PROPUESTA DE UNA ARQUITECTURA Y UN COMPONENTE GENERICO PARA ACCESO A BASES DE DATOS EN APLICACIONES ORIENTADAS A OBJETOS

TESIS

MAESTRIA EN CIENCIAS EN TECNOLOGIA INFORMATICA

INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

POR

ANDREAS MOLINA VILLA

JUNIO DE 2000

PROPUESTA DE UNA ARQUITECTURA Y UN COMPONENTE GENERICO PARA ACCESO A BASES DE DATOS EN APLICACIONES ORIENTADAS A OBJETOS

TESIS

MAESTRIA EN CIENCIAS EN TECNOLOGIA INFORMATICA

INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

POR

ANDREAS MOLINA VILLA

JUNIO DE 2000

PROPUESTA DE UNA ARQUITECTURA Y UN COMPONENTE GENERICO PARA ACCESO A BASES DE DATOS EN APLIACIONES ORIENTADAS A OBJETOS

TESIS

MAESTRÍA EN CIENCIAS EN TECNOLOGÍA INFORMÁTICA

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

POR

ANDREAS MOLINA VILLA

JUNIO 2000

PROPUESTA DE UNA ARQUITECTURA Y UN COMPONENTE GENERICO PARA ACCESO A BASES DE DATOS EN APLIACIONES ORIENTADAS A OBJETOS

TESIS

MAESTRÍA EN CIENCIAS EN TECNOLOGÍA INFORMÁTICA

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

POR

ANDREAS MOLINA VILLA

JUNIO 2000

A mis papas y mi familia
A Arelis

Agradecimientos

A mi maestro Guillermo Jiménez, por todo su apoyo y sus aportaciones en este proyecto. Por ser un excelente maestro.

A Ciro y Carlos por la ayuda y el apoyo que me han brindado desde que los conozco.

A los tres gracias sobre todo por su amistad.

A todos los que han sido mis maestros, por contribuir en mi formación académica y personal. Gracias por sus enseñanzas.

A mi apreciada maestra Naoi sensei, por los cuatro años de alegría y ánimos que me dio. Por su calidad como persona y como maestra.

Al ITESM por la educación, formación académica, profesional y personal que me ha dado. Por todos los momentos gratos que he pasado en el Instituto.

A mis amigos por todos esos momentos alegres.

A Arelis por su ayuda en la redacción de la tesis, pero sobre todo por su alegría y motivación.

A la reina Yoalli por su cariño.

A mis tíos por considerarme parte de su familia.

A mis padres por todo su apoyo y su amor. Por el esfuerzo tan grande que hicieron en brindarme una educación.

andreas.

Resumen.

Muchas aplicaciones requieren el acceso a información almacenada en bases de datos.
En la actualidad las bases de datos más usadas son las bases de datos relacionales, y los ambientes de desarrollo orientados a objetos son los que predominan en el mercado.
Ambas tecnologías están basadas en paradigmas diferentes, lo que ocasiona que la comunicación e integración entre ambas sea difícil de lograr.
En un desarrollo orientado a objetos con el uso de bases de datos relacionales, gran parte del código de la aplicación se dedica a accesar los datos de la base de datos.
La comunicación entre la aplicación y la base de datos se realiza mediante el uso de instrucciones que la aplicación da a la base de datos y mediante datos que regresa la base de datos a la aplicación. La forma más común de especificar las instrucciones a las bases de datos relacionales es mediante SQL (Structured Query Language), el uso de SQL requiere el conocimiento del modelo de datos de la base de datos, y la manera más común que una base de datos regresa información a la aplicación es mediante registros.
En una aplicación orientada a objetos, la aplicación debe estar pensada e implementada en términos de clases, objetos y métodos, más no en términos de instrucciones SQL, ni registros, que son los elementos con los que trabajan las bases de datos relacionales.
Dada esta disparidad, gran parte del desarrollo en una aplicación es dedicado a construir las instrucciones para la base de datos y la conversión de las estructuras de datos usadas por la base de datos y las usadas por la aplicación.
Las instrucciones de acceso a datos, hacen que la aplicación quede acoplada a la base de datos. Si la base de datos sufre cambios estructurales, las probabilidades de que se tenga que modificar la aplicación son altas.
En el desarrollo de aplicaciones se busca que la aplicación sea adaptable a los cambios, ya sean funcionales o tecnológicos. También se busca tener que desarrollar la menor parte posible de una aplicación y reusar partes previamente construidas.
En el caso de las aplicaciones que tienen acceso a bases de datos es deseable que la aplicación se adapte a cambios en el modelo de datos de la base de datos y que el código desarrollado para la interacción con la base de datos sea el menor posible.
La tesis propone un modelo arquitectónico en capas para estructurar aplicaciones con acceso a datos, que le den independencia a la aplicación de la base de datos, facilitando así su adaptabilidad ante cambios en esta, y un componente genérico que asile la comunicación con la base de datos (creación de instrucciones de acceso a datos y conversión de estructuras de datos):
Las ventajas de esta arquitectura son una mayor flexibilidad y adaptabilidad con respecto a posibles cambios en la base de datos y evitar la necesidad de que en la aplicación se tenga que construir código para comunicarse con la base de datos.

Tabla de contenido

INTRODUCCIÓN 1
1.1 Antecedentes 1
1.2 Objetivo de la tesis 1
1.3 Justificación 2
1.3 Estructura de la tesis 2

CONCEPTOS TEÓRICOS 3
2.1 Introducción 3
2.2 Arquitecturas de Software 3
2.2.1 Definiciones 3
2.2.2 Componentes de software 4
2.2.3 Arquitecturas de Software e Implementación 4
2.2.4 Arquitecturas de Software y desarrollo de aplicaciones 5
2.3 Bases de Datos 6
2.3.1 Manejadores de Bases de Datos Relacionales 6
2.3.2 API's para acceso a base de datos 7
2.3 Aplicaciones n-tired (en capas) 8
2.3.1 Esquemas para implementar aplicaciones en capas 10
2.4 Metodología 11

ARQUITECTURA PARA ACCESO A DATOS 12
3.1 Introducción 12
3.2 Características deseables 12
3.3 Alternativas 12
3.4 Arquitectura propuesta 16

FRAMEWORK PARA ACCESO A DATOS 18
4.1 Introducción 18
4.2 Frameworks 18
4.3 Especificaciones funcionales del framework 18
4.4 Diseño General 22
4.4.1 Consulta de Objetos 23
4.4.2 Actualización de un objeto 24
4.4.3 Actualización de objetos 25
4.4.4 Borrado de un objeto 25
4.5 Detalle de las clases 25
4.6 Mecanismos para mapear objetos en la base de datos relacional 31
4.7 Metadatos 35
4.8 Forma de especificar criterios 36
4.9 Algoritmos detallados para operaciones de consulta transaccionales 37

4.10 Mecanismo de extensión del framework 42
4.11 Implementación en Java 42

EXPERIENCIAS 44
5.1 Ventajas y Desventajas del uso de la arquitectura y el framework 44
5.2 Problemas de implementación. 45
5.3 Escenario de prueba. 47

CONCLUSIONES Y TRABAJO FUTURO 49
6.1 Conclusiones 49
6.2 Trabajo Futuro 50

ANEXO. IMPLEMENTACIÓN DE LAS CLASES DEL COMPONENTE GENÉRICO 52

BIBLIOGRAFÍA 71

Lista de Figuras

Figura 1. Tipos de drivers de JDBC. 8

Figura 2. Esquema de una aplicación en capas. 10

Figura 3. Estructura básica de un componente vertical. 13

Figura 4. Diagrama de una aplicación construida a partir de componentes verticales. 13

Figure 5. Diagrama de una aplicación que usa componentes horizontales. 14

Figura 6. Diagrama de aplicación usando clases auxiliares. 15

Figura 7. Diagrama de una clase separada en niveles por medio de herencia. 15

Figura 8. Diagrama de arquitectura en niveles para acceso a datos. 16

Figura 9. Diagrama de acceso a múltiples bases de datos a través del framework. 19

Figura 10. Diagrama básico de la relación entre Alumno y Carrera. 20

Figura 11. Diagrama de clases de la relación Alumno y Carrera. 20

Figura 12. Diagrama general del framework de acceso a datos. 23

Figura 13. Diagrama de secuencias de la operación de consulta. 23

Figura 14. Diagrama de secuencia de actualización de un objeto. 24

Figura 15. Diagrama de la clase Broker. 25

Figura 16. Diagrama de la jerarquía de clases BaseDatos. 26

Figura 17. Diagrama de clases de la jerarquía Criterio. 27

Figura 18. Diagrama de clases de la jerarquía Operación. 28

Figura 19. Diagrama de la clase FabricaObjeto. 29

Figura 20. Diagrama de clases de la jerarquía Sql. 30

Figura 21. Diagrama de clases del mapa entre las clases y la base de datos. 30

Figura 22. Diagrama de clases de la relación entre Alumno y Carrera. 33

Figura 23. Diagrama de clases de la implementación final de la clase Alumno y Carrera. 33

Figura 24. Diagrama entidad relación representando la relación entre Alumno y Carrera. 33

Figura 25. Diagrama entidad relación entre Alumno y Carrera. 34

Figura 26. Diagrama de clases de la clase Alumno y Carrera. 37

Figura 27. Modelo relaciona de inscripciones 1. 48

Figura 28. Modelo relacional de inscripciones 2. 48

Lista de Tablas

Tabla 1. Ejemplo de formato de un registro de log. 21
Tabla 2. Mapa de los atributos de la clase Alumno y Carrera en la base de datos. 33
Tabla 3. Representación de la relación de las clases Alumno y Carrera. 34
Tabla 4. Mapa los atributos de las clases Alumno y Carrera en la base de datos. 34
Tabla 5. Mapa de la relación entre la clase Alumno y Carrera. 35

Capítulo1

Introducción

1.1 Antecedentes

Los sistemas de información son cada vez más importantes en nuestra época, pues los procesos dependen de su operación. Vivimos en un mundo que depende de la información; día a día surgen nuevas necesidades para aplicaciones que nos permitan administrar y aprovechar la información.

Los sistemas de información deben tener la capacidad de adaptarse a los cambios. No es posible concebir a una aplicación estática, puesto que los procesos de negocios cambian con el tiempo, las políticas cambian, y con éstas deben cambiar las aplicaciones en las que se apoyan estos procesos. Las aplicaciones deben ser diseñadas a modo que sean lo más fácil adaptables a los cambios.

También necesitamos tener la capacidad de construir aplicaciones con menor tiempo de desarrollo y de mejor calidad.

Necesitamos métodos y herramientas que nos permitan desarrollar aplicaciones, disminuir el tiempo de desarrollo y que faciliten hacer adaptaciones a las aplicaciones cuando sea necesario.

Una manera de lograr este objetivo es construyendo aplicaciones a partir de elementos previamente construidos, basados en una estructura genérica de la aplicación. De esta manera podemos disminuir el tiempo de construcción, puesto que en lugar de desarrollar completamente el sistema, podemos ensamblarlo a partir de componentes ya existentes. Con este enfoque es posible modificar el comportamiento del sistema cambiando o agregando los componentes (bloques de construcción) de la aplicación.

Para que un sistema pueda ser construido de esta manera debe contar con una estructura que lo identifique, en la que sea posible agregar, quitar o intercambiar componentes, y contar con los componentes de los que estará compuesta la aplicación.

Una aplicación puede ser dividida en capas o niveles según su funcionalidad. Se pueden identificar tres niveles presentes en los sistemas de información: nivel de acceso a datos (accesa los datos de una fuente de almacenamiento), nivel de lógica (provee la lógica de la aplicación) y nivel de presentación (presenta los datos al usuario)

El trabajo de tesis propone una arquitectura para acceso a base de datos relacionales en lenguajes orientados a objeto, basada en niveles. En esta arquitectura todas las operaciones de acceso a datos se realizan en un solo nivel, y se delegan a un componente genérico encargado de realizar la comunicación con la base de datos y realizar la conversión entre los objetos usados por la aplicación y las estructuras usadas por la base de datos.

El objetivo principal en el desarrollo de esta arquitectura es lograr independencia y adaptabilidad de una aplicación con respecto a la(s) base(s) de datos que usa. El segundo objetivo es disminuir el código necesario para el acceso a datos (consultas y actualizaciones)

La tesis incluye el diseño de un componente genérico de acceso a datos, para el lenguaje Java.

1.2 Objetivo de la tesis

Proponer un modelo o arquitectura para la construcción de aplicaciones que tengan acceso intensivo a bases de datos, e implementar un componente genérico y reutilizable que implemente todas las operaciones que sean requeridas sobre la base de datos.

El objetivo de la arquitectura es hacer a las aplicaciones adaptables a los cambios que se efectúen en la base de datos y hacer más reutilizables los componentes de la aplicación. De esta manera lo que se busca es que el desarrollador no tenga que dedicarse a escribir el código de acceso a los datos, y que el código de la aplicación no tenga que cambiar si ocurre algún cambio en la base de datos.

1.3 Justificación

Características muy deseables en el desarrollo de una aplicación es que es que sea capaz de adaptarse a los cambios, que se puedan reusar componentes ya creados en su desarrollo y que los componentes de la aplicación puedan ser utilizados en otras aplicaciones.

Por la necesidad de comunicación que existe entre la aplicación y la base de datos, se produce un acoplamiento de la aplicación a la base de datos. Mientras más acoplada esté la aplicación a la base de datos, menor será la adaptabilidad de la aplicación con respecto a cambios en la base de datos y menor será la posibilidad de reusar los componentes de la aplicación en otros desarrollos.

Por lo tanto es necesario estructurar las aplicaciones de la manera que sea lo más independiente posible a la base de datos. La tesis propone una arquitectura que hace posible la independencia de la aplicación y la base de datos, así como un componente que realiza las operaciones de acceso a los datos, de tal manera que la aplicación especifica las operaciones sobre los datos en el mismo lenguaje orientado a objetos sin necesidad de conocer la estructura de la base de datos.

Por otro lado mientras más independiente sea la aplicación de la base de datos, menor será la preocupación por el detalle de cómo accesar los datos. Esto ocasiona una programación de más alto nivel. Si la aplicación es independiente de la implementación de la base de datos los componentes de la aplicación no tienen por que depender de la estructura con la que estén representados los datos en la base de datos, y esto les da una posibilidad mayor de ser reusados en otras aplicaciones. Si la estructura de la base de datos cambia, no tienen por que cambiar los componentes de la aplicación, ni el código de la aplicación que accesa a los datos.

Al hacer a los componentes de la aplicación independientes del mecanismo en el que se almacenan los datos y la manera de accesarlos, los hace más reusables en otras aplicaciones.

Si se cuenta con un componente genérico que provea el acceso a los datos, se evita la necesidad de escribir código para realizar esta función en cada aplicación.

1.3 Estructura de la tesis

En el capítulo 2 se presentará el marco teórico referente a arquitecturas de software, bases de datos relacionales y mecanismos de acceso a base de datos en Java. Las arquitecturas de software y las bases de datos relaciones son los fundamentos de la arquitectura propuesta, mientras que el lenguaje Java proporciona herramientas que dan lineamiento al diseño de la aplicación y los componentes genéricos.

En el capítulo 3 se especifica la arquitectura para el desarrollo de aplicaciones que requieren de acceso a datos. Aquí se detallan las características de la arquitectura, sus beneficios, y la forma de construir una aplicación bajo esta arquitectura.

En el capítulo 4 se especifica un componente genérico y reutilizable para el acceso a datos. Se definen sus funciones y se propone un posible diseño para el componente.

En el capítulo 5 se analizan las ventajas y desventajas del uso de esta arquitectura en un prototipo de aplicación.

En el capítulo 6 presentan las conclusiones y comentarios a los que se llegó en el trabajo de investigación, y se propone un trabajo futuro de investigación, en base a las experiencias obtenidas.

Al final se incluye un anexo que contiene el código Java de las clases principales que implementan el prototipo elaborado.

Capítulo 2

Conceptos teóricos

2.1 Introducción

En este capítulo se definen conceptos de arquitecturas de software que nos darán la pauta para construir aplicaciones en las que se facilite la adaptabilidad de las mismas y el reuso de código. En el capítulo se hace un especial énfasis en la arquitectura de aplicaciones en capas, pues es la arquitectura en la que se trabajará en el resto de la tesis. Posteriormente se tratan las bases de datos relacionales, medios para acceso a base de datos desde el lenguaje Java.

2.2 Arquitecturas de Software

2.2.1 Definiciones

No hay una definición única para arquitectura de software.

En términos generales puede definirse como la estructura de un sistema, el cual está integrado por componentes y sus relaciones.

Una arquitectura es una especificación de los componentes de un sistema y la comunicación entre ellos [LUCK95].

Mientras el tamaño y la complejidad de un sistema (de software) aumenta, el problema de diseño, va más allá de los algoritmos y las estructuras de datos de el sistema, diseñar y especificar la estructura global del mismo emerge como un nuevo tipo de problema [GARL94]. Entender la estructura de un sistema involucra entender la forma en la que están organizados los componentes o módulos, la estructura de control, protocolos de comunicación, distribución física y funcionalidad de los elementos.

Una arquitectura es importante porque a partir de ésta se definen ciertas características o propiedades del sistema, provee restricciones sobre la manera en la que deben de ser construidos los componentes y la manera en que se comunican. Proporciona una abstracción de la estructura de la aplicación y omite los detalles de implementación. De esta manera es posible entender al sistema, y usar a la arquitectura como patrón para el desarrollo de sistemas similares. Una arquitectura puede guiar el proceso de desarrollo para satisfacer sus requerimientos funcionales.

Una arquitectura de software impone restricciones al sistema: cualquier cosa que no esté especificada en la arquitectura puede tomar cualquier forma deseada por el diseñador o el implementador [PERR92].

Existen otras ventajas en el uso de arquitecturas, como la posibilidad de definir los roles del equipo de desarrollo. Puesto que la arquitectura puede verse como un plano del sistema, a partir de éste es muy fácil decidir los elementos que deben ser construidos y sus características, con esto es posible asignar los roles al equipo de desarrollo, por ejemplo un equipo puede dedicarse a construir la interfaz del usuario, otra los protocolos de comunicación, y otros lo concerniente al acceso a las base de datos.

Una *arquitectura de referencia* es una estructura o mapa de un sistema que es usada para construir aplicaciones para un dominio en particular y que son muy similares entre sí. Una arquitectura de referencia se deriva analizando el dominio de la aplicación [BATO95].

Una *familia de sistemas* es un conjunto de sistemas que están fuertemente relacionados. Esta relación puede darse en funcionalidad, estructura, etc. El concepto implica que en lugar de construir productos individuales, se construyan conjuntos de sistemas a los cuales sólo es necesario configurar cierta parte que es específica para la aplicación, más no así las partes que son generales a las aplicaciones. Por ejemplo se podría pensar como familias de sistemas a los sistemas de contabilidad, sistemas de administración de escuelas, sistemas de navegación de

aviones, etc. Por supuesto las variables con las que podemos identificar la similitud de los sistemas dependen del dominio de la aplicación.

Las decisiones de diseño especificadas en una arquitectura pueden ser usadas por cada sistema que pertenezca a la familia de sistemas que defina la arquitectura de referencia, de tal forma que no tienen que ser reinventadas en cada nueva aplicación.

Una posible manera de construir software es mediante líneas de productos. Una *línea de productos* es una guía para construir aplicaciones relacionadas o similares. El concepto se basa en crear una serie de productos altamente similares, de tal forma que los esfuerzos que se hagan en desarrollar un producto se puedan aplicar para realizar los demás productos de la línea de productos[BATO98].

Se pueden construir líneas de productos si las aplicaciones a construir comparten una arquitectura común.

El reuso a gran escala es posible mediante planeación a nivel arquitectónico [CLEM96]. Este enfoque es de particular interés para la investigación, puesto que se busca encontrar características comunes que nos permitan definir una arquitectura para los sistemas de información y poder producir aplicaciones mediante el reuso de la estructura de la aplicación y componentes existentes.

El enfoque que se da en la investigación es el definir una arquitectura de referencia buscando el reuso de componentes y la facilidad para adaptar el sistema a nuevos requerimientos.

2.2.2 Componentes de software

Por *componente de software* podemos entender cualquier objeto de software con existencia independiente, éste puede ser un procedimiento, un objeto, una aplicación completa, etc.

Un componente puede ser una clase o un conjunto de clases interrelacionadas, y su propósito es servir como bloque de construcción de una aplicación [BATO98].

Un problema al definir un componente, es que es difícil determinar las fronteras de éste y los servicios que debe ofrecer. La razón es por que las clases se relacionan entre si, y los servicios que ofrezcan depende del contexto en que se use el componente. De esta manera si un componente se usa en cierto contexto requiere proporcionar ciertos servicios, los cuales pueden ser muy diferentes a los que se requieran en otro contexto. Es posible que dos sistemas implementen una misma clase y sin embargo el conjunto de operaciones que requieran de ésta sea muy diferente [BATO96].

Una característica deseable de un componente es que tenga una interfaz o un punto lógico de conexión entre el componente y el sistema. Una *interfaz* debe de ocultar los detalles de implementación del componente, pero debe de proveer información necesaria sobre sus características al sistema.

De esta manera un componente puede representar un elemento visual en una interfaz de usuario, una entidad en algún dominio (empleado, alumno), un manejador de base de datos, etc.

Es importante la manera en que se definen los componentes. Para que éstos puedan ser re-utilizados, deben de proveer una función específica, así como tener un mecanismo mediante el cual se accese su funcionalidad, pero que a la vez oculte los detalles de su implementación.

2.2.3 Arquitecturas de Software e Implementación

La arquitectura de referencia debe proveer medios para conectar estos elementos. Es decir especifica los mecanismos en los que se van a comunicar los elementos que la componen. Por ejemplo una arquitectura en particular puede especificar que los elementos se comunican mediante la invocación de métodos públicos, o tal vez mediante el uso de interfaces, o mediante el uso de la herencia.

Como en la arquitectura de construcción, en la arquitectura de software es importante tener múltiples vistas del sistema: los estilos arquitectónicos, estilos e ingeniería y estilo y materiales [PERR92].

El estilo de materiales define los elementos y herramientas que se tienen disponibles para construir el software, éstas incluyen las características del lenguaje de programación, los componentes que se tengan implementados, etc.

El estilo arquitectónico define arreglos de componentes y restricciones propias del estilo, así mismo provee un vocabulario reducido con el cual es posible definir a una arquitectura en particular. Limita el tipo de elementos que se usarán en la construcción y sus relaciones [PERR92].

Los materiales representan las herramientas y elementos con los cuales quedará construido el sistema, como el lenguaje de programación, hardware, etc. Un lenguaje de programación restringe la manera en que se desarrolla un sistema, favorece a ciertas formas y desfavorece a otras. Por ejemplo un lenguaje orientado a objetos puede favorecer el desarrollo orientado a objetos, ser "neutral" ante el desarrollo procedural y desfavorecer el desarrollo funcional.

Los materiales tienen ciertas propiedades que son explotados en un estilo particular [PERR92].

Es de importancia este aspecto para la investigación puesto que se explorará la manera de explotar las características del lenguaje de programación Java.

Una vez que se tienen definidos los componentes que constituyen la aplicación, es necesario conectarlos para formar la operación del sistema. Éste es un problema importante para una arquitectura. Los esquemas que se usen para la conexión de componentes dependen en gran parte de las características del lenguaje en el que se desarrollan los componentes. En el caso de Java se explorarán dos formas para conectar los componentes: basado en clases e interfaces. Ambos conceptos son soportados por el lenguaje

2.2.4 Arquitecturas de Software y desarrollo de aplicaciones

Los desarrollos deben centrarse en satisfacer los requerimientos funcionales, y no en aspectos tecnológicos como en accesar los datos y presentarlos. Para que esto sea posible, se necesita contar con una infraestructura de servicios genéricos que nos permita dedicar la mayor parte del tiempo en satisfacer las necesidades del usuario y no en escribir cada vez código para lidiar con la tecnología (base de datos, internet, etc.) [FINC98].

Los componentes que nos pueden proveer la infraestructura que pueda ser reusada en cada aplicación que se construya, son *componentes genéricos horizontales*. Estos deben cumplir con ciertos requisitos para que puedan ser usados en cada aplicación, los cuales son especificados por la arquitectura.

Sin embargo no basta con tener una serie de componentes horizontales que nos provean servicios genéricos. Para que el desarrollador pueda satisfacer adecuadamente los requerimientos de la aplicación necesita pensar en términos de la aplicación y sus componentes (del dominio de la aplicación).

El contar con una arquitectura en la que se tengan definidos los componentes de funcionalidad genérica, y se puedan construir sistemas usando la infraestructura existente permite centrarse en los aspectos propios de la aplicación e incrementar la facilidad para construir aplicaciones.

Inclusive la facilidad para adaptar los sistemas a nuevos requerimientos es incrementada, puesto que se tienen bien localizadas las funciones que se realizan.

Mientras menos código deba ser desarrollado para accesar a cada una de las funcionalidades genéricas, obtendremos mayor facilidad y velocidad en el desarrollo de una aplicación.

Una alternativa para que esto sea posible, es que los componentes genéricos tengan la capacidad de averiguar las necesidades de los clientes que les solicitan un servicio y realizar todas las actividades necesarias para satisfacerlo.

En el caso del acceso a datos, esto significa que el componente debe de averiguar de su cliente la operación que requiere (actualizar o consultar), y la información que requiere de la base de datos (columnas y tablas), obtener el esquema de la base de datos, construir la consulta que accese los datos y regresarlos al usuario.

2.3 Bases de Datos

2.3.1 Manejadores de Bases de Datos Relacionales

Un ambiente de un sistema manejador de bases de datos es un sistema integrado de hardware, software y gente, que esta diseñado para facilitar el almacenamiento, recuperación y control de recursos de información [HAGE95].

Una *base de datos* es un repositorio de datos almacenados, que es tanto integrada como compartida [DATE86].

Para describir la estructura de una base de datos es necesario definir el concepto de *modelo de datos*, el cual se puede definir como una colección de conceptos para describir datos, relaciones y la semántica asociada con los datos, así como restricciones de consistencia. El propósito del modelo de datos es representar los datos y hacerlos entendibles [HAGE95].

Un modelo de datos consta de los siguientes conceptos: entidad, atributo y relación.

- Una *entidad* es un concepto o evento acerca del cual se desea almacenar información.
- Un *atributo* es una propiedad que describe a la entidad, es una propiedad o característica.
- Mientras que una *relación* describe una conexión o asociación entre dos entidades [HAGE95].

Las asociaciones entre dos entidades pueden ser de los siguientes tipos:

- Asociación uno a uno: Dadas dos entidades A y B, un valor dado de A tiene cero o un valor asociado con B. Viceversa un valor de la entidad B tiene a lo más un solo valor asociado con A
- Asociación uno a muchos: Significa que dadas dos entidades A y B, un valor de la entidad esta asociado con cero, uno o muchos valores de la entidad B, mientras que un valor de la entidad B está asociado con solo un valor de la entidad A.
- Asociación muchos a muchos: Cada valor de la entidad A está asociada con cero o muchos valores de le entidad B; también los valores de B están asociados con cero, uno o muchos valores de A [HAGE95]. Una asociación de muchos a muchos se puede descomponer como dos asociaciones de uno a muchos.

Una base de datos relacional está formada por una colección de tablas, a cada una de las cuales se le asigna un nombre único, y las cuales están formadas por columnas que también tienen un nombre único.

Las asociaciones entre entidades también se representan mediante tablas.

En una tabla, cada columna contiene valores acerca del mismo atributo de la entidad, y cada columna de la tabla debe tener un valor simple, cada columna tiene un nombre distinto, el orden de las columnas es indistinto [HAGE95]. En una tabla un registro está compuesto por una tupla de valores. Si se desea identificar únicamente a un renglón o registro, se usa el concepto de llave primaria:

Una *llave primaria* es un dato que identifica únicamente a un registro, es una combinación de uno o mas atributos cuyos valores diferencian cada registro de la tabla.

Uno de los objetivos de las bases de datos es mantener la integridad de la información. Las llaves primarias son un mecanismo para garantizar la integridad dentro de una tabla, el cual evita que se cuenten con registros repetidos. Otro mecanismo son las *llaves foráneas*, las cuales se usan para garantizar la integridad referencial entre dos tablas. Una llave foránea es un conjunto de columnas en una tabla que referencían los campos de la llave primaria de otra tabla.

Esta restricción requiere que la llave primaria de una tabla se mantenga con la llave foránea de otra [HAGE95].

Para administrar la base de datos, se requiere un software, el cual es propiamente el sistema manejador de la base de datos relacionales, este software es el que se encarga de almacenar y obtener los datos cuando se le pida, y mantener la integridad de la base de datos.

La manera más usada para comunicarse con los manejadores de base de datos es el lenguaje *SQL* (Structured Query Language), el SQL se ha establecido como el leguaje de base de datos relacional estándar [KORT93].

El lenguaje está formado por instrucciones de definición de datos (DDL), las cuales proporcionan comandos para definir tablas, relaciones, crear índices, etc.; y un conjunto de instrucciones de manipulación de datos (DML). Este lenguaje incluye instrucciones para consultar, modificar, agregar o eliminar tuplas de las tablas [HAGE95].

En la base de datos todo el acceso a los datos se hace a través de SQL.

2.3.2 API's para acceso a base de datos

Un *API* o Aplication Program Interface es una colección de métodos de acceso a un programa, módulo o componente de software, de tal manera que la funcionalidad de este software puede ser utilizada a partir de otro software independiente. La manera en la que se accesa su funcionalidad es mediante llamadas a las funciones del API. Generalmente los API's de un producto son estándares y son publicadas sus especificaciones (para que puedan ser accesadas desde otros programas).

Para poder tener acceso a una base de datos, es necesario contar con una herramienta que nos permita establecer la comunicación con un manejador de base de datos y pasar instrucciones y datos al servidor. Para el caso específico de Java existen varias herramientas: como el JDBC y el JSQL.

JDBC es un API de acceso a base de datos, en términos simples el API de JDBC define un conjunto de clases de Java que permiten a un programa en Java tener acceso a una base de datos. El JDBC por si solo no es ningún producto, sino una especificación de una interfaz estándar para el acceso a base de datos. Basados en esta especificación, existen drivers específicos para tener acceso a alguna base de datos en particular. Un driver de JDBC provee clases que definen un medio estándar de acceso a los datos y de paso de instrucciones al servidor de base de datos y capacidades como la conversión de tipos de dato del servidor de base de datos, a los tipos de datos de Java.

Es decir las clases de JDBC permiten al programador ejecutar instrucciones de SQL para solicitar información de alguna base de datos y procesar el resultado (datos que regresa el servidor de base de datos) [LINT98].

Hay dos niveles que constituyen al JDBC: el API de JDBC y el driver de JDBC. El API de JDBC son las funciones que usan los programadores para accesar a las base de datos, por otro lado los proveedores de los drivers deben cumplir con esta API y realizar las acciones sobre la base de datos. Un driver de JDBC es en realidad un grupo de clases.

Por las características del lenguaje Java existen 4 tipos de drivers (Figura 1):

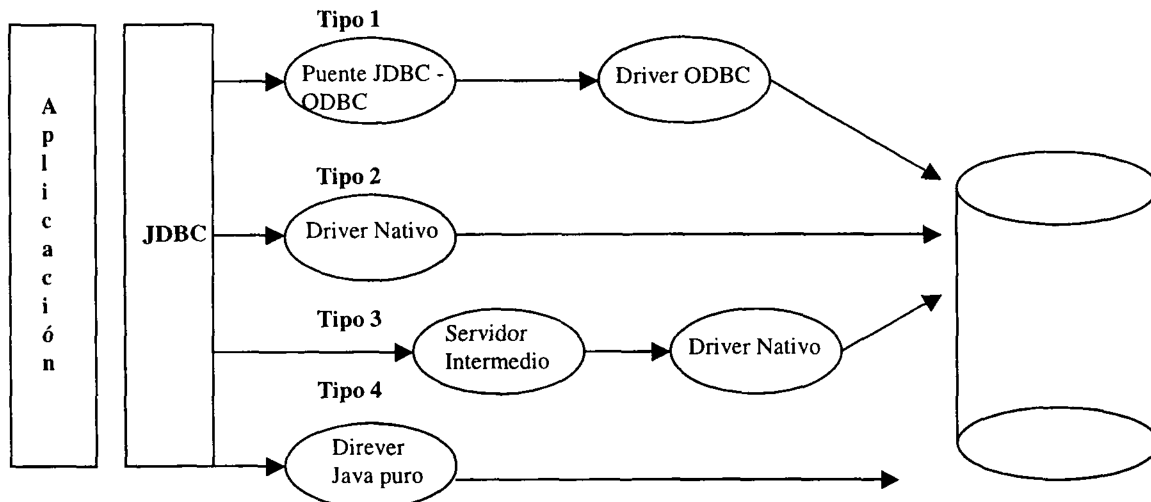

Figura 1. Tipos de drivers de JDBC.

Drivers tipo 1: Estos drivers proveen el acceso a la base de datos mediante drivers de ODBC, es decir usan a ODBC como intermediario. El driver simplemente convierte las peticiones de JDBC a ODBC. Para que este tipo de drivers funcione, el driver de ODBC debe de estar instalado en la computadora en la cual se va a usar el JDBC.

ODBC (Open Data Base Connectivity) es un estándar para acceso a base de datos propuesto por la compañía Microsoft. Es uno de los métodos de acceso a las base de datos más usados. El estándar está definido en C, sin embargo es posible encontrar drivers de ODBC casi para cualquier lenguaje de programación y casi para cualquier manejador de base de datos (es por eso que es tan popular).

Drivers tipo 2: Este tipo de drivers convierte las llamadas a un API nativo del manejador de base de datos. Estas librerías toman ventaja de las características específicas del manejador de base de datos. El código de tanto los drivers tipo 1 como tipo 2 no esta escrito completamente en Java, lo cual dificulta su portabilidad.

Drivers tipo 3: Este tipo de drivers son un tipo de drivers escritos completamente en Java, que trasladan las peticiones de JDBC al protocolo de comunicación específico de un manejador de base de datos, y posteriormente es traducido a ODBC o a un API nativo del manejador. En este caso se requiere un midleware que pase las peticiones a la base de datos.

Drivers tipo 4: Este tipo de drivers están escritos completamente en Java e invocan al protocolo de red del servidor de base de datos directamente. Un ejemplo de drivers de tipo 4 son implementaciones en Java que se comunican mediante sockets con el servidor usando el protocolo propietario de Oracle SQL*Net. Los proveedores de los sistemas manejadores de base de datos son la fuente principal de este tipo de drivers [KARA99].

JSQL es una sintaxis que permite intercalar estatutos de SQL en Java. La tecnología consiste en un preprocesador que traduce el código JSQL al lenguaje estándar de Java mediante el uso de JDBC. En la actualidad existe una propuesta por parte de Oracle, IBM y Tandem para hacer de JSQL un elemento estándar del lenguaje Java.

La manera de funcionar de JSQL es que el precompilador genera código de llamadas a JDBC, pero además checa en tiempo de compilación la sintaxis de los estatutos de SQL y los valida contra la base de datos, disminuyendo la probabilidad de errores en ejecución.

2.3 Aplicaciones n-tired (en capas)

El concepto de arquitectura en capas, consiste en dividir las funciones en niveles, los cuales proveen una serie de servicios (Figura 2). Los accesos a los servicios de estos niveles se hace a través de las interfaces que provee cada capa.

De una manera muy general, para una aplicación orientada a objetos se pueden identificar los siguiente servicios y niveles que podrían componer a la aplicación:

- Objetos de vista, que implementan la interfaz del usuario.
- Objetos de la aplicación que manipulan y conectan a objetos de negocio, para proporcionar la funcionalidad de la aplicación.
- Objetos de negocios, en los que están encapsuladas las reglas del negocio.
- Nivel de persistencia que permite que los datos relevantes para la aplicación se hagan permanentes en alguna fuente de almacenamiento (un manejador de base de datos relacionales, por ejemplo) y que provee de datos a los objetos.
- Nivel de comunicaciones y transacción que provee servicios de comunicaciones y transaccionales al servicio de persistencia [PAGA98].

Una perspectiva más simplista de particionar a la aplicación es mediante la definición de los siguientes servicios

- Presentación o interfaz de usuario: En este nivel es donde se lleva a cabo la interacción con el usuario.
- Interfaz de datos: En este nivel es donde están contenidos y manipulados los datos en memoria. El nivel de datos soporta al nivel de presentación.
- Servicio de transacciones: En este nivel es donde el proceso transaccional ocurre. Este nivel coordina todo el almacenamiento permanente de los datos. Este nivel soporta al nivel de datos y depende de una interfaz externa.
- Nivel externo: Este nivel es responsable de la comunicación entre la aplicación y los medios externos de almacenamiento, ya sean base de datos, archivos, dispositivos de hardware, etc. [BERG96] .

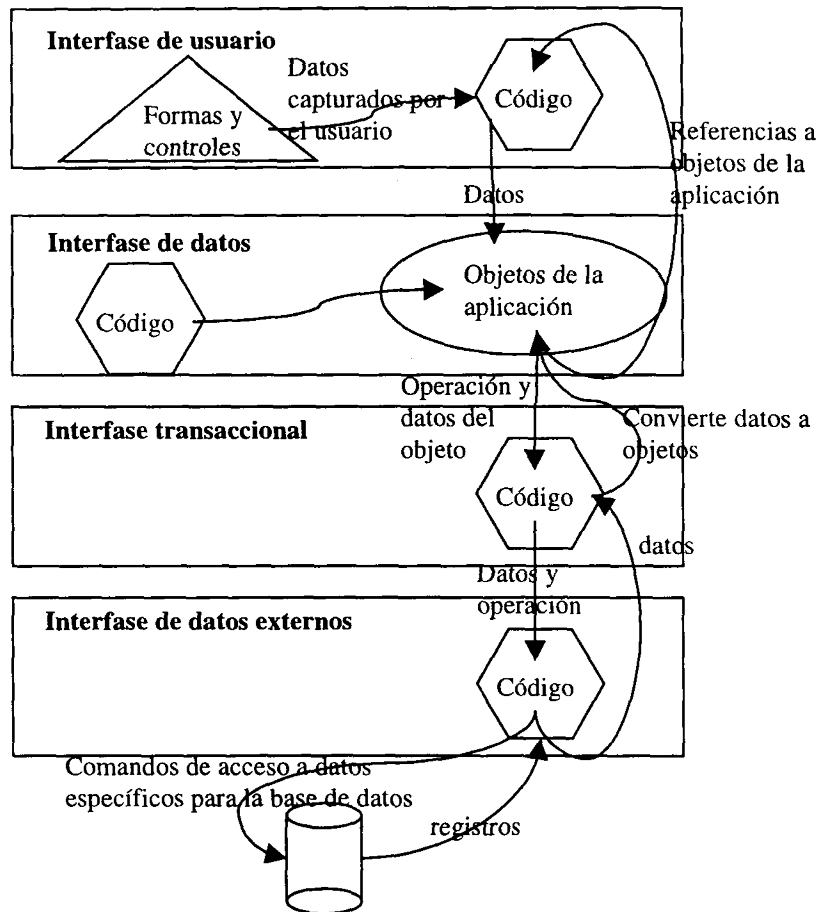

Figura 2. Esquema de una aplicación en capas.

Adicionalmente a los servicios mencionados, se pueden agregar más capas dependiendo de la complejidad de la aplicación. Una posible capa es la capa de control de acceso a la aplicación, la cual es responsable de restringir el acceso y las funciones permisibles a un usuario.

2.3.1 Esquemas para implementar aplicaciones en capas

Existen varios esquemas para el desarrollo de aplicaciones en niveles.

Una de éstas es hacer que cada componente de la aplicación implemente todos los servicios de todos los niveles requeridos. Este tipo de componentes (verticales) tiene la ventaja de que una vez construidos, son muy fáciles de utilizar, y el código necesario para accesar al componente se reduce al mínimo. Un ejemplo de este tipo de componentes es una clase (por ejemplo Alumno) en la que ya esta construida tanto la funcionalidad de presentarse en una interfaz, tiene implementadas las restricciones propias del dominio de la aplicación y las operaciones válidas de la clase (por ejemplo inscribir una materia, etc.), y además que tiene implementada la capacidad de hacer persistentes los datos en algún dispositivo. Si se tienen implementados componentes de este tipo, la construcción de las aplicaciones se podrán hacer muy rápidas y con alta calidad. Esto es debido a que los desarrolladores no tendrían que preocuparse por cuestiones técnicas de cómo conectar los diferentes niveles de la aplicación (puesto que eso ya esta hecho en el componente), y a cambio de eso los desarrolladores pueden enfocarse en satisfacer las necesidades de los usuarios [FINC98].

Sin embargo el implementar componentes de esta manera resulta una tarea difícil ya que se tienen que entender e implementar diferentes funciones dentro de un mismo componente (presentación, persistencia, etc.), para esto se requiere entender las tecnología involucradas, como base de datos, controles visuales, etc. Además debe conocerse los mecanismos de comunicación entre todos los

niveles. Otra dificultad es que un componente desarrollado de esta manera, no puede adaptarse tan fácilmente a los cambios, por ejemplo si decidimos cambiar la interfaz de un componente de modo carácter por interfaz en el WWW, tendríamos que hacer los cambios directamente sobre el componente. Otra desventaja es que si no se analiza lo que tengan en común diferentes componentes, habrá partes que estarán repetidas en varios componentes, teniendo así un pobre nivel de reuso.

La otra alternativa al desarrollar una aplicación es construir los servicios horizontales o de infraestructura y construir posteriormente la aplicación usando estos niveles previamente construidos. La desventaja de un desarrollo llevado de esta forma es que se hace énfasis en la cuestión técnica, los desarrolladores de los diferentes servicios pueden no lograr ver el panorama completo de la aplicación y por lo tanto no poder entender y satisfacer los requerimientos del usuario [FINC98]. La ventaja que presenta esta estrategia es que una vez que se tengan desarrollados los servicios generales, estos podrán ser usados en siguientes desarrollos.

Una tercera alternativa que podría ser muy conveniente es el combinar ambos enfoques y lograr desarrollar componentes verticales que implementan todas las características necesarias del componente (servicios de la aplicación, presentación, datos, persistencia, etc.) de tal manera que los desarrolladores de las aplicaciones simplemente tienen que lidiar con unas cuantas clases que ya implementan todo lo necesario a ellas, pero que a la vez la implementación de cada uno de los servicios generales (presentación, persistencia, etc.) puedan ser construidos como componentes que proveen servicios generales y que pudieran ser reusados en los componentes verticales.

2.4 Metodología

Para el desarrollo del prototipo y de los componentes se hará uso del lenguaje de modelación UML.

UML es el lenguaje estándar para modelación de aplicaciones orientadas a objetos [LARM97]. Consta de una serie de diagramas para especificar diferentes aspectos de una aplicación. Los tipos de diagramas que se usarán son los siguientes siguientes:

Modelo conceptual: Representa una vista conceptual de la aplicación o componente, en éste se incluyen todos las entidades que sean relevantes para la aplicación y sus relaciones, sin llegar al nivel de detalle.

Diagramas de secuencia: Mediante este diagrama se especifican la secuencia de pasos que se llevan a cabo entre las clases para realizar una cierta operación.

Diagramas de clase: Mediante estos diagramas se especifica el diseño detallado de una clase. En estos se incluye tanto la jerarquía de herencia, la asociación entre clases y la estructura de la misma.

Con estos diagramas es posible definir tanto la característica general del prototipo, la estructura detallada de sus clases y los pasos para realizar las operaciones que realizan los componentes que lo integran.

La implementación tanto de la aplicación de las clases de la aplicación como del componente de acceso a datos se hará en el lenguaje de programación Java versión 1.2.1, con el uso de JDBC.

Aunque el componente de acceso a datos se diseñó para el acceso a datos de varios manejadores de datos relacionales, la base de datos de la aplicación se montará en Oracle 7.2.3 y las pruebas del prototipo se harán sobre esta plataforma (usando los drivers de JDBC para Oracle).

Capítulo 3

Arquitectura para acceso a datos

3.1 Introducción

Como vimos en el capitulo anterior una arquitectura de referencia nos permitirá construir familias de sistemas con características comunes. En este capítulo se examinan las características que debe poseer una arquitectura de referencia que facilite el acceso a bases de datos, se proponen alternativas de la arquitectura y se detalla la arquitectura más favorable.

3.2 Características deseables

Las características que buscamos de una arquitectura para el acceso a base de datos son:

- Independencia del mecanismo de almacenamiento: La arquitectura debe ser la misma independientemente del mecanismo de persistencia que se decida utilizar variantes del mecanismo de almacenamiento pueden ser bases de datos de un proveedor X o Y, archivos locales, etc.
- Reuso de los componentes de la aplicación: La aplicación tratará de ser construida a partir de componentes ya existentes, en la medida que sea posible. Se debe facilitar el reuso de los componentes propios de la aplicación, como los componentes que proporcionan servicios genéricos.
- Hacer menos dependiente a la aplicación de la base de datos. Esto tiene dos finalidades, que el desarrollo de la aplicación se centre en la funcionalidad deseada y no en la comunicación con la base de datos; y hacer más adaptable a la aplicación. Se busca evitar que por cada aplicación se codifiquen manualmente los comandos de modificación de datos y de consulta sobre la base de datos, así como evitar el proceso de conversión entre las estructuras manejadas por la base de datos y las manejadas por la aplicación. Adicionalmente, si cambia la estructura de la base de datos, se buscará que cambie lo menos posible el diseño y código de la aplicación
- La arquitectura debe proveer un modelo en el que sea fácil desarrollar las aplicaciones. Si la manera de construirlas es complicada, su probabilidad de uso será reducida, y la probabilidad de errores será grande.

3.3 Alternativas

3.3.1 Componentes Verticales

Una alternativa es el uso de componentes verticales. Un componente vertical es aquel que implementa o provee servicios de varios niveles en la aplicación, tales como la interfase de usuario, el acceso a base de datos, etc. Los componentes verticales se pueden definir en términos de entradas, reglas de negocio y salidas. La ventaja de esta clase de componentes, es que es muy fácil usarlos dentro de una aplicación, puesto que éstos contienen toda la funcionalidad necesaria que requiere la aplicación [FINCH98].

Por ejemplo si se cuenta con un componente Alumno, éste podría implementar el nivel de interfase, persistencia y la lógica necesaria de las funciones que puede llevar a cabo un alumno.

Componente Vertical

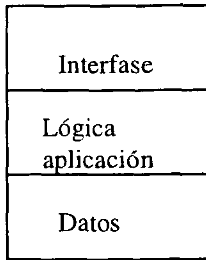

Figura 3. Estructura básica de un componente vertical.

El componente debería implementar todos los servicios que se requieran del él por parte de la aplicación, desde la presentación hasta el servicio de persistencia. Por ejemplo si cuenta con un nivel de datos, el componente implementa todas las operaciones necesarias para obtener y actualizar los datos requeridos por el componente.

Como ejemplo supongamos el componente Alumno, dentro de su nivel de datos podría tener funciones que formaran y ejecutaran las instrucciones SQL para obtener y actualizar los alumnos en la base de datos, también podría tener rutinas que asignaran las propiedades del componente Alumno de acuerdo a los registros obtenidos de la base de datos.

La motivación para construir componentes verticales es que proveen un mayor pago a los programadores, puesto que con estos componentes se necesita escribir menos código. Es más fácil componer una aplicación a partir de un conjunto pequeño de componentes grandes, que de muchos componentes pequeños [BIGG94].

Sin embargo, estos componentes presentan 3 desventajas. Cada vez que se construye un componente, deben de implementarse todas las capas de la aplicación; si ocurre un cambio en algunos de los niveles se deberá modificar todos los componentes de la aplicación. Por ejemplo supongamos que cambia el medio de almacenamiento de los datos, todos los componentes que implementen el nivel de datos, deberán ser modificados.

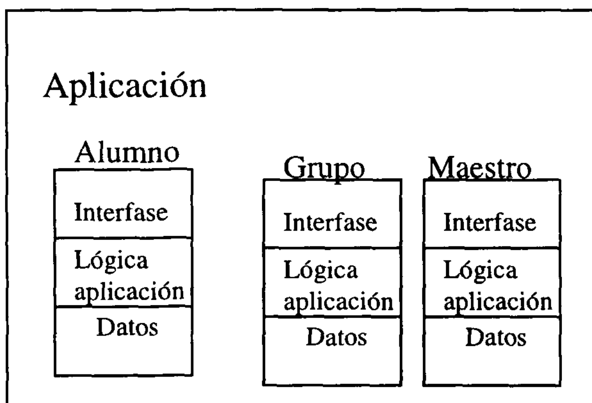

Figura 4. Diagrama de una aplicación construida a partir de componentes verticales.

Una desventaja menos evidente es que disminuye su posibilidad de reuso, puesto que un componente vertical provee una funcionalidad más específica, su probabilidad de reuso disminuye mientras éste se vuelva más especifico o se haga mas grande [BIGG94].

3.3.2 Componentes Horizontales Genéricos

Otra alternativa de estructurar la aplicación es mediante capas independientes o componentes horizontales con funciones específicas. En el caso del acceso a los datos persistentes, implica contar con un componente que realice el acceso a los datos, y que pueda ser usado por los demás componentes de la aplicación. El uso de componentes horizontales centraliza todo código para cierta funcionalidad en un solo sitio. De esta manera cualquier cambio concerniente a esa

funcionalidad afectará únicamente al componente encargado de esa función. Estos componentes pueden ser reutilizados en varias aplicaciones. Adicionalmente, los componentes de la aplicación también llegan a ser más reusables puesto que su funcionalidad es más específica (no implementan los servicios que proveen los componentes horizontales), y por lo tanto pueden ser trasladados entre aplicaciones con más facilidad.

La desventaja con el uso de estos componentes, es que su desarrollo es complejo, además de que puede desviar la atención de la aplicación en el uso de estos componentes en lugar de la funcionalidad de la aplicación (implica conocer el API de los componentes).

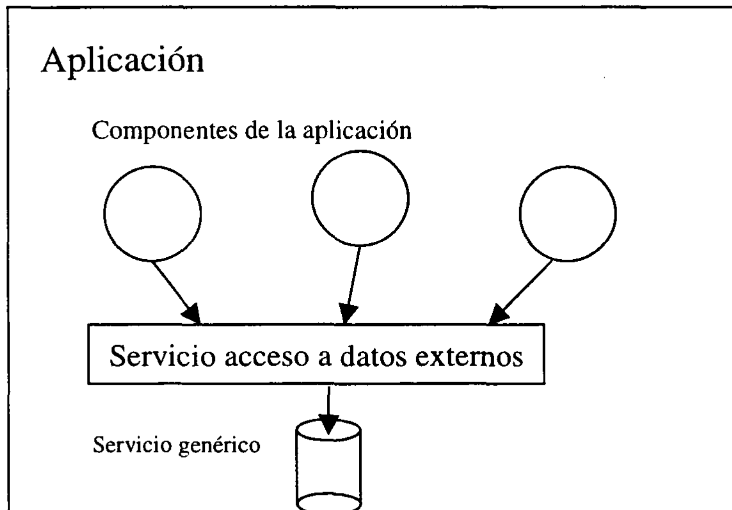

Figure 5. Diagrama de una aplicación que usa componentes horizontales.

3.3.3 Clases Auxiliares

También es posible crear un nivel de clases auxiliares que proporcionen los servicios a la aplicación. Bajo este modelo hay una clase auxiliar por cada clase de la aplicación que requiere el servicio [KURA98]. Por ejemplo si la aplicación cuenta con una clase Alumno que requiere el servicio de acceso a datos externos, se requerirá crear una clase DatosAlumno que provea estos servicios de persistencia a la clase Alumno.

De esta manera tendríamos una clase DatosAlumno, que se encargaría tanto de obtener, como de almacenar los datos de Alumno en la base de datos [KURA98]. Lo que se está haciendo es construir un nivel adicional para el acceso a datos. Tiene la ventaja que el código de acceso a datos está localizado en esta capa, y que cualquier cambio que se dé en los requerimientos de almacenamiento, sólo afectara a ésta, la capa provee una interfase estándar que facilita su uso, es decir, todas las clases auxiliares, contarán con métodos estándares para accesar la información, por ejemplo: `obten()` y `actualiza()`.

Sin embargo este enfoque tiene la desventaja que por cada nueva clase de la aplicación que se requiera, se debe construir su contraparte que se encargue de la persistencia.

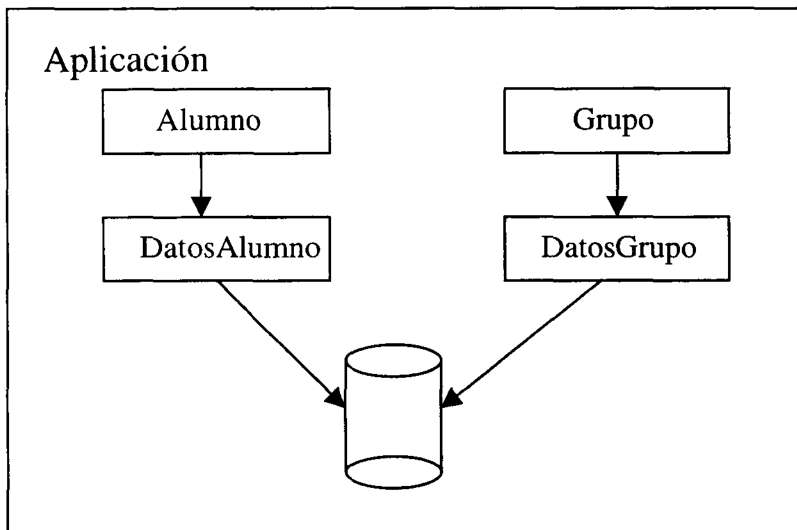

Figura 6. Diagrama de aplicación usando clases auxiliares.

3.3.4 Uso de Herencia Para Separar Niveles

La *herencia* es un mecanismo mediante el cual una entidad recibe propiedades de otra entidad [TAIV96].

Por medio de la herencia se pueden construir componentes que incorporen los servicios existentes. La manera de lograr esto es heredando directamente del nivel que se desea usar, y el nuevo componente adquirirá las propiedades o funcionalidad que el servicio genérico provee. Si una clase requiere del acceso a base de datos, podría heredar de una clase que provee los servicios necesarios, para obtener y hacer persistentes los datos de la clase en una base de datos.

Así, si requiriéramos que los objetos de la clase Alumno incorporaran la funcionalidad de actualizar sus atributos en una base de datos, heredaría de una clase que se encargara de dar estos servicios, por ejemplo `ObjetoPersistente` (Figura 7) [DUGU00]. `ObjetoPersistente` sería una clase abstracta, la cual define los métodos requeridos para la persistencia del objeto. La clase que requiriera los servicios, tendría que sobrescribir algunos métodos de la superclase para adecuar su comportamiento y adaptarlo a los requerimientos de la clase.

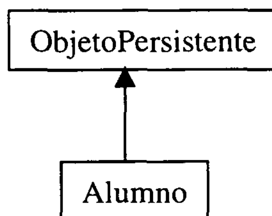

Figura 7. Diagrama de una clase separada en niveles por medio de herencia.

La ventaja con esta arquitectura es que se provee una interfase estándar para realizar las operaciones de persistencia. Además cada clase puede extender el comportamiento y adaptarlo a su necesidad, se pueden agregar validaciones antes de pasar la información a la base de datos, etc.

Su desventaja es que acopla a una clase del dominio de la aplicación a una clase que realiza funciones de "utilería", reduciendo la posibilidad de reuso de la clase en otras aplicaciones. También tiene la limitante de que en lenguajes que no soportan herencia múltiple, el hecho de hacerla heredar de una superclase, evitaría la capacidad de hacerla heredar de otra clase.

3.4 Arquitectura propuesta

La arquitectura que se propone es una arquitectura en capas, en la que se usa un componente genérico que consta de dos capas: una para el acceso a los datos y la otra para la conversión entre registros y objetos.

El objetivo de la arquitectura es mantener a las clases de la aplicación lo menos acopladas a los servicios de acceso a datos, pero aún así proveer todos los servicios requeridos a la aplicación. Este se logra definiendo una capa dedicada al *acceso a datos*. Dentro de la aplicación, todas las peticiones sobre los datos se hacen a través de esta capa. Los *servicios de conversión* entre los objetos de la aplicación y las estructuras manejadas por la base de datos se realizan por otro nivel, sin embargo para aislar la complejidad y facilitar su uso, se integran las dos capas en una sola, de tal manera que la aplicación solo ve la necesidad de comunicarse con un nivel. Esto se ilustra en la Figura 8.

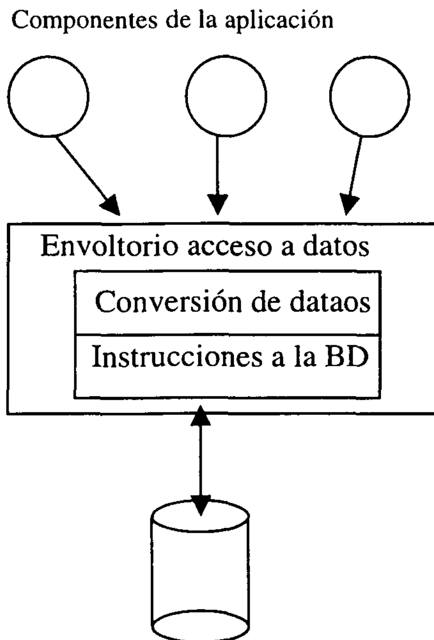

Figura 8. Diagrama de arquitectura en niveles para acceso a datos.

Los servicios que provee la capa de acceso a datos son servicios de consulta y transacciones con objetos. Todas las peticiones que se le hacen a esta capa están hechas en términos de las clases y los objetos de la aplicación y no en términos de la base de datos. Es decir, si se requiere la consulta de los objetos de una clase, ésta se especifica usando los atributos de la clase, y no los atributos de las tablas en los que está almacenada la información de una clase.

La capa de acceso a datos es responsable de las siguientes acciones:

- Conectarse a una o más bases de datos.
- Identificar la estructura de la base de datos y su relación con las clases de la aplicación.
- En caso de una consulta:

    Formar las instrucciones de SQL que obtengan la información solicitada.

    Ejecutar dichas instrucciones y obtener el resultado de la base de datos.

        Convertir los registros regresados por la base de datos a objetos de la clase de la que se especifico la consulta.

- En caso de una actualización de información:

    Crea una transacción.

Obtiene la información de los objetos que se desean actualizar.

Forma la instrucción SQL con los valores que se obtuvieron de los objetos a actualizar.

Ejecuta las instrucciones SQL.

Concluye o deshace la transacción.

Este modelo tiene las siguientes ventajas:

- Hay un esquema en el que se usan componentes genéricos, que pueden ser usados en cualquier aplicación.
- La aplicación hace uso de un componente que provee los servicios de persistencia, sin necesidad de acoplar las clases de la aplicación a este servicio. Esto hace más reusables las clases de la aplicación.
- Todas las peticiones sobre los datos se hacen en términos de las clases de la aplicación y no en términos de la estructura de la base de datos. Esta es la ventaja más grande de la arquitectura, puesto que todas las peticiones se hacen en términos de clases y objetos, el desarrollador no necesita conocer la estructura de la base de datos. Esto conlleva dos ventajas: Evita al desarrollador conocer el modelo de datos y la necesidad de codificar las instrucciones necesarias para realizar la operación.

Si hay algún cambio en la estructura de la base de datos, la aplicación no sufrirá ningún cambio, puesto que toda la interacción con la base de datos la realiza el componente que provee el servicio de acceso a datos, ya que las operaciones se hacen en términos de objetos propios de la aplicación y no de objetos de la base de datos.

Como todo el código de acceso a datos se encuentra localizado en una sola capa y su acceso es dinámico, dentro de esta capa se puede esconder el acceso a varias bases de datos, y se pueden ir agregando características a esta capa, como optimización de querys, algoritmos de obtención de datos. Incorporar funcionalidad, como un log de los accesos a datos, etc., todo esto sin que la aplicación se tenga que preocupar por estos aspectos.

Puesto que todas las instrucciones de la base de datos y la construcción de los objetos se hace dinámicamente, la desventaja más grande que posee es la inhabilidad de analizar en tiempo de compilación las operaciones. En dado caso de que ocurriera un error, éste se haría notar hasta la ejecución. El hecho de que las instrucciones se generen dinámicamente ocasiona que el costo en el desempeño sea alto. El tiempo que consume una operación de consulta o actualización de objetos es igual a la suma del tiempo de generar las instrucciones ejecutar y convertir los datos de las estructuras manejadas por la base de datos a instancias de objetos manejados por la aplicación.

Capítulo 4

Framework para acceso a datos.

4.1 Introducción

La arquitectura de la aplicación contempla una capa que proporciona los servicios de acceso a datos. En este capítulo se presenta una descripción detallada de los servicios que proporciona esta capa y su diseño.

4.2 Frameworks

Un *framework* es un conjunto de clases que cubren un diseño abstracto de una solución para una familia de problemas relacionados. Consta de una serie de bloques de construcción preconstruidos que los programadores pueden usar, extender o adaptar para un problema específico, pero que define el comportamiento básico de una colección de objetos [TALI93].

Un framework de persistencia es un conjunto de clases reusables, que proveen servicios para lograr la persistencia de objetos. El framework puede ser extendido, de tal manera que se agregue funcionalidad o se adecue a los requerimientos específicos de una aplicación, tales como la comunicación con un RDBMS en específico [LARM98].

La estrategia de acceso a datos es un framework de persistencia. Provee el comportamiento básico y la implementación casi completa para el acceso a base de datos. El framework puede ser extendido para funcionar con la base de datos que se desee, proporcionando las clases necesarias para la comunicación con el medio de persistencia. De esta manera se puede cambiar transparentemente el producto en el que se almacenan los datos, y aún así el framework puede ser usado.

Los frameworks se pueden clasificar como framework de caja blanca o framework de caja negra. En un framework de *caja blanca* se agregan subclases al framework en las que se pueden agregar los métodos que el framework provee. En este esquema se debe conocer el funcionamiento interno del framework para poderlo usar. Un framework de *caja negra* es un framework en el que el comportamiento especifico se hace agregando componentes que cumplan con un protocolo especificado por el framework [JOHN98].

Los frameworks de caja negra son más fáciles de usar que los frameworks de caja blanca puesto que no se requiere conocer su implementación interna, sino solo los protocolos que especifica el framework [D'souza 98].

4.3 Especificaciones funcionales del framework

4.3.1 Descripción General

El framework de acceso a datos consiste en un mecanismo en el que se especifican las actualizaciones y consultas sobre objetos de la aplicación. La aplicación especifica al framework una operación y los objetos con los que se llevará a cabo la operación. El framework obtiene metadatos de la base de datos, las clases de los objetos sobre los que se pide la operación y la relación que hay entre clases y base de datos. En base a esta información, construye dinámicamente los comandos SQL que ejecutará la base de datos y ejecuta los comandos sobre ésta. En caso de que la operación haya sido una consulta, convierte de los registros obtenidos por la base de datos a objetos de la aplicación y regresa los objetos a la aplicación. En el caso de una actualización de un objeto, obtiene los valores que los atributos de éste para formar los estatutos SQL que actualizarán sus valores en la base de datos.

4.3.2 Objetivo del framework

Los objetivos del framework para proporcionar los servicios de acceso a datos son:

- **Proveer servicios de consulta y transaccionales:** El objetivo del framework es proveer servicios de consulta y servicios transaccionales sobre bases de datos.

  Las operaciones de consulta incluyen obtener objetos y los objetos con los que está relacionado, así como la creación dinámica de los objetos con los datos obtenidos.

  Las operaciones transaccionales constan en actualizar los datos de un objeto en la base de datos y opcionalmente de los objetos con los que está relacionado.
- **Aislar de la base de datos:** El framework provee herramientas para especificar las operaciones en términos de objetos y de clases de la aplicación. El acceso a la base de datos directo no está permitido, para garantizar la independencia de ésta.
- **Fácil de usar:** Para que el framework tenga una ventaja real sobre el acceso directo a la base de datos para las consultas, su utilización debe ser más fácil que crear directamente un canal de comunicación con la base de datos y especificar directamente los comandos SQL. El framework provee un conjunto de instrucciones muy reducido, para facilitar el aprendizaje y uso de este. Puesto que es un framework de caja negra, no se requiere conocer los detalles de implementación.
- **Extensible a varios métodos de almacenamiento:** Para que el framework pueda proveer de independencia a la aplicación sobre la base de datos y pueda ser usado en diferentes aplicaciones, es necesario que el framework pueda accesar cualquier base de datos o fuente de almacenamiento de datos. Para eso, se provee un mecanismo en el que se puedan incorporar clases que representen a la base de datos y que provean la comunicación con un manejador de base de datos en particular. Estos componentes deben cumplir con un protocolo, de tal forma que ni el framework, ni la aplicación se vean afectados si ocurre un cambio en el medio de almacenamiento.

4.3.3 Servicios que provee

Conexión a múltiples base de datos: El framework permite establecer conexiones a varias bases de datos simultáneamente. Esto permite que los datos que contiene una clase pueden estar localizados en bases de datos distintas. Al intentar consultar o actualizar los datos de un objeto, se forman los comandos SQL para cada base de datos donde esten contenidos los datos y se ejecutan en las bases de datos necesarias (Figura 9). Si es necesario, posteriormente se forma una consolidación de los datos obtenidos, de tal forma que el conjunto de operaciones se comporte como si hubiera sido una sola operación.

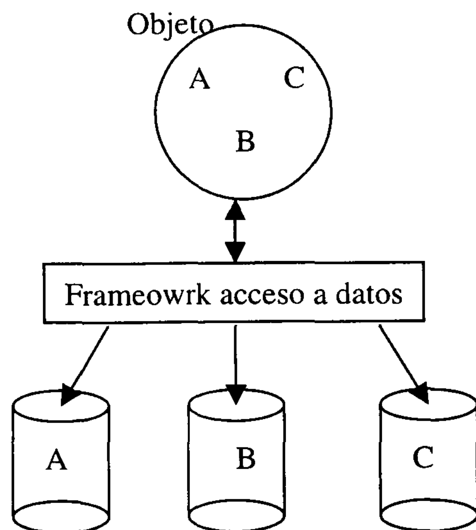

Figura 9. Diagrama de acceso a múltiples bases de datos a través del framework.

Consulta de objetos en base a criterios especificados en base a atributos de la clase a la que pertenece el objeto: El framework cuenta con un mecanismo para especificar las consultas en términos de los atributos de una clase. El framework proporciona clases para representar operadores boleanos como AND, OR, NOT y mecanismos para hacer querys sobre las relaciones entre los objetos.

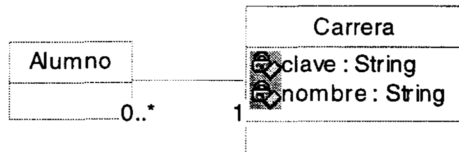

Figura 10. Diagrama básico de la relación entre Alumno y Carrera.

En el diagrama de clases anterior tenemos dos clases que están relacionadas entre sí. El diagrama representa la relación entre un Alumno y la Carrera que cursa. Una posible consulta sería obtener todos los alumnos cuya carrera sea 'ISC'. La consulta podría leerse de la siguiente forma. Obtener todos los objetos de la clase Alumno para los cuales el objeto de la clase Carrera con el que están relacionados tenga como su atributo clave el valor 'ISC'. Para construir esta consulta requerimos especificar un criterio usando el nombre y los atributos de las clases sobre las que se requiere la consulta, y no el nombre de las tablas o campos que contienen los datos.

Construcción dinámica de objetos: Los resultados de las consultas se entregan a la aplicación en forma de objetos de la clase sobre la que se especifica una consulta. Esto involucra un proceso de traducción de los datos tal y como los entrega la base de datos a objetos. Para esto se requiere tener equivalencias entre los atributos y relaciones de las clases y las tablas y campos de la base de datos. Una vez que se realiza la consulta y se obtiene el resultado, se crean los objetos y se inicializan con los valores obtenidos de las base de datos.

Consulta de objetos: Este constituye uno de los dos servicios principales que provee el framework. El servicio consiste en especificar un criterio de consulta, basado en los atributos y relaciones de las clases, y como resultado obtener un conjunto de objetos de la clase sobre la que se hizo la consulta que cumplen con el criterio que se especificó.

Una clase puede estar relacionada con 0 o más clases. Si la clase tiene relaciones con otras clases, la consulta abre dos posibilidades, obtener los datos que pertenecen únicamente a las clase sobre la que se especifica la consulta u obtener inclusive los objetos de las clases con las que esta relacionado el objeto de la clase sobre la cual se especificó la consulta.

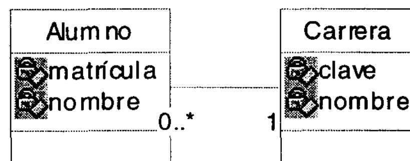

Figura 11. Diagrama de clases de la relación Alumno y Carrera.

Supongamos que queremos obtener al alumno con una matricula dada. La consulta puede consistir en obtener únicamente los datos que contiene la clase Alumno, en este caso la matrícula y nombre. Otra opción es obtener los objetos con los cuales estaría relacionados el objeto, que en este caso es la carrera a la que pertenece el alumno. En este último caso se obtendrían los datos de la carrera, se crea un objeto de la clase Carrera y se agrega al alumno la referencia al objeto carrera.

Actualización de objetos: La operación consiste en actualizar sobre la base de datos los valores de los atributos de el objeto que se está actualizando y opcionalmente los valores que representan la relación de el objeto con otros. Al igual que el caso anterior en el que es posible obtener y crear los objetos que forman la relación con un objeto de la clase sobre la cual se especifica una consulta, también es posible actualizar en la base de datos la información de los objetos que están relacionados con el objeto que se desea actualizar.

Supongamos que en el ejemplo anterior deseamos actualizar a un objeto de la clase Alumno el cual tiene está relacionado con una carrera. Hay 3 opciones a seguir al actualizar el objeto. Guardar únicamente los datos de la clase alumno: matrícula y nombre; la segunda opción es actualizar los datos del alumno y la relación entre el alumno y la carrera; la tercera opción es actualizar tanto los datos de el alumno, la relación entre el alumno y la carrera y la información de la carrera. Este último caso puede ser necesario cuando no existe en la base de datos información sobre la carrera con la que está relacionada el objeto.

Traducción entre elementos de la base de datos y elementos de la aplicación: El framework es responsable de presentar toda la información a la aplicación en términos entendibles para esta. Es decir a través de los objetos de la aplicación. Si se desea la consulta de un objeto, el framework regresará los resultados como objetos de la clase sobre la que se especifico la consulta.

Similarmente si se desea actualizar un objeto, solo se necesita especificar el objeto que se desea actualizar y el framework es responsable de obtener los datos de éste, formar los estatutos SQL para realizar la actualización en la base de datos y ejecutarlos.

La aplicación no necesita porque tener conocimiento de los comandos que estuvieron involucradas, ni el mecanismo en el que la base de datos regresa la información, de estos detalles se encarga el framework.

Transacciones entre objetos: Puesto que las operaciones en la aplicación siempre se hacen entre objetos, el framework provee un mecanismo con el cual se pueden especificar transacciones para la actualización de varios objetos, e inclusive la capacidad de manejar subtransacciones.

Este mecanismo permite realizar actualizaciones sobre objetos de una manera atómica. De esta manera si en algún momento falla la actualización de un objeto, la transacción se deshace y las bases de datos regresan al estado en que se encontraban antes de intentar ejecutar la transacción.

Con los servicios anteriores se puede realizar cualquier consulta o actividad transaccional que requiera la aplicación. Sin embargo hay una serie de servicios adicionales que pueden ayudar con el desempeño de la aplicación o en el desarrollo de la misma. Estos servicios incluyen:

Registro de log de operaciones: Una operación común en las aplicaciones es el registro de log de eventos o actividad. Este mecanismo permite detectar y solucionar problemas, monitorear el uso de la aplicación, etc. El servicio de log de operaciones, consiste en registrar las operaciones que realiza el framework, incluyendo el timestamp, la descripción de la operación y el usuario que realiza la operación. Un ejemplo de un registro en el log sería el mostrado en la Tabla 1.

Operación	**Detalle**	**TimeStamp**	**Usuario**	**Status**
Consulta	Alumno-Matricula = "581663"	03/05/2000 11:25am	amolina	Exitoso
Actualización	Alumno-Matricula = "581663" Datos = {...}	03/05/2000 11:26am	amolina	Exitoso

Tabla 1. Ejemplo de formato de un registro de log.

Cache de objetos: Una vez que se hace una consulta sobre un objeto, el framework debe construir los querys para obtener la información del objeto, ejecutar los querys sobre las bases de datos y construir el objeto con los datos obtenidos. Si se hace dos veces una consulta sobre el mismo objeto, se realizarán dos veces las operaciones anteriores. Si se mantiene un cache de objetos consultados, la segunda vez que se requiera consultar un objeto, el objeto simplemente se obtiene del cache y se pasa a la aplicación. Esto resulta en un ahorro del tiempo en el que se le entregan los datos a la aplicación.

Otra ventaja del uso de cache es que se puede evitar actualizar la información de un objeto que no ha sido modificado. En el momento en que un objeto modifica sus datos, es marcado como "sucio" en el cache de objetos. Al momento de actualizar un objeto en la base de datos, este sólo se actualiza si esta marcado como sucio [LARM97].

Manejo de cursor de objetos: Una consulta puede resultar en miles de objetos que deberán ser entregados a la aplicación. Supongamos que requerimos obtener todos los Alumnos que están inscritos. Si la cantidad de alumnos es grande y se regresara toda la información en un solo momento, el costo de recursos sería muy alto y probablemente excedería los recursos de la computadora. Una solución a este problema es el uso de cursores. Un *cursor* es un mecanismo mediante en cual se puede accesar uno o más objetos en un momento dado usando un mecanismo controlado [AMBL98b]. Por ejemplo podríamos escoger traer los primeros 10 alumnos y trabajar con ellos, posteriormente los siguientes 10 y así sucesivamente.

Manejo de registros: A pesar de que se busca independizar a la aplicación de las bases de datos, es necesario que el framework pueda regresar registros en una consulta. Esto se debe a que muchos de los productos que hay en el mercado, como reporteadores trabajan con registros y no con objetos [AMBL98b].

Prioridades de diseño

Las prioridades del diseño del framework es proveer un mecanismo fácil de usar que permita almacenar y obtener objetos de una o varias bases de datos sin importar el formato o la estructura en la que se encuentren la información en ella, permitir la extensibilidad del framework, que sea fácil de mantener y por último con un buen desempeño. Es decir, lo más importante es: que funcione, sea fácil de usar, el código sea fácil de mantener y extender, y por último que el desempeño sea bueno.

4.4 Diseño General

El framework para el acceso a datos cuenta con los siguientes componentes básicos.

- **Broker:** Es el componente con el cual se comunica la aplicación. Tiene los métodos para ejecutar todas las funciones del framework. Es el responsable de mantener las conexiones con las bases de datos, ejecutar las operaciones y regresar los datos a la aplicación. Funciona como una envoltura o *(wrapper)* de todo el framework, está diseñado para que sea fácil de usar y provea un punto integrador de todos los servicios del framework.
- **BaseDatos:** Es un conjunto de interfases y componentes que implementan la comunicación con una base de datos. Dentro de estos componentes se cargan los drivers para una base de datos en específico, se obtienen los metadatos de esta y se ejecutan comandos de SQL. La razón por la que se construye un mecanismo adicional a los que provee el JDBC es para aislar los detalles particulares entre diferentes bases de datos.
- **Operación:** Son un conjunto de operaciones que son ejecutadas dentro del framework. Estas operaciones incluyen consulta de objetos, actualización de estos en las bases de datos y transacciones. Las operaciones están divididas en operaciones de consulta y operaciones de actualización,. Esta clasificación permite proveer una interfase estándar para cualquier operación de consulta o de actualización, lo cual facilita agregar nuevas operaciones al framework
- **Criterio:** Son un conjunto de clases que permiten especificar criterios de consulta y actualización en términos de los atributos y relaciones de las clases. Proveen métodos para unir lógicamente varios criterios usando operadores boleanos.
- **Mapa de clases:** Forman el conjunto de clases que provee información sobre como está relacionada una clase con el modelo relacional. Esta información incluye, que conjunto de tablas y columnas corresponde a una tributo de una clase, en que tablas y atributos está representada una relación entre clase. La manera en la que se deben de crear identificadores para un objeto, etc.
- **SQL:** Son un conjunto de clases que se encarga de construir los estatutos de SQL para satisfacer una operación. Existe una clase para cada comando SQL que soporta el framework, de esta manera, si se desea agregar soporte para un nuevo comando, basta con agregar la clase que lo construya. Los comandos básicos que se incluyen son: `Select`, `Insert` y `Update`. Estas clases se basan en los mapas de clases para decidir de que tablas y columnas se requerirán para obtener o manipular la información.

- **Fabrica de Objetos:** Es un componente que permite accesar las propiedades de un objeto. Su función es obtener los valores de los atributos de un objeto, crear dinámicamente un objeto de una determinada clase y asignarle valor a los atributos de un objeto [AMBL98c].

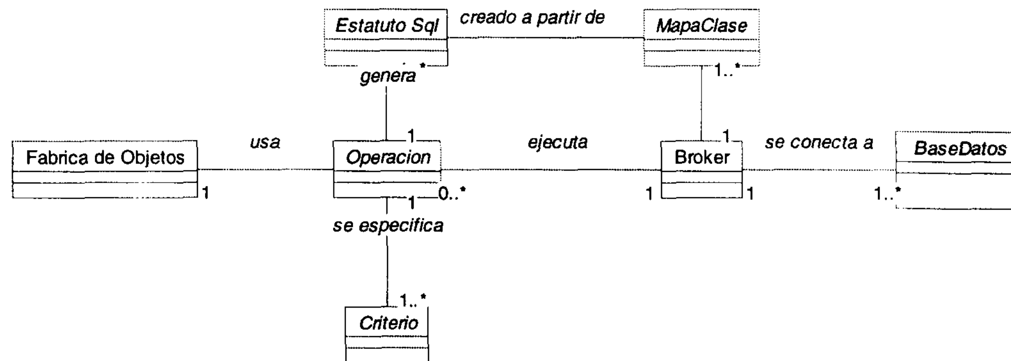

Figura 12. Diagrama general del framework de acceso a datos.

El diagrama de la Figura 12 muestra la estructura general interna del framework. La Figura 13 muestra la secuencia de acciones realizadas por el framework ante la petición de consultar un objeto de la base de datos y la actualización de un objeto.

4.4.1 Consulta de Objetos

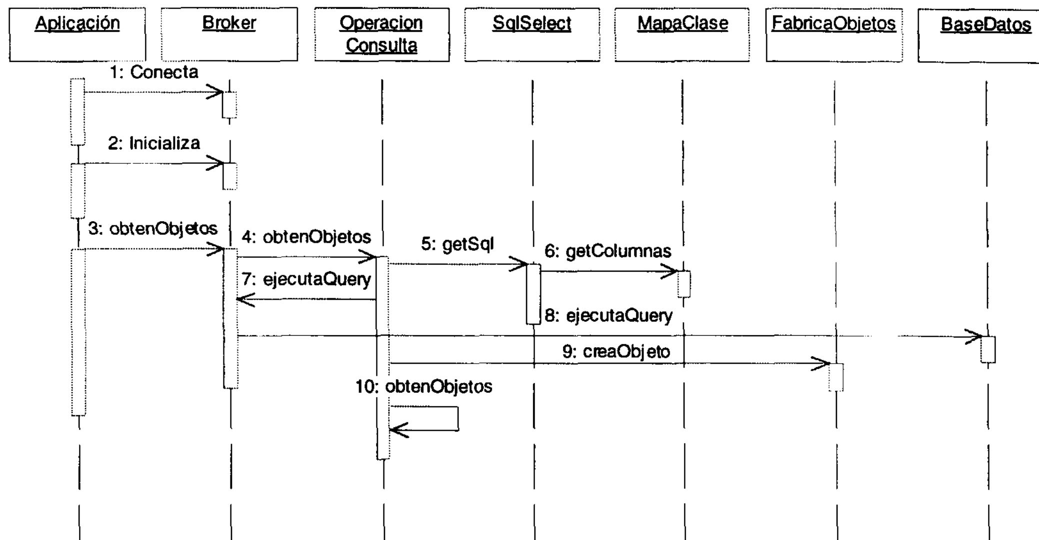

Figura 13. Diagrama de secuencias de la operación de consulta.

El proceso para obtener un conjunto de objetos de la base de datos es como sigue:

La aplicación establece una conexión a la base de datos por medio del `Broker` e inicializa el framework para cargar su configuración. En este paso se cargan los metadatos de la base de datos y las manera de mapear las clases con el modelo de la base de datos.

La aplicación especifica el criterio de los objetos que se quieren obtener y los pide al `Broker`.

El `Broker` crea una operación de consulta con el criterio especificado y la ejecuta. Como resultado la `OperacionConsulta` regresa al Broker un conjunto de objetos de la clase

especificada en el criterio, con los valores que se obtuvieron de la base de datos. El Broker a su vez regresa estos valores a la aplicación.

Para obtener los objetos, la clase OperacionConsulta construye una secuencia de querys para obtener los datos necesarios de la base de datos, ejecuta los querys a través del Broker y por último utiliza a la Fabrica de Objetos para crear los objetos consultados y asignar sus atributos con los datos obtenidos por los querys. Finalmente OperacionConsulta regresa el conjunto de objetos obtenidos al Broker y este a su vez los propaga hacia la aplicación.

4.4.2 Actualización de un objeto

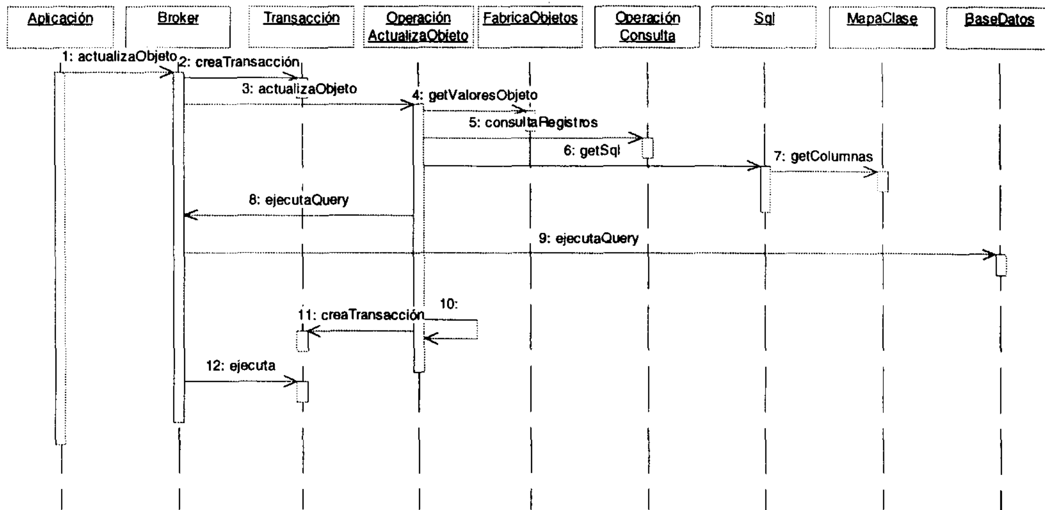

Figura 14. Diagrama de secuencia de actualización de un objeto.

El proceso para actualizar un objeto es como sigue (ver Figura 14):

La aplicación pide al Broker que actualice un objeto dado. La aplicación especifica si el broker debe actualizar nada mas los atributos del objeto, las relaciones con los demás objetos, o inclusive debe actualizar los objetos con los que está relacionado.

Broker crea una Transacción y una Operación de Actualización con el objeto a actualizar, posteriormente agrega la operación a la transacción. Si se decidiera actualizar varios objetos dentro de una sola transacción, bastará con agregar las operaciones a la transacción.

Para actualizar un objeto primero se analiza a partir de los metadatos todas las tablas en las que se debe actualizar el objeto y se priorizan de tal manera que no se violen las restricciones de integridad de la base de datos.

Para cada tabla en la que se necesite actualizar información del objeto primero se crea una OperaciónConsulta con el fin de determinar si ya existen datos del objeto en la tabla sobre la que se va a actualizar la información (esto es necesario para determinar que estatuto SQL debe formarse). Si no existe información del objeto en la tabla se creará un comando INSERT, mientras que si ya existen los registros del objeto en esta tabla se deberá crear y ejecutar un comando UPDATE.

Una vez que se decide qué comando de SQL debe crearse, se crea el objeto de la clase correspondiente (SqlInsert o SqlUpdate) y se forma el estatuto con los valores del objeto, (los cuales son obtenidos de la Fabrica de Objetos) y se agrega la operación a la transacción, de tal manera que la actualización de los datos de un objeto en varias tablas se comporta de manera atómica.

Por último se ejecutan el comando sobre la base de datos, por medio del broker.

Si la aplicación especifico que debían actualizarse las relaciones del objeto, primero se borran los registros de la base de datos que representan las relaciones y posteriormente se actualizan sobre la base de datos. Esto se debe a que si un objeto pierde la relación con otro objeto, esta pérdida de la relación debe de reflejarse en la base de datos. La única manera de garantizar esto es borrando todas las relaciones del objeto de la base de datos y luego actualizarlas.

Por último se ejecuta la transacción con todas las operaciones que se crearon.

4.4.3 Actualización de objetos

El proceso para actualizar varios objetos a las vez, consiste en crear un criterio que represente al conjunto de objetos, realizar la consulta de objetos bajo este criterio a través del `Broker`, y para cada objeto obtenido, realizar las modificaciones sobre éste y posteriormente se actualiza de manera individual. Es decir se usan los servicios de consulta de objetos, y actualización de un objeto.

4.4.4 Borrado de un objeto

Para borrar los valores que representan a un objeto en la base de datos primero. Se pasa el objeto a borrar al `Broker`, este crea el criterio que identifica al objeto, crea una transacción y agrega una `Operación de Actualización` a ésta. La `Operación` en base al criterio obtiene todas las tablas en las que está representado el objeto sobre la base de datos, y para cada tabla en la que contenga información, crea un objeto que representa el estatuto `DELETE` de SQL, forma el comando adecuado y lo ejecuta en la base de datos.

4.5 Detalle de las clases

4.5.1 Broker

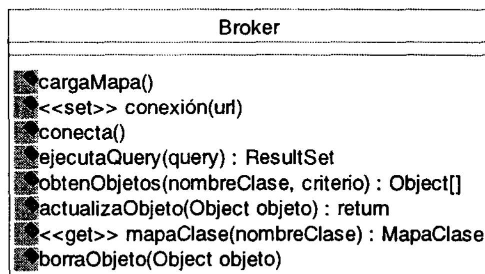

Figura 15. Diagrama de la clase Broker.

La clase `Broker` (Figura 15) es la clase mediante la cual la aplicación accesa todos los servicios del framework. La parte visible de la clase, es muy sencilla, sólo contiene los métodos hacia los servicios de conexión, inicialización del mapa entre clases y base de datos, consulta de objetos y actualización de objetos y borrado de objetos.

El método `cargaMapa`, inicializa el mapa de relaciones entre las clases y la base de datos. Este mapa es especificado por el usuario mediante un archivo de configuración.

El método `setConexión` especifica las propiedades sobre las que se conectará a una base de datos y conecta efectúa la conexión.

El método `ejecutaQuery` ejecuta un comando SQL en la base de datos y regresa un `ResultSet` en caso de que el comando haya obtenido registros de la base de datos. Este método es útil si el desarrollador de la aplicación desea ejecutar un query directamente sobre la base de datos y no usar los servicios del framework.

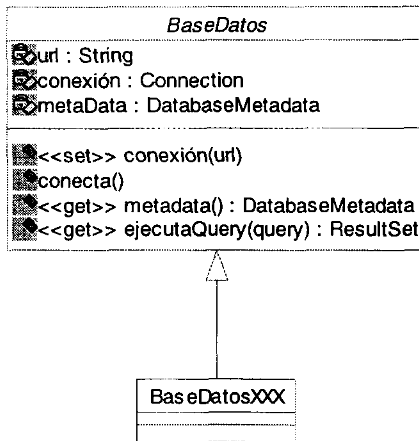

Figura 16. Diagrama de la jerarquía de clases BaseDatos.

Las tres operaciones del framework son:

- `ObtenObjetos`, que en base a un criterio especificado por el usuario, regresa un conjunto de objetos (arreglo de objetos) que cumplan con ese criterio en la base de datos.
- `ActualizaObjeto`: Actualiza los datos del objeto en la base de datos.
- `BorraObjeto`: Borra los datos del objeto de la base de datos.

4.5.2 BaseDatos

La clase `BaseDatos` es una clase abstracta que funciona como interfase para especificar un contrato que deben cumplir las clases que implementen la conexión a una base de datos en especifico. Los mecanismos y drivers necesarios para conectarse y obtener los metadatos entre distintas bases de datos es diferente. La clase `BaseDatosXXX` (donde XXX es una base de datos en particular, por ejemplo `BaseDatosOracle`, `BaseDatosInformix`, etc.), es responsable de proveer estos mecanismos. Puesto que la clase cumple con el contrato de `BaseDatos`, estas pueden ser usadas indistintamente por el framework.

Los métodos que conforman el contrato son:

- `setConexión`: especifica los parámetros de conexión a la base de datos.
- `conecta`: realiza la conexión a la base de datos
- `getMetaData`: Obtiene los metadatos de la base de datos
- `ejecutaQuery`: Ejecuta un query sobre la base de datos.

4.5.3 Criterios

El conjunto de clases que representan un criterio son usadas por el programador para especificar criterios de consulta. Por medio de estas clases se expresan las consultas en base a los atributos y valores de las clases. Por medio de los métodos que proveen estas clases el programador puede anidar varios criterios a través de operadores boleanos.

La jerarquía de clase de `Criterio` (Figura 17) además del `Broker` son las únicas clases del framework a las que tiene acceso la aplicación. Las clases `CriterioXXX` permiten al desarrollador especificar los criterios sobre los que se aplicarán las operaciones sobre los objetos en la base de datos.

La clase abstracta `Criterio`, provee una estructura y métodos para especificar composiciones lógicas entre criterios.

Los métodos `getCriterioX()` obtienen uno de los dos criterios que están relacionados por un operador lógico.

Los métodos `addAnd` y `addOr`, permiten crear criterios compuestos por varios criterios usando los operadores lógicos And y Or respectivamente.

En esta jerarquía el `CriterioSimple` permite especificar un criterio basado en el valor de un atributo de una clase.

El `CriterioRelación` permite especificar un criterio basado en la relación de dos clases.

La clase `CriterioAnd` representa el AND lógico entre dos criterios

La clase `CriterioOr` representa el OR lógico entre dos criterios

Por último `CriterioNot` representa el NOT lógico de un criterio.

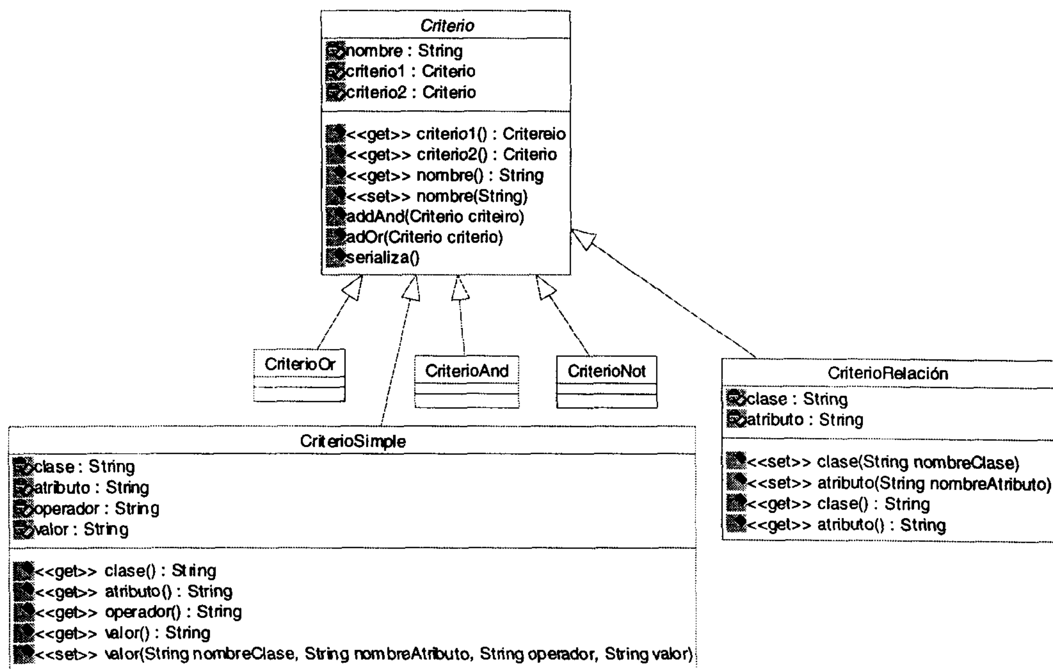

Figura 17. Diagrama de clases de la jerarquía Criterio.

4.5.4 Operaciones

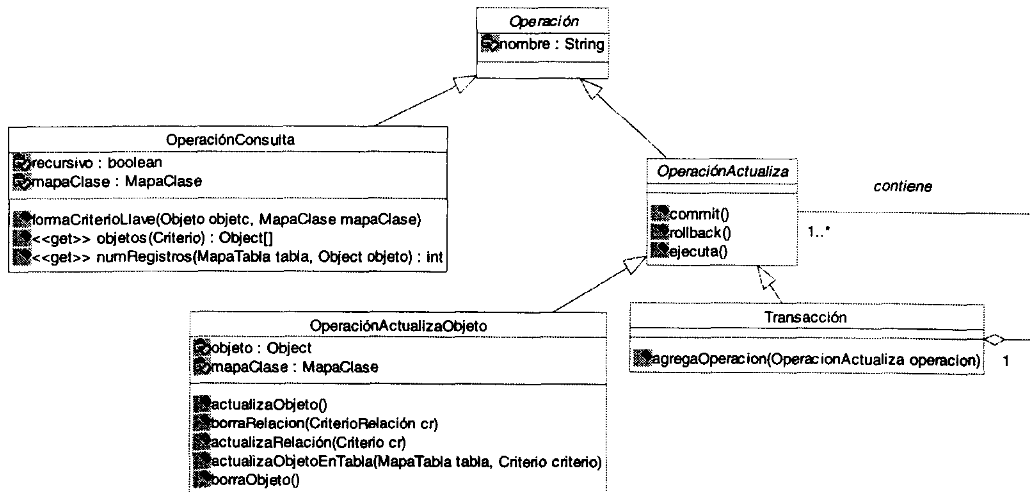

Figura 18. Diagrama de clases de la jerarquía Operación.

Por esta jerarquía de clases se realizan internamente los servicios que ofrece el framework. Dentro de éstas se encuentra la lógica para ejecutar los servicios.

Existen dos tipos de operaciones: operaciones de consulta de objetos y operaciones de actualización de objetos. Las operación de actualización proveen los servicios transaccionales sobre los objetos.

La clase `OperacionConsulta` provee métodos para las realizar las consultas que puede requerir tanto la aplicación como las demás clases del framework en la base de datos. La clase se apoya en los servicios que provee la jerarquía de clases SQL en las clases que provee la jerarquía del mapa de clases, y en la clase `FabricaObjetos`.

El método `obtenObjetos` obtiene los objetos que cumplen con un criterio dado (especificado por medio de las clases `CriterioXXX`).

El método `numRegistros` obtiene el número de registros en una tabla que tiene un criterio. Esta operación es usada por la `OperaciónActualizaObjeto`, para decidir si el estatuto SQL para actualizar los datos de un objeto debe ser INSERT o UPDATE.

El método `formaCriterioLlave` forma un `Criterio` que identifica únicamente un objeto a partir de un mapa de la clase y un objeto. Este criterio es usado para la operación de consulta de objetos, como para la actualización de objetos.

Los servicios transaccionales se componen de la operación de actualizar un objeto sobre la base de datos, borrar un objeto de la base de datos y el uso de transacciones en las dos operaciones anteriores.

La clase `OperacionActualiza` es una clase abstracta que define los métodos `Commit`, `rollback` y `ejecuta`, los cuales deberán ser implementados por sus subclases.

Se proporcionan dos clases bases:

`OperaciónActualizaObjeto`: Contiene los métodos para actualizar o borrar un objeto de la base de datos. Utiliza los servicios de la jerarquía de clases SQL, `OperacionConsulta`, `MapaClase` y `FabricaObjetos`.

El método `actualizaObjeto` actualiza un objeto en la base de datos.

El método `borraObjeto` borra un objeto de la base de datos

El método `borraRelacion` borra de la base de datos los registros necesarios para que la relación entre dos objetos no quede representada.

El método `actualizaRelacion` representa en la base de datos la relación entre dos objetos.

La clase `Transacción` solo agrega un método a la jerarquía.

El método `agregaOperacion`, permite agregar una operación de actualización a Transacción, puesto que una transacción es una operación de actualización, esta puede contener tanto operaciones de actualización de objetos como otras transacciones. Esto permite manejar transacciones dentro de transacciones.

4.5.5 FabricaObjeto

Esta clase funge como traductor entre los registros que se obtienen de la base de datos y los objetos que el framework regresa a la aplicación en el caso de una consulta. Básicamente sirve para manipular propiedades de objetos que sólo son conocidos en tiempos de ejecución.

Sus funciones principales son crear objetos de una cierta clase, asignarle valores a los atributos de un objeto, obtener los valores y tipo de datos de un atributo de un objeto (Figura 19).

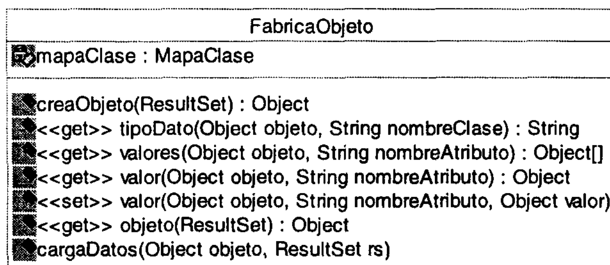

Figura 19. Diagrama de la clase FabricaObjeto.

El método `creaObjeto`, crea un objeto de la clase especificada y asigna los valores contenidos en un `ResultSet` (conjunto de renglones obtenidos mediante una consulta) a los atributos del objeto. Esta operación es usada después de que se ejecuto un query en la base de datos y los datos que se obtuvieron deben de ser convertidos en objetos para regresarlos a la aplicación.

El método `getTipoDato` obtiene el tipo de dato de un atributo en una clase. Esta información es necesaria en el momento de crear el objeto y para realizar conversiones entre los tipos de dato manejados dentro de la clase y dentro de la base de datos.

Los métodos `getValor` y `getValores` obtienen el valor o valores que tiene el atributo de un objeto

El método `setValor` asigna un valor al atributo de un objeto. Este método es usado en el proceso de crear un objeto a partir de un `ResultSet`.

4.5.6 Sql

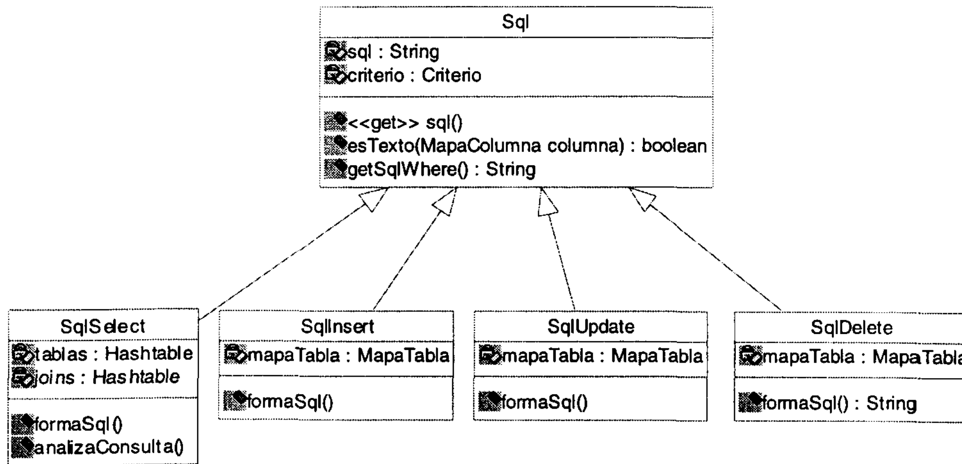

Figura 20. Diagrama de clases de la jerarquía Sql.

Este conjunto de clases (Figura 20) forma un motor para generar estatutos `Sql` a partir de un `Criterio`. Se cuenta con una clase para cada comando que se implementa. En este caso `SELECT`, `INSERT`, `UPDATE` y `DELETE`. Cada clase es responsable de generar un comando SQL válido.

La clase `Sql` implementa 3 métodos comunes para todas sus subclases, estos son:

`getSql`, el cual regresa un string con el comando Sql formado

`esTexto`, el cual determina si una columna es de tipo texto, esto se usa para que dentro de los estatutos generados de SQL, los valores de columnas que son texto sean puestos entre comillas.

`getSqlWhere` obtiene la cláusula del Where en base a un criterio. Este método se coloca dentro de la clase Sql, puesto tanto los estatutos de `SELECT`, `UPDATE` y `DELETE` necesitan una cláusula de Where.

Todas las clases implementan el método `formaSql`, que forma el estatuto SQL.

4.5.7 Mapa de Clase

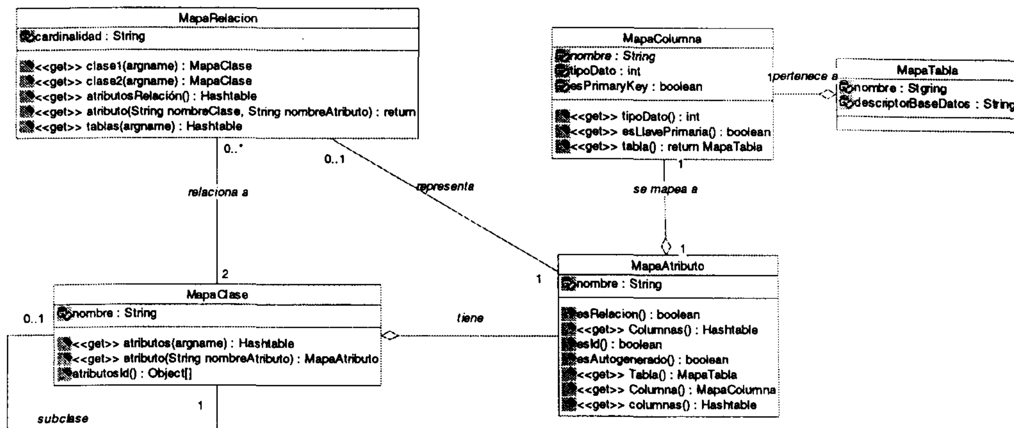

Figura 21. Diagrama de clases del mapa entre las clases y la base de datos.

Este conjunto de clases sirve para representar la relación que existe entre las clases sus atributos con las tablas y los campos de la base de datos.

La clase principal del conjunto de clases es `MapaClase`, puesto que esta contiene toda la información de cómo una clase debe ser mapeada a la base de datos. Un `MapaClase` tiene un conjunto de mapa de atributos.

`MapaAtributo` especifica la relación que tiene un atributo con una columna (`MapaColumna`). Esta clase contiene la tabla y la columna en la que está mapeado el atributo, si su valor es autogenerado y si forma parte de los atributos que identifican únicamente a un objeto de la clase, también se especifica si el atributo representa una relación con otra clase.

La clase `MapaColumna` simplemente contiene información sobre una columna en la base de datos. Contiene el tipo de dato de la columna, la tabla a la que pertenece y si la columna forma parte de la llave primaria de la tabla.

La clase `MapaRelacion` representa la relación que hay entre dos clases. Contiene referencias a los mapas de ambas clases y dentro de estos mapas especifica a que columnas se debe mapear los atributos que identifican a cada clase para especificar la relación entre ambas clases.

Un atributo de una clase puede quedar mapeado a columnas diferentes si el atributo además de formar parte de la información de un objeto, forma parte de los atributos que identifican al objeto y la clase del objeto esta relacionado con otra clase.

4.6 Mecanismos para mapear objetos en la base de datos relacional

La estructura del modelo relacional en el que se almacena la información y la estructura de las clases puede ser muy diferente. En el paradigma orientado a objetos, la aplicación se basa en objetos que tienen tanto datos como comportamiento, mientras que el paradigma relacional se basa en almacenar datos en registros de tablas. En el paradigma de objetos se viaja entre los objetos a través de sus relaciones en memoria, mientras que en el enfoque relacional se unen registros de tablas en base a sus relaciones. Esta diferencia hace difícil la combinación de ambos paradigmas en una aplicación.

Por lo tanto un objetivo de la arquitectura y del framework es que el diseño de las clases de la aplicación no tenga que estar ligado al diseño de la base de datos.

Para lograr el punto anterior se requiere un mecanismo de mapear los datos de las clases a la estructura relacional de la base de datos.

Para determinar de donde se debe obtener o actualizar la información de un objeto en la base de datos se cuenta conceptualmente con una función, que dada una clase y un atributo de esta regresa una tupla formada por {base de datos, tabla, columna}.

Una relación entre objetos también debe ser representada en columnas de una tabla. De esta forma existe una función que dada una relación entre dos clases obtiene para cada atributo que identifique a las dos clases relacionadas una tupla que la identifique en la base de datos.

F(clase, atributo)={base de datos, tabla, columna}

F(clase, atributo, relación)= {base de datos, tabla, columna}

Con estas dos funciones es posible mapear tanto atributos como relaciones de clases a columnas y tablas en la base de datos.

La relación de herencia es transparente puesto que una clase que herede de otra, simplemente hereda sus atributos, y la clase se comporta como si todos los atributos hubieran sido definidos por ella.

En la práctica se aplican los siguientes lineamientos para representar los datos de los objetos en la base de datos:

- **Atributos:** Un atributo que desee hacerse persistente puede ser escrito o leído de una sola columna. La tabla a la que pertenece la columna debe tener un conjunto de columnas que permitan relacionar a un registro con los atributos que identifican a un objeto.
- **Identificador de Objeto:** Todo objeto debe tener una serie de atributos que lo identifiquen únicamente. Estos atributos deben de tener como contra parte una serie de columnas del modelo relacional. En caso que el identificador del objeto no sea asignado por el dominio de la aplicación, es decir sea simplemente un identificador único

autogenerado por la aplicación, el framework asignará automáticamente el identificador del objeto y con este identificador se almacenarán en la base de datos. La manera en la que se genera este identificador depende de la base de datos que se use. Cada implementación concreta de la clase `BaseDatos` debe proveer un método para auto generar valores llave para una determinada tabla. Por ejemplo en la implementación de `BaseDatosOracle` se usa el objeto secuencia para generar los valores.

- **Conversión de tipos:** Las conversiones entre tipos de dato utilizados por un atributo de una clase y una columna en la base de datos, en el caso que estén relacionados por la información que almacenan, se hace automáticamente siempre y cuando los tipos sean compatibles. Las conversiones válidas son:

  Número → String

  Fecha → String

  Entero → Número de punto flotante

- **Relaciones:** Las relaciones entre clases son los elementos que tienen más restricciones en la traducción del modelo de objetos al modelo relacional. Una relación bidireccional entre dos clases se pude descomponer como dos relaciones unidireccionales entre las clases. Es decir si se tienen dos clases relacionadas A←→B, esta relación se puede descomponer como dos relaciones tales que A→ B y B →A. Se asume que tanto la clase A como la clase B tienen un conjunto de atributos que identifican únicamente a cada objeto de la clase.

  La cardinalidad de cada una de estas relaciones puede ser de 1 a 1, lo que significa que en una relacion A→ B, por cada objeto de A, existe solo un objeto de B con el que esta relacionado. Por ejemplo si se tiene la clase Alumno que está relacionado con la clase Carrera, y esta relación indica la carrera en la que esta inscrito el Alumno, para cada objeto Alumno hay solo un obejto Carrera que esta relacionado con el. Esta relacion se representa como una referencia a un objeto Carrera dentro de la clase Alumno.

  En el caso contrario una carrera tiene una relación con de 1 a muchos, una objeto de la clase tiene relación con al menos un objeto de la clase con al que está relacionada. Esta relación se manifiesta con un conjunto de objetos de la clase B en el objeto A. En el caso inverso del ejemplo un objeto de la clase Carrera, solo puede tener un conjunto de objetos Alumno asociados a ésta. Esta relación se representa dentro de la clase Carrera como una colección de Alumnos, esta colección puede estar dada por un arreglo, o cualquier estructura de datos que permita el almacenamiento de varios objetos.

  En el caso en el que dos clases estén relacionadas en ambos lados con cardinalidad de muchos hacia muchos, esta relación se puede descomponer como dos relaciones de 1 a muchos, y cada una tratarla como casos separados. Es decir A←→B ➔ A→B ^ B→ A.

  Para representar una relación de clases con cardinalidad de 1 a 1 del modelo de objetos en el modelo relacional utilizamos las siguientes reglas:

  Sea una relación A→ B con cardinalidad 1.

  > Cada atributo que identifique a la clase A será mapeado a una columna dentro de una tabla, al igual que los atributos que identifican a los objetos de la clase B. Los atributos que identifican a ambas clases, podrán estar mapeadas a dos columnas diferentes en la base de datos, una representando la información de los objetos y otra representando la relación entre las clases.

Ejemplo:

Supongamos que tenemos el diagrama de clases de la Figura 22 y en el modelo relacional de la Figura 24 que representan la relación entre alumnos y carrera.

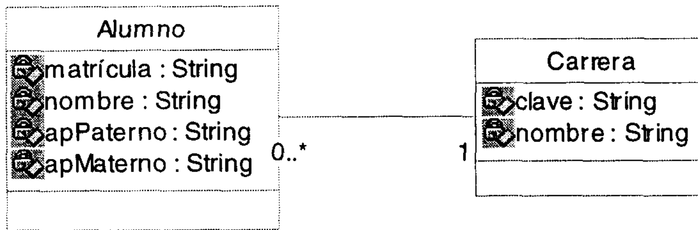

Figura 22. Diagrama de clases de la relación entre Alumno y Carrera.

En este caso la cardindalidad de la relación de Alumno → Carrera es de 1 a 1. Mientras que la relación Carrera → Alumno es de 1 a muchos.

La implementación final de las clases quedaría como sigue:

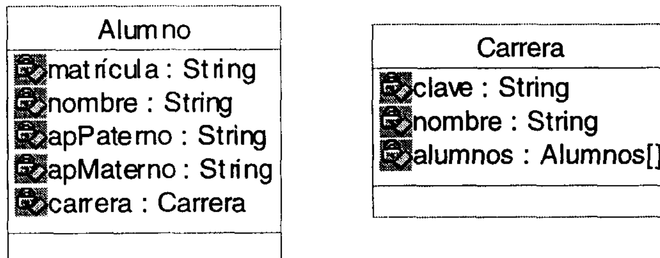

Figura 23. Diagrama de clases de la implementación final de la clase Alumno y Carrera.

La estructura de las tablas es la mostrada en la Figura 24.

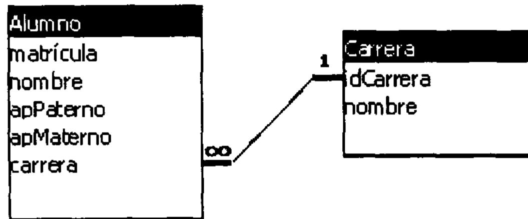

Figura 24. Diagrama entidad relación representando la relación entre Alumno y Carrera.

En este caso los atributos quedarían mapeados como muestra la Tabla 2:

Clase	Atributo	Tabla	Columna
Alumno	Matrícula	Alumno	Matrícula
Alumno	Nombre	Alumno	Nombre
Alumno	ApPaterno	Alumno	ApPaterno
Carrera	Clave	Carrera	IdCarrera
Carrera	Nombre	Carrera	Nombre

Tabla 2. Mapa de los atributos de la clase Alumno y Carrera en la base de datos.

Como se puede observar el mapear los atributos de una clase es bastante sencillo, puesto que solo se tiene que especificar la columna y la tabla que los representa.

Las relaciones quedarían como ilustra la Tabla 3:

Recordemos que para las dos clases que están relacionadas el conjunto de atributos que identifican únicamente a los objetos de ambas clases deben estar mapeados a una columna, para representar la relación

Relación Alumno → Carrera

Clase	Atributo	Tabla	Columna
Alumno	Matrícula	Alumno	Matrícula
Carrera	Clave	Alumno	Carrera

Tabla 3. Representación de la relación de las clases Alumno y Carrera sobre la base de datos realcional.

En este ejemplo se puede apreciar que el atributo clave de la carrera que es el atributo que identifica únicamente a un objeto de la clase carrera, esta mapeado a dos tablas. En una epresenta la información de la carrera y en la otra representa la relación entre el Alumno y la Carrera.

El caso de la matrícula del alumno, que es el atributo que identifica a la clase Alumno solo aparece mapeado en una sola tabla, por la manera en la que están construidas las tablas. En realidad se encuentra mapeado a dos columnas, pero estas columnas resultan ser la misma en este ejemplo.

Si se decidiera cambiar a la estructura de las tablas de la Figura 24, la función de mapeo entre clases y tablas cambiaría también.

La estructura de tablas sería como muestra la Figura 25.

Figura 25. Diagrama entidad relación entre Alumno y Carrera.

En este modelo la relación entre alumno y carrera ya no se mantiene en la tabla de Alumno, sino que se crea una tabla especialmente para representar la relación.

El mapeo de las clases en este esquema es mostrado en las Tablas 4 y 5.

Clase	Atributo	Tabla	Columna
Alumno	Matrícula	Alumno	Matrícula
Alumno	Nombre	Alumno	Nombre
Alumno	ApPaterno	Alumno	ApPaterno
Carrera	Clave	Carrera	IdCarrera
Carrera	Nombre	Carrera	Nombre

Tabla 4. Mapa los atributos de las clases Alumno y Carrera en la base de datos.

Relación Alumno → Carrera

Clase	Atributo	Tabla	Columna
Alumno	Matrícula	AlumnoCarrera	Matrícula
Carrera	Clave	AlumnoCarrera	IdCarrera

Tabla 5. Mapa de la relación entre la clase Alumno y Carrera.

En este caso tanto la matrícula del alumno como la clave de la carrera están mapeadas a dos columnas diferentes, una representando la información de la clase y la otra representando la relación entre las dos clases.

Si se decidiera cambiar la estructura de la base de datos tal como en el ejemplo, lo único que es necesario ajustar es el mapa de relación entre las clases y el modelo de datos relacional. El framework realizará las operaciones de acuerdo a la nueva estructura sin necesidad que cambie ni el código de la aplicación ni el del framework. Con esto se demuestra la flexibilidad que proporciona la arquitectura de la aplicación y el framework para aplicaciones que involucran el acceso a base de datos.

4.7 Metadatos

Puesto que los comandos para la base de datos se forman dinámicamente dependiendo de la operación que solicite la aplicación, se necesita tener información sobre la estructura de la base de datos, la estructura de las clases sobre la que se aplican operaciones de consulta o actualización de datos, y la relación entre la estructura de la base de datos y las clases de la aplicación.

De la estructura de la base de datos se requiere saber

- Tipo de dato de las columnas: Esta información se requiere para saber de que tipo de dato se están representando los datos en la base de datos y realizar las conversiones entre el tipo de dato con el que esta representado en la clase y el tipo de dato con el que se representa en la base de datos.

  También se requiere esta información para poder construir adecuadamente los comandos SQL que ejecutará la base de datos. Por ejemplo en el lenguaje SQL los valores que son representados como texto requieren ir entre comillas. Para poder agregar las comillas al estatuto SQL es necesario saber el tipo de dato de la columna sobre la que se realiza la operación.

- Campos que forman llaves primarias: Esta información es muy valiosa puesto que gran parte de las consultas que se realizan se hacen sobre las llaves primarias de las tablas. En caso que se realice una actualización sobre los datos de un objeto, se necesita construir los estatutos que lo actualicen en todas las tablas de todas las bases de datos en las que se encuentre información del objeto. Para construir los estatutos SQL se requiere saber si es necesario crear un `INSERT` o un `UPDATE`, de acuerdo a si existe o no la información a actualizar en una tabla dada. La manera de saber si ya existen datos del objeto en esa tabla es haciendo un query sobre la llave primaria de la tabla.

  Cuando se requieren actualizar registros ya existentes en una tabla se requiere construir un `UPDATE` en donde la condición de la actualización será dada por los campos que forman la llave primaria de la tabla y los valores del objeto que corresponden.

- Llaves foráneas: Esta información es valiosa puesto que permite establecer un orden en el que se deben ejecutar los estatutos de SQL de actualización. Por ejemplo si se desea actualizar la información de un objeto cuya clase esta mapeada a dos tablas, de las cuales una es llave foránea de la otra, el comando SQL debe de ejecutarse primero en la tabla que contiene la llave foránea, para cumplir con las restricciones de integridad de la base de datos (no se puede insertar un registro en una tabla que tenga una relación de llave foránea con otra tabla, y no exista un registro con que se relacione en la otra tabla)

Los metadatos sobre la base de datos son provistos por la conexión a la base de datos del driver de JDBC. La clase `DatabaseMetadata` provee una serie de *métodos para*

determinar las tablas, esquemas y campos con sus propiedades que tiene visibles el usuario que estableció la conexión en la base de datos.

Sobre las clases se requiere los atributos con sus respectivas propiedades y los constructores.

- Atributos de las clases: Se requieren los atributos de una clase así como su tipo de dato y los valores que posee un objeto en estos atributos para poder formar los estatutos SQL en caso de una consulta y posteriormente asignar los valores que regresó la base de datos a los objetos. Para el caso de la actualización de un objeto se requieren obtener los valores de sus atributos para formar los estatutos de actualización y ejecutarlos en la base de datos.

  Tanto en el caso de la consulta como actualización se requieren los tipos de datos de los atributos para realizar la conversión de tipos entre los que están especificados en la base de datos y los de las clases.

Esta información la provee el *Reflection API* de Java. Este consiste en una serie de clases que permiten obtener información sobre un objeto y la definición de la clase a la que pertenece tal como: nombre de la clase, lista de atributos públicos, tipo de dato o clase a la que pertenece cada atributos y la clase de la cual hereda. También permite obtener y asignar un valor sobre un atributo público.

Proporciona información sobre los métodos de la clase y los parámetros que lo conforman, permite ejecutar un método sobre un objeto. También permite la creación dinámica de objetos. Tanto los metadatos de la base de datos como los metadatos de las clases, las provee el lenguaje Java, a través de sus APIs JDBC y Reflection API respectivamente.

Hay un tipo de metadatos que se requiere y no son provistos por el lenguaje. Estos son la relación que existe entre las clases y el modelo relacional. El framework almacena esta información en estructuras internas. La relación entre las clases y el modelo relacional lo especifica el usuario en la inicialización del framework, para que estos datos estén disponibles durante todo el tiempo de vida de la aplicación.

4.8 Forma de especificar criterios

En una base de datos relacional los estatutos de manipulación de datos se especifican por medio del lenguaje SQL. Por medio de éste se especifican tanto los datos, como las estructuras en donde se manipularán. Estas estructuras son las tablas y campos.

En la arquitectura de la aplicación orientada a objetos, la manipulación de datos se especifica por los atributos de las clases y los valores que éstos tengan. Para esto se requiere una serie de herramientas para poder especificar tanto consultas como actualizaciones sobre la base de datos en términos de las clases, sus atributos y valores de éstos.

El framework provee un conjunto de clases que permite especificar criterios de consulta y actualización basados en los atributos y relaciones de las clases.

Las clases proveen servicios para especificar el valor de un atributo de una clase y operadores de igualdad (=,<=,<,>,>=,like)con el que relaciona al atributo con el valor.

Se puede hacer una combinación de criterios combinando los criterio con operadores boleanos (AND, OR, NOT).

Las clases que proporcionan estos servicios son:

- `CriterioSimple`: Representa un atributo de una clase, su valor y el operador de igualdad o desigualdad que relaciona al atributo con el valor
- `CriterioAnd`: Representa el AND lógico entre dos criterios
- `CriterioOr`: Representa el OR lógico entre dos criterios
- `CriterioNot`: Representa el NOT o la negación de un criterio.

Puesto que las 4 clases son subclase de `Criterio`, esta jerarquía permite especificar criterios anidados con tanta profundidad como sea necesario.

Con estos dos servicios se pude especificar un criterio de consulta o actualización basado en los atributos de una clase. Sin embargo también es necesario especificar criterios en base a las relaciones entre las clases. Para lograrlo se hace uso de una clase `CriterioRelación`, que permite especificar un criterio en base a la relación entre dos clases. Dada una relación entre las clases A→ B, el criterio permite especificar a los objetos de A que estén relacionados con objetos de la clase B que cumplan un criterio dado. Primero se aplica un criterio sobre la clase B (basado en sus atributos o relaciones) y posteriormente, para todos los objetos de B que cumplan con el criterio se obtienen los objetos de A con los que están relacionados.

Ejemplo:

Supongamos el diagrama de clases de la Figura 26 que representa la información básica de un alumno y su relación con la carrera que está cursando.

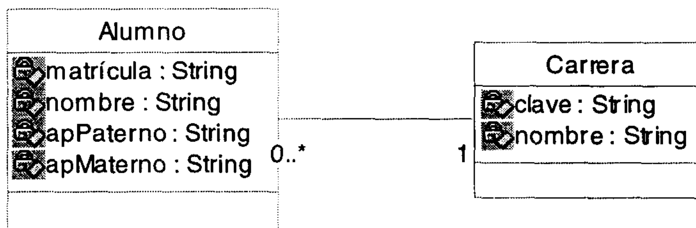

Figura 26. Diagrama de clases de la clase Alumno y Carrera.

Consulta por atributos:

Supongamos que queremos obtener al alumno con matrícula X. El criterio se especificaría con un `CriterioSimple` de la siguiente forma:

CriterioSimple("Alumno","matricula","=",X).

Si deseamos especificar a todos los Alumnos cuyo apellido empiece con "M" y cuyo nombre empieza con "A" o con "L", se requeriría la combinación de CirterioSimple, `CriterioOr` y `CriterioAnd`.

Usando notación polaca podría representarse como:

(CriterioAnd (CriterioSimple ("Alumno","apPaterno","like","M%"), CriterioOR (CriterioSimple "Alumno","nombre","like","A%"), CriterioSimple("Alumno","nombre","like","L%")))).

Ahora supongamos que deseamos obtener todos los alumnos de la carrera "ISC", cuyo apellido paterno empiece con "M". En esta consulta se están usando los atributos de ambas clases, por lo que es necesario el uso de `CriterioSimple`, `CriterioRleacion`, y `CriterioAnd`. El criterio quedaría representado como sigue:

(CriterioAnd(CriterioSimple("Alumno","apPaterno","like", "M%"),CriterioRelacion("Alumno", "carrera", CriterioSimple("Carrera", "clave","=","ISC"))))

En este ejemplo se está asumiendo que en la implementación de la clase Alumno hay un objeto de clase Carrera, que representa la relación del Alumno con la Carrera

4.9 Algoritmos detallados para operaciones de consulta transaccionales

A continuación se presenta el algoritmo detallado para realizar la consulta de objetos de una clase.

La consulta de objetos de una clase se pude dar de tres formas:

4.9.1 Consulta de atributos

- La aplicación solicita los objetos de una clase X y especifica un criterio.

- El Broker crea una operación de consulta con el mapa de la clase y ejecuta la consulta. La operación regresa un arreglo de objetos que cumplen con el criterio.
    - Crea un objeto `SqlSelect` y en base al mapa de la clase y al criterio se forma un estatuto `Select` para obtener los valores de los atributos de los objetos que cumplen con el criterio especificado, de la base de datos.
        - El objeto `SqlSelect` analiza la consulta basado en el criterio y en el mapa de clases y obtiene todas las tablas y campos que hay que consultar en la base de datos y los joins que hay que hacer para satisfacer la consulta. Este análisis se realiza utilizando los metadatos de la base de datos (la existencia de llaves foráneas) y los mapas de relación entre las clases y la estructura de la base de datos.
        - Una vez que se tiene todo el conjunto de tablas, campos y joins que se ocuparán para realizar la consulta, simplemente se forma el estatuto `Select` con la sintaxis apropiada. El estatuto `Select` tiene el formato:

          Select {lista de campos} from {lista de tablas} where {condición}
        - Los campos que se obtuvieron del análisis de la consulta se colocan en la lista de campos. Las tablas que intervienen en la consulta se colocan en la lista de tablas y la condición del `where` se forma a partir de los `joins` que se detectaron y de los valores que se especificaron en el criterio.
    - Se ejecuta el query y se obtiene un `ResultSet` con los resultados de la consulta.
    - Se crea un objeto `FabricaObjeto` para materializar los objetos a partir de los datos del `ResultSet`:
        - Para cada registro obtenido en el `ResultSet`. El *ResultSet* es una clase por la cual el JDBC regresa los datos de una consulta a la aplicación. Esta clase representa a los registros de una tabla en el modelo relacional.
            - Se crea un objeto de la clase con los datos del renglón.
            - El objeto `FabricaObjeto` crea una instancia de un objeto de la clase sobre la que se hizo la consulta.
            - Para cada atributo del objeto que no representa una relación con otro objeto de la aplicación:
                - Se obtiene el valor de la columna que lo representa.
                - Se asigna el valor al atributo.

4.9.2 Consulta de atributos y de objetos con los que esta relacionado un objeto

- Se realiza el proceso anterior hasta el punto donde se obtiene el `ResultSet` con los atributos del objeto.
- Para cada registro obtenido en el `ResultSet`:
    - Se crea un objeto de la clase con los datos del registro.
        - El objeto `FabricaObjeto` crea una instancia del objeto de la clase sobre la que se hizo la consulta.
            - Para cada atributo del objeto que no represente una relación con otro objeto:

      - Se obtiene el valor de la columna que lo representa.
      - Se asigna el valor al atributo.
    - Para cada atributo que representa una relación con otro objeto:
      - Se forma un criterio especificando la relación entre el objeto que se materializó y los objetos de la clase con la que está relacionado. Este criterio se especifica usando los valores de los atributos que identifican al objeto materializado.
      - Se crea una `OperacionConsulta` para la clase con la que el objeto está relacionado mediante el atributo con el que se está trabajando
      - Se especifica que la consulta se hará únicamente con los atributos del objeto, es decir no se tomarán en cuenta las relaciones.
      - Se ejecuta la operación. De manera recursiva se obtiene un conjunto de objetos siguiendo los pasos que se especifican en el primer tipo de consulta.
      - Se asignan los objetos de la consulta al atributo que representa la relación.
      - Si se obtuvo solo un objeto, simplemente se asigna la referencia del objeto al atributo.
      - Si el número de objetos es mayor que 1, se crea un arreglo de objetos y se trata de asignar al atributo. El atributo deberá ser un arreglo de objetos para que la asignación sea válida.

4.9.3 Consulta recursiva de objetos

El proceso es parecido al anterior, la parte en la que difiere es que cuando se crea una `OperacionConsuta` para obtener los objetos con los que está relacionado, se especifica que la consulta sea recursiva. De esta manera se obtendrán todos los objetos con los que está relacionado directa o indirectamente el objeto.

4.9.4 Actualización de un Objeto

La actualización de objetos se puede realizar de tres maneras. Una es actualizar únicamente los valores de los atributos del objeto, otra es actualizar los atributos y las relaciones del objeto con otros. La tercera consiste en actualizar los atributos del objeto, las relaciones entre objetos y la actualización de los objetos con los que tiene relación.

4.9.5. Actualizacion de atributos

- La aplicación especifica al `Broker` que se desea actualizar un objeto. Invoca el método `actualizaObjeto` con el objeto a actualizar como parámetro.
- El `Broker` crea una `Transacción`.
- El `Broker` crea una `Operación de Actualización del Objeto`.
- El `Broker` agrega la `Operación` a la `Transacción`.
- El Broker ejecuta la actualización (sin hacer `Commit`)
  - Analiza la actualización.

- Obtiene los valores de los atributos del objeto usando la clase `FabricaObjeto`.
- Se obtiene el criterio que satisface la identificación del objeto a actualizar.
- Se determinan todas las tablas en las que deberán ser actualizados los valores de los atributos del objeto y estas se ordenan en la secuencia adecuada de actualización (para satisfacer las restricciones de integridad de la base de datos).

o Para cada tabla que este involucrada en la actualización del objeto

- Se actualiza el objeto en la tabla correspondiente:
    - Se crea una `OperacionConsulta` con el criterio que identifica al objeto (creado anteriormente).
    - La `OperacionConsulta` determina el número de registros que tiene un objeto sobre una tabla.
    - Si el número es 0 se crea un objeto `SqlInsert`.
    - Si el número es mayor que 1 se crea un objeto `SqlUpdate`.
    - Se forma el comando `Sql` respectivo.
    - En caso que algún valor de una campo en la tabla necesite ser autogenerado, se le pide a la base de datos que genere el valor. La manera de generar un valor único depende de la implementación de la base de datos
    - En el caso de `SqlInsert` se forma un comando con la siguiente forma :

      INSERT INTO nombre Tabla (lista de campos) values (lista de valores).

      La lista de campos se obtiene analizando el mapa de la clase y los metadatos de la base de datos. La lista de valores se obtiene de los valores de los atributos del objeto.
    - En el caso de `SqlUpdate` se forma un comando con la forma *UPDATE nombreTabla SET campo1=valor1,..n, campon= valorn where condición.*

      La correlación entre campo y valor se obtiene del mapa de la clase, los metadatos de la base de datos y los valores de los atributos del objeto. La condición del where se obtiene a partir del criterio formado anteriormente.
- Se hace `Commit` a la base de datos.

4.9.6 Actualización de relaciones

- Para actualizar los valores de los atributos de un objeto, así como sus relaciones con otros objetos en la base de datos, se sigue el proceso anterior para actualizar los valores de los atributos.
- Para cada atributo en el objeto que represente una relación:
    - o Se borran de la base de datos todos los registros que representen la relación del objeto con cualquier otro objeto de la clase con la que está relacionado.
        - Se crea un `CriterioRelación` que representa la relación de los dos objetos.
        - Para cada tabla en la que se represente la relación entre las dos clases
            - Crear un objeto `SqlDelete`.

- Formar un estatuto `DELETE` que elimine los renglones que representen la relación del objeto.
- El estatuto `DELETE` tiene la siguiente forma:

  DELETE from nombreTabla Where condición. La condición del where se forma a partir del `CriterioRelacion` formado anteriormente.
- Para cada objeto con el que este relacionado por medio del atributo en cuestión:
  - o Para cada tabla en la que se represente la relación entre los objetos.
  - o Se consulta el numero de registros que representan la relación en la tabla.
  - o Si el número de registros es 0 se crea un objeto `SqlInsert`.
  - o De lo contrario se crea un objeto `SqlUpdate`.
  - o Se genera el estatuto de `Sql` correspondiente y se ejecuta en la base de datos. El estatuto `Sql` se genera a partir de los campos que están mapeados a los atributos que identifican únicamente a un objeto de la clase, y con los valores que tienen estos atributos en los dos objetos relacionados.

▪ Se aplica `Commit` a la transacción.

- Se genera `Commit` en la base de datos con todas las operaciones que se realizaron.

4.9.7 Actualización de objetos con los que se relaciona

▪ El algoritmo actualiza tanto los atributos del objeto, sus relaciones con otros objetos e inclusive la actualización de los objetos con los que está relacionado, excepto que antes de actualizar las relaciones se actualizan los objetos con los que están relacionados. Esto se hace para garantizar que no se violen las restricciones de integridad de la base de datos que imponen las llaves foráneas.

4.9.8 Borrar un objeto

El algoritmo para borrar los datos de un objeto es el que sigue:

▪ Se crea un objeto `Transacción.`

▪ Se crea una `OperacionActualizaObjeto` y se agrega a la transacción.

▪ Se obtienen los atributos y los valores que identifican al objeto.

▪ Con estos atributos y valores se forma un criterio que identifique al objeto.

▪ En base a este criterio y el mapa de la clase se hace un análisis de todas las tablas en donde esta almacenada la información de una clase.

  - o Para cada tabla en la que se almacene información de la clase
  - o Se priorizan en orden de tal manera que no se violen las restricciones de llave foránea de la base de datos al borrar registros de una de estas, cuando hay registros "hijos" en otra tabla con la que tiene una relación de llave foránea.
  - o Se crea un objeto `SqlDelete`
  - o Se forma el estatuto `DELETE` con la siguiente forma:

    DELETE nombreTabla where condición.
  - o Se ejecuta el estatuto `DELETE`.

- Se da `commit` a la transacción

4.9.9 Transacciones entre múltiples operaciones.

- Si se desea realizar una transacción con varias operaciones sobre objetos, como por ejemplo actualizar varios objetos sobre la base de datos y borrar un objeto de la base de datos en una sola transacción, el proceso es el siguiente:
- Se crea una transacción.
- Se crea la operación que solicita el usuario.
- Se agrega la operación a la transacción.
- Se ejecuta la transacción
  - Cada subtransacción y operación se ejecuta.

4.10 Mecanismo de extensión del framework

El framework cuenta con un mecanismo de extensión de tal manera que pueda ser usado contra cualquier base de datos relacional. El framework cuenta con una interfase que define el comportamiento y las operaciones de una base de datos. Todas las operaciones internas del framework se hacen usando esta interfase. De esta manera se pueden proveer clases que cumplan con esta interfase y que implementen el comportamiento necesario para el funcionamiento con una base de datos en particular. Esta clase es responsable de cargar el driver necesario de JDBC para establecer la conexión a la base de datos, obtener los metadatos de ésta y ejecutar los comandos sobre una base de datos en particular.

El framework cuenta con una clase a través de la cual la aplicación realiza las operaciones. Es la única clase que es visible a la aplicación. Para agregar una nueva funcionalidad al framework es necesario agregar los métodos necesarios para que la aplicación pueda accesar los nuevas funciones. Esta clase es solo un *wrapper* que presenta un frente de acceso único a la aplicación y une a los demás componentes del framework. Un wrapper es un patrón de diseño que encapsula objetos, los cuales proveen servicios diferentes a los del wrapper. Un wrapper provee una manera flexible de combinar comportamiento y propiedades de varios componentes [GAMM95].La clase que funciona como wrapper es el Broker puesto que encapsula el comportamiento de muchas otras clases.

La clase `Operación` es la que realiza las acciones del framework. Una operación es clasificada como una operación de consulta o una operación de actualización. Bajo esta jerarquía se definen operaciones como la consulta de objetos o su actualización. Si se desea agregar una nueva operación al framework se deberá derivar una clase de la jerarquía de Operación e implementar su funcionalidad (además de agregar el método en Broker para ejecutar la operación).

4.11 Implementación en Java

La implementación del framework se realizó en Java. El lenguaje provee una serie de mecanismos para:

- El acceso a base de datos. En el nivel más bajo todos las operaciones sobre la base de datos se especifican por medio de JDBC. Desde la conexión hasta la ejecución de estatutos de manipulación de datos, y la obtención de los metadatos de la base de datos, son realizados por JDBC.
- Manipulación en tiempo de ejecución de objetos desconocidos en la compilación. El API Reflection hace posible que se puedan crear objetos dinámicamente sin conocer ni su clase ni sus propiedades en tiempo de compilación. Esta característica es la que permite que los resultados de las consultas se regresen a la aplicación en forma de objetos y que se puedan obtener dinámicamente los valores almacenados en los objetos para actualizarlos en la base de datos. Gracias a esto se puede independizar a la aplicación de la base de datos.
- Una librería extensa de estructuras de datos (tablas de hash, arreglos, listas, etc). Dentro del paquete java.util, el lenguaje incorpora una serie de clases para el manejo de

colecciones de objetos. Incluye la implementación de las estructuras de datos más comunes. Esto permite que el desarrollador se centre en la aplicación y evite la necesidad de programar sus propias estructuras de datos.

Estas tres librerías facilitaron mucho la implementación del framework. Es de hacer notar que las 3 librerías forman parte del lenguaje de programación, es decir no son propietarias de algún proveedor, y éstas están definidas dentro de la especificación del lenguaje.

Capítulo 5

Experiencias.

Basados en la arquitectura propuesta en el capítulo 3 y en el diseño de la capa de acceso a datos propuesta en el capítulo 4, se desarrolló un prototipo del framework. A partir de su implementación y su uso se obtuvieron los siguientes resultados.

5.1 Ventajas y Desventajas del uso de la arquitectura y el framework

El modelo presenta las siguientes ventajas y desventajas.

5.1.1 Ventajas:

- Construcción rápida de aplicaciones: El uso de la arquitectura y el framework permiten un desarrollo rápido de aplicaciones puesto que se usa un diseño general de la aplicación previamente construido, y un mecanismo ya construido que provee el acceso a los datos en una fuente de almacenamiento externo, sin la necesidad de escribir código para lograrlo.
- Integración de diferentes bases de datos en una aplicación: La capa de acceso a datos puede tanto almacenar como obtener los datos de los objetos de la aplicación de una base de datos. El detalle del acceso a la base de datos es aislado por la capa de acceso a datos. Gracias a esto es posible cambiar las bases de datos que usa la aplicación a otro producto y aún así la aplicación seguirá funcionando sin la necesidad de hacerle algún cambio.
- Mecanismo sencillo para acceso a los datos: Para accesar los datos de las bases de datos, solo es necesario incluir la librería en la que se implementa la capa de acceso a datos, crear un objeto Broker, cargar en su configuración inicial el mapa que relaciona las clases de la aplicación con la base de datos y usar los métodos para consultar objetos, actualizar objetos y crear transacciones. El uso de la arquitectura y la capa de acceso a datos se vuelve bastante fácil de usar, esto disminuye la curva de aprendizaje de la herramienta.
- Aísla a la aplicación de la estructura de la base de datos: Los querys son generados por la capa de acceso a datos. Una vez que se tenga el mapa entre las clases y la estructura de la base de datos, el desarrollador no tiene por que preocuparse por como está representada la información en la base de datos. Puesto que se elimina la necesidad de que el desarrollador construya los querys para accesar los datos de la base de datos, el desarrollador puede construir la aplicación sin necesidad de saber siquiera el nombre de las tablas en donde se almacenan los datos relevantes para la aplicación.
- Amortigua los cambios en la estructura de la base de datos: Si cambia la estructura de la base de datos, las estructura de las clases de la aplicación no tienen por que cambiar. Puesto que todos los querys que se realizan a la base de datos se construyen dinámicamente usando los metadatos, su construcción se adaptará automáticamente ante un cambio en la estructura de la base de datos. Esta ventaja hace muy flexible a la aplicación y disminuye su costo de mantenimiento.
- Evita mezclar el paradigma de orientación a objetos con el paradigma relacional: El desarrollador evita el tener que trabajar con el paradigma orientado a objetos y el paradigma relacional. El nivel de acceso a datos, realiza la conversión entre los elementos usados por el paradigma relacional y los elementos usados por el paradigma orientado a objetos. De esta manera el desarrollador usa únicamente los elementos del paradigma orientado a objetos.
- Hace más reusables a las clases de la aplicación: La arquitectura evita que las clases implementen el mecanismo para hacer persistentes sus datos por tanto las hace más reusables en otras aplicaciones, donde tal vez el mecanismo de persistencia es distinto o donde no se requiere que los objetos se hagan persistentes.
- La capa de acceso a datos es reusable en cualquier aplicación: La capa de acceso a datos es construida como un servicio genérico y puede ser usada en cualquier aplicación que lo

requiera, evitando con esto la necesidad de escribir código para accesar los datos de la base de datos en cualquier aplicación implementada en el lenguaje Java.

5.1.2 Desventajas

- No existe un proceso de validación de tablas, campos, ni de clases: Puesto que la construcción de querys se hace dinámicamente no hay un proceso de validación en el que se verifique que las tablas y campos sobre las que se accesan los datos exista en el momento de compilación. Tampoco existe una verificación de que exista la definición de una clase en el momento de hacer una consulta sobre ésta a partir de un criterio. Todos los errores que existan se generarán en tiempo de ejecución y no de compilación.
- Depende de la construcción correcta del mapa entre las clases y la estructura de la base de datos: El mapa entre las clases y la estructura de la base de datos es definida por el usuario. Si el mapa está mal construido, las operaciones que se especifiquen al Broker de la base de datos, serán mal ejecutadas. Cualquier aplicación que use ese mapa será defectuosa.
- No permite que el desarrollador optimice el código SQL: Puesto que los estatutos de SQL se generan dinámicamente por la capa de acceso a datos, no da la oportunidad al desarrollador de optimizar el código de SQL que se genera. Si el desarrollador conoce la estructura de la base de datos y la naturaleza de la información almacenada, así como su volumen podría ser capaz de optimizar la manera en la que se accesa a las bases de datos.
- Costo computacional: Todo, desde la construcción de los querys hasta la creación de objetos se hace dinámicamente. Cada vez que se realiza cualquiera de estas operaciones, es necesario consultar los metadatos, lo cual involucra un tiempo de ejecución y hace que el desempeño de la aplicación sea menor, comparado con la especificación de querys y objetos en tiempo de compilación. Se elimina la posibilidad de que el compilador pueda realizar optimizaciones.
- Restringe el diseño de las clases de la aplicación: Por las restricciones del lenguaje, es requisito que todos los atributos sobre los que se pueda especificar un criterio de consulta, o aquellos cuyo valor sea almacenado en la base de datos, sean declarados públicos. Esto se debe a que el Reflection API de Java solo puede accesar y asignar valores de atributos públicos. Esta es una de las desventajas mas grandes del modelo, puesto que viola el principio de ocultamiento de información.

  En el caso de las relaciones entre dos clases con cardinalidad de 1 a N, la relación debe de representarse como un arreglo de objetos de la clase con la que se tiene la relación.
- Parte esencial del modelo consiste en la conversión automática de registros de la base de datos a objetos en el lenguaje de implementación. Esto fuerza a que toda la aplicación tenga que ser desarrollada en el mismo lenguaje de programación que en el que está implementada la conversión de registros a objetos. Por ejemplo, si el framework de acceso a datos está implementado en Java, toda la aplicación deberá estar implementada en Java. Esto limita el uso de tecnología dentro de el desarrollo de una aplicación.

5.2 Problemas de implementación.

En el diseño e implementación del componente de acceso a datos se detectaron los siguientes problemas en su diseño e implementación:

- No hay cache de objetos: En el diseño propuesto no se cuenta con un mecanismo de caché de objetos que evite la necesidad de construir repetidamente un query y la construcción de un objeto, el cual ya se había obtenido de la base de datos. También evitaría la necesidad de intentar actualizar un objeto en la base de datos sobre el cual no se han realizado modificaciones en los datos.
- No existen mecanismos para discriminar que información de un objeto debe ser almacenada u obtenida en una operación sobre la base de datos: La operación atómica

consiste en actualizar o consultar los valores de todos los atributos de un objeto. En cada operación de consulta se obtienen y se asignan a un objeto todos los valores de los atributos que están definidos en el mapa de relación de la clase con la base de datos. Sin embargo para el propósito de la aplicación pueden ser necesarios solo un subconjunto de estos valores. Aún así la capa de acceso a datos obtendría el valor de todos los atributos.

En el caso de la actualización de un objeto se puede dar el caso que sólo algunos valores de los atributos sean validos en el momento de la actualización, pero como la operación atómica es actualizar todos los valores, la información que se escribe a la base de datos puede ser incorrecta.

- Se requiere que los atributos de donde se leen y asignen valores en una clase sean declarados como públicos: el Reflection API solo permite trabajar con atributos públicos. Sin embargo es una mala técnica de diseño mantener a todos los atributos como públicos.
- Las relaciones con cardinalidad 1 a N se representan como arreglos de objetos: Para que el framework de acceso a datos pueda asignar una relación a un objeto, así como obtener los objetos con los que está relacionado, exige que las relaciones de cardinalidad 1 a N se representen como un arreglo de objetos. Es decir si se tiene la relación A→ B de cardinalidad 1 a N. La clase A tiene definido un atributo que es un arreglo de objetos de la clase B.

  Esta restricción limita al programador a que no pueda usar otras formas de representar las relaciones entre objetos.
- Problemas con actualizaciones y consultas recursivas: Supongamos que tenemos dos clases A y B, para las cuales A→ B y B→A es decir la clase A mantiene una relación con objetos de la clase B y la clase B mantiene una relación con objetos de la clase A. Si se especifica una consulta sobre los objetos de la clase A, en la que se desee inclusive obtener los objetos de la clase B con los que estén relacionados los objetos de A, se presenta el siguiente problema. Al tratar de materializar los objetos de B, se obtiene que como hay objetos de A con los que hay una relación, también deben obtenerse de la base de datos los valores A y crear los respectivos objetos. Sin embargo puesto que la consulta original se especificó sobre la clase A, es probable que los objetos que requiere el objeto de clase B ya estén creados. Cuando se trata de materializar el objeto de A se da cuenta que requiere objetos de B, y se cae en un ciclo infinito. A → B → A→ B.....
- Los objetos se crean con un constructor vacío: Al momento de realizar una consulta sobre los objetos de una clase, éstos se crean usando su constructor vacío. Esto requiere que todas las calases que se requieran sean persistentes, deben implementar un constructor vacío. Si la clase tiene algún otro constructor en el que se realicen acciones como inicialización del objeto, validaciones, etc., éstas no podrán ejecutarse al momento de crear el objeto.
- No se valida que las tablas o clases existan: Si en el mapa que relaciona a las clases con la base de datos, hay algún campo o tabla que no exista dentro de la base de datos, la construcción tomará a estos elementos como si en realidad existieran y al momento de ejecución, se generará un error en la aplicación.

  Igualmente, si se realiza una consulta de objetos sobre una clase que no existe dentro de la aplicación, generará un error hasta el tiempo de ejecución.
- No se usan proxys: Un *proxy* es un objeto que representa a otro. El proxy puede tener la información mínima necesaria para identificar al objeto al que representa. El uso de proxys ayuda a que al momento de realizar una consulta sobre objetos de una clase, solo se obtengan los proxys de los objetos con la información mínima necesaria. A medida que se van accesando cada uno de estos proxys, se van materializando los objetos a los que representa. De esta manera si solo se usa un subconjunto de los objetos que genera una consulta, se evita tener todos los datos de todos los objetos en memoria (que muy probablemente no se utilizarán) [LARM98].
- La implementación de toda la aplicación debe hacerse sobre el mismo lenguaje y este debe proveer mecanismos para crear, obtener y modificar dinámicamente las propiedades de objeto en tiempo de ejecución. El modelo de construcción de la aplicación se basa en

la manipulación de objetos que no son conocidos en tiempo de compilación. Esto restringe la posibilidad de uso a lenguajes que permiten esta característica, tales como Java y Objective C.

- El desarrollador se ve limitado a utilizar un mismo lenguaje en todo el desarrollo de la aplicación. La arquitectura no especifica un mecanismo en el que se puedan combinar diferentes lenguajes de implementación. Por ejemplo, que el desarrollador decidiera utilizar C++ para una parte del código por su buen desempeño, algún lenguaje visual para el desarrollo de las intefases de usuario y el lenguaje java para el acceso a datos con el framework propuesto. En este caso el desarrollador de la aplicación se vería obligado a construir un puente que tradujera la información de la estructura usada por un leguaje a la estructura usada por otro. Por ejemplo, convertir los objetos de java, a objetos de C++.

5.3 Escenario de prueba.

Se desarrolló una aplicación de complejidad mediana que simula el proceso de inscripción de un alumno en los grupos en un período escolar bajo dos modelos de la base de datos, y un solo código de la aplicación. La aplicación incluía la siguientes consulta y actualización sobre la base de datos de el siguiente conjunto de datos:

- Datos del alumno
- Materias que puede cursar el alumno
- Grupos que imparte una materia
- Carrera del alumno

Los resultados obtenidos indican que se puede cambiar el modelo de datos, y el framework de acceso a datos hace posible que la aplicación no tenga que ser modificada en aspecto alguno, para soportar los cambios sobre el esquema de la base de datos. Los cambios que se realizaron sobre la base de datos son:

- Cambio de nombre de una tabla
- Cambio del nombre de las columnas de una tabla
- Segmentar las columnas de una tabla en dos tablas
- Agregar una tabla intermedia para representar la relación entre dos entidades

Los modelos de datos con los que se probó la aplicación son los mostrados en las Figuras 27(modelo original) y 28(modelo nuevo):

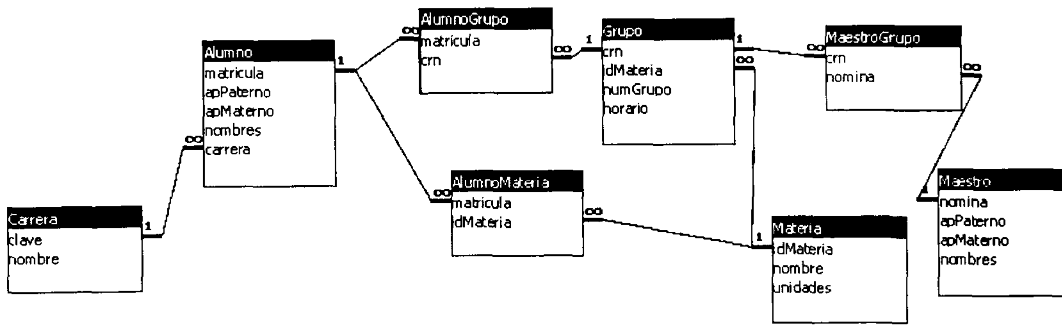

Figura 27. Modelo relaciona de inscripciones 1.

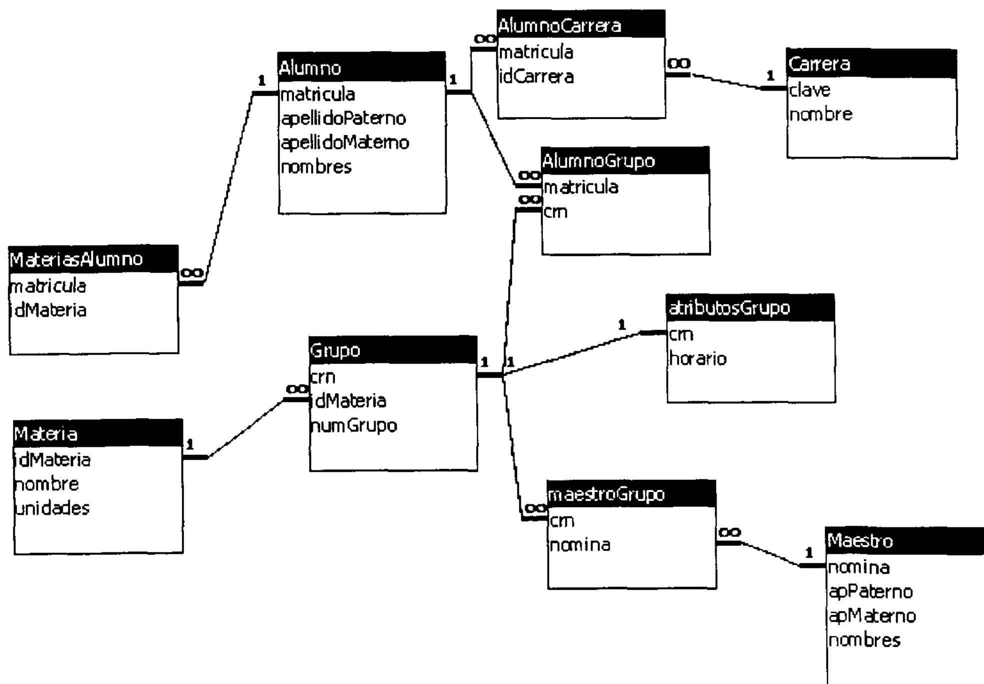

Figura 28. Modelo relacional de inscripciones 2.

Capítulo 6

Conclusiones y Trabajo futuro

6.1 Conclusiones

Existe la necesidad de que las aplicaciones se construyan cada vez más rápido y que tengan la capacidad de adaptarse. en el menor tiempo posible, a las restricciones que imponen los negocios, sin sacrificar la calidad del software.

Una alternativa para hacer aplicaciones más rápido, más adaptables y de mejor calidad, es usar el concepto de líneas de producción, en donde los nuevos productos se construyen a partir de partes prefabricadas.

Para facilitar este proceso en la producción de software se requieren entre otras cosas, una arquitectura definida para una familia de sistemas similares y un conjunto de componentes disponibles para construir la mayor parte de la aplicación.

La arquitectura, determina la estructura interna de la aplicación e impone las restricciones con las que ese conectarán los componentes de la misma. Si conocemos la arquitectura de una aplicación podemos analizarla y entenderla sin necesidad de conocer los detalles de la aplicación. En base a la arquitectura se pueden diseñar los componentes de los que estará constituida la aplicación.

La ventaja de usar una arquitectura previamente definida al construir una nueva aplicación es que estamos reusando desde el diseño de aplicaciones anteriores y disminuimos así el tiempo de desarrollo, a la vez que aumentando la facilidad de construcción y mantenimiento de la aplicación.

En el caso de la aplicaciones orientadas a objetos que accesan a bases de datos, realizan en común la conexión a las bases de datos, la especificación de querys por medio de SQL y la conversión de objetos a las estructura requerida por la base de datos y vise versa. A partir de estas características comunes entre las aplicaciones que realizan accesos intensivos a bases de datos se define una arquitectura en común para las aplicaciones.

La arquitectura es una arquitectura en capas en la que todas las operaciones de acceso a datos en fuentes de almacenamiento persistente se localiza en una capa. La capa es responsable de conectarse con las bases de datos necesarias, construir los querys para satisfacer los requerimientos de datos de la aplicación, ejecutarlos, y realizar la traducción entre el paradigma relaciona y el paradigma orientado a objetos.

En esta arquitectura se propone un componente genérico, previamente construido que pueda ser reutilizado por cualquier aplicación que requiera acceso a bases de datos. El objetivo de el componente es actualizar los valores de objetos en las bases de datos y consultar directamente objetos de ésta. Ambas operaciones se especifican usando la terminología de el paradigma orientado a objetos y no del paradigma relacional. Es decir se especifica que se quieren obtener los objetos de una clase que cumplan con un criterio (especificado a partir de los atributos de la clase), o que se desea actualizar un objeto.

En el caso de consultas, el componente de acceso a datos construye los querys que ejecutará la base de datos, obtiene los resultados éstas y crea los objetos que solicitó la aplicación a partir del resultado de la consulta.

En el caso de la actualización de un objeto, la capa de acceso a datos obtiene los valores del objeto, forma el query necesario para actualizar los valores en la base de datos y ejecuta la actualización.

Para poder generar los querys automáticamente se requiere tener información sobre la estructura de la base de datos, la estructura de las clases que usa la aplicación y la relación entre ambas.

Puesto que la capa de acceso a datos realiza dinámicamente los querys de la base de datos y construye los objetos basado en los resultados obtenidos, la aplicación no accesa directamente a la base de datos y no está acoplada a la estructura de la misma. Si ocurre algún cambio en la base

de datos la aplicación no tiene que realizar ningún cambio para soportarlo, únicamente se debe actualizar el mapa que relaciona a las clases de la aplicación con la estructura de la base de datos.

Las ventajas que trae este enfoque son:

- Se elimina la necesidad de escribir código para accesar la información de la base de datos, esta labor la realiza el componente de acceso a datos.
- La aplicación se puede adaptar rápidamente a los cambios tanto en la estructura de la base de datos como en el producto de la base de datos.

Como conclusión final, podemos decir que el uso de una arquitectura estándar para construcción de aplicaciones familiares facilita el proceso de construcción del software y permite el reuso no solo de código sino desde el diseño de las aplicaciones. Igualmente mientras más componentes se tengan previamente construidos menos esfuerzo en desarrollar las aplicaciones. Puesto que los componentes están probados y cumplen con una función específica, la posibilidad de errores en la aplicación disminuye.

En el caso de la tesis se desarrolla una arquitectura para aplicaciones que tienen acceso a bases de datos, que evite la construcción de el código para el acceso a los datos por cada aplicación que se construya y que facilite la adaptabilidad de la aplicación ante cambios en la base de datos. Se provee un prototipo de un componente para acceso a datos. A pesar de que el componente es solamente un prototipo y carece de características para uso en producción. El resultado de su uso es favorable puesto que las aplicaciones que se construyen con este componente, evita la codificación manual de los querys y permiten modificaciones a la estructura de la base de datos.

El diseño de la arquitectura y del componente de acceso a los datos dan lineamiento para implementar mecanismos similares en otros lenguajes de programación.

La conclusión que se obtuvo de la aplicación que se desarrollo con un modelo inicial de la base de datos al que luego se le hicieron modificaciones, es que el esquema propuesto permite una adaptabilidad e independencia de la aplicación a cambios estructurales en la base de datos, tales como cambio de nombre de tablas, nombres de columnas y pasar las columna de una tabla a otra. El esquema es valioso puesto que no requiere ningún cambio en el código de la aplicación sino un cambio único en el mapa de relación de clases con el modelo de la base de datos.

6.2 Trabajo Futuro

En base a las experiencias obtenidas se recomienda trabajo futuro en las siguientes áreas:

- Proponer un mecanismo en el que se puedan desarrollar aplicaciones usando este esquema y diferente leguaje de implementación entre la aplicación y el framework de acceso a datos. Esto implica contar con un mecanismo para crear y manipular directamente objetos en tiempo de ejecución independiente del lenguaje de implementación. Este trabajo incluye la investigación y evaluación de las características de los lenguajes orientados a objeto y su capacidad tanto para accesar medios de almacenamiento externo, manipulación dinámica de objetos y mecanismos para especificar contratos en la aplicación.
- Un aspecto que representa una carencia grande de la arquitectura es que no cuenta con la capacidad de validar que los querys que se especifican sean válidos y que el mapa que relaciona a las clases con la estructura de la base de datos sea correcta. Debido a esto, la aplicación es propensa a tener errores en tiempo de ejecución. Una enorme mejora sería el proponer un esquema de validación tanto de querys como del mapa de relación entre clases y base de datos.
- La exploración de maneras alternativas de lograr independencia sobre la base de datos en el desarrollo de aplicaciones, tales como el uso de lenguajes de representación de datos como XML.
- Optimización de arquitectura: La arquitectura propuesta no tiene una optimización en su diseño interno para el tiempo de respuesta. Un diseño para optimizar recursos de la computadora debería incluir: manejos de un cache de querys y objetos, optimización de objetos solo cuando hayan sufrido cambios. También podría incluirse un optimizador de consultas.
- Implantación: El desempeño de las aplicaciones no solo depende de el software, sino de la plataforma de hardware en la que se ejecutan. Un esquema de computo distribuido podría mejorar

significativamente el desempeño de una aplicación construida con la arquitectura y el componente genérico. Si las operaciones del componente genérico se realizaran de manera distribuida, se podría lograr una mejora en desempeño. Sería muy valioso una propuesta en como dividir las funciones del componente genérico de acceso a datos en un ambiente distribuido.

Anexo

Implementación de las clases del componente genérico

Clase BaseDatos

```
/**
 * Clase Abstracta BaseDatos: Mantiene una
coneccion a una base de datos y mantiene los
metadatos de la misma
 * @author Andreas Molina
 * @since 26/9/99
 * @version 1.0
*/

import java.sql.*;

public abstract class  BaseDatos
{
 String strConexion; //Contiene el string de
conecxion a la base de datos sin usuario y
passwd
 String usuario;
 String password;
 Connection conexion;
 DatabaseMetaData metaData; //Clase de
metadatos. Se instancia al realizar la
coneccion en la clase concreta

 /**
  *Constructor vacio
 */
 public BaseDatos() {};

 /**
  * Constructor que inicializa el string de
conexion.
  * @param strConexion el string de conexion
(sin incluir usuario y password)
 */
 public BaseDatos(String strConexion)
 {
   this.strConexion= new String(strConexion);

 };

 /**
  *Constructor que inicializa tanto el string
de conexion como el usuario y password
  *@param strConexion el string de conexion
(sin usuario y password)
  *@param usuario usuario
  *@param password password
 */
 public BaseDatos(String strConexion,String
usuario, String password)
 {
  this.strConexion=new String(strConexion);
  this.usuario=new String(usuario);
  this.password=new String(password);
 }

 //Metodos para accesar las propiedades

 /**
  *Asigna el string de coneccion
  *@param strConexion el string de conexion
(sin uaurio y password)
 */
 public void setConexion(String strConexion)
 {this.strConexion=new String(strConexion);
 };


 /**
  *Asgina el valor del usuario
  *@param  usuario usuario
 */
 public void setUsuario(String usuario)
 {this.usuario=new String(usuario);
 }

 /**
  *Asigna el valor del password
  *@param password password
 */
 public void setPassword(String password)
 {this.password=new String(password);
 }

 /**
  Obtiene la conexion
  @return regresa la coneccion a la base de
datos
 */
 public  Connection getConexion()
 {
  return conexion;
 }

/**
 *Obtiene el usuario
 *@return regresa el usuario
*/
public String getUsuario()
{
 return new String(usuario);
}

/**
 *Obtiene los metadatos de la coneccion
 *@return regresa el objeto que contiene los
metadatos de la conexion
*/
public DatabaseMetaData getMetaData()
{
 return metaData;
}

 /**
  *Metodo abstracto que realiza la coneccion e
instancia el objeto de metadatos (dependiente
de la base de datos)
  *@exception SQLException si no se pude
conectar a la base de datos con los parametros
especificados
 */
 public abstract void conecta() throws
SQLException;


 public abstract ResultSet  ejecutaQuery(String
query);

 public abstract int getTipoDato(String schema,
String tabla, String campo);

 public abstract boolean esPrimaryKey(String
schema,String tabla,String campo);

 public abstract ResultSet
getColumnasTabla(String schema,String tabla);

 public abstract ResultSet
getForeignKeys(String schema,String tabla);

 public abstract int getValorSequencia(String
nombreSequencia);
}
```

Clase BaseDatosOracle

```
/**
 *Clase  BaseDatosOracle: Hereda de BaseDatos
implementa una conneccion y metadatos de una
base de datos Oracle.
 *Usa los drivers de jdbc de Oracle
 *@author Andreas Molina
 *@since 26/9/99
*/

import java.sql.*;
import
oracle.jdbc.driver.OracleDatabaseMetaData;
import oracle.jdbc.driver.OracleConnection;

public  class  BaseDatosOracle  extends
BaseDatos
{
```

```
 public BaseDatosOracle()
 {
  super();
 }

 /**
  *Constructor que inicializa el string de
conexion.
  *@param strConexion El string de conexion sin
el usuairo y el password
  */
 public BaseDatosOracle(String strConexion)
 {
   super(strConexion);
 };

 /**
  *Constructor que inicializa tanto el string
de conexion como el usuario y password
  *@param strConexion especifica el string de
conexion
  *@param usuario especifica el usuario con el
que se conectara a la base de datos
  *@param password especifica el password con
el que se conectara a la base de datos
  */
 public BaseDatosOracle(String
strConexion,String usuario, String password)
 {
   super(strConexion,usuario,password);
 }


/**
 *Se conecta a la base de datos con los
parametros especificados en las variables
miembro
 *@exception SQLException en caso de que la
conexion no se pueda realizar
*/
 public void conecta() throws SQLException
 {
  DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
  conexion = DriverManager.getConnection
(strConexion,usuario,password);
  metaData= new
OracleDatabaseMetaData((OracleConnection)
conexion);
 }

 public ResultSet ejecutaQuery(String query)
 {
  try
  {
  Statement stmnt = conexion.createStatement();
  ResultSet rs=stmnt.executeQuery(query);
  return rs;
  }
  catch (Exception e)
{System.out.println(e.toString());}
  return null;
 }

 public  int getTipoDato(String schema, String
tabla, String campo)
 {
  String schemaAux;
  int tipoDato=-1;
  if (schema==null)
   schemaAux=usuario;
  else
   schemaAux=schema;
  try{
  ResultSet
columnas=metaData.getColumns(null,schemaAux.toU
pperCase(),tabla.toUpperCase(),campo.toUpperCas
e());
  if (columnas.next())
    tipoDato=columnas.getInt("DATA_TYPE");
  }
  catch (Exception e)
{System.out.println(e.toString());};
  return tipoDato;
 }


 public boolean esPrimaryKey(String
schema,String tabla,String campo)
 {
  String schemaAux;
  boolean esPK=false;
  if (schema==null)
   schemaAux=usuario;
  else
   schemaAux=schema;
  try{
  ResultSet
columnas=metaData.getPrimaryKeys(null,schemaAux
.toUpperCase(),tabla.toUpperCase());
  while (columnas.next())
   {
     if
(columnas.getString("COLUMN_NAME").equals(campo
.toUpperCase()))
            return true;
   }

  }
  catch (Exception e)
{System.out.println(e.toString());};
  return esPK;

 }

 public ResultSet getColumnasTabla(String
schema,String tabla)
 {
  String schemaAux;
  if (schema==null)
   schemaAux=usuario;
  else
   schemaAux=schema;
  try {
  ResultSet
columnas=metaData.getColumns(null,schemaAux.toU
pperCase(),tabla.toUpperCase(),"%");
  return columnas;
  }
  catch (Exception e)
{System.out.println(e.toString());}
  return null;
 }


 public  ResultSet getForeignKeys(String
schema,String tabla)
 {
  String schemaAux;
  if (schema==null)
   schemaAux=usuario;
  else
   schemaAux=schema;
  try {
  ResultSet
llavesImportadas=metaData.getImportedKeys(null,
schemaAux.toUpperCase(),tabla.toUpperCase());
  return llavesImportadas;
  }
  catch (Exception e)
{System.out.println(e.toString());}
  return null;
 }

 public int getValorSequencia(String
nombreSequencia)
 {
  String sql;
  sql="Select "+nombreSequencia+".nextval from
dual";
  try
  {
   Statement stmnt =
conexion.createStatement();
   ResultSet rs=stmnt.executeQuery(sql);
   int valor=rs.getInt(1);
   return valor;
  }
  catch (Exception e)
{System.out.println(e.toString());}
 return -1;
 }

}
```

Clase Broker

```
import java.util.*;
import java.sql.*;
import java.io.*;
```

```
class Broker
{
  Hashtable mapaClases;
  String nombreArchivoMapa;
  CargaMapa cargaMapa;
  BaseDatos baseDatos;
  boolean trace=true;
  FileOutputStream archivoTrace=null;
  Broker()
  {baseDatos=new BaseDatosOracle();
   try {
   archivoTrace=new
FileOutputStream("trace.txt");
   } catch (Exception e)
{System.out.println(e.toString());}
  };

  Broker(String nombreArchivoMapa)
  {
   this.nombreArchivoMapa=new
String(nombreArchivoMapa);
   baseDatos=new BaseDatosOracle();
   try {
   archivoTrace=new
FileOutputStream("trace.txt");
   } catch (Exception e)
{System.out.println(e.toString());}
  }

  public Hashtable getMapaClases()
  {
   return mapaClases;
  }

  public MapaClase getMapaClase(String nombre)
  {

         return (MapaClase)
mapaClases.get(nombre);
  }

  public void cargaMapa()
  {
   cargaMapa = new
CargaMapa(this,nombreArchivoMapa);
   cargaMapa.carga();
   mapaClases=cargaMapa.getClases();

  }

  public void setConexion(String
strConexion,String usuario, String password)
  {
   baseDatos.setConexion(strConexion);
   baseDatos.setUsuario(usuario);
   baseDatos.setPassword(password);

  }

  public void setConexion(String strConexion)
  {
   baseDatos.setConexion(strConexion);
  }

  public void conecta() throws SQLException
  {
    baseDatos.conecta();
  }

  ResultSet ejecutaQuery(String query)
  {
   escribeTrace("--Ejecutando query: "+query);
   return baseDatos.ejecutaQuery(query);
  }

  public BaseDatos getBaseDatos()
  {
   return baseDatos;
  }

  public Object[] obtenObjetos(String
nombreClase)
  {
   MapaClase clase=getMapaClase(nombreClase);
   OperacionConsulta consulta=new
OperacionConsulta(this,clase);

   return consulta.getObjetos(null);
  }

  public Object[] obtenObjetos(String
nombreClase,Criterio criterio)
  {
   MapaClase clase=getMapaClase(nombreClase);
   OperacionConsulta consulta=new
OperacionConsulta(this,clase);
   escribeTrace("--Consultando objetos de:
"+nombreClase);
   return consulta.getObjetos(criterio);
  }

  public void actualizaObjeto(Object objeto)
  {
   if (objeto==null)
    return;
   Class clase=objeto.getClass();

   String nombreClase=clase.getName();

   MapaClase
mapaClase=getMapaClase(nombreClase);
   OperacionActualizaObjeto op=new
OperacionActualizaObjeto(this,objeto);

op.setModo(OperacionActualizaObjeto.RELACIONES)
;
   escribeTrace("--Actualizando objeto de:
"+nombreClase);
   op.ejecuta();
  }

 public void escribeTrace(String linea)
 {
  if (trace)
  try {
   archivoTrace.write((linea+"\n").getBytes());
  } catch (Exception e)
{System.out.println(e.toString());}
 }
}
```

Clase Criterio

```
/**
 *Clase criterio. Especifica un criterio para
seleccionar o actualizar informacion en base a
los atributos de un objeto
 *@author Andreas Molina
 *@since 02/10/99
*/

import java.util.*;

public abstract class  Criterio
{
 String nombre; //nombre del criterio, (asiciado
al operador logico)

 Criterio criterio1;
 Criterio criterio2;

 //Constructor vacio
 Criterio() {};



 /**
  *Copy constructor
  */
 Criterio(Criterio criterio)
 {
  nombre=criterio.getNombre();
  criterio1=criterio.getCriterio1();
  criterio2=criterio.getCriterio2();
 }


 Criterio(String nombre,Criterio criterio1,
Criterio criterio2)
 {
  this.nombre= new String(nombre);
  this.criterio1=criterio1;
  this.criterio2=criterio2;
 }

 public  Criterio getCriterio1()
 {
  return criterio1;
 };
```

```
 public  Criterio getCriterio2()
 {
  return criterio2;
 };

 /**
  *Constructor que incializa el nombre del
criterio
  *@param el nombre del criterio
  */
 Criterio(String nombre)
 {
  this.nombre=new String(nombre);
 }



/**
 *Asigna el nombre al criterio (and, or, etc)
 *@return el nombre del criterio
 */
 public String getNombre()
 {
   return new String(nombre);
 }

 /**
  *Obtiene el nombre del criterio
  *@param el nombre del criterio
  */
 public void setNombre(String nombre)
 {
  this.nombre=new String(nombre);
 }

 public void setCriterio(Criterio criterio1,
Criterio criterio2)
 {
  this.criterio1=criterio1;
  this.criterio2=criterio2;
 }

 public abstract Criterio addAnd(Criterio
criterio);

 public abstract Criterio addOr(Criterio
criterio);

 public abstract void imprime();

 public abstract Criterio[] serializa();

}

class CriterioSimple  extends Criterio
{
 String clase;
 String atributo;
 String operador;
 String valor;
 public static final String
CRITERIOSIMPLE="SIMPLE";

 CriterioSimple()
 {super(CRITERIOSIMPLE);
  setCriterio(null,null);
 }

 CriterioSimple(String clase,String
atributo,String operador,String valor)
 {super(CRITERIOSIMPLE);

  this.clase=new String(clase);
  this.atributo=new String(atributo);
  this.operador=new String(operador);
  if (valor!=null)
   this.valor=new String(valor);
  else
   this.valor=null;
 }

public String getClase()
{
 return new String(clase);
}

public String getAtributo()
{
 return new String(atributo);
}

public String getOperador()
{
 return new String(operador);
}

public String getValor()
{
 if (valor==null)
  return null;
 return new String(valor);
}

public Criterio addAnd(Criterio criterio)
{
 return new CriterioAnd (this,criterio);
}

public Criterio addOr(Criterio criterio)
{
 return new CriterioOr(this,criterio);
}

public void imprime()
{
 System.out.println("S "+clase+" "+atributo+"
"+" "+operador+" "+valor);
}

 public Criterio[] serializa()
 {
   Criterio criterio[]=new Criterio[1];
   criterio[0]=this;
   return criterio;
 }

}
```

Clase CriterioAnd

```
/**
 *Clase que reperesenta el operador logico AND
entre dos criteiros
 *@author Andreas Molina
 *@since 03/10/99
 */

class  CriterioAnd extends Criterio
{
 public static final String CRITERIOAND="AND";
 CriterioAnd()
 {
   super(CRITERIOAND);

 }

 CriterioAnd(Criterio criterio1, Criterio
criterio2)
 {
  super(CRITERIOAND,criterio1,criterio2);
 }

 public Criterio addAnd(Criterio criterio)
 {
  if (criterio1==null)
   {criterio1=criterio;
    return this;
   }
  if (criterio2==null)
   {criterio2=criterio;
    return this;
   }
  return new CriterioAnd(this,criterio);
 }

 public Criterio addOr(Criterio criterio)
 {
  return new CriterioOr(this,criterio);
 }

 public void imprime()
 {
  System.out.println("AND ");
  criterio1.imprime();
  criterio2.imprime();
 }


 public Criterio[] serializa()
 {
  Criterio c1[]=criterio1.serializa();
  Criterio c2[]=criterio2.serializa();
```

```
  Criterio c[]=new
Criterio[c1.length+c2.length];
  for (int i=0;i<c1.length;i++)
  {
   c[i]=c1[i];
  }
  for (int i=0;i<c2.length;i++)
  {
   c[i]=c2[i];
  }
  return c;
 }

}
```

Clase CriterioOr

```
/**
 *Clase que reperesenta el operador logico OR
entre dos criterios
 *@author Andreas Molina
 *@since 03/10/99
 */

class  CriterioOr extends Criterio
{
 public static final String CRITERIOOR="OR";
 CriterioOr()
 {
   super(CRITERIOOR);
 }

 CriterioOr(Criterio criterio1, Criterio
criterio2)
 {
  super(CRITERIOOR,criterio1,criterio2);
 }

 public Criterio addOr(Criterio criterio)
 {
  if (criterio1==null)
   {criterio1=criterio;
    return this;
   }
  if (criterio2==null)
   {criterio2=criterio;
    return this;
   }
  return new CriterioOr(this,criterio);
 }

 public Criterio addAnd(Criterio criterio)
 {
  return new CriterioAnd(this,criterio);
 }

 public void imprime()
 {
  System.out.println("OR");
  criterio1.imprime();
  criterio2.imprime();
 }

 public Criterio[] serializa()
 {
  Criterio c1[]=criterio1.serializa();
  Criterio c2[]=criterio2.serializa();
  Criterio c[]=new
Criterio[c1.length+c2.length];
  for (int i=0;i<c1.length;i++)
  {
   c[i]=c1[i];
  }
  for (int i=0;i<c2.length;i++)
  {
   c[i]=c2[i];
  }
  return c;
 }
}
```

Clase CriterioNot

```
/**
 *Clase que reperesenta el operador logico NOT
de un criterio
 *@author Andreas Molina
 *@since 03/10/99
 */

class  CriterioNot extends Criterio
{
 public static final String CRITERIONOT="NOT";
 CriterioNot()
 {
   super(CRITERIONOT);
 }


 CriterioNot(Criterio criterio)
 {
  super(CRITERIONOT,criterio,null);
 }

 public Criterio addAnd(Criterio criterio)
 {
   return new CriterioAnd(this,criterio);
 }

 public Criterio addOr(Criterio criterio)
 {
  return new CriterioOr(this,criterio);
 }

  public Criterio[] serializa()
  {
   return criterio1.serializa();
  }

  public void imprime()
  {
    System.out.println("NOT");
    criterio1.imprime();
  }

}

class CriterioRelacion extends Criterio
{

 String clase;
 String atributo; //campo que mantiene la
relacion

 public static final String
CRITERIORELACION="RELACION";

 CriterioRelacion()
 {
  super(CRITERIORELACION);
  setCriterio(null,null);
 }

 CriterioRelacion(String clase,String
atributo,Criterio criterioRelacion)
 {super(CRITERIORELACION);
  setCriterio(criterioRelacion,null);
  this.clase= new String(clase);
  this.atributo= new String(atributo);
 }

 public void setClase(String clase)
 {
   this.clase=new String(clase);
 }

 public void setAtributo(String atributo)
 {
   this.atributo=new String(atributo);
 }

 public void setAtributo(String clase,String
atributo)
 {
   setClase(clase);
   setAtributo(atributo);
 }

 public String getClase()
 {
  return new String(clase);
 }

 public String getAtributo()
 {
   return new String(atributo);
 }

 public Criterio addAnd(Criterio criterio)
 {
  return new CriterioAnd(this,criterio);
```

```
 }
 public Criterio addOr(Criterio criterio)
 {
  return new CriterioOr(this,criterio);
 }

 public void imprime()
 {
  System.out.println("R "+clase+ " "+atributo);
  criterio1.imprime();
 }

  public Criterio[] serializa()
  {
   return criterio1.serializa();
  }

}
```

Clase FabricaObjeto

```
import java.util.*;
import java.sql.*;
import java.lang.reflect.*;


class  FabricaObjeto
{
  MapaClase mapaClase;

  FabricaObjeto()
  {
  };

  FabricaObjeto(MapaClase mapaClase)
  {this.mapaClase=mapaClase;
  }

 public void setClase(MapaClase clase)
 {
  this.mapaClase=clase;
 }


 public boolean tieneConstructorVacio(Class
clase)
 {
  return true;
 }

 public Object creaObjeto(ResultSet rs)
 {
  Object objeto=null;
  Class c=null;
  try
  {
  c=Class.forName(mapaClase.getNombre());


  if (tieneConstructorVacio(c))
  {
        objeto=c.newInstance();
  }
  }
  catch (Exception e)
{System.out.println(e.toString());}

  return objeto;
 }


 public static String getTipoDato(Object
objeto,String nombreAtributo)
 {
   try
   {
   Class clase=objeto.getClass();
   Field campo=clase.getField(nombreAtributo);
   Class claseTipo = campo.getType();
   return claseTipo.getName();
   }
   catch (Exception e)
{System.out.println(e.toString());}
  return null;
 }

 public static Object[] getValores(Object
objeto, String nombreAtributo)
 {
 try
 {
   Class clase=objeto.getClass();
   Field campo=clase.getField(nombreAtributo);
   Class claseTipo = campo.getType();
   Object so=campo.get(objeto);
   Object sarr[]=(Object[])so;
   return sarr;
 }
 catch (Exception e)
{System.out.println(e.toString());}
 return null;
 }


 public static Object getValor(Object
objeto,String nombreAtributo)
 {
  try
  {
   String
tipo=getTipoDato(objeto,nombreAtributo);
   Class clase=objeto.getClass();
   Field campo=clase.getField(nombreAtributo);
   if(tipo.equals("boolean"))
   {
    boolean valor=campo.getBoolean(objeto);
    Boolean b= new Boolean(valor);
         return b.toString();
   }
   else if(tipo.equals("byte"))
   {byte valor=campo.getByte(objeto);
    return Byte.toString(valor);
   }
   else if(tipo.equals("char"))
   {char valor=(char) campo.getInt(objeto);
    return Integer.toString((int) valor);
   }
   else if(tipo.equals("double"))
   {double valor=campo.getDouble(objeto);
    return Double.toString(valor);
   }
   else if(tipo.equals("float"))
   {float valor=campo.getFloat(objeto);
    return Float.toString(valor);
   }
   else if(tipo.equals("int"))
   {int valor=campo.getInt(objeto);
    return Integer.toString(valor);
   }
   else if(tipo.equals("long"))
   {long valor=campo.getLong(objeto);
    return Long.toString(valor);

   }
   else if(tipo.equals("short"))
   {short valor=campo.getShort(objeto);
    return Short.toString(valor);
   }
   return campo.get(objeto);
  }
  catch (Exception e)
{System.out.println(e.toString());}
  return null;
 }


 public static void setValor(Object
objeto,String nombreAtributo,Object valor)
 {
  try
  {Class clase=objeto.getClass();
   Field campo=clase.getField(nombreAtributo);
   campo.set(objeto,valor);
  }
  catch (Exception e)
{System.out.println(e.toString());}
 }

 public void setValor(Object objeto,String
nombreAtributo,String nombreTabla,String
nombreCampo,ResultSet rs)
 {

   try
   {
   Class clase=objeto.getClass();
   Field campo=clase.getField(nombreAtributo);
   Class claseTipo = campo.getType();
   String tipo = claseTipo.getName();//obtiene
el tipo de dato del atributo

   if(tipo.equals("boolean"))
   {boolean valor=rs.getBoolean(nombreCampo);
```

```
    campo.setBoolean(objeto,valor);
   }
   else if(tipo.equals("byte"))
   {byte valor=rs.getByte(nombreCampo);
    campo.setByte(objeto,valor);
   }
   else if(tipo.equals("char"))
   {char valor=(char) rs.getInt(nombreCampo);
    campo.setChar(objeto,valor);
   }
   else if(tipo.equals("double"))
   {double valor=rs.getDouble(nombreCampo);
    campo.setDouble(objeto,valor);
   }
   else if(tipo.equals("float"))
   {float valor=rs.getFloat(nombreCampo);
    campo.setFloat(objeto,valor);
   }
   else if(tipo.equals("int"))
   {int valor=rs.getInt(nombreCampo);
    campo.setInt(objeto,valor);
   }
   else if(tipo.equals("long"))
   {long valor=rs.getLong(nombreCampo);
    campo.setLong(objeto,valor);
   }
   else if(tipo.equals("short"))
   {short valor=rs.getShort(nombreCampo);
    campo.setShort(objeto,valor);
   }
   else if (tipo.equals("String"))
   {String valor=rs.getString(nombreCampo);
    campo.set(objeto, valor);
   }
   else
   {
    Object valor=rs.getObject(nombreCampo);
    campo.set(objeto, valor);
   }
   }
   catch (Exception e)
{System.out.println(e.toString());}
 }

 public void cargaDatos(Object objeto,ResultSet
rs)
 {
  Class clase=objeto.getClass();
  Field campo=null;
  Object valor=null;
//  ResultSetMetaData
rsMetaData=rs.getMetaData();
  for (Enumeration
e=mapaClase.getAtributos().elements();
e.hasMoreElements(); )
  {MapaAtributo ma=(MapaAtributo)
e.nextElement();
   if(!ma.esRelacion())
   {
   String nombreAtributo=ma.getNombre();
   MapaColumna mc=ma.getColumna();
   MapaTabla mt=mc.getTabla();
   String nombreTabla=mt.getNombre();
   String nombreCampo=mc.getNombre();


   try
   {
        campo=clase.getField(nombreAtributo);

setValor(objeto,nombreAtributo,nombreTabla,nomb
reCampo,rs);
   }
   catch (Exception ex)
{System.out.println(ex.toString());}



        }
  }
 }

 public Object getObjeto(ResultSet rs)
 {
   Object objeto=creaObjeto(rs);
   cargaDatos(objeto,rs);
   return objeto;
 }

}
```

Clase Join

```
class Join
{
 MapaColumna c1;
 MapaColumna c2;

 Join(MapaColumna c1, MapaColumna c2)
 {
  this.c1=c1;
  this.c2=c2;
 }

 public MapaColumna getColumna1()
 {
  return c1;
 }

 public MapaColumna getColumna2()
 {
  return c2;
 }

 public String getSql()
 {String sql;

sql=c1.getTabla().getNombre()+"."+c1.getNombre(
)+"="+c2.getTabla().getNombre()+"."+c2.getNombr
e();
  return sql;
 }

}
```

Clase MapaClase

```
/**
 *Clase MapaClase, mapea los atributos de una
clase con un atributo o relacion en una base de
datos relacional
 *@author Andreas Molina
 *@since 5/10/99
 */

import java.util.*;

class  MapaClase
{
  String nombre; //El nombre de la clase

  MapaClase padre; //Apuntador a la infomracion
sobre la clase de la cual hereda
  Hashtable atributos; //Coleccion de los mapas
de los atributos


  MapaClase()
  {
   atributos=new Hashtable(10);
  };

  /**
   *Constructor que inicializa el nombre de la
clase
   */
  MapaClase(String nombre)
  {
    this.nombre=new String(nombre);
    atributos=new Hashtable(10);

  }

  /**
   *Obtiene el nombre de la clase
   *@return el nombre de la clase
   */
  public String getNombre()
  {
    return new String(nombre);
  }

  /**
   *Asigna el nombre de la clase
   *@param el nombre de la clase
   */
 public void setNombre(String nombre)
 {
  this.nombre=new String(nombre);
 }

 /**
```

```
  *Asigna el apuntador a la informacion de la
clase padre
  *@return la informacion de la clase padre
  */
 public MapaClase getPadre()
 {
   return padre;
 }

 /**
  *Metodo que asigna la informacin de la clase
padre a la calse.  El metodo es protected por
que
  *no debe ser posible cambiar la informacion
de la super clase desde la aplicacion. Esta
informacion
  *se obtien usando el reflection api.
  */
 protected void setPadre(MapaClase padre)
 {
   this.padre=padre;
 }


 public void agregaAtributo(MapaAtributo
atributo)
 {

atributos.put(atributo.getNombre(),atributo);
 }

/**
 *Obtiene la coleccion con la informacion de
los atributos
 */
public Hashtable getAtributos()
{
  return atributos;
}

public MapaAtributo getAtributo(String nombre)
{
 return (MapaAtributo) atributos.get(nombre);
}

public boolean esUniforme()
{String bd;
 Enumeration e = atributos.elements();
 MapaAtributo ma=(MapaAtributo)
e.nextElement();
 bd=ma.getBaseDatos();
 while ( e.hasMoreElements() )
    {
            if(!ma.esUniforme())
             return false;
        if (!ma.getBaseDatos().equals(bd))
          return false;
            bd=ma.getBaseDatos();
            ma=(MapaAtributo)e.nextElement();
         }
return true;
}



public void imprimeAtributos() {
System.out.println("Atributos de: "+nombre);
     for (Enumeration e = atributos.elements()
; e.hasMoreElements() ;) {
          MapaAtributo ma=(MapaAtributo)
e.nextElement();
          System.out.println(ma.getNombre());
         }
}


public MapaRelacion getMapaRelacion(String
atributoRelacion) {
  MapaAtributo
ma=getAtributo(atributoRelacion);
  return ma.getMapaRelacion();
}


public Object[] getAtributosId()
{
  ArrayList a=new ArrayList(1);
  for (Enumeration e = atributos.elements() ;
e.hasMoreElements() ;) {
        MapaAtributo ma=(MapaAtributo)
e.nextElement();
        if (ma.esId())
        {
                 a.add(ma);
        }
 }
return a.toArray();
}

}
```

Clase MapaAtributo

```
/**
 *Clase mapa atributo tiene la informacion para
mapear a un atributo de una clase con su
correspondiente elemento en la base de datos
 *@author Andreas Molina
 *@since 5/10/99
 */

import java.util.*;

class  MapaAtributo
{
  String nombre; //nombre del atributo
  MapaClase mapaClase; //apuntador al mapa de
clase al cual pertenece
  String secuencia; //secuencia usada para
genera valores en caso de que sean
autogenerados
                    //si es no nulo indica que
se generan por medio de la secuencia
especificada

  MapaRelacion mapaRelacion; //indica si el
atributo representa una relacion

  Hashtable columnas; //coleccion de columnas a
las que es mapeado el atributo



  //Constructor vacio
  MapaAtributo()
  {
   columnas=new Hashtable(1);
  };

  /**
   *Constructor que inicializa el nombre del
atributo
   */
  MapaAtributo(String nombre)
  {
   this.nombre=new String(nombre);
   columnas=new Hashtable(1);
  }



  /**
   *Constructor que inicializa el mapa de clase
al cual pertenece el atributo
   */
  MapaAtributo(MapaClase mapaClase)
  {
   columnas=new Hashtable(1);
   this.mapaClase=mapaClase;
  }


  /**
   *Constructor que inicializa el nombre del
atriburo y su mapaClase
   */
   MapaAtributo(String nombre,MapaClase
mapaClase)
    {
     columnas=new Hashtable(1);
          this.nombre=new String(nombre);
          this.mapaClase=mapaClase;
    }

  /**
   *Constructor que inicializa el nombre del
atributo, mapaClase y un mapa de la relacion
que representa el atributo
   */
   MapaAtributo(String nombre,MapaClase
mapaClase,MapaRelacion mapaRelacion)
    {
     columnas=new Hashtable(1);
```

```
        this.nombre=new String(nombre);
        this.mapaClase=mapaClase;
        this.mapaRelacion=mapaRelacion;
  }


 /**
  *Obtiene el nombre del atributo que mapea
  *@return el nombre del atributo
  */
  public String getNombre()
  {
    return new String(nombre);
  }

  /**
   *Asgina el nombre del atributo que mapea
   *@param nombre es el nombre del atriburo
   */
  public void setNombre(String nombre)
  {
    this.nombre=new String(nombre);
  }

 /**
  *Obtiene el mapa de la clase a la cual
pertenece el atributo
  *@return el mapa de clase
  */
 public MapaClase getMapaClase()
 {
   return mapaClase;
 }

 /**
  *Asigna el mapa de clase al atributo
  *@param el mada de clase
  */
 public void setMapaClase(MapaClase mapaClase)
 {
   this.mapaClase=mapaClase;
 }


 /**
  *Obtiene el mapa de la relacion al que
representa el atibuto
  *@return el mapa de la relacion que
representa el atributo
  */
public MapaRelacion getMapaRelacion()
{
   return mapaRelacion;
}


/**
 *Asigan el mapa de la relacion al objeto
 *@param el mapa de la relacion
 */
 public void setMapaRelacion(MapaRelacion
mapaRelacion)
 {
  this.mapaRelacion=mapaRelacion;
 }

 /**
  *Determina si el atributo representa una
relacion
  *@return true si el atributo representa una
relacion
  */
 public boolean esRelacion()
 {
   if (mapaRelacion==null)
     return false;
   return true;
 }

 public void agregaCampo(MapaColumna campo)
 {

columnas.put(campo.getTabla().getBaseDatos()+ca
mpo.getTabla().getNombre()+campo.getNombre(),ca
mpo);
 }

 public Hashtable getColumnas()
 {
   return columnas;
 }
 public MapaColumna getColumna(String
baseDatos,String tabla, String campo)
 {
   return (MapaColumna)
columnas.get(baseDatos+tabla+campo);
 }

 public boolean esUniforme()
 {
   if (esRelacion())
    return mapaRelacion.esUniforme();
   if (columnas.size()>1)
    return false;
   return true;

 }

 public MapaColumna getColumna()
 {
   Enumeration e= columnas.elements();
   if (e==null)
   {
    System.out.println("es null el e");
   }
  MapaColumna mc=(MapaColumna) e.nextElement();

  return mc;
 }

 public MapaTabla getTabla()
 {
  MapaColumna mc=getColumna();
  return mc.getTabla();
 }

 public void setSecuencia(String
nombreSecuencia)
 {
  if (nombreSecuencia!=null)
   this.secuencia=new String(nombreSecuencia);
 }

 public boolean esAutoGenerado()
 {
  return (secuencia!=null);
 }

 public String getSecuencia()
 {
  if (secuencia==null)
   return null;

  return new String(secuencia);
 }

 public String getBaseDatos()
 {
   if (esRelacion())
   {
    return mapaRelacion.getBaseDatos();
   }
   return getTabla().getBaseDatos();
 }

 public boolean esId()
 {
   if (esRelacion())
    return false;

   if (getColumna().esPrimaryKey())
    return true;

  return false;
 }

}
```

Clase MapaColumna

```
/**
 *MapaColumna representa la informacion de una
columna en una tabla
 *@author Andreas Molina
 *@since 5/10/99
 */
class  MapaColumna
{
  String nombre;
  MapaTabla tabla; //apuntador a la tabla a la
que pertenece
  int tipoDato; //tipo de dato de sql
```

```
  boolean esPrimaryKey=false;

  //Constructor vacio
  MapaColumna() {};

  /**
   *Asigna el nombre de la columna
   *@param el nombre de la columna
   */
  MapaColumna(String nombre)
  {
   this.nombre=new String(nombre);
  }

  /**
   *Asigna la tabal a la que pertenece la
columna
   *@param la tabla a la que pertenece la
clumna
   */
  MapaColumna(MapaTabla tabla)
  {
   this.tabla=tabla;
  }

  /**
   *Asigan la columna y la tabla a la que
pertenece la columna
   *@param el nombre e la columna
   *@param la tabla a la que pertenece al
columna
   */
  MapaColumna(String nombre,MapaTabla tabla)
  {
    this.nombre=new String(nombre);
        this.tabla=tabla;
  }

 MapaTabla getTabla()
 {
   return tabla;
 }

 String getNombre()
 {
   return new String(nombre);
 }

 public void setTipoDato(int tipoDato)
 {
  this.tipoDato=tipoDato;
 }

 public void setEsPrimaryKey(boolean
esPrimaryKey)
 {
  this.esPrimaryKey=esPrimaryKey;
 }

 public boolean esPrimaryKey()
 {
   return esPrimaryKey;
 }

 public int getTipoDato()
 {
  return tipoDato;
 }

}
```

Clase MapaTabla

```
/**
 *MapaTabla representa la informacion de una
tabla en la base de datos
 *@author Andreas Molina
 *@since 5/10/99
 */
class MapaTabla
{
  String descriptorBaseDatos; //url de la
coneccion a la base de datos
  String nombre; //nombre de la tabla

  //Constructor vacio
  MapaTabla() {};

  /**
   *Constructor que asigna el nombre del
schema, la tabla y la base de datos a la que
pertenece
   */
  MapaTabla(String descriptorBaseDatos,String
nombre)
  {

this.descriptorBaseDatos=descriptorBaseDatos;
   this.nombre=new String(nombre);
  }


  /**
   *Obtiene el nombre de la tabla
   *@return el nombre de la tabla
   */
  public String getNombre()
  {
    return new String(nombre);
  }

 /**
  *Obtiene la base de datos a la que pertenece
la tabla
  *@return el apuntador a la base de datos
  */
  public String getBaseDatos()
  {
    return new String(descriptorBaseDatos);
  }


   /**
   *Asigna el nombre de la tabla
   *@param el nombre de la tabla
   */
   public void setTabla(String nombre)
   {
     this.nombre=new String(nombre);
   }

   /**
    *Asgina la base de datos
        *@param el aputnador a la base de
datos
        */
        public void setBaseDatos(String
descriptorBaseDatos)
        {

this.descriptorBaseDatos=descriptorBaseDatos;
        }

   public boolean equals(MapaTabla tabla)
   {
    if
(tabla.getNombre().equalsIgnoreCase(nombre))
         return true;
        return false;
   }
}
```

Clase Operacion

```
/**
*Clase Abstracta Operacion. Representa una
operacion que se realiza con un objeto
*@author Andreas Molina
*@since 29/9/99
*/
public abstract class Operacion
{
 String nombre; //El nombre de la operacion
identifica al tipo de operacion: guardarObjeto,
borrarObjeto, consultaObjeto, etc.
 Broker broker;

 //Constructor vacio
 Operacion(Broker broker)
 {this.broker=broker;
 }

 //Constructor que asigna el nombre de la
operacion
 Operacion(Broker broker,String nombre)
 {
  this.broker=broker;
  this.nombre=new String(nombre);
 }
```

```
 /**
  *Asigna el nombre de la operacion
  *@param nombre asigna el nombre de la
operacion
 */
 public void setNombre(String nombre)
 {
  this.nombre=new String(nombre);
 }
 /**
  *Obtiene el nombre de la operacion
  *@return el nombre de la operacion
 */
 public String getNombre()
 {
   return new String(nombre);
 }
}
```

Clase OperacionAcutaliza

```
/**
 *Clase OperacionActualiza, representa una
operacion que realiza
 *alguna clase de actualizacion
(actualizarObjeto, borrarObjeto,etc)
 *agrega los metodos de commit y rollback
*/
public abstract class OperacionActualiza
extends Operacion
{
 //Constructor vacio.
 OperacionActualiza(Broker broker)
 {super(broker);
 };
 //Actualiza el nombre de la operacion. Pasa la
llamada a la superclase
 OperacionActualiza(Broker broker,String
nombre)
 {
   super(broker,nombre);
 };
 public abstract void commit();
 public abstract void rollback();

 /**
 *Metodo que deben implementar todas las
Operaciones. Este metdodo es el que ejecuta la
operacion
 */
 public abstract boolean ejecuta();

}
```

Clase OperacionActualizaObjeto

```
import java.util.*;
import java.sql.*;

class OperacionActualizaObjeto  extends
OperacionActualiza
{
 public static final String
ACTUALIZA_OBJETO="ACTUALIZAOBJETO";
 public static final String
ATRIBUTOS="ATRIBUTOS";
 public static final String
RELACIONES="RELACION";
 public static final String
RELACIONES_COMPLETAS="RELACIONES_COMPLETAS";
 Object objeto=null; //El objeto a actualizar;
 Class clase=null;
 String nombreClase=null;
 MapaClase mapaClase=null;
 Hashtable tablas;
 ArrayList valores;

 Hashtable valoresRelaciones;
 ArrayList relaciones;
 Criterio criterioLlave;
 String modo=null;

 OperacionActualizaObjeto(Broker broker)
 {
  super(broker,ACTUALIZA_OBJETO);
  tablas= new Hashtable(1);
  valores=new ArrayList(1);
  valoresRelaciones= new Hashtable(1);
  relaciones=new ArrayList(1);
 }

 OperacionActualizaObjeto(Broker broker,Object
objeto)
 {
  super(broker,ACTUALIZA_OBJETO);
  tablas= new Hashtable(1);
  valores=new ArrayList(1);
  relaciones=new ArrayList(1);
  valoresRelaciones= new Hashtable(1);
  this.objeto=objeto;
  clase=objeto.getClass();
  nombreClase=clase.getName();
  mapaClase=broker.getMapaClase(nombreClase);
 }

 public void commit() {};
 public void rollback() {};
 public boolean ejecuta()
 {
  actualizaObjeto();
  return true;
 }

 public MapaRelacion
obtenMapaRelacion(CriterioRelacion cr)
 {
  MapaClase
mc=broker.getMapaClase(cr.getClase());
  MapaAtributo
ma=mc.getAtributo(cr.getAtributo());
  if (ma.esRelacion())
  {
    return ma.getMapaRelacion();
  }
  return null;
 }

 public boolean columnasEnMapa(ResultSet
columnas,MapaRelacion mr)
 {

  try {
  while (columnas.next())
  {
   String
nombreColumna=columnas.getString("COLUMN_NAME")
;
   String
nombreTabla=columnas.getString("TABLE_NAME");

   if
(!mr.tieneColumna(nombreTabla,nombreColumna))
    return false;
  }
  }
  catch (Exception e)
{System.out.println(e.toString());}
  return true;
 }

 public String
obtenIdHashRelacion(CriterioRelacion cr)
 {
  return cr.getAtributo();
 }

 public ArrayList
getValoresRelacion(MapaRelacion
mr,CriterioRelacion relacion,MapaTabla tabla)
 {ArrayList valoresTabla=new ArrayList(1);


  Criterio
criteriosSimple[]=relacion.serializa();

  String nombreTabla=tabla.getNombre();

  for (int i=0;i<criteriosSimple.length;i++)
  {CriterioSimple cs=(CriterioSimple)
criteriosSimple[i];
   String nombreClase=cs.getClase();
   String nombreAtributo=cs.getAtributo();

   MapaAtributo
ma=mr.getAtributo(nombreClase,nombreAtributo);
   MapaTabla mt=ma.getTabla();
   if (tabla.equals(mt))
          valoresTabla.add(cs);
  }
  return valoresTabla;
 }
```

```
 public void actualizaObjeto()
 {

criterioLlave=OperacionConsulta.formaCriterioLl
ave(objeto,mapaClase);
   formaValores();


 //falta ordenar a las tablas de acuerdo a las
restricciones
   for (Enumeration
e=tablas.elements();e.hasMoreElements();)
   {
     MapaTabla tabla=(MapaTabla)
e.nextElement();

actualizaObjetoEnTabla(tabla,criterioLlave);
   }


 //actualiza las relaciones
  for (Enumeration
e=valoresRelaciones.elements();e.hasMoreElement
s();)
  {CriterioRelacion cr;
   ArrayList criteriosRelacion=(ArrayList)
e.nextElement();
   cr=(CriterioRelacion)
criteriosRelacion.get(0);
   borraRelacion(cr);
   for (int i=0;i<criteriosRelacion.size();i++)
   {cr=(CriterioRelacion)
criteriosRelacion.get(i);
    actualizaRelacion(cr);
   }
  }
 }
 public void borraRelacion(CriterioRelacion cr)
 {
   MapaRelacion mr=obtenMapaRelacion(cr);
   Object tablasRelacion[]=mr.obtenTablas();
   for (int it=0;it<tablasRelacion.length;it++)
   {

    MapaTabla tabla=(MapaTabla)
tablasRelacion[it];

         //compara el mapa de la relacion con
los campos de la tabla
    ResultSet
columnas=broker.getBaseDatos().getColumnasTabla
(null,tabla.getNombre());

         ArrayList
valoresTabla=getValoresRelacion(mr,cr,tabla);

    String sql=null;

         if (columnasEnMapa(columnas,mr))
         {//la tabla corresponde completamente
a la relacion  hacer insert
          SqlDelete sqlDelete=new
SqlDelete(broker,tabla,mr,criterioLlave);
      sqlDelete.formaSql();
          sql=sqlDelete.getSql();
         }
         else
         {//se debe hacer un update
          Criterio
criterioUpdate=criterioLlave;
          SqlUpdate sqlUpdate=new
SqlUpdate(broker,tabla,valoresTabla,mr,criterio
Update,true);
          sqlUpdate.formaSql();
          sql=sqlUpdate.getSql();
         }
     //System.out.println(sql);
     broker.ejecutaQuery(sql);

   }
  }


 public void actualizaRelacion(CriterioRelacion
cr)
 {
   MapaRelacion mr=obtenMapaRelacion(cr);
   Object tablasRelacion[]=mr.obtenTablas();
   for (int it=0;it<tablasRelacion.length;it++)
   {
    MapaTabla tabla=(MapaTabla)
tablasRelacion[it];

         //compara el mapa de la relacion con
los campos de la tabla
    ResultSet
columnas=broker.getBaseDatos().getColumnasTabla
(null,tabla.getNombre());

         ArrayList
valoresTabla=getValoresRelacion(mr,cr,tabla);

         String sql=null;

         if (columnasEnMapa(columnas,mr))
         {//la tabla corresponde completamente
a la relacion  hacer insert
     Criterio
criterios[]=criterioLlave.serializa();
          for (int i=0;i<criterios.length;i++)
          {
            valoresTabla.add(criterios[i]);
          }
          SqlInsert sqlInsert=new
SqlInsert(broker,tabla,valoresTabla,mr);
     sqlInsert.formaSql();
          sql=sqlInsert.getSql();
         }
         else
         {//se debe hacer un update
          Criterio
criterioUpdate=criterioLlave;
          SqlUpdate sqlUpdate=new
SqlUpdate(broker,tabla,valoresTabla,mr,criterio
Update);
          sqlUpdate.formaSql();
          sql=sqlUpdate.getSql();
         }
//    System.out.println(sql);
    broker.ejecutaQuery(sql);
   }
  }


 public void actualizaObjetoEnTabla(MapaTabla
tabla,Criterio criterio)
 {
  //Verifica si los datos se encuentran en la
tabla, para determinar si se hace una
actualizacion o consulta
  String sql=null;
  OperacionConsulta consulta=new
OperacionConsulta(broker,mapaClase);
  int
numRegistros=consulta.getNumRegistros(tabla,obj
eto);
  if (numRegistros>0) //hace un update
  {
   SqlUpdate sqlUpdate=new
SqlUpdate(broker,tabla,valores,criterio);
   sqlUpdate.formaSql();
   sql=sqlUpdate.getSql();
  }
  else
  {
   SqlInsert sqlInsert=new
SqlInsert(broker,tabla,valores);
   sqlInsert.formaSql();
   sql=sqlInsert.getSql();
  }

  broker.ejecutaQuery(sql);
 }

 public String obtenIdHashTabla(MapaTabla
tabla)
 {
  return tabla.getNombre();
 }


 public void formaValores()
 {
  CriterioSimple criterio=null;
  String valor=null;
  MapaTabla tabla=null;
```

```
  for (Enumeration
e=mapaClase.getAtributos().elements();e.hasMore
Elements();)
   {

   MapaAtributo ma=(MapaAtributo)
e.nextElement();
   if (!ma.esRelacion())
   {
   tabla=ma.getColumna().getTabla();
   tablas.put(obtenIdHashTabla(tabla),tabla);
   valor=(String)
FabricaObjeto.getValor(objeto,ma.getNombre());
   if (ma.esAutoGenerado()&&valor==null)
   {

valor=""+broker.getBaseDatos().getValorSequenci
a(ma.getSecuencia());
   }
   criterio= new
CriterioSimple(nombreClase,ma.getNombre(),"=",v
alor);
   valores.add(criterio);
   }
   else
   {
    if
(modo.equals(RELACIONES)||modo.equals(RELACIONE
S_COMPLETAS))
          {


          MapaRelacion mr=ma.getMapaRelacion();
          MapaClase claseRel=mr.getClase2();

    //debe de ir en un ciclo si la cardinalidad
es de mayor de 1
          if (mr.esCardinalidadUno())
          {
          //formar el criterio relacion y
agregarlo
           Object
objetoValor=FabricaObjeto.getValor(objeto,mr.ge
tAtributo().getNombre());
      if (objetoValor!=null)
           {
           Criterio
criterioLlave=OperacionConsulta.formaCriterioLl
ave(objetoValor,claseRel);
           CriterioRelacion cr= new
CriterioRelacion(mapaClase.getNombre(),mr.getAt
ributo().getNombre(),criterioLlave);
      agregaRelacion(cr);
//       relaciones.add(cr);
           }
          }
     else //cardinalidad muchos
          {

//
System.out.println(mr.getAtributo().getNombre()
);


           Object
objetos[]=FabricaObjeto.getValores(objeto,mr.ge
tAtributo().getNombre());

           if (objetos!=null)
           {
             for (int i=0;i<objetos.length;i++)
             {Object objetoValor=objetos[i];
              Criterio
criterioLlave=OperacionConsulta.formaCriterioLl
ave(objetoValor,claseRel);

                     CriterioRelacion cr= new
CriterioRelacion(mapaClase.getNombre(),mr.getAt
ributo().getNombre(),criterioLlave);

                     agregaRelacion(cr);
//                   relaciones.add(cr);
       }
           }
          }
      }
     }
    }
  }

 public void agregaRelacion(CriterioRelacion
cr)
 {
  String idHash=obtenIdHashRelacion(cr);
  ArrayList arregloCriterios=(ArrayList)
valoresRelaciones.get(idHash);
  if (arregloCriterios==null) //no existe el
criterio, hay que creearlo
   {arregloCriterios=new ArrayList(1);

valoresRelaciones.put(idHash,arregloCriterios);
  }
 arregloCriterios.add(cr);
 }


 public ArrayList getValores()
 {
  return valores;
 }

 public void setPropiedades()
 {
  clase=objeto.getClass();
  nombreClase=clase.getName();
  mapaClase=broker.getMapaClase(nombreClase);
 }

 public void setObjeto(Object objeto)
 {
  this.objeto=objeto;
 }

 public void setModo(String modo)
 {
  this.modo=new String(modo);
 }

 public String getModo()
 {
  return new String(modo);
 }
}
```

Clase OperacionConsulta

```
import java.util.*;
import java.sql.*;

class OperacionConsulta extends Operacion
{
    boolean recursivo=true;
    MapaClase mapaClase=null;

         OperacionConsulta(Broker broker)
         { super(broker,"Consulta");
         }

    OperacionConsulta(Broker broker,MapaClase
mapaClase)
         {
           super(broker,"Consulta");
           this.mapaClase=mapaClase;
         }

    OperacionConsulta(Broker broker,MapaClase
mapaClase,boolean recursivo)
         {
           super(broker,"Consulta");
           this.mapaClase=mapaClase;
      this.recursivo=recursivo;
         }

    public  Criterio formaCriterio(Object
objeto,MapaRelacion mapaRelacion)
         {Criterio criterio=null;
     Criterio criterioAux=null;
          MapaClase
claseRel=mapaRelacion.getClase1();
          String
nombreClaseRel=claseRel.getNombre();
     String nombreClase=mapaClase.getNombre();
          String valor;


criterioAux=formaCriterioLlave(objeto,claseRel)
;
          criterio=new
CriterioRelacion(nombreClase,mapaRelacion.getAt
ributo().getNombre(),criterioAux);
          return criterio;
         }
```

```
        public void asignaRelacion(Object
objeto,String nombreAtributo,boolean
esCardinalidadUno,Object [] objetos)
        {

      if (objetos==null)
          return;
          if
(esCardinalidadUno&&objetos.length>0)
          {

FabricaObjeto.setValor(objeto,nombreAtributo,ob
jetos[0]);
          }
      else
          {

FabricaObjeto.setValor(objeto,nombreAtributo,ob
jetos);
          }
        }

    public static Criterio
formaCriterioLlave(Object objeto,MapaClase
mapaClase)
        {
     Criterio criterio=null;
     Criterio criterioAux=null;
     String valor=null;
         String
nombreClase=mapaClase.getNombre();
         Object[]
atributosId=mapaClase.getAtributosId();
         for (int i=0;i<atributosId.length
;i++)
         {MapaAtributo ma=(MapaAtributo)
atributosId[i];
          valor=(String)
FabricaObjeto.getValor(objeto,ma.getNombre());
          criterioAux= new
CriterioSimple(nombreClase,ma.getNombre(),"=",v
alor);
      if (i>0)

criterio=criterio.addAnd(criterioAux);
          else
           criterio=criterioAux;
         }
         return criterio;
        }

        public Object[] getObjetos(Criterio
criterio)
        {
     String strSql;
     broker.escribeTrace("Obteniendo objetos de
: "+mapaClase.getNombre());
         ArrayList arreglo=new ArrayList(1);
         SqlSelect sql=new
SqlSelect(broker,mapaClase,criterio);
         sql.formaSql();
         strSql=sql.getSql();
//      System.out.println(strSql);
         ResultSet
rs=broker.ejecutaQuery(strSql);
         FabricaObjeto f= new
FabricaObjeto(mapaClase);

         try
         {
         while(rs.next())
         {
      broker.escribeTrace("Creando objeto de :
" +mapaClase.getNombre());
           Object objeto=f.getObjeto(rs);
           if(recursivo)
           {
        for (Enumeration
e=mapaClase.getAtributos().elements();e.hasMore
Elements() ; )
        {MapaAtributo ma=(MapaAtributo)
e.nextElement();
                  if (ma.esRelacion())
                  {

                   MapaRelacion
mr=ma.getMapaRelacion();
                   Criterio criterioRel=
formaCriterio(objeto,mr);
                   MapaClase
claseObtener=broker.getMapaClase(mr.getClase2()
.getNombre());

                   OperacionConsulta
consulta=new
OperacionConsulta(this.broker,claseObtener);
          consulta.setRecursivo(false);
                   Object[]
objetos=consulta.getObjetos(criterioRel);
          if (objetos!=null)

asignaRelacion(objeto,ma.getNombre(),mr.esCardi
nalidadUno(),objetos);
                  }
                 }

           }
           arreglo.add(objeto);


          }
         }
         catch (Exception e)
{System.out.println("En OperacionConsulta
\n"+e.toString());}


        return arreglo.toArray();
        }


        public boolean getRecursivo()
        {
         return recursivo;
        }

    public void setRecursivo(boolean recursivo)
        {
         this.recursivo=recursivo;
        }

    public int getNumRegistros(MapaTabla
tabla,Object objeto)
        {String sql=null;
         int numRegistros=-1;
     Criterio
criterio=formaCriterioLlave(objeto,mapaClase);
         try
         {
          SqlSelect sqlSelect= new
SqlSelect(broker,mapaClase,criterio);

sqlSelect.formaSqlCount(tabla.getNombre());
          sql=sqlSelect.getSql();
          ResultSet
rs=broker.ejecutaQuery(sql);
      rs.next();
          numRegistros=rs.getInt(1);
          }
          catch (Exception e)
{System.out.println(e.toString());}
          return numRegistros;
        }

}
```

Clase Sql

```
import java.sql.Types;
import java.util.*;

/*
 *Clase que forma el estatuto sql de una
operacion
 */
class Sql
{
 Broker broker;
 String sql;
 Criterio criterio;

 static final String delimitadorStr="'";

 public Sql(Broker broker)
 {this.broker=broker;}

 public Sql(Broker broker,Criterio criterio)
 {this.broker=broker;
  this.criterio=criterio;
```

```
}
public void setCriterio(Criterio criterio)
{
 this.criterio=criterio;
}

String getSql()
{
 return sql;
}


public boolean esTexto(MapaColumna columna)
{//int
tipoDato=broker.getBaseDatos().getTipoDato(null
,columna.getTabla().getNombre(),columna.getNomb
re());
 int tipoDato=columna.getTipoDato();
 switch (tipoDato)
 {
 case Types.DATE :
 case Types.LONGVARCHAR :
 case Types.OTHER :
 case Types.VARCHAR :
 case Types.TIME :
 case Types.CHAR : return true;

 }
 return false;
}

public String
getSqlCriterioSimple(CriterioSimple criterio)
{
        String nombreClase=
criterio.getClase();
        String
nombreAtributo=criterio.getAtributo();
        String sql;
        String valor;
        MapaClase
clase=broker.getMapaClase(nombreClase);
    MapaAtributo
atributo=clase.getAtributo(nombreAtributo);
    MapaColumna columna=atributo.getColumna();
        MapaTabla tabla=columna.getTabla();
    if (esTexto(columna))
     valor="'"+criterio.getValor()+"'";
        else
         valor=criterio.getValor();
        sql=tabla.getNombre()+"."+columna.getN
ombre()+criterio.getOperador()+valor;

        return sql;
}

public String
getSqlCriterioSimple(MapaRelacion
mr,CriterioSimple criterio)
{
        String nombreClase=
criterio.getClase();
        String
nombreAtributo=criterio.getAtributo();
        String sql;
        String valor;


        MapaAtributo
atributo=mr.getAtributo(nombreClase,nombreAtrib
uto);
    if (atributo!=null)
    {
        MapaColumna
columna=atributo.getColumna();
        MapaTabla tabla=columna.getTabla();
    if (esTexto(columna))
     valor="'"+criterio.getValor()+"'";
        else
         valor=criterio.getValor();
        sql=tabla.getNombre()+"."+columna.getN
ombre()+criterio.getOperador()+valor;
        }
        else
     sql=getSqlCriterioSimple(criterio);
        return sql;

}

public String getSqlWhere(Criterio criterio)
{
  String tipoCriterio=criterio.getNombre();
   //Operaciones para un criterio simple
   if
(tipoCriterio.equals(CriterioSimple.CRITERIOSIM
PLE))
     {
          return
getSqlCriterioSimple((CriterioSimple)
criterio);
          }

   if
(tipoCriterio.equals(CriterioRelacion.CRITERIOR
ELACION))
     {

          return
getSqlWhere(criterio.getCriterio1());
         }


   if
(tipoCriterio.equals(CriterioAnd.CRITERIOAND))
   {
    return " ( "+
getSqlWhere(criterio.getCriterio1()) + ")"+ "
and " +"("
+getSqlWhere(criterio.getCriterio2())+ ")";
   }

   if
(tipoCriterio.equals(CriterioOr.CRITERIOOR))
    {
         return " ( "+
getSqlWhere(criterio.getCriterio1()) + ")"+ "
or " +"("
+getSqlWhere(criterio.getCriterio2())+ ")";
         }
  return "";
 }

 public String getSqlWhere(MapaRelacion
mr,Criterio criterio)
 {MapaClase mc=mr.getClase1();
  String nombreClase=mc.getNombre();
  String tipoCriterio=criterio.getNombre();
   //Operaciones para un criterio simple
   if
(tipoCriterio.equals(CriterioSimple.CRITERIOSIM
PLE))
    return
getSqlCriterioSimple(mr,(CriterioSimple)
criterio);

   if
(tipoCriterio.equals(CriterioRelacion.CRITERIOR
ELACION))
     {

          return
getSqlWhere(mr,criterio.getCriterio1());
         }


   if
(tipoCriterio.equals(CriterioAnd.CRITERIOAND))
   {
    return " ( "+
getSqlWhere(mr,criterio.getCriterio1()) + ")"+
" and " +"("
+getSqlWhere(mr,criterio.getCriterio2())+ ")";
   }

   if
(tipoCriterio.equals(CriterioOr.CRITERIOOR))
    {
         return " ( "+
getSqlWhere(mr,criterio.getCriterio1()) + ")"+
" or " +"("
+getSqlWhere(mr,criterio.getCriterio2())+ ")";
         }
  return "";


 }


 MapaColumna getColumna(MapaRelacion
mr,CriterioSimple criterio)
 {
  String nombreClase=criterio.getClase();
```

```
  String nombreAtributo=criterio.getAtributo();

  for (Enumeration
e=mr.getAtributosRelacion().elements();e.hasMor
eElements();)
   {
    MapaAtributo ma=(MapaAtributo)
e.nextElement();
    MapaClase mc=ma.getMapaClase();
    if
(mc.getNombre().equals(nombreClase)&&ma.getNomb
re().equals(nombreAtributo))
      return ma.getColumna();
   }
  return null;
 }

 MapaColumna getColumna(CriterioSimple
criterio)
 {MapaClase
mapaClase=broker.getMapaClase(criterio.getClase
());
  MapaColumna
mapaColumna=mapaClase.getAtributo(criterio.getA
tributo()).getColumna();
  return mapaColumna;
 }


}
```

Clase SqlSelect

```
import java.util.*;
import java.sql.*;

class SqlSelect  extends Sql
{
 MapaClase clase; //clase de la que se desea
hacer la consulta
 Hashtable tablasFrom;
 Hashtable camposSelect;
 Hashtable joins;
 boolean relaciones=false;

 SqlSelect(Broker broker,MapaClase
mapaClase,Criterio criterio)
 {super(broker,criterio);
  clase=mapaClase;
  tablasFrom=new Hashtable(1);
  camposSelect=new Hashtable(5);
  joins=new Hashtable(3);
 }

 SqlSelect(Broker broker,String
nombreClase,Criterio criterio)
 {super(broker,criterio);
  clase=broker.getMapaClase(nombreClase);
  tablasFrom=new Hashtable(1);

  camposSelect=new Hashtable(5);
  joins=new Hashtable(3);
 }


 public String obtenIdHashTabla(MapaTabla
tabla)
 {
  return tabla.getNombre().toUpperCase();
 }

 public String obtenIdHashColumna(MapaColumna
columna)
 {
  return
columna.getTabla().getNombre()+columna.getNombr
e();
 }


 public void getCampos()
 {boolean esUniforme=true;
  String nombreTablaCampo=null;
  for (Enumeration
e=clase.getAtributos().elements();e.hasMoreElem
ents() ; )
  {
   MapaAtributo ma=(MapaAtributo)
e.nextElement();
   if (!ma.esRelacion())
   {MapaColumna mc=ma.getColumna();

camposSelect.put(obtenIdHashColumna(mc),mc);
    MapaTabla tabla=mc.getTabla();
         if
(nombreTablaCampo!=null&&!tabla.getNombre().equ
als(nombreTablaCampo))
          esUniforme=false;
         nombreTablaCampo=tabla.getNombre();

tablasFrom.put(obtenIdHashTabla(tabla),tabla);
   }
  }
  if (!esUniforme)
   {

         //para cada una de las tablas del from
formar los joins necesarios
         for (Enumeration
e=tablasFrom.elements();e.hasMoreElements(); )
         {
          MapaTabla tabla=(MapaTabla)
e.nextElement();
      //verificar si tiene llave formanea

          ResultSet
llavesImportadas=broker.getBaseDatos().getForei
gnKeys(null,tabla.getNombre());
      try {
          while (llavesImportadas.next())
          {
           String
nombreTablaImporta=llavesImportadas.getString("
PKTABLE_NAME");
           if
(tablasFrom.containsKey(nombreTablaImporta))
            {
          //agrega las columnas a la relacion
            MapaColumna columnaJoin1=new
MapaColumna(llavesImportadas.getString("PKCOLUM
N_NAME"),new
MapaTabla(null,llavesImportadas.getString("PKTA
BLE_NAME")));
            MapaColumna columnaJoin2=new
MapaColumna(llavesImportadas.getString("FKCOLUM
N_NAME"),new
MapaTabla(null,llavesImportadas.getString("FKTA
BLE_NAME")));
        Join join= new
Join(columnaJoin1,columnaJoin2);
            agregaJoin(join);

            }
          }
          } catch (Exception ex)
{System.out.println(ex.toString());}
         }
   }
 }


 public String getSqlCampos()
 {String sql= new String();
 for (Enumeration
e=camposSelect.elements();e.hasMoreElements() ;
)
  {
   MapaColumna mc=(MapaColumna)
e.nextElement();
    sql=sql+
mc.getTabla().getNombre()+"."+mc.getNombre();
         if (e.hasMoreElements())
      sql=sql+",";
  }
  return sql;
 }



 public String obtenIdHashJoin(Join join)
 {
  return
obtenIdHashColumna(join.getColumna1())+obtenIdH
ashColumna(join.getColumna2());
 }

 public void agregaJoin(Join join)
 {
  joins.put(obtenIdHashJoin(join),join);
 }
```

```
 public void getTablasCriterio(Criterio
criterio)
 {
  String tipoCriterio;
  MapaTabla tabla;
  MapaAtributo atributo;
  MapaClase clase;
  String nombreClase;
  String nombreAtributo;
  Criterio criterioAux=criterio;
  tipoCriterio=criterioAux.getNombre();



   //Operaciones para un criterio simple
   if
(tipoCriterio.equals(CriterioSimple.CRITERIOSIM
PLE))
   {
     CriterioSimple
cs=(CriterioSimple)criterioAux;
          nombreClase= cs.getClase();
          nombreAtributo=cs.getAtributo();

clase=broker.getMapaClase(nombreClase);

atributo=clase.getAtributo(nombreAtributo);

tabla=atributo.getColumna().getTabla();

tablasFrom.put(obtenIdHashTabla(tabla),tabla);
    }


   //creiterio relacion
   //falta
   if
(tipoCriterio.equals(CriterioRelacion.CRITERIOR
ELACION))
   {

        CriterioRelacion cr=(CriterioRelacion)
criterioAux;

        MapaClase
caux=broker.getMapaClase(cr.getClase());


        MapaRelacion
relacion=broker.getMapaClase(cr.getClase()).get
MapaRelacion(cr.getAtributo());
        Hashtable atributos=
relacion.getAtributosRelacion();

        for (Enumeration
e=atributos.elements();e.hasMoreElements(
) ; )
    {atributo=(MapaAtributo) e.nextElement();
         MapaColumna
columnaJoin1=atributo.getColumna();
     MapaColumna
columnaJoin2=broker.getMapaClase(atributo.getMa
paClase().getNombre()).getAtributo(atributo.get
Nombre()).getColumna();
     Join join= new
Join(columnaJoin1,columnaJoin2);
         agregaJoin(join);
         tabla=atributo.getTabla();

tablasFrom.put(obtenIdHashTabla(tabla),tabla);
    }

        getTablasCriterio(cr.getCriterio1());

   }



   //Operaciones para un And
   if
(tipoCriterio.equals(CriterioAnd.CRITERIOAND))
   {

getTablasCriterio(criterioAux.getCriterio1());

getTablasCriterio(criterioAux.getCriterio2());
   }

   //Operaciones par un or
   if
(tipoCriterio.equals(CriterioOr.CRITERIOOR))
   {

getTablasCriterio(criterioAux.getCriterio1());

getTablasCriterio(criterioAux.getCriterio2());
   }



 }

 public void analizaConsulta()
 {
  //obtiene las tablas que se usan en el from

 getCampos();
 getTablasCriterio(this.criterio);
 }


 public void formaSql()
 {
   analizaConsulta();

   String sql=new String();
   String sqlCampos=getSqlCampos();
   String sqlFrom=new String();
   String sqlWhere=new String();
   if (criterio!=null)
    sqlWhere=getSqlWhere(criterio);
   String sqlJoins=new String();
   MapaTabla tabla;
   Enumeration e;
   for
(e=tablasFrom.elements();e.hasMoreElements() ;
)
   {tabla= (MapaTabla) e.nextElement();
    sqlFrom=sqlFrom+ " "+ tabla.getNombre();
         if (e.hasMoreElements())
          sqlFrom=sqlFrom+ ",";
   }

   for (e=joins.elements();e.hasMoreElements()
; )
   {Join join =(Join) e.nextElement();
    sqlJoins=sqlJoins+join.getSql();
         if (e.hasMoreElements())
         {
           sqlJoins=sqlJoins + " and ";
         }
   }

   sql="Select " + sqlCampos + " from " +
sqlFrom;
   if (!sqlJoins.equals("")
||!sqlWhere.equals(""))
    sql=sql + " where ";
   if (!sqlJoins.equals(""))
    {sql=sql + sqlJoins;
          if (!sqlWhere.equals(""))
           sql=sql+ " and ";
         }
   if (!sqlWhere.equals(""))
    sql=sql+  sqlWhere;

  this.sql=sql;


 }


 public void formaSqlCount(String nombreTabla)
 {
  String sqlWhere=getSqlWhere(criterio);
  String sql="Select count(*) from "+
nombreTabla;
  if (!sqlWhere.equals(""))
   sql=sql + " where "+sqlWhere;
  this.sql=sql;
 }

}
```

Clase SqlInsert

```
import java.util.*;
```

```
class SqlInsert  extends Sql
{
 MapaTabla mapaTabla=null;
 ArrayList valores=null;
 MapaRelacion mapaRelacion=null;

 SqlInsert(Broker broker,MapaTabla
mapaTabla,ArrayList valores)
 {super(broker);
  this.mapaTabla=mapaTabla;
  this.valores=valores;
 }

 SqlInsert(Broker broker,MapaTabla
mapaTabla,ArrayList valores,MapaRelacion
mapaRelacion)
 {super(broker);
  this.mapaTabla=mapaTabla;
  this.valores=valores;
  this.mapaRelacion=mapaRelacion;
 }


 public void formaSql()
 {String sql=new String();
  String campos=new String();
  String values=new String();
  String valor;
  String nombreTabla=mapaTabla.getNombre();


  for (int i=0;i<valores.size();i++)
  {CriterioSimple
criterioSimple=(CriterioSimple) valores.get(i);
   MapaColumna mc;
   if (mapaRelacion==null)
    mc=getColumna(criterioSimple);
   else
    mc=getColumna(mapaRelacion,criterioSimple);

   if
(mc.getTabla().getNombre().equals(nombreTabla))
    {
         valor=((CriterioSimple)
valores.get(i)).getValor();
         if (valor==null)
          valor="NULL";
         else
     if (esTexto(mc))

valor=delimitadorStr+valor+delimitadorStr;

         if (i==0)
          {campos=campos+mc.getNombre();
           values=values + valor;
          }
         else
          {campos=campos+","+mc.getNombre();
           values=values+","+valor;
          }
        }

  }
  sql="insert into " + nombreTabla + " ("
+campos+" ) values ( " +values+ " )";
  this.sql=sql;
 }

}
```

Clase SqlUpdate

```
import java.util.*;
import java.sql.*;

class SqlUpdate extends Sql
{
 MapaTabla mapaTabla=null;
 ArrayList valores=null;
 MapaRelacion mapaRelacion=null;
 boolean valoresNull=false;

 SqlUpdate(Broker broker,MapaTabla
mapaTabla,ArrayList valores,Criterio criterio)
 {super(broker,criterio);
  this.mapaTabla=mapaTabla;
  this.valores=valores;
 }

 SqlUpdate(Broker broker,MapaTabla
mapaTabla,ArrayList valores,MapaRelacion
mapaRelacion,Criterio criterio)
 {super(broker,criterio);
  this.mapaTabla=mapaTabla;
  this.valores=valores;
  this.mapaRelacion=mapaRelacion;
 }

 SqlUpdate(Broker broker,MapaTabla
mapaTabla,ArrayList valores,MapaRelacion
mapaRelacion,Criterio criterio,boolean
valoresNull)
 {super(broker,criterio);
  this.mapaTabla=mapaTabla;
  this.valores=valores;
  this.mapaRelacion=mapaRelacion;
  this.valoresNull=valoresNull;
 }


 public void formaSql()
 {String sql=new String();
  String valor;
  String nombreTabla=mapaTabla.getNombre();
  String strSet=new String();
  String sqlWhere=" where ";
  String nombreCampo=null;

  if (mapaRelacion==null)
   sqlWhere=sqlWhere+getSqlWhere(criterio);
  else

sqlWhere=sqlWhere+getSqlWhere(mapaRelacion,crit
erio);


  for (int i=0;i<valores.size();i++)
  {
   CriterioSimple
criterioSimple=(CriterioSimple) valores.get(i);
   MapaColumna mc=null;
   if (mapaRelacion==null)
    mc=getColumna(criterioSimple);
   else
    mc=getColumna(mapaRelacion,criterioSimple);

nombreCampo=mc.getTabla().getNombre()+"."+mc.ge
tNombre();

   if
(mc.getTabla().getNombre().equals(nombreTabla))
    {
     if (valoresNull)
     {valor="NULL";
     }
         else
         {
         valor=((CriterioSimple)
valores.get(i)).getValor();
     if (valor==null)
          valor="NULL";
         else
         if (esTexto(mc))

valor=delimitadorStr+valor+delimitadorStr;
         }
         if (i==0)

{strSet=strSet+nombreCampo+"="+valor;
          }
         else
          {

strSet=strSet+","+nombreCampo+"="+valor;
          }
        }
  }
  sql="update " +nombreTabla + " set " + strSet
+ sqlWhere;
  this.sql=sql;
 }

}
```

Clase SqlDelete

```
class SqlDelete extends Sql
{
 MapaTabla mapaTabla=null;
```

```
 MapaRelacion mapaRelacion=null;

 SqlDelete(Broker broker,MapaTabla
mapaTabla,Criterio criterio)
 {
  super(broker,criterio);
  this.mapaTabla=mapaTabla;
 }

 SqlDelete(Broker broker,MapaTabla
mapaTabla,MapaRelacion mapaRelacion,Criterio
criterio)
 {super(broker,criterio);
  this.mapaTabla=mapaTabla;
  this.mapaRelacion=mapaRelacion;
 }

 public void formaSql()
 {
  String sql=new String();
  String nombreTabla=mapaTabla.getNombre();
  String sqlWhere=" where ";

  if (mapaRelacion==null)
   {sqlWhere=sqlWhere+getSqlWhere(criterio);

   }
  else

{sqlWhere=sqlWhere+getSqlWhere(mapaRelacion,cri
terio);

   }

  sql="delete " +nombreTabla +  sqlWhere;
  this.sql=sql;
 }
}
```

Bibliografía

[AMBL98a] Scott Ambler, "Persistent Layer Requirements", *Software Development*, Enero 1998.

[AMBL98b] Scott Ambler, "Robust Persistence Layers", *Software Development*, Febrero 1998.

[AMBL98c] Scott Ambler, "Designing a Robust Persistence Layer", *Software Development* Marzo 1998.

[AMBL98d] Scott Ambler, "Designing a Robust Persistent Layer ", *Software Development,* Abril 1998.

[BASS98] Len Bass, Paul Clements y Rick Kazman, "Software Architecture in Practice", Addison-Wesley, 1998.

[BATO95] Don Batory, Lou Coglianese, Mark Goodwin, Steve Shafer, "Creating Reference Architectures: An Example from Avionics", *Symposium on Software Reusability*, Seattle, Washington. 1995.

[BATO96] Don Batory, "Subjectivity and GenVoca Generators", *International Conference on Software Reuse*, Orlando, Florida 1996.

[BATO98] Don Batory, "Product-Line Architectures", *Invited Presentation, Smaltalk and Java in Industry and Practical Training*, Erfurt, Germany Octubre 1998.

[BERG96] Ken Bergman, "Client/Server Solutions: Implementing the Layered Paradigm", *Microsoft Developer Network*, Abril 1996.

[BIGG94] Ted J. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *Third International Conference on Reuse*, Noviembre 1994.

[CLEME95] Paul C. Clements, "From Subroutines to Subsystems: Component Based Software Development", *The American Programmer,* Noviembre 1995.

[CLEME96] Paul C. Clements, "Comming Attractions in Software Architecture", *Technical Report CMU/SEI-96-TR-008 ESC-TR-96-008,* Enero 1996.

[DATE86] C.J. Date, "Introducción a los sistemas de bases de datos", *Adison Wesley* 1986.

[D'SOUZA98] Desmond D'souza y Alan Willis, "Objects, Components and Frameworks with UML: the catalisis approach ",Addison-Wesley, 1998.

[DUGU00] Claude Duguay, "Object/Relational Database Mapping", *Java Pro,* Enero 2000.

[FINC98] Lawrence Finch, "A Hybrid Approach to Component-Based Development", *Software Development,* Septiembre 1998.

[GAMM95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch. Design Patterns : " Elements of Reusable Object-Oriented Software", *Addison-Wesley Pub Co,* 1995.

[GARL94] David Garlan, Mary Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering Vol I,* Enero 1994.

[HAGE95] José Alessio Hagen Estrada, *Puente Objeto Relacional Para Integrar Aplicaciones Orientadas a Objetos y Bases de Datos Relacionales,* Diciembre 1995.

[JOHN98] Ralph E. Johnson, Brian Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, Julio 1998.

[KARA99] Henry F. Korth, Abraham Silberschatz, "The Four Faces of JDBC", *Component Strategies*, Febrero 1999.

[KORT93] Henry F. Korth, Abraham Silberschatz, *Fundamentos de Bases de Datos, Segunda edición*, McGrawHill 1993.

[KURA98] Deborah Kurata, "Use a Design Pattern to Access Data", *Visual Basic Programmer's Journal,* Mayo 1998.

[LARM98] Craig Larman, "Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design", *Prentice Hall,* 1998.

[LINT98] David S. Linthicum, "Database APIs and Java", *Component Strategies,* Agosto 1998.

[LUCK95] David C. Luckham, James Vera, Sigurd Medal, *Three Concepts of System Architecture,*Julio 1995.

[PAGA98] Dennis Pagano, George Kosmides, "*Understanding Application Architectures", Component Strategies,* Agosto 1998.

[PERR92] Dewayne E. Perry, Alexander L. Wolf, "Foundations for the Study of Software Architecture", *Software Engineering Notes*, Octubre 1992.

[TAIV96] Antero Taivalsaari, *ACM Computing Surverys, Vol 28 No #3,* Septiembre 1996.

[TALI93] Taligent, "Leveraging Object-Oriented Frameworks", *Technical White Paper,* 1993.

Centro de Información-Biblioteca
30002005826722