica gira en torno a la gestión de índices. Era objetivo de este primer prototipo implementar un mecanismo de indexación extensible, que pudiese con posterioridad ser ampliado con otras funcionalidades como la optimización de consultas.

16.3.1 Gestión de los esquemas

El motor proporciona la gestión del esquema o catálogo de la base de datos. Para ello almacena información sobre las clases, los índices especificados y la propia extensión de la clase.

Sin embargo, hay ligeras diferencias con la propuesta realizada en el capítulo 10. Dichas diferencias derivan principalmente del hecho de que en esta versión de la máquina abstracta no es posible interrogar el área de clases para conocer las características: atributos, tipos de éstos, métodos, etc. de la clase de un determinado objeto.

16.3.2 Gestión de las extensiones

El motor construido en este prototipo gestiona las extensiones de las clases automáticamente, tal y como se describió en el capítulo 10, no siendo necesario que el usuario declare las colecciones (como en PSE Pro), ni las mantenga. Es decir, es totalmente transparente para el usuario.

16.3.2.1 Colecciones para el almacenamiento de las extensiones

En este prototipo para facilitar la gestión de las extensiones se acude al empleo de colecciones, al igual que hace por ejemplo PSE PRO, pero a diferencia de éste la gestión de las mismas es automática. Para mayor flexibilidad en la gestión de las extensiones en este prototipo se incorporan al sistema las clases *TList*, *TSet* y *Thash.*

16.3.3 Gestión de los índices

Este prototipo incluye un mecanismo de indexación extensible que permite fácilmente la incorporación de nuevas técnicas de indexación. Las características de este mecanismo en el prototipo son analizadas en el apartado siguiente.

16.3.4 Facilidad de consultas limitada

Este prototipo no incluye un gestor de consultas, con las características descritas en el capítulo 10. Incluye únicamente una forma básica de recuperación de los datos almacenados en la base de datos. La recuperación puede realizarse de dos formas: empleando cualquiera de los índices definidos sobre las clases, o bien a modo de iterador, recorriendo simplemente las colecciones que representan las extensiones de las mismas.

En los métodos empleados para la recuperación de los datos, si está contemplada la existencia de las jerarquías de herencia, de forma que es posible indicar si en la consulta que se está realizando se desean obtener todos los objetos de la jerarquía cuya raíz es la clase objeto de la consulta, o bien las instancias de esa clase exclusivamente.

16.4 Características del mecanismo de indexación implementado

Como se ha expuesto en el capítulo12 la indexación es primordial a la hora de recuperar los datos ya almacenados en un SGBDOO. En este apartado se presentan las principales características del mecanismo de indexación proporcionado por el motor implementado en este primer prototipo.

16.4.1 Incorporación de nuevas técnicas de indexación

Este mecanismo de indexación permite la incorporación de nuevas técnicas de indexación, aplicables a cualquier tipo de datos. Sin embargo, inicialmente no se ha permitido la incorporación de nuevos operadores, por lo que las técnicas a incorporar han de estar basadas en los operadores mayor, menor e igual. Esto evidentemente es una limitación, que se está intentando subsanar en otros prototipos, no obstante, nos

permitiría incorporar la mayoría de las técnicas de indexación vistas para bases de datos orientadas a objetos, ya que la mayoría están basadas en árboles B^+, tal y como se pudo observar en el capítulo 11.

16.4.2 Indexación aplicable a cualquier tipo de dato

El mecanismo aquí diseñado esta preparado para soportar la indexación sobre cualquier tipo de datos, no únicamente tipos de datos primitivos como *Integer* o *String*. El índice puede estar constituido también por la combinación de diferentes atributos.

16.4.3 Configurabilidad del mecanismo

El mecanismo que nos ocupa permite especificar la técnica de indexación con la que se desea trabajar.

16.4.4 Gestión automática de los índices

A partir de la definición de los índices asociados a una clase el mantenimiento de los mismos es realizado automáticamente por el motor, de forma transparente para el usuario, a diferencia de otros sistemas como StreamStore [Str99] en el que el mantenimiento debe realizarlo el propio programador.

16.4.5 Propagación de los índices por la jerarquía

En este prototipo, los índices especificados para las clases base son heredados por las clases derivadas, al igual que en Jeevan [Jee99], y también mantenidos automáticamente.

16.4.6 Técnicas implementadas

El mecanismo de indexación de este prototipo [MCA+98] implementa las tres técnicas de indexación seleccionadas en el capítulo 12: *SC, CH-Tree* y *Path Index*.

16.4.7 Flexibilidad estática

El prototipo aquí presentado permite la incorporación de nuevas técnicas de indexación sin necesidad de reescribir el código fuente ya existente. Dicha ampliación se basa principalmente en la definición de una nueva clase que hereda de una clase base, y en la implementación de una serie de métodos básicos que permitan la utilización de la nueva técnica. La inserción de nuevas técnicas de indexación no podrá realizarse en tiempo de ejecución en este prototipo, siendo necesario registrar el índice en el motor previamente a cualquier utilización.

16.4.8 Plataforma de experimentación

Al soportar la incorporación de nuevas técnicas de indexación se convierte en una buena plataforma de experimentación (*benchmarking)* que permitirá llegar a resultados concluyentes sobre el comportamiento de los índices ante la gestión de determinados tipos de datos y aplicaciones.

16.5 Arquitectura

La funcionalidad de este prototipo es proporcionada en forma de API, y en su construcción se pueden distinguir cuatro elementos básicos:

- **Motor (OODB)**, que será un objeto que representa y proporciona toda la funcionalidad de este prototipo. Permite el acceso a las definiciones y extensiones de las clases que constituyen la base de datos. También permite la realización de consultas, y el mantenimiento y definición de los índices.
- **Gestor de extensiones** (*Repository*), como su nombre indica gestionará las extensiones de las clases.
- **Gestor de Esquemas** (*Catalog*), que almacena para cada una de las clases su extensión, su definición, y los índices especificados sobre ella. Permite la creación de los índices y las extensiones de las clases.
- **Gestor de índices** (*Indexes*), que gestiona los índices especificados para las clases que constituyen el esquema de la base de datos.

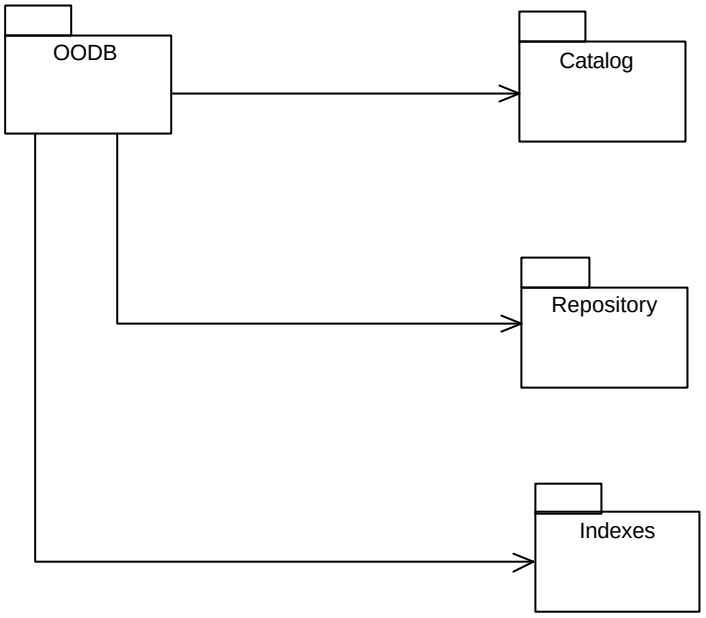


Figura 16.2. Esquema simplificado de los elementos que constituyen el prototipo

El objeto OODB no está definido en la máquina abstracta, y debe ser creado y almacenado por el futuro programador del sistema. Además, deberá existir un único objeto OODB en el sistema de persistencia de Carbayonia, que deberá ser recuperado para poder acceder a la funcionalidad que proporciona.

16.6 Descripción de la jerarquía de clases

El motor de este prototipo ha sido construido utilizando el lenguaje Carbayón proporcionado por la máquina abstracta, y por tanto ha seguido el paradigma de orientación de orientación a objetos. En el diseño de este motor han sido claves conceptos como la herencia y el polimorfismo, que facilitan considerablemente la definición de la jerarquía de clases ya que, por un lado, la herencia permite definir un objeto abstracto, raíz de la jerarquía, a partir del cual, mediante sucesivos refinamientos, se pueden derivar distintas especializaciones. Por su parte el polimorfismo permite que el hecho de qué clase se trate sea totalmente transparente al resto del sistema.

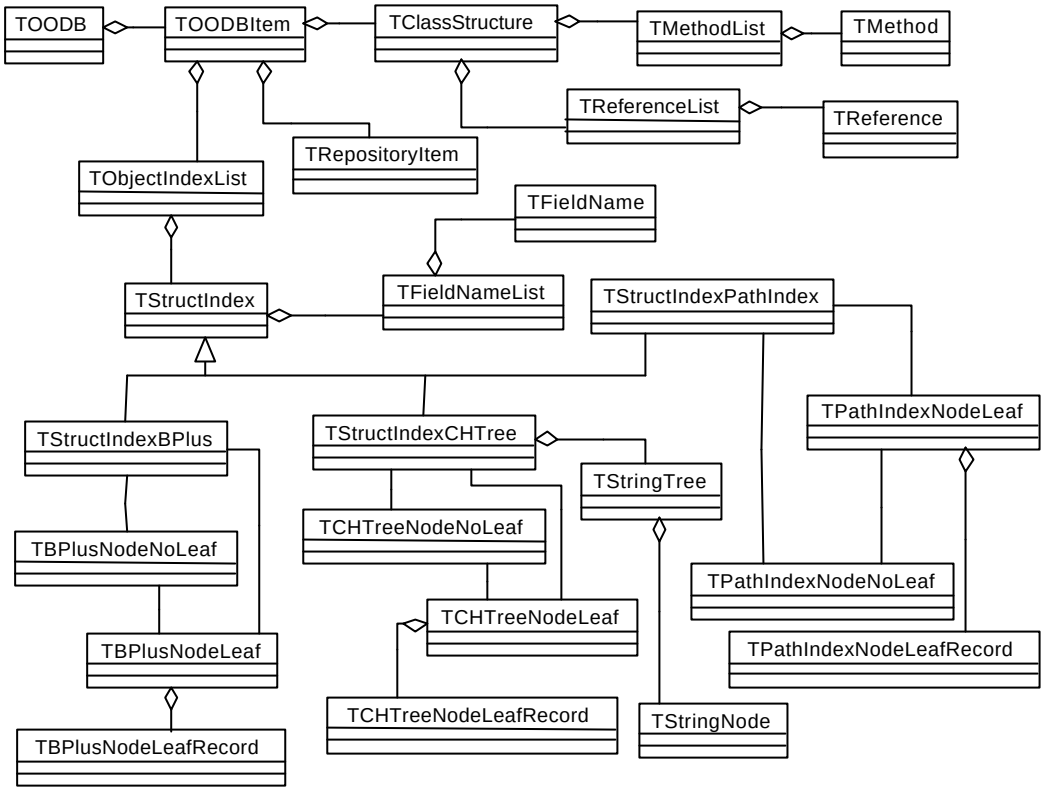


Figura 16.3. Diagrama de clases del motor

A continuación se comentarán brevemente la funcionalidad de las clases más representativas del diagrama anterior.

16.6.1 Clase TOODB

Esta clase representa el API del motor. Los métodos de esta clase suponen la interfaz de acceso a los índices, a las extensiones de las clases, y a las descripciones de las mismas. Posee un único atributo (`classes`) que es una tabla hash de elementos de tipo *TOODBItem***.** Cada uno de estos objetos almacenará información sobre la descripción de una clase, los índices y la extensión de la misma.

Los métodos proporcionados por este objeto, son en definitiva, las funcionalidades proporcionadas por el motor, y son un compendio de las funcionalidades suministradas por el resto de componentes, y que serán analizadas a continuación.

16.6.2 Clase TOODBItem

Una instancia de la clase *TOODBItem* almacena los índices, la extensión y la descripción de una única clase (*Catalog*). Mediante una instancia de *TOODBItem* se puede conocer la jerarquía de clases en la que se encuentra implicada una determinada clase, ya que mantiene referencias a las instancias de *TOODBItem* que almacenan las clases de las que hereda y también mantiene referencias a las instancias de *TOODBItem* que almacenan las clases que heredan de ella.

Como se puede observar esta clase junto con la anterior proporcionan la funcionalidad de un gestor de esquemas (como el descrito en el capítulo 10) pero ampliado, al permitir además de la funcionalidad de éste, otra serie de operaciones como por ejemplo la creación de los índices, o de las extensiones de las clases.

A continuación se van a comentar brevemente los atributos de la clase, así como algunos de los métodos más relevantes de la misma.

Atributos principales

ClassName:String;

Used:Bool; //indica si la clase ya ha sido usada en una consulta

Indexes: TObjectIndexList;

Repository: TRepositoryItem;

ClassDefinition:TClassStructure;

Fathers:TList; //clases de las que hereda la clase

Inheritors:TList; // clases que heredan de la clase

Métodos principales

AllObjects(Attributes:TAttributeList;Hierarchy:Bool):TSet

Este método retorna un conjunto con todos los objetos, o conjunto de atributos de los objetos, que se almacenan para la clase. Recibe como entrada la lista de atributos que se desean obtener y una referencia booleana que indica si se desean obtener todas las referencias de la jerarquía.

Si se consulta sólo la clase se retorna el conjunto resultante de llamar al método *AllObjects* del atributo *Repository*. Si por el contrario desea consultarse toda la jerarquía de clases se retorna el conjunto resultante de llamar al método *AllHierarchyObjects.*

AllHierarchyObjects():TSet

Este método retorna un conjunto con todos los objetos de la jerarquía de clases cuya clase raíz es la de la instancia que invoca el método.

QueryObjects(QueryFields:TQueryFieldList;Attributes:TAttributeList;Hierarchy:Bool):TSet

Este método retorna un conjunto con todos los objetos, o conjunto de atributos de los objetos, que cumplen los requisitos especificados en la consulta. Recibe como entrada los campos de consulta, la lista de atributos que se desean obtener, y una referencia booleana que indica si se desean obtener todas las referencias de la jerarquía.

Si se consulta sólo la clase se retorna el conjunto resultante de llamar al método *QueryObjects* del atributo Repository. Si por el contrario, desea consultarse toda la jerarquía de clases se retorna el conjunto resultante de llamar al método *QueryHierarchy.*

QueryHierarchy(QueryFields:TQueryFieldList):TSet

Este método retorna un conjunto con todos los objetos de la jerarquía de clases, cuya clase raíz es la de la instancia que invoca el método, que cumplen las condiciones de búsqueda recibidas como parámetro.

RetrieveObjects(anObject:Object;Operator:String;Attributes:TAttributeList;Hierarchy:Bool):TSet

Este método realiza una búsqueda de un conjunto de objetos mediante el índice por defecto de la clase. Recibe como entrada una instancia de la clase a consultar, con los campos de búsqueda completos, el tipo de comparación a realizar, la lista de atributos a obtener, y una referencia booleana que indica si se desea consultar toda la jerarquía.

Si hay que consultar sólo la clase, entonces se retorna el conjunto de objetos resultantes de invocar el método *Retrieve* del atributo Indexes. Si se desea consultar toda la jerarquía, entonces se retorna el conjunto resultado de invocar al método *InternRetrieve* de la propia clase.

RetrieveObjectsByIndex(anObject:Object;Operator:String;Order:TStringList;Attributes:TAttributeList;Hierarchy:Bool):TSet

Este método realiza una búsqueda de un conjunto de objetos mediante el índice indicado como parámetro. Recibe como entrada una instancia de la clase a consultar, con los campos de búsqueda completos, el operador de búsqueda, una lista de cadenas con el índice a utilizar, una lista con los atributos que se desean obtener, y una referencia booleana que indica si se desea consultar toda la jerarquía.

Si hay que consultar sólo la clase, entonces se retorna el conjunto de objetos resultantes de invocar el método *RetrieveByIndex* del atributo Indexes. Si se desea consultar toda la jerarquía, entonces se retorna el conjunto resultado de invocar al método *InternRetrieveByIndex* de la propia clase.

InternRetrieve(anObject:Object;Operator:String;Order:TStringList;ClassesList:TStringList):TSet

Este método consulta toda la jerarquía de clases a partir de la instancia para la que se invoca el método. Recibe como entrada un objeto de la clase a consultar con los campos de consulta del índice completos, el tipo de comparación que se desea realizar, los nombres de los atributos del índice por el que se desea consultar, y una lista de cadenas con las clases que ya han sido consultadas.

RetrieveObjectsOrderBy(QueryFields;TqueryFieldList;Order:TStringList;Attributes:TAttributeList;Hierarchy:Bool):TSet

Este método realiza una consulta sobre las instancias de una clase y retorna un conjunto ordenado por los campos especificados en *Order* (si existe un índice). Recibe como entrada la lista de campos a consultar, una lista de cadenas con el índice a utilizar, una lista con los atributos que se desean obtener, y una referencia booleana que indica si se desea consultar toda la jerarquía.

Si hay que consultar sólo la clase, entonces se retorna el conjunto de objetos resultantes de invocar el método *RetrieveOrderBy* del atributo *Indexes*. Si se desea consultar toda la jerarquía, entonces se retorna el conjunto resultado de invocar al método *InternRetrieveOrderBy* de la propia clase.

InternRetrieveOrderBy(QueryFields:TqueryFieldList;Order:TStringList;ClassesList:TStringList):TSet

Este método consulta toda la jerarquía de clases a partir de la instancia para la que se invoca el método. Recibe como entrada una lista con los campos de consulta, los nombres de los atributos del índice por el que se desea consultar, y una lista de cadenas con las clases que ya han sido consultadas.

NewIndex(FieldList:TStringList):Bool

Este método crea un nuevo índice para la clase. Recibe como entrada una lista de cadenas, que contienen los atributos a indexar. Una vez creado el índice en la clase hay que crear los índices en los hijos, para permitir la búsqueda en la jerarquía de objetos. Esto se hace recorriendo la lista de hijos, e invocando al método *NewIndex* para cada objeto.

EraseIndex(FieldList:TStringList):Bool

Este método borra un índice de la clase. Recibe como entrada los campos del índice que se desea borrar. Borra el índice de la clase si existe, y retorna cierto. Si el índice no existe se retorna falso.

SelectIndex(FieldList:TStringList):Bool

Este método establece el índice cuyos campos se reciben como parámetro como índice por defecto. Recibe como entrada los campos del índice que se desea establecer como índice por defecto. Busca el índice en la clase y si existe lo establece por defecto y retorna cierto. Si el índice no existe se retorna falso. Se retorna el resultado de la llamada al método *SetSelected* del atributo *Indexes*.

ReIndex()

Este método reordena todos los índices de la clase. Se invoca el método *ReOrder* del atributo *Indexes*.

NewObject(anObject:Object):Bool

Añade un nuevo objeto de la clase tanto a la extensión como a los índices. Recibe como entrada una instancia que es necesario almacenar. Retorna cierto si esa instancia no ha sido incluida anteriormente y la puede añadir, y falso en caso contrario.

EraseObject(anObject:Object):Bool

Borra una instancia de la clase tanto de la extensión como de los índices. Recibe como entrada la instancia que se desea borrar. Retorna cierto si la instancia existe y la pudo borrar, y falso en caso contrario.

FindObject(anObject:Object):Object

Busca una instancia en una clase. Recibe como entrada una instancia y retorna una referencia a la instancia si se encuentra almacenada en la clase, y una referencia nula en caso contrario. Para ello comprueba si la clase tiene definidos índices, y si los tiene, los emplea. Si no hay definidos índices, se utiliza la extensión para localizar el objeto.

CountObjects():Integer

Este método retorna el número de instancias de una determinada clase. Retorna el resultado de invocar el método *CountObjects* del atributo *Repository*.

CountIndexes():Integer

Retorna el número de índices de la clase. Retorna el resultado de invocar el método *IndexNumber* del atributo *Indexes*.

ProcessQueryResult(Attributes:TattributeLista;aSet:TSet)

Este método se encarga de procesar un conjunto, y comprobar si hay que retornar los objetos completos o bien el conjunto de atributos especificados en *Attributes.*

16.6.3 Clase TClassStructure

Esta clase posibilita la gestión de las clases del esquema. Contiene la descripción de una clase, almacenando la definición de los atributos y métodos de la misma.

Esta clase ha tenido que ser incorporada debido a que la versión de la máquina abstracta sobre la que ha sido construido este prototipo no permite la interrogación del área de clases.

Atributos principales

AttributeList:TReferenceList; //almacena las definiciones de los atributos

MethodList:TMethodList; // almacena las definiciones de los métodos

Métodos principales

AddAttribute (aName:String;aType:String)

Añade la definición de un atributo a la definición de la clase. Recibe como entrada el nombre del atributo y el tipo.

FindAttribute (aName:String):Treference

Busca un atributo en la definición de la clase. Recibe el nombre del atributo a buscar, y retorna copia del atributo si existe, y una referencia nula en caso contrario.

EraseAttribute (aName:String):Bool

Borra un atributo de la definición de la clase. Recibe como entrada el nombre del atributo a borrar y lo borra de la definición de atributos de la clase. Retorna cierto si existe y lo puede borrar, y falso en caso contrario.

AttributesCount():Integer

Retorna un entero que indica el número de atributos de la clase.

AddMethod (aName:String;aResult:String;Parameters:TReferenceList)

Añade un nuevo método a la definición de la clase. Recibe como entrada el nombre del método, el tipo de resultado que retorna, y una lista con los parámetros de entrada del método.

FindMethod (aName:String):Tmethod

Comprueba la existencia de un método en la definición de la clase. Recibe como entrada el nombre del método a buscar. Retorna una copia de la definición del método si existe, y una referencia nula en caso contrario.

EraseMethod(aName:String):Bool

Borra un método de la definición de la clase. Recibe como entrada el nombre del método a borrar y lo borra de la definición de métodos de la clase. Retorna cierto si existe y lo puede borrar, y falso en caso contrario.

MethodsCount():Integer

Retorna un entero que indica el número de métodos existente en la definición de la clase.

16.6.4 Clase TRepositoryItem

Esta clase contiene la extensión de la clase (almacén). Almacena por lo tanto todas las referencias a los objetos de la clase. Representa el gestor de extensiones (*Repository*).

Atributos principales

Objects:TOrderListByID; // lista de objetos ordenados por su IDO

Métodos principales

EraseObject (Element:Object):Bool

Este método recibe como entrada un objeto y lo borra de la lista de objetos que se encuentra almacenada en el atributo *Objects.* Retorna cierto si existe y lo pudo borrar, y falso en caso contrario.

FindObject (Element:Object):Object

Este método recibe como entrada un objeto y lo localiza en la estructura que representa la extensión de la clase. Retorna una referencia al objeto si lo encuentra, y una referencia nula en caso contrario.

InsertObject (Element:Object):Bool

Inserta el objeto que recibe como parámetro en la estructura que representa la extensión. Retorna cierto si lo puede añadir, y falso en caso contrario.

AllObjects():TSet

Este método devuelve un conjunto con todos los objetos almacenados en la extensión. Si no contiene ningún objeto retorna un conjunto vacío.

QueryObjects(QueryFields:TQueryFieldList):TSet

Este método realiza una consulta sobre la extensión de la clase. Recibe como entrada los campos de la consulta, y busca los objetos que cumplan las condiciones de búsqueda. Si no existe ningún objeto que cumpla las condiciones de búsqueda se retorna un conjunto vacío.

CountObjects():Integer

Este método devuelve el número de objetos que se encuentran almacenados.

16.6.5 Clase TObjectIndexList

Esta clase almacena el conjunto de índices de una determinada clase y se encarga del mantenimiento de los mismos.

Atributos principales

ListIndex:TList;

Selected: Integer; // especifica el índice por defecto para una clase

Métodos principales

IndexNumber():Integer

Este método devuelve el número de índices que contiene la clase.

NewIndex(anIndex:TStructIndex):Bool

Este método recibe como entrada una estructura de almacenamiento de índices y la añade a la lista de índices de la clase. Devuelve cierto si pudo añadir el nuevo objeto a la lista de índices, y falso en caso contrario.

EraseIndex(ClassName:String;FieldList:TStringList):Bool

Este método recibe como entrada el nombre de la clase de la que se borra el índice (se utiliza en el caso de que el índice admita jerarquía), y una lista de cadenas que representan los atributos de ordenación de un índice. Borra el índice cuyos atributos de ordenación coincidan.

GetStructIndex(Position:Integer):TStructIndex

Este método recibe como entrada un entero que identifica una posición en la lista de índices de la clase, y devuelve una referencia a la estructura de almacenamiento que se encuentra en esa posición de la lista.

RetrieveIndex(FieldList:TStringList):Integer

Este método busca un índice en la lista de índices. Recibe como entrada una lista de cadenas con los campos del índice que se busca. Devuelve la posición en la que se encuentra el índice con esos campos en la lista de índices, o -1 si no lo encuentra.

Insert(ClassName:String;NewObject:Object):Bool

Añade el objeto a todos los índices de la clase. Este método recibe como entrada el nombre de la clase del objeto, y el objeto. Devuelve cierto si lo pudo hacer, y falso en caso contrario.

Erase(ClassName:String;NewObject:Object):Bool

Este método recibe como entrada el nombre de la clase del objeto a borrar, y el objeto a borrar. Elimina el objeto que recibe como entrada de todos los índices de la clase. Devuelve cierto si lo pudo hacer y falso en caso contrario.

Reorder():Bool

Este método reordena todos los índices de la clase.

SetSelected(OrderList:TstringList):Bool

Este método se encarga de establecer el índice de búsqueda por defecto.

Retrieve(ClassName:String;NewObject:Object;Operator:String;ClassesList:TStringList;Hierarchy:Bool):TSet

Este método realiza una consulta basándose en el índice por defecto. Recibe como entrada el nombre de la clase que se desea consultar (por si el índice admite jerarquía), el objeto con los datos que hay que buscar, el tipo de comparación a realizar, la lista de clases consultadas hasta el momento, y una

variable booleana que indica si hay que buscar todos los objetos de la jerarquía cuya clase raíz es la del objeto para el cual se invoca el método.

RetrieveByIndex(ClassName:String;NewObject:Object;Operator:String;Order:TStringList;ClassesList:TStringList;Hierarchy:Bool):TSet

Este método realiza una consulta en el índice seleccionado como parámetro. Recibe como entrada el nombre de la clase que se desea consultar (por si el índice admite jerarquía), el objeto con los datos que hay que buscar, el tipo de comparación a realizar, la lista de cadenas que identifica el índice a consultar, la lista de clases consultadas hasta el momento, y una variable booleana, que indica si hay que buscar todos los objetos de la jerarquía cuya clase raíz es la del objeto para el cual se invoca el método.

RetrieveOrderBy(ClassName:String;QueryFields:TQueryFieldList;Order:TstringList;ClassesList:TstringList;Hierarchy:Bool):TSet

Este método realiza una consulta en el índice por el que se quieren obtener ordenados los objetos. Recibe como entrada el nombre de la clase que se desea consultar (por si el índice admite jerarquía), los campos de consulta, la lista de cadenas que identifica el índice a utilizar, la lista de clases consultadas hasta el momento, y una variable booleana, que indica si hay que buscar todos los objetos de la jerarquía cuya clase raíz es la del objeto para el cual se invoca el método.

16.6.6 Clase TStructIndex

Esta clase es abstracta y se utiliza como plantilla para la implementación de nuevas estructuras de almacenamiento de índices. Contiene una serie de métodos que son comunes a todas las estructuras de almacenamiento y que deben ser utilizados por cualquiera de éstas. Cualquier nueva clase que represente una estructura de almacenamiento de índices, debe derivarse de ésta.

Atributos principales

Fields:TFieldNameList; //campos del índice

AllowHierarchy:Bool; // la estructura admite jerarquías de herencia

StructIndexName: String; //nombre de la técnica

Métodos principales

Insert(ClassName:String;NewElement:Object):Bool

Este método recibe como entrada el nombre de la clase del objeto que se inserta, y el objeto que se inserta. Devuelve cierto si lo pudo insertar y falso en caso contrario.

Erase(ClassName:String;NewElement:Object):Bool

Este método recibe como entrada el nombre de la clase del objeto que se pretende borrar y el objeto que se desea borrar. Devuelve cierto si lo pudo borrar y falso en caso contrario.

Reorder():Bool

Este método reordena el índice.

SetFields(NewFields:TStringList)

Este método recibe como entrada una lista de cadenas que serán los nombres de los atributos de ordenación.

Retrieve(ClassName:String;NewElement:Object;Operator:String;ClassesList:TStringList;AllObjects:Bool):TSet

Este método recibe como entrada el nombre de la clase que se desea consultar, un objeto con los campos definidos con los valores a consultar, un parámetro que se utiliza para almacenar las clases consultadas, y un valor booleano que indica si hay que consultar la jerarquía completa. Devuelve un conjunto con los objetos encontrados. Si no encuentra ningún objeto entonces se retorna un conjunto vacío. Se pasa como parámetro el nombre de la clase, por si acaso la estructura de almacenamiento admite jerarquía.

RetrieveAll(ClassName:StringClassesList:TStringList;AllObjects:Bool):TSet

Este método recibe como entrada el nombre de la clase que se desea consultar, un parámetro que se utiliza para almacenar las clases consultadas y un valor booleano que indica si hay que consultar la jerarquía completa. Devuelve un conjunto con todos los objetos del índice. Si no encuentra ningún objeto entonces se retorna un conjunto vacío.

NewHierarchyClass(ClassName:String;Fathers:TStringList):Bool

Este método sólo tiene sentido en aquellas estructuras de almacenamiento que admitan jerarquía de herencia. Si la estructura no admite jerarquía debe ignorarse. Este método se ejecuta cuando se está creando un nuevo índice para una clase y se va a almacenar en la misma estructura que el índice de la clase padre. Recibe como entrada el nombre de la nueva clase que se va a almacenar y la lista de padres de la que hereda. Devuelve cierto si pudo procesar la información y falso en caso contrario.

Count(ClassName:String):Integer

Devuelve el número de objetos insertados en el índice para una determinada clase.

16.6.7 Clases para la implementación de la técnica SC

- **TStructIndexBPlus.** Esta clase representa una estructura de almacenamiento en forma de árbol B+. Mantiene referencias a instancias de las clases que representan los nodos hoja, *TBPlusNodeLeaf*, y nodos internos del árbol, *TBPlusNodeNoLeaf*.
- **TBPlusNodeNoLeaf.** Esta clase representa los nodos no hoja de una estructura de almacenamiento de tipo B+. Esta clase es utilizada por *TStructIndexBPlus.*
- **TBPlusNodeLeaf.** Esta clase representa los nodos hoja de una estructura de almacenamiento de tipo B+. Esta clase es utilizada por *TStructIndexBPlus.*

- **TBPlusNodeLeafRecord.** Esta clase representa cada uno de los elementos que se almacenan en un nodo hoja de un árbol B+. En una instancia de esta clase se almacena el valor clave que representa, y el conjunto de instancias cuyo valor, para el índice en el que se encuentran, coincide con el valor de la clave.

16.6.8 Clases para la implementación de la técnica CH-Tree

- **TStructIndexCHTree.** Esta clase representa una estructura de almacenamiento en forma de árbol CH-Tree. Mantiene referencias a instancias de las clases que representan los nodos hoja, *TCHTreeNodeLeaf*, y los nodos internos del árbol, *TCHTreeNodeNoLeaf*.
- **TCHTreeNodeNoLeaf.** Esta clase representa los nodos no hoja de una estructura de almacenamiento de tipo CH-Tree. Es utilizada por *TStructIndexCHTree.*
- **TCHTreeNodeLeaf.** Esta clase representa los nodos hoja de una estructura de almacenamiento de tipo CH-Tree. Es utilizada por *TStructIndexCHTree.*
- **TCHTreeNodeLeafRecord.** Esta clase representa cada uno de los elementos que se almacenan en un nodo hoja de un árbol CH-Tree. En una instancia de esta clase se almacena el valor clave que representa, y el conjunto de instancias cuyo valor, para el índice en el que se encuentran, coincide con el valor de la clave.
- **TStringTree.** Esta clase representa un árbol de cadenas con los nombres de las clases que se almacenan en el árbol, organizadas en función de su jerarquía. Se utiliza en el árbol CH-Tree para poder conocer en todo momento la jerarquía de clases que realmente se almacena en él. Para la representación de los nodos se utilizan instancias de la clase *TStringNode*.
- **TStringNode.** Representa los nodos del árbol de cadenas. Almacena además de la cadena las referencias tanto a los padres del nodo como a los herederos.

16.6.9 Clases para la implementación de la técnica Path Index

- **TStructIndexPathIndex.** Esta clase representa una estructura de almacenamiento en forma de árbol Path Index. Mantiene referencias a instancias de las clases que representan los nodos hoja, *TPathIndexNodeLeaf,* y los nodos internos del árbol, *TPathIndexNodeNoLeaf.*
- **TPathIndexNodeNoLeaf.** Esta clase representa los nodos no hoja de una estructura de almacenamiento de tipo Path Index. Es utilizada por *TStructIndexPathIndex.*
- **TPathIndexNodeLeaf.** Esta clase representa los nodos hoja de una estructura de almacenamiento de tipo Path Index. Es utilizada por *TStructIndexPathIndex.*
- **TPathIndexNodeLeafRecord.** Esta clase representa cada uno de los elementos que se almacenan en un nodo hoja de un árbol Path Index. En una instancia de esta clase se almacena el valor clave que representa, y el conjunto de instancias cuyo valor, para el índice en el que se encuentran, coincide con el valor de la clave.

16.7 Descripción dinámica del prototipo

A continuación se muestra el funcionamiento básico del motor en forma de diagramas de colaboración para las operaciones más habituales.

16.7.1 Gestión de instancias

Para la inserción de una instancia, el objeto *OODB* recibe la petición de inserción. En primer lugar se localiza la clase en la que se realizará la operación. Una vez localizada la instancia de *TOODBItem* que almacena la clase, se inserta en la extensión *RepositoryItem* y en los índices *ObjectIndexList*. El objeto *ObjectIndexList* se encargará de añadir esa instancia a todos los índices definidos para la clase (si es que hay alguno).

Para la eliminación de una instancia el objeto *OODB* recibe la petición de borrado, y se procede de igual forma. En primer lugar se localiza la clase en la que se realizará la operación. Una vez localizada la instancia de *TOODBItem* que almacena la clase, se borra la instancia de la extensión (*RepositoryItem)* y de los índices *(ObjectIndexList).* El objeto *ObjectIndexList* se encargará de eliminar esa instancia de cada uno de los índices definidos para la clase.

También es posible la localización de un objeto. Para ello, primero se localiza la clase en la que se realizará la operación para a continuación invocar el método *FindObject* de *RepositoryItem* que comprueba si la instancia existe.

Para realizar un recuento del número de instancias existentes de una determinada clase se invoca el método *CountObjects* de *RepositoryItem*, previa localización de la instancia *OODBItem* que almacena la clase**.**

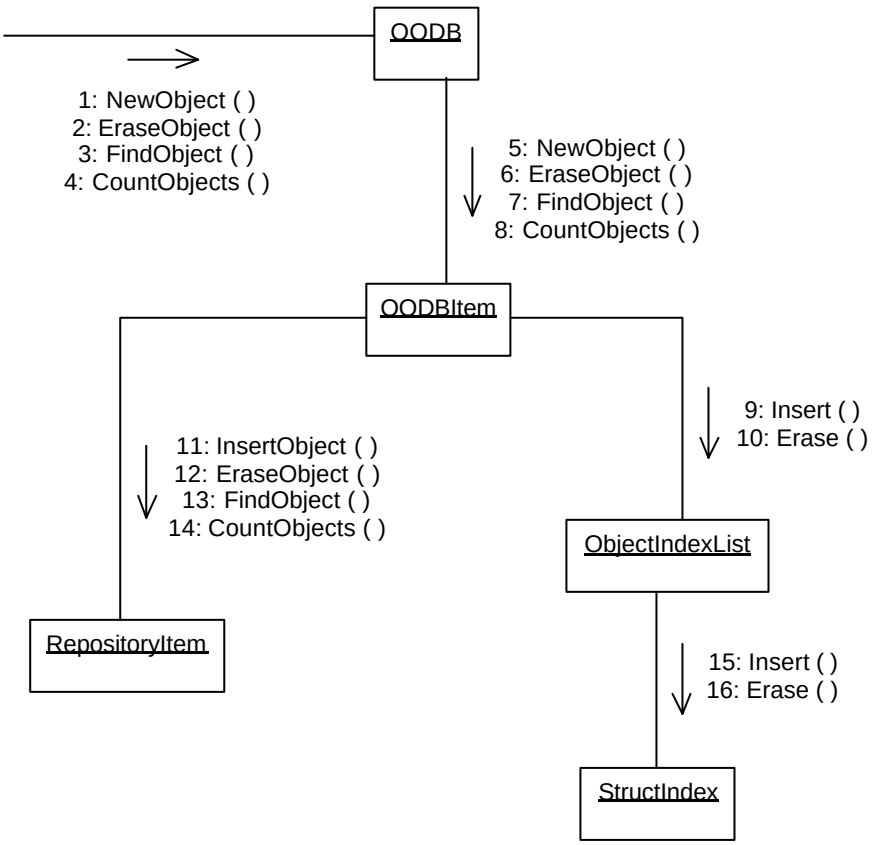


Figura 16.4. Diagrama que refleja la gestión de instancias

16.7.2 Gestión de clases

Para la inserción de una nueva clase se pueden utilizar dos métodos: *InsertClass*, que añade una clase base, e *InsertClassWithHierarchy*, que define una clase derivada de otra ya existente. El funcionamiento es, básicamente el mismo. Se recibe el mensaje en el objeto *OODB* y se crea una nueva instancia de *TOODBItem* que se añade. Si se utiliza el método *InsertClassWithHierarchy,* además se crean los índices que se hayan definido en las clases de las que herede.

Para la eliminación, el objeto OODB localiza la instancia de *TOODBItem* a borrar y la elimina. El método empleado para localizar la instancia de *TOODBItem* es *FindClass*.

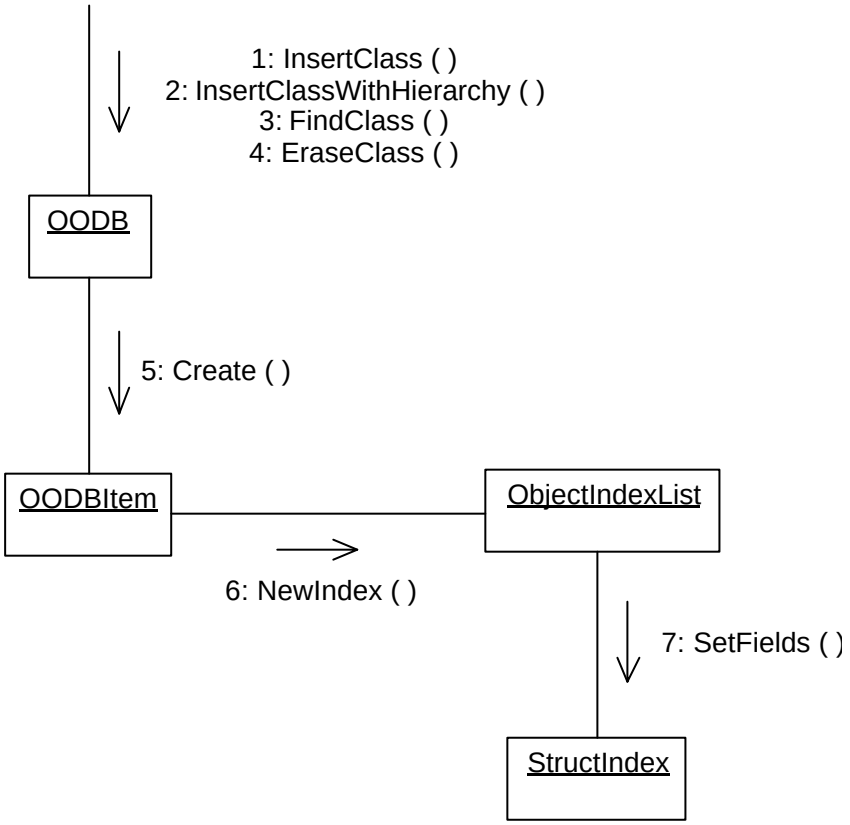


Figura 16.5. Diagrama que refleja la gestión de las clases

Para el almacenamiento del descriptor de la clase todos los métodos que se reciben en el objeto OODB se transmiten al atributo de tipo *TClassStructure* de la instancia de tipo *TOODBItem*.

16.7.3 Gestión de índices

Para la creación de un nuevo índice, el objeto *OODB* recibe la petición. La primera operación es localizar el objeto *OODBItem* en el que se va a realizar la operación. Una vez localizado se comprueba la configuración para la creación de un nuevo índice. Se crea el nuevo índice y se añade a la instancia *ObjectIndexList*.

Para el borrado de un índice, se localiza la clase en la que se va a producir la eliminación y se invoca el método *EraseIndex* del objeto *ObjectIndexList*.

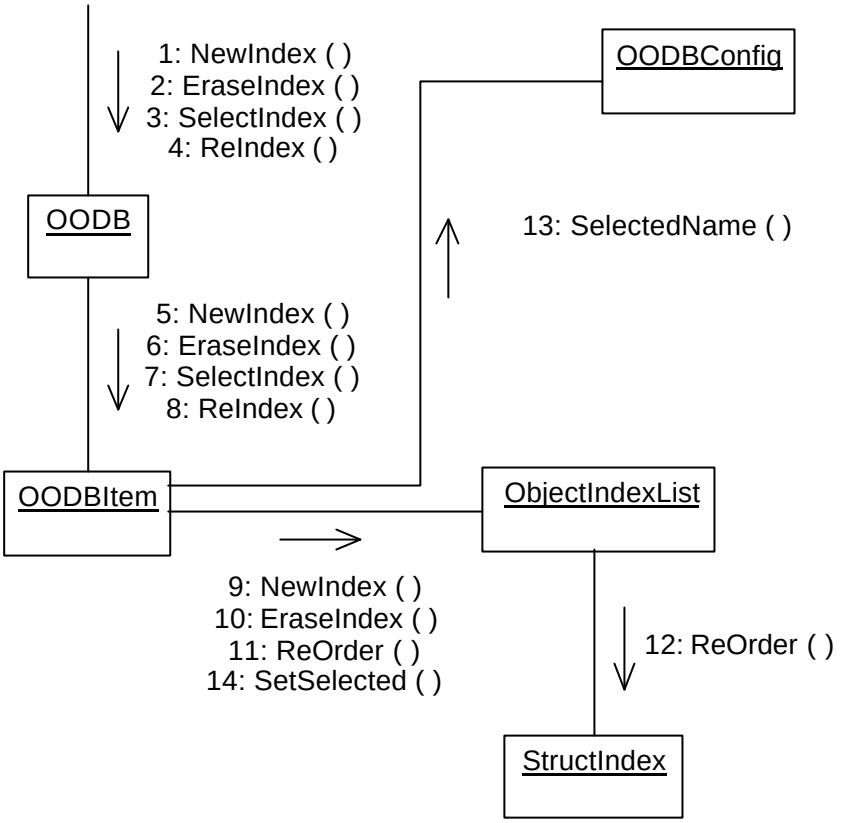


Figura 16.6. Diagrama que refleja la gestión de índices

El establecimiento de un índice por defecto se realiza localizando primero la instancia *OODBItem* de la que se desea cambiar el índice por defecto. Una vez hecho esto se llama al método *SetSelected* del objeto *ObjectIndexList*.

La reordenación de los índices de una clase se realiza, una vez localizada la instancia, invocando el método *ReOrder* del objeto *ObjectIndexList*. Mediante este método se recorren todos los índices de la lista y para cada uno se invoca el método *ReOrder*.

16.7.4 Realización de consultas elementales

Las consultas al sistema pueden realizarse basándose en los mecanismos de indexación o directamente sobre la colección que implementa la extensión de la clase.

El método *QueryJoin* realiza las consultas sobre varias clases sin emplear ningún mecanismo de indexación. El método *AllObjects* realiza consultas sobre la extensión de la clase. Una vez localizada la instancia *OODBItem* en la que se realizará la consulta se invoca el método *AllObjects* de la extensión. De forma similar funciona el método *QueryObjects.*

Mediante el método *RetrieveObjects* se realizan consultas en el índice por defecto de la clase. En primer lugar se localiza la instancia *OODBItem* que almacena la clase. Se invoca el método *Retrieve* del objeto *ObjectIndexList* que localiza el índice por defecto y realiza una llamada al método *Retrieve* del objeto *StructIndex* que representa el índice por defecto.

Mediante el método *RetrieveObjectsByIndex*, se realizan consultas por cualquier índice de la clase. Primero se localiza la instancia *OODBItem* que almacena la clase. Se

invoca el método *RetrieveByIndex* del objeto *ObjectIndexList*, que se encarga de localizar el índice solicitado, y realiza una llamada al método *Retrieve* del objeto *StructIndex* que representa el índice solicitado.

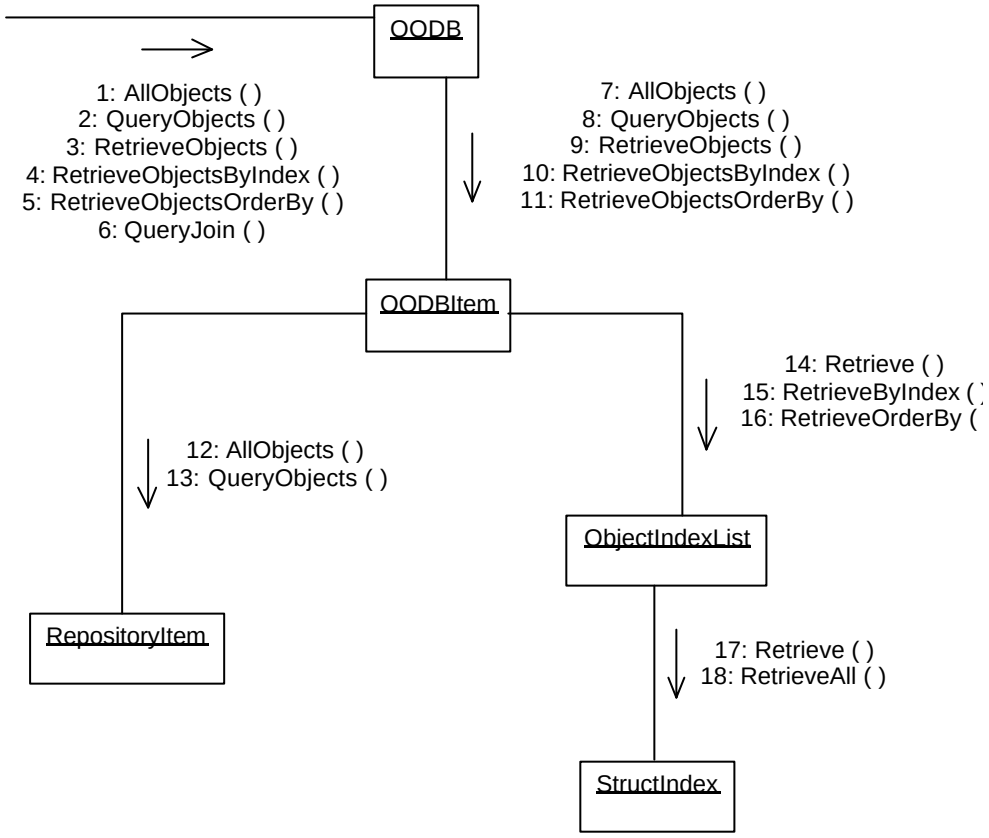


Figura 16.7. Diagrama asociado a las consultas

Mediante el método *RetrieveObjectsOrderBy* se realizan consultas ordenadas por índices. Primero se localiza la instancia de *OODBItem* que almacena la clase. Se invoca el método *RetrieveObjectsOrderBy* que se encarga de localizar el índice por el que se desea ordenar. Una vez localizado se obtienen todas las instancias de la clase mediante *RetrieveAll* y se realiza un filtrado para comprobar cuales cumplen las condiciones de búsqueda

16.8 Extensibilidad del mecanismo

Una vez analizado el mecanismo de indexación diseñado, se van a puntualizar las operaciones a realizar para la extensión del sistema con una nueva técnica de indexación. Para ello se consideraran dos aspectos:

- La incorporación de nuevas técnicas de indexación
- La selección de la técnica de indexación a emplear.

16.8.1 Incorporación de nuevas técnicas de indexación

El prototipo permite incorporar nuevas técnicas de indexación sin necesidad de modificar el código fuente de la jerarquía de clases. Los pasos que hay que realizar para ello se detallan a continuación.

En primer lugar, cualquier clase que represente una técnica de indexación debe heredar de la clase *TStructIndex* e implementar los métodos que se enumeran a continuación para mantener la compatibilidad con las estructuras ya definidas.

- **Free**. Libera el índice
- **Insert.** Inserta una nueva instancia en el índice.
- **Erase.** Borra una instancia del índice.
- **Retrieve.** Realiza una consulta en el índice.
- **RetrieveAll**. Devuelve un conjunto con todas las instancias almacenadas en el índice.
- **ReOrder.** Reordena las instancias del índice.
- **NewHierarchyClass**. Se le indica que va almacenar una clase (sólo en el caso de que la estructura admita almacenamiento de jerarquías de clases).
- **Count.** Retorna el número de instancias en el índice.

Como se comentó al principio, este prototipo permite la indexación sobre cualquier tipo de dato, no sólo datos primitivos. Para ello en el prototipo diseñado es obligatorio definir los métodos *Less*, *Equal* y *Greater* para cada clase no primitiva que sea susceptible de ser indexada.

16.8.2 Registro y selección de la técnica de indexación

Una vez implementado el nuevo tipo de índice ya sólo falta registrarlo para que pueda ser utilizado por el objeto motor. Para registrar una nueva técnica de indexación hay que añadir una nueva instancia al objeto *OODBConfig*, de tipo *TOODBConf*, que es la clase que almacena los diferentes tipos de índices que se pueden utilizar en el sistema.

Los pasos a realizar son los siguientes:

- En primer lugar, deberá definirse una clase, que definirá las características del nuevo tipo de índice, y que deriva de la clase *TStructIndexConf*. Esta nueva clase simplemente definirá el método *NewObject*, que retornará una instancia de la clase que representa el nuevo tipo de índice.
- Una vez definida esta clase, hay que añadir una instancia de esta clase al objeto *OODBConfig* junto con el nombre de la técnica de indexación. De esta forma ya estará registrada la nueva clase. Esta clase *OODBConfig*, proporciona una serie de métodos que entre otras posibilidades nos permite seleccionar la técnica de indexación a emplear (*SelectByName, SelectByIndex*).

Capítulo 17 Introducción de los lenguajes en el SGBDOO

Para cualquier SGBDOO es necesario un lenguaje que permita manipular y consultar la base de datos. En el caso de BDOviedo3 también y, como se justificó en el capítulo 13, se ha optado por el *binding* Java de ODMG para ello.

En este capítulo se describen algunas matizaciones realizadas sobre el *binding* Java a emplear, y se describe una herramienta que permite generar código para diferentes gestores de objetos compatibles con el *binding* Java de ODMG con el fin de realizar comparaciones entre ellos.

17.1 Incorporación de los lenguajes al SGBDOO

Los lenguajes seleccionados para el SGBDOO BDOviedo3 son, tal y como se comentó en el capítulo 13, los propuestos por el estándar ODMG 2.0 en su *binding* para Java.

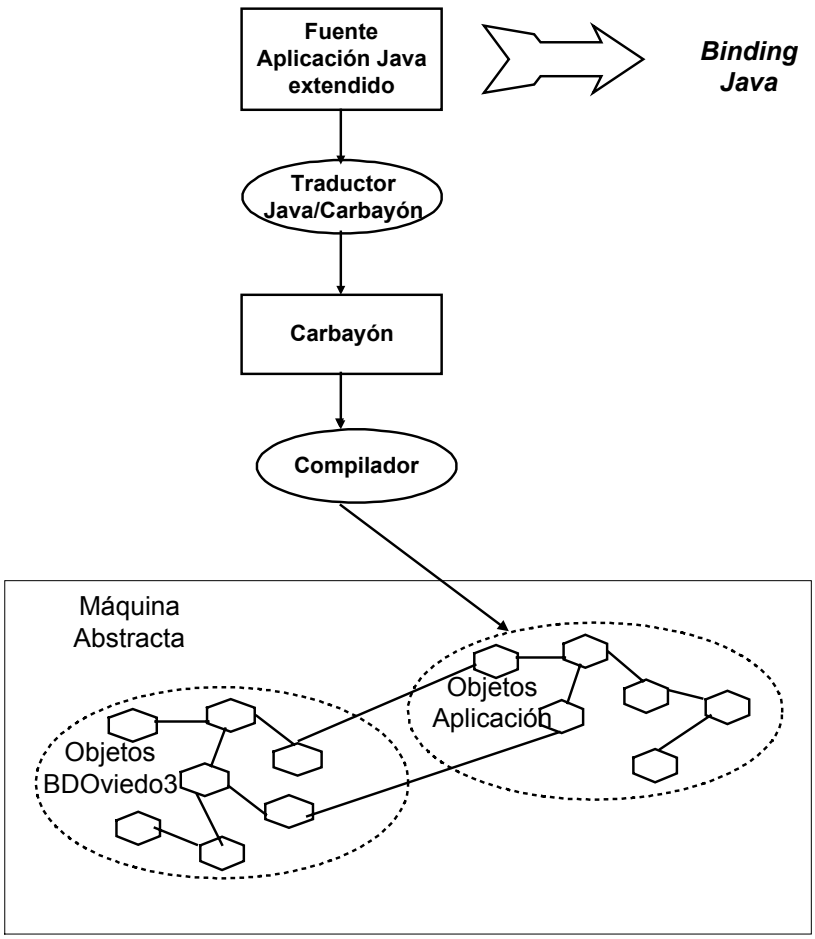


Figura 17.1. Elementos necesarios para la incorporación de los lenguajes a BDOviedo3

Para incorporar los lenguajes a BDOviedo3 es entonces imprescindible contar con un traductor que a partir de las especificaciones realizadas en Java extendido (para dar soporte al *binding* ODMG) genere las instrucciones correspondientes en el lenguaje de la máquina abstracta, que en el prototipo implementado es el lenguaje Carbayón. Una vez traducida la aplicación y compilada (con el compilador de la máquina) los objetos de la aplicación ya podrán hacer uso de los servicios proporcionados por el motor implementado.

De esto se deduce, que el único elemento realmente imprescindible para la incorporación de los lenguajes en el sistema es un traductor de Java ampliado a Carbayón. Dicho traductor no ha podido ser incorporado en esta memoria porque está en proceso de finalización, sin embargo, si se pueden especificar determinadas observaciones que han sido consideradas sobre el *binding* Java, así como las gramáticas definitivas resultantes (véase Apéndice C) que son soportadas por dicho traductor.

17.1.1 Características básicas del traductor Java/Carbayón

A continuación se enumeran brevemente las características principales a considerar a la hora de construir el traductor de Java a Carbayón.

17.1.1.1 Java *binding* completo

El traductor debe reconocer la totalidad del código Java más las clases e interfaces propias del *binding* sobre Java de ODMG.

17.1.1.2 Aprovechamiento de las características de la OO para su construcción

El traductor y el entorno han de ser construidos empleando el paradigma de orientación a objetos para aprovechar entre otras las posibilidades de extensibilidad que éste proporciona.

17.1.1.3 Portabilidad

El entorno ha de ser realizado de forma que se facilite su portabilidad a diferentes plataformas. Para ello, ha de emplearse un lenguaje de programación como puede ser Java, que facilita dicha portabilidad.

17.2 Observaciones sobre el *binding* Java implementado

El traductor diseñado trabaja con el *binding* Java completo, sin embargo, se han realizado algunas extensiones sobre todo en aspectos en los que el *binding* Java no se pronuncia. En este apartado se tratarán en detalle, las extensiones que se van añadir al lenguaje de soporte (Java).

Las nuevas palabras reservadas que se añaden al conjunto de palabras reservadas de Java, se les antepondrá el carácter "_", para evitar la coincidencia con otros métodos, campos o clases que hayan sido programadas anteriormente, y que no podrían ser tratados debido a que estarían compuestos por palabras reservadas a los ojos del traductor.

17.2.1 Claves

ODMG, propone utilizar la palabra reservada *key/keys*, para indicar que atributos se utilizarán como claves para identificar unívocamente a un objeto. Aunque en el *binding* Java de la versión 2.0 no aparece esta característica, aquí se autoriza su uso, por

ejemplo, con el fin de servir de utilidad a la hora de describir claves en tablas relacionales si es que el motor a utilizar es relacional.

Se utilizarán las palabras reservadas **_key, _keys**, dentro de paréntesis, justo antes del comienzo del cuerpo de la clase.

(... _key name_atributo_key (, name_atributo_key))* o para claves compuestas

(... _keys (name_attributo_key , name_attributo_key) +)

17.2.2 Estructuras

Las estructuras o *structs* son una forma de agrupar en un objeto atributos o tipos de datos que tienen algún tipo de utilidad en una aplicación. En el *binding* Java, no se les hace referencia, y aquí se simularán mediante clases Java. La sintaxis de uso siguiendo una notación BNF es:

```
_struct name {
        (tipo_de_dato id;)+
};
```

17.2.3 Excepciones

ODMG propone una forma sencilla de definir excepciones para utilizar con las clases persistentes y sus operaciones, sin embargo, en el *binding* Java no se referencia. Aquí serán simuladas en clases Java extendiendo la clase *RuntimeException*. La sintaxis de uso siguiendo una notación BNF es:

```
_exception name {
        (tipo_de_dato id;)*
};
```

17.2.4 Extents

El *binding* Java declara como responsabilidad del programador definir una colección que represente la extensión de la clase, y los métodos para mantenerla.

La extensión (*extent*) de una clase se suele declarar después de la cabecera de la clase y las interfaces implementadas, y antes de la llave que indica el comienzo del cuerpo de la clase. La forma de declarar un *extent*, es opcional, y aquí se ha optado por la siguiente:

(_extent name_extent ...)

17.2.5 Índices

Aunque ODMG no los tiene en cuenta en su lenguaje de definición, se refiere a ellos como una forma de acelerar el acceso a las instancias que constituyen la extensión de una clase. Sin embargo, aquí por el carácter experimental de este prototipo y por la importancia que se le ha dado a los mecanismos de indexación en la implementación del motor se ha decidido su incorporación. Su definición va ligada a la definición de la extensión de la clase de la siguiente forma:

(_extent name_extent _index name_index (name_index)+)

o para índices compuestos de varios atributos :

(_extent name_extent _index (name_index , name_index) +)

17.2.6 Modificadores

Los modificadores aquí presentados son permitidos por motivos de compatibilidad y posibles cambios en futuras versiones.

- **OneWay.** Se aplica sobre el tipo de retorno de las operaciones, en este caso sobre el tipo de retorno de los métodos. Se antepondrá al tipo de dato retornado como *_oneway*.
- **In.** Se aplica sobre el tipo de los parámetros en las operaciones, en este caso sobre el tipo de dato del parámetro en los métodos. Indica que los datos van a ser tratados en la operación o método por su valor. Se antepondrá al tipo de dato como *_in.*
- **Out.** Se aplica sobre el tipo de los parámetros en las operaciones, en este caso sobre el tipo de dato del parámetro en los métodos. Indica que un dato va a ser tratado en la operación o método y va a retornar un valor. Se antepondrá al tipo de dato como *_out*.
- **InOut.** Se aplica sobre el tipo de los parámetros en las operaciones, en este caso sobre el tipo de dato del parámetro en los métodos. Indica que un dato va a ser tratado en la operación o método, siendo modificado su valor inicial y retornando un nuevo valor. Se antepondrá al tipo de dato como *_inout*.

17.2.7 Módulos

Los módulos son una forma de agrupar partes o la descripción completa de la base de datos para poder ser consultada por otras aplicaciones, y así reconstruir la base de datos en función de su descripción interna. Se permite su utilización también por motivos de compatibilidad y cambios en futuras versiones. La forma de declararlo es:

_module nombre-modulo {

definiciones de clases, interfaces, etc.

}

17.2.8 Clase *Database*

El *binding* Java propone utilizar objetos de la clase *Database* para manejar bases de datos en Java. Esta clase permite a las aplicaciones manejar objetos dentro de la base de datos. La forma de crear un objeto y aplicar sus métodos, se hará siguiendo la sintaxis Java, para objetos y métodos tradicionales.

```
public abstract class Database{

      public static final int notOpen = 0;
      public static final int openReadOnly = 1;
      public static final int openReadWrite = 2;
      public static final int openExclusive = 3;
      protected static Object obj;

      public static Database open(String name, int accessMode) throws
DatabaseNotFoundException {
      return (Database) obj; }
      public abstract void close() throws ODMGException;

      public abstract void bind(Object obj, String name)
                           throws  TransactionNotInProgressException;

      public abstract Object lookup(String name)
                           throws  ObjectNameNotFoundException;
      public abstract void unbind(String name)
                           throws  ObjectNameNotFoundException,
                                                TransactionNotInProgressException;
      public abstract Database create(String name, int createMode) throws
ODMGException;

}
```

17.2.9 Clase *Transaction*

El Java *binding* propone utilizar objetos de la clase *Transaction* para manejar bases de datos en Java y asegurar la integridad de la misma. La forma de crear un objeto y aplicar sus métodos, se hará siguiendo la sintaxis Java, para objetos y métodos tradicionales.

```
public abstract class Transaction {

      public static final int READ = 1,UPGRADE = 2,WRITE = 3;
      protected static Object obj;

      public Transaction(){ }

      public abstract void join();

      public abstract void leave();

      public static Transaction current(){
                           return (Transaction) obj;  }

      public abstract void begin()
                           throws TransactionNotInProgressException,
                                        TransactionInProgressException;

      public abstract boolean isOpen();

      public abstract void commit() throws TransactionNotInProgressException;

      public abstract void abort() throws TransactionNotInProgressException;

      public abstract void checkpoint() throws TransactionNotInProgressException;

      public abstract void lock( Object obj, int mode) throws LockNotGrantedException;

}
```

17.2.10 Relaciones

El *binding* para Java en su versión 2.0 no se pronuncia acerca de las relaciones, es decir, no ofrece ningún tipo de solución a las relaciones en ninguna de sus cardinalidades.

Aquí se ha optado por permitir su utilización realizando su definición en la parte de declaraciones de métodos y propiedades de la clase, nunca dentro de métodos o cuerpos estáticos. Se seguirá la siguiente sintaxis:

- Para relaciones de grado 1
  _relationship nombreclaserelacionada nombrerelacion;
- Para relaciones de grado n
  _relationship TipoColección < nombreclaserelacionada > nombrerelacion;
  dónde TipoColección será: *Set* , *List* , *Bag* o *Array*

Para establecer integridades referenciales entre clases se acude a los atributos inversos:

_inverse nombreclaserelacionada :: nombrerelacionenclaserelacionada ;

17.2.11 Tipos de datos

ODMG propone una serie de tipos de datos, como tipos primitivos o básicos. En el *binding* para Java es necesario añadir a los tipos primitivos de Java unos nuevos que tienen su correspondencia con tipos soportados por Java. Se permite su uso simplemente por motivo de compatibilidad con herramientas que puedan utilizar estos tipos.

- **Any**. Tipo de dato que como su nombre indica, acepta a cualquier tipo existente. La correspondencia con Java es *Object*. Un identificador o método de este tipo se declarará con la palabra reservada *_any*.
- **Octet**. La correspondencia con Java es *byte*, y representa un octeto. Un identificador o método de este tipo se declarará con la palabra reservada *_octet*.
- **U_Long.** Java no reconoce el tipo *long* sin signo, o lo que es lo mismo *unsigned long*, por lo que su correspondencia será con el tipo con signo, es decir, *long*. Un identificador o método de este tipo se declarará con la palabra reservada *_u_long*.
- **U_Short**. Java no reconoce el tipo *short* sin signo, o lo que es lo mismo *unsigned short*, por lo que su correspondencia será con el tipo con signo, es decir, *short*. Un identificador o método de este tipo se declarará con la palabra reservada *_u_short*.

17.3 Herramienta para la experimentación

Siguiendo con la idea de plataforma de experimentación con la que nace este sistema, sería interesante contar con un entorno que permitiera comparar el rendimiento del sistema aquí diseñado con diferentes gestores (tanto objeto-relacionales como orientados a objetos), siempre que éstos sean compatibles con el *binding* Java del estándar ODMG 2.0. De esta manera se siguen las últimas pautas impuestas por ODMG, en las que se destaca que lo importante es la adopción del *binding* Java, sin importar el método de almacenamiento subyacente.

En este trabajo, se ha desarrollado una herramienta que cumple exactamente estas expectativas, y que ha sido empleada para realizar comparaciones entre diferentes gestores, tal y como se puede observar en el capítulo 18.

17.3.1 Objetivos

Con el fin de conseguir esto, hay que implementar un traductor que a partir del lenguaje Java extendido (OML/Java), por ejemplo, genere las instrucciones Java correspondientes, más las llamadas pertinentes al gestor de almacenamiento persistente, que permita, la actualización, borrado, etc. de la información almacenada en la base de datos.

Pero el objetivo principal a conseguir por esta herramienta es no ligar la implementación de este traductor a un gestor persistente concreto. Su diseño ha de permitir la generación de código fácilmente para cualquier motor tanto de los existentes como de otros nuevos que puedan surgir [Pér99]. En el apéndice B de esta memoria se puede ver un ejemplo de aplicación de esta herramienta para los gestores ObjectDRIVER y PSE Pro.

Es también deseable que dicha herramienta cuente con una interfaz amigable que facilite el trabajo con la misma.

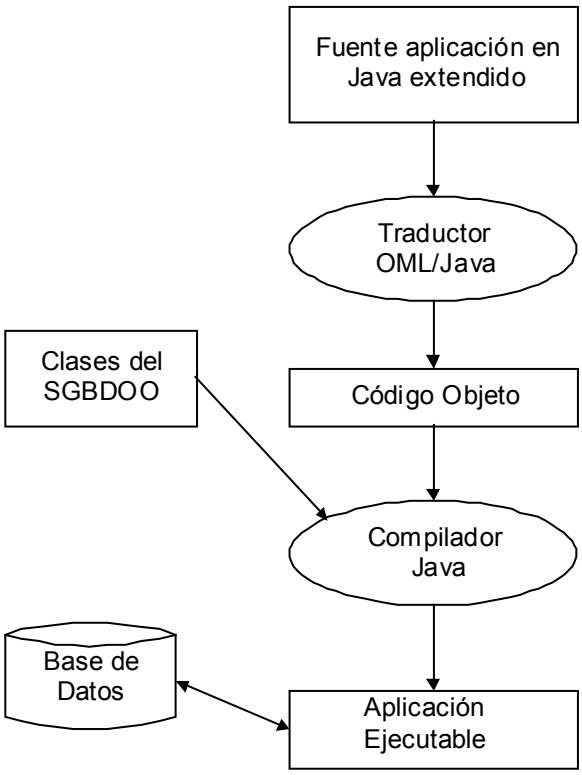


Figura 17.2. Esquema de la traducción

17.4 Diseño y construcción de la herramienta

A continuación se analizan las clases principales empleadas en la construcción de la herramienta.

17.4.1 Diagrama de clases básicas

A continuación se muestra el diagrama de clases empleado para la construcción de la herramienta, en el que se aprecia la relación entre las clases básicas que componen el traductor, su herencia, y la relación de agrupamiento.

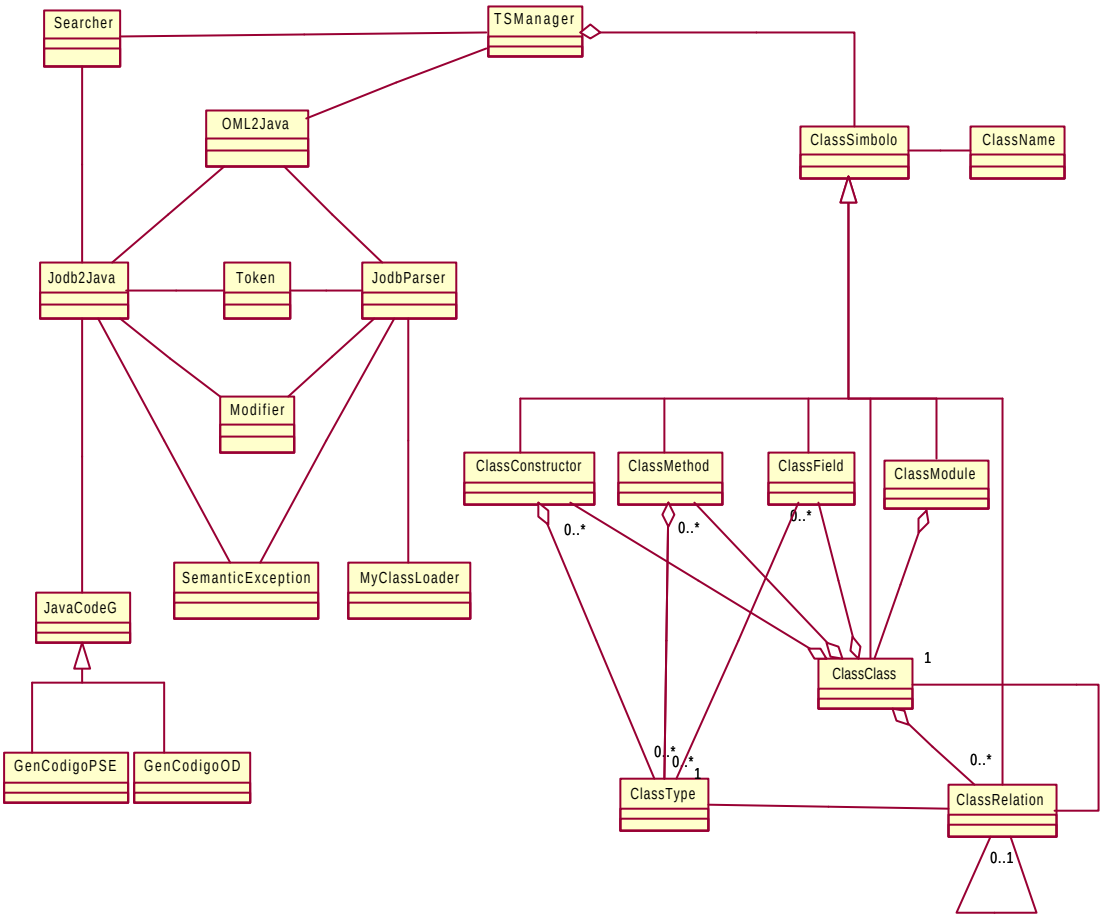


Figura 17.3. Diagrama de clases del traductor

Como se ha mencionado anteriormente la herramienta es básicamente un traductor y su construcción ha decidido realizarse en dos pasos.

En el primer paso se reconoce el código fuente y se realizan comprobaciones léxicas y sintácticas correspondientes al lenguaje y a sus construcciones. Es en este paso dónde se construye la tabla de símbolos, con la información básica acerca de las clases (Java u OML) que aparecen en el fichero OML[1], de las estructuras, de las excepciones, de los módulos, etc. Se realizan también algunas comprobaciones semánticas (comprobaciones de redundancia, repeticiones,...).

En el segundo paso, el traductor realiza la comprobación semántica en mayor profundidad, debido a que dispone en la tabla de símbolos de mucha más información sobre las construcciones, así como también de otras clases que intervienen y no

[1] Los ficheros con extensión OML son la entrada a la herramienta y contienen código en Java extendido

pertenecen al fichero fuente. En este paso, se realiza también la generación de código. A medida que se van leyendo las instrucciones y declaraciones, se van modificando o agregando a un buffer de líneas que a posteriori formará el archivo Java generado.

17.4.2 Gestión de la tabla de símbolos

Como se ha observado a la hora de construir un traductor tiene especial relevancia la tabla de símbolos ya que contiene información que permitirá entre otras cosas validar semánticamente las aplicaciones.

17.4.2.1 TSManager

Esta clase, ofrece toda la funcionalidad para trabajar con la tabla de símbolos, desde su creación, hasta las diferentes operaciones *set* y *reset* que son necesarias para establecer los nuevos estados y espacio de nombres. Los métodos de búsqueda e inserción constituyen su principal funcionalidad.

Atributos

CurTClass

Tabla hash para cada bloque, clase, o estructura que necesite un nuevo espacio de nombres.

pila

Pila donde se van colocando las subtablas que se van creando a medida que se lee el código.

Tsimb

Tabla hash que almacena las definiciones globales en el fichero oml.

Métodos

addSimbol(java.lang.String nsimbol, ClassSimbolo simbolo)

Añade un símbolo a la tabla de símbolos.

containsSimbol(java.lang.String nsimbol)

Busca en la tabla de símbolos la referencia pasada como parámetro.

getCurrent()

Retorna la subtabla de símbolos en el estado actual.

getSimbol(java.lang.String nsimbol)

Busca un símbolo en la tabla y lo devuelve.

getTSimb()

Devuelve la tabla de símbolos.

reset()

Finaliza la definición de componentes de una estructura.

set()

Establece una nueva subtabla de símbolos para la estructura actual.

Set(java.lang.String ns)

Establece la subtabla de símbolos para la estructura básica indicada.

toString()

Devuelve una cadena con los valores básicos de la tabla de símbolos.

17.4.2.2 ClassSimbolo

Esta clase está constituida por un único atributo que guarda el identificador de cada objeto tratado en la traducción. Ofrece un único método que sirve para consultar el identificador del objeto. La totalidad de objetos que son tratados durante el proceso de traducción se almacenan en la tabla de símbolos así que todos heredarán de esta clase.

17.4.2.3 ClassName

Representa la forma en la que Java identifica a clases e interfaces, es decir, siguiendo un formato de identificadores separados por puntos, que indican la ruta completa donde se puede encontrar a la clase, o bien, un sólo valor para las clases de las cuales se conoce su posición (porque pertenecen al mismo paquete o porque se importó el paquete al que pertenecen en la cláusula *import*). Ej. *java.lang.String.*

17.4.2.4 ClassClass

Esta clase es la encargada de recoger toda la información necesaria acerca de una clase OML, o una clase Java, que se intenta traducir. Contendrá constructores propios y heredados, métodos propios y heredados, campos propios y heredados, relaciones propias y heredadas, *extents*, claves, índices, modificadores, etc.

Toda esta información es necesaria para poder realizar las comprobaciones semánticas durante el proceso de traducción y generación de código. Posee además, los métodos necesarios para recoger esta información y ofrecerla a los diferentes módulos que lo necesiten.

17.4.2.5 ClassConstructor

Cada constructor de la clase tratada se ve reflejado en una clase de este tipo. Esta clase ofrece un método para poder crear un constructor nuevo sin parámetros ni excepciones, métodos para añadir parámetros, y una lista de excepciones obtenidas de las definiciones del fichero OML.

17.4.2.6 ClassField

Esta clase almacena información sobre como ha sido definido un campo, los modificadores que se establecen y el tipo de datos del mismo.

17.4.2.7 ClassMethod

Esta clase recoge la información sobre métodos u operaciones definidas. Ofrece un constructor para crear un método a partir de su nombre. También ofrece métodos para establecer los parámetros del método, su tipo de retorno, y las excepciones.

17.4.2.8 ClassModule

Un módulo, contiene definiciones de clases, interfaces, excepciones, etc. Esta clase posee una tabla donde se guardarán estas definiciones sin permitir duplicados. Aunque en este primer prototipo no se implementan, si se contempla su utilización y se almacenan los valores. Esta clase representa un primer paso para la modificación de los esquemas en las bases de datos orientadas a objetos.

17.4.2.9 ClassRelation

Esta clase sirve para construir y recoger toda la información que se necesita para procesar una relación entre dos clases. Ofrece un constructor para crear una instancia a partir del nombre y el tipo de la relación y la clase a la que se relaciona, y también métodos para indicar que la relación se definió en los dos sentidos, así como para consultar toda la información de la relación. Además, se pueden agregar los atributos que ordenarán las instancias de la relación.

17.4.2.10 ClassType

Esta clase se utiliza para guardar información sobre los tipos de datos de las relaciones, atributos, métodos, argumentos, etc. Se utiliza tanto para tipos primitivos como para clases, interfaces, etc. Almacena también los modificadores asociados a estos tipos como *final*, *in*, *out*, etc.

17.4.3 Gestión del traductor

La implementación del traductor se ha realizado en dos pasos. En este apartado se describe la clase que proporciona la traducción (OML2Java), así como las clases que se han empleado para la implementación de cada uno de los pasos del traductor.

17.4.3.1 OML2Java

Es la clase principal del sistema. Es la encargada de realizar la traducción, es decir, llama a las dos clases que realizan el proceso de traducción pasándoles el fichero con el código fuente a traducir. Invoca también al generador de código del motor elegido que se encargará de crear el fichero objeto, y a la interfaz del traductor.

Atributos

Filein

Referencia al fichero con el código a traducir

genjava

Referencia al objeto que realiza la generación de código

omlgen

Referencia al objeto que realiza el segundo paso de la traducción

parser

Referencia al objeto que realiza el primer paso de la traducción

TS

Referencia a la tabla de símbolos

Métodos

main(java.lang.String[] args)

Ejecuta la aplicación y crea los objetos que realizan la traducción.

parser (OMLInterface oi)

Método llamado por la interfaz para realizar la traducción.

parser(java.lang.String afilename, boolean apersis, int amotor)

Traduce el código oml a código Java.

print(java.lang.String ms)

Muestra un mensaje por la salida por defecto.

17.4.3.2 JodbParser

Es la clase encargada de realizar el proceso de traducción en su primer paso. Esta clase es generada a partir del fichero del mismo nombre y extensión "jj" que se pasa a la herramienta *javacc*. Inspecciona el código OML del fichero de entrada, y construye la tabla de símbolos, además de realizar las comprobaciones léxicas y sintácticas pertinentes, para lo cual dispone de los métodos necesarios.

17.4.3.3 Jodb2Java

Es la clase encargada de realizar el proceso de traducción en su segundo paso. Esta clase es generada a partir del fichero del mismo nombre y extensión "jj" que se pasa a la herramienta *javacc*. Realiza las comprobaciones semánticas sobre el código OML del fichero de entrada utilizando para ello la tabla de símbolos. También dispone de los métodos necesarios para la generación de código apoyándose en un objeto de la clase *JavaCodeG*.

17.4.3.4 OMLInterface

Esta clase construye la interfaz gráfica de usuario del sistema. La interfaz gráfica es una interfaz WIMP (*Windows Icons Menus Pointers*) que proporciona menús con las opciones disponibles en el prototipo.

17.4.3.5 OQLParser

Es la clase encargada de realizar el proceso de comprobaciones de cadenas OQL. Esta clase es generada a partir del fichero del mismo nombre y extensión "jj" que se pasa a la herramienta *javacc*. Con esta clase, se conoce en tiempo de compilación, si la cadena que sigue la sintaxis OQL es correcta, o por el contrario, generará errores cuando sea interpretada por los motores persistentes.

17.4.3.6 MyClassLoader

Esta clase se utilizará para buscar clases de Java, y retornar su declaración en formato de objeto de tipo *Class*. Por defecto, las clases declaradas en el paquete *java.lang* son accesibles por todas las aplicaciones, así como las que pertenecen a los paquetes importados en la claúsula *import,* y las clases propias importadas en la misma cláusula.

17.4.3.7 Searcher

Clase que permite realizar la búsqueda de un determinado componente, sea éste una clase, un tipo de dato primitivo, una relación, un método, etc. Realizará la búsqueda inicialmente en la tabla de símbolos, si no encuentra al componente en ella, lo buscará como un componente de la clase que se está analizando (campo, método o relación), para posteriormente si no lo ha encontrado utilizar el cargador de clases buscándolo en las clases básicas o en las importadas. Retornará un objeto que se analizará utilizando la reflectividad para saber si se trata de un método, clase, campo, etc.

17.4.3.8 Modifier

Clase que ofrece la funcionalidad de la clase *modifier* del paquete *java.lang.reflect*, y además añade los métodos para poder crear enteros que representen a los modificadores, y añadir nuevos modificadores.

17.4.3.9 Token

Esta clase es generada por la herramienta *javacc*, y es la encargada de recoger información sobre un determinado token del fichero fuente. Los tokens han sido descritos en los ficheros JodbParser.jj para los ficheros OML, y OQLParser.jj para las cadenas de texto que describen peticiones según la sintaxis OQL.

17.4.4 Generación de código

La clase encargada de generar código es *JavaCodeG*. Esta clase se encarga de la generación de código independiente del gestor persistente con el que se esté trabajando. Proporciona una serie de métodos que se encargan de la construcción del fichero de destino, así como de la gestión del buffer donde se escribe el código Java que será el lenguaje objeto. Incluye también los métodos de la interfaz *BasicMotorMethods* que es la que proporciona los métodos básicos que deben ser implementados para la generación de código por cualquier gestor persistente empleado. Por la importancia de esta clase en el diseño se detalla un poco más su estructura.

17.4.4.1 Estructura de la clase JavaCodeG

En este apartado se reflejan los atributos de la clase así como los métodos propios de la misma para la gestión del buffer, construcción del fichero de destino, etc.

Atributos

addpersistence

Indica si se está generando una clase persistente

buffer

Buffer donde se escribe el código generado

EXTENSION

Constante con la extensión de los ficheros generados

fich

Referencia al fichero generado

linea

Buffer donde se escribe temporalmente código

Métodos

addPersistence(boolean add)

Indica que los gestores deben añadir persistencia al código que se genera.

closeFile()

Cierra el fichero donde se está escribiendo código.

createJavaFile(java.lang.String fichero)

Crea el fichero que recibe el código.

endBody()

Escribe el código java perteneciente al fin de un bloque.

error(java.lang.String str)

Saca un mensaje por pantalla y aborta la ejecución.

flushBuffer()

Vacía el contenido de los buffers que contienen el código a generar.

genExceptionBody(ClassClass exception)

Crea constructores para cada tipo definido en el cuerpo de la excepción.

genStructBody(ClassClass tstruct)

Genera el código Java correspondiente al cuerpo de la estructura.

genType(ClassType tipo)

Retorna una cadena con el código para el tipo indicado.

makeExceptionClass(ClassClass exception)

Crea una clase a partir de una declaración de excepción.

makeStructClass (ClassClass tstruct)

Crea una clase a partir de una declaración de estructura.

print(java.lang.String ms)

Visualiza un mensaje por pantalla.

startBody()

Escribe el código Java perteneciente al inicio de un bloque.

write2Buffer(java.lang.String cod)

Escribe el contenido del parámetro en el buffer de líneas.

write2File(java.lang.String cod)

Escribe el contenido del parámetro directamente en el fichero.

write2Linea(java.lang.String cod)

Escribe el contenido del parámetro en el Stringbuffer Linea.

17.4.4.2 Interfaz BasicMotorMethods

A parte de los métodos anteriores, esta clase incluye los métodos de la interfaz *BasicMotorMethods* en la que se encuentran los métodos que deben ser implementados por cada uno de los gestores persistentes que serán soportados por la herramienta con el fin de generar código apropiado para cada uno de ellos.

Métodos

WriteExtent(java.lang.String nextent)

Devuelve el código necesario para simular un *extent* mediante programación.

WriteFirstInMethod(java.lang.String nmet)

Devuelve las instrucciones que deben contener los métodos al inicio.

WriteIndex(java.lang.String nindex)

Devuelve el código necesario para simular los índices mediante programación.

WriteLastInMethod(java.lang.String nmet)

Devuelve las instrucciones que deben contener los métodos al final.

WriteMyExtends()

Devuelve las clases que debe extender la clase.

WriteMyImplements(boolean more)

Devuelve las sentencias donde aparecen las interfaces que se deben implementar.

WriteMyImports()

Devuelve los paquetes que debe implementar.

WriteRelationship(classRelation rel)

Devuelve el código necesario para simular una relación mediante programación.

17.4.5 Gestión de la extensiones de las clases

Como el propio ODMG indica, es responsabilidad del programador en Java utilizar un tipo de datos que represente la extensión de la clase, y escribir los métodos para mantenerla. Siguiendo estas recomendaciones se utilizará una clase denominada *extent* para representar al conjunto, y se escribirán métodos en ella para mantenerlo. En la generación de código para cada clase se añadirá una instrucción que defina el *extent*, el objeto soporte, y un método para mantener al *extent*. Este método se denominará *updateExtent*.

17.4.5.1 Método updateExtent

Este método, es el encargado de mantener al *extent* de la clase, es decir, al conjunto que almacenará a las instancias de la clase en la base de datos. Por mantener se entiende añadir y eliminar de dicho *extent* instancias de la clase.

En función de la descripción realizada, este método tendrá como argumentos:

- La referencia a la base de datos en la que se encuentra el *extent*.

- Un *flag* o indicador del tipo de operación a realizar. Como son dos las operaciones posibles, se opta por utilizar un valor booleano, en el que un valor cierto, indicará añadir y falso indicará eliminar.

17.4.6 Gestión de las relaciones

ODMG no ofrece ningún tipo de solución a las relaciones, por lo que mediante programación se van a ofrecer un conjunto de métodos para poder tratarlas.

Una relación de grado 1 se corresponderá con un campo cuyo valor será una referencia a una instancia de la clase relacionada. Se ofrecerán métodos para establecer y recuperar esta instancia.

Una relación de grado N se puede entender como un *extent* particular para cada instancia. Se corresponderá con un campo de tipo colección cuyo valor será una colección de instancias de la clase relacionada. Se ofrecerán también métodos para mantener y recuperar esta colección.

17.4.6.1 Método updateRel_

Este método cumple con el propósito de mantener[2] una colección de instancias que representan una relación de grado N. A este método se le añadirá el nombre de la relación, es decir, si la relación se denomina *alumnos*, el método se llamará *updateRel_alumnos*, ya que pueden ser varias las relaciones de la clase.

En función de la descripción realizada, este método tendrá como argumentos:

- La referencia a la instancia que se quiere añadir o eliminar.
- Un flag o indicador del tipo de operación a realizar (verdadero para añadir y falso para eliminar).

17.4.6.2 Método setRel_

Este método cumple con el propósito de establecer el valor de la instancia en una relación de grado 1. A este método se le añadirá el nombre de la relación como se explica en el punto anterior. Sustituye a *updateRel_*, es decir, es la imagen del método *updateRel* para relaciones de grado 1. Cuando se establece la relación con una instancia, utilizaremos este método para crearla; cuando esta relación se pierda, utilizaremos el método con un valor nulo (*null*), para eliminarla.

17.4.6.3 Método getRel_

Este método cumple con el propósito de recuperar la instancia relacionada, en relaciones de grado 1, o el conjunto de instancias relacionadas en relaciones de grado N. Al igual que en los casos anteriores a este método también se le añadirá el nombre de la relación.

No necesita argumentos, y el tipo de retorno estará en función del campo que simula la relación. Si la relación es de grado 1, el tipo de retorno será la clase con la que se establece la relación; si es de tipo N, será del tipo colección que se ha definido en la relación.

[2] Por mantener se entiende añadir y eliminar a la colección instancias de la clase relacionada.

17.4.7 Paquete OML

En el paquete Java denominado OML se pueden encontrar las definiciones de las clases que nos van a permitir manejar tanto bases de datos como objetos persistentes, siguiendo las indicaciones del grupo de desarrollo. Este paquete incorpora clases abstractas e interfaces que cumplen los requisitos expuestos por el *binding* Java de ODMG con ligeras modificaciones para poder ser utilizadas en una implementación real.

17.4.7.1 Clase Database

Esta clase ha sido definida por el *binding* Java de ODMG, y se respeta fielmente la definición establecida. La única modificación realizada, es como consecuencia de utilizar una implementación real, y consiste en utilizar una referencia a un tipo *Database* para poder retornarlo en el método estático *open*.

Se ha considerado necesario también un método para poder crear bases de datos. Al considerar este método necesario, se ha añadido a la definición dada por ODMG, un método denominado *create*, que toma como argumentos una cadena de caracteres, que representa al nombre de la base de datos que queremos crear, y un entero, indicando el modo de creación, es decir, la base de datos se crea para un usuario en particular, para un grupo, o para todos.

```
public abstract Database create(String name, int createMode) throws ODMGException;
```

Con relación a los diferentes modos de creación, se ha utilizado una interfaz, que posee los valores necesarios a priori para crear una base de datos, utilizando una terminología consistente con otras herramientas. La interfaz se denomina *OMLConstants* [Pér99], y posee los valores enteros para utilizar en el argumento *createMode*. Se ha utilizado una interfaz para evitar ambigüedades con los valores de acceso definidos en la clase Database, y para modificar lo menos posible la clase.

```
/** Database createMode */

public static final int ALL_WRITE = 1;

public static final int GROUP_WRITE = 2;

public static final int OWNER_WRITE = 3;
```

17.4.7.2 Clase Transaction

Al igual que la clase *Database*, en la implementación de la clase abstracta *Transaction* se ha seguido fielmente la descripción del *binding* Java propuesto. Las únicas modificaciónes surgen también de la necesidad de retornar un valor en un método estático (por lo que se ha introducido una instrucción *return* con un objeto de este tipo), y de asignar un valor a las constantes declaradas en esta clase.

17.4.7.3 Interface Collection

Es la interfaz principal de la que heredan las tres siguientes. Su implementación es la que se ofrece en el *binding* Java.

17.4.7.4 Interface Set

Una clase que implemente esta interfaz será una clase que almacene un conjunto de instancias, de las cuales, no se permitirán instancias duplicadas, y no existirá un orden determinado.

17.4.7.5 Interface List

Una clase que implemente esta interfaz será una clase que almacene un conjunto de instancias, de las cuales, se permitirán instancias duplicadas y existirá un orden determinado.

17.4.7.6 Interface Bag

Una clase que implemente esta interfaz, será una clase que almacene un conjunto de instancias, de las cuales, se permitirán instancias duplicadas, y no existirá un orden determinado.

17.4.7.7 Interface Array

Esta interfaz tiene dos maneras de implementarse en Java, mediante arrays de Java o con la clase *Vector*. Esta última ha sido la elegida en esta implementación.

17.4.7.8 Clase OQLQuery

La clase OQLQuery se ha definido en el *binding* para Java para la utilización del lenguaje de consulta OQL. La implementación refleja los métodos propuestos por el estándar.

17.4.7.9 Excepciones ODMG

Cuando una condición de error es detectada, se lanza una excepción utilizando el mecanismo de excepciones de Java. A continuación se indican las excepciones para las que se ha realizado una implementación siguiendo el *binding* Java.

ODMGException

A esta clase de excepción corresponden las condiciones de error que se pueden dar en la base de datos, en sus operaciones o peticiones. Estos posibles errores, se identifican más en detalle en las excepciones que heredan de ésta y que son definidas en los siguientes subapartados.

- *DatabaseNotFoundException.* Lanzada cuando se pretende abrir una base de datos que no existe.
- *DatabaseClosedException.* Lanzada cuando se pretende llamar a un método de una base de datos que ha sido cerrada o que todavía no ha sido abierta.
- *DatabaseOpenException.* Lanzada cuando se pretende abrir una base de datos que ya ha sido abierta.
- *ObjectNameNotFoundException.* Lanzada cuando se pretende acceder o solicitar un objeto con identificador asociado en la base de datos, y éste no se encuentra.
- *ObjectNameNotUniqueException.* Lanzada cuando se pretende enlazar o asociar un identificador a un objeto persistente, y el identificador ha sido utilizado o ya existe en la base de datos.
- *QueryParameterCountInvalidException.* Lanzada cuando el número de parámetros a enlazar en la consulta que se pretende ejecutar, es diferente al número de enlaces para variables.
- *QueryParameterTypeInvalidException.* Lanzada cuando el tipo de un parámetro que se enlaza en la consulta no es el que se esperaba.

ODMGRuntimeException

A esta clase de excepción corresponden las condiciones de error que se pueden dar cuando se trabaja con objetos persistentes. Se detallan a continuación las principales.

- *DatabaseIsReadOnlyException.* Lanzada cuando se pretende llamar a un método que modifica la base de datos que está abierta en modo de sólo lectura.
- *LockNotGrantedException.* Lanzada cuando no se puede conceder un bloqueo sobre un objeto.
- *TransactionAbortedException.* Lanzada cuando el sistema de base de datos provoca que la transacción aborte, debido a inter-bloqueos, fallos de recursos, etc.
- *TransactionInProgressException.* Lanzada cuando se pretende llamar a un método dentro de una transacción, y este método debe ser llamado cuando no hay una transacción en curso.
- *TransactionNotInProgressException.* Lanzada cuando se pretende realizar una operación fuera de una transacción, y esta operación debe ser llamada cuando una transacción está en curso.

17.5 Ejemplo sencillo de utilización de la herramienta

Esta herramienta (llamada *OML2Java*) proporciona una interfaz en línea de comandos y una interfaz WIMP con exactamente la misma funcionalidad. El objetivo de este apartado es proporcionar una idea general sobre su funcionamiento, sin convertirse en un manual de usuario para la misma. Los gestores inicialmente considerados en la misma son ObjectDRIVER y PSE Pro.

17.5.1 En línea de comandos

La clase OML2Java ofrece una interfaz en modo comando que exige de la existencia de una serie de parámetros para su correcto funcionamiento:

- **filename**: nombre del fichero que se desea traducir.
- **Persistencia**: ON | OFF, con este parámetro se quiere indicar que el fichero que se está traduciendo es una clase que será almacenada en el sistema o motor de persistencia (ON), o por el contrario, es una clase que se utiliza para consultar al motor o realizar alguna otra tarea (OFF).
- **Motor**: 1 | 2 | 3, cada número indica un tipo de gestor reconocido por el traductor y que se puede generar código para él. Actualmente, se reconoce con el número 2, al gestor PSE Pro, con el 3 al gestor ObjectDRIVER, y con el 1 código Java solamente, omitiendo las extensiones del *binding* Java de ODMG.

Por ejemplo, la siguiente línea de comando se utiliza para traducir el fichero Emp.oml, que se entiende lleva descrita a la clase *Emp* con extensiones del *binding* Java. Esta clase es persistente y almacenada mediante el gestor PSE Pro.

```
java OML2Java Emp.oml ON 2
```

17.5.2 Mediante la interfaz WIMP

Esta herramienta ofrece también una interfaz visual basada en el empleo de menús. Los pasos para realizar la operación anterior mediante esta interfaz serían los siguientes:

1. Creación del fichero que contiene la especificación en Java extendido, o bien abrir el fichero (*Emp.oml*) que contiene dicha especificación si ya está creado.

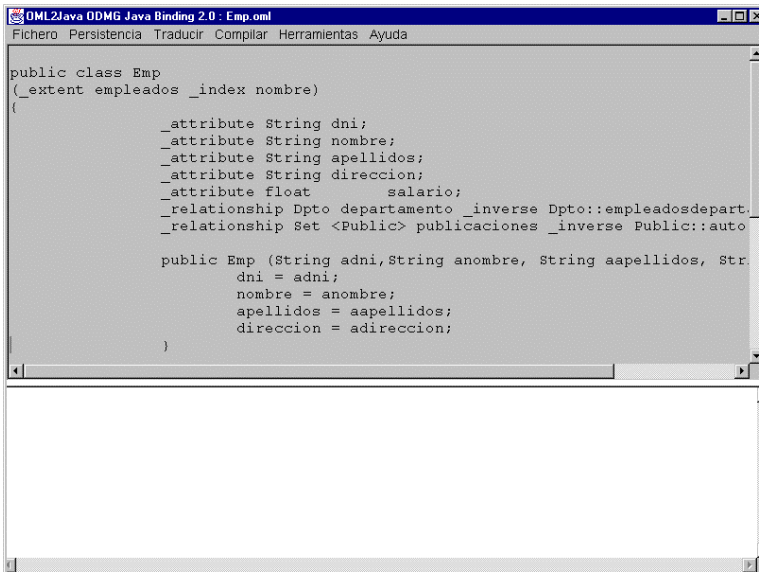


Figura 17.4. Especificación de la clase *Emp* en OML/Java

2. Indicación de que se desea que esta clase sea persistente. Para ello se activa la opción correspondiente en el menú.

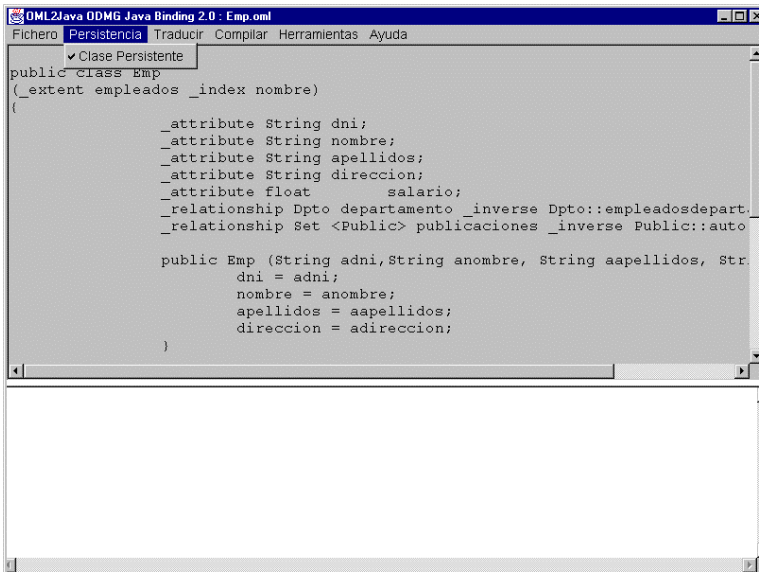


Figura 17.5. Especificación de la persistencia para la clase tratada

3. Selección del gestor elegido para proporcionar la funcionalidad de base de datos (en este caso es PSE Pro), y realizar la traducción. En la parte inferior de la ventana se muestran los mensajes indicando, en este caso, que la traducción se ha realizado con éxito.

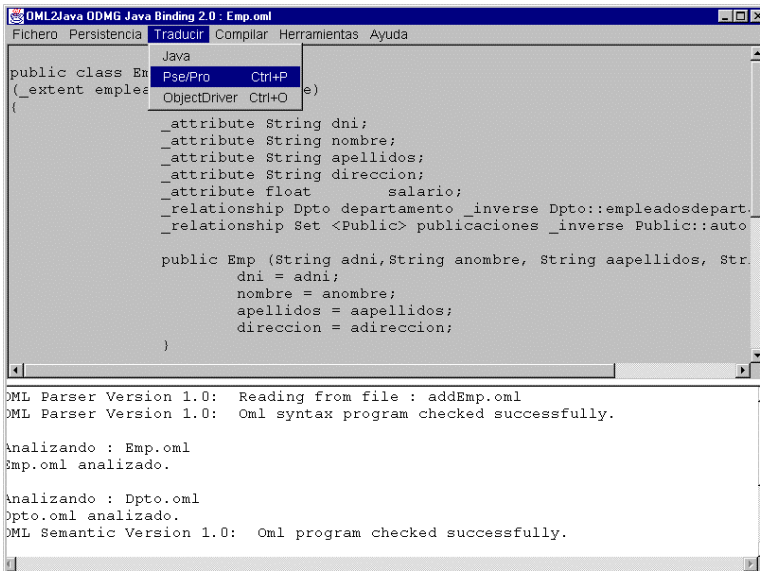


Figura 17.6. Selección del gestor PSE Pro y generación de código para el mismo

4. Finalmente se puede observar el código Java generado.

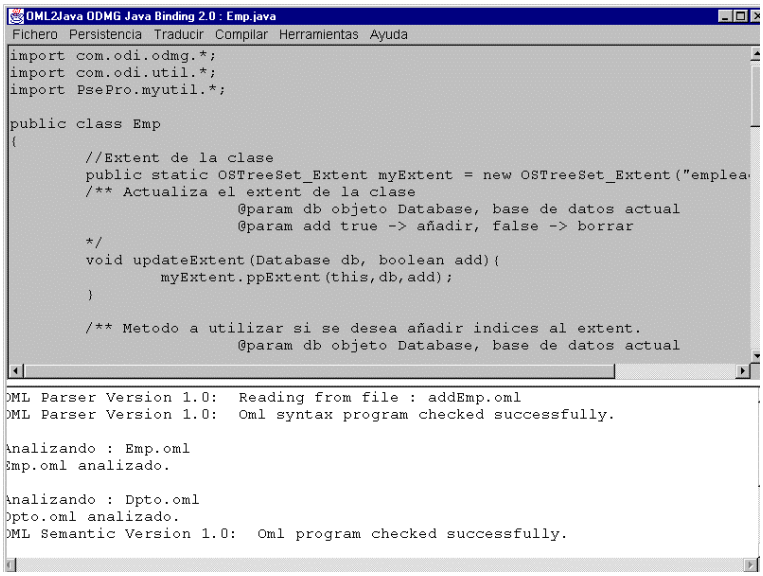


Figura 17.7. Código Java generado después de la traducción

17.6 Utilidad de la herramienta

La herramienta descrita, por su diseño y características, es ideal para comparar diferentes gestores, investigar sus rendimientos, etc. ya que el mismo código fuente compatible con el *binding* Java de ODMG se puede generar sin esfuerzo para varios gestores persistentes.

Además, la incorporación de nuevos gestores es muy fácil, ya que únicamente hay que añadir a la herramienta la parte que genera código para cada gestor, y el sistema ha sido diseñado de forma que esta tarea sea sencilla.

Se convierte, por tanto, en una herramienta ideal para el investigador ya que le facilita mucho el trabajo, ahorrándole mucho tiempo.

Capítulo 18 Evaluación del prototipo

En este capítulo se procede a la evaluación del prototipo implementado, y con ese fin se ha realizado un análisis del rendimiento desde dos puntos de vista. En primer lugar, empleando el prototipo como plataforma de *benchmarking* para analizar el comportamiento de diferentes técnicas de indexación de bases de datos orientadas a objetos, y en segundo lugar, analizando el rendimiento de este sistema de gestión de bases de datos en comparación con otros gestores.

18.1 Introducción

La evaluación del prototipo que aquí se presenta se ha encaminado en dos direcciones:

- Por un lado, observar el comportamiento de diferentes técnicas de indexación empleadas en bases de datos orientadas a objetos, pero sobre el sistema gestor aquí diseñado. Las técnicas empleadas en la comparación son las inicialmente incorporadas en el prototipo: *SC*, *CH-Tree* y *Path Index(PX)*.
- Por otro lado, mostrar los rendimientos que ofrece este sistema comparado con otros gestores existentes en el mercado, en este caso el *wrapper* ObjectDRIVER y el gestor de objetos PSE Pro, ambos en sus versiones para Java.

Es importante tener en cuenta que éste es un proyecto de investigación cuya finalidad principal sobre todo en este primer prototipo es ofrecer una buena plataforma de experimentación. Los estudios realizados se han llevado a cabo teniendo en cuenta las técnicas de indexación implementadas, sin olvidar que el rendimiento del sistema ha quedado en un segundo plano en el desarrollo de este primer prototipo.

La comparativa realizada no es exhaustiva; simplemente se trata de obtener una idea aproximada del rendimiento de un sistema como el propuesto aquí.

18.2 Comparación de las diferentes técnicas de indexación implementadas

18.2.1 Condiciones de la comparativa

Para realizar el estudio comparativo se parte de una estructura de clases que incluye jerarquía de herencia y relaciones de agregación de forma que pueda observarse el comportamiento de las tres técnicas de indexación implementadas. El estudio llevado a cabo aquí no considera los requerimientos de espacio de las estructuras, únicamente considera los tiempos empleados para la realización de las operaciones.

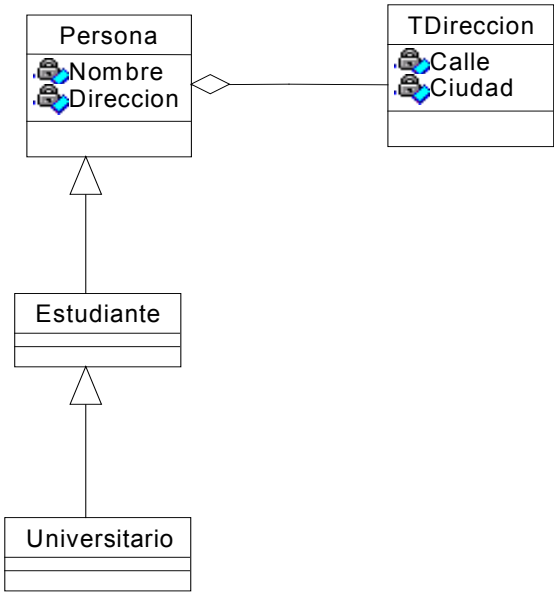


Figura 18.1. Jerarquía de clases para la comparativa

Las pruebas fueron realizadas en un equipo Pentium 233, con 64 MB de RAM y bajo el sistema operativo Windows NT 4.0. Los resultados son expresados en una unidad de medida independiente de la máquina sobre la que se está trabajando. La unidad de medida es representada por el tiempo que la máquina tarda en recuperar un array de 500 objetos de tipo entero del sistema de persistencia. Los tiempos reflejados se obtienen del cálculo de tiempos medios para un número elevado de operaciones.

Las operaciones que se han tomado como base para el estudio son la inserción, el borrado y las búsquedas.

18.2.2 Rendimiento de las técnicas de indexación

El estudio tiene como objetivo evaluar las técnicas de indexación implementadas. Para ello se ha estudiado el comportamiento de las técnicas implementadas que soportan jerarquía de herencia (SC y CH-Tree), y se ha estudiado el comportamiento de la técnica Path Index (PX) para consultas que impliquen jerarquía de agregación.

18.2.2.1 Técnicas que soportan jerarquías de herencia

En un primer bloque de baterías de test se ha partido de la definición de un índice cuya clave no implica una agregación, es decir, es un objeto simple (*Persona.Nombre*). Con el mecanismo de indexación diseñado, al especificar un índice para una clase base automáticamente queda definido para todas sus subclases. En este caso las clases *Estudiante* y *Universitario* dispondrán también del mismo índice, que será también mantenido por el sistema.

Inserción

Representa el tiempo empleado en añadir un objeto a la estructura que representa la técnica de indexación correspondiente.

T.Indexación	Inserción
SC	1,14
CH-Tree	1,42
PX	1,34

Tabla 18.1. Tiempos medios para la inserción

Los resultados obtenidos demuestran que el tiempo de inserción es menor en el SC, lo que parece lógico, ya que es la técnica que menos información almacena en el nodo hoja.

Borrado

De la misma manera representa el tiempo empleado en eliminar un objeto de la estructura que representa la técnica de indexación correspondiente.

T.Indexación	Borrado
SC	1,45
CH-Tree	2,29
PX	1,62

Tabla 18.2. Tiempos medios para la eliminación

El tiempo de borrado es menor también en el SC, al ser la estructura más simple. La estructura que requiere más operaciones (CH-Tree) se hace notar con un tiempo de eliminación mayor.

Búsquedas en una clase

Hacen referencia a búsquedas realizadas sobre la clase objetivo, pero sin incluir sus subclases. Las búsquedas serán de la forma *"Seleccionar todas las personas cuyo nombre sea XX"* (no incluyendo ni *estudiantes* ni *universitarios*).

De todos los métodos proporcionados por el motor de BDOviedo3 para la realización de consultas, este tipo de consultas se han realizado con el método *RetrieveObjects(...)* que recupera todos los objetos de la clase teniendo en cuenta el índice por defecto de la clase.

T.Indexación	Búsqueda en una clase
SC	0,63
CH-Tree	0,96
PX	0,61

Tabla 18.3. Tiempos medios para las búsquedas en una clase

Cuando se realiza una búsqueda que implica una única clase, se observa que no es recomendable el empleo de la técnica CH-Tree. Esta diferencia entre las técnicas obedece principalmente a que el CH-Tree tiene que hacer una selección dentro de la información que tiene almacenada en su nodo hoja, mientras que las técnicas SC y PX no.

Búsquedas en una jerarquía

Hacen referencia a búsquedas realizadas sobre la clase objetivo, pero incluyendo sus subclases. Las búsquedas serán de la misma forma que la anterior *"Seleccionar todas las personas cuyo nombre sea XX"*, pero ahora incluyendo *estudiantes* y *universitarios*.

El método empleado para la recuperación en este caso es el mismo que en el caso anterior *RetrieveObjects(..., True),* pero con el último parámetro a *True* para indicar que devuelva instancias de todas las clases de la jerarquía.

T.Indexación	Búsqueda en una jerarquía
SC	4,70
CH-Tree	1,83
PX	4,60

Tabla 18.4. Tiempos medios para las búsquedas en una jerarquía

Como se observa la técnica de indexación más adecuada para búsquedas sobre una jerarquía de clases, y además con mucha diferencia, es CH-Tree. Eso obedece principalmente a que en un único árbol tiene almacenadas todas las clases de la jerarquía que contienen ese valor 'XX' en el atributo indexado, mientras que en las otras dos técnicas, es necesario recorrer tantos árboles como clases compongan la jerarquía.

Conviene destacar que la técnica Path Index por si misma no soporta consultas sobre la jerarquía de herencia, sin embargo, el mecanismo de indexación diseñado se ha encargado de ir recorriendo las estructuras índices (*path*) definidas para sus subclases (*estudiante* y *universitario*) y componiendo los resultados, al igual que hace con el SC.

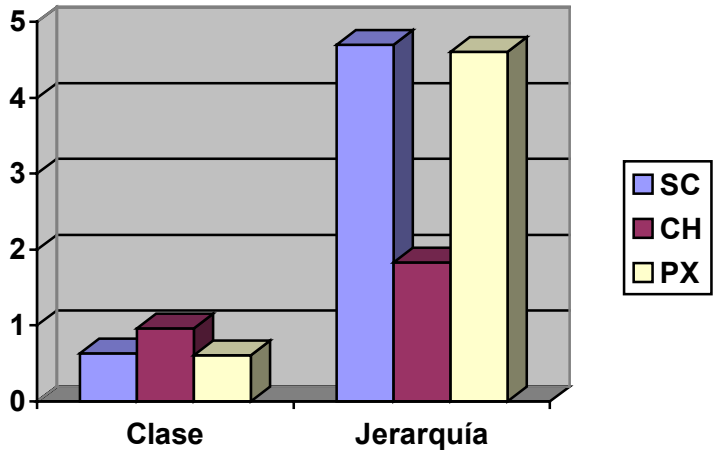


Figura 18.2. Resultados para la búsqueda en una clase y en una jerarquía

18.2.2.2 Técnica que soporta jerarquía de agregación

Supongamos que se desean ahora realizar consultas que impliquen el recorrido de una jerarquía de agregación ("*Seleccionar todas las personas que viven en la calle XX*" o por ejemplo, *"Seleccionar las direcciones de todas las personas que viven en la calle XX*"). Este tipo de consultas se ven favorecidas, a priori, por la existencia de técnicas de indexación que soporten las jerarquías de agregación.

En este apartado se muestran los resultados obtenidos al realizar búsquedas que implican el recorrido de una jerarquía de agregación. Como se ha mencionado anteriormente, las técnicas de indexación basadas en árboles B^+ y CH-Tree no permiten las consultas por agregaciones de atributos, con lo que en el prototipo implementado hay que recurrir a un recorrido sin tener en cuenta los índices (basado en las extensiones de las clases), o bien emplear la técnica PX para indexar el camino *Persona.Direccion.Calle.*

Indexación	Tiempo medio
PX	0,71
Sin indexación	7,48

Tabla 18.5. Tiempos de búsqueda en agregaciones

Como se puede observar la búsqueda mediante el PX es más rápida que la consulta sin indexación. Hay que tener en cuenta que la búsqueda sin indexación implica recorrer todos los objetos de la clase y comprobar el valor de su atributo agregado.

18.2.3 Conclusiones

Los resultados son los esperados de acuerdo a lo comentado en el capítulo 5. Como se puede observar no existe una técnica de indexación mejor que las demás en todos los aspectos. Si bien una determinada técnica ofrece un mejor resultado en un determinado aspecto (de los que se han observado), tiene unos rendimientos bajos en otros. Tómese por ejemplo la técnica CH-Tree. Esta técnica proporciona una búsqueda rápida en jerarquías de clases a condición de incrementar la complejidad del árbol de almacenamiento. Y esto se traduce en un tiempo superior en las otras operaciones

comunes en los índices, como son la inserción y el borrado. Además, no soporta las consultas basadas en las jerarquías de agregación.

Esto demuestra lo importante que es para un sistema contar con la técnica de indexación adecuada al tipo de dato y al tipo de consulta más corriente a emplear, y refuerza por tanto la idea de proporcionar en los SGBDOO mecanismos de indexación extensibles que permitan incorporar las técnicas adecuadas de una forma sencilla, y sin que eso suponga una sobrecarga en el sistema que por supuesto repercuta en su rendimiento.

Por otro lado, la existencia de una plataforma de experimentación como la que proporciona este prototipo supone un avance importante, ya que permite experimentar con diferentes técnicas de indexación para conocer cuales son las que mejor se comportan y bajo que circunstancias.

18.3 Comparación con otros gestores

Para tener una idea de la bondad del sistema aquí construido era necesario compararlo con otros gestores existentes en el mercado. En este caso los gestores seleccionados han sido el envoltorio objeto-relacional ObjectDRIVER, y el gestor orientado a objetos PSE Pro, ambos compatibles con el *binding Java* de ODMG 2.0. La justificación sobre la elección de estos gestores puede encontrarse en el apéndice B.

18.3.1 Consideraciones previas

En primer lugar, hay que precisar algunas características que pueden influir en los resultados de la comparativa, y que no pueden ser recogidas en la toma de tiempos.

En el caso del gestor ObjectDRIVER se realiza un proceso previo a cualquier operación, en el que se establece la correspondencia entre los atributos de las clases y los campos de las tablas relacionales. Este proceso se realiza cuando se abre la base de datos objeto de las operaciones, por tanto, este tiempo de preproceso no ha sido tenido en cuenta a la hora de medir los tiempos de inserción y búsqueda.

Si bien es un tiempo constante, influye en la productividad de la herramienta o gestor, porque una vez establecidas las correspondencias, el motor encargado de realizar las operaciones de mantenimiento de la base de datos es el motor objeto del recurso del sistema ODBC, en este caso, el recurso será MsAccess, y el gestor será el Microsoft Jet.

18.3.2 Condiciones de la comparativa

El estudio es realizado partiendo de una estructura de clases que consta de relaciones de herencia y de agregación, y es realizado en una máquina con un sistema operativo Windows NT 4.0, con microprocesador Pentium/200MMX, con 96MB de memoria RAM y 3Gb de disco duro.

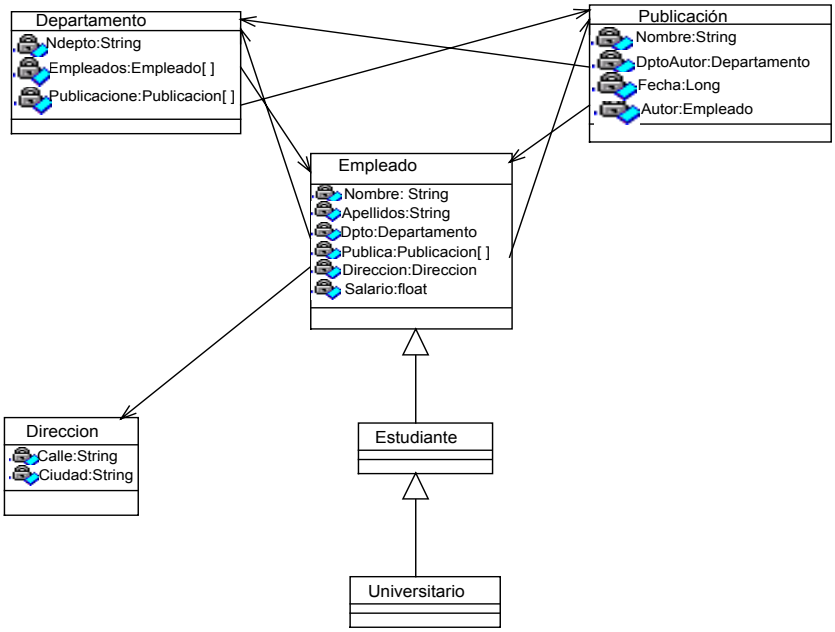


Figura 18.3. Jerarquía de clases empleada para la comparación entre gestores

Las operaciones a tener en cuenta en este estudio serán las inserciones y consultas sencillas, ya que el prototipo implementado no soporta interrogaciones sofisticadas a la base de datos.

Inserción

Los tiempos de inserción han sido tomados desde que se realiza la operación de inserción hasta que la base de datos ha sido modificada correctamente, y la nueva instancia queda registrada y accesible para cualquier usuario.

Consulta

Las consultas empleadas para realizar las mediciones tienen las siguientes características:

- Son realizadas sobre una única clase, es decir, no se han realizado uniones entre clases, ya que esto únicamente lo permite ObjectDRIVER y con muchas restricciones BDOviedo3.
- Si la clase consultada es la clase base de una jerarquía de clases, los resultados de la consulta implican únicamente la clase base, no la jerarquía, ya que esto no lo soporta PSE Pro.
- No se han empleado caminos de la forma (*Empleado.publica.nombre*), ya que esto no lo soporta PSE Pro.
- Tienen como objetivo interrogar al sistema sobre las instancias con un valor concreto en un determinado atributo: *"Seleccionar todos los empleados cuyo nombre sea XX"*.
- Son realizadas en OQL en el caso de ObjectDRIVER. En el caso de PSE Pro y BDOviedo3 se acude a las clases y métodos proporcionados por los motores para este cometido, y atendiendo a las convenciones por ellos fijadas.

Los resultados obtenidos se mostrarán con la misma unidad de medida que en el apartado anterior. Esto facilitará la reproducción de las pruebas sin crear una

dependencia de la máquina sobre la que se realizan. También, al igual que en el caso anterior, todos los tiempos se obtienen del cálculo de los tiempos medios para un número elevado de operaciones.

18.3.3 Rendimientos sin considerar la indexación en BDOViedo3

18.3.3.1 Inserción y Consulta

En este apartado, se muestran los resultados obtenidos para las operaciones de inserción y consultas básicas realizadas sobre los tres gestores.

Gestor	Inserción	Consulta
PSE Pro	0,3	1,21
ObjectDRIVER	0,12	0,19
BDOviedo3	1,13	2,37

Tabla 18.6 Tiempos medios para inserción y consulta

Los resultados obtenidos nos muestran que los motores comerciales se comportan mejor que BDOviedo3, pero eso se debe en gran parte a que este primer prototipo implementado no incorpora ningún método de optimización de consultas, y además está basado en un primer prototipo de la máquina abstracta Carbayonia que no está optimizado.

El envoltorio objeto-relacional (ObjectDRIVER) tiene un tiempo de inserción medio, del orden de casi tres veces menor que el gestor orientado a objetos. Sin embargo, hay que tener en cuenta lo especificado en el apartado 18.3.1, en el que se indica que cuando se abre la base de datos hay que especificar la correspondencia entre los atributos de las clases y los campos de las tablas relacionales correspondientes, y este tiempo no es considerado en las mediciones. Además, el motor de ObjectDRIVER, es un motor relacional (MsAccess), y se aprovecha, por tanto, de la experiencia acumulada a lo largo de los años en dicha tecnología.

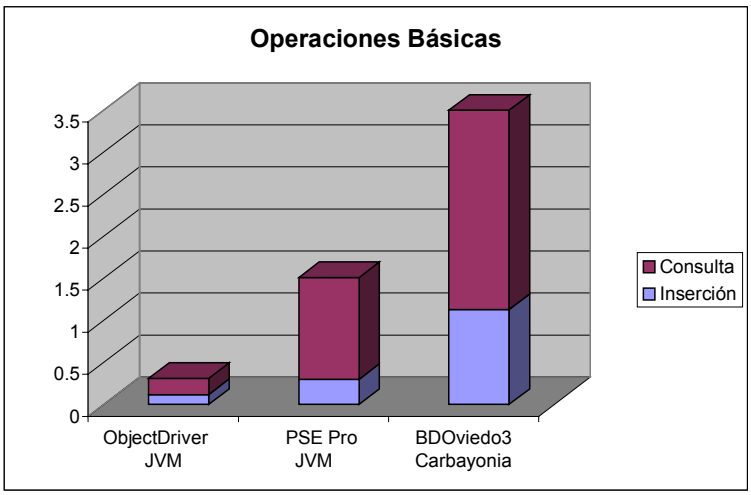


Figura 18.4. Gráfico operaciones de inserción y consulta

18.3.4 Rendimientos considerando la indexación en BDOViedo3

El objetivo de este apartado es comprobar el rendimiento del sistema implementado con otros gestores, cuando se emplea una técnica de indexación adecuada al tipo de

consulta. Es decir, para el estudio seleccionaremos siempre el índice que mejor se comporte (que en este caso y por el tipo de consulta a realizar es SC).

Las comparaciones en este caso se centran en la declaración explícita de índices por parte del programador. Va a declararse un índice sobre el campo sobre el que se centra la consulta. ObjectDRIVER no permite la declaración explícita de índices. Esta es la razón por la que se consideran PSE Pro y BDOviedo3 únicamente.

En cuanto a PSE Pro hay que decir que soporta un mecanismo de indexación mucho más básico que BDOviedo3. Tal y como se comentó en el capítulo 5, PSE Pro solamente ofrece posibilidad de indexar la colección *OSTree_Set* que se corresponde con las características de un conjunto o *Set*, y los índices, se aplicarán solamente sobre atributos de tipos simples, y no sobre agregaciones.

18.3.4.1 Inserción y Consulta

Los resultados para ambas operaciones una vez que se han especificado índices son los que se muestran en la siguiente tabla.

Gestor	Inserción	Consulta
PSE Pro	0,3	1,32
BDOviedo3	1,14	1,83

Tabla 18.7. Tiempos medios de inserción y consulta aprovechando indexación

Como se puede apreciar el efecto de la utilización de índices en la operación de inserción no es significativo (en el caso de BDOviedo3 es porque la técnica de indexación que se emplea es la SC que se vio con anterioridad que es la que menos sobrecarga supone en la inserción y borrado).

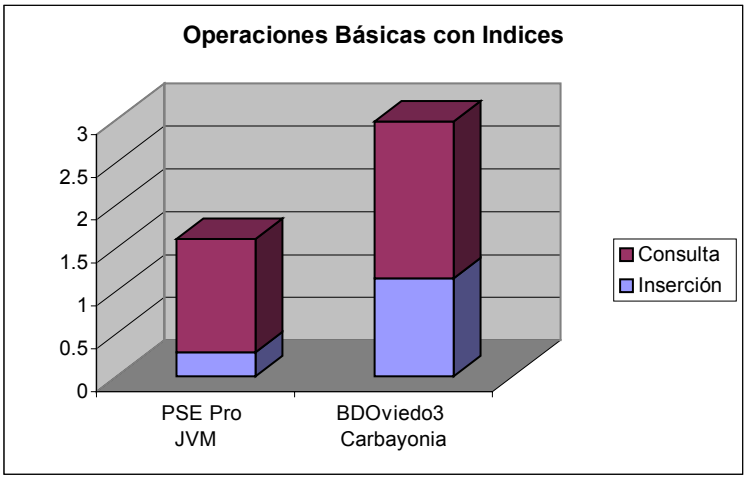


Figura 18.5. Gráfico operaciones básicas empleando indexación

Con relación a las consultas los tiempos han variado. Curiosamente, en el caso de PSE Pro se ha incrementado el tiempo de respuesta a la hora de realizar las consultas utilizando índices para los atributos definidos por el programador. Esto puede obedecer al funcionamiento interno del optimizador de consultas. Sin embargo, se puede observar que para el caso de BDOviedo3 el rendimiento ha mejorado considerablemente.

	Velocidad relativa de la máquina de Java	Orden de magnitud de la velocidad relativa la máquina de Java
ARRAY	123,5	2
INTEGER	12,7	1
NEW	115,8	2
STRING	27,9	1

Tabla 18.8. Tabla comparativa máquina virtual de Java y Carbayonia

18.3.5 Conclusiones

Del estudio realizado se puede concluir que el envoltorio objeto-relacional ha ofrecido los mejores tiempos para todas las operaciones, luego podemos suponerlo como el claro vencedor de esta comparación. Sin embargo, hay que tener en cuenta dos consideraciones que tienen una influencia directa sobre el resultado:

- En primer lugar, es necesario que el usuario cree los dos esquemas (orientado a objetos y relacional), y que posteriormente, previa a cualquier operación con la base de datos se cree la correspondencia entre los atributos de las clases y los campos de las tablas relacionales. Esto, además de dejar patente la desadaptación que hay entre los dos modelos, y de suponer un trabajo poco gratificante para el programador, no es considerado en ningún momento en las mediciones.
- En segundo lugar, el motor que se está empleando realmente es un motor relacional comercial muy utilizado hoy en día (MsAccess). El optimizador de consultas empleado, se beneficia por tanto de los años de experiencia acumulada en torno a la tecnología relacional y de la que carece la tecnología de objetos en lo relativo a las bases de datos.

Por otro lado, PSE Pro se comporta mejor que BDOviedo3. Sin embargo, a este respecto también hay que realizar algunas puntualizaciones:

- BDOviedo3 está basado en un primer prototipo de máquina abstracta sin optimizar, mientras que PSE Pro es ejecutado sobre la máquina virtual de Java que es una máquina mucho más madura, con fines comerciales, y que ha recibido un gran esfuerzo de desarrollo.
- Si se toman como referencia los datos que aparecen en la Tabla 18.8 en la que se aprecian los ordenes de magnitud de la máquina de Java con relación a Carbayonia para operaciones realizadas sobre los tipos de datos que se indican, y suponiendo que esta diferencia pueda mantenerse con relación a los accesos a disco (por supuesto mucho más optimizada en la de Java que en Carbayonia), los resultados no son en absoluto desalentadores.
- BDOviedo3, es un primer prototipo experimental que proporciona un mecanismo de indexación extensible, y no incorpora ningún tipo de optimizador de consultas, mientras que PSE Pro si lo hace. Además, en el desarrollo de este primer prototipo el rendimiento no es el aspecto que más se ha primado, estando por tanto, sujeto a mejoras que tendrían una influencia directa en el comportamiento del sistema.

De esto se puede concluir que el sistema integral orientado a objetos, y en concreto BDOviedo3, aún siendo un proyecto de investigación apenas iniciado ha reflejado una buena línea de investigación en la que se puede seguir trabajando y augurar buenos resultados.

Capítulo 19 Trabajo relacionado

En el presente capítulo se describen tres trabajos que guardan cierta relación con el desarrollado en esta tesis, en algunos aspectos de la misma.

En primer lugar, el sistema GOA++, que es un gestor de objetos experimental compatible con el estándar ODMG, que pretende servir también como plataforma de test para diferentes mecanismos de indexación, pero únicamente basados en árboles B^+.

En segundo lugar, el sistema TIGUKAT, que es un SGBDOO construido sobre un modelo de objetos propio (no ligado a ningún lenguaje de programación orientado a objetos) con algunas características de SGBD. El modelo de objetos que plantea es comportacional puro y tiene como objetivo que todos los elementos del sistema sean objetos que puedan ser tratados de una forma uniforme, lo que facilitará la extensibilidad del mismo. Incorpora un optimizador de consultas extensible.

En tercer lugar, el sistema MOOD, que es un SGBDOO que está relacionado con el tratado en esta tesis en la medida que está basado en un gestor de almacenamiento básico como EXODUS que le proporciona funcionalidades básicas al núcleo como la gestión del almacenamiento y la concurrencia.

19.1 GOA++

GOA++ [MZB+98] es un gestor de objetos experimental que pretende servir como plataforma de experimentación para el estudio de soluciones de la tecnología OO para los SGBD. El modelo de objetos de GOA++ es compatible con el estándar ODMG 2.0, empleando ODL como lenguaje de definición y OQL como lenguaje de consulta.

A pesar de no ser todavía un gestor de objetos completo (carece por ejemplo de la gestión de transacciones) ofrece una serie de recursos ya implementados:

- Persistencia y servicios de gestión de colecciones de objetos
- Procesamiento de predicados y consultas sobre colecciones
- Agrupamiento de objetos en disco
- Gestión de esquemas
- Gestión de caché
- Distribución y paralelismo

19.1.1 Arquitectura

Posee una arquitectura cliente-servidor (Figura 19.1), siendo la unidad de transferencia el objeto, y empleando como base para la comunicación el protocolo TCP-IP.

En el núcleo de GOA++ se fija toda la parte de gestión de objetos, incluyendo los servicios de persistencia de objetos con gestión de la caché, manipulación de atributos e

integridad de relaciones, gestión de esquemas y procesamiento de consultas OQL (haciendo uso de las técnicas de indexación implementadas).

GOA++ posee además API's para ser utilizadas directamente desde C++ o Java.

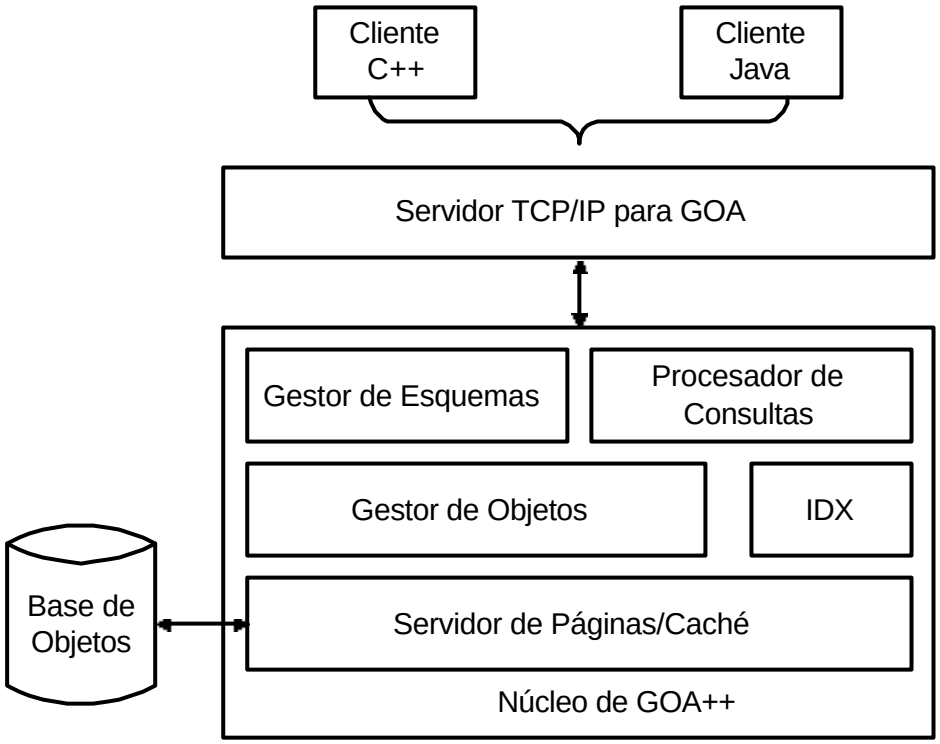


Figura 19.1. Arquitectura de GOA++

19.1.1.1 Gestor de esquemas

GOA++ incorpora un procesador de ODL que representa la información del esquema que debe ser almacenada, en una estructura de datos que será empleada por el gestor de esquemas.

El gestor de esquemas está construido sobre el gestor de objetos, y utiliza las funcionalidades de éste último para crear los tipos como objetos almacenados. Las funcionalidades principales de este gestor de esquemas son la creación y eliminación de tipos almacenados. De momento no tiene implementadas las restricciones que no le permitan borrar un tipo si tiene instancias, si es supertipo de algún otro, o si es referenciado por otro.

19.1.1.2 Procesador de consultas

El procesador de consultas para OQL genera un código intermedio que va a ser ejecutado en el servidor GOA++ con ayuda del procesador de predicados. GOA++ presenta tres estrategias para la evaluación de predicados: ascendente, descendente y ascendente/descendente.

19.1.1.3 Servidor de páginas

El servidor de páginas es el módulo encargado de hacer el acceso a disco directamente buscando y escribiendo las páginas solicitadas. El servidor de páginas mantiene algunas páginas en memoria a modo de caché interna, cuyo tamaño puede ser configurado al inicializar el sistema. Para acelerar el proceso de descubrir si una página está o no en memoria emplea una tabla hash, implementada mediante una lista

doblemente enlazada. Las páginas en la caché son también almacenadas de forma que las próximas veces que el sistema sea inicializado no se parte de cero.

El método para la gestión de páginas libres es muy simple, se basa en una lista, de forma que cuando se solicita una página se informa a la primera de la lista, e inversamente cuando una página es liberada esa pasa a ser la última de la lista. Los números de la primera y la última página son almacenados en una variable en memoria para hacer el proceso de asignación/desasignación de las páginas rápidamente.

La asignación de objetos comunes (entran en una página) se realiza siempre en segmentos, agrupando objetos que serán recuperados siempre en conjunto. Esta segmentación de los objetos puede realizarse a nivel de clase o de relación.

19.1.2 Soporte para distribución basado en CORBA

Los servicios del servidor GOA++ están disponibles para ser empleados en una plataforma de objetos distribuidos compatible con CORBA. Los servicios de GOA++ están disponibles vía CORBA por medio de una interfaz IDL. Esta interfaz está compuesta por operaciones mediante las que un cliente puede acceder a los servicios del gestor. Cada servicio GOA++ posee un método asociado en esta interfaz.

De esta forma un cliente hace una petición pasando los parámetros necesarios. El servidor prepara los parámetros e invoca la operación correspondiente en GOA++; esta operación realiza el servicio, y devuelve la respuesta al objeto servidor GOA++ que a su vez devuelve la respuesta al cliente. De esta forma cualquier cliente CORBA puede utilizar los servicios GOA++.

19.1.3 Procesamiento paralelo

GOA++ permite emplear servidores en paralelo coordinados por un maestro que distribuirá las operaciones sobre los objetos. El paralelismo es implementado independientemente de la arquitectura del sistema de computación, ya que emplea para ello una biblioteca de funciones PVM (*Parallel Virtual Machine*) que crea un nivel de abstracción en la ejecución, y controla los procesos emulando una máquina paralela en ambientes homogéneos/heterogéneos de hardware. La estructura de procesamiento de los servidores en paralelo consiste en el modelo maestro-esclavo, en la que el maestro es el encargado de la creación y control de los procesos esclavos (entre otras cosas).

19.1.4 Experimentación con la indexación

Las estructuras para la indexación en este sistema fueron implementadas en el núcleo del sistema, haciendo uso de los servicios de almacenamiento de objetos y del gestor de la caché. Los índices definidos serán empleados por el procesador de consultas.

El módulo encargado de la indexación (IDX, en la Figura 19.1) está constituido por una serie de clases que permiten la implementación de diferentes técnicas de indexación, siempre que estén basadas en árboles B+. En [SQ99] se presenta de hecho un estudio resultado de la comprobación del rendimiento entre las técnicas SC (*Single Class*) y CH (*Class Hierarchy*).

19.1.5 Relación

Este trabajo como se puede apreciar está relacionado muy directamente con el mostrado en esta tesis doctoral. Están relacionados en la medida en la que ambos proporcionan:

- Un gestor de objetos experimental
- Un mecanismo de indexación extensible con fines experimentales
- Compatibilidad con el estándar ODMG 2.0
- Un soporte para la distribución
- Gestión de esquemas

Sin embargo, también hay claras diferencias entre ellos. La primera de ellas es que BDOviedo3 está construido sobre un SIOO en cuya base se encuentra una máquina abstracta persistente, que proporciona portabilidad (GOA++ no es portable) y homogeneidad a todo el sistema, más un sistema operativo que facilita la distribución sin necesidad de acudir a una capa intermedia como es CORBA. Atendiendo concretamente al mecanismo de indexación diseñado en GOA++, éste es extensible, permitiendo la incorporación de nuevas técnicas de indexación, pero siempre que la nueva técnica esté basada en un árbol B+. BDOviedo3, por su parte, proporciona un mecanismo de indexación más completo, permitiendo la incorporación tanto de nuevas técnicas de indexación (independientemente de que estén basadas en árboles B+ o no), como de nuevos operadores.

19.2 TIGUKAT

Es un SGBDOO que se basa en el desarrollo de un modelo de objetos independiente del lenguaje de programación, que es extendido con características de un SGBD. El modelo de objetos de TIGUKAT [ÖPR+93] es comportacional (*behavioral*) puro, de forma que los usuarios no tienen que distinguir entre atributos y métodos.

19.2.1 Modelo de objetos

El modelo de objetos de TIGUKAT es uniforme ya que cada componente del sistema, incluida su semántica, es modelado como objetos con un comportamiento bien definido. Los objetos primitivos de este modelo son:

- **Entidades atómicas** (enteros, reales, strings, etc.).
- **Tipos**, para definir características comunes de los objetos.
- **Comportamientos**, para especificar la semántica de las operaciones que pueden hacerse sobre los objetos.
- **Funciones**, para especificar la implementación de los comportamientos.
- **Clases**, para la agrupación automática de objetos basada en el tipo.
- **Colecciones**, para soportar agrupaciones heterogéneas de objetos.

El modelo separa la definición de las características de un objeto (tipo) del mecanismo para mantener instancias de un tipo particular (clase). Los objetos de un tipo no pueden existir sin una clase, y cada clase es asociada únicamente con un tipo simple. Un tipo debe tener asociada una clase antes de poder crear instancias de ese tipo. El conjunto de objetos de un tipo se conoce como *extensión del tipo*, y el modelo distingue extensión en profundidad de extensión superficial (no considera los subtipos). Una *colección* por el contrario es una agrupación de objetos mantenida por el usuario y en la que los objetos pueden ser de diferentes clases. De hecho, la clase es considerada un subtipo de una colección.

Este modelo separa claramente la definición de un comportamiento de sus posibles implementaciones, lo que significa que un comportamiento común puede tener una implementación diferente para cada uno de los tipos. La semántica de cada operación sobre un objeto es especificada por un comportamiento definido sobre su tipo. Una función (o método) implementa la semántica de un comportamiento. La implementación de un comportamiento particular puede variar sobre los tipos que soporta. Sin embargo, la semántica del comportamiento permanece consistente sobre todos los tipos que soportan el comportamiento.

19.2.2 Lenguajes

TIGUKAT proporciona tres lenguajes: **lenguaje de definición** (TDL, *Tigukat Definition Language*) que soporta la definición de tipos, colecciones, clases, comportamientos y funciones; **lenguaje de control** (TCL, *Tigukat Control Language*) que soporta operaciones específicas de sesión (abrir, cerrar, guardar, etc.), y **lenguaje de consulta** (TQL, *Tigukat Query Language*) para manipular y recuperar objetos.

TQL está basado es una sintaxis extendida de SQL para soportar expresiones de camino en las cláusulas *select-from-where*. Este lenguaje de consulta permite consultar tanto objetos transitorios como persistentes, y permite también especificar si el ámbito de la consulta es la extensión en profundidad o únicamente la superficial para el tipo.

19.2.3 Arquitectura

El modelo de objetos de TIGUKAT está implementado sobre EXODUS, pero de todas las funcionalidades que este gestor proporciona sólo se aprovecha el método de almacenamiento para suministrar persistencia a los objetos TIGUKAT. La implementación del modelo de objetos de TIGUKAT se enlaza con el módulo cliente de ESM para formar un módulo ejecutable que requiere que el proceso servidor esté ejecutándose para realizar operaciones de entrada/salida persistentes.

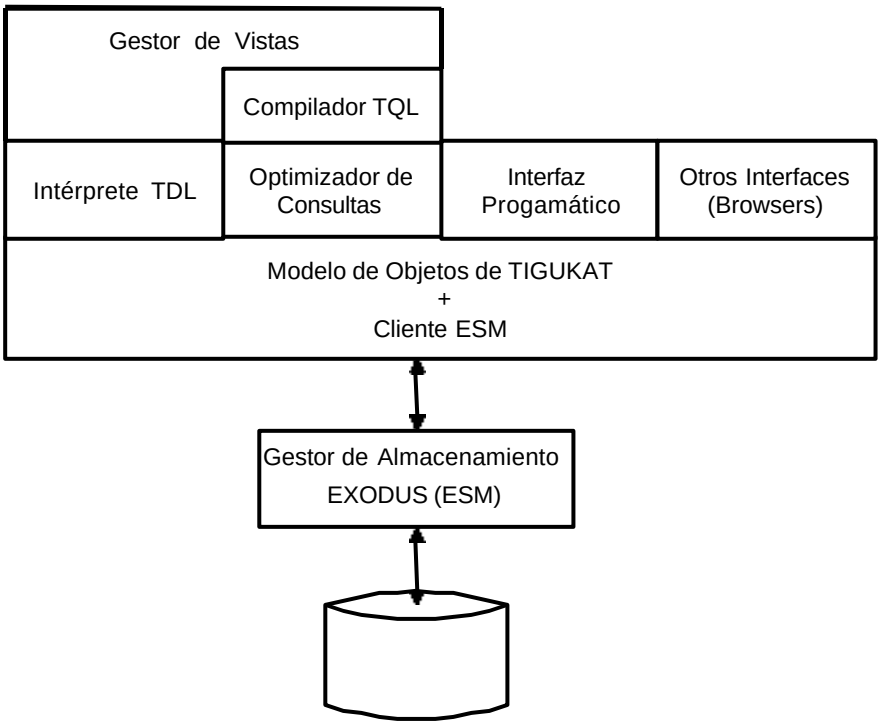


Figura 19.2. Arquitectura de TIGUKAT

El prototipo actual está implementado en C++, pero en vez de hacer una traducción del sistema de tipos de TIGUKAT al de C++, se emplea una clase C++, *TgObject,* que es definida como la plantilla principal para la instanciación de todos los demás objetos.

19.2.3.1 Modelo de consultas

El modelo de consultas es una extensión directa del modelo de objetos, de hecho, las consultas son incorporadas como una especialización de las funciones (T_function), y por tanto, pueden ser empleadas como implementaciones de comportamientos, compiladas y ejecutadas.

Esto hace que las consultas soporten la semántica uniforme de los objetos y puedan ser mantenidas dentro de la base de datos, pudiéndose acceder a ellas por medio del paradigma comportacional del modelo de objetos.

El modelo de consulta es, por tanto, extensible ya que el tipo que representa las consultas puede ser especializado por subtipos, lo que significa que se pueden definir subclases adicionales para nuevos tipos de consultas.

Este modelo de consultas es completado con un cálculo y un álgebra de objetos equivalente.

19.2.3.2 Optimizador de consultas extensible

El optimizador sigue con la filosofía de representar conceptos del sistema como objetos. El espacio de búsquedas, la estrategia de búsqueda, y las funciones de costo son modeladas como objetos. La incorporación de estos componentes del optimizador en el sistema de tipos proporciona extensibilidad por medio del subtipado y la especialización.

De esta forma, el administrador de la base de datos tiene la opción de definir nuevas funciones de costo, y nuevas estrategias de búsqueda o funciones de transformación para nuevas clases de consultas.

19.2.4 Relación

Este sistema, TIGUKAT, guarda una cierta relación con BDOviedo3 en algunos de sus planteamientos y objetivos. Así, TIGUKAT es un SGBDOO basado en el diseño de un modelo de objetos propio, que pretende que todos los elementos del sistema sean objetos con un comportamiento definido, lo que en definitiva tiene como objetivo la extensibilidad del sistema. De esta forma por ejemplo plantea la posibilidad de un optimizador de consultas extensible. Los desarrolladores de TIGUKAT están planteando la incorporación técnicas de indexación en las que pueda apoyarse el optimizador de consultas.

Sin embargo, el modelo de objetos empleado está alejado de los modelos empleados por la mayoría de los lenguajes de programación OO, lo que obliga a una adaptación desde el modelo del lenguaje de programación al modelo de TIGUKAT. Además, tampoco proporciona una opción para la persistencia, con lo que se ve obligado a acudir a un gestor de objetos, como EXODUS, con la consiguiente traducción.

Entre los objetivos de TIGUKAT tampoco figuran ni la portabilidad del sistema, ni por supuesto la interoperabilidad con otros sistemas, ya que los lenguajes de definición (TDL) y consulta (TQL) son propios y no siguen ningún estándar.

19.3 MOOD

MOOD (*Metu Object-Oriented DBMS*) [DAO+95] es un SGBDOO construido sobre el *EXODUS Storage Manager* (ESM) y por lo tanto, algunas de las funciones del núcleo de MOOD como la gestión del almacenamiento, el control de la concurrencia, backup y recuperación son proporcionadas por éste. El resto de funciones del núcleo de MOOD son:

- Optimización e interpretación de consultas (SQL)
- Ligadura dinámica de funciones
- Gestión del catálogo

Además, ESM tiene una arquitectura cliente-servidor y cada proceso MOOD es una aplicación cliente en ESM.

MOOD está codificado en GNU C++ sobre estaciones de trabajo Sun Sparc2.

19.3.1 Modelo de objetos

El sistema de tipos de MOOD se deriva de C++, eliminando así la desadaptación de impedancias entre C++ y MOOD. Como lenguaje de consulta proporciona MOODSQL, un lenguaje de consulta estilo SQL que puede ser invocado desde el lenguaje de programación C++.

En MOOD los datos básicos son *Integer*, *Float*, *LongInteger*, *String*, *Char* y *Boolean*. Cualquier dato de tipo complejo se define usando estos tipos y por aplicación recursiva de los constructores *Tuple, Set*, *List* y *Ref*. Soporta también herencia múltiple.

Las clases en MOOD tienen extensiones, que son implementadas como ficheros ESM. Una clase tiene un identificador de tipo único, que es por el que se accede al catálogo para obtener la información que permite la interpretación de los objetos del ESM (arrays de bytes sin signo).

MOOD permite modificaciones dinámicas del esquema, y las consultas MOODSQL devuelven objetos. Empleando estas dos características un usuario puede almacenar nuevos objetos obtenidos como resultado de una consulta en una base de datos, pero es responsabilidad del usuario colocar la nueva clase en la jerarquía de herencia.

19.3.2 Arquitectura

Los módulos que constituyen el SGBDOO MOOD, así como las relaciones entre ellos pueden observarse en la siguiente figura, y serán analizados brevemente a continuación.

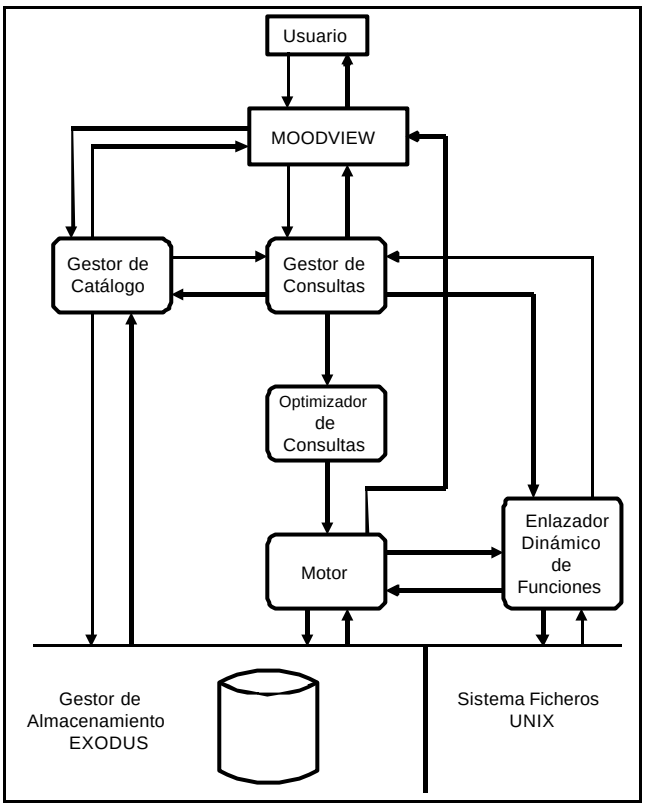


Figura 19.3. Arquitectura de MOOD

19.3.2.1 Interfaz gráfica

MoodView es la interfaz gráfica del sistema que muestra información del esquema, así como los resultados de las consultas gráficamente. Permite también actualizar el esquema.

Para operaciones primitivas que implican una simple llamada a una función, MoodView directamente se comunica con el catálogo; sin embargo operaciones complejas se pasan al gestor de consultas. MoodView pasa las consultas MOODSQL al gestor de consultas sin ninguna modificación.

MOOD además presenta un interfaz textual a la base de datos. Para esta interfaz MOODSQL proporciona comandos para la definición y modificación del esquema.

19.3.2.2 Gestor de consultas

Para comandos de modificación del esquema, el gestor de consultas interactua con el catálogo y en el caso de definiciones de métodos interactua también con el enlazador dinámico de funciones (*dynamic function linker)*.

En el caso de las consultas, el gestor de consultas obtiene la información necesaria desde el catálogo y realiza el chequeo sintáctico y semántico sobre las consultas. Si no detecta error, genera un árbol de consulta que es pasado al optimizador de consultas. Después de la optimización del árbol, ya está preparada para ser ejecutada por el motor. Durante la ejecución, las extensiones de las clases son leídas desde el ESM y los resultados temporales almacenados en el ESM. Si se emplean métodos en las consultas son activados por medio del subsistema enlazador dinámico de funciones. Al final de la ejecución los resultados son devueltos al MoodView.

19.3.2.3 Enlazador dinámico de funciones

Proporciona ligadura tardía de métodos a los objetos durante la interpretación de las consultas. Durante la definición de los métodos este módulo construye objetos compartidos, que almacena en un directorio especial de la jerarquía de ficheros de UNIX, y proporciona la información necesaria sobre los métodos al catálogo por medio del gestor de consultas.

Por otra parte, durante la interpretación de consultas este módulo trae los objetos compartidos de los métodos a memoria y encuentra las direcciones de un método dentro del objeto compartido empleando la signatura del método construido con el gestor de consultas. Después de esto el motor ejecuta el método para cada instancia. El cuerpo del método es compilado con MOODCC, compilador AT&T C++ (versión 2) modificado. Los métodos también pueden incluir llamadas a ESM, lo que les proporciona la capacidad de hacer operaciones con objetos a bajo nivel.

19.3.2.4 Optimizador de consultas

El optimizador MOOD es implementado empleando el generador extensible de optimizadores *Volcano* [Mck93], herramienta que permite construir optimizadores de consultas para SGBD. Los optimizadores generados por Volcano producen el plan de ejecución óptimo cuando las reglas de transformación y funciones soportadas son proporcionadas apropiadamente.

19.3.2.5 Motor de la base de datos

El motor de la base de datos es el subsistema de ejecución de consultas de MOOD. Consta de cuatro partes principales: generador del intérprete de expresiones, intérprete de consultas, subsistema de ejecución de expresiones y buffer de objetos.

19.3.2.6 Gestor del catálogo

Contiene la definición de clases, tipos y funciones miembro en una estructura similar a una tabla de símbolos de un compilador. El catálogo es almacenado sobre el ESM. Las clases del catálogo (*MoodTypes*, *MoodsAttributes* y *MoodsFunctions*) no son diferentes a las definidas por el usuario, por lo que MOODSQL puede emplearse para acceder a la información del catálogo.

MOOD proporciona actualizaciones dinámicas del esquema, sin embargo, los objetos en las extensiones de clases cuya definición ha sido modificada son eliminados.

19.3.3 Relación

Este SGBDOO se caracteriza por el empleo de un gestor de objetos básico (EXODUS) que proporciona parte de las funciones del núcleo, como gestión de almacenamiento o gestión de la concurrencia. Sobre ese núcleo básico se incorporan los demás elementos (gestor del catálogo, gestor de esquemas, interfaz de usuario, etc.). En este sentido es similar a BDOviedo3, que parte de un sistema subyacente que le proporciona la persistencia básica al sistema, y el substrato para la gestión de la distribución, la concurrencia y la seguridad, y sobre ese núcleo básico se incorporan una serie de mecanismos para proporcionar el resto de la funcionalidad de base de datos.

Sin embargo, las diferencias de este sistema con BDOviedo3 son muy abundantes, ya que entre los objetivos de MOOD no se considera la portabilidad del sistema, ni la interoperabilidad con otros SGBDOO (no considera el estándar ODMG), ni por supuesto la extensibilidad.

Capítulo 20 Conclusiones

Al inicio de este trabajo se revisaron las diferentes posibilidades existentes para la gestión de objetos, y se justificaron las basadas en el modelo de objetos como las más acertadas conceptualmente hablando para la gestión de los objetos de las aplicaciones. En esta revisión se apuntó que unas de las posibilidades más empleadas para la gestión de objetos eran las que estaban basadas en el modelo relacional, sin embargo se vio que su uso mayoritario básicamente estaba condicionado a la prioridad que se da a las ganancias comerciales sobre la claridad conceptual y los principios de ingeniería del software.

De las posibilidades para la gestión basadas en el modelo de objetos se concluyó que la más completa era la ofrecida por un SGBDOO o por un gestor de objetos, aunque este último ofrece menos prestaciones. Sin embargo, pese a ser la mejor posibilidad para la gestión de objetos se vio que estos sistemas no estaban exentos de problemas. De entre estos problemas se destacaron varios: *la carencia de interoperabilidad*, *la difícil portabilidad*, la *falta de flexibilidad/extensibilidad* de la que adolecen la mayoría de los ellos, y por último, lo *inadecuado de los sistemas operativos tradicionales* como soporte para estos sistemas.

Se observó que la falta de flexibilidad y extensibilidad es una constante en la mayoría de los SGBDOO existentes. Estos sistemas están constituidos por un núcleo rígido más una serie de mecanismos que utilizan unas políticas (o técnicas) concretas difícilmente modificables. Sin embargo, los SGBDOO por su naturaleza deben adaptarse a una multitud de campos de aplicación diferentes, y es probable que una misma política no proporcione el mejor rendimiento para todas las situaciones.

Otra constante en la construcción de los SGBD actuales es la re-implementación de la mayoría de las abstracciones ofrecidas por los sistemas operativos, ya que éstas normalmente son fijas y resultan insuficientes para las necesidades de los SGBD. Además, los sistemas operativos tradicionales trabajan con unas abstracciones no adecuadas para el paradigma de orientación a objetos lo que provoca un problema de desadaptación de impedancias que repercute en el resto de capas situadas sobre el sistema operativo.

Se planteó entonces la necesidad de un SGBDOO flexible, basado en unos mecanismos básicos que admiten fácilmente la incorporación de diferentes políticas (o técnicas) que le permitan adaptarse con facilidad a los requerimientos de las diferentes aplicaciones. Los requisitos generales que debe cumplir dicho SGBDOO se especificaron en el capítulo 4.

Para la construcción de este SGBDOO se tomó como base un sistema integral OO, constituido por una máquina abstracta OO y un sistema operativo OO. La necesidad de dicho sistema integral se analizó en el capítulo 6, y una posible arquitectura para el mismo fue planteada en el capítulo 7.

La máquina abstracta proporciona un modelo de objetos que será compartido por el resto del sistema y que será el empleado en la construcción del propio SGBDOO, eliminando de esta forma la desadaptación de impedancias antes mencionada, y

facilitando la interoperabilidad. Además, la máquina abstracta dotará de portabilidad a todo el sistema.

El sistema operativo del sistema integral proporciona unas abstracciones (véase capítulos 8 y 9) que pueden ser fácilmente modificadas o ampliadas lo que permitirá que el SGBD no tenga que reimplementarlas de nuevo, consiguiendo así una mayor integración entre ambos elementos.

Bajo este marco se diseñó un SGBDOO cuya arquitectura básica es descrita en el capítulo 10 de esta memoria. De todos los módulos que constituyen dicha arquitectura en esta tesis se profundizó en el motor del mismo, y concretamente, en la consecución de un mecanismo de indexación extensible (véase capítulos 11 y 12) en el que se reflejasen los objetivos antes mencionados, y que permitiera por tanto, la incorporación de nuevas técnicas de indexación. El mecanismo diseñado y la flexibilidad del sistema le convierten en una extraordinaria plataforma de experimentación que nos permite obtener resultados sobre el comportamiento de las diferentes técnicas empleadas.

Este SGBDOO necesitaba un lenguaje que permitiera su manipulación y consulta. Para ello, y tras analizar diferentes posibilidades se optó inicialmente por la adopción de los lenguajes propuestos por el estándar ODMG en su *binding* para Java. La justificación de dicha elección fue analizada en el capítulo 13. También es necesario un entorno visual usable que facilite el trabajo con dicho SGBDOO. En el capítulo 14 se exponen los requisitos básicos que debe proporcionar dicho entorno.

Con el fin de evaluar este sistema se implementó un primer prototipo (véase capítulos 16 y 17) y se realizó un estudio (véase capítulo 18) que pretendía servir como indicador de la bondad del sistema diseñado. El estudio realizado tenía como objetivo principal comprobar el rendimiento del sistema aquí diseñado en comparación con otros gestores existentes. La justificación de los gestores seleccionados para la comparación se realiza en el apéndice B. Otro de los objetivos del estudio era aprovechar la posibilidad que ofrecía el mecanismo de indexación diseñado como plataforma de experimentación, para comprobar el rendimiento de diferentes técnicas de indexación. Las técnicas de indexación comparadas inicialmente han sido la *Single Class*, *CH-Tree* y *Path Index*. Adicionalmente, se desarrolló también una herramienta que implementa los lenguajes propuestos por el estándar ODMG 2.0 en su *binding* para Java, y que genera código para diferentes gestores persistentes con la finalidad de comprobar los rendimientos de los mismos.

20.1 Resultados destacables

A continuación se resumen algunos resultados que se pueden destacar, agrupados en diferentes apartados.

20.1.1 Uniformidad

En la construcción del motor del SGBDOO se mantiene totalmente la uniformidad en la orientación a objetos. Los objetos encargados de proporcionar las distintas funcionalidades del gestor siguen siendo objetos del mismo modelo único del sistema integral. No se distinguen del resto de objetos del sistema, como pueden ser por ejemplo, los objetos que proporcionan los diferentes servicios del sistema operativo.

20.1.2 Integración transparente con el sistema operativo

El hecho de que el SGBDOO haya sido implementado sobre un sistema que la única abstracción que ofrece son los objetos y con el mismo modelo de objetos del sistema operativo subyacente, evita la desadaptación de impedancias existente actualmente entre la mayoría de los SGBDOO y los sistemas operativos convencionales. Pero además la flexibilidad ofrecida por el sistema operativo para modificar sus servicios evitará la reimplementación de las abstracciones ofrecidas por el sistema operativo. Estas abstracciones podrán ser extendidas y ampliadas pero no es necesario reimplementarlas completamente.

20.1.3 Portabilidad

Se ha conseguido portabilidad para el SGBDOO. Esta es una característica derivada directamente de la construcción del SGBDOO sobre un sistema integral cuya base la constituye una máquina abstracta.

20.1.4 Interoperabilidad

Independientemente del lenguaje que emplee la base de datos para su manejo, los objetos serán almacenados atendiendo al modelo de objetos de la máquina abstracta, lo que permitirá su recuperación y tratamiento independientemente del lenguaje empleado por el gestor. De esta manera se facilita, por tanto, la existencia de un SGBDOO multi-lenguaje.

20.1.5 Flexibilidad

La separación entre técnicas y mecanismos realizada, por ejemplo en el mecanismo de indexación, se traduce en un incremento de la flexibilidad al ser posible añadir o reemplazar las técnicas que rigen los mecanismos de forma no traumática.

Las técnicas serán implementadas como un conjunto de objetos, con una interfaz determinada para los objetos que implementan los mecanismos. Las técnicas del mecanismo podrán ser cambiadas o podrán añadirse otras nuevas, siempre y cuando los objetos respeten la interfaz especificada.

20.1.6 Extensibilidad

El sistema descrito en este trabajo facilita la extensibilidad, entendida ésta en términos de reutilización de código. El código de la base de datos está organizado como un marco orientado a objetos, en el que cada una de las clases ofrece una determinada funcionalidad, que podrá ser empleada y extendida por cualquier usuario del sistema para adaptarla a nuevas condiciones de operación.

20.1.7 Adaptabilidad

La adaptabilidad era uno de los objetivos a conseguir y ésta queda patente por ejemplo con el mecanismo de indexación diseñado. Si se desea, dicho mecanismo nos permitirá seleccionar la técnica o técnicas que mejor se adaptan a los tipos de datos a gestionar, sin necesidad de incluir el resto de técnicas, y sin que suponga por lo tanto, una sobrecarga en ejecución.

20.1.8 Portabilidad de las aplicaciones

La selección inicial de los lenguajes de manipulación y consulta para la base de datos propuestos por el estándar ODMG 2.0, en su *binding* para Java, proporciona y

garantiza la portabilidad de las aplicaciones desde y hacia otros gestores siempre que sean compatibles con dicho estándar.

20.1.9 Implementación completa de los lenguajes propuestos por ODMG 2.0 en su *binding* para Java

Se ha realizado una implementación completa del estándar ODMG 2.0 (ODL, OML y OQL) en su *binding* para Java, y se ha construido una herramienta que permite la comparación de diferentes gestores de bases de datos que sean compatibles con dicho estándar. El objetivo es tener una referencia de los rendimientos de los diferentes gestores, así como una medida del grado de compatibilidad que dichas herramientas tienen con el estándar.

20.2 Ventajas del sistema diseñado

En función de los resultados destacables mostrados en el apartado anterior y como consecuencia del diseño del sistema realizado, se puede concluir que este sistema proporciona una serie de ventajas de gran interés para la comunidad de base de datos.

20.2.1 Base de datos como sistema de almacenamiento global

Hasta ahora se ha considerado el sistema diseñado desde el punto de vista de un sistema de gestión de bases de datos tradicional, sin embargo, sus características le permiten no ser considerado como un entorno separado del resto del sistema, como en los sistemas tradicionales.

Dada la cercanía entre el sistema integral orientado a objetos y el SGBDOO puede considerarse éste último como parte integrante del entorno de computación. Es decir, los objetos de la base de datos son simplemente unos objetos más dentro del sistema que proporcionan unos servicios. Esto permite transformar el SGBDOO en un elemento que cumpla el papel de los sistemas de ficheros en los sistemas operativos tradicionales.

De esta manera el SGBDOO BDOviedo3 se puede integrar de manera transparente en el entorno de computación como sistema de almacenamiento global, de forma que no tiene porque ser empleado como un sistema independiente del sistema operativo, si no que puede ser utilizado como un sistema de gestión para todos los objetos del sistema. Cualquier objeto que se cree en el sistema quedará automáticamente registrado en la base de datos de forma totalmente transparente. Una posibilidad para conseguir esto puede ser empleando la reflectividad (modificando, por ejemplo, el operador *new)* para que el objeto quede automáticamente registrado en la base de datos en el momento de su creación.

Este planteamiento permite aprovechar todas las ventajas que proporciona un SGBD frente a un sistema de ficheros tradicional (ej. facilidad de consultas), y convierte al SGBD en el almacenamiento global del sistema, evitando de esta forma la dualidad sistema de ficheros-sistemas de bases de datos existente actualmente.

20.2.2 Plataforma de experimentación

El mecanismo de indexación diseñado convierte al sistema en una buena plataforma de experimentación para las técnicas de indexación, permitiendo fácilmente la incorporación de nuevas técnicas que puedan adaptarse a determinados patrones de aplicación y que puedan incrementar el rendimiento para determinados tipos de datos.

Esa incorporación se realiza de forma no traumática, y sin que eso además suponga una sobrecarga en el rendimiento del sistema.

El hecho de contar con una plataforma como ésta, ayudará con posterioridad a reconocer que tipos de índices son adecuados para que tipo de consultas y bajo que condiciones (número de clases en la jerarquía, número de instancias de cada clase, etc). Esto finalmente se traducirá en un mejor rendimiento de los SGBDOO en general, y reforzará por tanto su presencia en el mercado. En definitiva, aportará la experiencia que estos sistemas necesitan en comparación con otros ya totalmente establecidos como son los SGBD relacionales.

Siguiendo este modelo el sistema podría convertirse en plataforma de experimentación para cualquiera de los mecanismos constituyentes del SGBDOO diseñado.

20.2.3 Desarrollo de sistemas de alta productividad

El SGBDOO que nos ocupa es un sistema especialmente dotado para mejorar la productividad en el desarrollo de sistemas. Las causas son básicamente dos:

- **No hay cambio de paradigma**. La programación de aplicaciones de bases de datos es más productiva ya que no es necesario que el programador cambie constantemente de filosofías: una para trabajar con la base de datos y otra para manejar el sistema operativo. Ambos elementos emplean ahora el mismo paradigma de orientación a objetos.
- **Compatibilidad con el estándar**. Como se comentó en el capítulo 13 el lenguaje adoptado por este sistema es el propuesto por ODMG en su *binding* para Java. Esto facilitará tanto la migración de aplicaciones realizadas desde otros sistemas al aquí descrito, siempre que dichos sistemas sean totalmente compatibles con dicho estándar, como el proceso inverso.

20.3 Trabajo y líneas de investigación futuras

Esta tesis sienta las bases para un SGBDOO integrado en un entorno de computación orientado a objetos, pero deja abiertas diferentes líneas de investigación, para mejorar y completar lo ya existente y para desarrollar nuevos elementos. A continuación se reseñan algunas de las líneas de investigación más inmediatas relacionadas con los temas discutidos en esta tesis.

20.3.1 Incorporación del resto de funcionalidades del gestor

Cuando se describió la arquitectura para el SGBDOO se identificaron una serie de módulos encargados de proporcionar la funcionalidad básica del gestor y se describieron las características que deberían proporcionar. Sería necesario realizar un diseño más detallado de cada uno de esos módulos de la misma forma que se realizó con el mecanismo de indexación tratado en el capítulo 12.

20.3.2 Desarrollo de una interfaz de usuario orientada a objetos

Como se mencionó en el capítulo 3, otro aspecto importante de un SGBDOO es la construcción de una interfaz de usuario amigable que facilite el trabajo a los usuarios y mediante la cual estén disponibles todas las funciones proporcionadas por el sistema. Los requisitos para dicho entorno son expuestos en el capítulo 14, pero no han sido todavía plasmados en una interfaz para el sistema.

20.3.3 Implementación de los lenguajes propuestos por el estándar ODMG 3.0

En esta tesis se ha estado trabajando con los lenguajes propuestos por la versión 2.0 del estándar ODMG en su *binding* para Java. Es necesario, por tanto, adaptar el trabajo realizado para incorporar los cambios introducidos en el *binding* para Java de la versión 3.0.

20.3.4 Mejora en la implementación de la máquina

La introducción de optimizaciones en la implementación interna de la máquina es un campo muy amplio. Un aspecto en general es el de la optimización del funcionamiento de la máquina en general, por ejemplo, empleando técnicas de compilación dinámica. Otra mejora en el rendimiento se conseguiría eliminando el nivel de indireccionamiento provocado por el área de instancias y de clases, y consiguiendo que el objeto tuviese directamente una referencia a la clase a la que pertenece (relación 1 a 1 entre clase y objeto).

Relacionado ya directamente con el SGBDOO sería interesante que la máquina estuviese dotada de formas de acceso dinámico, como *hashing*, y no únicamente secuencial como lo es ahora (*vector*), lo que incrementaría mucho el rendimiento. Para dicha implementación podría emplearse por ejemplo *map* de STL, con lo que la mejora en el rendimiento sería notable.

20.3.5 Aprovechamiento de la reflectividad para la construcción del SGBDOO

Como se mencionó en el capítulo 7 el sistema integral está dotado de reflectividad. La *reflectividad* es la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo así como ajustar su comportamiento en función de ciertas condiciones.

Por lo tanto, el siguiente paso en la construcción de este gestor sería su implementación apoyándose en la reflectividad proporcionada por la máquina abstracta [OMA+99b]. Esto hará que se simplifique mucho la programación del motor [OMA+99a] puesto que se podrá acceder a todas las propiedades de los objetos en tiempo de ejecución.

20.3.6 Subsistema de persistencia apoyado en el motor de la base de datos

Aprovechando la reflectividad computacional del sistema [OMA+99c] también se puede pensar en un sistema de persistencia implícita como una nueva forma de añadir la funcionalidad de persistencia a un lenguaje de programación. El sistema definirá un subsistema de persistencia que se apoyará en el motor de la base de datos.

En la mejora de la flexibilidad, el subsistema de persistencia será capaz de llevar a cabo cambios dinámicos en los servicios pedidos al motor (ej. un cambio en la técnica de indexación a utilizar) sin necesidad de modificar la aplicación [OMA+99d]. Tanto el motor como el subsistema de persistencia serán incorporados al sistema integral, para su utilización en el desarrollo de aplicaciones.

Apéndice A Estándar ODMG

ODMG, *Object Database Management Group,* representa el primer intento de estandarización de las bases de datos orientadas a objetos. El grupo que propuso este estándar está encabezado por Rick Cattell, y sus integrantes eran representantes de los principales SGBDOO existentes en el mercado (Barry & Associates, Lucent Technologies, O_2 Technologies, GemStone Systems, Versant Object Technologies, etc). Todos ellos consideraban que la carencia de un estándar frenaba el desarrollo de los SGBDOO y por ello se comprometían también a que sus productos fueran compatibles con dicho estándar. Con posterioridad, este estándar ha cambiado su denominación a *Object Data Management Group*, queriendo abarcar de esta manera cualquier mecanismo de almacenamiento que use la especificación ODMG (no únicamente los SGBDOO).

En este apéndice se hace un breve resumen de las principales características del mismo: arquitectura, modelo de objetos, lenguaje de definición, lenguaje de consulta y *bindings* existentes para los principales lenguajes de programación orientados a objetos.

A.1 Arquitectura

La arquitectura de los SGBDOO es sustancialmente diferente de la de los otros SGBD, ya que más que proporcionar un lenguaje de alto nivel como SQL para la manipulación de datos, un SGBDOO integra transparentemente las capacidades de la base de datos con las del lenguaje de programación de la aplicación, y esta transparencia hace innecesario aprender un lenguaje de manipulación (*Data Manipulation Language DML*) separado, evitando tener que traducir explícitamente los datos entre las representaciones de la base de datos y la de los lenguajes de programación.

A.1.1 Arquitectura inicial: ODMG-93 y ODMG 2.0

Inicialmente el estándar ODMG-93 [CAD+94]-ODMG 2.0 [CBB+97] intentaba estandarizar una arquitectura como la mostrada en la Figura A.1. En esta arquitectura el programador escribe declaraciones para el esquema de la aplicación (datos e interfaces), más un programa fuente para la implementación de la aplicación. El programa fuente se escribe en un lenguaje de programación (PL) como C++, empleando una librería de clases que proporciona la manipulación completa de la base de datos, incluyendo transacciones y consulta de objetos. Las declaraciones del esquema pueden escribirse mediante una extensión del lenguaje de programación, PL ODL, o en un lenguaje de programación independiente ODL. Esta última opción podría emplearse como un lenguaje de diseño de más alto nivel, o para permitir definiciones de esquemas independientes de los lenguajes de programación.

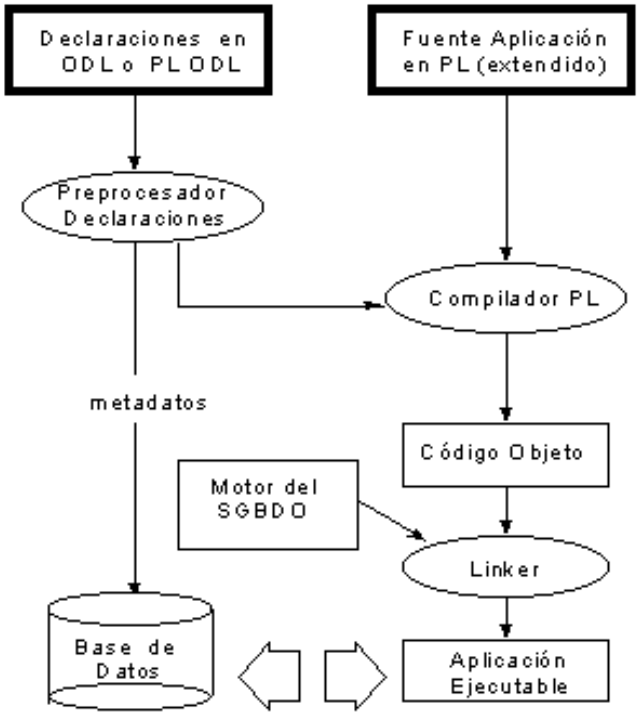


Figura A.1. Arquitectura propuesta inicialmente por ODMG

Las declaraciones y programa fuente se compilan y enlazan con el SGBDOO para producir la aplicación ejecutable. La aplicación accede a bases de datos nuevas o ya existentes, cuyos tipos deben estar de acuerdo con las declaraciones. Las bases de datos pueden ser compartidas por otras aplicaciones sobre una red. El SGBDOO proporciona un servicio para la gestión de bloqueos y transacciones, permitiendo que los datos sean almacenados en la caché en la aplicación.

A.1.2 Arquitectura Actual : ODMG 3.0

En esta versión [CBB+99] ya se produce un cambio en el enfoque del estándar. Por supuesto sigue manteniendo la misma filosofía de considerar los objetos de la base de datos como objetos en el lenguaje de programación, en uno o más lenguajes de programación existentes y extendiendo éstos con control de concurrencia, consultas asociativas, etc. Sin embargo, ahora se extiende para permitir cualquier tipo de almacenamiento, no sólo orientado a objetos (ej. relacional). En esta versión es sustituida la palabra SGBDOO por SGDO (Sistema de Gestión de Objetos, ODMS). De esta forma, ahora el esfuerzo queda centrado en conseguir aplicaciones portables, esto es, aplicaciones que pueden ejecutarse en más de un producto, por lo que para ello es necesario que sean portables tanto el esquema de la base de datos, como los *binding*s a los lenguajes de programación, como los lenguajes de manipulación y consulta.

A.2 Características del modelo de objetos

A continuación se van a resumir muy brevemente las características básicas del modelo propuesto por ODMG. Este modelo tiene dos primitivas básicas que son los objetos y los literales. Los primeros tienen identificadores, mientras que los segundos no.

A.2.1 Objetos

Los objetos se crean invocando operaciones de creación especificadas en una interfaz (*ObjectFactory*) proporcionada al programador por la implementación al ligarla a un lenguaje de programación. Todos los objetos tienen una interfaz que es heredada implícitamente por las definiciones de todos los objetos definidos por el usuario (incluye operaciones de copia, de bloqueo, de comparación, etc.).

A.2.1.1 Identificadores

Los identificadores de objetos permiten distinguir un objeto de otro dentro de una base de datos, ya que su valor es único dentro de la misma. La identidad del objeto no varía aunque se cambie el valor de los atributos del objeto o las relaciones en las que participa. Los identificadores son generados por el propio SGDO, y aunque existen muchas formas posibles de generarlos el modelo de objetos no se decanta por ninguno.

A.2.1.2 Nombre

A un objeto además de un identificador se le pueden asociar uno o más nombres que sean significativos para el programador o el usuario. Esos nombres serán empleados como puntos de entrada en la base de datos desde las aplicaciones (*roots*), y para ello el SGDO proporciona una función que permite llegar desde el nombre del objeto al objeto mismo.

A.2.1.3 Vida de un objeto

La vida de un objeto determina como es gestionada la memoria y el almacenamiento asignado a ese objeto. La vida se especifica cuando se crea el objeto. Este modelo soporta dos modos: *transitorio* (la memoria asignada es gestionada por el lenguaje de programación) y *persistente* (la memoria y almacenamiento son gestionados por el SGDO).

La vida de un objeto es independiente del tipo. Un tipo puede tener instancias que son persistentes y otras que son transitorias. Esto permite que los objetos persistentes y transitorios puedan ser manipulados empleando las mismas operaciones.

A.2.1.4 Tipos de objetos

En este modelo los tipos de los objetos pueden ser:

- **Atómicos.** Un tipo objeto atómico es definido por el usuario. En este modelo no hay tipos objeto atómicos predefinidos.
- **Estructurados.** El modelo de objetos incluye los siguientes objetos estructurados: *Date, Interval, Time* y *Timestamp*, definidos como en la especificación ANSI SQL por medio de una serie de interfaces.
- **Colección.** Los objetos colección están formados por instancias de tipos atómicos, otras colecciones o un tipo literal, sin embargo, todos los elementos de la colección deben ser del mismo tipo. El modelo especifica tres interfaces para la manipulación de colecciones:
    - *Collection*, que incluye operaciones para insertar y borrar un elemento, crear un iterador, indicar el número de elementos, conocer si está vacía o llena, etc.

- *Iterator*, que incluye operaciones como obtener el elemento actual, ir al siguiente, reemplazar un elemento (en arrays o listas), posicionar el iterador en el primer elemento, etc.
- *BidirectionalIterator*, que permite el recorrido bidireccional de una colección ordenada y para ello incluye un método que permite ir a la posición anterior.

Las colecciones soportadas por el modelo son: *Set* (colección no ordenada de elementos sin duplicados), *Bag* (colección no ordenada con duplicados), *List* (colección ordenada de elementos), *Array* (colección ordenada de tamaño variable que puede ser localizada por posición) *y Dictionary* (secuencia no ordenada de pares valor-clave con claves no duplicadas). Para cada una de estas colecciones propone una interfaz (que, por supuesto, hereda de *Collection*) con una serie de operaciones para manipularlas.

A.2.2 Literales

Los literales no tienen identificadores. El modelo de objetos soporta cuatro tipos de literales: *atómicos*, *colecciones*, *estructurados* y *null* (para cada tipo literal existe otro literal que soporta el valor null). Incluye también un tipo *table* para expresar tablas SQL, que es semánticamente equivalente a colecciones de estructuras.

- Los **literales atómicos** (*long, short, unsigned long, unsigned short, float, double, boolean, octet, char, string* y *enum*) deberían ser proporcionados por el lenguaje de programación al que se ligue el modelo de objetos, y si el lenguaje no los contiene entonces una librería de clases debería proporcionarlos.
- Los **literales colección** son similares a los objetos colección (*set, bag, list, array, dictionary*) pero no tienen identificadores de objetos. Sus elementos pueden ser de tipo literal o de tipo objeto.
- Un **literal estructurado,** o simplemente una estructura, tiene un número fijo de elementos cada uno de los cuales tiene un nombre de variable y puede contener un valor literal o un objeto. Los tipos estructura soportados por el modelo son *date, interval, time* y *timestamp*, y además, permite la definición de estructuras por el usuario, empleando para ello el generador *struct*. El modelo soporta conjuntos de estructuras, estructuras de conjuntos, arrays de estructuras, etc.

A.2.3 Tipos

Tanto los objetos como los literales se pueden categorizar por sus *tipos*. Un tipo tiene una *especificación* externa y una o más *implementaciones*. La especificación define las características externas del tipo, es decir, los aspectos visibles al usuario como son las *operaciones* (que pueden ser invocadas sobre sus instancias), las *propiedades* o variables de estado cuyos valores pueden ser accedidos, y cualquier *excepción* que pueda ser elevada por esas operaciones. La implementación por el contrario define aspectos internos de los objetos del tipo (la implementación de las operaciones y otros detalles).

Una especificación externa de un tipo es una descripción independiente de la implementación de las operaciones, propiedades y excepciones. Una interfaz especifica únicamente el comportamiento abstracto de un tipo objeto. Una definición de clase especifica tanto el comportamiento como el estado abstracto de un tipo objeto, y en la definición de un literal se define únicamente el estado abstracto de un tipo literal.

Una implementación de un tipo objeto consta de una representación y un conjunto de métodos. La representación es una estructura de datos que se deriva del estado abstracto del tipo al ligarlo a un lenguaje (para cada propiedad en el estado abstracto hay unas variables instancia del tipo apropiado). Los métodos son cuerpos de procedimientos que se derivan del comportamiento abstracto del tipo al ligarlo al lenguaje. Para cada operación definida en el comportamiento abstracto del tipo se define un método. Este método implementa el comportamiento visible de un tipo objeto.

Todos los elementos de un tipo tienen el mismo conjunto de propiedades y un comportamiento común. Un tipo puede tener más de una implementación, aunque solamente se suele emplear una en un programa particular. El hecho de separar las especificaciones de las implementaciones es un paso hacia los accesos multilenguaje a objetos de un tipo simple, y hacia la compartición de objetos entre entornos de computación heterogéneos.

Identidad y compatibilidad de tipos

El modelo ODMG es fuertemente tipado. Cada objeto o literal tiene un tipo y cada operación requiere operandos con tipo. Dos objetos o dos literales tienen el mismo tipo si han sido declarados instancias del mismo tipo. La compatibilidad de tipos sigue la relación de subtipado definida por la jerarquía de tipos, es decir, una instancia de un subtipo puede ser asociada a una variable del supertipo, pero no a la inversa. Además, el modelo no proporciona conversiones implícitas entre tipos.

Dos literales atómicos tienen el mismo tipo si pertenecen al mismo conjunto de literales. Dependiendo de la ligadura al lenguaje de programación pueden producirse conversiones implícitas entre tipos literales escalares, sin embargo, no hay conversión implícita para literales estructurados.

A.2.3.1 Herencia de comportamiento

El modelo de objetos de ODMG soporta la herencia basada en la relación de subtipo/supertipo (ISA). Soporta también herencia múltiple del comportamiento de objetos, así es posible que un tipo herede operaciones que tienen el mismo nombre pero parámetros diferentes desde dos interfaces diferentes, sin embargo el modelo excluye esta posibilidad no permitiendo la sobrecarga de nombres durante la herencia.

El subtipado pertenece a la herencia de comportamiento únicamente, así las interfaces pueden heredar de otras interfaces y las clases pueden heredar de interfaces. Sin embargo, debido a las ineficiencias y ambigüedades de la herencia múltiple de estado las interfaces no pueden heredar de clases, ni las clases pueden heredar de otras clases.

A.2.3.2 Herencia de estado

Además de la relación ISA que define la herencia de comportamiento entre tipos objeto, el modelo ODMG define una relación EXTENDS para la herencia de estado, que se aplica únicamente a tipos objeto (los literales no pueden heredar estado). Representa una relación de herencia simple entre dos clases, y por tanto la clase subordinada hereda todas las propiedades y todo el comportamiento de la clase que extiende.

A.2.3.3 Extensiones

La extensión de un tipo es el conjunto de todas las instancias del tipo dentro de una base de datos particular. Si un objeto es una instancia de un tipo, entonces

necesariamente será miembro de la extensión de ese tipo. Si este tipo es subtipo de otro, entonces la extensión del primero será un subconjunto de la extensión del segundo. El diseñador de la base de datos puede decidir si el SGDO mantiene automáticamente la extensión de cada tipo.

A.2.3.4 Claves

Las claves permiten identificar unívocamente instancias individuales de un tipo. Estas claves pueden ser simples (una única propiedad) o compuestas (conjunto de propiedades). Un tipo debe tener extensión para tener clave.

A.2.4 Propiedades

El estado de un objeto es definido por los valores que lleva en un conjunto de propiedades a las que el usuario puede acceder. En este modelo se definen dos clases de propiedades: atributos y relaciones.

A.2.4.1 Atributos

Un atributo es de un tipo determinado y su valor es siempre un literal o un identificador de objeto. Es común que los atributos sean implementados como estructuras de datos, pero algunas veces es apropiado implementarlos como un método. Un atributo no es un objeto y por tanto, no tiene identificador, por lo que no es posible definir atributos de atributos o relaciones entre atributos.

A.2.4.2 Relaciones

Una relación es definida entre dos tipos, cada uno de los cuales debe tener instancias que son referenciables por identificadores de objetos. Sólo se soportan relaciones binarias que pueden ser uno-a-uno, uno-a-muchos ó muchos-a-muchos. Las relaciones no tienen nombre ni identificador, y se definen implícitamente mediante la declaración de *caminos de recorrido* (*traversal paths)* que permiten a las aplicaciones emplear las conexiones lógicas entre los objetos que participan en la relación. Estos *caminos de recorrido* son declarados en pares, uno para cada dirección de recorrido de las relaciones binarias.

El SGDO es responsable de mantener la integridad referencial de las aplicaciones, de tal manera que si un objeto que participa en una relación es borrado entonces cualquier *camino de recorrido* a ese objeto debe ser también borrado. Conviene destacar que las relaciones no son punteros, y que además el modelo permite que un atributo referencie a otro objeto, pero esto no debe ser considerado como una relación en sentido estricto ya que en este caso no se garantiza la integridad referencial.

La implementación de una relación es encapsulada por operaciones públicas que añaden o borran miembros de la relación, más una serie de operaciones sobre las clases implicadas que permiten controlar la integridad referencial.

A.2.5 Operaciones

El comportamiento de un objeto es definido por el conjunto de operaciones que pueden ser ejecutadas sobre o por el objeto. Para cada operación hay que especificar su nombre, el nombre y tipo de cada uno de sus argumentos, los tipos de los valores devueltos y los nombres de cualquier excepción que la operación pueda generar.

Las operaciones se definen únicamente sobre tipos simples. En este modelo no se permiten operaciones que existen independientemente del tipo, o bien operaciones

definidas sobre dos o más tipos. Permite que diferentes tipos tengan operaciones definidas con el mismo nombre (sobrecarga), realizando la selección de la operación a ejecutar en función del tipo más específico del objeto proporcionado como el primer argumento de la llamada actual.

El modelo supone una ejecución secuencial de las operaciones, pero no excluye que un SGDO aproveche un soporte multiprocesador.

A.2.5.1 Excepciones

Las excepciones en este modelo son objetos y tienen una interfaz que les permite estar relacionadas con otras excepciones en una jerarquía de generalización-especialización. Soporta dinámicamente manejadores de excepciones anidadas. Las operaciones pueden generar excepciones y las excepciones pueden comunicar los resultados de las mismas.

A.2.6 Características de bases de datos

A.2.6.1 Metadatos

Los metadatos representan información descriptiva acerca de los objetos de la base de datos que definen su esquema. Esta información es empleada por el SGDO para definir la estructura de la base de datos y guiar en tiempo de ejecución el acceso a la misma. Los metadatos son almacenados en un repositorio (*ODL Schema Repository*) que además es accesible a herramientas y aplicaciones empleando las mismas operaciones que aplican los tipos definidos por el usuario.

Este modelo especifica un conjunto de interfaces que definen la estructura interna del repositorio, y que son definidas en ODL empleando relaciones que definen el grafo de interconexión entre metaobjetos (producidos por ejemplo en la compilación del fuente ODL). Todos los metaobjetos son agrupados en un módulo (*module ODLMetaObjects*).

A.2.6.2 Control de bloqueo y concurrencia

El modelo ODMG emplea un mecanismo convencional basado en bloqueo para el control de la concurrencia. El SGDO soporta la serialización monitorizando peticiones para bloqueos y concediendo un bloqueo solamente si no existe conflicto de bloqueos. El modelo soporta como política por defecto un control de concurrencia pesimista, pero no excluye un SGDO que soporte un rango más amplio de políticas de control de concurrencia.

Tipos de bloqueo

Se permiten tres tipos de bloqueo: *Read* (permite acceso compartido a un objeto), *Write* (indica acceso exclusivo al objeto, entra en conflicto con otro *Write* o *Read*), y *Upgrade* (se usa para prevenir estancamientos que se producen cuando hay bloqueos de lectura sobre un objeto y se intenta hacer un bloqueo de escritura sobre el mismo). Los *Upgrade* son compatibles con *Read*, pero entran en conflicto con otros *Upgrade* o *Write*. Los estancamientos se evitan obteniendo inicialmente bloqueos *Upgrade*, en lugar de *Read*, para todos los objetos que se vayan a modificar. Esto evita un conflicto cuando se realiza un bloqueo *Write* para modificar el objeto. El bloqueo permanece hasta que la transacción finaliza o es abortada.

Bloqueo implícito y explícito

El bloqueo puede ser implícito (se obtiene bloqueo *Read* cada vez que un objeto es accedido y bloqueo *Write* cada vez que un objeto es modificado) y explícito. Los bloqueos *Read* y *Write* pueden ser obtenidos tanto implícita como explícitamente, mientras que el *Upgrade* sólo puede obtenerse explícitamente (operaciones *lock* y *try_lock* de la interfaz *Object*).

A.2.6.3 Modelo de transacciones

Los programas que usan objetos persistentes se organizan en transacciones, de tal forma que cualquier acceso, creación, modificación y borrado de objetos persistentes debe ser realizado dentro de una transacción. Además, el resultado de la ejecución de transacciones concurrentemente debe ser el mismo que se obtendría con una ejecución secuencial de las mismas (*serialización*). Por otro lado, los objetos transitorios en un espacio de direcciones no están sujetos a la semántica de las transacciones

Proporciona dos tipos para soportar las transacciones dentro de un SGDO: *TransactionFactory* (empleado para crear objetos de tipo *Transaction*) y *Transaction*, que incorpora una serie de operaciones que permiten ejecutar la transacción, abrirla, abortarla, ligarla a un hilo de ejecución, etc.

El modelo supone una secuencia lineal de transacciones que se ejecutan dentro de un hilo de control. Cuando el lenguaje seleccionado para el *binding* soporta múltiples hilos en un espacio de direcciones, debe proporcionarse el aislamiento entre los hilos, aunque el aislamiento debe proporcionarse también entre hilos en diferentes espacios de direcciones o hilos ejecutándose sobre diferentes máquinas. El modelo de transacciones ni requiere ni excluye el soporte para transacciones que abarcan múltiples hilos, múltiples espacios de direcciones o más de una base de datos lógica. ODMG no define una interfaz para transacciones distribuidas (abarcan más de un proceso y/o más de una base de datos), y no es obligatorio que los sistemas lo soportan, pero si es así deben seguir la norma ISO XA.

A.2.6.4 Operaciones con la base de datos

Un SGDO puede gestionar una o más bases de datos lógicas, cada una de las cuales puede ser almacenada en una o más bases de datos físicas. Cada base de datos lógica es una instancia del tipo *Database* que es proporcionado por el SGDO. Para gestionar las bases de datos incluye dos interfaces *DatabaseFactory* (crea instancias de *Database*) y *Database*, que incorpora una serie de operaciones para abrir, cerrar, buscar un identificador de un objeto a partir de su nombre y asociar un nombre a un objeto. Incluye también una operación para acceder a los metaobjetos que definen el esquema de la base de datos. Además, puede incorporar operaciones diseñadas para la administración de la base de datos: copiar, mover, crear, eliminar, reorganizar, backup, etc.

A.3 Lenguajes de especificación de objetos

En este apartado del estándar se definen lenguajes de especificación independientes de los lenguajes de programación que se pueden emplear para definir el esquema, operaciones y estado de SGDO compatibles con el estándar. El objetivo principal de estos lenguajes es facilitar la portabilidad de esquemas de bases de datos entre implementaciones compatibles con el estándar. Proporcionan además un paso para la interoperabilidad entre múltiples fabricantes de BDOO.

A.3.1 Lenguaje de definición de objetos

El lenguaje de definición de objetos (*Object Definition Language ODL*) es un lenguaje de especificación que se emplea para definir las especificaciones de tipos objeto de acuerdo al modelo ODMG. Permite la portabilidad de esquemas de bases de datos entre SGDO.

A.3.1.1 Características

Lenguaje de programación no completo

ODL no es un lenguaje de programación completo; es un lenguaje de definición de datos (*Data Definition Language DDL*) para tipos objeto. Define las características de los tipos, incluyendo sus propiedades y operaciones. ODL define solamente las signaturas para las operaciones pero no permite la definición de los métodos que implementan estas operaciones.

No ligado a la sintaxis de ningún lenguaje de programación concreto

ODL define tipos objetos que pueden ser implementados en diferentes lenguajes de programación, por lo tanto no está ligado a la sintaxis de ningún lenguaje concreto. Un esquema especificado en ODL puede ser soportado por cualquier SGDO compatible con ODMG y con diferentes lenguajes de implementación. Esta portabilidad es necesaria para que una aplicación sea capaz de ejecutarse con cambios mínimos en una variedad de SGDO.

Las ligaduras del ODL a lenguajes como C++, Smalltalk y Java están diseñadas para introducirlo fácilmente en la sintaxis declarativa de su lenguaje de programación. Debido a las diferencias inherentes en los modelos de objetos nativos a estos lenguajes de programación, no es siempre posible mantener una semántica consistente entre lenguajes de programación (versiones específicas de ODL), pero el objetivo es minimizar estas inconsistencias.

Extensión del IDL de CORBA

La sintaxis de ODL extiende la de IDL (*Interface Definition Language*) desarrollado por OMG como parte de CORBA (*Common Object Request Broker Architecture*). ODL añade a IDL las construcciones requeridas para especificar la semántica completa del modelo de objetos de ODMG.

ODL además proporciona un contexto para integrar esquemas desde múltiples fuentes y aplicaciones. De hecho, estos esquemas pueden haber sido definidos con cualquier número de modelos de objetos y lenguajes de definición de datos. Así, otras organizaciones como STEP/PDES, ANSI X3H2 o ANSI X3H7 han desarrollado otros modelos de objetos e incluso lenguajes de definición, y cualquiera de estos modelos puede ser traducido a una especificación ODL. Esta base común permite que varios modelos sean integrados con semántica común. Una especificación ODL puede ser llevada a cabo concretamente en un lenguaje de programación orientado a objetos como C++, Smalltalk o Java.

A.3.1.2 Especificación

Un tipo es definido en ODL especificando su interfaz o su clase. En la definición del tipo primero aparecen sus características (que no son aplicables a las instancias), y a continuación ya aparece la lista de propiedades (atributos y relaciones) y operaciones.

Las características del tipo incluyen información del supertipo, el nombre de la extensión de la clase si es que se decide incluir, y la especificación de las claves (tampoco es obligatorio). No debería incluir más de una definición de clave o extensión.

```
class Profesor

(extent profesores
   Key nombre)

{
  attribute string nombre;
  attribute unsigned short id_facultad[6];
  attribute struct Direccion {string calle, string    ciudad}    direccion;
  attribute set<string> títulos;
  relationship set<Estudiante>asesora inverse Estudiante::Asesorado
  void cambia (in string nueva_direccion);
};
```

Código A.1. Ejemplo de especificación en ODL

La especificación de una relación implica la especificación de los nombres y de un *camino de recorrido*. Se emplean tipos colección (*set, bags, list*) para especificar una cardinalidad en la relación superior a uno.

A.3.2 Lenguaje de intercambio de objetos

OIF (*Object Interchange Format*) es un lenguaje de especificación empleado para volcar y cargar el estado actual de una base de datos de objetos desde un fichero o conjunto de ficheros. Este lenguaje no aparecía en ODMG-93 [CAD+94]. OIF puede emplearse entre otras cosas para intercambiar objetos entre bases de datos.

A.3.2.1 Características

Para caracterizar los estados de todos los objetos contenidos en la base de datos se emplean identificadores de objetos, ligaduras de tipos, valores de los atributos y enlaces a otros objetos. Es por eso que un fichero OIF contiene las definiciones de los objetos y éstas especifican el tipo, los valores de los atributos y las relaciones del objeto con otros objetos. Un identificador de objeto es especificado con nombres (*tag*) únicos dentro del fichero o ficheros OIF.

Con este lenguaje se pueden crear instancias de clases e inicializar los valores para sus atributos (literales, tipos estructurados, colecciones, etc). Se puede también especificar en la creación que un objeto esté físicamente próximo a otro (*clustering*).

```
Sonia Persona{Nombre "Sonia",

DireccionPersona{Calle "Uria",

                 Ciudad "Oviedo",

                 Telefono{CodigoPais 34, CodigoArea 98, Numero 5556666}}}
```

Código A.2. Ejemplo de creación de un objeto e inicialización de sus atributos con OIF

A.3.2.2 Utilidades

Cada SGDO que soporte OIF debe proporcionar dos utilidades:

- **odbdump**, que se emplea para volcar a una base de datos. Creará una representación OIF de lo especificado en la base de datos. Los nombres (*tag*) que identifican los objetos se crean automáticamente empleando algoritmos de generación de nombres dependientes de la implementación.
- **odbload**, se emplea para cargar la base de datos. Almacena en la base de datos los objetos definidos en los ficheros especificados.

A.4 Lenguaje de consulta

El lenguaje de consulta OQL (*Object Query Language*), no es computacionalmente completo, y proporciona un acceso declarativo a los objetos. Incluye primitivas de alto nivel para tratar con colecciones, listas, estructuras y arrays.

Los operadores pueden componerse libremente mientras que los operandos respeten el sistema de tipos, debido a que el resultado de cualquier consulta tiene un tipo que pertenece al modelo de tipos de ODMG y puede ser consultado de nuevo. OQL puede ser invocado desde dentro de los lenguajes de programación para los que se define una ligadura, e inversamente OQL puede invocar operaciones programadas en estos lenguajes debido a que están basados en el mismo sistema de tipos.

OQL es un lenguaje tipado. Esto quiere decir que cada expresión de consulta tiene un tipo. Este tipo puede ser derivado de la estructura de la expresión de consulta, de las declaraciones del esquema y el tipo de los objetos nombrados y literales. Así, las consultas pueden ser analizadas en tiempo de compilación y el tipo contrastado con el esquema para su corrección.

```
select distinct struct(nombre:x.nombre, sap:(select y from x.subordinados as y where
y.salario>100000))

from Empleados x
```

Código A.3. Consulta que devuelve una estructura en OQL

No proporciona operadores de actualización explícita, si no que invoca operaciones para ello, no rompiendo la semántica de un modelo de objetos, que, por definición, es gestionado por los métodos definidos en los objetos.

```
select struct (a:x.edad, s:x.sexo)

from (select y from Empleados y where y.antigüedad='10') as x

where x.nombre ="Pat"
```

Código A.4. Anidamiento de consulta en OQL

El resultado de una consulta puede ser un objeto con o sin identidad. Algunos objetos son generados por el intérprete del lenguaje de consulta y otros producidos desde la base de datos actual.

Invocación de Métodos

OQL permite llamar un método con o sin parámetros en cualquier lugar en el que el tipo resultado del método encaje con el tipo esperado en la consulta. Por supuesto, un método puede devolver un objeto complejo o una colección, y su invocación puede realizarse desde una expresión de camino compleja.

Polimorfismo

OQL es un lenguaje de consulta tipado, en el sentido de que una consulta es correcta si los tipos de los operandos encajan con los requeridos por los operadores. Cuando se filtra una colección polimórfica, una propiedad de una subclase (atributo o método) no puede ser aplicada a un elemento de la misma, excepto en el caso de una ligadura tardía a un método o bien mediante la realización de una indicación explícita de clase.

Composición de operadores

OQL es un lenguaje puramente funcional en el que todos los operadores se pueden componer libremente mientras se respete el sistema de tipos. A diferencia de SQL adopta una ortogonalidad completa sin perder la sintaxis SQL para las consultas más simples. No obstante, cuando una sintaxis SQL muy específica no entra en una categoría funcional pura, OQL acepta estas peculiaridades SQL como posibles variaciones sintácticas.

A.5 Binding para Java

Como se ha mencionado anteriormente ODMG no propone un lenguaje de manipulación de objetos (*Object Manipulation Language*, *OML*) separado, si no que simplemente sugiere que este lenguaje sea la extensión de un lenguaje de programación. De todos los *bindings* propuestos, ha sido el de Java el que ha tenido una mayor repercusión. De hecho ODMG lo ha enviado al *Java Community Process* como una base para el *Java Data Objects* [JDO00].

A.5.1 Principios de diseño

El *binding* de ODMG con Java está basado en la idea de que el programador debe percibir el *binding* como un lenguaje simple para expresar operaciones de la base de datos y de programación, no como dos lenguajes separados con límites arbitrarios entre ellos. Esto se traduce en que:

- Hay un sistema de tipos unificado compartido por el lenguaje Java y la base de datos de objetos; las instancias individuales de estos tipos comunes pueden ser persistentes o transitorias.
- El *binding* respeta la sintaxis Java, lo que indica que el lenguaje Java no tiene que ser modificado para acomodarse a este *binding*.
- Los *bindings* respetan la semántica de gestión de almacenamiento automático de Java. Los objetos serán persistentes cuando sean referenciados por otros objetos persistentes en la base de datos y se eliminarán cuando no sean alcanzados de esta manera. Es decir, el Java *binding* proporciona *persistencia por alcance* (igual que el *binding* de Smalltalk), de forma que todos los objetos alcanzables desde los objetos raíz son almacenados en la base de datos.

A.5.2 Binding del lenguaje

El Java *binding* proporciona dos formas de declarar las clases capaces de ser persistentes:

- Las clases Java existentes se pueden convertir en clases *capaces de ser persistentes*.

- Las declaraciones de las clases, al igual que las del esquema de la base de datos, pueden ser automáticamente generadas por un preprocesador de ODMG ODL.

Una posible implementación podría ser un postprocesador que tome como entrada los ficheros *.class*, produciendo un nuevo fichero de *bytecodes* que soporte la persistencia. Otra posible solución, sería un preprocesador que modifique el código java antes de compilarlo. E incluso una tercera solución sería un intérprete de Java modificado.

En esta versión del estándar queda reflejado el hecho de que el sistema tiene que poder distinguir clases que sean capaces de ser persistentes, sin embargo, no define como llegan a ser capaces de ser persistentes. También se especifica que el API ODMG deberá ser definido utilizando un paquete, y además, la implementación debe proporcionar la funcionalidad expuesta, pudiendo utilizar variantes sobre estos métodos con diferentes tipos o parámetros adicionales.

A.5.3 Traducción del modelo de objetos ODMG a Java

Se utilizará el modelo de objetos que provee Java™. Un tipo objeto de ODMG se traduce en un tipo objeto Java. Los tipos literales atómicos en ODMG se traducen en tipo primitivos Java, y los literales estructurados no existen en el Java *binding*. Una estructura se traduce en una clase Java. Las interfaces y clases abstractas no pueden ser instanciadas y por tanto, no son capaces de ser persistentes. Los objetos pueden ser nombrados empleando métodos de la clase *Database* definida en Java OML. Los objetos raíz y cualquier objeto alcanzable desde ellos son persistentes.

Java proporciona interfaces (con al menos una implementación) para las colecciones *set*, *list* y *bag*, y además proporciona una sintaxis para crear y acceder a secuencias contiguas e indexables de objetos, y una clase separada *Vector* para secuencias extensibles.

Las relaciones, extensiones y claves (*keys*) todavía no son soportadas por el Java *binding*, al igual que los accesos al meta-esquema. Cuando se detecten errores, se utilizará el mecanismo de excepciones de Java.

A.5.4 Java ODL

El Java ODL permite la descripción del esquema de la base de datos como un conjunto de clases que emplean sintaxis Java. Las instancias de estas clases pueden ser manipuladas empleando Java OML. La declaración de atributos será sintácticamente idéntica a las definiciones de campos en Java y se definirán utilizando la sintaxis y la semántica Java para las definiciones de clases.

A continuación se muestra una tabla con la traducción de los tipos de datos del modelo de objetos propuesto en sus equivalentes Java. La traducción Java para los tipos *Enum* e *Interval* no está todavía definida por el estándar. Como se puede observar los tipos primitivos pueden ser representados por sus clases equivalentes, ya que ambas formas son capaces de ser persistentes y pueden emplearse indistintamente.

Tipo ODMG	Tipo Java equivalente
Long	Int(primitive), Integer (class)
Short	short(primitive), Short (class)
Unsigned long	long (primitive), Integer (class)
Unsigned short	int (primitive), Integer (class)
Float	float (primitive), Float (class)
Double	double (primitive), Double (class)
Boolean	boolean (primitive), Boolean (class)
Octet	byte (primitive), Integer (class)
Char	char (primitive), Character (class)
String	String
Date	java.sql.Date
Time	java.sql.Time
TimeStamp	java.sql.TimeStamp
Set	interface Set
Bag	interface Bag
List	interface List
Array	array type [] or Vector
Iterator	Enumeration

Tabla A.1. Equivalencia de tipos

La declaración de operaciones en Java ODL es sintácticamente idéntica a la declaración de los métodos en Java.

A.5.5 Java OML

El principio que guía el diseño de OML es que la sintaxis empleada para crear, borrar, identificar, referenciar, acceder a campos-atributos e invocar métodos sobre objetos persistentes debería ser igual a la empleada para objetos transitorios. De esta forma una expresión puede mezclar libremente referencias a objetos persistentes y transitorios. Todas las operaciones Java OML son invocadas por llamadas a métodos sobre los objetos apropiados.

Persistencia por alcance

Como se mencionó con anterioridad, la persistencia es por alcance, pero si una clase no se especifica como capaz de ser persistente sus instancias no se harán persistentes

nunca aunque sean referenciadas por objetos persistentes. Si se desea que el valor de un atributo no sea almacenado en la base de datos se puede usar la palabra reservada *transient* en su declaración. Los campos estáticos son tratados de forma similar a los atributos *transient*.

No existe el borrado explícito

En el Java *binding* no existe el concepto de borrado explícito. Es decir, un objeto es borrado automáticamente de la base de datos si el objeto no es nombrado nunca o referenciado por ningún otro objeto persistente. Las modificaciones realizadas sobre objetos persistentes se verán reflejadas en la base de datos cuando la transacción haya finalizado.

Nombrado de objetos

Soporta el nombrado de objetos, no como un atributo del objeto, si no mediante operaciones para manipular nombres definidos en la clase *Database*. También soporta el bloqueo explícito de objetos mediante métodos del objeto *Transaction*.

Operaciones

Las operaciones son definidas en el Java OML como métodos en el lenguaje Java. Las operaciones sobre objetos persistentes y transitorios se comportan idénticamente con el contexto operacional definido por Java. Esto incluye sobrecarga, evaluación de expresiones, invocación de métodos, manejo de excepciones, etc.

A.5.5.1 Transacciones

Antes de hacer cualquier operación de base de datos un hilo debe crear explícitamente un objeto transacción o asociarlo (*join*) con un objeto transacción existente, y esa transacción debe ser abierta. Todas las operaciones que siguen en el hilo (lecturas, escrituras, y adquisiciones de bloqueo) se hacen bajo la transacción actual del hilo. Un hilo sólo puede operar sobre su transacción actual. Las transacciones deben ser creadas y abiertas explícitamente.

La creación de un nuevo objeto transacción lo asocia implícitamente con el hilo que lo llama. Cuando una transacción se ejecuta (*commit*) se realizan todas las modificaciones a los objetos persistentes dentro de la transacción, y se libera cualquier bloqueo realizado por la transacción. Si la modificación al objeto persistente se traduce en una referencia de un objeto persistente existente a un objeto transitorio, el objeto transitorio es movido a la base de datos y todas las referencias actualizadas convenientemente. El hecho de mover un objeto transitorio a la base de datos puede crear todavía mas referencias persistentes a objetos transitorios con lo que sus referencias deben ser examinadas y movidas también.

Los bloqueos de lectura se obtienen implícitamente al acceder a los objetos. Los bloqueos de escritura se obtienen implícitamente cuando son modificados.

Transacciones largas no definidas

En este estándar los objetos transitorios no están sujetos a la semántica de las transacciones. Además, los objetos transacción no pueden ser almacenados en la base de datos, lo que se traduce en que no pueden hacerse persistentes. Por tanto, la noción de transacción larga no está definida en esta especificación.

A.5.5.2 Base de Datos

El objeto *Database* al igual que el *Transaction* es transitorio. Las bases de datos no pueden ser creadas programáticamente usando el Java OML definido por este estándar. Las bases de datos deben ser abiertas antes de comenzar cualquier transacción que use la base de datos y cerradas después de que finalicen estas transacciones.

A.5.6 Java OQL

La funcionalidad completa del OQL está disponible en el Java *binding* de dos formas: por métodos *query* sobre la clase *Collection*, o por medio de consultas que emplean una clase *OQLQuery* autónoma.

Método de la clase colección

La función *query* filtra la colección empleando el predicado que se le pasa como parámetro, y devuelve el resultado. El predicado se le pasa como *string* con la sintaxis de la cláusula *where* de OQL.

Clase OQLQuery

La clase *OQLQuery* permite al programador crear una consulta, pasarle parámetros, ejecutar la consulta y tomar los resultados. Los parámetros deben ser objetos y el resultado es un objeto también. Así, cuando OQL devuelve una colección cualquiera que sea su clase (literal u objeto), el resultado es siempre una colección Java del mismo tipo. Los parámetros son ligados con el método *bind*.

A.6 Otros Bindings

El estándar define *bindings* también para otros lenguajes de programación orientados a objetos como Smalltalk o C++. Sin embargo, su repercusión ha sido mucho menor que el de Java. Para más información sobre estos *bindings* véase [CBB+99] y [Jor98].

Apéndice B ObjectDRIVER y PSE Pro

En el capítulo 17 se presentó una herramienta con la que a partir de las especificaciones realizadas con los lenguajes del *binding* Java de ODMG 2.0 se generaba código Java más código para un determinado gestor encargado de procesar las instrucciones de base de datos. La herramienta fue diseñada de forma que puedan incorporarse fácilmente nuevos gestores de bases de datos, lo que facilitará entre otras cosas la posibilidad de comparar y evaluar sus rendimientos.

Con el fin de evaluar el sistema objeto de esta tesis era necesario seleccionar otros gestores que dieran una idea de la bondad del mismo. Los gestores seleccionados han sido un gestor orientado a objetos como es PSE Pro, y un *wrapper* o envoltorio objeto-relacional como es ObjectDRIVER. Los resultados obtenidos fueron debidamente comentados en el capítulo 18 de esta memoria.

En este apéndice trata de justificarse la elección de dichos gestores, así como comentar brevemente el mecanismo construido con el fin de automatizar el proceso de traducción de clases en tablas para el mediador objeto-relacional, ObjectDRIVER, empleado.

B.1 Elección de los gestores para la evaluación

A la hora de seleccionar los gestores para realizar un estudio, la idea era contrastar el sistema diseñado con gestores basados en el **modelo de objetos** por encontrarse conceptualmente más cercanos al aquí presentado.

PSE Pro para Java es el gestor elegido como representación de los gestores basados en el modelo de objetos. Se ha seleccionado por ser un gestor construido en Java y que se ejecuta enteramente dentro de una máquina virtual (la de Java), aspecto éste en el que es coincidente con el sistema objeto de esta tesis. PSE Pro es además, conforme con el estándar ODMG (al menos con la mayoría), y permite una declaración explícita de índices.

Por otro lado, sería ideal aprovechar la oportunidad que brinda el sistema diseñado como plataforma de experimentación, para comparar el rendimiento de sistemas basados en el modelo de objetos con sistemas basados en el modelo relacional. Con este fin se opta por la incorporación del *wrapper* ObjectDRIVER, basado en el modelo relacional. La principal consideración a la hora de seleccionarlo es su conformidad con el estándar ODMG (incluso con el lenguaje de consulta).

B.1.1 ODMG y PSE Pro

PSE Pro ofrece el *binding* con Java en un paquete denominado **com.odi.odmg**, en el cual se implementan las clases definidas por el ODMG para el manejo de objetos, es decir, la clase *Database* y *Transaction*. A la clase *Database* le añade un método denominado *create* con el que se puede crear la base de datos que se desea utilizar en la aplicación, y que no está contemplado en el estándar.

Como deficiencias en relación con el seguimiento del estándar hay que hacer notar que PSE Pro no soporta el lenguaje OQL; utiliza su propio estilo para hacer consultas. No tiene en cuenta tampoco la gestión de las extensiones de las clases, las claves, ni las relaciones.

B.1.2 ODMG y ObjectDRIVER

ObjectDRIVER ofrece el *binding* con Java en un paquete denominado **fr.cermics.dbteam.objectdriver.odmg**, en el cual se implementan las clases definidas por el ODMG para el manejo de objetos, es decir, *Database* y *Transaction*. Ofrece también la clase *OQLQuery,* dentro del paquete, para manejar y poder realizar las consultas según la sintaxis OQL, aunque con algunas restricciones.

Al igual que el anterior, ObjectDRIVER, no tiene en cuenta tampoco las extensiones de las clases. Por otro lado, las relaciones, se constituirán utilizando sentencias *join* mediante la correspondiente programación reflejada en los esquemas de objetos.

B.2 Incorporación de los gestores en la herramienta

La inclusión de estos gestores en la herramienta simplemente ha supuesto la incorporación de dos clases que se encargan de implementar los métodos necesarios descritos en la interfaz *BasicMotorMethods* (véase capítulo 17).

B.2.1 Generación de código para ObjectDRIVER

La generación de código para el gestor objeto-relacional ObjectDRIVER se realiza mediante la clase **GenCodigoOD.** Para conseguir un mayor detalle sobre la misma véase [Pér99].

Métodos principales

writeExtent(java.lang.String nextent)

Devuelve el código necesario para simular la extensión de la clase mediante programación

writeFirstInMethod(java.lang.String nmet)

Devuelve las instrucciones que deben contener los métodos al inicio.

writeMyExtends()

Devuelve las clases que debe extender la clase.

writeMyImports()

Devuelve los paquetes que debe implementar la clase.

writeRelationship(ClassRelation rel)

Devuelve el código necesario para simular una relación mediante programación.

B.2.2 Generación de código para PSE Pro

La generación de código para el gestor de objetos PSE Pro se realiza mediante la clase **GenCodigoPSE.** Para conseguir un mayor detalle sobre la misma véase [Pér99].

Métodos principales

genType (ClassType tipo)

Devuelve el equivalente al tipo pasado como parámetro para el gestor PSE Pro.

writeExtent (java.lang.String nextent)

Devuelve el código necesario para simular la extensión de la clase mediante programación.

writeFirstInMethod (java.lang.String nmet)

Devuelve las instrucciones que deben contener los métodos al inicio.

writeIndex (java.lang.String nclase)

Devuelve el código necesario para simular los índices mediante programación.

writeLastInMethod (java.lang.String nmet)

Devuelve las instrucciones que deben contener los métodos al final.

writeMyImplements (boolean more)

Devuelve las sentencias donde aparecen las interfaces que se deben implementar.

writeMyImports()

Devuelve los paquetes que debe implementar la clase.

WriteRelationship (ClassRelation rel)

Devuelve el código necesario para simular una relación mediante programación.

B.3 Generador de esquemas para ObjectDRIVER

Como se ha comentado en el capítulo 5, el concepto clave de ObjectDRIVER es la noción de *mapping*. Es decir, para establecer la correspondencia entre clases y tablas relacionales es necesaria la existencia de unos ficheros que contengan tal correspondencia. Es por tanto imprescindible contar con:

- **Una descripción del esquema relacional.** En este esquema, se define la estructura de las tuplas para cada una de las tablas que intervienen en la base de datos, y que tienen especial relevancia en el mapeo o correspondencia con el esquema de objetos. Consta de un fichero donde se describe el tipo de controlador que utiliza el origen de datos (Microsoft Access, Oracle, etc.), y una descripción de las tablas a partir de los campos y su tipo, las claves primarias y los modificadores oportunos de cada campo (véase Código B.1).
- **Una descripción del esquema de definición de objetos**. En este esquema, se define la estructura y la jerarquía de todas las clases que intervienen en la aplicación, así como sus relaciones y campos que se verán reflejados en otros distintos dentro de las tablas del esquema relacional. Dentro de este esquema, se incluyen sentencias adicionales, como es el caso de *join*, que realizan el trabajo de mantenimiento de las correspondencias entre objetos, colecciones de objetos y clases descritas como tuplas.

```
input ::= <vacio> | input schema
schema ::= schema_name { schema_def };
schema_name ::= define schema word_or_id
schema_def ::= schema_elt_opt table_def
schema_elt_opt ::= <vacio>    | schema_elt_opt dmbs;
dbms ::= relationalDbms word_or_id
table_def ::= table | table_def table
table ::= table_name {attr_or_constr};
table_name ::= define table word_or_id;
attr_or_constr ::= atribute_def
| constraint_def
| attr_or_const , attribute_def
| attr_or_const , constraint_def
attr_def ::= attribute_name type_def attribute_elt_opt
attribute_name ::= word_or_id
type_def ::= type type_length | type
type ::= word_or_id
type_length ::= ( 0 | [1-9][0-9]*)
attribute_let_opt ::= <vacio> | keyPart | unique | notNull | not null
constraint_def ::= primary_key ( keys )
primary_key ::= primaryKey
keys ::= <vacio> | a_key | keys , a_key
a_key ::= word_or_id
word_or_id ::= [a-zA-Z]+ | [a-zA-Z\-_]+[a-zA-Z_0-9]*
```

Código B.1. Gramática en BNF del lenguaje para la descripción del esquema relacional

Uno de los objetivos que se plantea el equipo de desarrollo (CERMICS) de ObjectDRIVER es que la correspondencia entre los dos esquemas se haga de forma automática, y no manual como se hace actualmente (al menos, en la versión con la que se ha trabajado 1.1).

Con el fin de automatizar este proceso se ha implementado un generador de esquemas automático. Dicho generador a partir de la información obtenida del análisis de las clases especificadas en ficheros *oml*, donde se declaran los atributos y las relaciones entre las clases, es capaz de generar los dos esquemas, incluyendo el mapeo de los atributos de las clases en los campos de las tablas, y garantizando el buen funcionamiento de las aplicaciones.

Sin la existencia de un generador de este tipo, sería necesario crear los esquemas en función de la jerarquía de clases de la aplicación y analizar la estructura de la información a almacenar siguiendo las normas de diseño relacionales.

B.3.1 Diagrama de clases

Para la generación de los esquemas mencionados, son necesarios la totalidad de los ficheros *oml*, en los que se describen las clases implicadas cuyas instancias se almacenan en la base de datos y las relaciones entre ellas. Para ello, y debido a que se necesita una estructura en la cual se pueda almacenar esta información, se requiere el uso de la tabla de símbolos (*TSManager*) comentada en el capítulo 17.

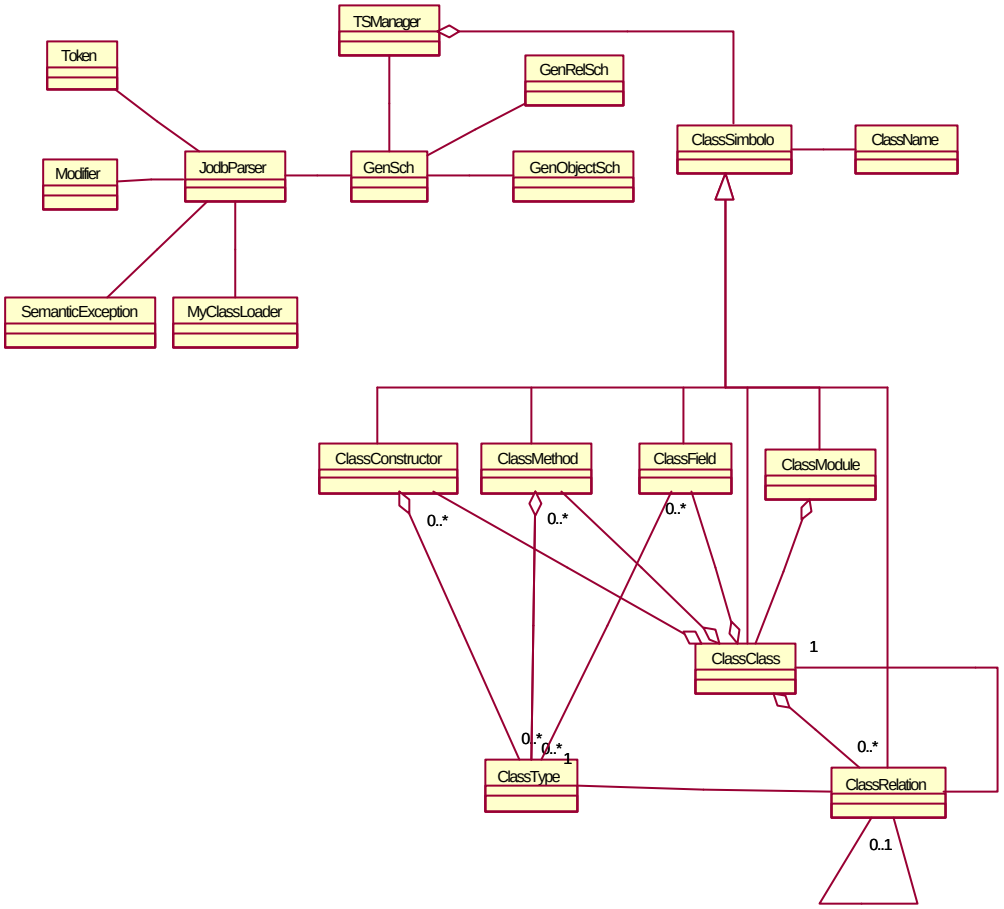


Figura B.1. Diagrama de clases del generador de esquemas para ObjectDRIVER

Clase GenSch

Se encarga de obtener la información sobre las clases persistentes, así como de otras clases que son necesarias para almacenar información en la base de datos. En definitiva, esta clase es la encargada de:

- Leer los ficheros con la descripción de las clases que van a ser persistentes.
- Generar los dos esquemas anteriormente mencionados, y que son necesarios para establecer la correspondencia entre clases y tablas.

Para ello utiliza un objeto de la clase *JodbParser*, que recogerá la información de las clases indicadas en una tabla de símbolos como la utilizada en la traducción. A partir de esta tabla, se utilizarán objetos *GenObjectSch y GenRelSch* para obtener los esquemas de correspondencias que necesita ObjectDRIVER para simular una base de datos orientada a objetos sobre una relacional.

```
define schema Publica
{
relationalDbms MsAccess;
    define table DPTO     {
    DPTONO long keyPart,
    NOMBREDPTO      string(30),
primaryKey (DPTONO)
    };

define table PUBLIC{
    PUBLICNO       long keyPart,
    NOMBRE   string(30),
    A_PUB    long,
DPTOAUTOR long,
AUTOR long,
primaryKey (PUBLICNO)
    };

    define table EMP      {
    EMPNO   long keyPart,
    NOMBRE   string(30),
    SALARIO double,
    APELLIDOS       string(30),
    DIRECCION       string(30),
    DNI      string(30),
DEPARTAMENTO long,
primaryKey (EMPNO)
    };
};
```

Código B.2. Ejemplo de la descripción de un esquema relacional

Clase GenRelSch

Esta clase será la encargada de tomar la tabla de símbolos resultado del paso anterior, y generar un archivo estructurado según la sintaxis de los esquemas relacionales. Esta sintaxis se puede resumir en:

- Crear un esquema con un nombre y para un controlador relacional (ej. MsAccess).
- Para cada clase se crea una tabla.
- Para cada atributo de la clase se crea un campo dentro de la tabla.
- Si el atributo es una clave, el campo será parte de una clave relacional.
- Si el atributo es de solo lectura, el campo será de sólo lectura.
- Si el tipo del atributo es un tipo de datos reconocido como tipo básico, se corresponderá con un tipo en el esquema relacional establecido.
- Si el tipo del atributo es una clase que se encuentra dentro de la tabla de símbolos, el tipo del campo será del tipo de la clave primaria de esa clase.
- Si el tipo del atributo no cumple las condiciones previas, no se podrá asociar a un campo en la tabla.
- Para cada relación de grado 1, se establece un campo cuyo tipo será el tipo de la clave primaria de la clase relacionada, y como identificador, el nombre de la relación.

- Cada relación de grado N no se corresponde con nada.

```
class Dpto on DPTO
type Tuple (
publicaciones Set on O<PUBLIC>
(
anObject Public on O.PUBLICNO,
join DPTO to O by(DPTO.DPTONO == O.DPTOAUTOR)
 )
readonly
inverse Public.dptoautor,
empleadosdepartamento Set on O<EMP>
(
anObject Emp on O.EMPNO,
join DPTO to O by(DPTO.DPTONO == O.DEPARTAMENTO)
 )
readonly
inverse Emp.departamento,
nombredpto String
on DPTO.NOMBREDPTO
)
end;
class Public on PUBLIC
type Tuple (
dptoautor Dpto
on PUBLIC.DPTOAUTOR
inverse Dpto.publicaciones,
autor Emp
on PUBLIC.AUTOR
inverse Emp.publicaciones,
nombre String
on PUBLIC.NOMBRE,
a_pub integer
on PUBLIC.A_PUB
)
end;
class Emp on EMP
type Tuple (
departamento Dpto
on EMP.DEPARTAMENTO
inverse Dpto.empleadosdepartamento,
publicaciones Set on O<PUBLIC>
(
anObject Public on O.PUBLICNO,
join EMP to O by (EMP.EMPNO == O.AUTOR)
 )
readonly
inverse Public.autor,
```

```
nombre String
on EMP.NOMBRE,
salario float
on EMP.SALARIO,
apellidos String
on EMP.APELLIDOS,
direccion String
on EMP.DIRECCION,
dni String
on EMP.DNI
)
end;
```

Código B.3. Ejemplo generado con la descripción del esquema de definición de objetos

Clase GenObjectSch

Esta clase, será la encargada de tomar la tabla de símbolos resultado del paso anterior, y generar un archivo estructurado según la sintaxis de los esquemas de objetos. Esta sintaxis se puede resumir en (véase Código B.3):

- Para cada clase se crea una estructura denominada tupla.
- Cada atributo de la clase, se ve reflejado dentro de la estructura anterior. Opcionalmente con una correspondencia sobre un campo de una tabla relacional (mapeo).
- Cada relación de grado 1, se ve reflejada dentro de la estructura como si fuera un atributo, y si posee relación inversa, también se refleja ésta. Opcionalmente, con una correspondencia sobre un campo de una tabla relacional (mapeo).
- Cada relación de grado N, se ve reflejada como un atributo y si posee relación inversa, también se refleja ésta. Opcionalmente, se añade una correspondencia como una operación *join* sobre los elementos de la clase relacionada (mapeo).
- Los tipos de los atributos y relaciones, siguen una correspondencia directa sobre los tipos del esquema.

Las opciones de mapeo se establecen para que la herramienta, automáticamente genere estas correspondencias entre el esquema de objetos y el relacional.

B.3.2 Utilización

Para la creación de los dos ficheros de esquemas únicamente son necesarios los ficheros que contienen su descripción Java con extensiones ODMG, es decir, archivos con la extensión *oml*. Una vez que éstos estén disponibles la generación se puede realizar de dos formas: en línea de comandos o desde la interfaz gráfica.

Línea de comandos

Para la ejecución en línea de comandos hay que tener en cuenta que la clase GenSch, toma como parámetros de entrada:

- **filename:** nombre base del fichero que contendrá los esquemas. La clase añadirá la extensión Obj.txt al fichero con el esquema de objetos, y Rel.txt, al fichero con el esquema relacional, es decir, las tablas que intervienen.
- **mapeo**: (ON | OFF), con esta opción, indicaremos a la clase que genere en la traducción, la parte correspondiente al mapeo de los campos y las relaciones del esquema de objetos, en sus correspondientes campos en las tablas relacionales.
- **classoml1 (classoml2 ...)*** nombre de las clases persistentes que se reflejarán en los esquemas.

Ejemplo

```
C:> Java GenSch Publica ON Emp Dpto Public
```

Con la instrucción anterior se obtienen los esquemas para la aplicación que utiliza las clases Public (Publicación), Dpto (Departamento), y Emp (Empleado).

Interfaz gráfico

Otra forma de obtener los esquemas mencionados, es por medio de la aplicación que se encarga de traducir los ficheros oml. Esta aplicación OML2Java, incluye en su menú herramientas, una extensión para ObjectDriver como se refleja en la siguiente figura.

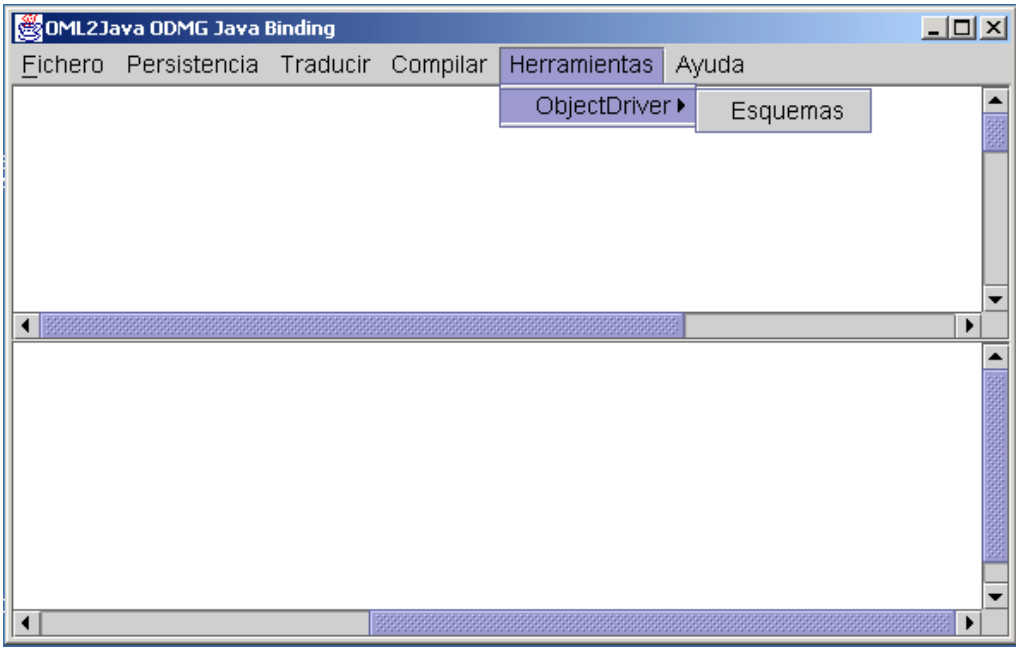


Figura B.2. Opción inicial de la aplicación para generar esquemas

Seleccionando ésta opción se pasa a configurar el proceso de generación de esquemas. Dicho proceso se divide en tres partes:

- La primera, consiste en introducir el nombre de la base de datos relacional que se utilizará.
- El segundo paso, es indicar si se desea que en el esquema de objetos, se incluyan los mapeos de los distintos atributos de las clases a sus correspondientes campos en las tablas relacionales. Si la base de datos ya existe, y los nombres de los atributos no coinciden con los nombres de los campos en las tablas, se podrá rellenar esta información independientemente. Por el contrario, si se selecciona

esta opción, automáticamente se generarán los mapeos en campos con el mismo nombre en la tabla.

- El tercer paso, consiste en seleccionar todas las clases oml que van a ser almacenadas en la base de datos, así como otras clases que sirvan para almacenar información sobre las principales o persistentes. Por ejemplo, si un atributo de una clase persistente tiene un tipo de datos que se corresponde con una clase oml o Java, y esta clase no se va a gestionar independientemente ni posee relaciones con otras clases se debería añadir también.

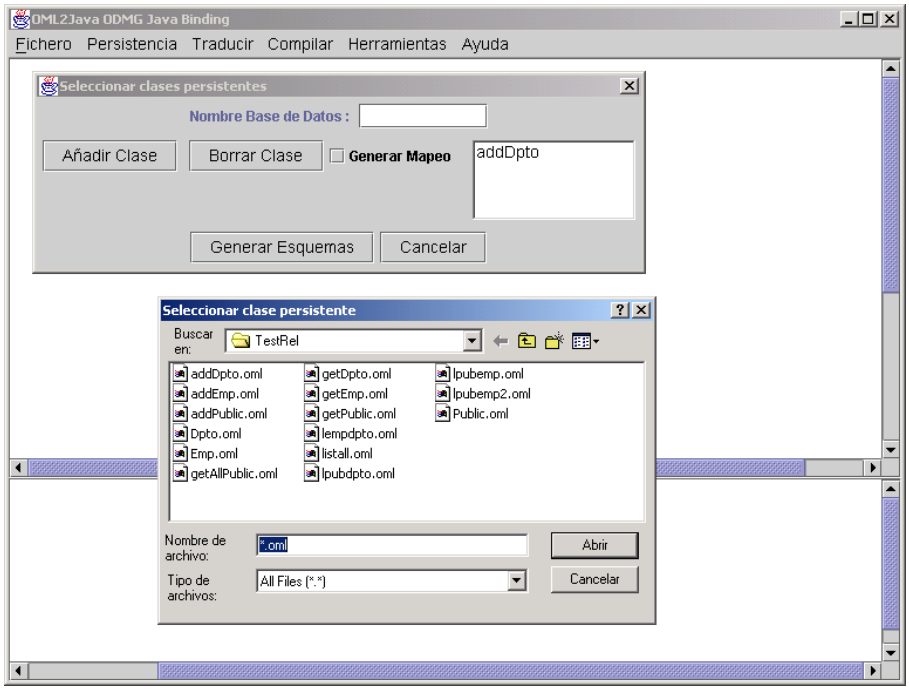


Figura B.3. Carga de clases y generación definitiva de los esquemas

Posteriormente, simplemente pulsaremos en el botón *Generar Esquemas* y se obtendrán dos ficheros, uno de ellos denominado igual que el nombre de la base de datos más la terminación *Obj.txt* con el esquema de objetos, y otro de ellos con la terminación *Rel.txt*, con el esquema relacional.

Apéndice C Gramáticas ODL - OML y OQL para Java

El SGBDOO BDOviedo3 adopta como lenguajes para la definición, manipulación y consulta los propuestos por el estándar ODMG en su *binding* para Java, y la justificación de dicha adopción puede encontrarse en el capítulo 13 de esta memoria. Por otro lado, en el capítulo 17 se describe una herramienta compatible con el *binding* Java de ODMG, y que permite generar código para diferentes gestores persistentes compatibles con dicho *binding*.

El objeto de este apéndice es mostrar las gramáticas definitivas de los lenguajes de definición (ODL), de manipulación (OML) y de consulta (OQL) propuestos por el estándar ODMG en su *binding* para Java, que han sido considerados en este trabajo.

C.1 Gramática ODL + OML

En este apartado se presenta la gramática definitiva para el lenguaje de definición y de manipulación de objetos ligado al lenguaje Java, y que incluye ya por ejemplo la posibilidad de declarar explícitamente índices, no contemplada inicialmente en el estándar. Esta será la gramática reconocida por el traductor mencionado en el capítulo 17. Para más información véase [Pér99].

```
compilation_unit  =
 [ package_statement ]
 < import_statement >
 < type_declaration >  .

package_statement  =
 "package" package_name  ";"  .

import_statement  =
 "import"  (  ( package_name  "."  "*"  ";"  )
 /  ( class_name  / interface_name )  )  ";"  .

type_declaration  =
 [ doc_comment ]  ( class_declaration  / interface_declaration / module_dcl /
struct_dcl / exception_dcl)  ";"  .

doc_comment  =  "/**"  "... text ..."  "*/"  .

module_dcl = "module" identifier "{" < type_declaration > "}" ";"

exception_dcl = "exception" identifier "{" < member > "}" ";"

struct_dcl = "struct" identifier "{" < member > "}" ";"

member = type variable_declarator < "," type variable_declarator >

class_declaration  =
 < modifier >  "class" identifier
 [  "extends" class_name ] [ "implements" interface_name  <  " , "
interface_name  >  ]
```

```
 [ type_declaration ]
 "{"  < field_declaration >  "}"  .
 [ “implements” interface_name  <  “ , ”  interface_name  >  ]
 [ type_declaration ]
"{"  < field_declaration >  "}"  .
interface_declaration  =
 < modifier >  "interface" identifier
 [  "extends" interface_name  <  "," interface_name >  ]
 [ type_declaration ]
 "{"  < field_declaration >  "}"  . type_declaration = “(“ [extent_dcl ]
[key_dcl] “)”

extent_dcl = “extent” identifier [index_dcl]
index_dcl = “index” index_list < “,” index_list >

index_list = identifier / “(“ identifier “,” identifier < ”,” identifier> “)
key_dcl = “key”/”keys” key_list
key_list = identifier / “(“ identifier “,” identifier < “,” identifier > “)”
field_declaration  =
 (  [ doc_comment ]  ( method_declaration
 / constructor_declaration
 / variable_declaration )  )
 / static_initializer
 / rel_dcl
 /  ";"  .

rel_dcl = “relationship” [ collection_type ] “<” type “>”  identifier
 [inverse_rel] [order_dcl]

collection_type = ( “set” / “list” / “bag” / “array” )

inverse_rel = “inverse” identifier “::” identifier

order_dcl = “order_by” “(“ identifier < “,” identifier > “)” “;”

method_declaration  =
 < modifier > type identifier
 "("  [ parameter_list ]  ")"  <  "["  "]"  >
 [ “throws” identifier_list ]
 [“context” “(“ identifier_list “)” ]
 ( statement_block  /  ";"  )  .

constructor_declaration  =
 < modifier > identifier  "("  [ parameter_list ]  ")"
statement_block  .

statement_block  =  "{"  < statement >  "}"  .

variable_declaration  =
 < modifier > type variable_declarator
 <  "," variable_declarator  >  ";"  .

variable_declarator  =
 identifier  <  "["  "]"  >  [  "=" variable_initializer ]  .

variable_initializer  =
expression
 /  (  "{"  [ variable_initializer
 <  "," variable_initializer  >  [  ","  ]  ]  "}"  )  .

static_initializer  =
 "static" statement_block  .

parameter_list  =
parameter  <  "," parameter >  .

parameter  =
type identifier  <  "["  "]"  >  .
statement  =
```

```
variable_declaration
 /  ( expression  ";"  )
 /  ( statment_block  )
 /  ( if_statment   )
 /  ( while_statement )
 /  ( for_statement )
 /  ( try_statement )
 /  ( switch_statement )
 /  (  "synchronized"  "(" expression  ")" statement  )
 /  (  "return"  [ expression ]  ";"  )
 /  (  "throw" expression ";"  )
 /  ( identifier  ":" statement )
 /  (  "break"  [ identifier ]  ";"  )
 /  (  "continue"  [ identifier ]  ";"  )
 /  (  ";"  )  .

if_statement =
 "if"  "(" expression  ")" statement
 [  "else" statement ]  .

do_statement =
 "do" statement  "while"  "(" expression  ")"  ";"  .

while_statement =
 "while"  "(" expression  ")" statement  .

for_statement =
 "for"  "("  ( variable_declaration  /  ( expression  ";"  )  /  ";"  )
 [ expression ]  ";"
 [ expression ]  ";"
 ")" statement  .

try_statement =
 "try" statement
 <  "catch"  "(" parameter  ")" statement >
 [  "finally" statement ]  .

switch_statement =
 "switch"  "(" expression  ")"  "{"
 <  (  "case" expression  ":"  )
 /  (  "default"  ":"  )
 / statement >
 "}"  .

expression  =
numeric_expression
 / testing_expression
 / logical_expression
 / string_expression
 / bit_expression
 / casting_expression
 / creating_expression
 / literal_expression
 /  "null"
 /  "super"
 /  "this"
 / identifier
 /  (  "(" expression  ")"  )
 /  ( expression
 (  (  "("  [ arglist ]  ")"  )
 /  (  "[" expression  "]"  )
 /  (  "." expression  )
 /  (  "," expression  )
 /  (  "instanceof"  ( class_name / interface_name )  )
 )  )  .

numeric_expression  =
 (  (  "-"
 /  "++"
 /  "--"  )
```

```
expression )
/  ( expression
 (  "++"
 /  "--"  )  )
 /  ( expression
 (  "+"
 /  "+="
 /  "-"
 /  "-="
 /  "*"
 /  "*="
 /  "/"
 /  "/="
 /  "%"
 /  "%="  )
expression  )  .

testing_expression  =
 ( expression
 (  ">"
 /  "<"
 /  ">="
 /  "<="
 /  "=="
 /  "!="  )
expression  )  .

logical_expression  =
 (  "!" expression )
 /  ( expression
 (  "ampersand"
 /  "ampersand="
 /  "|"
 /  "|="
 /  "^"
 /  "^="
 /  (  "ampersand"  "ampersand"  )
 /  "||="
 /  "%"
 /  "%="  )
expression  )
 /  ( expression  "?" expression  ":" expression )
 /  "true"
 /  "false"  .

string_expression  =  ( expression
 (  "+"
 /  "+="  )
expression  )  .

bit_expression  =
 (  "~" expression )
 /  ( expression
 (  ">>="
 /  "<<"
 /  ">>"
 /  ">>>"  )
expression  )  .

casting_expression  =
 "(" type  ")" expression  .

creating_expression  =
 "new"  (  ( classe_name  "("  [ arglist ]  ")"  )
 /  ( type_specifier  [  "[" expression  "]"  ]  <  "["  "]"  >  )
 /  (  "(" expression  ")"  )  )  .

literal_expression  =
```

```
integer_literal
 / float_literal
 / string
 / character  .
arglist  =
expression  <  "," expression  >  .
type  =
type_specifier  <  "["  "]"  >  .

type_specifier =
 "boolean"
 /  "byte"
 /  "char"
 /  "short"
 /  "int"
 /  "float"
 /  "long"
 /  "double"
 /  “u_short”
 /  “u_long”
 /  “octet”
 /  “any”
 / class_name
 / interface_name  .

modifier =
 "public"
 /  "private"
 /  "protected"
 /  "static"
 /  "final"
 /  "native"
 /  "synchronized"
 /  "abstract"
 /  "threadsafe"
 /  “in”
 /  “out”
 /  “inout”
 /  “readonly”
 /  “attribute”
 /  “oneway”
 /  "transient"  .

package_name =
identifier
 /  ( package_name  "." identifier )  .

class_name  =
identifier
 /  ( package_name  "." identifier )  .

interface_name =
identifier
 /  ( package_name  "." identifier )  .

integer_literal =
 (  (  "1..9"  <  "0..9"  >  )
 /  <  "0..7"  >
 /  (  "0"  "x"  "0..9a..f"  <  "0..9a..f"  >  )  )
 [  "l"  ]  .

float_literal  =
 ( decimal_digits  "."  [ decimal_digits ]  [ exponent_part ]  [
float_type_suffix ]  )
 /  (  "." decimal_digits  [ exponent_part ]  [ float_type_suffix ]  )
 /  ( decimal_digits  [ exponent_part ]  [ float_type_suffix ]  )  .
```

```
decimal_digits  =
 "0..9"  <  "0..9"  >  .
exponent_part  =
 "e"  [  "+"  /  "-"  ] decimal_digits  .
float_type_suffix  =
"f"  /  "d"  .
character  =
"based on the unicode character set"  .
string  =   "'"  < character  >  "'"  .
identifier  =
  "a..z,$,_"  <  "a..z,$,_,0..9,unicode character over 00C0"  >  .
```

C.2 Gramática OQL

En este apartado se presenta la gramática definitiva empleada para el lenguaje de consulta OQL ligado al lenguaje Java. La gramática OQL aquí presentada se obtiene analizando la gramática propuesta por el estándar ODMG, que no cumple ningún formato en su definición, si bien, tiene ciertos parecidos con una gramática estilo BNF, y construyendo una gramática LL1 que se expresa en formato BNF.

Utilizando esta gramática, se determinará si el predicado de una determinada consulta así como todos los métodos que utilicen como parámetro una cadena que deba corresponderse con la sintaxis OQL han sido construidos correctamente.

```
<QueryProgram> ::= <OptDefineQuery> <Query>

<OptDefineQuery> ::= <DefineQueryList>
<OptDefineQuery> ::= <Vacio>

<DefineQueryList> ::= <DefineQuery> ; <OptDefineQuery>

<DefineQuery> ::= "define" <Identifier> "as" <Query>


#Query

<Query> ::= <SQuery> <MasQuery>
<Query> ::= "select" <QueryOptDistinct> <ProjectionAttributes> "from"
<VarDeclaList>
<QueryOptWhere>
 <QueryOptGroup>
<QueryOptHaving>
 <QueryOptOrder>

<SQuery> ::= <Identifier>
<SQuery> ::= <Identifier> ( <OptQueryList> )        #identifier = function_name
<SQuery> ::= ( <Identifier> ) <Query>               #identifier = class_name
<SQuery> ::= <Identifier> ( <NamedQueryList> )      #identifier = type_name
<SQuery> ::= <Identifier> ( <OptQuery> )    #identifier = type_name

<SQuery> ::= "list" ( <Query> <ModListQuery>

<ModListQuery> ::= )
<ModListQuery> ::= , <QueryList> )
<ModListQuery> ::= .. <Query> )

#<Squery> ::= ( <Query> .. <Query> )

<SQuery> ::= "nil"
<SQuery> ::= "true"
<SQuery> ::= "false"
<SQuery> ::= <IntegerLiteral>
<SQuery> ::= <FloatLiteral>
<SQuery> ::= <StringLiteral>
<SQuery> ::= ( <Query> )
<SQuery> ::= - <Query>
<SQuery> ::= "abs" ( <Query> )
<SQuery> ::= "not" <Query>
<SQuery> ::= "struct" ( <NamedQueryList> )
```

```
<SQuery> ::= "set" ( <OptQueryList> )
<SQuery> ::= "bag" ( <OptQueryList> )
<SQuery> ::= "array" ( <OptQueryList> )
<SQuery> ::= * <Query>
<SQuery> ::= "first" ( <Query> )
<SQuery> ::= "last" ( <Query> )
<SQuery> ::= "for" "all" <Identifier> "in" <Query> : <Query>
<SQuery> ::= "exists" <Identifier> "in" <Query> : <Query>
<SQuery> ::= "exists" ( <Query> )
<SQuery> ::= "unique" ( <Query> )
<SQuery> ::= "count" ( <Query> )
<SQuery> ::= "count" ( * )
<SQuery> ::= "sum" ( <Query> )
<SQuery> ::= "min" ( <Query> )
<SQuery> ::= "max" ( <Query> )
<SQuery> ::= "avg" ( <Query> )
<SQuery> ::= "listtoset" ( <Query> )
<SQuery> ::= "element" ( <Query> )
#<SQuery> ::= "distinct" ( <Query> )
<SQuery> ::= "flatten" ( <Query> )

<MasQuery> ::= + <Query>
<MasQuery> ::= - <Query>
<MasQuery> ::= * <Query>
<MasQuery> ::= / <Query>
<MasQuery> ::= "mod" <Query>
<MasQuery> ::= || <Query>
<MasQuery> ::= "like" <StringLiteral>
<MasQuery> ::= "and" <Query>
<MasQuery> ::= "or" <Query>
<MasQuery> ::= <ComparisonOperator> <Quantifier> <Query>
<ComparisonOperator> ::= =
<ComparisonOperator> ::= ¡=
<ComparisonOperator> ::= >
<ComparisonOperator> ::= <
<ComparisonOperator> ::= >=
<ComparisonOperator> ::= <=
<MasQuery> ::= . <Identifier> <OptModDot>
<MasQuery> ::= -> <Identifier> <OptModDot>
<MasQuery> ::= "in" <Query>
<MasQuery> ::= "intersect" <Query>
<MasQuery> ::= "union" <Query>
<MasQuery> ::= "except" <Query>
<MasQuery> ::= [ <Query> <MasMasQuery>
<MasQuery> ::= <Vacio>

<MasMasQuery> ::= : <Query> ]
<MasMasQuery> ::= ]

<OptModDot> ::= <Vacio>
<OptModDot> ::= ( <QueryList> )

<NamedQueryList> ::= <Identifier> : <Query> <MasNamedQueryList>

<MasNamedQueryList> ::= , <NamedQueryList>
<MasNamedQueryList> ::= <Vacio>

<QueryList> ::= <Query> <MasQueryList>

<MasQueryList> ::= , <QueryList>
<MasQueryList> ::= <Vacio>

<OptQueryList> ::= <Vacio>
<OptQueryList> ::= <QueryList>

<OptQuery> ::= <Vacio>
<OptQuery> ::= <Query>
<QueryOptDistintct> ::= <Vacio>
<QueryOptDistintct> ::= "distinct"
```

```
<ProjectionAttributes> ::= <ProjectionList>

<ProjectionAttributes> ::= *
<ProjectionList> ::= <Projection> <MasProjectionList>

<MasProjectionList> ::= , <ProjectionList>
<MasProjectionList> ::= <Vacio>

<Projection> ::= <Identifier> : <Query>
<Projection> ::= <Query> <MasProjection>

<MasProjection> ::= AS <Identifier>
<MasProjection> ::= <Vacio>

<VarDeclaList> ::= <VarDeclaration> <MasVarDeclaList>

<MasVarDeclaList> ::= , <VarDeclaList>
<MasVarDeclaList> ::= <Vacio>

<VarDeclaration> ::= <Query> <MasVarDeclaration>

<MasVarDeclaration> ::= <Identifier>
<MasVarDeclaration> ::= "as" <Identifier>
<MasVarDeclaration> ::= <Vacio>

<QueryOptWhere> ::= <Vacio>
<QueryOptWhere> ::= "where" <Query>

<QueryOptGroup> ::= <Vacio>
<QueryOptGroup> ::= "group" "by" <PartitionAttributes>

<PartitionAttributes> ::= <ProjectionList>

<QueryOptHaving> ::= <Vacio>
<QueryOptHaving> ::= "having" <Query>

<QueryOptOrder> ::= <Vacio>
<QueryOptOrder> ::= "order" "by" <SortCriterionList>

<SortCriterionList> ::= <SortCriterion> <MasListSortCriterion>

<MasListSortCriterion> ::= , <SortCriterionList>
<MasListSortCriterion> ::= <Vacio>

<SortCriterion> ::= <Query> <MasSortCriterion>

<MasSortCriterion> ::= "asc"
<MasSortCriterion> ::= "desc"
<MasSortCriterion> ::= <Vacio>

<Quantifier> ::= "some"
<Quantifier> ::= "any"
<Quantifier> ::= "all"

<Vacio> ::= VACIO
```

Glosario de traducciones

Expresión en inglés	Traducciones preferidas	Otras Traducciones
API (Applications Program Interface)	API (Interfaz de los programas de aplicación)	
Attachments	Ataduras	
Benchmarking	Experimentación	Prueba
Binding	Ligadura	
Browser	Navegador	
Clustering	Agrupamiento	
Compliant	Compatible	Conforme
Customize	Personalizar	
DBI(Database Implementator)	IBD (Implementador de base de datos)	
Dynamic Function Linker	Enlazador dinámico de funciones	
Extent	Extensión de la clase	Extensional
Framework	Marco de aplicación	
Garbage Collection	Recolección de basura	
Impedance mismatch	Desadaptación de impedancias	
Intension	Intensional	
Interface	Interfaz	
Key	Clave	
Late-binding	Ligadura tardía	
Lifetime	Vida	Duración
Map, Mapping	Hacer corresponder,”mapear”	Correspondencia
Method Materialization	Materialización de métodos	
Middleware	Software intermedio	
Object Request Broker	Gestor de peticiones de objetos	Intermediario entre objetos
ODL (Object Definition Language)	Lenguaje de definición de objetos	

ODMS (Object Data Management System)	SGDO (Sistemas de gestión de objetos)	
OID (Object Identifier)	IDO (Identificador de objetos)	
OIF (Object Interchange Format)	Formato de intercambio de objetos	
OML (Object Manipulation Language)	Lenguaje de manipulación de objetos	
OODBMS (Object-Oriented Database Management System)	SGBDOO (Sistemas de gestión de bases de datos orientadas a objetos)	
OQL(Object Query Language)	Lenguaje de consulta de objetos	
Overloading	Sobrecarga (de métodos)	
Overriding	Redefinición (de métodos)	Anulación
Path expressions	Expresiones de camino	
Persistence capable classes	Clases susceptibles de ser persistentes	
Persistent	Persistente	
Pointer swizzling	Transformación de punteros	
RDBMS (Relational Database Management System)	SGBDR (Sistema de gestión de bases de datos relacionales)	
Reflection	Reflectividad	
Roots	Raíces	
Safeness	Fiabilidad	
Stores	Almacenes	
Table driven	Dirigido por tablas	
Transient	Temporal	Transitorio
Traversal paths	Caminos de recorrido	
Triggers	Disparadores	
Wrapper	Envoltorio	Mediador

Referencias

[ABB+86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian y M. Young. *Mach: A New Kernel Foundation for Unix Development.* Summer USENIX Conference. Atlanta, 1986

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisolm, W.P. Cockshott y R. Morrison. *An approach to persistent programming.* Computer Journal, 26(4): 360-365, 1983.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier y S. Zdonik. *The Object-Oriented Database System Manifesto.* En Proc. of 1st International Conference on Deductive and Object-Oriented Databases (DOOD). Japón, 1989.

[ACC81] M.P. Atkinson, K.J. Chisholm y W.P. Cockshott. *PS-Algol: An Algol with a Persistent Heap.* ACM SIGPLAN Notices, 17(7). 1981. Pág. 24-31.

[ADA+96] Darío Álvarez Gutiérrez, Mª Angeles Díaz Fondón, Fernando Álvarez García, Lourdes Tajes Martínez, Ana Belén Martínez Prieto y Raúl Izquierdo Castanedo. *Persistencia en Sistemas Operativos Orientados a Objetos. Ventajas para los Sistemas de Gestión de Bases de Datos.* Actas de las I Jornadas de Investigación y Docencia en Bases de Datos. La Coruña, Junio 1996.

[ADJ+96] M.P. Atkinson, L. Daynés, M.J. Jordan, T. Printezis y S. Spence. *An Orthogonally Persistent Java.* ACM Record, Dic 1996.

[AGS90] R. Agrawal, N.H. Gehani y J. Srinivasan. *OdeView: The Graphical Interface to Ode.* Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp 34-43, 1990.

[AK95] Shailesh Agarwal y Arthur M. Keller. *Architecting Object Applications for High Performance with Relational Databases.* White Paper. 1995.

[Ala97] S. Alagic. *The ODMG Object Model. Does it Make Sense?*. OOPSLA, 1997.

[Álv00] Fernando Álvarez García. *AGRA: Sistema de Distribución de Objetos para un Sistema Distribuido Orientado a Objetos soportado por una Máquina Abstracta.* Tesis Doctoral. Departamento de Informática, Universidad de Oviedo, Septiembre 2000.

[Álv98] Darío Álvarez Gutiérrez. *Persistencia completa para un sistema operativo orientado a objetos usando una máquina abstracta con arquitectura reflectiva.* Tesis Doctoral. Departamento de Informática, Universidad de Oviedo, Marzo 1998

[Aok91] P.M. Aoki. *Implementation of Extended Indexes in POSTGRES.* Special Interest Group on Information Retrieval Forum 25, 1, 1991.

[ATA+96] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, Mª Angeles Díaz Fondón, Juan Manuel Cueva Lovelle y Raúl Izquierdo Castanedo. *Un sistema operativo para el sistema orientado a objetos integral Oviedo3.* Actas de las II Jornadas de Informática. Almuñécar, Granada. Julio de 1996

[ATA+97] Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Fernando Álvarez García, Mª Angeles Díaz Fondón, Raúl Izquierdo Castanedo y Juan Manuel Cueva Lovelle. *An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System*. Eleventh European Conference in Object Oriented Programming (ECOOP'97), Workshop in Object Orientation in Operating Sytems. Jyväskylä, Finlandia. Junio de 1997.

[ATD+00] Fernando Álvarez García, Lourdes Tajes Martínez, Mª Angeles Díaz Fondón y Darío Álvarez Gutiérrez. *Introducing Distribution in an OS Environment of Reflective OO Abstract Machines.* 14th European Conference on Object-Oriented Programming (Workshop), 2000.

[ATD+98a] Fernando Álvarez García, Lourdes Tajes Martínez, Mª Angeles Díaz Fondón, Darío Álvarez Gutiérrez y Juan Manuel Cueva Lovelle. *Agra: The Object Distribution Subsystem of the SO4 Object-Oriented Operating System.* En Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98). H.R.Arabnia, ed. pp. 255-258. CSREA Press. 1998.

[ATD+98b] Fernando Álvarez García, Lourdes Tajes Martínez, Mª Ángeles Díaz Fondón, Darío Álvarez Gutiérrez y Juan Manuel Cueva Lovelle. *Object Distribution Subsystem for an Integral Object-Oriented System.* En Anales del Simposio Argentino en Orientación a Objetos(ASOO'98), Argentina, Octubre de 1998

[Bar98] D. Barry. *ODMG 2.0: A Standard for Object Storage*. Component Stategies. Julio 1998.

[BC99] E. Bertino y B. Chin. *The Indispensability of Dispensable Indexes.* IEEE Transactions on Knowledge and Data Engineering, vol. 11, nº 1, 1999.

[Ber94] E. Bertino. *Index Configuration in Object Oriented Databases.* VLDB Journal,3, 1994.

[BF95] E. Bertino y P. Foscoli. *Index Organizations for Object-Oriented Database Systems.* IEEE Transactions on Knowledge and Data Engineering. Vol.7, 1995.

[BFG+98] E. Bertino, E. Ferrari, G. Guerrini y I. Merlo. *Extending the ODMG Object Model with Time.* ECOOP, 1998.

[BG98] E. Bertino y G. Guerrini. *Extending the ODMG Object Model with Composite Objects.* OOPSLA, 1998.

[BK89] E. Bertino y W. Kim. *Indexing Techniques for Queries on Nested Objects.* IEEE Transactions on Knowledge and Data Engineering. Vol 1 nº2, 1989.

[BKK+88] J. Banerjee, W. Kim, S.J. Kim y J.F. Garza. *Clustering a DAG for CAD databases.* IEEE Transactions on Software Engineering,1988.

[BM93] E. Bertino y L. Martino. *Object-Oriented Database Systems: Concepts and Architectures.* Addison-Wesley, 1993. Traducción castellano: Sistemas de Bases de Datos Orientadas a Objetos. Conceptos y Arquitecturas. Addison-Wesley/Diaz de Santos, 1995.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*, 2nd Edition. Benjamin Cummings, 1994. Versión en español: Análisis y diseño orientado a objetos con aplicaciones, 2ª edición. Addison-Wesley/Diaz de Santos, 1996.

[CAD+94] R. Cattell, T. Atwood, J. Duhl, G. Ferran, M. Loomis, D. Wade, D. Barry, J. Eastman y D. Jordan. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994.

[CAI98] CAI. Computer Associates International. *Jasmine*. Diponible en URL http://www.cai.com/products/jasmine, 1998.

[Cat94] R. Cattell. *Object Data Management. Object Oriented and Extended Relational Database Systems (Revised Edition)*. Addison Wesley, 1994.

[CBB+97] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland y D. Wade. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[CBB+99] R. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda y F. Velez. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann Publishers, 1999.

[CCL92] C. Chin, B. Chin, H. Lu. *H-trees: A Dinamic Associative Search Index for OODB.* ACM SIGMOD, 1992.

[CDF+86] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson y E. J. Shekita. *The Architecture of the EXODUS Extensible DBMS.* Proceedings of International Workshop on Object-oriented database system, 1986.

[CDK94] G.Coulouris, J. Dollimore y T. Kindberg. *Distributed Systems. Concepts and Design (2ª edición).* Addison-Wesley, 1994.

[CFL+91] M.J Carey, M.J. Franklin, M. Livny y E.J. Shekita. *Data Caching Tradeoffs in Client-Server DBMS Architectures*. Proceedings of the ACM SIGMOD International Conference on Management of Data and Symposium on Principles of Database Systems, Denver 1991.

[CIA+96] Juan Manuel Cueva Lovelle, Raúl Izquierdo Castanedo, Darío Álvarez Gutiérrez, Lourdes Tajes Martínez, Mª Ángeles Díaz Fondón, Fernando Álvarez García y Ana Belén Martínez Prieto. *Oviedo3: Acercando las tecnologías orientadas a objetos al hardware.* I Jornadas de trabajo en Ingeniería de Software. Sevilla. Noviembre de 1996.

[Cla91] Stewart M. Clamen. *Data Persistence in Programming Languages. A Survey*. CMU-CS-91-155. Carnegie Mellon University, 1991.

[CH91] J.R. Cheng y A.R. Hurson. *Effective Clustering of Complex Objects in Object Oriented Databases.* Proceedings of ACM SIGMOD International Conference on Management of Data, Denver, 1991.

[CHL+96] B. Chin, J. Han, H. Lu y K. Lee. *Index Nesting - an Efficient Approach to Indexing in Object-Oriented Databases.* VLDB Journal, 1996.

[DAG99] Mª Angeles Díaz Fondón, Darío Álvarez Gutiérrez, Armando García-Mendoza Sánchez, Fernando Álvarez García, Lourdes Tajes Martínez, Juan Manuel Cueva Lovelle. *Integrating Capabílities into the Object Model to Protect Distributed Object Systems.* International Symposium on Distributed Objects and Applications. Edimburgh, Scotland 1999.

[DAO+95] A. Dogac, M. Altinel, C. Ozkan y I. Durusoy. *Implementation Aspects of an Object-Oriented DBMS.* SIGMOD Record, Vol 24 nº 1, Marzo 1995.

[Dev97] Ricardo Devis Botella. *C++. STL-Plantillas-Excepciones-Roles y Objetos.* Paraninfo, 1997.

[DFM+90] D.J. DeWitt, P. Futtersack, D.Maier y F. Velez. *A Study of Three Alternative Worksation-Server Architectures for Object-Oriented Database Systems.* Proceedings of the 16th VLDB Conference. Australia, 1990.

[Día00] Mª Ángeles Díaz Fondón. *Núcleo de Seguridad para un Sistema Operativo Orientado a Objetos Soportado por una Máquina Abstracta.* Tesis Doctoral. Departamento de Informática,Universidad de Oviedo, 2000.

[DRH+92] A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan y K. Maciunas. *An Examination of Operating System Support for Persistent Object Systems.* En Proceedings of the 25th Hawaii International Conference on System Sciences, Hawaii, EE.UU. Pág. 779-789, 1992.

[EM99] A. Eisenberg y J. Melton. *SQL: 1999, formerly known as SQL 3.* SIGMOD Record 28 (1), Marzo 1999.

[Exc97] Excelon corp. *Comparing ODBMS and RDBMS Implementations of Qantum Objects.* http://www.exceloncorp.com/products/objectstore/os_white_papers.html, 1997.

[Feg99] Leónidas Fegaras. *VOODOO: A Visual Object-Oriented Database Language for ODMG OQL.* 13th European Conference for Object-Oriented Programming, (ECOOP) Lisboa 1999.

[FP99] P.R. Falcone y N. W. Paton. *A Deductive Extension for ODMG Compliant Object Databases.* Journal: L'Objet-Logiciel, Bases de Donnees, Reseaux, Vol 5, Num 1. Hermes Science Publications, 1999.

[Gar99] Modesto García García. *Diseño y Construcción de un Motor de Base de Datos para el Sistema de Gestión de Base de Datos Orientado a Objetos BDOviedo3.* Proyecto Fin de Carrera 982059. Escuela Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón. Universidad de Oviedo, 1999.

[Gar99b] Armando García-Mendoza Sánchez. *Implantación de un mecanismo uniforme de protección en el Sistema Integral Orientado a Objetos Oviedo3.* Proyecto Fin de Carrera 982021. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Noviembre de 1999.

[GBC+97] G. Guerrini, E. Bertino, B. Catania y J. García Molina. *A Formal Model of Views for Object-Oriented Database Systems*. Theory and Practice of Object Systems, Vol.3, Num 4, 1997.

[Gem00] *GemStone*. Disponible en URL http://www.gemstone.com/, marzo 2000.

[GR83] A. Goldberg y D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley. 1983.

[GUW00] H. García-Molina, J.D. Ullman y J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[HFL+89] L.M.Haas, J.C. Freytag, G.M. Lohman y H. Pirahesh. *Extensible Query Processing in Starburst*. Proceedings of the ACM SIGMOD international conference on Management of data, pp. 377-388, 1989.

[HHR91] E.N. Hanson, T. M. Harvey y M.A. Roth. *Experiences in DBMS Implementation Using an Object-oriented Persistent Programming Language and a Database Toolkit*. OOPSLA, pp.314-328, 1991.

[HMC+00] Adolfo Hernández Aguirre, Ana Belén Martínez Prieto, Juan Manuel Cueva Lovelle y Darío Álvarez Gutiérrez. *Requisitos para una Interfaz Visual de un SGBDOO construido sobre un Sistema Integral OO*. Actas de las I Jornadas de Interacción Persona Ordenador, Granada Junio 2000.

[HXF99] J. Han, Z. Xie y Y. Fu. *Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases*. IEEE Transactions on Knowledge and Data Engineering. Vol 11, nº2, 1999.

[II96] H. Ishikawa y Y. Izumida. *An Object-Oriented Database System Jasmine: Implementation, Application, and Extension*. IEEE Transactions on Knowledge and Data Engineering, vol 8 nº 2, Abril, 1996.

[Izq96] Raúl Izquierdo Castanedo. *Máquina Abstracta Orientada a Objetos*. Proyecto Fin de Carrera 9521I. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Septiembre de 1996.

[Jan95] J. Jannink. *Implementing Deletion in B+ Trees*. SIGMOD Record, vol 24, nº1. Marzo, 1995.

[JDO00] JDO Specification (version 0.8). Disponible en URL http://access1.sun.com/jdo/, Diciembre 2000.

[Jee99] *Jeevan User Guide*. Disponible en URL http://www.w3apps.com/ . Noviembre, 1999.

[Jor98] David Jordan. *C++ Object Databases. Programming with the ODMG Standard*. Addison Wesley, 1998.

[KC86] S. Khoshafian y G. Copeland. *Object Identity*. Proceedings of the Conference on Object-Oriented Programming Systems and Languages (OOPSLA). Oregon, 1986.

[KEL+62] T. Kilburn, D.B.G. Edwards, M.J. Lanigan y F.H. Summer. *One-level Storage System*. IRE Transactions on Electronic Computers. Abril de 1962.

[KGB+90] W. Kim, J.F. Garza, N. Ballou y D.Woelk. *Architecture of the ORION Next-Generation Database System.* IEEE Transactions on Knowledge and Data Engineering, Marzo 1990.

[Kim91] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press. London, 1991.

[Kir99] M. Kirtland. *Designing Component-Based Applications.* Microsoft Press, 1999.

[KKD89] W. Kim, K.C. Kim, A. Dale. *Indexing Techniques for Object-Oriented Databases.* En W. Kim y F.H. Lochovsky (ed): Object-Oriented Concepts, Databases, and Applications. Addison-Wesley, 1989.

[KKM94] A. Kemper, C. Kilger y G. Moerkotte. *Function Materialization in Object Bases: Design, Realization and Evaluation.* IEEE Transactions on Knowledge and Data Engineering, 1994.

[KV92] J.L. Keedy y K. Vosseberg. *Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System.* En Proceedings of the 25th Hawaii International Conference on System Sciences, Hawaii, EE.UU. 1992. Pág. 747-756.

[LMP87] B. Lindsay, J. McPherson y H. Pirahesh. *A Data Management Extension Architecture.* Proceedings of the ACM SIGMOD Annual Conference on Management of data, pp 220-226, 1987.

[LO93] J. Lopes de Oliveira y R. De Oliveira. *Browsing and Querying in Object-Oriented Databases.* Proceedings of the Second International Conference on Information and Knowledge Management pp. 364-373, 1993.

[LOO01] *Laboratory of Object-Oriented Technologies.* Disponible en URL http://www.ootlab.uniovi.es, marzo 2001.

[Loo95] Mary Loomis. *Object Databases. The Essentials.* Addison-Wesley, 1995.

[Lop94] J. Lopes de Oliveira. *On the Development of User Interface Systems for Object-Oriented Databases.* Proceedings of the Workshop on Advanced Visual Interfaces, pp 237-239, 1994.

[LY97] Tim Lindholm y Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley. 1997.

[LL98] W. Lee y D.L. Lee. *Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases.* IEEE Transactions on Knowledge and Data Engineering. Vol 10, nº3, 1998.

[LLP+91] G.M. Lohman, B. Lindsay, H. Pirahesh y K.B. Schiefer. *Extensions to Starburst: Objects, Types, Functions and Rules.* Communications of the ACM, vol 34. Nº10, pp. 94-109, 1991.

[MAC+98a] Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez, Juan Manuel Cueva Lovelle, Francisco Ortín Soler y Jesús Arturo Pérez Díaz. *Incorporating an Object-Oriented DBMS Into an Integral Object-Oriented System.* Proceedings of SCI-ISAS 1998.Orlando, Julio 1998.

[MAC+98b] Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez, Juan Manuel Cueva Lovelle, Francisco Ortín Soler. *BDOviedo3 an Object-Oriented DBMS Incorporated to an Integral Object-Oriented System*. Argentine Symposium on Object Orientation.. Argentina, Septiembre 1998.

[Mae87] P. Maes. *Concepts and Experiments in Computational Reflection.* En Proceedings of the *1987 OOPSLA*. 1987. Pág. 147-155.

[Mck93] W.J. McKenna. *Efficiente Search in Extensible Database Query Optimization: The Volcano Optimizer Generator.* PhD thesis, Department of Computer Science. University of Colorado, 1993.

[MAO+00] Ana Belén Martínez Prieto, Fernando Álvarez García, Francisco Ortín Soler y Juan Manuel Cueva Lovelle. *Un SGBDOO soportado por el Sistema Operativo Distribuido del Sistema Integral Oviedo3.* Simposio Español de Informática Distribuida, Orense, septiembre 2000.

[Mar96] Ana Belén Martínez Prieto. *Introducción a los SGBDOO*. II Jornadas de Tecnologías Orientadas a Objetos. Oviedo, marzo 1996.

[MC98] Ana Belén Martínez Prieto y Juan Manuel Cueva Lovelle. *BDOViedo3: Un SGBDOO sobre una máquina abstracta persistente.* III Jornadas de Investigación y Docencia en Bases de Datos, Valencia, 1998.

[MC99] Ana Belén Martínez Prieto y Juan Manuel Cueva Lovelle. *Técnicas de Indexación para las Bases de Datos Orientadas a Objetos.* Novática, Monográfico Bases de Datos Avanzadas, nº 140, julio-agosto 1999.

[MCA+98] Ana Belén Martínez Prieto, Juan Manuel Cueva Lovelle, Darío Álvarez Gutiérrez y Modesto García García. *Técnicas de Indexación para BDOviedo3.* IV Jornadas de Orientación a Objetos. Bilbao, Octubre 1998.

[MCA96] Ana Belén Martínez Prieto, Juan Manuel Cueva Lovelle y Darío Álvarez Gutiérrez. *Desarrollo del SGBDOO en Oviedo3.* Actas de las I Jornadas de Investigación y Docencia en Bases de Datos. La Coruña, Junio 1996.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction* (2ª Edición). Prentice-Hall, 1997.

[Moo96] D. Moore. *An Ode to Persistence*. Journal Object Oriented Programming, November-December, 1996.

[Mos90] J.E. Moss. *Working with Persitent Objects: to Swizzle or not to Swizzle*. COINS Technical Report 90-38. Department of Computer and Information Science. University of Massachusetts, 1990.

[MS86] D. Maier y J. Stein. *Indexing in an Object-Oriented DBMS*. Proc. IEEE Workshop on Object-Oriented DBMSs. Asilomar, CA, septiembre 1986.

[MS87] D, Maier y J. Stein. *Development and Implementation of an Object-Oriented Database System.* En *Research Directions in Object-Oriented Programming*, B. Shriver, P. Wegner Eds. MIT Press, 1987.

[MS97] D. McKeown y H. Saiedian. *Triggers for object-oriented database systems.* Journal of Object Oriented Programming. Mayo, 1997.

[MVS99] E. Marcos, B. Vera y J. Sáenz. *Propuesta de una Extensión del ODMG para el Soporte de Relaciones Ternarias.* IV Jornadas de Ingeniería del Software y Bases de Datos. Noviembre, 1999.

[MZB+98] R.C. Mauro, G. Zimbrão, T.S Brügger, F.O. Tavares, M.Duran, A.A.B. Lima, P.F. Pires, E. Bezerra, J.A. Soares, F.A. Baião, M.L.Q. Mattoso, G. Xexéo. *GOA++: Tecnologia, Implementação e Extensões aos Serviços de Gerência de Objetos.* 12th Brazilian Symposium on Database Systems. Fortaleza (Brasil) Octubre 1997.

[OAI+97] Francisco Ortín Soler, Darío Álvarez Gutiérrez, Raúl Izquierdo Castanedo, Ana Belén Martínez Prieto y Juan Manuel Cueva Lovelle. *El sistema de persistencia en Oviedo3.* III Jornadas de Tecnologías de Objetos. Sevilla. Octubre de 1997.

[Obj00a] Object Design. *ObjectStore.* Disponible en URL http://www.odi.com/, marzo 2000.

[Obj00b] *ObjectDRIVER Reference Manual*. Versión 1.1. Disponible en URL http://www.inria.fr/cermics/dbteam/. Enero 2000.

[Obj00c] *ObjectStore Database Designer.* Disponible en URL http://www.object-store.net/objectstore/po.html, 2000.

[Obj99] *ObjectFile*. Disponible en URL http://www.ifl-holdings.com/vip/ofile/ofile.html. Noviembre 1999.

[ODM99] *Object Database Management Group*. Disponible en URL http://www.odmg.org. Septiembre 1999.

[ÖDV94] M. T.Özsu, U. Dayal y P. Valduriez. *An Introduction to Distributed Object Management*. Distributed Object Management, Morgan Kaufmann, 1994.

[OMA+99a] Franciso Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez y Juan Manuel Cueva Lovelle. *An Implicit Persistece System on an OO Database using Reflection.* SCI-ISAS, Florida, Julio 1999.

[OMA+99b] Franciso Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez y Juan Manuel Cueva Lovelle. *Implicit Object Persistence on a Reflective Abstract Machine.* Argentine Symposium on Object Orientation. Buenos Aires (Argentina), Septiembre 1999.

[OMA+99c] Franciso Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez y Juan Manuel Cueva Lovelle. *A Reflective Pesistence Middleware over an Object Oriented Database Engine.* 14th Brazilian Symposium on Databases. Florianópolis (Brasil) Octubre 1999.

[OMA+99d] Franciso Ortín Soler, Ana Belén Martínez Prieto, Darío Álvarez Gutiérrez y Juan Manuel Cueva Lovelle. *Diseño de un Sistema de Persistencia Implícita mediante Reflectividad Computacional.* IV Jornadas de Ingenería del Software y Bases de Datos, Cáceres, Noviembre 1999.

[OMG97] Object Management Group. *Common Object Request Broker Architecture and Specification (CORBA), revision 2.1*. Object Management Group. Agosto de 1997. Disponible en URL: http://www.omg.org.

[ÖPR+93] M.T. Özsu, R. Peters, B. Irani, A. Lipka, A. Munoz y D. Szafron. *TIGUKAT Object Management System: Initial Design and Current Directions*. Proc. Of the Centre for Advanced Studies Conference (CASCON), Octubre 1993.

[Ora99a] Oracle Corporation. *Oracle 8i™ Objects and Extensibility Option. Features Overview*. Disponible en URL http://www.oracle.com, Febrero 1999.

[Ora99b] Oracle Corporation. *All Your Data: The Oracle Extensibility Architecture.* White paper. Disponible en URL http://www.oracle.com, Febrero 1999.

[Ort97] Francisco Ortín Soler. *Diseño y Construcción del Sistema de Persistencia en Oviedo3*. Proyecto Fin de Carrera 972001. Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón de la Universidad de Oviedo. Septiembre de 1997.

[Ozs94] T. Ozsu. *Transaction Models and Transaction Management in Object-Oriented Database Management Systems.* En Advances in Object Oriented Database Systems. A. Dogac, T. Ozsu, A. Biliris y T. Sellis, Springer Verlag, 1994.

[Pér99] José Antonio Pérez Pérez. *Implementación del Estándar ODMG sobre Motores de Bases de Datos Orientados a Objetos y Relacionales.* Proyecto Fin de Carrera 992029. Escuela Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón. Universidad de Oviedo, 1999.

[Poe00] *Poet Object Server Suite 6.0 User Guide*. Disponible en URL http://www.poet.com/, 2000.

[PSE99] *ObjectStore PSE/PSE Pro for Java API User Guide*. Disponible en URL http://www.odi.com/objectstore/. Mayo 1999.

[PSS+87] H.B. Paul, H.J. Schek, M.H. Scholl, G. Weikum y U. Deppisch. *Architecture and Implementation of the Darmstadt Database Kernel System.* Proceedings of the ACM SIGMOD Annual Conference on Management of Data, pp 196-207, 1987.

[RBK+91] F. Rabitti, E. Bertino, W. Kim y D. Woelk. *A Model of Authorization for Next-Generation Database Systems.* ACM Transactions on Database Systems, 16(1), pp 88-131, 1991.

[RC87] J.E. Richardson y M.J. Carey. *Programming Constructs for Database System Implementation in EXODUS.* Proceedings of the ACM SIGMOD Annual Conference on Management of Data, 1987

[RCS93] J. E. Richardson, M. J. Carey y D.T. Schuh. *The Design of the E Programming Language.* ACM Transactions on Programming Languages and Systems, vol 15, nº3, 1993.

[Ren97] M. Rennhackkamp. *Extending Relational DBMs*. DBMS, Diciembre 1997

[RK95] S. Ramaswamy y C. Kanellakis. *OODB Indexing by Class Division*. ACM SIGMOD, 1995.

[Rog96] Dale Rogerdson. *Inside COM.* Microsoft Press. 1996.

[Rot91] D. Rotem. *Spatial Join Indices*. Proc. Seventh Int'l Conf. Data Eng., Japan,Abril 1991.

[SKS97] A. Silberschatz, H.F. Korth y S. Sudarshan. *Database System Concepts.* Third Edition. McGraw-Hill, 1997.

[SM96] M. Stonebraker y D. *Moore. Object–Relational DBMSs. The Next Great Wave.* Morgan Kaufmann Publishers, 1996.

[SMF96] Alan C. Skousen, Donald S. Miller y Ronald G. Feigen. *The Sombrero Operating System: An Operating System for a Distributed Single Very Large Address Space – General Introduction*. Technical Report TR-96-005, Arizona State University, EE.UU.1996.

[SPS+90] H.J. Schek , H.B. Paul, M.H. Scholl y G. Weikum. *The DASDBS Project: Objectives, Experiences, and Future Prospects* in Special Issue on Database Prototype System, IEEE Transactions on Knowledge and Data Engineering, 2,1, 1990.

[SQ99] E. Soares Ogasawara y M.L. de Queirós Mattoso. *Uma Avaliação Experimental sobre Técnicas de Indexação em Bancos de Dados Orientadados a Objetos.* En Proc. 14th Brazilian Symposium on Databases. Florianópolis (Brasil) Octubre 1999.

[SR86] M.R. Stonebraker y L.A. Rowe. *The Design of POSTGRES*. Proc. 1986 ACM SIGMOD Conference on Management of Data, Washington DC, 1986.

[SRL+90] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Berstein y D. Beech. *Third-Generation Database System Manifesto.* SIGMOD Record 19(3), Julio 1990.

[SS86] H.J. Schek, M.H. Scholl. *The Relational Model with Relation-Valued Attributes.* Information Systems 11,2. 1986.

[SS94] B. Sreenath y S. Seshadri. *The hcC-Tree: An Efficient Index Structure for Object Oriented Databases.* International Conference on VLDB, 1994.

[Sto86] M.R. Stonebraker. *Inclusion of New Types in Relational Data Bases Systems.* Proc. 2nd IEEE Data Engineering Conf. Los Angeles, CA, 1986.

[Str99] StreamStore™ Home. *StreamStore Technical Home.* Disponible en URL http://www.bluestream.com/products/streamstore/technical/index.htm, Noviembre 1999.

[Sub96] Kazimierz Subieta. *Object-Oriented Standards: Can ODMG OQL be Extended to a Programming Language?* Proceedings or the International Symposium on Cooperative Database Systems for Advanced Applications, pp546-555, Japon, Diciembre 1996.

[Sun00] Sun Microsystems. *Development Tools Java BlendTM.* Disponible en URL http://www.sun.com/software/javablend/index.html, enero 2000.

[Taj00] Lourdes Tajes Martínez. *Sistema de Computación para un Sistema Operativo Orientado a Objetos basado en una Máquina Abstracta Reflectiva Orientada a Objetos.*Tesis Doctoral. Universidad de Oviedo, 2000.

[TYT92] T. Tenma, Y. Yokote y M. Tokoro. *Implementing Persistent Objects in the Apertos Operating System.* En Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOOS'92). Dourdan, Francia. Septiembre 1992.

[Val87] Patrick Valduriez. *Join Indices.* ACM Transactions on Database Systems, vol. 12 nº 2, 1987.

[VRH93] J. Vochteloo, S. Russell y G. Heiser. *Capability-Based Protection in the Mungi OS.* En Proceedings of the IWOOOS'93. Diciembre 1993.

[YM98] C. Yu y W. Meng. *Principles of Database Query Processing for Advanced Applications.* Morgan Kaufmann, 1998.