UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DESARROLLO DE CAPAS DE ABSTRACCIÓN PARA MEJORAR LA EFICIENCIA EN LA CONSTRUCCIÓN DE APLICACIONES WEB J2EE

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

SERGIO EDUARDO ACEVEDO MUÑOZ

PROFESOR GUÍA:
PABLO GONZÁLEZ JURE

MIEMBROS DE LA COMISIÓN:
SERGIO OCHOA DELORENZI
JOSE BENGURIA DONOSO

SANTIAGO DE CHILE
ENERO 2008

Agradecimientos

Dedicado a mis padres por la constante preocupación, cariño y apoyo brindado durante toda mi vida, a Priscilla por su apoyo y esfuerzo para sacar este proyecto adelante y una mención particular a mi profesor guía por la dedicación brindada a este tema.

Resumen

Las capas de abstracción son componentes de software que permiten encapsular funcionalidades de un sistema permitiendo la reutilización de componentes a través de una interfaz de programación estandarizada.

El objetivo del presente trabajo de título es desarrollar 2 capas de abstracción que permitan agilizar los tiempos de desarrollo de aplicaciones Web bajo la arquitectura J2EE. Estas capas están enfocadas al desarrollo de componentes EJB, y la otra para el desarrollo de capas de persistencia vía JDBC.

El surgimiento de estas capas se plantea debido a los largos tiempos de desarrollo que se tienen al usar EJB y JDBC directo para desarrollar servicios y persistencia.

Lo anterior se justifica por varias razones, por ejemplo, en el caso de los servicios EJB existe una gran cantidad de archivos que se tienen que manipular para hacer un EJB, lo cual trae multiples conflictos en el sistema de control de versiones cuando trabajan varios desarrolladores simultáneamente en la capa de servicios, o bien la dificultad para tener traza de errores cuando se tienen problemas de configuración de estos archivos y otras clases de problemas referidos a la falta de automatización de ciertos procesos repetitivos de invocación de implementaciones finales y manejo de excepciones.

En el caso de la persistencia, los problemas típicos que se plantean son las tareas repetitivas y conducentes a error, como por ejemplo, obtención y cierre de los recursos, consultas mal escritas, validaciones de largo, tipo y otras que no se efectúan antes de la ejecución de consultas, complejidad y código poco mantenible para hacer consultas con filtros dinámicos, la asociación de valores de los resultados de las consultas a objetos de negocios muchas veces se ve replicada, etc.

Todo lo anterior induce la necesidad de crear una abstracción que permita hacer todas estas tareas de una manera lo más simple posible tratando de no perder algunas de las características que tienen estas tecnologías.

El resultado final corresponde a 2 capas de abstracción, la primera tiende a facilitar la creación de EJB, de manera de que el desarrollo de estos mismos involucre menos procesos de configuración y manejo de archivos, como a su vez tener cierto nivel de independencia de la tecnología EJB en términos de encapsularla al desarrollador. La segunda por su parte, permite encapsular las tareas de obtención de recursos y conectividad a nivel de aplicación contra la base de datos, en conjunto con la generación y ejecución de consultas SQL y la transformación de datos de objetos de negocios a datos del modelo relacional.

Como resultado de este trabajo, a través del uso de métricas de comparación, se pueden notar las diferencias en tiempos de desarrollo, facilidad de mantención y complejidad entre una solución usando capas de abstracción versus una solución sin ellas.

Cabe destacar que ambas capas de abstracción en definitiva son capas generadoras de código, con la característica que el código generado se hace en tiempo de ejecución, lo cual tiene el beneficio que no es necesario tener código generado dentro del código de las aplicaciones que utilicen estas capas de abstracción.

Índice general

1. Antecedentes **1**

1.1. Introducción 1

1.2. Problemática Encontrada 3

1.3. Capas de Abstracción 5

1.4. Organización del documento 5

2. Justificación, Objetivos y Alcances **6**

2.1. Objetivo General 6

2.2. Objetivos Específicos 7

2.3. Trabajo Realizado 7

2.4. Alcances 7

3. Revisión Bibliográfica **9**

3.1. Tecnologías y Estándares 9

3.1.1. SQL 9

3.1.2. XML 10

3.1.3. J2EE 10

3.1.4. Patrones de Diseño 15

3.2. Capa de Abstracción de Servicios (EJBProxy) 26

3.2.1. Introducción y Conceptos 26

3.2.2. Estado del Arte 26

3.2.3. Tecnologías Alternativas 27
3.2.3.1. Jini Network Technology 27
3.2.3.2. Spring 33
3.3. Capa de Abstracción de Persistencia (ORM) 38
3.3.1. Introducción y Conceptos 38
3.3.2. Estado del Arte 39
3.3.3. Tecnologías Alternativas 39
3.3.3.1. iBATIS 39
3.3.3.2. Hibernate 42

4. Análisis, Arquitectura y Diseño **47**
4.1. Requerimientos 47
4.1.1. EJBProxy 47
4.1.2. ORM 48
4.2. Arquitectura y Diseño 49
4.2.1. EJBProxy 49
4.2.2. ORM 53

5. Implementación **61**
5.1. EJBProxy 61
5.1.1. Problemas enfrentados durante el desarrollo 61
5.1.2. Obtención de servicios a través de EJBProxy 61
5.1.3. Exposición de servicios a través de EJBProxy 62
5.1.4. Alcances particulares de la implementación 67
5.2. ORM 68
5.2.1. Problemas enfrentados durante el desarrollo 68
5.2.2. Generación de consultas SQL 68
5.2.3. Ejecución de consultas 73
5.2.4. Transformación de datos 75
5.2.5. Camino 77
5.2.6. Alcances de la implementación 79

6. Aplicaciones Ejemplo **80**
6.1. Arquitectura de aplicación J2EE bajo estándares Telefónica 80
6.2. Modificaciones necesarias para el uso de EJB-Proxy 83
6.3. Modificaciones necesarias para el uso de ORM 90
7. Conclusiones **99**
7.1. Métricas 99
7.1.1. Líneas de código (LOC) 99
7.1.2. Cantidad de Archivos 101
7.1.3. Procesos involucrados (configuración) 102
7.1.4. Vendor-Lockin 102
7.1.5. Coordinación 102
7.2. Conclusiones Finales 103
7.3. Trabajos futuros y mejoras 105
Bibliografía **106**

Capítulo 1

Antecedentes

1.1. Introducción

En la actualidad, es innegable que la Red se ha convertido en un medio de uso cotidiano y prácticamente indispensable dentro de la vida cotidiana y el trabajo de las personas. Multitud de entidades, empresas y organizaciones tienen su representación en este Medio donde ofrecen sus productos y servicios.

Es por ello que se ha generado una necesidad de desarrollar aplicaciones web de calidad las cuales siempre deben adecuarse a las necesidades para las que son desarrolladas. Lamentablemente muchas veces se suele tender a usar las últimas novedades o herramientas que no son adecuadas, cuando lo que se requiere son aplicaciones estables, eficientes, flexibles y seguras.

En ingeniería de software, una aplicación web es aquella que puede accederse mediante un navegador a través de una red, como una intranet o bien a través de internet.

Por lo anterior, las aplicaciones web se han vuelto muy populares, principalmente debido a su fácil mantención y capacidad de actualización mucho más simple que una aplicación cliente-servidor estándar. Ello se debe a que no es necesario distribuir a cada uno de los usuarios una nueva versión o actualización de la aplicación, ya que esas actualizaciones se realizan de manera centralizada y automáticamente cada usuario cuando ingrese a la aplicación verá reflejados los cambios de manera transparente.

Una de las tecnologías predominantes en el inicio del desarrollo de aplicaciones web fue CGI[1]. CGI fue una de las primeras técnicas que permitían tener contenido dinámico en una aplicación web, el cual era programado en prácticamente cualquier lenguaje tales como C/C++, Perl, C Shell, Bourne Shell, etc.

Gracias a lo anterior, fue posible implementar todo un nuevo conjunto de funcionalidades en páginas web, por lo cual CGI rápidamente se transformó en el estándar de facto siendo utilizado en múltiples servidores web.

[1] Common Gateway Interface.

Lamentablemente CGI tiene algunas desventajas que fueron notorias en el mediano plazo, dentro las cuales y la de mayor impacto es el ciclo de vida del requerimiento (request) en una aplicación CGI.

Cuando el servidor recibe un request que accede al programa CGI, éste debe crear un nuevo proceso para ejecutar el programa CGI y luego pasarle vía variables de ambiente o por la entrada estándar, cada bit de información necesaria a la aplicación CGI para que pueda generar la respuesta del servidor.

El crear un proceso cada vez que llega un nuevo request al servidor toma un tiempo significativo, y además consume una cantidad considerable de recursos del mismo, por lo cual el número de request que puede manejar el servidor de manera concurrente es limitado. La figura 1.1 muestra el ciclo de vida de un CGI.

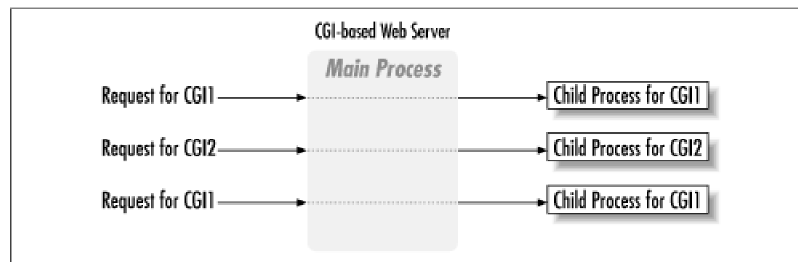

Figura 1.1: Ciclo de Vida del request en CGI

Dado este problema, algunas compañías y comunidades de Código Abierto decidieron hacer ciertas mejoras a CGI, con lo cual surgieron distintas alternativas como FastCGI, mod_perl, PerlEx y otros. Sin embargo muchas de estas nuevas soluciones involucraban perder portabilidad y mantenibilidad del código en sacrificio de rendimiento y escalabilidad a nivel de servidor. Por ello, Java pudo imponerse gracias a su portabilidad y su diseño orientado a objetos.

De manera natural, Sun (8) creó Java pensando en una funcionalidad en Internet. Para el año 1997 Sun anunció la API de Servlet, la cual tenía como objetivo reemplazar definitivamente a CGI.

A diferencia de CGI, el cual inicia un proceso por cada request, los Servlet ejecutan sólo un proceso y por cada request generan un nuevo thread. Esto representa una arquitectura mucho más eficiente, dado que el overhead adicional por cada nuevo request simultáneo es muy pequeño en relación a CGI.

Esto representó una mejora sustantiva en el desarrollo de aplicaciones web, sin embargo a nivel de servidor esto representaba sólo la capa de visualización de las aplicaciones.

Hacia 1999 Sun libera los primeros beta de su especificación J2EE[2], la cual tiene por objetivo promover el desarrollo de aplicaciones web multicapa en las que el contenedor web almacena

[2]J2EE: Java 2 Enterprise Edition o bien JEE

componentes web que están dedicados a manejar la lógica de presentación de una aplicación dada, y responden a las peticiones del cliente (como un navegador web). Por otro lado, el contenedor EJB, almacena componentes de aplicación que responden a las peticiones de la capa web como se muestra en la figura 1.2.

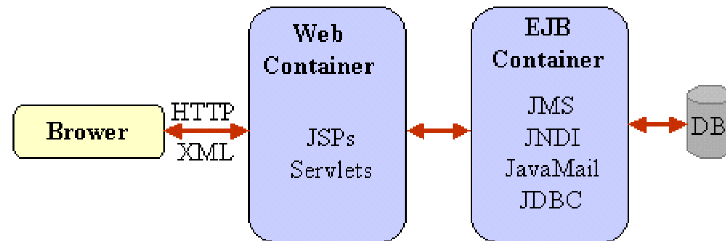

Figura 1.2: Arquitectura Multicapa J2EE

De este modo, se tiene naturalmente una arquitectura multicapa en una aplicación J2EE. La capa de visualización, la cual se desarrolla principalmente con JSP y Servlets, la capa de servicios en la cual se utilizan principalmente EJB y finalmente una capa de persistencia en la cual se usa JDBC.

1.2. Problemática Encontrada

La especificación J2EE, a pesar de aportar mucho en el desarrollo de aplicaciones Web, tiene el problema de ser muy extensa y por lo mismo existe una tendencia a llegar y usar la tecnología sin preocuparse mucho de cuáles son los problemas comunes que se tienen en su uso.

En particular algunos de los problemas más comunes que se pueden encontrar en las aplicaciones web desarrolladas en una arquitectura J2EE son los siguientes:

Problemas comunes al usar Servlets y JSP directo

- A pesar de ser una evolución en las aplicaciones web tradicionales, los Servlets tienen grandes limitaciones en términos de presentación.
- El uso de JSP tiende a resolver este problema, debido a que gran parte de la diagramación ya está diseñada en HTML, pero genera otro que puede llegar a ser aún peor el cual es el uso de "scriptlets". El problema de ello es que se pierde mucha mantenibilidad, capacidad de reusar código y flexibilidad frente a los cambios.

Afortunadamente, se han desarrollado diversas soluciones de código abierto que tratan de resolver estos problemas como por ejemplo Struts (17), el cual está basado en el patrón MVC. Otra alternativa es JSF (18), el cual está basado en componentes. En ambos casos, estas soluciones ofrecen un nivel de abstracción un poco más alto que permite al desarrollador no tener que usar directamente JSP y/o Servlets, sino que termina utilizándolos aunque de manera indirecta a través de la plataforma escogida.

Problemas comunes al usar EJB

- Dependencia del IDE[3] para la generación de las interfaces Remotas y Locales en conjunto con los archivos de configuración.
- Dificultad de Testing. Esto ocurre debido a que es necesario desplegar los EJB en el contenedor para poder ejecutar las pruebas, y por tanto el testing es una tarea bastante engorrosa y lenta.
- Para poder desplegar un EJB, usualmente se necesita modificar al menos 3 archivos de configuración. Si se tiene un sistema de tamaño mediano a grande esto tiende a ser un problema ya que se generan conflictos en los archivos de configuración en el sistema de control de versiones (CVS[4], SVN[5] u otros). Esto ocurre debido a la cantidad de desarrolladores trabajando en un mismo archivo con la consecuente pérdida de tiempo para resolver los conflictos.

Existen distintas alternativas a EJB que resuelven algunos de estos problemas (las cuales veremos en detalle posteriormente en el capítulo Metodología) como RMI, Web Services, Spring Remoting, y otros.

Problemas comunes al usar JDBC directo

- Connection-Leak. Este es el problema más común que se tiene al usar JDBC directo y significa que en la aplicación no se están cerrando de manera correcta y consistente sus conexiones a bases de datos.
- La escritura de SQL es una tarea muy tendiente a errores. Además mucho SQL que se escribe dentro de una aplicación puede considerarse como SQL trivial, por ejemplo los inserts en una tabla, querys a 1 o 2 tablas, etc.

En este campo, se han desarrollado muchos frameworks que resuelven parte de estas problemáticas tales como Ibatis (6), Hibernate (14) y el más reciente JPA (15), el cual es parte de la especificación J2EE 1.5 (llamada ahora JEE 5). El detalle de estas tecnologías se verá en el capítulo Metodología de este tema.

En resumen, puede decirse que en la mayoría de los casos podemos notar un común denominador, el cual es la falta de algún nivel o capa de abstracción que permita evitar la mayoría de los problemas típicos que se tienen en el desarrollo de la aplicación web J2EE.

[3] IDE: Integrated Development Environment o Ambiente Integrado de Desarrollo
[4] Concurrent Versioning System
[5] Subversion

1.3. Capas de Abstracción

Una capa de abstracción es un elemento o componente que permite eliminar o bien ocultar ciertas complejidades o rutinas recurrentes que se tienen a la hora de desarrollar software así como también minimizar el esfuerzo en caso de necesitar cambios el sistema frente a nuevos requerimientos.

Idealmente una capa de abstracción se construye utilizando 1 o más patrones de diseño, debido a que éstos por su naturaleza son soluciones probadas dentro de la industria y resuelven alguna parte de la problemática global que pretende resolver la capa de abstracción. Para el caso de este tema, se construirán 2 capas de abstracción:

EJB Proxy: Capa que se encarga principalmente de exponer servicios implementados con interfaces y clases POJO[6] como si fueran un EJB. Para ello se utilizaron diversos patrones de diseño como el patrón Fachada, Command, Delegate y Proxy.

ORM: Un ORM[7] es una capa que permite relacionar objetos con un modelo de datos relacional, de modo de ocultar todo el mecanismo de conexión al motor de base de datos y además no tener que escribir las sentencias SQL necesarias para efectos de hacer consultas y/o modificaciones a los registros de la base de datos.

1.4. Organización del documento

Este documento esta organizado de la siguiente manera:

Capítulo 1: Introducción al tema indicando la problema encontrada y definiendo conceptos generales.

Capítulo 2: Indica los objetivos, justificación y alcances de este tema.

Capítulo 3: Muestra la revisión bibliográfica para introducir al lector en alternativas tecnológicas de similar alcance.

Capítulo 4: Presenta la arquitectura y diseño de las capas desarrolladas.

Capítulo 5: Desarrollo de la implementación de las capas de abstracción.

Capítulo 5: Presenta aplicaciones ejemplo donde se usan las capas de abstracción desarrolladas.

Capítulo 7: Muestra las métricas, conclusiones y mejoras posibles a las capas desarrolladas.

[6]POJO: Plain Old Java Object o Clases Javas Planas
[7]ORM: Object-Relational mapping

Capítulo 2

Justificación, Objetivos y Alcances

El proceso de desarrollo de software puede definirse como un conjunto de procedimientos que utilizan herramientas, métodos y prácticas que se emplean para producir software. Como cualquier otra organización, las dedicadas al desarrollo de software mantienen entre sus principales fines, la producción de software de acuerdo con la planificación inicial realizada, además de una constante mejora con el fin de lograr los tres objetivos últimos de cualquier proceso de producción: alta calidad y bajo costo, en el mínimo tiempo.

Es por ello que el desarrollo de prácticas, metodologías y plataformas para un desarrollo óptimo es esencial. En este punto, las capas de abstracción dentro del desarrollo cumplen un rol fundamental, el cual es poder mejorar la gestión, mantenibilidad, flexibilidad y bajar los tiempos y costos de los proyectos de desarrollo de software, de manera de poder hacer predicciones acertivas con respecto a los plazos y compromisos comprometidos.

Lamentablemente muchas veces por restricciones administrativas, contractuales o de estándares internos que poseen los demandantes de sistemas (clientes) a nivel técnico, el uso de plataformas o bibliotecas externas que facilitan el desarrollo no es posible. El tener capas de abstracción permite tener cierta independencia tecnológica con respecto a los requerimientos planteados por los clientes, evitando lo que se denomina vendor-lockin o dependencia del vendedor.

La motivación de este tema en particular es poder ejemplificar con un caso real de cuanto son estos tiempos, de manera de poder contrastar (con alguna métrica a definir basada en líneas de código) el desarrollo con y sin estas capas. Además presentar estas capas de abstracción en su diseño conceptual e implementación, mostrando a su vez el problema que resuelven cada una de ellas

2.1. Objetivo General

El objetivo general de este tema es desarrollar 2 capas de abstracción, en lo que corresponde a capa de servicios y de persistencia. Para ello se debe presentar el problema que resuelve

cada capa y a su vez el mecanismo de resolución con el respectivo uso de las mismas. También poder determinar cuanto es el ahorro que se obtiene gracias a estas capas de abstracción y poder compararlas con otras tecnologías alternativas.

2.2. Objetivos Específicos

1. Presentar la arquitectura de aplicación web J2EE con las restricciones de Telefónica.
2. Desarrollar una aplicación ejemplo de mediana complejidad bajo la arquitectura anterior.
3. Desarrollar la capa de abstracción a nivel de Servicios (EJBProxy).
4. Desarrollar la capa de abstracción a nivel de Persistencia(ORM).
5. Desarrollar la aplicación ejemplo, utilizando esta vez las capas de abstracción.
6. Presentar la diferencia entre las 2 alternativas en términos de líneas de código, tiempo de desarrollo, conocimiento necesario para desarrollar, procesos involucrados y necesidades de coordinación u organización en etapa de desarrollo.
7. Presentar otras tecnologías alternativas que resuelven la misma problemática, comparándola con las anteriores.

2.3. Trabajo Realizado

El trabajo principal realizado en este tema, fue el desarrollo de 2 capas de abstracción que permitieran tener un desarrollo ágil, flexible y de fácil mantención. La primera de estas capas es una capa de abstracción de servicios, la cual permite exponer interfaces POJO como servicios EJB a través de un EJBProxy. La segunda capa de abstracción es un ORM para la capa de persistencia, cuya finalidad es que el desarrollador no tenga que escribir gran parte de las sentencias SQL típicas que son necesarias dentro del desarrollo de aplicaciones web, sino más bien, que esta capa de abstracción permita generar la mayor cantidad de sentencias SQL posibles y que las sentencias más complejas tengan que escribirse a mano.

Ambas permiten que el desarrollador haga la menor cantidad de tareas mecánicas en términos de programación, esto quiere decir que estas capas atacan los puntos donde se originan principalmente malas prácticas como “copy and paste”, cambios a los diseños ya efectuados, mucho código repetido, etc.

2.4. Alcances

Cabe destacar que este tema no tuvo por objetivo plantear metologías de desarrollo que tengan que ser adoptadas por la industria y para todo tipo de desarrollo web J2EE, sino más

bien, presentar y ejemplificar con un caso concreto cuan importante son ciertas decisiones de diseño (en este caso la creación y uso de capas de abstracción) en lo que es el desarrollo del producto final.

Capítulo 3

Revisión Bibliográfica

3.1. Tecnologías y Estándares

3.1.1. SQL

El Lenguaje de Consulta Estructurado (Structured Query Language) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones sobre las mismas. Reúne características del álgebra y el cálculo relacional permitiendo lanzar consultas con el fin de recuperar información de interés de una base de datos, de una forma sencilla.

Los orígenes del SQL están ligados a los de las bases de datos relacionales. En 1970 Codd[1] propone el modelo relacional y asociado a éste un sublenguaje de acceso a los datos basado en el cálculo de predicados (4). Basándose en estas ideas los laboratorios de IBM (9) definen el lenguaje SEQUEL (Structured English Query Language) que más tarde sería ampliamente implementado por el motor de base de datos experimental *System R*, desarrollado en 1977 también por IBM. Sin embargo, fue Oracle (10) quien lo introdujo por primera vez en 1979 en un programa comercial.

SEQUEL terminaría siendo el predecesor de SQL, siendo éste una versión evolucionada del primero. SQL pasa a ser el lenguaje por excelencia de los diversos motores de base de datos relacionales surgidos en los años siguientes y es por fin estandarizado en 1986 por el ANSI (12), dando lugar a la primera versión estándar de este lenguaje, el SQL-86 o SQL1. Al año siguiente este estándar es también adoptado por la ISO (13).

Hasta el día de hoy, SQL y los modelos relacionales son el estándar de facto en la industria de las bases de datos. Para efectos de este tema, se espera que el lector esté familiarizado al menos con las sentencias básicas del lenguaje de consulta SQL, como sentencias de consulta, inserción y modificación de registros.

[1] Edgar Frank Codd

3.1.2. XML

XML, sigla en inglés de e**X**tensible **M**arkup **L**anguage (lenguaje de marcas extensible), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (7).

XML es una simplificación y adaptación del SGML y permite definir la gramática de lenguajes específicos (de la misma manera que HTML es a su vez un lenguaje definido por SGML). Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

Los lenguajes de marcas no son equivalentes a los lenguajes de programación, aunque se definan igualmente como "lenguajes". Son sistemas de descripción de información, normalmente documentos, que típicamente se pueden controlar desde cualquier editor de texto.

Las marcas más utilizadas suelen describirse por textos descriptivos encerrados entre signos de "menor" ($<$) y "mayor" ($>$), siendo lo más usual que existan una marca de principio y otra de final.

Se puede decir que existen tres utilizaciones básicas de los lenguajes de marcas: los que sirven principalmente para describir su contenido, los que sirven más que nada para definir su formato y los que realizan las dos funciones indistintamente.

Las aplicaciones de bases de datos son buenas referencias del primer sistema, los programas de tratamiento de textos son ejemplos típicos del segundo tipo, y HTML es la muestra más conocida del tercer modelo.

El uso de XML en el desarrollo de las capas de abstracción de este tema, se tiene en la capa de persistencia. La función que se le dará será el de representar la traducción de objetos de base de datos en objetos de transferencia que se utilizan en la aplicación, junto con el manejo del modelo de datos relacional a nivel de objetos que se obtendrá a partir de la base de datos.

3.1.3. J2EE

J2EE es el estándar de la industria para desarrollar applicaciones Java portables, robustas, escalables y seguras en el lado del servidor. Está basado en Java y proporciona APIs para servicios web, modelo de componentes, gestión y comunicación que lo convierten en el estándar de la industria para implementar aplicaciones Web y aplicaciones con arquitectura orientada a servicios.

J2EE proporciona una arquitectura multi-capa. La capa cliente puede estar constituida por aplicaciones Java de escritorio o navegadores HTML. Las capas proporcionadas por J2EE propiamente tal son las capas Web (mediante las tecnologías Servlets, JSP) y las capas de Negocio (mediante tecnologías como EJB, JMS o Web Services). Por último, estas capas se comunican con una capa de datos (base de datos o aplicaciones y sistemas legacy).

Las razones que empujan a la creación de la plataforma J2EE:

- ***Programación eficiente.***

Para conseguir productividad es importante que los equipos de desarrollo tengan una forma estándar de construir múltiples aplicaciones en diversas capas (cliente, servidor web, etc.). La idea de J2EE es que los desarrolladores se pueden especializar en una capa, haciendo que el desarrollo sea mucho más eficiente.

- ***Extensibilidad frente a la demanda del negocio.***

En un contexto de crecimiento de número de usuarios es precisa la gestión de recursos, como conexiones a bases de datos, transacciones o balance de carga. Además los equipos de desarrollo deben aplicar un estándar que les permita abstraerse de la implementación del servidor, con aplicaciones que puedan ejecutarse en múltiples servidores, desde un simple servidor hasta una arquitectura de alta disponibilidad con balance de carga entre diversas máquinas.

- ***Integración.***

Los equipos de ingeniería precisan estándares que favorezcan la integración entre diversas capas de software.

La plataforma J2EE implica una forma estándar de implementar y desplegar aplicaciones empresariales. Se supone que las aplicaciones empresariales son desarrolladas internamente en las empresas por desarrolladores que tienen conocimientos acabados del negocio, pero no así de tecnologías técnicamente más complejas como sockets, threading, programación paralela, programación distribuida, sistemas en cluster, sistemas de alta disponibilidad, etc.

Sin embargo esas características sí son deseables en un ambiente de producción, por lo que se propone que sean los *Servidores de Aplicaciones* los que se preocupen de esos temas más técnicos, de modo que los desarrolladores de aplicaciones empresariales se preocupen de resolver problemas de negocio.

El estándar J2EE se ha abierto a numerosos fabricantes de software para conseguir satisfacer una amplia variedad de requisitos empresariales. La arquitectura J2EE implica un modelo de aplicaciones distribuidas en diversas capas o niveles. La capa cliente admite diversos tipos de clientes, la capa intermedia a su vez contiene subcapas y la capa de aplicaciones "*backend*" como ERP, EIS, bases de datos, etc.

Otro concepto clave a la arquitectura es el de contenedor, que dicho de forma genérica no es más que un entorno de ejecución estandarizado que ofrece unos servicios por medio de componentes. Los componentes externos al contenedor tienen una forma estándar de acceder a los servicios de dicho contenedor, con independencia del fabricante.

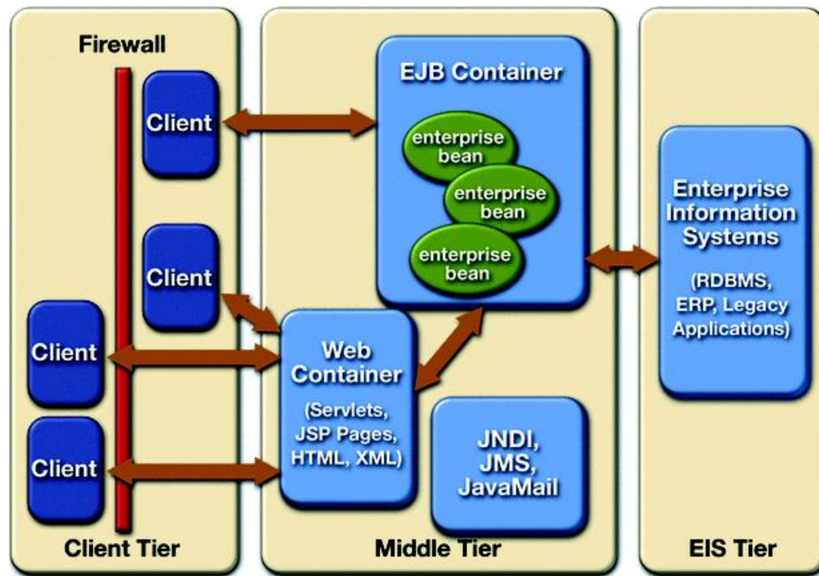

Figura 3.1: Modelo de Capas J2EE

En J2EE existen 2 tipos de contenedores:

- Contenedor Web, también denominado contenedor de Servlet/JSP, maneja la ejecución de los servlets y páginas JSP. Estos componentes típicamente se ejecutan sobre un servidor J2EE. También se usa como sinónimo de "*servidor de aplicaciones J2EE*", aunque esta es una confusión de la parte por el todo: el servidor de aplicaciones incluye un contenedor web entre otras cosas.
- Contenedor EJB (Enterprise JavaBeans), que gestiona la ejecución de los EJB. Esta ejecución requiere de un servidor J2EE.

Los contenedores incluyen descriptores de despliegue, que son archivos XML que nos sirven para configurar el entorno de ejecución: rutas de acceso a aplicaciones, control de transacciones, parámetros de inicialización, etc.

La plataforma J2EE incluye APIs para el acceso a sistemas empresariales:

- JDBC es el API para accceso a Base de Datos desde Java.
- Java Transaction API (JTA) es el API para manejo de transacciones de BD.
- Java Naming and Directory Interface (JNDI) es el API para acceso a servicios de nombres y directorios.
- Java Message Service (JMS) es el API para el envío y recepción de mensajes por medio de sistemas de mensajería empresarial como IBM MQ Series u otros proveedores de JMS.

- JavaMail es el API para envío y recepción de email.
- Java IDL es el API para llamar a servicios CORBA.

Servidor de aplicaciones J2EE

La arquitectura de un servidor de aplicaciones incluye una serie de servicios, al menos encontraremos un servicio de "*listener*" y los servicios propios de un contenedor Web:

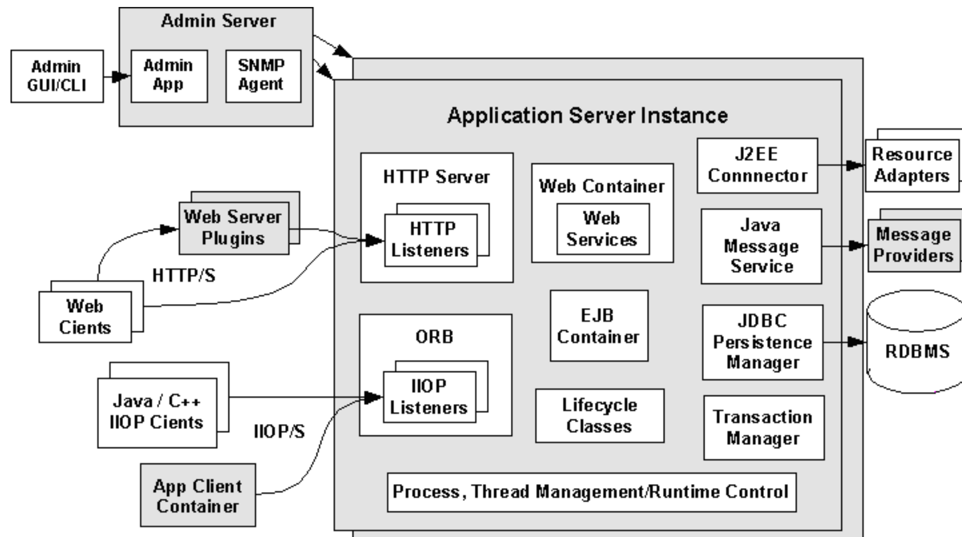

Figura 3.2: Servidor de Aplicaciones J2EE

Algunos aspectos relevantes a considerar son:

- Sobre una misma máquina podemos tener diversas instancias de un Servidor de Aplicaciones.
- Un Servidor de Aplicaciones tendrá un servidor de administración.
- Un servicio crucial es la capacidad de recibir peticiones HTTP, para lo cual tenemos un HTTP Listener (aunque puede tener listeners para otros protocolos como IIOP).
- El módulo ORB recibe peticiones IIOP que normalmente son reenviadas a los EJBs y capacita al Servidor de Aplicaciones para invocar EJBs de otros sistemas.
- El contenedor web gestiona la ejecución de Servlets y JSP.
- Los conectores capacitan al Servidor de Aplicaciones para acceder a sistemas empresariales (backends).
- El Java Message Service (JMS) ofrece conectividad con sistemas de mensajería como MQSeries.

- El API JDBC da la capacidad de gestionar bases de datos, pero además permite ofrecer servicios como un pool de conexiones.
- El Servidor de Aplicaciones proporciona la gestión de hebras (threads), ya que será necesario controlar una situación en la que tenemos una instancia de un componente (por ejemplo, un servlet) que da respuesta a varias peticiones, donde cada petición se resuelve en una hebra.

Capas

En la arquitectura J2EE se contemplan cuatro capas, en función del tipo de servicio y contenedor:

- Capa de cliente, también conocida como capa de presentación o de aplicación. Nos encontramos con componentes Java (applets o aplicaciones) y no-Java (HTML, JavaScript, etc.).
- Capa Web. Intermediario entre el cliente y otras capas. Sus componentes principales son los servlets y las JSP. Los servlets pueden llamar a los EJB directamente, mientras que las JSP utilizan etiquetas propias. Tanto servlets como JSP se ejecutan en el servidor.
- Capa Enterprise JavaBeans. Permite a múltiples aplicaciones tener acceso de forma concurrente a datos y lógica de negocio. Los EJB se encuentran en un servidor EJB, que no es más que un servidor de objetos distribuidos. Un EJB puede conectarse a cualquier capa, aunque su misión esencial es conectarse con los sistemas de información empresarial (un gestor de base de datos, ERP, etc.)
- Capa de sistemas de información empresarial.

La visión de la arquitectura es un esquema lógico, no físico. Cuando hablamos de capas nos referimos sobre todo a servicios diferentes (que pueden estar físicamente dentro de la misma máquina e incluso compartir servidor de aplicaciones y máquina virtual).

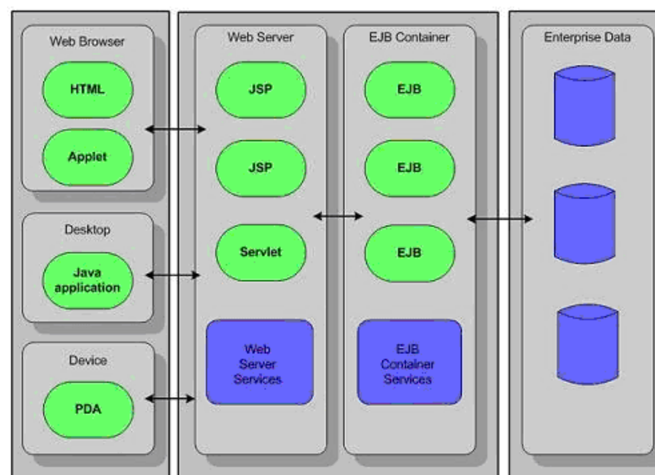

Figura 3.3: Modelo de Capas en Servidor de Aplicaciones

Esta arquitectura pretende ser utilizada en diversos ambientes empresariales, de modo que existen diferentes escenarios en los que se puede aplicar y parta los cuales tiene diferentes configuraciones posibles. El escenario más común es el escenario desde un navegador.

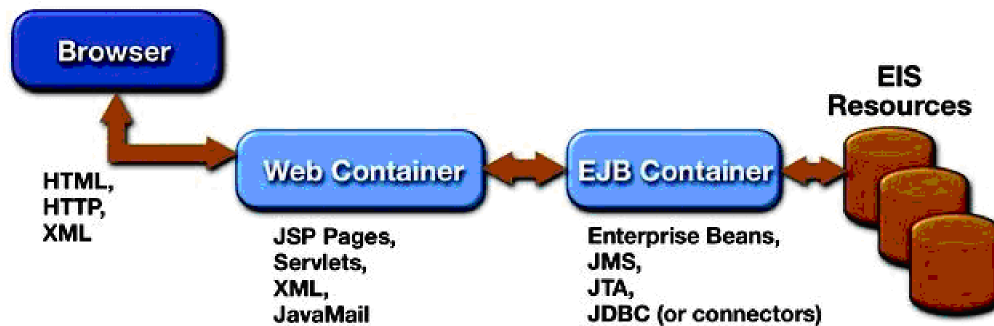

Figura 3.4: Conexión a aplicación desde navegador

3.1.4. Patrones de Diseño

Un patrón describe una solución probada a un problema recurrente en el diseño de software, poniendo énfasis particular en el contexto y las fuerzas que rodean al problema, junto con las consecuencias e impacto de la solución. El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones.

El término patrón se utilizó inicialmente en el campo de la arquitectura por Christopher Alexander a finales de los 70s. Este conocimiento fue aplicado al desarrollo de software orientado a objetos, y de ahí a otras actividades relacionadas con el desarrollo de software.

Desde 1964 hasta 1979, Christopher Alexander escribe varios libros acerca del planeamiento urbano y la construcción de edificios. Entre ellos se destaca "A pattern Language: Towns, Building, Construction" (1).

A principios de los 80 se hizo evidente la escasez de arquitectos de diseño orientado a objetos. La comunidad académica no proporcionaba el conocimiento necesario para lidiar con los requerimientos cambiantes de los proyectos. Este conocimiento requería años de experiencia, pero las demandas de la industria no favorecían la demora necesaria para que los arquitectos instruyesen a sus colegas. Era necesario un modo de difundir este conocimiento.

En 1987, Ward Cunningham y Kent Beck diseñaban interfaces de usuario con Smaltalk. Decidieron, para ello, utilizar alguna de las ideas de Alexander para desarrollar un lenguaje

de patrones que sirviese de guía a los programadores de Smaltalk. El resultado fue el paper "Using Pattern Languages for Object-Oriented Programs" (2).

Posteriormente desde 1990 a 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (GoF)[2], realizaron un catálogo de patrones de diseño para desarrollar aplicaciones de software del cual surge el libro "Design Patterns: Elements of Reusable Object-Oriented Software" (3) .

A este libro se le suele llamar GoF, por el sobrenombre de sus autores. La reacción a este libro fue universalmente positiva y se convirtió en una obra fundamental en la materia. Muchos arquitectos reconocieron en las soluciones propuestas descripciones a las que ellos mismos habían llegado independientemente.

Los patrones de diseño se catalogan en 3 grandes categorías basadas en su propósito: creacionales, estructurales y de comportamiento.

Creacionales: Patrones creacionales tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.

Estructurales: Los patrones estructurales describen cómo las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.

Comportamiento: Los patrones de comportamiento nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

Algunos de los patrones que fueron utilizados en el desarrollo de las capas de abstracción son los siguientes:

[2] Gang of Four

Business Delegate.

- ***Contexto***

Un sistema multi-capa distribuido requiere invocación remota de métodos para enviar y recibir datos entre las capas. Los clientes están expuestos a la complejidad de tratar con componentes distribuidos.

- ***Problema***

Los componentes de la capa de presentación interactúan directamente con servicios de negocio. Esta interacción directa expone los detalles directos de la implementación del servicio de negocio a la capa de presentación. Como resultado, los componentes de la capa de presentación son vulnerables a los cambios en la implementación de los servicios de negocio.

Además, podría haber una reducción de rendimiento en la red porque los componentes de la capa de presentación que utilizan la API de los servicios de negocio hacen demasiadas invocaciones sobre la red. Esto sucede cuando los componentes de la capa de presentación usan directamente la API subyacente, sin cambiar el mecanismo del lado del cliente o agregar servicios.

Por último, exponer directamente las APIs de servicios al cliente fuerza a éste a tratar con los problemas de red asociados con la naturaleza distribuida de la tecnología Enterprise JavaBeans (EJB).

- ***Causas***

1. Los clientes de la capa de presentación necesitan acceder a servicios de negocio.
2. Diferentes clientes, dispositivos, clientes Web, y programas, necesitan acceder a los servicios de negocio.
3. Las APIs de los servicios de negocio podrían cambiar según evolucionan los requerimientos del negocio.
4. Es deseable minimizar el acoplamiento entre los clientes de la capa de presentación y los servicios de negocio, y así ocultar los detalles de la implementación del servicio.
5. Los clientes podrían necesitar implementar mecanismos de caché para la información del servicio de negocio.
6. Es deseable reducir el tráfico de red entre el cliente y los servicios de negocio.

- ***Solución***

Utilizamos un Business Delegate para reducir el acoplamiento entre los clientes de la capa de presentación y los servicios de negocio. El Business Delegate oculta los detalles de la implementación del servicio de negocio, como los detalles de búsqueda y acceso de la arquitectura EJB.

La figura 3.5 muestra el diagrama de clases que representa al patrón Business Delegate. El cliente solicita al BusinessDelegate que le proporcione acceso al servicio de negocio subyacente. El BusinessDelegate utiliza un LookupService para localizar el componente BusinessService requerido.

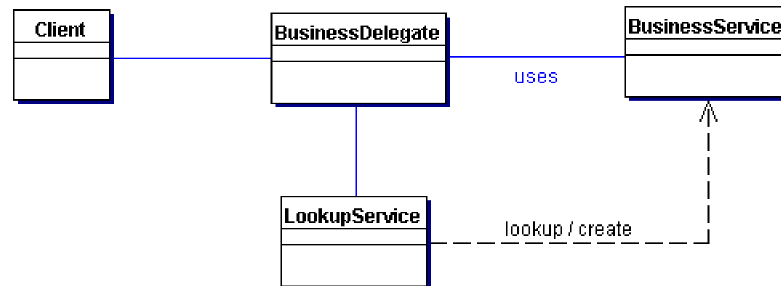

Figura 3.5: Patrón Business Delegate

- ***Consecuencias***

1. Reduce el Acoplamiento, Mejora la Mantenibilidad: El Business Delegate reduce el acoplamiento entre la capas de presentación y de negocio ocultando todos los detalles de implementación de la capa de negocio. Es fácil manejar los cambios porque están centralizados en un sólo lugar, el Business Delegate.

2. Traduce las Excepciones del Servicio de Negocio: El Business Delegate es el responsable de traducir cualquier excepción de red o relacionada con la infraestructura en excepciones de negocio, aislando a los clientes del conocimiento de las especificidades de la implementación.

3. Implementa Recuperación de Fallos y Sincronización de Threads: Cuando el Business Delegate encuentra un fallo en el servicio de negocio, puede implementar características de recuperación automática sin exponer el problema al cliente. Si la recuperación tiene éxito, el cliente no necesita saber nada sobre el fallo. Si el intento de recuperación no tiene éxito, entonces el Business Delegate necesita informar al cliente del fallo. Además, los métodos del Business Delegate podrían estar sincronizados, si fuera necesario.

4. Expone una Interfaz Simple y Uniforme a la Capa de Negocio.

5. Impacto en el Rendimiento: El Business Delegate podría proporcionar servicio de caché (y un mejor rendimiento) a la capa de presentación para las peticiones de servicios comunes.

6. Presenta una Capa Adicional: El Business Delegate podría verse como la adición de una capa innecesaria entre el cliente y el servicio, y con eso incrementar la complejidad y disminuir la flexibilidad. Algunos desarrolladores podrían sentir esto como un esfuerzo extra.

7. Oculta los elementos Remotos: Aunque la localización transparente es uno de los beneficios de este patrón, podría surgir un problema diferente debido a que el desarrollador está tratando con un servicio remoto como si fuera un servicio local. Esto podría suceder si el desarrollador del cliente no entiende que el Business Delegate es cliente-proxy a un servicio remoto.

Service Locator.

- ***Contexto***

La búsqueda y creación de servicios implican interfaces complejas y operaciones de red.

- ***Problema***

Los clientes J2EE interactúan con componentes de servicio, como componentes JavaBeans Enterprise (EJB) y Java Message Service (JMS), que proporcionan servicios de negocio y capacidades de persistencia. Para interactuar con estos componentes, los clientes deben localizar el componente de servicio (referido como una operación de búsqueda) o crear un nuevo componente. Por ejemplo, un cliente EJB debe localizar el objeto home del bean enterprise, que el cliente puede utilizar para encontrar un objeto o para crear uno o más beans enterprise. De forma similar, un cliente JMS primero debe localizar la Fabrica de Conexiones JMS para obtener una Conexión JMS o una Sesión JMS.

Por eso, localizar un objeto servicio administrado por JNDI es un tarea común para todos los clientes que necesiten acceder al objeto de servicio. Por ejemplo, es fácil ver que muchos tipos de clientes utilizan repetidamente el servicio JNDI, y que el código JNDI aparece varias veces en esos clientes. Esto resulta en una duplicación de código innecesaria en los clientes que necesitan buscar servicios.

- ***Causas***

1. Los clientes EJB necesitan utilizar el API JNDI para buscar objetos EJBHome utilizando el nombre registrado del bean enterprise.
2. Los clientes JMS necesitan utilizar el API JNDI para buscar componentes JMS utilizando los nombres registrados en JDNI para esos componentes JMS.
3. La fábrica de contextos utilizada para la creación del contexto inicial JNDI la proporciona el vendedor del proveedor del servicio y por lo tanto es dependiente del vendedor. La fábrica de contexto también es dependiente del tipo de objeto que se está buscando. El contexto para un JMS es diferente que el contexto para EJBs, con diferentes proveedores.
4. La búsqueda y creación de componentes de servicio podría ser compleja y se podría utilizar repetidamente en múltiples clientes en la aplicación.
5. La creación del contexto inicial y el servicio de búsqueda de objetos, si se utiliza frecuentemente, puede consumir muchos recursos e impactar en el rendimiento de la aplicación. Esto es especialmente cierto si los clientes y los servicios están localizados en diferentes capas.

6. Los clientes EJB podrían necesitar reestablecer conexiones a un bean enterprise al que se ha accedido préviamente, teniendo solamente su objeto Handle.

- ***Solución***

Utilizar un objeto Service Locator para abstraer toda la utilización JNDI y para ocultar las complejidades de la creación del contexto inicial, de búsqueda de objetos home EJB y de re-creación de objetos EJB. Varios clientes pueden reutilizar el objeto Service Locator para reducir la complejidad del código, proporcionando un punto de control, y mejorando el rendimiento proporcionando facilidades de caché.

Este patrón reduce la complejidad del cliente que resulta de las dependencias del cliente y de la necesidad de realizar los procesos de búsqueda y creación, que consumen muchos recursos. Para eliminar estos problemas, este patrón proporciona un mecanismo para abstraer todas las dependencias y detalles de red dentro del Service Locator.

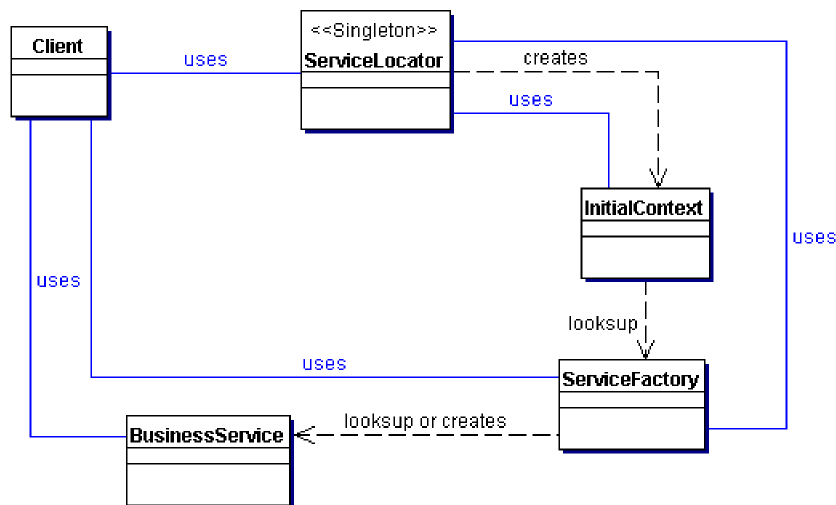

Figura 3.6: Patrón Service Locator

- ***Consecuencias***

1. Abstrae la Complejidad: El patrón Service Locator encapsula la complejidad de este proceso de búsqueda y creación (descrito en el problema) y lo mantiene oculto del cliente. El cliente no necesita tratar con la búsqueda de componentes ni fabricas de objetos (EJBHome, QueueConnectionFactory, y TopicConnectionFactory, entre otros) porque se ha delegado esta responsabilidad en el ServiceLocator.

2. Proporciona a los Clientes un Acceso Uniforme a los Servicios: El patrón Service Locator abstrae todas las complejidades, como acabamos de ver. Haciendo esto, proporciona una interfaz precisa que todos los clientes pueden utilizar. Esta interfaz asegura que todos los tipos de clientes de la aplicación acceden de forma uniforme a los objetos de negocio, en términos de búsqueda y creación. Esta uniformidad reduce la sobrecarga de desarrollo y mantenimiento.

3. Facilita la Adición de Nuevos Componentes de Negocio: Como los clientes de beans enterprise no se preocupan de los objetos EJBHome, es posible añadir nuevos objetos EJBHome para beans enterprise y desplegarlos posteriormente sin impactar en los clientes. Los clientes JMS no se preocupan directamente de las fábricas de conexiones JMS, por eso se pueden añadir nuevas fábricas sin impactar en los clientes.

4. Mejora el Rendimiento de la Red: Los clientes no están implicados en la búsqueda JNDI y la creación de objetos (factory/home). Como el Service Locator realiza este trabajo, puede asumir las llamadas de red requeridas para buscar y crear objetos de negocio.

5. Mejora el Rendimiento del Cliente mediante el Caché: El Service Locator puede poner en un caché los objetos y referencias a objetos del contexto inicial para eliminar actividad JNDI innecesaria que ocurre cuando se obtiene el contexto inicial u otro objetos. Esto mejora el rendimiento de la aplicación.

Data Access Object (DAO).

- ***Contexto***

El acceso a los datos varía dependiendo de la fuente de los datos. El acceso al almacenamiento persistente, como una base de datos, varía en gran medida dependiendo del tipo de almacenamiento (bases de datos relacionales, bases de datos orientadas a objetos, ficheros planos, etc.) y de la implementación del vendedor.

- ***Problema***

Muchas aplicaciones de la plataforma J2EE en el mundo real necesitan utilizar datos persistentes en algún momento. Para muchas de ellas, este almacenamiento persistente se implementa utilizando diferentes mecanismos, y hay marcadas diferencias en los APIS utilizados para acceder a esos mecanismos de almacenamiento diferentes. Otras aplicaciones podrían necesitar acceder a datos que residen en sistemas diferentes. Por ejemplo, los datos podrían residir en sistemas mainframe, repositorios LDAP, etc. Otro ejemplo es donde los datos los proporcionan servicios a través de sistemas externos como los sistemas de integración negocio-a-negocio (B2B), servicios de tarjetas de crédito, etc.

Las aplicaciones pueden utilizar el API JDBC para acceder a los datos en un sistema de control de bases de datos relacionales (RDBMS). Esta API permite una forma estándar de acceder y manipular datos en un almacenamiento persistente, como una base de datos relacional. La API JDBC permite a las aplicaciones J2EE utilizar sentencias SQL, que son el método estándar para acceder a tablas RDBMS. Sin embargo, incluso dentro de un entorno RDBMS, la sintaxis y formatos actuales de las sentencias SQL podrían variar dependiendo de la propia base de datos en particular.

Cuando los componentes de negocio necesitan acceder a una fuente de datos, pueden utilizar la API apropiada para conseguir la conectividad y manipular la fuente de datos. Pero introducir el código de conectividad y de acceso a datos dentro de estos componentes genera un fuerte acoplamiento entre los componentes y la implementación de la fuente de datos. Dichas dependencias de código en los componentes hace difícil y tedioso migrar la aplicación de un tipo de fuente de datos a otro. Cuando cambia la fuente de datos, también deben cambiar los componentes para manejar el nuevo tipo de fuente de datos.

- ***Causas***

1. Los componentes como los beans de entidad controlados por el bean, los beans de sesión, los servlets, y otros objetos como beans de apoyo para páginas JSP necesitan recuperar y almacenar información desde almacenamientos persistentes y otras fuentes de datos como sistemas legales, B2B, LDAP, etc.

2. Las APIs para almacenamiento persistente varían dependiendo del vendedor del producto. Otras fuentes de datos podrían tener APIS que no son estándar y/o propietarios. Estas APIs y sus capacidades también varían dependiendo del tipo de almacenamiento ya sea bases de datos relacionales, bases de datos orientadas a objetos, documentos XML, ficheros planos, etc. Hay una falta de APIs uniformes para corregir los requerimientos de acceso a sistemas tan dispares.

3. Los componentes normalmente utilizan APIs propietarios para acceder a sistemas externos y/o legales para recuperar y almacenar datos.

4. La portabilidad de los componentes se ve afectada directamente cuando se incluyen APIs y mecanismos de acceso específicos.

5. Los componentes necesitan ser transparentes al almacenamiento persistente real o la implementación de la fuente de datos para proporcionar una migración sencilla a diferentes productos, diferentes tipos de almacenamiento y diferentes tipos de fuentes de datos.

- ***Solución***

Utilizar un Data Access Object (DAO) para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos.

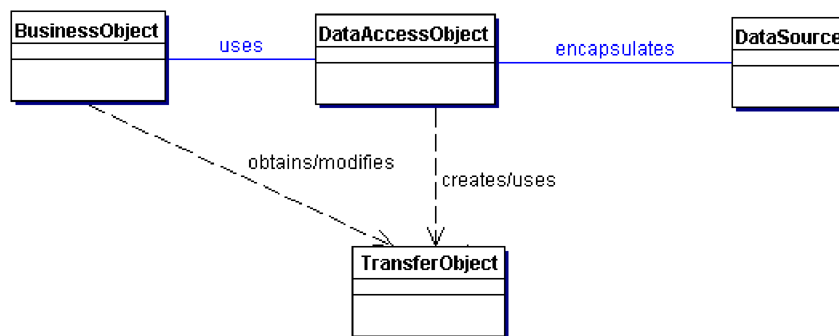

Figura 3.7: Patrón DAO

- ***Consecuencias***

1. Permite la Transparencia: Los objetos de negocio puede utilizar la fuente de datos sin conocer los detalles específicos de su implementación. El acceso es transparente porque los detalles de la implementación se ocultan dentro del DAO.

2. Permite una Migración más Fácil: Una capa de DAOs hace más fácil que una aplicación pueda migrar a una implementación de base de datos diferente. Los objetos de negocio no conocen la implementación de datos subyacente, la migración implica cambios sólo en la capa DAO. Además, si se emplea la estrategia de fábricas, es posible proporcionar una implementación de fábricas concretas por cada implementación del almacenamiento subyacente. En este caso, la migración a un almacenamiento diferente significa proporcionar a la aplicación una nueva implementación de la fábrica.

3. Reduce la Complejidad del Código de los Objetos de Negocio: Como los DAOs manejan todas las complejidades del acceso a los datos, se simplifica el código de los objetos de negocio y de otros clientes que utilizan los DAOs. Todo el código relacionado con la implementación (como las sentencias SQL) están dentro del DAO y no en el objeto de negocio. Esto mejora la lectura del código y la productividad del desarrollo.

4. Centraliza Todos los Accesos a Datos en una Capa Independiente: Como todas las operaciones de acceso a los datos se ha delegado en los DAOs, ésto se puede ver como una capa que aísla el resto de la aplicación de la implementación de acceso a los datos. Esta centralización hace que la aplicación sea más sencilla de mantener y de manejar.

5. No es útil para Persistencia Manejada por el Contenedor: Como el contenedor EJB maneja los beans de entidad con persistencia manejada por el contenedor (CMP), sirve automáticamente todo el acceso al almacenamiento persistente. Las aplicacións que utilizan este tipo de beans no necesitan la capa DAO, ya que el servidor de aplicaciones proporciona de forma transparente esta funcionalidad. Sin embargo, los DAOs aún son útiles cuando se necesita una combinación de CMP (para beans de entidad) y BMP (para beans de sesión, servlets).

6. Añade una Capa Extra: Los DAOs crean un capa de objetos adicional entre el cliente y la fuente de datos que necesitamos diseñar e implementar para obtener los beneficios de este patrón. Pero para obtener estos beneficios debemos pagarlos con un esfuerzo adicional.

7. Necesita Diseñar un Árbol de Clases: Cuando se utiliza una estrategia de fábricas, necesitamos diseñar e implementar el árbol de fábricas concretas y el árbol de productos concretos producidos por las fábricas. Necesitamos justificar este esfuerzo adicional para ver si merece la pena dicha flexibilidad. Esto incrementa la complejidad del diseño. Sin embargo, podemos elegir la implementación de la estrategia de fábricas empezando primero con el patrón Factory Method, y luego avanzar hasta el patrón Abstract Factory si es necesario.

Para efectos de dar mayor agilidad al desarrollo de este tema, algunos patrones como Factory Method, Singleton, Command y Proxy, los cuales fueron utilizados en el transcurso del desarrollo de las capas de abstracción de este tema de memoria, no han sido presentados en este punto. Para mayor información al respecto de éstos y otros patrones, se puede revisar la bibliografía mencionada en la introducción de este punto así como de los patrones J2EE de Sun (11).

3.2. Capa de Abstracción de Servicios (EJBProxy)

3.2.1. Introducción y Conceptos

La invocación remota de procedimientos o RPC[3] es una tecnología que permite que un programa en una máquina sea capaz de invocar una rutina o procedimiento que está almacenado en otro espacio de direcciones (comúnmente en otro computador o bien en otra red) sin que el programador de manera explícita utilice detalles particulares para esta invocación remota. Esto permite que el programador esencialmente escriba código sin tener noción de si el programa está invocando rutinas locales o bien remotas.

Usualmente cuando el software en cuestión está usando principios de orientación a objetos, nos estamos refiriendo a invocación remota de métodos.

El Origen de los Servicios Remotos

La idea de RPC surge a partir del año 1976 cuando se escribe el RFC 707 (19). Uno de los primeros fabricantes que usaron RPC fue Xerox, bajo el nombre de "Courier" en el año 1981 (20). La primera implementación popular de RPC bajo Unix fue Sun RPC ahora llamada ONC RPC, la cual fue usada como base para el sistema NFS de Sun. Hasta el día de hoy ONC RPC tiene amplio uso en la industria en diversas plataformas.

Otra implementación temprana de RPC fue *Apollo Computer's Network computing Systems* (NCS). Tiempo más tarde NCS fue usada como base de DCE/RPC en el ambiente de computación distribuida de OSF (21). Una década más tarde Microsoft adoptó DCE/RPC como base para su mecanimo de RPC, y la subsiguiente implementación DCOM sobre él. A mediados de los 90's, el *Object Management Group* (22) crea CORBA (23) el cual ofrece un nuevo mecanismo de RPC basado en objetos distribuidos con un mecanismo de herencia.

Con el paso del tiempo y el boom de Java, llegó RMI, el cual es un mecanismo ofrecido en Java para invocar un método remotamente. Al ser RMI parte estándar del entorno de ejecución Java, usarlo provee un mecanismo simple en una aplicación distribuida que solamente necesita comunicar servidores codificados para Java. RMI es la base fundamental para la creación de EJB dentro de la especificación J2EE.

3.2.2. Estado del Arte

Dados los pasos históricos, el estado del arte respecto de la creación y presentación de servicios remotos es la tecnología Enterprise Java Beans (EJB). En la actualidad la versión de EJB 3.0 es una profunda simplificación de la especificación anterior EJB. Sus objetivos son simplificar el desarrollo, facilitar el desarrollo de pruebas (unitarias y de integración) y enfocarse más en

[3] RPC: Remote Procedure Call

el desarrollo de objetos de negocios que sean objetos planos Java más que en la complejidad de la API de EJB.

Para ello EJB 3.0 se aprovecha principalmente de las nuevas características que ofrece Java en su versión 5.0, principalmente el uso de anotaciones, las cuales simplifican la API para los EJB de entidad (CMP) usando el nuevo mecanismos de persistencia creado para EJB 3.0 es cual es Java Persistence API (JPA). Junto con lo anterior, existe la posibilidad de usar esta nueva característica de persistencia sin necesidad de usar un contenedor J2EE, lo que otorga la capacidad de poder tener EJB de session o de mensajería fuera de un contenedor J2EE en programas standalone, test unitarios o bien en contenedor web.

3.2.3. Tecnologías Alternativas

3.2.3.1. Jini Network Technology

Jini (5) es una tecnología desarrollada por Sun Microsystems (8), que proporciona un mecanismo sencillo para que diversos dispositivos conectados a una red puedan colaborar y compartir recursos sin necesidad de que el usuario final tenga que planificar y configurar dicha red. En esta red de equipos, llamada "*comunidad*", cada uno proporciona a los demás los servicios, controladores e interfaces necesarios para distribuirse de forma óptima la carga de trabajo o las tareas que deben realizar.

Jini ha sido desarrollado aprovechando la experiencia y muchos de los conceptos en los que está inspirado el lenguaje Java y, sobre todo, en la filosofía de la Máquina Virtual Java (JVM). Por lo tanto, el Jini puede funcionar sobre potentes estaciones de trabajo, en PCs, en pequeños dispositivos (PDAs, cámaras de fotos, móviles, reproductores mp3, etc).

La base de Jini es un trío de protocolos: discovery, join y lookup. Los protocolos de discovery y lookup se utilizan cuando se conecta un dispositivo a la red, discovery ocurre cuando un servicio busca un servicio lookup donde pueda registrarse y join ocurre cuando un servicio localiza un servicio de lookup y desea suscribirse a éste. Lookup ocurre cuando un cliente localiza e invoca un servicio descrito por su interfaz.

Conceptos Claves

La tecnología Jini se compone de una infraestructura y un modelo de programación que determina cómo los dispositivos se conectan con otros para formar la comunidad. Para ello Jini usa métodos de invocación remota de Java (RMI). Los conceptos claves que permiten comprender la tencnología JINI están resumidos a continuación.

- ***Servicios***

Los servicios son la entidad que le dan sentido al sistema distribuido: dispositivos, datos, almacenaje, filtros, cálculos, etc. Todo aquello que pueda ser útil para un usuario u otros servicios.

Para ello, JINI provee mecanismos de creación, búsqueda, comunicación y utilización de los servicios de la red o comunidad. Los servicios se comunican entre sí utilizando un protocolo de servicios, un conjunto de interfaces implementados en Java.

- ***Lookup Service***

El Lookup Service es un directorio de servicios donde los servicios se inscriben y los usuarios pueden consultar por los servicios disponibles. El Lookup Service puede a su vez, contener otros Lookup Service, formando una búsqueda jerárquica.

Los servicios nuevos deben añadirse a un Lookup Service y lo hacen gracias a los protocolos discovery y join. El servicio que quiere agregarse al sistema busca un Lookup Service mediante el protocolo discovery y una vez que encuentra el Lookup apropiado, se une a él con join.

- ***Java Remote Method Invocation (RMI)***

RMI o método de invocación remota de Java permite que un objeto de Java sea invocado desde otro objeto o clase remota, lo que supone poder compartir todo tipo de objetos Java a través de la red.

En RMI se define una interfaz con los métodos que serán invocados remotamente. Luego se implementa en una clase y esa clase se registra en el lado del servidor. A partir de ese momento, una clase remota puede llamar a los métodos que se ofrecen.

Para lograr la comunicación entre los objetos remotos, se debe post-procesar la clase que implementa la interfaz remota, generando dos archivos, el stub y el skel que son imprescindibles para la comunicación entre objetos, ya que el traspaso de código se hace gracias a ellos. El archivo que se utiliza al lado del cliente es el *stub* mientras que el *skel* es necesario al lado del servidor.

RMI forma parte de la infraestructura de JINI, ya que facilita la comunicación entre servicios; gracias a los mecanismos RMI podemos buscar, activar e invocar métodos de otros objetos que se encuentran en la red o sistema.

- ***Leasing***

Un sistema como JINI requiere organización en cuanto al uso de recursos de los cuales dispone. JINI utiliza el Leasing para asignar un tiempo concreto para la utilización de un servicio. Ese tiempo se negocia entre el usuario y el proveedor del servicio (es parte del protocolo), tras lo cual se asigna un espacio de tiempo. Una vez agotado ese tiempo, el usuario puede renovar el arriendo por otro período de tiempo o dejar de utilizarlo.

- ***Transacciones***

Las operaciones entre uno o varios servicios pueden ser contenidas en una transacción. Las Transacciones son interfaces de JINI que ofrecen un protocolo para coordinar un commit de dos fases. Esas transacciones son responsabilidad del servicio que las utiliza.

- ***Eventos***

Los servicios y los usuarios de JINI pueden utilizar eventos, que en este caso son distribuidos por la red. Algunos ejemplos del uso de eventos remotos son: Ingreso de nuevos servicios al sistema; notificación de un servicio a los usuarios; etc.

Componentes del Sistema

Los componentes del Sistema Jini pueden segmentarse en 3 categorías: infraestructura, modelo de programación y servicios. La infraestructura es el conjunto de componentes que permiten construir un sistema Jini, mientras que los servicios son entidades dentro de este sistema. El modelo de programación es el conjunto de interfaces que permiten la construcción de servicios confiables, incluídas aquellas que son parte de la infraestructura y también las que participan del sistema.

Un sistema Jini puede ser visto como una extensión de una infraestructura de red, modelo de programación y servicios hechos en Java pero en el caso de una sola máquina. La representación de estas categorías en el modelo queda visualizada en la siguiente figura.

	Infrastructure	Programming Model	Services
Base Java	Java VM RMI Java Security	Java APIs JavaBeans™ ...	JNDI Enterprise Beans JTS ...
Java + Jini	Discovery/Join Distributed Security Lookup	Leasing Transactions Events	Printing Transaction Manager JavaSpaces™ Service ...

Figura 3.8: Categorías

- **Infraestructura**

La infraestructura de la tecnología Jini define una base mínima que incluye:

- Un Sistema de seguridad distribuido integrado a RMI, el cual extiende el modelo de seguridad de la tecnología Java al mundo de sistemas distribuidos
- Los protocolos de *discovery/join*, los cuales permiten a los servicios (tanto como software o hardware) descubrir, ser parte y anunciar servicios a otros miembros de la comunidad.
- El *Lookup Service*, que actúa como repositorio de servicios. Cada entrada en el lookup service son objetos Java, los cuales pueden ser descargados como parte de la operación de lookup y actuar como proxies locales del servicio que está en el lookup service.

Los protocolos de *discovery/join* definen la manera en que un servicio forma parte de una comunidad Jini y RMI define el lenguaje base con el cual los servicios Jini se comunican.

- **Modelo de Programación**

La programación en un sistema JINI se basa en transacciones, arriendo de servicios o *leasing* y eventos distribuidos. Cuando un servicio es añadido o quitado del sistema, los demás miembros del sistema reciben la correspondiente notificación del evento. El uso de los servicios se reparte según el leasing que tenga cada servicio, y una vez conseguido un servicio, las transacciones se ocupan de la comunicación entre usuario y servicio.

Para unir esta infraestructura del sistema y los servicios del nivel superior, JINI posee una serie de interfaces (otros servicios); estas interfaces forman una extensión del modelo de programación distribuida de Java, y constituyen el modelo de programación JINI. Las interfaces básicas son: leasing, eventos y transacciones.

- Leasing

La interfaz Leasing, un modelo de utilización de recursos basado en la duración de un tiempo determinado; está definido de tal forma de que los recursos del sistema se arriendan al usuario durante un lapso de tiempo, para luego ser renovados o liberados. Esta interfaz extiende el modelo tradicional de Java añadiéndole la propiedad del tiempo a las referencias a un objeto o recurso; esas referencias pueden ser renovadas.

- Eventos

La interfaz de eventos y notificaciones, permite la comunicación entre servicios JINI basado en eventos; esta interfaz no es más que una extensión del modelo de eventos utilizado por las componentes gráficas. Gracias a esta extensión del modelo de eventos, se puede hacer que los eventos tengan su respuesta o soporte por parte de objetos remotos; también se tienen en cuenta detalles como el posible retraso en las notificaciones distribuidas.

- Transacciones

La interfaz de transacciones asegura que las distintas entidades trabajen de tal forma que todas las operaciones tengan la propiedad de ser atómicas: o se ejecutan todas o ninguna. Esta interfaz ofrece un sencillo protocolo orientado a objetos para permitir a las aplicaciones JINI coordinar sus cambios de estado. El protocolo consiste en dos sencillos pasos: en el primero, la fase de votación, cada objeto vota si cree que ha completado su parte de tarea y esta preparado para hacer un commit de los cambios que ha realizado. En el segundo paso, un coordinador da un commit request a cada objeto. La responsabilidad de una correcta implementación de las transacciones queda en manos de cada objeto que forma parte del sistema y de la transacción en particular. El objetivo final del protocolo de transacciones JINI no es otro que definir las interacciones que tales objetos deben tener para coordinar las operaciones que se lleven a cabo.

- **Servicios**

La infraestructura y el modelo de programación de Jini fueron construidos para permitir que servicios puedan ser publicados y encontrados en la comunidad. Estos servicios hacen uso de la infraestructura para hacer llamadas entre ellos, descubrirse unos a otros, y para anunciar su presencia a otros servicios y usuarios.

Programáticamente, los servicios aparecen como objetos escritos en el lenguaje de programación Java. Un servicio tiene una interfaz que define sus operaciones que pueden ser requeridas. Algunas de estas interfaces están destinadas para ser usadas por otros programas, así como otras están destinadas para ser ejecutadas por un receptor de manera tal que el servicio tenga interacción con algún usuario.

El tipo de servicio determina la interfaz que se tendrá para él, y también define el conjunto de métodos que pueden ser utilizados en este servicio.

El corazón de Jini es un trío de protocolos llamados *discovery, join, and lookup.* Discovery ocurre cuando un servicio está buscando algún lookup service donde poder registrarse. Join ocurre cuando un servicio ha ubicado un lookup service y quiere unirse a él. Lookup ocurre cuando un cliente o usuario ubica e invoca un servicio descrito por una interfaz. El siguiente diagrama muestra el proceso de discovery (discovery, figura 3.9).

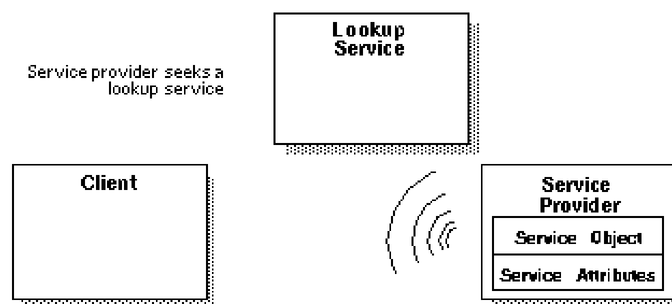

Figura 3.9: Proceso de discovery

Discovery/Join es el proceso de agregar nuevos servicios a una comunidad Jini. Un proveedor de un servicio (un dispositivo de hardware o software) ubica un lookup service vía multicast en la red local para poder identificarse (discovery, figura 3.9). Luego, un objeo de servicio es cargado dentro del lookup service (join, figura 3.10). Este objeto de servicio contiene la interfaz para los métodos que los usuarios y aplicaciones podrán invocar para ejecutar el servicio, junto con otros atributos descriptivos.

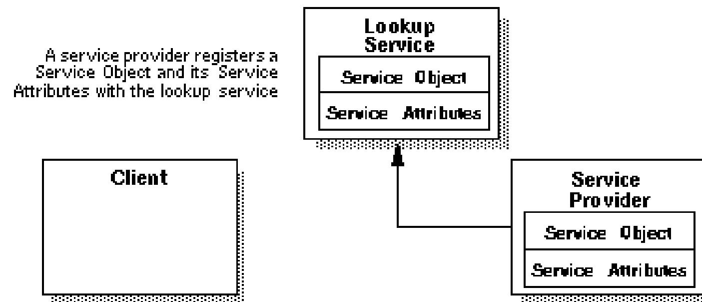

Figura 3.10: Proceso de join

Un servicio debe ser capaz de encontrar un lookup service, sin embargo, un servicio puede delegar la tarea de encontrar un lookup service a un tercero. El servicio ahora está listo para ser encontrado (lookup) y usado, como se muestra en la siguiente figura (lookup, Figura 3.11).

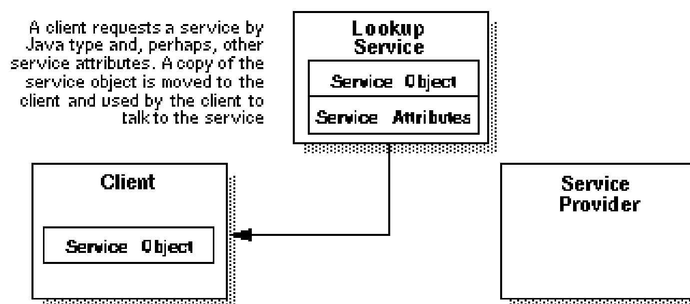

Figura 3.11: Proceso de lookup

Un cliente localiza un servicio apropiado por su tipo, es decir, por su interfaz escrita en el lenguaje de programación de Java y los atributos que serán usados en la interfaz de usuario por el lookup service. El objeto de servicio se carga dentro del cliente y finalmente se invoca el servicio como se muestra en la figura 4.5. De este modo el cliente interactúa directamente con el proveedor del Servicio vía el objeto de servicio.

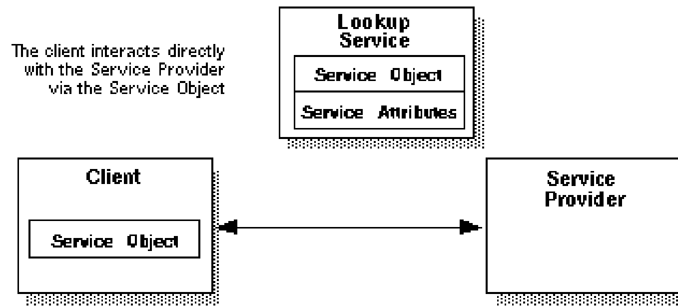

Figura 3.12: Invocación del servicio

La posibilidad de mover objetos y códigos desde el proveedor del servicio al lookup service y desde ahí al cliente del servicio da una gran libertad al proveedor del servicio en su comunicación entre el servicio y sus clientes. De este modo, el cliente sólo conoce que está negociando con una implementación de una interfaz, por lo tanto el código que implementa esta interfaz puede hacer todo lo necesario para proveer el servicio. Dado que el código viene originalmente del servicio en sí, el código puede tomar esta ventaja en detalles de implementación que sólo el servicio conoce.

3.2.3.2. Spring

Spring (26) es una plataforma de código abierto creada por Rod Johnson (24) la cual tiene por objetivo simplificar la complejidad del desarrollo de aplicaciones de tipo empresarial. Spring hace posible usar clases planas Java para poder lograr objetivos que solamente eran posibles de lograr sólo usando tecnología EJB, con la consiguiente complejidad que conlleva su uso.

Además, la utilidad de esta plataforma no está solamente limitada al desarrollo de componente de lado de servidor, sino también para cualquier aplicación Java. Spring fomenta el diseño de aplicaciones simples, que se puedan probar de manera unitaria y con componentes altamente desacoplados.

El corazón de Spring es un contenedor liviano de *inversión del control* también conocido como *inyección de dependencias*. A continuación se presentan algunas de las características más notables de Spring:

- Liviano

Spring es liviano en términos de tamaño y “*overhead*”[4]. La plataforma completa de Spring se puede distribuir en un solo archivo empaquetado el cual tiene un tamaño cercano a 1 MB. El proceso de overhead que requiere Spring es insignificante. Es más, Spring no es intrusivo ya que los objetos para que tengan la capacidad de usar spring no dependen de clases específicas de Spring.

[4] *overhead* en términos computacionales se considera al uso indirecto en tiempo de cpu, memoria, ancho de banda, u otro recurso que es necesario utilizar para lograr un objetivo

- Inversion del control

Spring promueve el desacoplamiento de los objetos gracias a una técnica conocida como *inversión del control* (IoC). Cuando se aplica IoC, los objetos obtienen sus dependencias de manera pasiva, sin preocuparse de la manera de cómo se obtienen o se crean esas dependencias. Uno puede pensar en IoC como una inversa de JNDI, en vez de que el objeto busque sus dependencias en el contenedor, el contenedor le entrega las dependencias al objeto en sí, instancia la clase que se necesita sin necesidad de consultar al objeto.

- Orientado a Aspectos

Spring tiene un gran soporte para la *programación orientada a aspectos* la cual permite separar la lógica de negocios de la aplicación de aspectos como servicios de auditoría, transaccionabilidad, seguridad y otros. La idea es que los objetos de la aplicación se preocupen de la lógica de negocios y nada más. Ellos no deberían ser responsables (o incluso estar al tanto) de otros aspectos de las aplicaciones como auditoría o seguridad.

- Contenedor

Spring es un contenedor en el sentido de que contiene y administra el ciclo de vida y configuración de los objetos de la aplicación. Uno puede configurar cómo cada objeto debe ser creado, por ejemplo como sólo 1 instancia o bien que se cree una instancia cada vez que sea necesario. Además maneja la configuración en el sentido de que se encarga de inyectar las dependencias entre objetos e instanciarlos cada vez que sea necesario, en conjunto con hacer las asociaciones correspondientes entre ellos.

- Plataforma

Spring hace posible configurar y "*componer*" aplicaciones complejas a partir de simples componentes. En Spring, los objetos de las aplicaciones se componen de manera declarativa, típicamente en un archivo XML. Spring también provee una insfraestructura de funcionalidades extras tales como manejo de transacciones, integración con plataformas de persistencia, sistemas empresariales como servicios de mail, servicios de programación y de mensajería.

Todos estos atributos permiten escribir código más limpio, más mantenible y fácil de probar. Spring también posee una variedad de sub-plataformas integradas, las cuales permiten al desarrollador simplificar su código al utilizarlas. Spring posee plataformas integradas para persistencia, servicios remotos y para el desarrollo de interfaces web. En particular la integración de Spring para dar soporte para servicios remotos son las siguientes:

- Remote Method Invocation (RMI). A través del uso de RmiProxyFactoryBean y de RmiServiceExporter Spring da soporte para RMI y su interfaz java.rmi.Remote y sus respectivas excepciones remotas java.rmi.RemoteException. Además permite de manera transparente la invocación a través de cualquier interfaz Java estos servicios RMI.

- Spring's HTTP invoker. Spring provee una estrategia especial de exposición de servicios remotos la cual permite el uso de la serialización de objetos Java vía HTTP, soportando para ello cualquier interfaz Java normal (al igual que en RMI). Las clases correspondientes para ello son HttpInvokerProxyFactoryBean y HttpInvokerServiceExporter.
- Hessian. Usando HessianProxyFactoryBean y HessianServiceExporter uno de manera transparente puede exponer los servicios usando este protocolo basado en HTTP el cual provee Caucho (25).
- Burlap. Burlap es una alternativa basada en XML que también provee Caucho la cual es alternativa a Hessian. Spring da soporte a esta tecnología con las clases BurlapProxyFactoryBean y BurlapServiceExporter.
- JAX RPC. Spring da soporte remoto para web services vía JAX-RPC.
- JMS. El soporte JMS que provee se da a través de las clases JmsInvokerServiceExporter y JmsInvokerProxyFactoryBean.

Para ejemplificar la facilidad de uso de Spring para la exposición de servicios remotos, se mostrará un ejemplo de código basado en la exposición de servicios vía RMI. Para el resto de los casos es bastante similar, por lo que se recomienda al lector ver la documentación correspondiente de Spring Remoting (16). El modelo de dominio y servicios para este ejemplo serán los siguientes:

```
public class Account implements Serializable{
  private String name;

  public String getName();

  public void setName(String name) {
    this.name = name;
  }
}
```

Figura 3.13: Clase Cuenta

```
public interface AccountService {

  public void insertAccount(Account account);

  public List getAccounts(String name);
}
```

Figura 3.14: Interfaz de Servicio de Cuenta

```
public class AccountServiceImpl implements AccountService {

  public void insertAccount(Account acc) {
    // do something...
  }

  public List getAccounts(String name) {
    // do something...
  }
}
```

Figura 3.15: Implementación de Servicio de Cuenta

- Exportando el servicio usando RmiServiceExporter

Usando RMIServiceExporter, se puede exponer la interfaz de nuestro servicio AccountService como un objeto RMI. A esta interfaz se puede acceder usando RmiProxyFactoryBean, o vía RMI normal en caso de un servicio tradicional RMI. RmiServiceExporter de manea explícita da soporte a cualquier servicio no RMI vía invocadores RMI.

Para ello, primero debemos configurar nuestro servicio en el contenedor Spring

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

Figura 3.16: Configuración de Servicio

El paso siguiente es exponer el servicio como servicio remoto, para ello usamos RmiService-Exporter:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
        <!-- does not necessarily have to be the same name as the bean to be exported -->
        <property name="serviceName" value="AccountService"/>
        <property name="service" ref="accountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
        <!-- defaults to 1099 -->
        <property name="registryPort" value="1199"/>
</bean>
```

Figura 3.17: Exposición del servicio como remoto

Como se puede ver, se está sobreescribiendo el puerto del registro de RMI. usualmente, un servidor de aplicaciones mantiene un registro RMI y es recomendado no interferir con el puerto de éste. Para tomar en cuenta, el nombre del servicio se usa para poder hacer el enlace dentro de Spring. Por lo tanto, el servicio será expuesto en la dirección url *'rmi://HOST:1199/AccountService'*. Esta dirección será usada para hacer el enlace del servicio con el cliente.

- Enlazando el servicio con el cliente

Nuestro cliente es un objeto simple, que usa el servicio de cuenta (AccountService) para manejar cuentas.

```
public class SimpleObject {

  private AccountService accountService;

  public void setAccountService(AccountService accountService) {
    this.accountService = accountService;
  }
}
```

Figura 3.18: Cliente RMI

Para poder enlazar el servicio al cliente, necesitamos crear un contenedor Spring separado del contenedor Spring del servidor. Este contenedor debe tener nuestro objeto cliente y el enlace al servicio remoto.

```
<bean class="example.SimpleObject">
        <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

Figura 3.19: Configuración del contenedor en cliente

Eso es todo lo que se necesita para dar soporte remoto a nuestro servicio de cuenta en el cliente. Spring hará de manera transparente la creación del invocador y habilitará el servicio remoto a través de la clase RmiServiceExporter. En el lado del cliente, el enlace se hará usando la clase RmiProxyFactoryBean.

3.3. Capa de Abstracción de Persistencia (ORM)

3.3.1. Introducción y Conceptos

Object-Relational Mapping (tambien conocido como O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre tipos incompatibles de sistemas en bases de datos y lenguajes de programación orientado a objetos. Esto significa que se crea un objeto virtual en base de datos el cual puede ser usado desde el lenguaje de programación.

Existen todo tipo de paquetes comerciales y de código abierto disponibles en la red que implementan esta técnica, aunque también muchas veces los programadores adaptan o bien desarrollan sus propias herramientas ORM.

Descripción del problema

La tarea de manejo de datos en la programación orientada a objetos típicamente se hace manipulando objetos, los cuales en su mayoría no son valores escalares. Por ejemplo consideremos un registro en una libreta de direcciones, el cual representa a una persona con uno o varios números telefónicos y una o varias direcciones. Esto puede ser modelado como un objeto "*persona*" con datos que representan a la persona, tales como nombre, rut, fecha nacimiento, etc, más una lista de números telefónicos y una lista de direcciones. La lista de teléfonos en sí misma puede contener objetos de tipo "*número telefónico*" y así en más. En definitiva la entrada en la libreta de direcciones puede ser tratada como sólo 1 objeto por el lenguaje de programación (en 1 sola variable por ejemplo).

Muchos productos de base de datos solamente puede guardar y manipular valores escalares tales como enteros, caracteres, etc, los cuales están organizado en tablas. Por lo tanto el programador debe convertir estos objetos complejos de la aplicación en grupos de objetos con valores simples y escalares en base de datos o solamente usar valores escalares dentro de la aplicación. Lo anterior no es algo aceptable dentro de un paradigma orientado a objetos, y los productos ORM son los que implementan la primera opción.

La clave del problema es cómo se traducen estos objetos de manera de que puedan ser guardados en base de datos, y más tarde ser recuperados de manera fácil, preservando las propiedades y relaciones entre los objetos.

Críticas

Algunos postulan que promover el uso de herramientas ORM es un síntoma de que se trata de resolver el lado incorrecto del problema. Esto viene del hecho de que los principios de información que establecieron las bases de datos y el modelo relacional implican que la orientación a objetos en sí misma no es adecuada para todas las necesidades de manipulación de datos que se requieren hacer de manera cotidiana. Ello implica que resolver este tema es en definitiva la generación de un nuevo paradigma y que ése es el problema que debería intentar resolverse.

3.3.2. Estado del Arte

El estado del arte en materia de persistencia y ORM se trata de **J**ava **P**ersistence **A**PI o JPA. El surgimiento de esta especificación se da principalmente por 2 razones. Primero, esta especificación simplifica el desarrollo de aplicaciones que usan datos persistentes. Segundo, la idea es que toda la comunidad usuario de Java tenga un estándar único de mecanismo de persistencia.

La especificación JPA surge a partir de las mejores ideas obtenidas de otros ORM tales como Hibernate, TopLink y JDP. Lamentablemente, el uso de cada uno de ellos compromete al desarrollador a escoger entre distintas implementaciones incompatibles entre ellas, por lo que la necesidad de un estándar surge de manera natural.

JPA se origina como parte del trabajo del Java Community Process(JCP) (27) en el requerimiento de especificación 220 (JSR 220) (28) para la simplificación de los EJB de entidad. Rápidamente se puso de manifiesto que la simplificación de los EJB de entidad no era suficiente y que era necesario tener una especificación para una plataforma de persistencia basada en objetos POJO en línea con las tecnologías ORM disponibles en la industria.

JPA es una especificación ORM basada en objetos POJO. Contiene una completa especificación de *mapping* objeto/relacional soportada por el uso del lenguaje de metadatos y anotaciones que posee Java estándar en su versión 5.0 o bien a través de descriptores XML los cuales definen las relaciones entre los objetos Java y la base de datos relacional. Junto con lo anterior ofrece un soporte para querys tipo SQL, las cuales pueden ser dinámicas o bien estáticas. Además posee soporte para el uso de proveedores de persistencia *enchufables*.

Una vez que la especificación estuvo disponible en la red, el grupo de experto encargados del requerimiento JSR-220 recibió muchos pedidos de la comunidad para que este trabajo de persistencia estuviese disponible sin necesidad de tener un contenedor EJB para su uso.

Por el momento no hay planes de incluir JPA dentro de la versión estándar de Java. Sin embargo este tema está siendo considerado por el grupo de expertos encargados de Java estándar para ver si se incorpora en revisiones futuras de Java.

3.3.3. Tecnologías Alternativas

3.3.3.1. iBATIS

iBATIS SQL Maps es un framework open-source sobre JDBC que provee un simple y flexible medio de mover datos entre un objeto java y una base de datos relacional. Esta plataforma ayuda a reducir la cantidad de código Java (JDBC) que es necesario para acceder a una base de datos relacional. El framework permite relacionar un java Bean a una sentencia SQL mediante el uso de archivos XML que permiten crear consultas complejas, inner y outer joins. La belleza de todo esto es que se logra sin manipular tablas especiales de base de datos, bytecode o generación de código.

iBATIS no fue hecho para reemplazar a herramientas ORM (Object Relational Mapping) como Hibernate, OJB, Entity Beans o TopLinks para nombrar algunas. Si no que es un framework de bajo nivel que permite escribir manualmente los comandos SQL y relacionarlos a un objeto Java. Una vez que uno relaciona la capa de persistencia a un modelo de objeto, entonces se está listo para utilizarlo dentro de cualquier aplicación java. Ya no es necesario buscar fuentes de datos, obtener conexiones, crear PreparedStatements, interpretar el ResultSet, o guardar los datos en memoria, pues iBATIS hace todo eso por uno. Resumiendo, iBATIS crea el PreparedStatement, configura los parámetros (si hay alguno), ejecuta los comandos e instancia los objetos utilizando el resultado de una consulta SQL. Si el comando SQL fue un insert o un update, entonces el número de columnas afectadas es retornado

Modo de Uso

Primero se deben configurar los archivos SQL Maps. Eso se hace creando un archivo de configuración XML, el cual provee detalle para la fuente de datos (DataSource), SQL Maps y otras opciones como administración de hebras (threads). La figura 3.20 muestra un archivo ejemplo de configuración (**sql-map-config.xml**).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN" "
    http://www.ibatis.com/dtd/sql-map-config-2.dtd"?>

<sqlmapconfig>

    <settings cacheModelsEnabled="false" enhancementEnabled=" true"
        lazyLoadingEnabled="true" maxRequests="32" maxSessions="100"
        maxTransactions="100" useStatementNamespaces="false" />
    <transactionmanager type="JDBC">
        <datasource type="JNDI">
            <property name="context.java.naming.factory.initial"
                value="weblogic.jndi.WLInitialContextFactory" />
            <property name="context.java.naming.provider.url"
                value="t3://localhost:7001" />
            <property name="DataSource" value="jdbc/jpetstoreDS" />
        </datasource>
    </transactionmanager>
    <sqlmap resource="com/j2eegeek/ibatis/dao/maps/Account.xml" />
</sqlmapconfig>
```

Figura 3.20: Archivo sql-map-config.xml

En este caso, el código se subirá a un servidor de aplicaciones que tiene un pool de conexiones, el cual ya está creado en conjunto con una fuente de datos. En el archivo de configuración se hace referencia a la fuente de datos creada en el servidor de aplicaciones, la cual corresponde a *jdbc/jpetstoreDS*

El archivo de configuración incluye una referencia a otro archivo XML, el cual contiene las sentencias SQL para hacer inserciones, modificaciones borrado y consultas, junto con los resultados y las relaciones de los parámetros. En este caso éste archivo se llama **Account.xml**.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlmap namespace="Account">

    <typealias alias="account"
        type="com.j2eegeek.ibatis.domain.Account" />

    <resultmap id="accounts" class="account">
        <result property="userID" column="userid" />
        <result property="email" column="email" />
        <result property="firstName" column="firstname" />
        <result property="lastName" column="lastname" />
    </resultmap>

    <select id="getAccount" resultClass="accounts"
        parameterClass="java.lang.Integer">
        SELECT userid, email, firstname, lastname FROM jpetstore.account
        where userid = #value#
    </select>

</sqlmap>
```

Figura 3.21: Archivo Account.xml

Este archivo es bastante auto explicativo. En él se describe una consulta SQL que toma un número entero como argumento y retorna una instancia de un objeto com.j2eegeek.ibatis.domain.Account. Las funcionalidades de inserción, modificación y borrado trabajan de manera similar, así como también procedimientos almacenados y consultas dinámicas.

La programación de estos archivos SQL Map es bastante directa y provee al desarrollador la habilidad de: relacionar objetos con tablas, ejecutar sentencias SQL, recuperar objetos a partir de sentencias SQL sin necesidad de efectuar conversiones manuales.

Finalmente falta probar un ejemplo de cómo se termina usando esta plataforma una vez configurado todo. Como se aprecia en la figura 3.22, primero es necesario otorgar el archivo de configuración “sql-map-config.xml”, y luego obtener un objeto SQLMapClient. Con este objeto, uno puede ejecutar las sentencias sql dando como parámetros el identificador de la sentencia en el archivo Account.xml y el dato necesario para esa consulta, en este caso un número entero.

```java
try
{
     String sqlMapConfigFile = "sql-map-config.xml";
     Reader sqlMapConfigReader = Resources.getResourceAsReader(sqlMapConfigFile);
     sqlMapClient = SqlMapClientBuilder.buildSqlMapClient(sqlMapConfigReader);

}
catch (IOException e)
{
     log.error(e);
}

try
{
     Account myAccount = (Account)
            sqlMapClient.queryForObject("getAccount", new Integer(1234));
     log.debug("myAccount.getFirstName() = " + myAccount.getFirstName());
}
catch (SQLException e)
{
     e.printStackTrace();
     log.error(e.toString());
}
```

Figura 3.22: Ejemplo de uso de iBATIS

iBATIS SQL Maps es una herramienta bastante poderosa, la cual aprovecha la experiencia y conocimiento de los desarrolladores SQL en conjunto con la flexibilidad de la programación orientada a objetos.

3.3.3.2. Hibernate

Hibernate (14) es una herramienta ORM para Java. Hibernate permite al desarrollador hacer persistentes objetos, siguiendo las propiedades del lenguaje Java tales como asociación, herencia, polimorfismo, composición y todo lo que son Java Collections.

Hibernate no solo provee relaciones entre clases Java y tablas de base de datos (y de tipos de datos Java a tipos de datos SQL), sino también provee recuperación de datos a través de un mecanismo propietario de consulta llamado HQL. Todo esto permite reducir los tiempos de desarrollo de manera drástica evitando el manejo manual de SQL y JDBC.

El objetivo final de Hibernate es poder permitir al desarrollador reducir hasta en un 95 % las tareas comunes de persistencia que perfectamente se pueden automatizar.

Hibernate existe bajo licencia LGPL la cual es suficientemente flexible para permitir usar esta herramienta en proyectos comerciales como en proyectos de código abierto.

La arquitectura de alto nivel de Hibernate se describe en la figura 3.23.

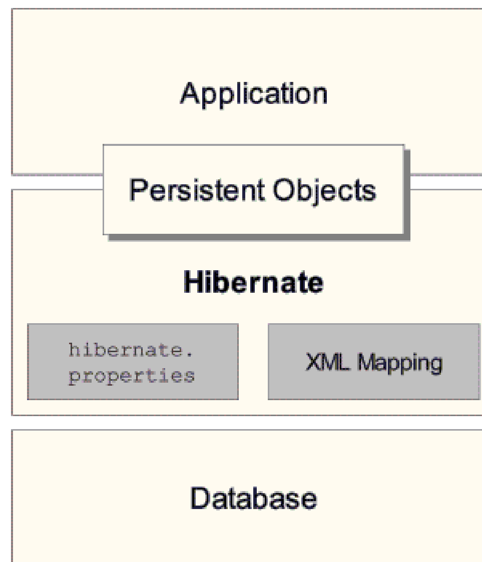

Figura 3.23: Arquitectura Hibernate

Hibernate hace uso de clases POJO java en conjunto con documentos XML que permiten relacionar estos objetos con la capa de base de datos. En vez de utilizar procesamiento de *byte code* o generación de código, Hibernate usa en tiempo de ejecución una característica de Java denominada reflexión, la cual permite hacer instrospección de un objeto en tiempo de ejecución.

Hibernate no está restringido en el uso de ningún tipo de datos Java ni objetos ni primitivos. Todos estos tipos pueden relacionarse, incluso clases que pertenecen a la plataforma de Java Collections. Estas se pueden asociar como valores, arreglos de valores o asociaciones a otras entidades en base de datos. En Hibernate existe una propiedad especial que es el campo id el cual representa el identificador o llave primaria de la clase.

Los objetos que se desean hacer persistentes se definen en un documento de *"mapping"* (el cual es un archivo XML), y que permite describir los campos persistentes con sus respectivas asociaciones, así como también las subclases que posee el objeto en cuestión. Estos documentos son compilados en el momento de inicialización de la aplicación y proveen a la herramienta la información necesaria para cada clase. Adicionalmente, Hibernate es capaz de generar los esquemas de base de datos a partir del modelo de clases e incluso crear clases Java a partir del modelo de datos.

Ahora veremos un ejemplo simple para mostrar el uso de esta plataforma.

- Crear las clases persistentes Java

La siguiente clase3.24 representa la estructura Java de la tabla AppLabsUser. Generalmente los objetos de dominio sólo contienen métodos getters y setters, ademas del constructor vacío e implementan la interfaz Serializable.

```java
package org.applabs.quickstart;

import java.io.Serializable;

public class AppLabsUser implements Serializable
{
    private static final long serialVersionUID = -2445074444430828173L;

    public AppLabsUser()
    {
    }
    private Long id;
    private String userName;
    private String userPassword;
    private String userFirstName;
    private String userLastName;
    private String userEmail;
    private Date userCreationDate;
    private Date userModificationDate;

    public Long getId()
    {
        return id;
    }
    public void setId(Long id)
    {
        this.id = id;
    }
```

Figura 3.24: Clase AppLabsUser

- Relacionar la clase POJO con la base de datos en el documento de *mapping*

Cada clase persistente necesita ser relacionada en su propio archivo de configuración. El siguiente código representa la asociación de la clase AppLabsUser con la tabla applabsuser.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
    <class name="org.applabs.hibernate.quickstart.AppLabsUser"
        table="applabsuser">

        <id column="USER_ID" name="id" type="java.lang.Long">
            <generator class="sequence" />
        </id>

        <property column="USER_NAME" length="255" name="userName"
            not-null="true" type="java.lang.String" />
        <property column="USER_PASSWORD" length="255"
            name="userPassword" not-null="true" type="java.lang.String" />
        <property column="USER_FIRST_NAME" length="255"
            name="userFirstName" type="java.lang.String" />
        <property column="USER_LAST_NAME" length="255"
            name="userLastName" type="java.lang.String" />
        <property column="USER_EMAIL" length="255" name="userEmail"
            type="java.lang.String" />
        <property column="USER_CREATION_DATE" length="10"
            name="userCreationDate" type="java.util.Date" />
        <property column="USER_MODIFICATION_DATE" length="10"
            name="userModificationDate" type="java.util.Date" />

    </class>
</hibernate-mapping>
```

Figura 3.25: Asociacion Tabla-Clase (*Applabuser.hbm.xml*)

Los documentos de asociación son bastantes autoexplicativos. El elemento <*class*> relaciona la tabla con la correspondiente clase. El elemento <*id*> representa la columna que es la llave primaria de la tabla, y su asociación con el objeto dentro del modelo de dominio. El elemento <*property*> representa todos los otros atributos disponibles en el objeto de dominio en conjunto con sus respectivas columnas a nivel de base de datos.

- Archivo de configuración de Hibernate

Ahora es necesario generar el archivo maestro de configuración de Hibernate, el cual permite enlazar los distintos archivos de "*mapping*" de tablas con clases. Además se puede especificar la fuente de datos o detalles de la conexión JDBC requerida para que Hibernate pueda obtener conexiones a la base de datos. El elemento <*mapping-resource*> hace referencia al documento de relación tabla-clase.

```xml
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="show_sql">true</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLMyISAMDialect</property>
        <property name="hibernate.connection.driver_class">org.gjt.mm.mysql.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/applabs</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">r00Tp@$wd</property>

        <mapping resource="org/applabs/hibernate/quickstart/Applabsuser.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Figura 3.26: Configuración de Hibernate(*hibernate.cfg.xml*)

- Ejemplo de uso, cómo insertar un registro

Este es el ejemplo de cómo realmente se usa Hibernate dentro de un programa. Típicamente el programa comienza con la inicialización de Hibernate, es decir la lectura del archivo de configuración. Esto se puede hacer de manera programática o bien a través del archivo que se muestra en la figura3.26.

En el modo de configuración por archivo, Hibernate busca el archivo de configuración *hibernate.cfg.xml* dentro del *classpath*. Basado en lo encontrado en el archivo, Hibernate crea las relaciones entre tablas y objetos de dominio. Luego es necesario obtener una sesión de Hibernate, comenzar una transacción y finalmente hacer un *commit*.

```java
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();
AppLabsUser user = new AppLabsUser();
Transaction tx = session.beginTransaction();
user.setUserCreationDate(new Date());
user.setUserEmail("user@allapplabs.com");
user.setUserFirstName("userFirstName");
user.setUserLastName("userLastName");
user.setUserName("userName-1");
user.setUserPassword("userPassword");
session.saveOrUpdate(user);
tx.commit();
session.close();
```

Figura 3.27: Código ejemplo de uso de Hibernate

De este modo queda un nuevo registro insertado en la base de datos

Capítulo 4

Análisis, Arquitectura y Diseño

4.1. Requerimientos

Dado que estas capas de abstracción fueron desarrolladas dentro del marco de un proyecto, el proceso de toma de requerimientos no existió como tal, sino más bien, hubo solicitudes informales con respecto a cuáles aspectos dentro del desarrollo estaban siendo muy repetitivos y de que manera se podían mejorar. Es por ello que en este caso, el listado de requerimientos que se presenta no tiene la formalidad esperada como típicamente se tiene en un proyecto de desarrollo de software.

4.1.1. EJBProxy

El objetivo fundamental que se tuvo al momento de contruir esta capa de abstracción, fue el encapsulamiento de la tecnología de exposición remota de servicios, en este caso EJB. Por esta razón los requerimientos de esta capa surgen de manera natural por la necesidad de encapsular la tecnología EJB al desarrollador.

El listado de los requerimientos de esta capa son los siguientes:

- Encapsular la complejidad de la tecnología EJB al desarrollador. Eso significa que el desarrollador no tiene que conocer el mecanismo de exposición ni recuperación de los servicios.
- Los servicios deben presentarse como clases POJO. Esto quiere decir que todo servicio que se quiera exponer de manera remota, necesita solamente definir su interfaz y su respectiva implementación.
- Minimizar el uso de archivos de configuración.
- La migración entre distintos contenedores J2EE debiese ser sólo configuración.
- Debe funcionar con Java 1.4.

4.1.2. ORM

El objetivo principal de esta capa, es otorgar un mecanismo que diera mayor flexibilidad al desarrollador frente a cambios en el modelo de datos y automatizar tareas de validación en conjunto con la generación correcta de consultas en lenguaje SQL.

El listado de requerimientos de esta capa son los siguientes:

- Debe ser de fácil configuración. Esto significa que integrarlo a lo que está construido no sea algo complicado.
- Debe permitir poder coexistir con JDBC directo. Esto permite hacer una migración gradual de un mecanismo de persistencia a otro con bajos impactos.
- Si bien debe permitir poder hacer consultas con SQL estándar, también debe proveer un lenguaje de consulta a través de componentes propios de esta capa.
- Dado el requerimiento anterior, se debe obtener información del modelo de datos (metadatos) de manera automatizada de manera de poder crear consultas SQL en tiempo de ejecución.
- La transaccionalidad debe ser manejada por el contenedor.
- Esta capa debe permitir independencia del motor de base de datos.
- Debe hacer validaciones del modelo, como existencia de llave primaria para las tablas, si existe o no una tabla, validaciones de largo, tipo o si permite valores nulos para las columnas.
- Debe funcionar con Java 1.4.

4.2. Arquitectura y Diseño

4.2.1. EJBProxy

Para poder explicar el diseño de esta capa, vamos a tomar como ejemplo una interfaz de servicio que llamaremos D1Srv con un método “echo”. Esta interfaz tendrá 2 implementaciones, una implementación real la cual hará la lógica de negocios exigida para este servicio, y otra implementación de prueba que permitirá integrarse de inmediato con otros sistemas o capas de visualización.

La implementación real del servicio corresponde a la clase D1RealMgr, y la implementación de prueba corresponde a la clase D1MockMgr. La figura 4.1 muestra el servicio D1Srv junto con sus implementaciones.

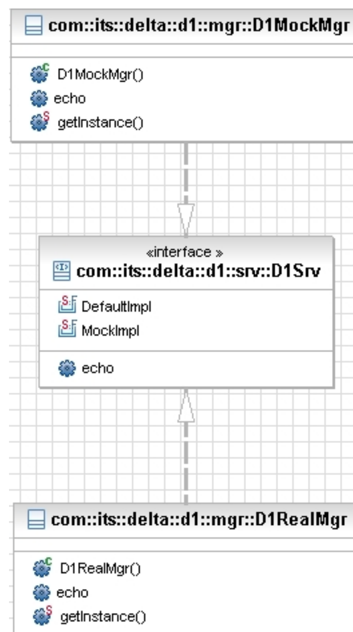

Figura 4.1: Interfaz de Servicio Echo y sus implementaciones

Esta capa la podemos dividir en 2 partes con distintas responsabilidades. La primera parte es la encargada de transformar la llamada al servicio ”echo” en una invocación EJB. La segunda parte es la encargada de que a través de la invocación EJB se ejectue el método “echo” en la implementación concreta del servicio.

- Transformación de llamada de servicio a una invocación EJB

Para poder invocar al servicio ”echo”, es necesario obtener una implementación de la interfaz D1Srv. La clase encargada de esto, es una clase que llamaremos **FactoryRemoteD1Service**

que implementa el patrón FactoryMethod. A través del método "**getD1Srv**" obtendremos la implementación correspondiente al servicio D1Srv.

Ahora bien, esto no es suficiente ya que en ningún momento hemos recuperado un EJB para efectuar la llamada a través de un EJB. Para hacer posible lo anterior, necesitamos "interceptar" la invocación al método "echo" cada vez que se invoque y transformar esa llamada en una invocación a un EJB.

La infraestuctura de Java nos provee esta propiedad a través de proxies dinámicos. En este caso, se implementó un proxy dinámico en una clase llamada **DelegateSrv**. Esta clase es la responsable de recuperar nuestro EJB genérico (**CommandFacade**) y hacer la llamada del método "echo" a través del EJB. De esta manera, cualquier invocación al metodo "echo", se podrá transformar en una invocación a un EJB.

Lo que necesitamos finalmente es poder tener toda la información del servicio, método a ejecutar y sus argumentos en una sola clase de manera de poder enviar esa información a través del EJB para que posteriormente se ejecute la clase encargada de la lógica concreta del servicio. La manera de obtener esa información es a través de la API de instrospección que posee Java, la cual nos provee de la información necesaria para la invocación. La clase que contiene toda esa información es la clase **ParametersCommandTO.**

Las figuras 4.2 y 4.3 muestran el diagrama de clases de las clases involucradas en el proceso de transformación de la llamada al método echo de D1Srv en una invocación EJB.

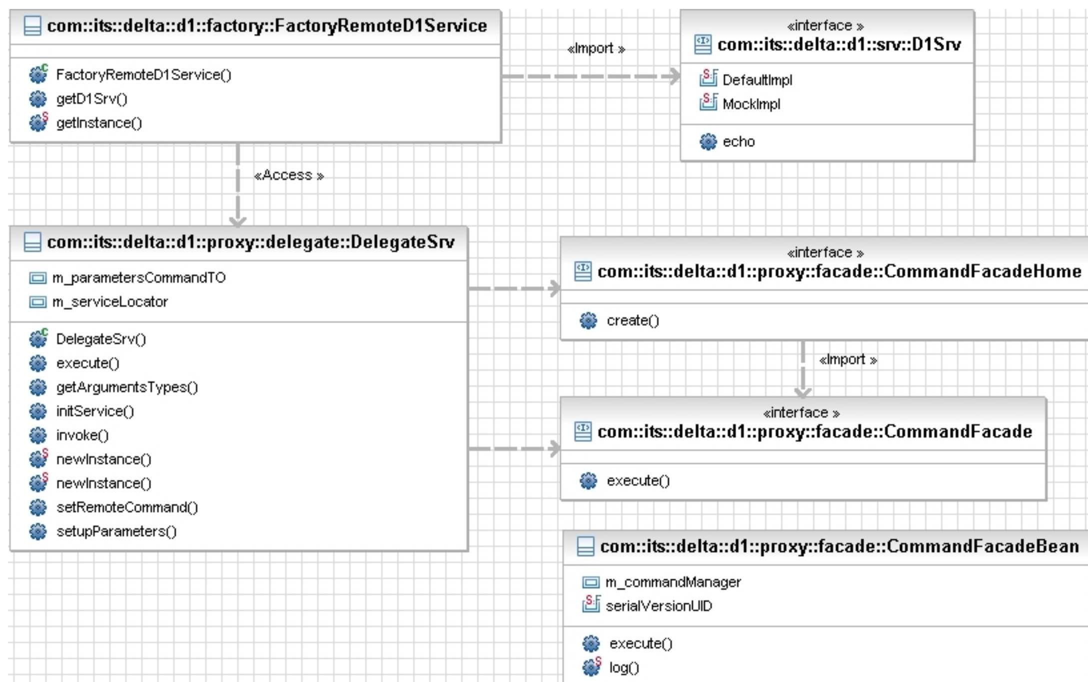

Figura 4.2: EJB Proxy

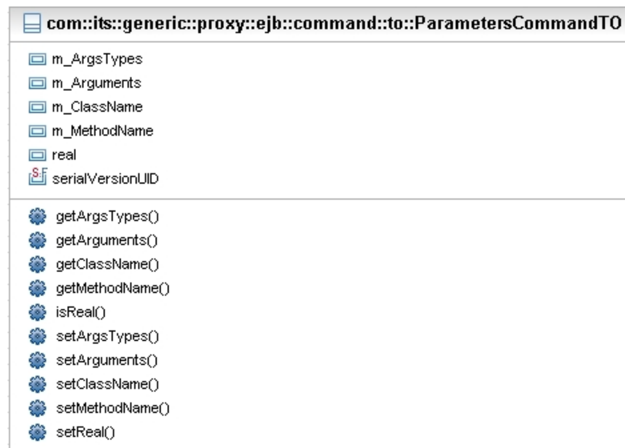

Figura 4.3: Clase ParametersCommandTO

- Ejecución del servicio real.

Como vimos anteriormente, el EJB CommandFacade invoca a un método genérico llamado execute on una instancia de ParametersCommandTO como parámetro. Este método de alguna manera debe ser capaz de resolver qué clase es la que tiene invocar para ejecutar la lógica concreta del servicio (en este caso D1RealMgr o bien D1MockMgr).

Para poder efectuar lo anterior, este método delega la responsabilidad de la ejecución en 2 interfaces. La interfaz CommandMgr y la interfaz Command. Estas interfaces tienen la siguiente responsabilidad.

CommandMgr : Interfaz encargada de obtener la clase que tiene la implementación concreta de D1Srv (ya sea la implementación de prueba o la implementación de la lógica de negocios real)

Command: Interfaz encargada de ejecutar el método correspondiente a la invocación inicial, dada la implementación concreta del servicio.

Las figuras 4.4 y 4.5 muestran el diagrama de clases de la interfaz CommandMgr y Command.

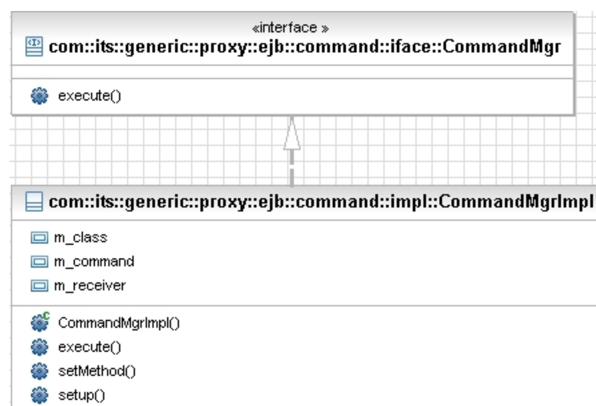

Figura 4.4: Interfaz CommandMgr y su Implementación

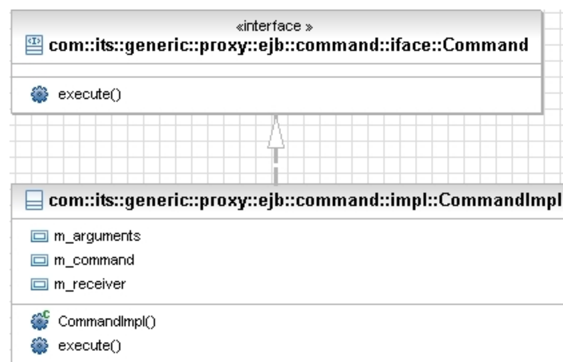

Figura 4.5: Interfaz Command y su implementación

En resumen, la secuencia de ejecuciones que ocurren al momento de invocar al método "echo" en D1Srv son los siguientes:

1. Se invoca a D1Srv.echo(argumentos)
2. DelegateSrv intercepta esta invocación
3. DelegateSrv obtiene el EJB CommandFacade
4. DelegateSrv crea una instancia de **ParametersCommandTO** con los parámetros de invocación de D1Srv.echo
5. DelegateSrv invoca a CommandFacade.execute(ParametersCommandTO)
6. CommandFacade.execute invoca CommandMgr.execute
7. CommandMgr determina cuál es la implementación concreta (D1RealMgr o D1MockMgr) que debe utilizarse para la invocación del método
8. CommandMgr.execute invoca a Command.execute
9. Command.execute invoca a la implementación concreta, por ejemplo D1RealMgr.echo(argumentos)
10. Se retorna el resultado de la invocación de D1RealMgr.echo(argumentos).

4.2.2. ORM

Esta capa se puede separar en 5 grandes componentes. Cada uno de estos componentes tiene responsabilidades acotadas de manera de disminuir el acoplamiento interno de este sistema.

Componentes de base de datos

Estos componentes son los objetos análogos a los objetos que se tienen en base de datos. Dentro de este grupo de componentes se cuentan los siguientes:

1. Clase Field: Esta clase es el objeto correspondiente a una columna de una tabla en base de datos. Por esa razón tiene propiedades tales como nombre, largo, tipo, si acepta valores nulos, si tiene valor autogenerado y en caso de ser una llave foránea, el nombre de tabla y columna a la cual hace referencia.

2. Clase PrimaryKey: Es el equivalente a la llave primaria en base de datos. Por lo tanto ésta se compone de 1 o más campos (Field).

3. Clase ForeignKey: Esta clase posee la lista de todos los campos que son llave foránea de una tabla.

4. Clase Table: Es el análogo a una tabla en base de datos. Posee una lista de columnas (Field), llave primaria y llaves foráneas. Además tiene otros datos que son extras como por ejemplo un alias, para efectos de la generación de consultas.

La figura 4.6 muestra el diagrama de clases de los objetos mencionados.

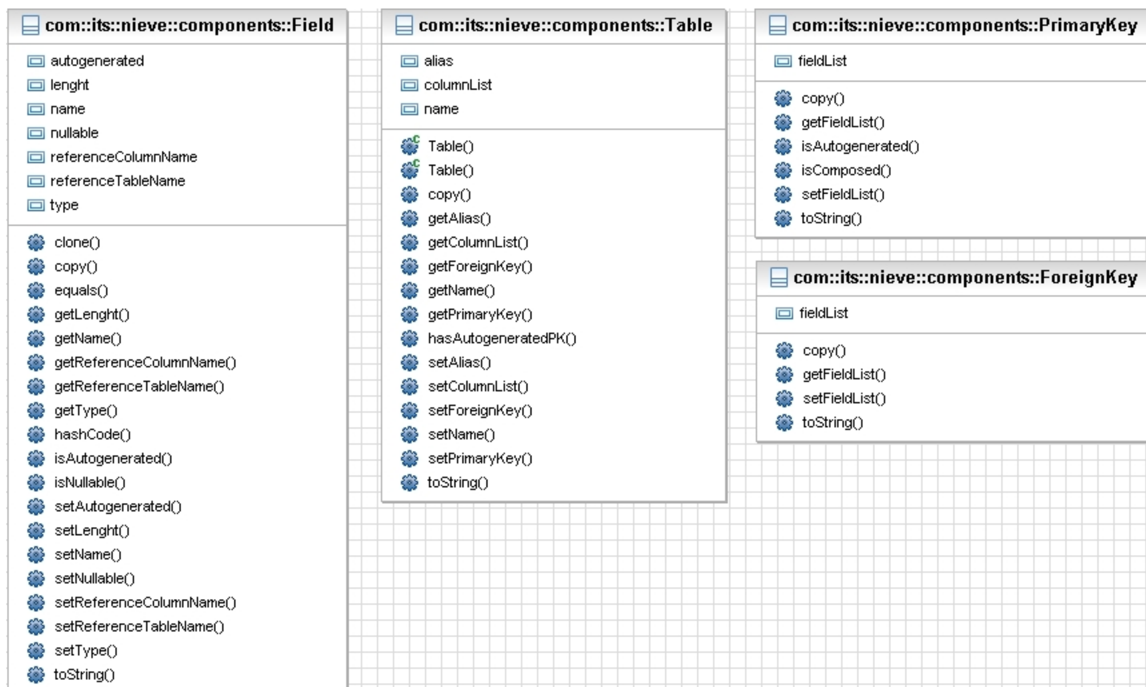

Figura 4.6: Componentes de Base de Datos

Componentes de consulta

Estos componentes son los objetos necesarios para efectuar consultas a la base de datos. En particular los componentes más importantes son:

- Clase Query: Esta clase posee la información de la consulta que se quiere generar, y a su vez con qué valores, conjunciones y comparadores se debe crear la consulta.
- Clase QueryParameters: Esta clase tiene la responsabilidad de tener los valores de los parámetros junto con el tipo de comparador (si es nulo , $<$, $>$, etc) que se quiere usar para ese campo.

La figura 4.7 muestra el diagrama de clases de Query y QueryParameters.

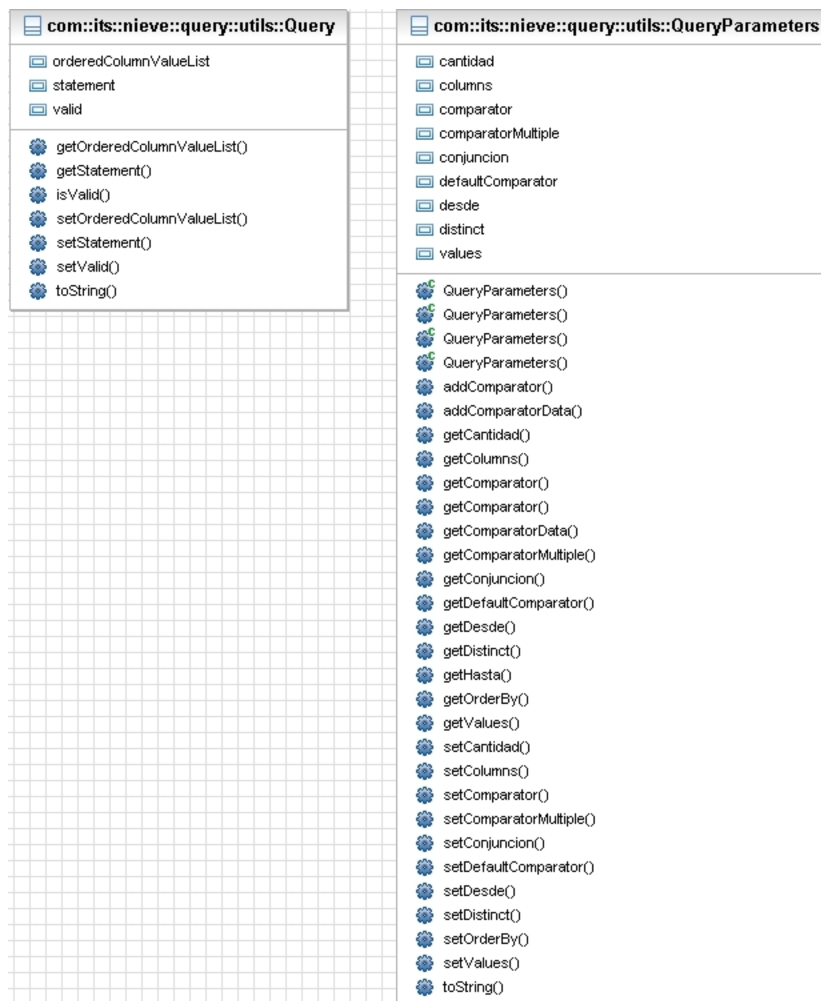

Figura 4.7: Componentes de Consulta

Obtención de Metadatos

Para la obtención de metadatos se definió una interfaz que permite guardar los metadatos en un archivo XML el cual tiene la representación del modelo de datos. De este modo se puede saber las relaciones que existen en la base de datos a partir de este archivo. Por lo mismo, para poder generar las consultas SQL era necesario también poder leer este archivo XML.

De este modo la interfaz que se definió para la lectura de metadatos tiene solamente 2 métodos, un método para guardar los metadatos en un archivo XML, y otro método para obtener los metadatos a partir de este archivo XML.

Ahora bien, la obtención de los metadatos para algunos casos particulares depende del motor de base de datos. Es por ello que se creó una clase abstracta que hace la lectura de los metadatos de manera genérica, y distintas clases que heredan de ella para cada motor de base de datos en particular.

Las clases involucradas en la obtención de metadatos son las siguientes:

1. Interfaz MetadataReader: Es la interfaz que expone los servicios de lectura de metadatos a partir de la base de datos y su persistencia en un archivo XML, así como también la lectura de este archivo para poder usarlo en la generación de consultas.

2. Clase MetadataReaderStd: Esta clase es la clase abstracta que tiene toda la implementación de la lectura de los metadatos de la base de datos y su transformación en componentes de base de datos de este ORM (Table, Field, PrimaryKey y ForeignKey).

3. Clases MetadataReader<**X**>Impl: Estas clases son las implementaciones por cada motor de base de datos que se quiere obtener los metadatos. Estas clases extienden de la clase abstracta MetadataReaderStd e implementan la interfaz MetadataReader.

La figura 4.8 presenta el diagrama de clases de los objetos involucrados en la obtención de los metadatos.

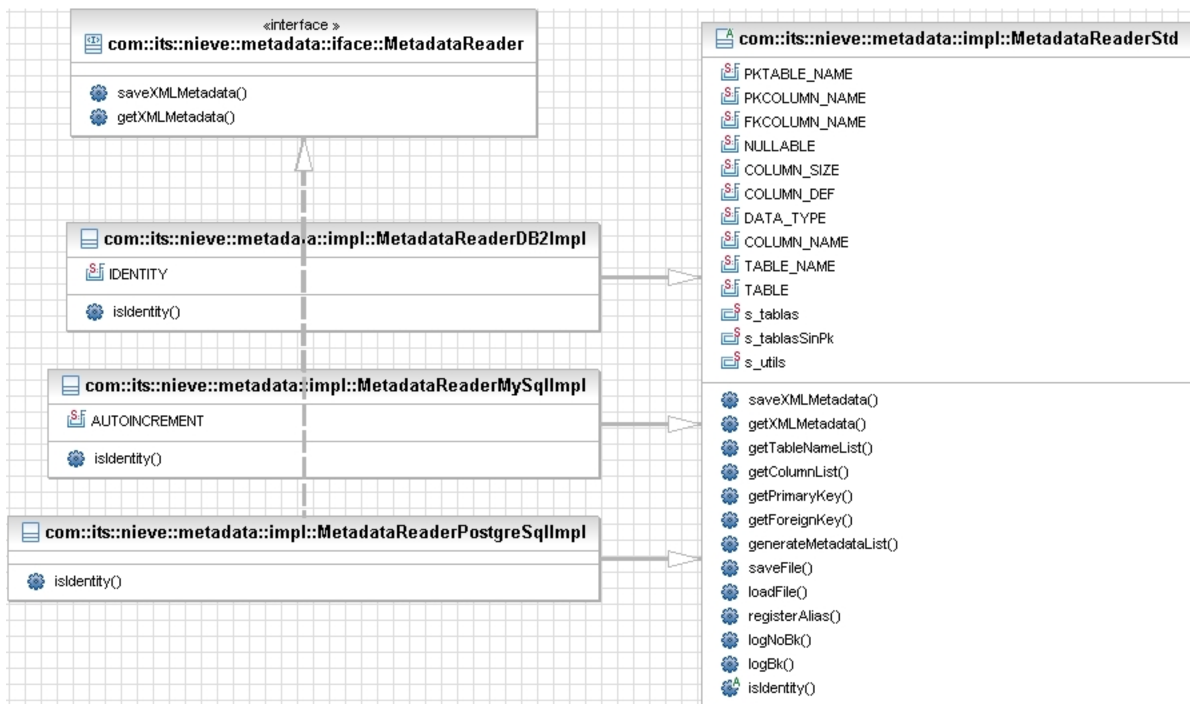

Figura 4.8: Interfaz MetadataReader

La instancia de la clase que se usa depende del motor de base de datos a utilizar, lo cual se indica en un archivo de configuración del ORM. La figura 4.9 muestra una sección de ejemplo del archivo XML de resultado de la obtención de los metadatos a partir de la base de datos.

```
<List>
  <Table>
    <name>DB2ADMIN.ADJ_ADJUNTO</name>
    <columnList>
      <Field>
        <name>ADJ_ADJUNTOID</name>
        <lenght>19</lenght>
        <type>-5</type>
        <nullable>false</nullable>
        <autogenerated>true</autogenerated>
      </Field>
      <Field>
        <name>COD_VIGENCIAADJUNTO</name>
        <lenght>19</lenght>
        <type>-5</type>
        <nullable>true</nullable>
        <autogenerated>false</autogenerated>
      </Field>
      <Field>
        <name>COD_TIPOADJUNTO</name>
        <lenght>19</lenght>
        <type>-5</type>
        <nullable>false</nullable>
        <autogenerated>false</autogenerated>
      </Field>
      <Field>
        <name>COD_ESTADJUNTO</name>
        <lenght>19</lenght>
        <type>-5</type>
        <nullable>false</nullable>
        <autogenerated>false</autogenerated>
      </Field>
      <Field>
        <name>ADJ_REFERENCIA</name>
        <lenght>30</lenght>
        <type>12</type>
        <nullable>true</nullable>
        <autogenerated>false</autogenerated>
```


Figura 4.9: XML de metadatos

La información que se tiene en este archivo corresponde al listado de tablas del sistema (Clase Table), por cada tabla se tiene su lista de columnas, cada columna sus datos (Clase Field), los datos de llave primaria (Clase PrimaryKey) y los datos de llave foránea (Clase ForeignKey). En la sección de implementación se verá en mayor detalle cómo se genera este archivo y su propósito dentro de la generación de consultas.

Generador de consultas

Para la generación de consultas, se definió una interfaz que provee de métodos genéricos para la creación de consultas. Esta interfaz tiene a su vez una implementación estándar y distintas implementaciones particulares para cada motor de base de datos que se desee implementar.

Lo anterior es necesario ya que no todos los motores de base de datos implementan completamente el estándar SQL, y por lo tanto para cierta clase de consultas, cada motor provee de extensiones del lenguaje SQL para poder efectuarlas. Ese tipo de consultas son las que se generan en las clases particulares de cada motor.

Las clases involucradas en la generación de consultas son las siguientes:

1. Interfaz QueryGenerator2: Esta es la interfaz que provee los métodos de generación de consultas.
2. Clase QueryGenerator2StandarImpl: Esta clase abstracta implementa la generación de consultas de manera estándar y tiene métodos abstractos para consultas que son particulares a cada motor de base de datos.
3. Clases QueryGenerator2<**X**>Impl: Estas clases corresponden a las distintas implementaciones por cada motor de base de datos que se desee implementar.
4. Clase QueryGeneratorFactory: Esta clase está encargada de recuperar la implementación correcta de acuerdo al motor de base de datos seleccionado en la configuración de la interfaz QueryGenerator2.

La figura 4.10 muestra el diagrama de clases de los objetos involucrados en la generación de consultas.

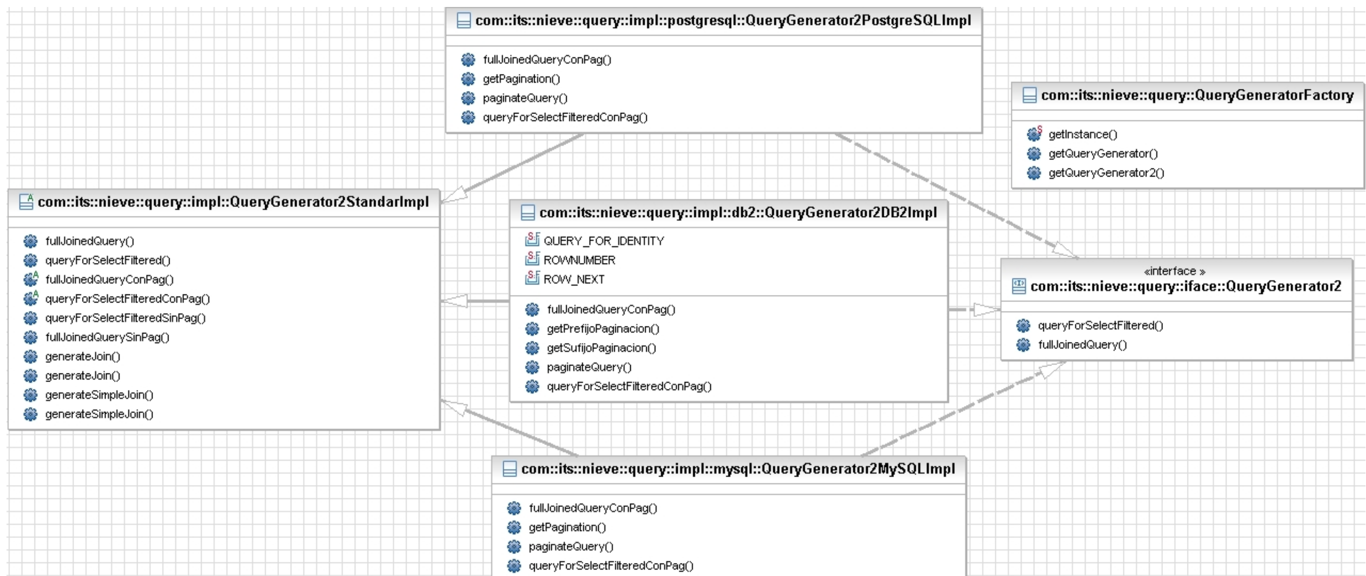

Figura 4.10: Interfaz QueryGenerator

Ejecutor de consultas

La ejecución de las consultas creadas a través del ORM (objeto Query), se puede separar en 3 fases. La primera, es la invocación a un objeto de acceso a base de datos (objeto DAO) usando el objeto Query como parámetro. Este objeto de acceso a base de datos se encarga de obtener una conexión a base de datos, ejecutar la consulta, retornar los resultados y liberar los recursos pedidos tales como conexiones u otros. Finalmente el proceso termina una vez que los datos retornados por el objeto DAO, se transforman en objetos de negocios a través de un convertidor de objetos.

Las clases relevantes en términos de diseño que están involucradas en este grupo de componentes son las siguientes:

1. Interfaz GenericExecutorMgr: Es la interfaz que expone los servicios de ejecución. Esto se define como interfaz debido a que no necesariamente queremos ejecutar consultas SQL, sino que podría ser que se quiera generar consultas en otro lenguaje y por lo tanto necesitamos distintas implementaciones del ejecutor de consultas, para distintos lenguajes.

2. Clase GenericExecutorMgrImpl: Es la implementación del servicio anterior. Principalmente usa una implementación de la clase GenericDao para efectos de ejecución de las consultas SQL, y se encarga de transformar los resultados de base de datos en objetos de negocio.

3. Interfaz GenericDao: Interfaz que presenta los servicios de ejecución de consultas SQL.

4. Clase GenericDaoNormalImpl: Clase que implementa la interfaz GenericDao. Se encarga de obtener la conexión a base de datos y de la ejecución misma de las consultas SQL.

Las figuras 4.11 y 4.12 muestran el diagrama de clases de los objetos involucrados en la ejecución de consultas.

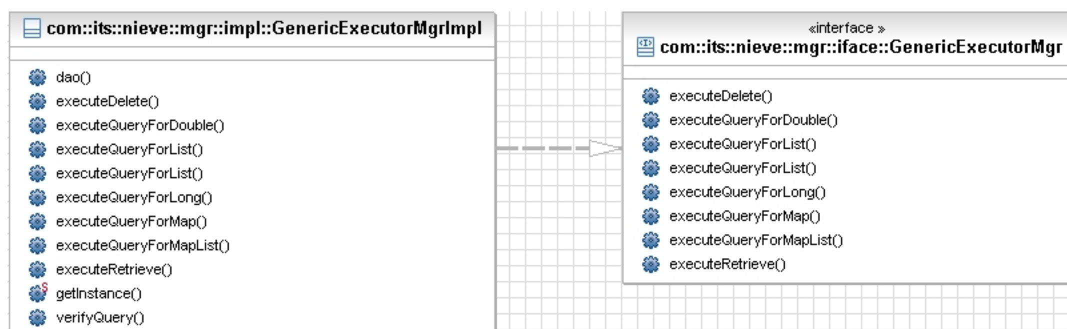

Figura 4.11: Interfaz GenericExecutorMgr

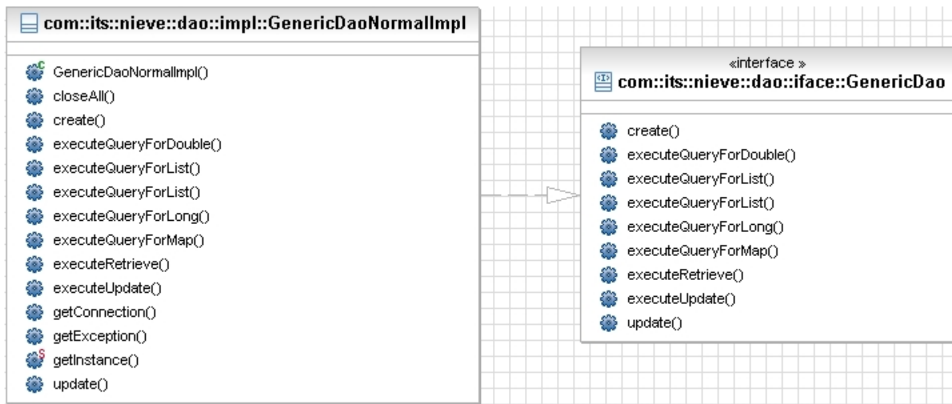

Figura 4.12: Interfaz GenericDao

Capítulo 5

Implementación

5.1. EJBProxy

5.1.1. Problemas enfrentados durante el desarrollo

Idealmente, una capa de abstracción debiese estar empaquetada en un puro archivo de manera de poder reutilizarla de manera fácil entre distintos proyectos o módulos de un mismo proyecto. Sin embargo, por la naturaleza de los EJB, esta capa no se pudo dejar empaquetada en 1 puro archivo debido a que en el minuto de hacer el despliegue de los EJB, se necesita tener disponible el bytecode de cada clase perteneciente a un EJB.

Por esta razón las clases correspondientes al EJB genérico y sus descriptores de EJB es necesario copiarlos en cada módulo, lo cual claramente no es lo ideal. Sin embargo las clases que implementan el patrón Command no tienen ese problema ya que trabajan de manera independiente de los EJB, por lo tanto esas clases sí están empaquetadas en sólo un archivo.

5.1.2. Obtención de servicios a través de EJBProxy

Partiremos con la continuación del código que se utilizó en el capítulo de arquitectura, el cual tenía un servicio D1Srv con su implementación. Para poder ejecutar un método del servicio D1Srv, es necesario obtener su implementación a través de un Factory, el cual en este caso corresponde a la clase FactoryRemoteD1Service. A continuación se muestra el código de como obtener el servicio y la ejecución del método "echo".

```
public void testEcho()
{
    D1TO to = new D1TO();
    to.setId( new Long( 1 ) );
    D1Srv service =  FactoryRemoteD1Service.getInstance().getD1Srv() ;
    service.echo( to );
}
```

La clase FactoryRemoteD1Service corresponde al Factory que recupera los servicios a través del EJBProxy usando la clase DelegateSrv, la cual es la encargada de obtener el EJB genérico que hace la invocación a la implementación real del servicio. El código de esta clase se muestra a continuación.

```
public class FactoryRemoteD1Service
{
  private static FactoryRemoteD1Service s_instance;
  private FactoryRemoteD1Service(){}
  public static FactoryRemoteD1Service getInstance()
  {
        if ( null == s_instance )
        {
            s_instance = new FactoryRemoteD1Service();
        }
        return s_instance;
  }
  public D1Srv getD1Srv()
  {
        try
        {
            return (D1Srv) DelegateSrv.newInstance( D1Srv.class ) ;
        }
        catch ( Exception ex )
        {
            throw new RuntimeException( ex.getMessage() , ex );
        }
  }
}
```

Podemos ver que de manera muy fácil, podemos recuperar un servicio a través del EJBProxy. En el siguiente punto veremos en detalle cómo se transforma la llamada del servicio a una llamada a través de un EJB, el cual termina con la invocación a la implementación real del servicio D1Srv.

5.1.3. Exposición de servicios a través de EJBProxy

Lo primero que es necesario entender, es el funcionamiento de la clase DelegateSrv. Esta clase tiene un conjunto de métodos y de varibles de instancia que mostramos a continuación.

```
public class DelegateSrv implements InvocationHandler
{
    private ServiceLocator m_serviceLocator;
    private CommandFacade m_commandFacade;
    private ParametersCommandTO m_parametersCommandTO;

    public static Object newInstance( Class aClass )...
    public static Object newInstance( Class aClass , boolean isReal )...
    protected DelegateSrv( Class aClass , boolean isReal )...
    protected void initService()...
    protected void setRemoteCommand()...
    protected Object execute( String aMethodName, Object [] arguments )...
    protected void setupParameters(String aMethodName, Object[] arguments)...
```

```
    protected Class [] getArgumentsTypes( Object [] anArguments )...
}
```

En primer lugar podemos ver que existen 3 variables de instancia las cuales son:

- ServiceLocator m_serviceLocator: Esta variable se utiliza para poder encontrar nuestro EJB genérico.
- CommandFacade m_commandFacade: Es la que tiene la referencia al EJB genérico.
- ParametersCommandTO m_parametersCommandTO: Es la que contiene los parámetros de invocación del método que se invocó (en este caso el método "echo").

Podemos notar también, que la clase DelegateSrv extiende de la clase InvocationHandler, esto nos permite que cada método de la interfaz (D1Srv) que se invoque a través de DelegateSrv, sea despachado a través de la ejecución del método "invoke" de la clase DelegateSrv. Esto significa que todas las llamadas a los métodos de D1Srv serán "interceptadas" a través del método "invoke" de la clase DelegateSrv.

Volviendo a nuestro ejemplo, cada vez que a través del FactoryRemoteD1Srv se quiere obtener una instancia del servicio D1Srv, se obtiene una instancia a través de DelegateSrv. El método que se invoca en este caso es el siguiente.

```
public static Object newInstance( Class aClass , boolean isReal )
    throws IllegalArgumentException , BusinessException
{
     return Proxy.newProxyInstance( aClass.getClassLoader(), new Class [] {
         aClass },new DelegateSrv( aClass , isReal ) );
}
```

Esto retorna una instancia de una clase Proxy para la interfaz especificada (aClass, en nuestro caso D1Srv) la cual despacha las invocaciones de los métodos de la interfaz a través del InvocationHandler especificado (en nuestro caso la clase DelegateSrv).

De este modo, debemos ver cuáles son las operaciones que ejecuta el constructor de nuestra clase DelegateSrv, las cuales son de inicialización de variables de instancia, como también de obtención de nuestro EJB genérico.

```
protected DelegateSrv( Class aClass , boolean isReal ) throws
    BusinessException
{
    m_parametersCommandTO = new ParametersCommandTO();
    m_parametersCommandTO.setClassName( aClass.getName() );
    m_parametersCommandTO.setReal( isReal );
    initService();
    setRemoteCommand();
}

//Inicialización de nuestro ServiceLocator
protected void initService() throws BusinessException
{
```

```
    try
    {
        m_serviceLocator = new RMILocator();
    }
    catch ( ServiceLocatorException se )
    {
        throw new BusinessException( se.getMessage(), se );
    }
}

//Obtención de nuestro EJB genérico ( CommandFacade ), a través del
    ServiceLocator
protected void setRemoteCommand() throws BusinessException
{
    try
    {
        CommandFacadeHome commandFacadeHome = (CommandFacadeHome)
            m_serviceLocator.getRemoteHome( CommandFacadeHome.class ,"
            CommandFacadeD1" );
        m_commandFacade = commandFacadeHome.create();
    }
    catch ( Exception re )
    {
        throw new BusinessException( re.getMessage(), re );
    }
}
```

Una vez que se tiene una instancia de DelegateSrv, se tiene que analizar lo que sucede cuando se invoca un método en nuestro servicio D1Srv. Por ejemplo cuando se invoca al método "echo", la clase DelegateSrv intercepta esta invocación y se pasa a ejecutar el método "invoke" de la clase DelegateSrv. De esta manera, como la llamada fue interceptada, podemos efectuar nuestra invocación a través del EJB genérico en el método "invoke". A continuación se muestra el código correspondiente al método "invoke" de la clase DelegateSrv junto con sus métodos auxiliares.

```
// Cada metodo que se ejecuta de la interfaz, se intercepta a través de este
    método
public Object invoke( Object aProxy, Method aMethod, Object [] args)throws
    Throwable
{
    return execute( aMethod.getName(), args );
}

// Este metodo se encarga de hacer la invocación a través del ejb generico (
    m_commandFacade).
protected Object execute( String aMethodName, Object [] arguments )throws
    BusinessException
{
    try
    {
        setupParameters(aMethodName, arguments);
        return m_commandFacade.execute( m_parametersCommandTO );
    }
    catch ( RemoteException re )
    {
```

```java
        throw new BusinessException( re.getMessage(), re );
    }
}

// Este método completa los parámetros restantes de la clase
   ParametersCommandTO
protected void setupParameters(String aMethodName, Object[] arguments)
{
    m_parametersCommandTO.setMethodName(aMethodName);
    m_parametersCommandTO.setArguments( arguments );
    m_parametersCommandTO.setArgsTypes( getArgumentsTypes( arguments ) );
}

//Este metodo obtiene los tipos de los argumentos de la invocación
protected Class [] getArgumentsTypes( Object [] anArguments )
{
    if( anArguments == null )
    {
        return null;
    }
    else
    {
        Class [] classes = new Class[ anArguments.length ];
        for ( int i = 0; i < anArguments.length; i++ )
        {
            classes [i] = anArguments[i].getClass();
        }
        return classes;
    }
}
```

Así podemos ver que cada invocación que se hace de un método de la interfaz D1Srv, termina en un invocación al EJB genérico en su método execute. El siguiente código corresponde al metodo “execute” de la clase ejb del EJB genérico.

```java
public Object execute( ParametersCommandTO commandTO ) throws
   BusinessException
{
    m_commandManager = new  CommandMgrImpl( commandTO.getClassName(),
       commandTO.isReal() );
    return m_commandManager.execute(
      commandTO.getMethodName() , commandTO.getArguments(),  commandTO.
         getArgsTypes() );
}
```

Como podemos ver, esta clase posee una instancia de CommandMgr, la cual es la encargada de obtener la implementación concreta del servicio D1Srv (en este caso la implementación es D1SrvRealImpl) y a su vez hacer la invocación del método correspondiente en la implementación concreta a través de la clase Command.

```java
public class CommandMgrImpl implements CommandMgr
{
    private Object m_receiver; // nueva instancia de m_class
    private Class m_class; // la clase del objeto
    private Method m_command; // método a ejecutar
```

```
    private Command m_executor; // quien hace la pega

    // Constructor, se encarga de obtener la implementacion concreta de la
        interfaz de Servicio
    public CommandMgrImpl( String aInterfaceName, boolean isReal )throws
        BusinessException
    {
        try
        {
             m_receiver = ReflectionUtils.getImplementationInstance(
                 aInterfaceName , isReal );
             m_class = m_receiver.getClass();
        }
        catch ( Exception ex )
        {
            throw new BusinessException( ex.getMessage(), ex );
        }
    }

     // Ejecutor
   public Object execute( String aMethodName,Object [] anArguments,Class []
      anArgsTypes )throws BusinessException
    {
        try
        {
            m_command =  m_class.getMethod( aMethodName, anArgsTypes );
            m_executor = new CommandImpl( m_receiver,m_command,anArguments );
        }
        catch ( Exception ex )
        {
            throw new BusinessException( ex.getMessage(), ex );
        }
        return m_executor.execute();
    }
}
```

Finalmente podemos ver que la clase Command hace la invocación a D1SrvRealImpl.

```
public class CommandImpl implements Command
{
    private Object m_receiver;
    private Method m_command;
    private Object [] m_arguments;

    //Constructor
    public CommandImpl( Object receiver,Method command,Object [] arguments )
    {
        m_receiver = receiver;
        m_command = command;
        m_arguments = arguments;
    }

    //Ejecución final
    public Object execute()throws BusinessException
    {
        try
        {
```

```
            return m_command.invoke( m_receiver, m_arguments );
        }
        catch ( Exception ex )
        {
            throw new BusinessException( ex.getMessage(), ex );
        }
    }
}
```

5.1.4. Alcances particulares de la implementación

Cabe destacar que esta capa de abstracción no tuvo dentro de sus alcances iniciales los siguientes requerimientos:

- Control de transaccionalidad por método de la interfaz de servicio. La transaccionalidad se maneja a nivel del método execute, por lo tanto todos los métodos que se invocan a través de esta capa pasan a ser transaccionales o no transaccionales dependiendo de la configuración que se indique a nivel del descriptor EJB del EJB genérico.
- Generación de implementaciones de prueba de servicios interfaces de prueba en tiempo de ejecución. Algo que puede ser muy deseado es poder tener implementaciones de prueba para los servicios, pero no tener que escribirlas (dada la inversión en tiempo). Esto es algo relativamente fácil de hacer y de agregar a la capa de abstracción, con la salvedad de que es necesario subir la versión de la máquina virtual a Java 5 para el uso de genéricos[1].
- Uso de anotaciones para validaciones en tiempo de compilación de existencia de implementación. En este minuto, la implementación concreta de la interfaz de servicio está indicada como una constante en la interfaz de servicio. Esto no permite hacer una validación de la existencia de la interfaz en tiempo de compilación. Idealmente, con el uso de anotaciones de Java 5, es posible hacer esta validación en tiempo de compilación[2].

[1] http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html
[2] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

5.2. ORM

5.2.1. Problemas enfrentados durante el desarrollo

Los principales problemas que se tuvieron al desarrollar esta capa de abstracción fueron los siguientes:

- Llenado de objetos de manera jerárquica. Este problema se tiene en particular por la falta de genéricos de Java 5 los cuales permiten distinguir la clase del objeto que va dentro de una lista de manera automática a través de la API de instrospección de Java. Además para las consultas paginadas, el poder llenar los objetos de manera jerárquica trae ciertos problemas con la cantidad de registros a traer por la consulta. Este problema trae como consecuencia que en algunos casos el llenado de los objetos de negocios sea algo tedioso.
- Existe un problema de generación de consultas en tablas donde se tienen múltiples referencias a otra tabla. En este caso particular, la generación de la consulta no queda correcta ya que el motor generador de consulta determina de manera automática el campo por el cual hacer el join entre las tablas. Por esta razón, no es posible generar de manera correcta la consulta SQL. La solución es relativamente simple y basta que para esos casos el desarrollador deba ingresar un parámetro más para la generación de la consulta.
- Falta de un identificador de campos con valor autogenerado en la API JDBC de Java. Esto no permite que de manera estándar se puedan obtener los campos que tienen valor autogenerado a través del mecanimo normal de JDBC. Para poder resolver este problema, en el momento de generar el archivo de metadatos de la base de datos, hay que hacer una consulta por cada campo si este campo tiene un valor autogenerado. Esta consulta es particular para cada motor de base de datos ya que no es estándar.

5.2.2. Generación de consultas SQL

Las clases encargadas de la generación de consultas son principalmente dos. La clase que se encarga de la generación de consultas estandar SQL (QueryGenerator2StandarImpl) y la clase particular de cada motor para consultas SQL que se escriben en lenguaje de cada motor. En el caso particular de esta implementación, las clases particulares de cada motor tienen que generar las consultas para efectuar paginación[3] a nivel de base de datos. El esqueleto de la clase generadora de consultas estándar se muestra a continuación.

[3] El término paginación a nivel de base de datos se refiere a cuando uno ejecuta una consulta en base de datos y no se quiere traer todos los registros de resultado de la consulta, sino más bien una "página" o subconjunto de los datos de la consulta. Esto es utilizado principalmente cuando la cantidad de registros es muy grande y por lo tanto existe una penalización grande a nivel de rendimiento por cada vez que se ejecuta una consulta sobre ese conjunto de datos.

```
public abstract class QueryGenerator2StandarImpl extends
    QueryGeneratorBaseImpl
{
public Query fullJoinedQuery( Table[] baseTable, Map baseFilter , List
    joinType,                    List tables, List queryParams ,
    QueryParameters queryParameters , Set showedTables) throws NieveException
    ....
public Query queryForSelectFiltered( Table tableName, Map map, QueryParameters
     queryParameters )throws NieveException....

// Metodos particulares de cada motor
protected abstract Query fullJoinedQueryConPag(Table[] baseTable, Map
    baseFilter , List joinType, List tables, List queryParams , QueryParameters
     queryParameters , Set showedTables ) throws NieveException ....

protected abstract Query queryForSelectFilteredConPag( Table tableName, Map
    map, QueryParameters queryParameters )throws NieveException....

// Metodos auxiliares
protected Query queryForSelectFilteredSinPag( Table table, Map map,
    QueryParameters queryParameters )throws NieveException....

protected Query fullJoinedQuerySinPag(Table[] baseTable, Map baseFilter , List
     joinType, List tables, List queryParams , QueryParameters queryParameters
    , Set showedTables) throws NieveException....

protected String generateJoin(List orderedColumnValueList, String joinType,
    Table[] baseTable, List tables, List queryParams) throws NieveException....

protected String generateJoin( List orderedColumnValueList ,List joinType ,
    Table[] tableBase ,List tables, List queryParams ) throws NieveException
    ....

protected String generateSimpleJoin(List orderedColumnValueList, String
    joinType, Table table, Map filterMap ,Table[] joinedTables, Map[]
    queryParameters)throws NieveException....

protected String generateSimpleJoin(List orderedColumnValueList, String[]
    joinType, Table table, Map filterMap ,Table[] joinedTables,Map[]
    queryParameters )throws NieveException....

}
```

Los métodos en los cuales se entrarán en detalle son los siguientes:

- public Query queryForSelectFiltered: Este método es responsable de generar las consultas sobre 1 tabla, dado los filtros indicados en los parámetros. Principalmente discrimina si la consulta va ser o no paginada e invoca al método respectivo para generar la consulta en cada caso.

```
public Query queryForSelectFiltered( Table tableName, Map map, QueryParameters
     queryParameters )throws NieveException
{
```

```
  if ( null != queryParameters && null != queryParameters.getDesde() && null
      != queryParameters.getCantidad() )
    {
        return queryForSelectFilteredConPag(tableName, map, queryParameters );
    }
    else
    {
        return queryForSelectFilteredSinPag( tableName, map, queryParameters )
            ;
    }
}
```

- public Query fullJoinedQuery: Este método es responsable de generar las consultas sobre varias tablas, dado los filtros indicados en los parámetros. Principalmente discrimina si la consulta va ser o no paginada e invoca al método respectivo para generar la consulta en cada caso.

```
public Query fullJoinedQuery( Table[] baseTable, Map baseFilter , List
    joinType,                          List tables, List queryParams ,
    QueryParameters queryParameters , Set showedTables) throws NieveException
{
  if ( null != queryParameters && null != queryParameters.getDesde() && null
      != queryParameters.getCantidad() )
    {
     return fullJoinedQueryConPag(baseTable, baseFilter, joinType, tables,
        queryParams, queryParameters , showedTables );
    }
    else
    {
   return fullJoinedQuerySinPag( baseTable, baseFilter, joinType, tables,
      queryParams, queryParameters , showedTables );
    }
}
```

- protected Query queryForSelectFilteredSinPag: Este es el método que genera las consultas sobre 1 tabla sin paginación. Es el que en definitiva arma la consulta SQL que tiene que estar dentro del objeto Query de retorno. Para ello utiliza varios métodos auxiliares los cuales se encargan principalmente de generar las distintas partes de la consulta, como por ejemplo las columnas que van en la consulta, los filtros, etc.

```
protected Query queryForSelectFilteredSinPag( Table table, Map map,
    QueryParameters queryParameters )throws NieveException
{
    if ( null != table )
    {
        Query ret = new Query();
        String columnas = getColumnsForSelect( table , false );
        String[] columnasArray = getColumnsForSelect( table.getName() ).split(
            COMMA_SEPARATOR );
        String filtros = "";
        List orderedColumnValueList = new ArrayList();
```

```
        for ( int i = 0; (i < columnasArray.length && null != map ); i++ )
        {
            QueryFilter queryFilter = (QueryFilter)map.get( columnasArray[i].
                trim() );
        filtros+=analyzerFilter(queryFilter,columnasArray[i], columnasArray[i
            ], table.getName(), orderedColumnValueList );
        }
        filtros = NieveStringUtils.getInstance().removeEndString( filtros ,
            QUANTITY_FOR_REMOVE_AND ) ;
        String query = "";
        if ( filtros.length() > 0 )
        {
         query = getSelectType( queryParameters ) + columnas + FROM + table.
             getName() + AS + table.getAlias() + WHERE + filtros;
            ret.setOrderedColumnValueList(orderedColumnValueList);
        }
        else
        {
        query = getSelectType( queryParameters ) + columnas + FROM +  table.
            getName() +AS + table.getAlias();
            ret.setOrderedColumnValueList(null);
        }
        query += NieveQueryUtils.getOrderBy( queryParameters );
        ret.setStatement( query );
        return ret;
    }
    else
    {
        throw new NieveException( "Tabla Nula" );
    }
}
```

- protected Query fullJoinedQuerySinPag: Este es el método que genera las consultas sobre varias tablas sin paginación. Es el que en definitiva arma la consulta SQL que tiene que estar dentro del objeto Query de retorno. Para ello utiliza varios métodos auxiliares los cuales se encargan principalmente de generar las distintas partes de la consulta, como por ejemplo las columnas que van en la consulta, los distintos tipo de join, los filtros, etc.

```
protected Query fullJoinedQuerySinPag(Table[] baseTable, Map baseFilter , List
     joinType, List tables, List queryParams , QueryParameters queryParameters
    , Set showedTables) throws NieveException
{
    if ( null != baseTable && baseTable.length > 0  )
    {
        Query ret = new Query();
        List orderedColumnValueList = new ArrayList();
        String columnas = getColumnsForSelectFull(baseTable[0], tables,
            showedTables);
        String join = generateJoin(orderedColumnValueList,joinType,baseTable ,
             tables, queryParams );
        String generateWhere = generateWhereForJoin( baseTable );
        String filtros = generateFilter( baseTable[0] , baseFilter ,
            orderedColumnValueList ,null );
```

```
        filtros = NieveStringUtils.getInstance().removeEndString( filtros ,
            QUANTITY_FOR_REMOVE_AND );
        String query = "";
        if ( filtros.length() > 0 && generateWhere.length() > 0 )
        {
            query = SELECT + columnas + FROM + join + WHERE + generateWhere +
                AND + filtros;
        }
        else if (filtros.length() > 0 && generateWhere.length() == 0)
        {
            query = SELECT + columnas + FROM + join + WHERE  + filtros;
        }
        else if (filtros.length() == 0 && generateWhere.length() > 0)
        {
            query = SELECT + columnas + FROM + join + WHERE + generateWhere ;
        }
        else
        {
            query = SELECT + columnas + FROM + join ;
        }
        ret.setOrderedColumnValueList(orderedColumnValueList);
        query += NieveQueryUtils.getOrderBy( queryParameters );
        ret.setStatement( query );
        return ret;
    }
    else
    {
        throw new NieveException("Tabla Nula");
    }
}
```

- protected abstract Query queryForSelectFilteredConPag: Este método retorna una consulta paginada sobre 1 tabla. Debe ser implementado por las clases que extienden de la clase de generación de consultas estándar, es decir para cada motor de base de datos. En este caso particular, se mostrará la implementación de este método para el motor de base de datos MySQL.

```
protected Query queryForSelectFilteredConPag(Table tableName, Map map,
    QueryParameters queryParameters) throws NieveException
{
    Query ret = queryForSelectFilteredSinPag( tableName, map, queryParameters
        );
    return paginateQuery(ret, queryParameters.getDesde() , queryParameters.
        getCantidad() );
}

public Query paginateQuery(Query query, Long desde, Long cantidad) throws
    NieveException
{
    Query ret = query;
    String statement = ret.getStatement();
    statement += getPagination(desde , cantidad );
    ret.setStatement( statement );
    return ret;
```

```
}

protected String getPagination(Long desde , Long cantidad )
{
    return " LIMIT " + desde + " , " + cantidad;
}
```

- protected abstract Query fullJoinedQueryConPag: Este método retorna una consulta paginada sobre varias tablas. Debe ser implementado por las clases que extienden de la clase de generación de consultas estándar, es decir para cada motor de base de datos. En este caso particular, se mostrará la implementación de este método para el motor de base de datos MySQL. El método paginateQuery es el mismo método que se utiliza para efectuar la paginación sobre consulta de 1 tabla.

```
protected Query fullJoinedQueryConPag(Table[] baseTable, Map baseFilter, List
    joinType, List tables, List queryParams, QueryParameters queryParameters,
    Set showedTables) throws NieveException
{
  Query ret = fullJoinedQuerySinPag( baseTable, baseFilter ,joinType , tables,
      queryParams ,queryParameters,showedTables);
    return paginateQuery(ret, queryParameters.getDesde(),queryParameters.
        getCantidad() );
}
```

5.2.3. Ejecución de consultas

En lo referido a la ejecución de consultas existen 2 clases encargadas de esta tarea. Ambas clases son las mencionadas en el capítulo de arquitectura, las cuales son GenericExecutorMgr y GenericDao. En particular las responsabilidades de cada una son las siguientes:

- GenericExecutorMgr: Recibe las consultas y las invoca a través de GenericDao junto con lo cual efectúa las transformaciones de datos desde base de datos a objetos de negocios a través de la interfaz de conversión. Esta clase posee una cantidad de métodos particulares para cada caso de ejecución, como por ejemplo para obtener listas de objetos, un solo objeto o bien un valor en particular. A continuación se detallan lo métodos que tiene esta clase.

```
public interface GenericExecutorMgr
{
public Long executeQueryForLong( Query query ) throws NieveException;
public Double executeQueryForDouble(Query query) throws NieveException;
public Long executeDelete( Query query ) throws NieveException;
public Map executeQueryForMap( Query query ) throws NieveException;
public Object executeRetrieve(Query query, String tableName , Class clazz )
   throws NieveException;
public List executeQueryForList( Query query , String tableName , Class clazz
     ) throws NieveException;
```

```
public List executeQueryForList( Query query, Table usuarioTable, Class clazz
    ) throws NieveException;
public List executeQueryForMapList( Query query ) throws NieveException;
}
```

En particular cada uno de estos métodos cumplen labores muy similares. En primer lugar efectúan una validación de la consulta, es decir verifican que los parámetros sean los correctos antes de ejecutarlas, invocan al método correspondiente en GenericDao y luego hacen la conversión a los objetos de negocios correspondientes al método. Por ese motivo, a continuación sólo vamos a ejemplificar con un método en particular ya que la implementación de los restantes es muy similar.

```
public List executeQueryForList( Query query, String tableName, Class clazz )
    throws NieveException
{
     verifyQuery( query );
     List aux = dao().executeQueryForList( query, tableName );
     List ret = new ArrayList();
     for ( Iterator iter = aux.iterator(); iter.hasNext(); )
     {
         Map map = (Map) iter.next();
         ret.add( FactoryConverter.getInstance().getNieveConverter().
             getToFromMap( map, clazz ) );
     }
     return ret;
}
```

- GenericDao: Se encarga de la efectuar la ejecución de las consultas. Para ello, debe recuperar conexiones a base datos y luego liberar estos recursos. Dada la relación que tiene la interfaz anterior y esta clase, los métodos que tiene son muy similares. El código de GenericDao se muestra a continuación

```
public interface GenericDao
{
public OID create( String tableName , Map map ) throws NieveException;
public Integer update( String tableName , Map map ) throws NieveException;
public Map executeRetrieve( String tableName , Query query ) throws
    NieveException;
public Map executeQueryForMap(Query query) throws NieveException;
public List executeQueryForList(Query query, String tableName ) throws
    NieveException;
public List executeQueryForList(Query query ) throws NieveException;
public Long executeQueryForLong( Query query )throws NieveException;
public Double executeQueryForDouble( Query query )throws NieveException;
public Long executeUpdate( Query query )throws NieveException;
}
```

Por el mismo motivo que para el caso de los métodos de GenericExecutorMgr, los métodos de GenericDao tienen todos características muy similares. En primer lugar se encargan de definir los recursos necesarios para la ejecución de consultas SQL, ésto es tener un objeto representativo de la conexión (objeto Connection), un ResultSet y PreparedStatement. Luego

se obtiene la conexión a base de datos, se obtienen los valores de los parámetros del filtro, se ingresan esos valores en el PreparedStatement, se ejecuta la consulta y se liberan los recursos. El código para un método en particular de GenericDao se detalla a continuación.

```
public List executeQueryForList(Query query) throws NieveException
{
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try
    {
        conn = getConnection();
        List orden = query.getOrderedColumnValueList();
        ps = conn.prepareStatement(query.getStatement());
        NieveSQLUtils.getInstance().setValuesIntoStatement(ps, orden);
        rs = ps.executeQuery();
        List ret = NieveSQLUtils.getInstance().toMapList(rs);
        return ret;
    }
    catch ( Exception ex )
    {
        throw getException(query, ex);
    }
    finally
    {
        closeAll( conn , ps , rs );
    }
}
```

5.2.4. Transformación de datos

El mecanismo de transformación de datos, se preocupa esencialmente de transformar los objetos de retorno de base de datos a objetos de negocios y viceversa (esto para el caso de los filtros para las consultas). Ahora bien, la necesidad de transformar los datos viene de los siguiente. Cuando uno ejecuta una consulta en base de datos, el resultado de ella corresponde a una tabla. La manera de relacionar esa tabla con objetos Java se hace mediante el objeto ResultSet de la API de Java.

Lo que hace este ORM, es recorrer el ResultSet y por cada registro crea una instance de un HashMap, dentro del cual pone los valores de cada columna de resultado dentro del HashMap. Por ejemplo, si hacemos una consulta sobre una tabla X y el resultado de esa consulta trae 2 registros, se generará una lista de largo 2, donde cada registro es un HashMap el cual cada uno de ellos corresponde a 1 registro del resultado de la consulta.

Viendo lo anterior gráficamente:

- **SQL**

SELECT A,B FROM X

A	**B**
1	2
2	3

- **JAVA**

```
List resultado = //Se obtiene de la consulta
HashMap registro1 = (HashMap) resultado.get(0);
HashMap registro2 = (HashMap) resultado.get(1);

Object registro1valorDeA = registro1.get("A"); //Debe ser 1
Object registro1valorDeB = registro1.get("B"); //Debe ser 2
Object registro2valorDeA = registro2.get("A"); //Debe ser 2
Object registro2valorDeB = registro2.get("B"); //Debe ser 3
```

El objetivo que tiene la transformación de datos, es tomar los respectivos HashMap de resultados y llevarlos a los objetos de negocios del sistema. Para ello, se utilizó una herramienta llamada Dozer[4] la cual permite transformar de objeto a objeto en Java, mediante un archivo de configuración XML. Por ejemplo, si queremos transformar el HashMap correspondiente a un Producto, el cual tiene como llaves NVPROD_ID, NVPROD_VERSION y NVPROD_DESC es necesario configurar lo siguiente :

```
<mapping map-id="ProductoTO">
    <class-a>com.its.nieve.example.mgr.to.ProductoTO</class-a>
    <class-b>java.util.HashMap</class-b>
    <field>
        <a>id</a>
        <b key="NVPROD_ID">this</b>
    </field>
    <field>
        <a>version</a>
        <b key="NVPROD_VERSION">this</b>
    </field>
    <field>
        <a>desc</a>
        <b key="NVPROD_DESC">this</b>
    </field>
</mapping>
```

Donde la clase ProductoTO es la siguiente:

```
public class ProductoTO implements Serializable
{
    private static final long serialVersionUID = -5754862381664462400L;
    private Long id;
    private Long version;
    private String desc;
    public ProductoTO(){}

    // Respectivos Getter y Setters de acuerdo a las convencioens JavaBean....
}
```

Recapitulando, y volviendo al código de ejecución, podemos verificar que una vez que se recuperan los datos a través de la invocación a GenericDao, se efectúa una transformación de los objetos recuperados a traves de la clase NieveConverter.

[4]http://sourceforge.net/projects/dozer

```
.....
    for ( Iterator iter = aux.iterator(); iter.hasNext(); )
    {
        Map map = (Map) iter.next();
        // Transformación de los objetos
        ret.add( FactoryConverter.getInstance().getNieveConverter().
            getToFromMap( map, clazz ) );
    }
.....
```

Finalmente, el código del método de la clase NieveConverter que hace la transformación, se encarga de invocar a la herramienta Dozer con los parámetros adecuados para la transformación de objeto a objeto.

```
public Object getToFromMap( Map map, Class clazz ) throws NieveException
{
    if ( null != map && null != clazz )
    {
        Object ret = s_mapper.map( map, clazz ); //Esta es la invocación a
            Dozer
        ret = NieveUtils.getUsableObject( ret ); //Esto verifica que si el
            objeto viene con todos sus valores nulos, entonces el objeto es
            nulo
        return ret ;
    }
    return null;
}
```

5.2.5. Camino

Una vez que estuvo en marcha el ORM, se empezaron a tener problemas debido a que la interfaz de generación de consultas recibe muchos parámetros, con lo cual se prestaba a mucha confusión y se hacía poco usable. Para evitar este problema, se creó un objeto que encapsulara el uso de la interfaz de generación de consultas el cual se llamó Camino. La idea de esto, es que uno va armando un "camino" a medida de que se arma la consulta, de manera que no es necesario pasar todos los parámetros de una vez al momento de armar la consulta, sino más bien se van ingresando en la medida que se va necesitando.

Camino también se encarga de ingresar los valores por omisión de cada uno de los parámetros de generación de consultas, en caso de que no se ingresen parámetros. Esto permite que el usuario tenga menos errores al momento de utilizar el ORM. Los métodos principales que tiene Camino son los siguientes:

```
public class Camino
{
public void join(Table table) throws NieveException;
public void leftJoin(Table table) throws NieveException;
public void rightJoin(Table table) throws NieveException;
public void join(Table table, boolean show) throws NieveException;
public void leftJoin(Table table, boolean show)throws NieveException;
public void rightJoin(Table table, boolean show)throws NieveException;
```

```
public void join(Table table, QueryParameters filter)throws NieveException;
public void leftJoin(Table table, QueryParameters filter)throws NieveException
    ;
public void rightJoin(Table table, QueryParameters filter)throws
    NieveException;
public void join(Table table, QueryParameters filter, boolean show)throws
    NieveException;
public void leftJoin(Table table, QueryParameters filter,boolean show) throws
    NieveException;
public void rightJoin(Table table, QueryParameters filter,boolean show) throws
     NieveException;
public Camino join(Camino camino) throws NieveException;
public Camino leftJoin(Camino camino) throws NieveException;
public Camino rightJoin(Camino camino) throws NieveException;
public Query getQuery() throws NieveException;
public void join(Table table, QueryParameters filter, String joinType, boolean
     show) throws NieveException;
public Camino join(Camino camino, String joinType)throws NieveException;
}
```

Principalmente con Camino uno define una tabla, y luego sobre esa tabla va haciendo los respectivos join con las tablas restantes de la consulta, en conjunto con los parámetros que se entregan de filtro por cada tabla. A continuación, un ejemplo simple de uso de Camino:

```
public List getCodigoTOListFullCaminoImpl(Long idTipoCodigo) throws
    NieveException
{
     Camino camino = new Camino( new Table(DatabaseConstant.TABLA_CODIGO, "DOS"
         ) );
     camino.join( new Table(DatabaseConstant.TABLA_TIPOCODIGO, "UNO"),new
         QueryParameters(new TipoCodigoTO(idTipoCodigo) ) );
     List lista = s_genericExecutor.executeQueryForMapList( camino.getQuery() )
         ;
     return getRealList(lista);
}
```

Esto genera una consulta como la siguiente:

```
SELECT DOS.COD_CODIGOID AS DOS_COD_CODIGOID , DOS.TIPCOD_TIPOCODIGO AS
    DOS_TIPCOD_TIPOCODIGO , DOS.COD_CODIGO AS DOS_COD_CODIGO , DOS.COD_ACTIVO
    AS DOS_COD_ACTIVO , DOS.COD_DESCRIPCION AS DOS_COD_DESCRIPCION , UNO.
    TIPCOD_TIPOCODIGO AS UNO_TIPCOD_TIPOCODIGO , UNO.TIPCOD_NOMBRE AS
    UNO_TIPCOD_NOMBRE , UNO.TIPCOD_DESCRIPCIONTIPO AS
    UNO_TIPCOD_DESCRIPCIONTIPO , UNO.TIPCOD_ESMODIFICABLE AS
    UNO_TIPCOD_ESMODIFICABLE

FROM DB2ADMIN.COD_CODIGO DOS INNER JOIN DB2ADMIN.TIPCOD_TIPOCODIGO UNO ON (
    DOS.TIPCOD_TIPOCODIGO = UNO.TIPCOD_TIPOCODIGO AND UNO.TIPCOD_TIPOCODIGO =
     ? )
```

La cual entrega todos los códigos del tipo “idTipoCodigo”.

5.2.6. Alcances de la implementación

Como esta capa de abstracción se desarrolló en paralelo con el proyecto en que se iba a utilizar, el énfasis particular de generación de consultas y pruebas se hicieron sobre el motor de base de datos del proyecto, el cual era DB2. Esto implicó que las funcionalidades y requerimientos pedidos fueran desarrollados siempre para que se ejecutaran correctamente en DB2, y las pruebas sobre otros motores eran un extra que se podía obviar en caso de estar escasos de tiempo. Es por ello que varias de las funcionalidades no han sido probadas todavía en otros motores, aunque a pesar de ello las pruebas sí están desarrolladas (son las mismas que se ejecutan en DB2) para los otros motores.

Capítulo 6

Aplicaciones Ejemplo

6.1. Arquitectura de aplicación J2EE bajo estándares Telefónica

Considerando que las capas de abstracción construidas fueron utilizadas para desarrollar el proyecto Delta para Telefónica, es necesario mostrar cuáles son las restricciones técnicas y de estándares que impone esta empresa. Las normas de desarrollo de aplicaciones J2EE bajo los estándares telefónica se muestran a continuación:

Arquitectura en capas

La arquitectura debe tener la siguiente descomposición en capas:

- Capa presentación en el browser:

1. HTML, Javascript básico, CSS.
2. No se usa Java applet, Macromedia Flash o componentes ActiveX.

- Capa presentación en el servidor:

1. Se usa JSP para desplegar HTML
2. Se usa la API Servlet para procesar los requerimientos HTTP y acceder a la lógica de negocio.
3. Se prohíbe ejecutar comandos SQL desde los servlet y los JSP,

4. En caso de usar EJB CMP 2.0 en la capa de persistencia, se autoriza el acceso solamente de lectura al modelo de Entity Beans. No se usa el patrón de "Value Object" (Objeto Valor).

- La capa lógica de negocio se implementa en Session Bean:

1. Son de tipo Stateless: no se usa Stateful Sessions Beans,
2. En caso de usar EJB CMP 2.0 en la capa persistencia, son los únicos responsables de actualizar el modelo de Entity Beans,
3. En caso de usar acceso directo vía JDBC en la capa persistencia, son los únicos responsables en ejecutar comandos SQL,
4. Son, por defecto, de acceso Local. Se usarán Session Bean de acceso Remoto, solamente cuando presten servicios a otras aplicaciones.

- Capa persistencia:

1. Se recomienda el uso de Entity Beans usando la norma EJB CMP 2.0. Son de acceso "Local".
2. En caso de no usar CMP 2.0, se debe acceder directamente a la base de datos vía JDBC.
3. No se autoriza ningún otro framework de persistencia como por ejemplo Hibernate, Torque o TopLink

- Capa de datos: base de datos relacional

1. Se prohíbe el uso de trigger para lógica de negocio,
2. El uso de store-procedure debe ser justificado (por ejemplo razones de rendimiento).

A continuación la figura 6.1 muestra un diagrama explicativo de los puntos anteriormente mencionados, correspondientes a las distintas capas de una aplicación J2EE estándar de Telefónica.

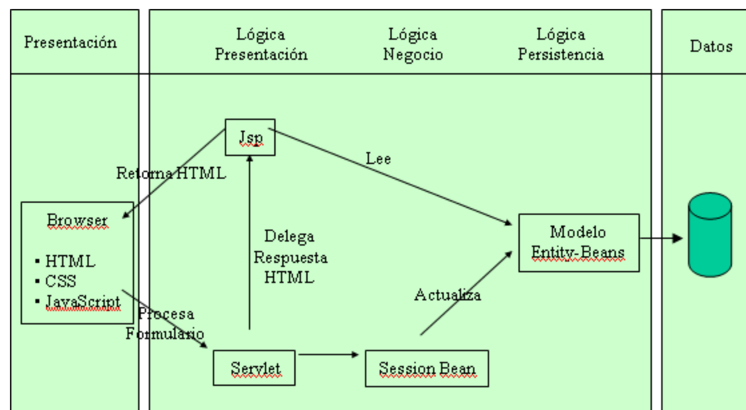

Figura 6.1: Arquitectura de aplicación bajo estándares Telefónica

Configuración

- Conexión a base de datos: se define el tag <resource-ref>en web.xml y se usa la API JNDI para obtener un Datasource. De esta forma, se usa la facilidad de pool de conexión del servidor de aplicación.
- Conexión a JMS: similar a conexión base de datos.
- Parámetros globales de una aplicación web o de un servlet se definen en el archivo web.xml con el tag <context-param>de la aplicación web o con el tag <init-param>del servlet.
- Autenticación y autorización: se prefiere el uso de los tags <login-config>y <security-constraint>en el archivo web.xml.
- Se debe minimizar el uso de variables de sesión de la API Servlet. Al usar esta facilidad, se debe asegurar la compatibilidad con la solución de cluster del servidor de aplicación.
- Log de error: evitar System.out y System.err.

Uso de bibliotecas o framework

- El uso de bibliotecas o framework no incluídos en el servidor de aplicación o en este documento, sean éstos comerciales u open source, debe ser previamente aprobado.
- Se ha autorizado el uso de las siguientes bibliotecas:

1. Log4j de Apache cuando la JVM (máquina virtual Java) es anterior a JDK 1.4. Se debe usar la API de log de Java para JVM superior o igual a JDK 1.4.
2. TN-MVC de Tecnautica

Todos los puntos anteriores corresponden al detalle de las restricciones técnicas que impone Telefónica para el desarrollo de aplicaciones Web J2EE. A continuación veremos como una aplicación que se rige bajo estos estándares se modifica para el uso de las capas de abstracción construidas.

6.2. Modificaciones necesarias para el uso de EJB-Proxy

Para este caso consideraremos un Servicio de operaciones básicas de códigos o tipologías dentro de la aplicación Delta. Recordando lo visto en el punto 5.2.1 de este documento, la secuencia de ejecuciones que siguen al invocar un servicio estándar bajo EJBProxy es la siguiente:

1. Se invoca a D1Srv.echo(argumentos)
2. DelegateSrv intercepta esta invocación
3. DelegateSrv obtiene el EJB CommandFacade
4. DelegateSrv crea una instancia de **ParametersCommandTO** con los parámetros de invocación de D1Srv.echo
5. DelegateSrv invoca a CommandFacade.execute(ParametersCommandTO)
6. CommandFacade.execute invoca CommandMgr.execute
7. CommandMgr determina cuál es la implementación concreta (D1RealMgr o D1MockMgr) que debe utilizarse para la invocación del método
8. CommandMgr.execute invoca a Command.execute
9. Command.execute invoca a la implementación concreta, por ejemplo D1RealMgr.echo(argumentos)
10. Se retorna el resultado de la invocación de D1RealMgr.echo(argumentos).

Para el caso en que no se usa EJBProxy la secuencia es distinta, ya que al no haber proxys involucrados el proceso es manual, es decir, el desarrollador debe escribir el código correspondiente a cada invocación. En el caso particular que veremos a continuación, para el servicio de mantención de códigos y tipologías la secuencia de invocaciones es la siguiente:

- Se recupera el servicio a través del Factory (línea 1), luego se crea una instancia de ServiceLocator para poder recuperar el EJB correspondiente al servicio de códigos (CodigoD3Svc, línea 5) y a través del ServiceLocator se obtiene el EJBHome (CodigoFacadeHome, línea 6) correspondiente al EJB del servicio de códigos. Finalmente con el EJBHome recuperado del JNDI, se crea el objeto EJB correspondiente al servicio (CodigoFacade, línea 7).

```
1 public CodigoD3Svc getCodigoD3SvcNoEjbProxy()
2 {
3     try
4     {
5         ServiceLocator serviceLocator = new RMILocator();
```

```
6           CodigoFacadeHome codigoFacadeHome = (CodigoFacadeHome)serviceLocator.
                getRemoteHome(CodigoFacadeHome.class ,"CodigoFacadeD3" );
7           CodigoFacade codigoFacade = codigoFacadeHome.create();
8           return new CodigoFacadeMgr( codigoFacade );
9       }
10      catch ( Exception ex )
11      {
12          throw new RuntimeException( ex.getMessage() , ex );
13      }
14  }
```

- El objeto que hace la invocación a la implementación final de la interfaz del servicio (CodigoFacadeBean) debe invocar por cada método de la interfaz a su correpondiente método a nivel de implementación.

```
public class CodigoFacadeBean extends BaseCommandFacadeBean
{
    private static final long serialVersionUID = 2114103412494505512L;
    private CodigoD3Svc m_codigoSvc = FactoryLocalD3Service.getInstance().
        getCodigoD3Svc();

    public Long addCodigo(CodigoTO codigo, Long tipoCodigo)throws
        BusinessException
    {
        return m_codigoSvc.addCodigo(codigo, tipoCodigo);
    }
    public void updateCodigo(CodigoTO codigo) throws BusinessException
    {
        m_codigoSvc.updateCodigo(codigo);
    }
    public void removeCodigo(Long idCodigo) throws BusinessException
    {
        m_codigoSvc.removeCodigo(idCodigo);
    }
    public CodigoTO getCodigo(Long idCodigo) throws BusinessException
    {
        return m_codigoSvc.getCodigo(idCodigo);
    }
    public CodigoTO[] listCodigos(Long tipoCodigo, Integer desde,Integer
        cantidad) throws BusinessException
    {
        return m_codigoSvc.listCodigos(tipoCodigo, desde, cantidad);
    }
    public Boolean relacionarCodigos(Long idCodigo1, Long idCodigo2,Long
        relacion) throws BusinessException
    {
        return m_codigoSvc.relacionarCodigos(idCodigo1, idCodigo2, relacion);
    }
    public void desRelacionarCodigos(Long idCodigo1, Long idCodigo2,Long
        relacion) throws BusinessException
    {
        m_codigoSvc.desRelacionarCodigos(idCodigo1, idCodigo2, relacion);
    }
    public List getCodigosRelacionados(Long idCodigo, Long relacion)throws
        BusinessException
```

```
    {
        return m_codigoSvc.getCodigosRelacionados(idCodigo, relacion );
    }
    public Long getCantidad(Long tipoCodigo) throws BusinessException
    {
        return m_codigoSvc.getCantidad(tipoCodigo);
    }
    public List getCodigosRelacionados(CodigoTO aCodigoDominio,CodigoTO
        aCodigoRecorrido, RelacionTipoCodigoTO aRelacionTipo)throws
        BusinessException
    {
        return m_codigoSvc.getCodigosRelacionados(aRelacionTipo);
    }
    public List getCodigosRelacionados(RelacionTipoCodigoTO aRelacionTipo)
        throws BusinessException
    {
        return m_codigoSvc.getCodigosRelacionados(aRelacionTipo);
    }
}
```

- Similar a lo anterior, se crea necesita una implementación del servicio la cual usa el objeto EJB creado para las invocaciones remotas. Esto es necesario para que el desarrollador no tenga conocimiento de la tecnología bajo la cual se implementó la invocación de servicios (ya sea EJB, Web Services, etc) y solamente use las interfaces de servicios definidas, sin tener que manejar las excepciones correspondientes a las invocaciones remotas que se tengan que hacer a nivel de EJB.

```
public class CodigoFacadeMgr implements Serializable, CodigoD3Svc
{
    private static final long serialVersionUID = -5496170205536762582L;
    private CodigoFacade facade;
    public CodigoFacadeMgr( CodigoFacade facade )
    {
        this.facade = facade;
    }
    public Long addCodigo(CodigoTO codigo, Long tipoCodigo) throws
        BusinessException
    {
        try
        {
            return facade.addCodigo(codigo, tipoCodigo);
        }
        catch (RemoteException e)
        {
            throw new BusinessException( e.getMessage() , e );
        }
    }
    public void desRelacionarCodigos(Long idCodigo1, Long idCodigo2, Long
        relacion) throws BusinessException
    {
        try
        {
            facade.desRelacionarCodigos(idCodigo1, idCodigo2, relacion);
```

```
        }
        catch (RemoteException e)
        {
            throw new BusinessException( e.getMessage() , e );
        }
    }
    public Long getCantidad(Long tipoCodigo) throws BusinessException
    {
        try
        {
            return facade.getCantidad(tipoCodigo);
        }
        catch (RemoteException e)
        {
            throw new BusinessException( e.getMessage() , e );
        }
    }
    public CodigoTO getCodigo(Long idCodigo) throws BusinessException
    {
        try
        {
            return facade.getCodigo(idCodigo);
        }
        catch (RemoteException e)
        {
            throw new BusinessException( e.getMessage() , e );
        }
    }
    public List getCodigosRelacionados(Long idCodigo, Long relacion) throws
        BusinessException
    {
        try
        {
            return facade.getCodigosRelacionados(idCodigo, relacion);
        }
        catch (RemoteException e)
        {
            throw new BusinessException( e.getMessage() , e );
        }
    }
    public List getCodigosRelacionados(CodigoTO aCodigoDominio, CodigoTO
        aCodigoRecorrido, RelacionTipoCodigoTO aRelacionTipo) throws
        BusinessException
    {
        try
        {
            return facade.getCodigosRelacionados(aCodigoDominio,
                aCodigoRecorrido , aRelacionTipo);
        }
        catch (RemoteException e)
        {
            throw new BusinessException( e.getMessage() , e );
        }
    }
    public List getCodigosRelacionados(RelacionTipoCodigoTO aRelacionTipo)
        throws BusinessException
```

```
{
    try
    {
        return facade.getCodigosRelacionados(aRelacionTipo);
    }
    catch (RemoteException e)
    {
        throw new BusinessException( e.getMessage() , e );
    }
}
public CodigoTO[] listCodigos(Long tipoCodigo, Integer desde, Integer
    cantidad) throws BusinessException
{
    try
    {
        return facade.listCodigos(tipoCodigo, desde, cantidad);
    }
    catch (RemoteException e)
    {
        throw new BusinessException( e.getMessage() , e );
    }
}
public Boolean relacionarCodigos(Long idCodigo1, Long idCodigo2, Long
    relacion) throws      BusinessException
{
    try
    {
        return facade.relacionarCodigos(idCodigo1, idCodigo2, relacion);
    }
    catch (RemoteException e)
    {
        throw new BusinessException( e.getMessage() , e );
    }
}
public void removeCodigo(Long idCodigo) throws BusinessException
{
    try
    {
        facade.removeCodigo(idCodigo);
    }
    catch (RemoteException e)
    {
        throw new BusinessException( e.getMessage() , e );
    }
}
public void updateCodigo(CodigoTO codigo) throws BusinessException
{
    try
    {
        facade.updateCodigo(codigo);
    }
    catch (RemoteException e)
    {
        throw new BusinessException( e.getMessage() , e );
    }
}
```

```
}
```

- Finalmente, la implementación anterior es la que devuelve el Factory.

Como bien se puede apreciar, existe mucho código repetitivo como los de las clases CodigoFacadeMgr y CodigoFacadeBean, los cuales eran los primeros candidatos a ser generados en tiempo de ejecución.

En el caso de CodigoFacadeBean, lo único que hace cada método es invocar a la implementación del servicio propiamente tal y nada más. En cambio CodigoFacadeMgr, hace practicamente lo mismo que CodigoFacadeBean con la salvedad de que la invocación no la hace a la implementación misma del servicio, sino que efectúa la invocación a través del EJB y por lo tanto tiene que manejar la excepción remota que envía el EJB y encapsularla en una excepción de servicios. La figura 6.2 muestra diagrama de clases representativo de las clases involucradas bajo esta arquitectura.

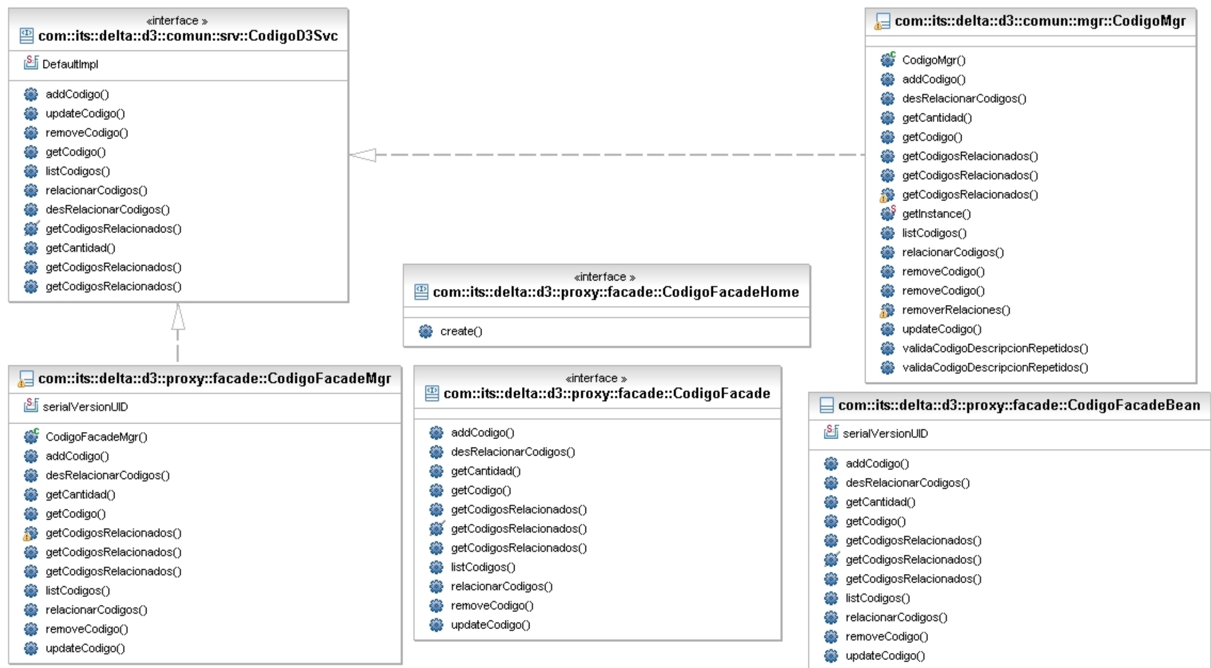

Figura 6.2: Clases involucradas en la presentación de un servicio como EJB

Cabe destacar que junto con las clases vistas anteriormente, es necesario declarar en el descriptor EJB las clases correspondientes al servicio que se está exponiendo como EJB. En este caso el código necesario es el siguiente.

```
<session id="CodigoFacade">
    <ejb-name>CodigoFacade</ejb-name>
    <home>com.its.delta.d3.proxy.facade.CodigoFacadeHome</home>
    <remote>com.its.delta.d3.proxy.facade.CodigoFacade</remote>
```

```
    <ejb-class>com.its.delta.d3.proxy.facade.CodigoFacadeBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
</session>
```

Este código es necesario incluirlo por cada interfaz que se quiera presentar como EJB. También se puede declarar sólo 1 EJB y llenarlo de métodos para evitar tener que declarar varios EJB en el descriptor, pero esa decisión hace prácticamente inmantenible las clases representativas del EJB debido a que tienden a crecer mucho en un sistema de mediana o alta complejidad.

Ahora bien, una vez visto los pasos necesarios bajo la arquitectura estándar, podemos ver cuáles son las modificaciones necesarias para que el sistema quede de la misma manera. El único cambio necesario que se debe efectuar es a nivel del Factory de obtención del servicio. Esto es :

```
public CodigoD3Svc getCodigoD3Svc()
{
    try
    {
        return (CodigoD3Svc) DelegateSrv.newInstance( CodigoD3Svc.class ) ;
    }
    catch ( Exception ex )
    {
        throw new RuntimeException( ex.getMessage() , ex );
    }
}
```

Este cambio se debe efectuar por cada método que corresponde al método que recupera el servicio. Como vemos es un cambio bastante simple y ahorra mucho código y posibles errores de programación.

Junto con lo anterior, es necesario hacer otra modificación que es a nivel de descriptor EJB. Este cambio, no es necesario repetirlo por cada interfaz de servicio que se quiera exponer como EJB, sino que basta hacerlo sólo 1 vez en la aplicación. La configuración correspondiente es similar a la de la exposición de servicios EJB normales, con la salvedad de que se presenta como EJB el EJB genérico visto en el capítulo de arquitectura.

```
<session id="CommandFacade">
    <ejb-name>CommandFacade</ejb-name>
    <home>com.its.delta.d3.proxy.facade.CommandFacadeHome</home>
    <remote>com.its.delta.d3.proxy.facade.CommandFacade</remote>
    <ejb-class>com.its.delta.d3.proxy.facade.CommandFacadeBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
</session>
```

Como podemos apreciar, la reducción en cantidad de código y de clases involucradas es enorme, lo cual permite presentar de manera sumamente simple servicios como EJB y de un modo transparente para el desarrollador.

6.3. Modificaciones necesarias para el uso de ORM

Para la muestra de uso del ORM y su contraste con el mecanismo de JDBC directo, veremos como se efectúan las operaciones básicas de inserción, modificación, borrado y consultas de base de datos utilizando el mecanismo de JDBC directo que provee la API de Java.

- Inserción JDBC Normal

El siguiente código muestra una inserción típica en base de datos utilizando el mecanismo de JDBC.

```
1  public Long create( Usuario usuario ) throws PersistenceException
2  {
3     Connection conn = null;
4     PreparedStatement ps = null;
5     ResultSet rs = null;
6     String sql = "";
7     try
8     {
9        conn = getConnection();
10       Long ret = null;
11       sql +="INSERT INTO DB2ADMIN.NVUSU_USUARIO(NVUSU_USERNAME,NVUSU_PASSWORD,
            NVUSU_FECHACREACION,NVUSU_VALID,NVROL_ID)" + " VALUES(?,?,?,?,?)";
12       ps = conn.prepareStatement( sql );
13       ps.setString(1, usuario.getUsername());
14       ps.setString(2, usuario.getPassword());
15       if ( null != usuario.getFechaCreacion() )
16       {
17          ps.setDate(3 , new java.sql.Date( usuario.getFechaCreacion().getTime
               () ) );
18       }
19       else
20       {
21          ps.setNull(3, java.sql.Types.DATE );
22       }
23       if ( null != usuario.getValid() )
24       {
25          ps.setBoolean(4, usuario.getValid().booleanValue());
26       }
27       else
28       {
29          ps.setNull( 4, java.sql.Types.BOOLEAN );
30       }
31       if ( null != usuario.getRol() && null != usuario.getRol().getId() )
32       {
33          ps.setLong( 5, usuario.getRol().getId().longValue() );
34       }
35       else
36       {
37          ps.setNull( 5 , java.sql.Types.BIGINT );
38       }
39       ps.executeUpdate();
40       // Obtengo el valor autogenerado
```

```
41          sql = "VALUES IDENTITY_VAL_LOCAL()";
42          ps = conn.prepareStatement(sql);
43          rs = ps.executeQuery();
44          if ( rs.next() )
45          {
46             ret = new Long ( rs.getLong( 1 ) );
47          }
48          return ret;
49       }
50       catch ( Exception ex )
51       {
52          throw new PersistenceException(ex.getMessage(),  ex);
53       }
54       finally
55       {
56          close( conn , ps , rs );
57       }
58 }
```

Este código se puede separar en 4 partes

1. La declaración e inicialización de los objetos necesarios para la consulta (líneas de 1 a 10).
2. Generación de la consulta SQL, y la respectiva asociación de los parámetros de entrada con la consulta (línea 11 a 38).
3. Ejecución de la consulta.
4. Cerrado de recursos involucrados en la consulta.

- Inserción ORM

En este caso el código es sumamente simple. Principalmente se invoca al ORM indicando la tabla involucrada en el proceso, en conjunto con el objeto de negocios que se desea guardar en base de datos. Todo el proceso de obtención de conexión, generación de la consulta apropidad, ejecución y finalmente liberación de los recursos es automático.

```
1 public Long create(Usuario usuario) throws Exception
2 {
3    OID ret = (OID) s_genericPM.create(DatabaseConstant.TABLA_NVUSU_USUARIO,
         usuario);
4    return ret.getLongValue();
5 }
```

- Modificación JDBC Normal

El código para la modificación de un registro en base de datos se muestra a continuación.

```
1  public Integer update(Usuario usuario) throws Exception
2  {
3     Connection conn = null;
4     PreparedStatement ps = null;
5     ResultSet rs = null;
6     String sql = "";
7     try
8     {
9        Integer ret = null;
10       conn = getConnection();
11       sql +="UPDATE DB2ADMIN.NVUSU_USUARIO SET NVUSU_USERNAME =  ?  , " + "
            NVUSU_PASSWORD =  ?  , NVUSU_FECHACREACION =  ?  , NVUSU_VALID =  ?
              WHERE NVUSU_ID =  ?    " ;
12       ps = conn.prepareStatement( sql );
13       ps.setString(1, usuario.getUsername());
14       ps.setString(2, usuario.getPassword());
15       if ( null != usuario.getFechaCreacion() )
16       {
17          ps.setDate(3, new java.sql.Date ( usuario.getFechaCreacion().getTime
               () ));
18       }
19       else
20       {
21          ps.setNull( 3, java.sql.Types.DATE );
22       }
23       if ( null !=  usuario.getValid() )
24       {
25          ps.setBoolean(4, usuario.getValid().booleanValue() );
26       }
27       else
28       {
29          ps.setNull(4, java.sql.Types.BOOLEAN );
30       }
31       if ( null != usuario.getId() )
32       {
33          ps.setLong(5, usuario.getId().longValue() );
34       }
35       else
36       {
37          ps.setNull(5, java.sql.Types.BIGINT );
38       }
39       ret = new Integer( ps.executeUpdate() );
40       return ret;
41    catch ( Exception ex )
42    {
43       throw new PersistenceException(ex.getMessage(),  ex);
44    }
45    finally
46    {
47       close( conn , ps , rs );
48    }
49 }
```

Como nos podemos percatar el código es muy similar al caso de la inserción en base de datos. Lo que varía principalmente es la consulta SQL la cual en este caso corresponde a una modificación del registro en vez de la inserción.

- Modificación ORM

Al igual que en el caso de la inserción se invoca al ORM indicando la tabla involucrada en el proceso, en conjunto con el objeto de negocios que se desea modificar en base de datos.

```
1 public Integer update(Usuario usuario) throws Exception
2 {
3     return s_genericPM.update(DatabaseConstant.TABLA_NVUSU_USUARIO, usuario);
4 }
```

- Borrado JDBC Normal

Como es esperado, el código para este caso es similar a los anteriores, con la respectiva modificación de la consulta SQL (en este caso de borrado) .

```
1  public Integer deleteById(Long id) throws Exception
2  {
3      Connection conn = null;
4      PreparedStatement ps = null;
5      ResultSet rs = null;
6      String sql = "";
7      try
8      {
9          Integer ret = null;
10         conn = getConnection();
11         sql +="DELETE FROM DB2ADMIN.NVUSU_USUARIO WHERE NVUSU_ID =  ?" ;
12         ps = conn.prepareStatement( sql );
13         ps.setLong(1, id.longValue() );
14         ret = new Integer( ps.executeUpdate() );
15         return ret;
16     }
17     catch ( Exception ex )
18     {
19         throw new PersistenceException(ex.getMessage(),  ex);
20     }
21     finally
22     {
23         close( conn  , ps  , rs );
24     }
25 }
```

- Borrado ORM

Al igual que en el casos anteriores se invoca al ORM indicando la tabla involucrada en el proceso y el identificador del objeto que se desea borrar. Para casos de llaves primarias compuestas, se envía el objeto de negocios completo como parámetros de entrada para el borrado.

```
1 public Integer deleteById(Long id) throws Exception
2 {
3     return s_genericPM.deleteById(DatabaseConstant.TABLA_NVUSU_USUARIO, id);
4 }
```

- Consulta JDBC Normal

Para el caso de la obtención de un registro de base de datos, el mecanimo cambia un poco debido a que es necesario recuperar los datos de la base de datos y asociarlos al objeto de negocio. Sin embargo la mecánica no varía mucho con respecto a los casos anteriores.

```
1  public Usuario retrieveById(Long id) throws Exception
2  {
3     Connection conn = null;
4     PreparedStatement ps = null;
5     ResultSet rs = null;
6     String sql = "";
7     try
8     {
9        conn = getConnection();
10       sql +="SELECT NVUSU_ID , NVUSU_USERNAME , NVUSU_PASSWORD ,
            NVUSU_FECHACREACION , NVUSU_VALID , NVROL_ID  " +"FROM DB2ADMIN.
            NVUSU_USUARIO WHERE NVUSU_ID =  ?" ;
11       ps = conn.prepareStatement( sql );
12       ps.setLong(1, id.longValue());
13       rs = ps.executeQuery();
14       Usuario retorno = null;
15       if ( rs.next() )
16       {
17          retorno = new Usuario();
18          retorno.setId( new Long ( rs.getLong( "NVUSU_ID" ) ) );
19          retorno.setFechaCreacion( rs.getDate( "NVUSU_FECHACREACION"));
20          retorno.setPassword(rs.getString("NVUSU_PASSWORD"));
21          retorno.setUsername(rs.getString("NVUSU_USERNAME"));
22          retorno.setValid( new Boolean ( rs.getBoolean("NVUSU_VALID") ));
23       }
24       return retorno;
25    }
26    catch ( Exception ex )
27    {
28       throw new PersistenceException(ex.getMessage(),  ex);
29    }
30    finally
31    {
32       close( conn , ps , rs );
33    }
34 }
```

- Consulta ORM

La obtención del objeto de negocios desde la base de datos es muy similar a los casos anteriores con la salvedad que se debe indicar un parámetro extra el cual es la clase del objeto de negocios que se quiere obtener.

```
1  public Usuario retrieveById(Long id) throws Exception
2  {
3     return (Usuario)s_genericPM.retrieveById(DatabaseConstant.
         TABLA_NVUSU_USUARIO, id,Usuario.class);
4  }
```

- Consulta Compleja JDBC Normal

Este caso la diferencia que se tiene con respecto a la consulta anterior es la complejidad de la consulta. Sin embargo el resto del código es sumamente similar, ya que luego de obtener el resultado de la consulta es necesario generar la lista de objetos de negocios obtenida a partir de la consulta.

```
1  public List getCodigoTOListFullCaminoImpl(Long idTipoCodigo) throws
       PersistenceException
2  {
3      Connection conn = null;
4      PreparedStatement ps = null;
5      ResultSet rs = null;
6      String sql = "";
7      try
8      {
9          conn = getConnection();
10         sql +="SELECT DOS.COD_CODIGOID AS DOS_COD_CODIGOID , DOS.
               TIPCOD_TIPOCODIGO AS DOS_TIPCOD_TIPOCODIGO , " + "DOS.COD_CODIGO AS
               DOS_COD_CODIGO , DOS.COD_ACTIVO AS DOS_COD_ACTIVO , DOS.
               COD_DESCRIPCION AS DOS_COD_DESCRIPCION , " + "UNO.TIPCOD_TIPOCODIGO
               AS UNO_TIPCOD_TIPOCODIGO , UNO.TIPCOD_NOMBRE AS UNO_TIPCOD_NOMBRE , "
               + "UNO.TIPCOD_DESCRIPCIONTIPO AS UNO_TIPCOD_DESCRIPCIONTIPO , " + "
               UNO.TIPCOD_ESMODIFICABLE AS UNO_TIPCOD_ESMODIFICABLE " + "FROM
               DB2ADMIN.COD_CODIGO DOS INNER JOIN DB2ADMIN.TIPCOD_TIPOCODIGO UNO ON
               " + "( DOS.TIPCOD_TIPOCODIGO = UNO.TIPCOD_TIPOCODIGO AND UNO.
               TIPCOD_TIPOCODIGO = ? )";
11         ps = conn.prepareStatement( sql );
12         ps.setLong(1, idTipoCodigo.longValue());
13         rs = ps.executeQuery();
14         List ret = new ArrayList();
15         while ( rs.next() )
16         {
17             CodigoTO codigo = new CodigoTO();
18             TipoCodigoTO tipoCodigo = new TipoCodigoTO();
19             codigo.setActivo( new Boolean ( rs.getBoolean("DOS_COD_ACTIVO") ) );
20             codigo.setCodigo( rs.getString("DOS_COD_CODIGO") );
21             codigo.setDescripcion(rs.getString("DOS_COD_DESCRIPCION"));
22             codigo.setId( new Long ( rs.getLong("DOS_COD_CODIGOID") ));
23             tipoCodigo.setCodigo( new Long ( rs.getLong("UNO_TIPCOD_TIPOCODIGO")
                   ) );
24             tipoCodigo.setDescripcion(rs.getString("UNO_TIPCOD_DESCRIPCIONTIPO"))
                   ;
25             tipoCodigo.setModificable( new Boolean ( rs.getBoolean("
                   UNO_TIPCOD_ESMODIFICABLE") ) );
26             tipoCodigo.setNombre(rs.getString("UNO_TIPCOD_NOMBRE"));
27             codigo.setTipo( tipoCodigo );
28             ret.add( codigo );
29         }
30         return ret;
31     }
32     catch ( Exception ex )
33     {
34         throw new PersistenceException(ex.getMessage(), ex);
35     }
```

```
36      finally
37      {
38          close( conn , ps , rs );
39      }
40  }
```

- Consulta Compleja ORM

Como vimos en el capítulo 6, para utilizar la clase Camino uno define una tabla, y luego sobre esa tabla va haciendo los respectivos join con las tablas restantes de la consulta, en conjunto con los parámetros que se entregan de filtro por cada tabla. En este caso, se parte de la Codigo y luego se hace un join con la tabla TipoCodigo usando como filtro del identificador del tipo de codigo. Esto genera la misma consulta que se mostró en el código anterior de consulta vía JDBC normal.

```
1  public List getCodigoTOListFullCaminoImpl(Long idTipoCodigo) throws Exception
2  {
3      Camino camino = new Camino( new Table(DatabaseConstant.TABLA_CODIGO, "DOS")
           );
4      camino.join( new Table(DatabaseConstant.TABLA_TIPOCODIGO, "UNO"),new
           QueryParameters(new TipoCodigoTO(idTipoCodigo) ) );
5      List lista = s_genericExecutor.executeQueryForMapList( camino.getQuery() );
6      return getRealList(lista);
7  }
```

Cabe destacar que para efectos de las operaciones efectuadas con el ORM, es necesario hacer la configuración respectiva de los objetos de negocios en el archivo XML que tiene la configuración para transformar de objetos de base de datos a objetos de negocios. Sin embargo esta configuración es necesario hacerla sólo 1 vez dentro del sistema, no por cada vez que se quiera operar con el objeto de negocios, como es el caso que acabamos de ver en los ejemplos de codificación para inserción, modificación y obtención del objeto de negocios a partir de la base de datos. Como muestra, la configuración para el objeto de negocios “Usuario” es la siguiente:

```
<mapping map-id="Usuario">
    <class-a>com.its.nieve.example.mgr.to.Usuario</class-a>
    <class-b>java.util.HashMap</class-b>

    <!-- Campos a excluir -->
    <field-exclude>
        <a>rol</a>
        <b key="EXCLUDED">this</b>
    </field-exclude>

    <!-- Campos a considerar -->
    <field>
        <a>id</a>
        <b key="NVUSU_ID">this</b>
    </field>
    <field>
        <a>username</a>
        <b key="NVUSU_USERNAME">this</b>
```

```
    </field>
    <field>
        <a>password</a>
        <b key="NVUSU_PASSWORD">this</b>
    </field>
    <field>
        <a>fechaCreacion</a>
        <b key="NVUSU_FECHACREACION">this</b>
    </field>
    <field>
        <a>valid</a>
        <b key="NVUSU_VALID">this</b>
    </field>
    <field>
        <a>rol.id</a>
        <b key="NVROL_ID">this</b>
    </field>
</mapping>
```

Dado los ejemplos de código, podemos observar varios puntos comunes y repetitivos en los ejemplos utilizando el mecanismo de JDBC directo a base de datos.

1. Obtención de conexión.
2. Consulta de SQL, la cual muchas veces es trivial y repetitiva.
3. Asociación de parámetros a la consulta.
4. Ejecución de consulta.
5. Liberación de recursos.

La idea principal de este ORM es precisamente disminuir todas las tareas que eran candidatas a ser automatizables, de manera de disminuir la carga del desarrollador y la tendencia a errores que naturalmente se tiene al efectuar tareas tan monótonas. Hay que considerar que además de los pasos mencionados anteriormente, ciertos procesos como la conversión de tipos, o bien validaciones de rango o nulidad de un valor se hacen de manera automática a partir de los metadatos obtenidos de la base de datos.

Para los casos de las operaciones básicas de mantención de registros en base de datos, el código necesario para ello utilizando el mecanismo de ORM, es prácticamente un 10 % del código necesario utilizando JDBC directo.

Viendo el contraste entre la codificación vía JDBC directo y la codificación necesaria cuando se utiliza el ORM, los cambios necesarios para migrar de un mecanismo a otro son los siguientes:

1. Definir los objetos de negocios y su asociación a los objetos de base de datos en el archivo XML de configuración

2. Obtener los metadatos utilizando el mecanismo que provee el ORM. Este es un proceso automático que se debe ejecutar cuando hay cambios en el modelo de datos.

3. Cambiar la codificación de JDBC directo al ORM

Capítulo 7

Conclusiones

7.1. Métricas

Para efectos de poder tener algunas métricas con respecto a las ventajas y desventajas que proporcionan estas capas de abstracción, es necesario tomar en cuenta los antecedentes y estándares de codificación dentro del Proyecto Delta. Algunas de las normas relevantes y que son de impacto para la cuantificación de estas métricas se enumeran a continuación las 2 más relevantes:

1. Los servicios no debiesen tener en promedio más de 30 métodos declarados en su interfaz.
2. Cada implementación de un servicio no debiese superar en promedio las 10 líneas por método.

También consideraremos que el costo de desarrollo de estas capas de abstracción no debiese ser tomando en cuenta para estas métricas, sino más bien, debiese ser considerado como un proyecto aparte.

7.1.1. Líneas de código (LOC)

- **EJBProxy**

Consideremos una cantidad **M** de interfaces de servicio donde cada una posee **N** métodos y cada implementación tiene **Y** líneas de código. El cuadro 7.1 muestra el detalle de cuantas líneas de código se tiene en el caso de la capa **EJBProxy**.

	Servicio	Facade	Mgr	Bean	Impl.	XML	Total
Normal	M*N	M*N	M*N*3	M*N*2	M*N*Y	M*15	M*(7*N + Y*N + 15)
EJBProxy	M*N	0	0	0	M*N*Y	15	M*N*(1 + Y) +15

Cuadro 7.1: Matriz de comparación de LOC entre EJB Normal y EJBProxy

Si consideramos las variables N e Y en sus valores promedios obtenemos lo siguiente.

	LOC	
Normal	M*(7*30 + 10*30+15)	M*525
EJBProxy	M*30(1+10) + 15	M*330 + 15

Cuadro 7.2: Resultado de comparación para capa EJBProxy

Esto nos permite ver que en términos de líneas de código existe una disminución de alrededor de un **37 %** usando EJBProxy

- **ORM**

En este caso, lo relevante a considerar es la cantidad de líneas de código en la implementación de cada método de persistencia, utilizando JDBC Directo o bien la capa de abstracción desarrollada.

En el caso de una consulta JDBC normal, los métodos de persistencia presentan siempre 5 :

1. Declaración de variables representativas de los recursos
2. Código SQL.
3. Ejecución de consulta
4. Asociación de objetos con resultado de consulta
5. Cierre de recursos

La experiencia de uso de esta capa de abstracción indicó que para los desarrolladores de las capas de persistencia que usaban el ORM, la cantidad de código necesario para la escritura de las consulta SQL se redujo un tercio de lo acostumbrado, y la asociación de objetos con los resultados de consulta se redujo en promedio a 1/4, ésto considerando que para una tabla con N columnas, se necesitaban en general por cada método N líneas de código para efectuar la asociación del resultado de la consulta con el objeto de negocio correspondiente.

Si consideramos que para cada tabla se tiene un mantenedor, es decir las 4 operaciones básicas de inserción, lectura, modificación y borrado, al tener la asociación configurada sólo 1 vez en el archivo XML de configuración, el código necesario para la asociación corresponde a 1/4. El cuadro 7.3 representa la relación entre JDBC directo contra ORM.

	Variables	SQL	Ejecución	Asociación	Cierre	XML
JDBC Directo	5	X	1	N	5	0
ORM	0	2*X/3	1	1	0	N/4

Cuadro 7.3: Relación JDBC Directo versus ORM

7.1.2. Cantidad de Archivos

- **EJBProxy**

Al igual que en el caso de la cantidad de líneas de código, consideremos una cantidad **M** de interfaces de servicio. La cantidad de archivos necesarios para crear o modificar un servicio en el caso de EJB Normal y EJBProxy se muestran en el cuadro 7.4:

	Cantidad de Archivos
Normal	M*6
EJBProxy	M*2 + 1

Cuadro 7.4: Cantidad de archivos necesarios a modificar en caso EJBProxy

Podemos notar que la cantidad de archivos a modificar se reduce a **1/3** de la cantidad original sin usar EJBProxy.

- **ORM**

Si consideramos M interfaces de persistencia, la cantidad de archivos necesarios para crear o modificar un servicio de persistencia en el caso de JDBC Directo y usando el ORM se muestran en el cuadro 7.5

	Cantidad de Archivos
JDBC Directo	M*2
ORM	M*3

Cuadro 7.5: Cantidad de archivos necesarios a modificar en caso de ORM

El incremento en este caso se justifica debido al archivo XML donde se encuentra la asociación de objetos de negocios con las tablas del modelo relacional.

7.1.3. Procesos involucrados (configuración)

Para ambos casos, no existen mayores cambios a nivel de configuración. A nivel de EJBProxy no es necesario ningún archivo de configuración extra.

Para el caso del ORM, se necesita indicar el archivo de configuración del ORM con el cual se va a trabajar. En este archivo se indican los datos de cual motor de base de datos se va a utilizar, para efectos de poder generar las consultas SQL correspondientes al motor correcto, y también se indican los datos donde se encuentran los archivos XML de relación tabla - objetos de negocios.

7.1.4. Vendor-Lockin

- **EJBProxy**

La dependencia del proveedor es baja debido a que la tecnología EJB tiene descriptores genéricos los cuales teóricamente debiesen ser suficientes para poder hacer despliegue de un EJB en cualquier contenedor J2EE. Si no fuese así, sería necesario hacer la modificación para poder crear el descriptor XML particular para el contenedor J2EE donde se necesite instalar una aplicación que use EJBProxy.

- **ORM**

En este caso, la dependencia del proveedor es nula gracias a que el generador de consultas genera consultas SQL estándar, y en el caso de que se generen consultas particulares para algún proveedor, éstas se deben generar también para los otros proveedores que se pueden escoger dentro de la configuración. De este modo dado el motor de base de datos, el generador de consultas genera las consultas para el proveedor correspondiente.

7.1.5. Coordinación

- **EJBProxy**

En el caso del EJBProxy existe una menor probabilidad de tener conflictos de versiones con otros desarrolladores debido a la menor cantidad de archivos a modificar cuando se necesita crear o actualizar un servicio, además de la menor cantidad de líneas de código necesarias para esto.

- **ORM**

En el caso del ORM esto no queda tan claro, ya que a pesar de haber una disminución en la cantidad de líneas de código, es necesario hacer modificaciones en más archivos a la hora de codificar la persistencia, lo cual lleva un porcentaje mayor de probabilidad de conflictos de versiones con otros desarrolladores.

7.2. Conclusiones Finales

Como es posible ver a través de las métricas, en el caso del EJBProxy existe una notoria baja en la cantidad de código necesario para poder exponer servicios vía EJB. El encapsulamiento de esta tecnología permite que desarrolladores sin mucha experiencia en el manejo de EJB, puedan desarrollar de manera ágil servicios, en conjunto con disminuir sensiblemente la probabilidad de tener conflictos de versiones con otros desarrolladores. Este encapsulamiento también permite ocultar al desarrollador todo el manejo de excepciones remotas que se tienen al trabajar con EJB, lo cual evita que la definición de las interfaces de servicios no estén ligadas a la tecnología misma EJB, obteniendo como beneficio extra que el costo de una posible migración a otra tecnología de exposición remota de servicios sea menos costosa.

Otra de las ventajas que uno puede concluir luego de este desarrollo, es que esta capa permite al desarrollador poder hacer pruebas unitarias de manera mucho más fácil, ya que no es necesario hacer un despligue de los servicios EJB para poder ejecutar las pruebas, sino más bien, puede hacer pruebas unitarias sin necesidad de efectuar el despliegue de los EJB. Este tiempo de despliegue no es un dato menor, ya que en sistemas de grande y mediana envergadura este tiempo puede tomar hasta varios minutos.

Para el caso de la capa de abstracción de persistencia, la disminución en líneas de código también es notoria a pesar de que se tengan que manipular más archivos que en el caso de JDBC Directo. Lo anterior puede parecer un contrasentido, pero en definitiva el mantener las relaciones de los objetos con las tablas de la base de datos en un archivo de configuración permite un grado de flexibilidad mucho más alto que el que se tiene en el caso JDBC Directo.

Por ejemplo, cuando se eliminan columnas de una tabla basta solamente eliminar la asociación de esa columna con el campo correspondiente al objeto de negocio y generar el archivo de metadatos. Eso tendrá el efecto deseado en la generación de consultas, completación de objetos recuperados de base de datos, validación de tipos y largos que se eliminan o agregan de manera automática, etc. Esto ocurre en toda la capa de persistencia sin necesidad de que el desarrollador tenga que intervenir el código.

Sin embargo viéndolo de una perspectiva de eficiencia, es posible que se pueda evitar el uso excesivo de archivos XML de configuración usando anotaciones Java (en su versión 5 o superior) o bien usando convenciones de nombres las cuales permitan inferir mediante algún algoritmo predefinido dado el nombre de una columna, obtener el nombre del campo en el objeto de negocio correspondiente (y viceversa).

Otra ventaja clara del ORM sobre JDBC directo es que permite generar prototipos sumamente rápido, sobre todo para las operaciones básicas de mantención de tablas. Es más, con un poco de trabajo es bastante alcanzable generar un mantenedor genérico de tablas debido a que en el ORM se tiene toda la información en los metadatos necesaria para generar los mantenedores. En muchos proyectos (sobre todo los de tamaño pequeño) esto puede llegar ser una razón de peso para escoger un ORM.

En definitiva podemos concluir que el desarrollo de capas de abstracción trae muchas más ventajas que perjuicios tales como:

1. Mejoras en tiempos de desarrollo
2. Disminuir y encapsular las complejidades sobre las plataformas que se utilizan al desarrollador
3. Evitar tareas monótonas y mecánicas que usualmente son proclives a error humano
4. Aumenta flexibilidad y mantenibilidad del software
5. Disminuir las dependencias tecnológicas sobre plataformas que se utilizan

7.3. Trabajos futuros y mejoras

Algunos de los trabajos futuros con los cuales sería interesante contar se enumeran a continuación:

EJBProxy

1. Seguridad declarativa a nivel de métodos de servicios. Tal como se tiene en la especificación EJB, un posible trabajo futuro sería implementar la seguridad declarativa a nivel de método. Esto podría hacerse a través de un archivo XML, o bien incluso a través de una clase especial la cual podría traer la configuración desde una base de datos, de manera de que fuese configurable en tiempo de ejecución.

2. Transaccionabilidad a nivel de métodos. Al igual que en el caso anterior, un futuro trabajo sería poder contar con una implementación de transaccionabilidad a nivel de método de servicio. En el caso actual, el EJBProxy tiene configurados todos los métodos como transaccionales, lo cual tiene una penalización de rendimiento que se podría evitar.

ORM

1. Manejo de transacciones fuera de contendor. En este minuto, el ORM no maneja transacciones, por lo que todas las transacciones se manejan a nivel de contenedor vía EJB. Esto obliga a que se tenga que usar este ORM dentro de un contenedor J2EE y además tener que usar EJB para los servicios. En muchos casos esto no es necesario (por ejemplo para proyectos chicos) y lo que se requiere es poder manejar las transacciones a nivel de los objetos de persistencia.

2. Generación de consultas NO-SQL. Sería interesante poder contar con un generador de consultas para lenguajes NO-SQL, por ejemplo como el lenguaje de consulta de otros ORM como Hibernate o bien JPA.

3. Limpieza de código. Existe mucho código que debiese ser eliminado, ya que este se mantuvo para dar compatibilidad a funcionalidades que fueron reemplazadas por implementaciones más genéricas.

4. Integración con Spring. Otro trabajo interesante sería crear las clases necesarias para poder integrar este ORM con Spring, algo que es bastante común en casi todos los ORM que existen actualmente como Hibernate, Ibatis, JPA y otros.

5. Manual de uso. Por la premura del origen de este ORM, no hubo tiempo para generar la documentación adecuada para su configuración y uso. Lo que se cuenta en la actualidad son los ejemplos y las respectivas pruebas unitarias, pero ellas no abarcan todas las funcionalidades que posee el ORM.

Bibliografía

[1] Christopher Alexander. *A pattern Language: Towns,Building,Construction*, 1977.

[2] Ward Cunningham, Kent Beck. *Using Pattern Languages for Object-Oriented Programs*,1987

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.*Design Patterns: Elements of Reusable Object-Oriented Software*, 1994

[4] Edgar Frank Codd, *A Relational Model of Data for Large Shared Data Banks*, 1970

[5] Jini: Jini Network Technology. *http://www.jini.org*

[6] IBATIS Data Mapper framework: *http://ibatis.apache.org/*

[7] W3C, World Wide Web Consortium: *http://www.w3.org/*

[8] Sun Microsystems: *http://www.sun.com*

[9] IBM: *http://www.ibm.com*

[10] Oracle: *http://www.oracle.com*

[11] Core J2EE Patterns: *http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html*

[12] **A**merican **N**ational **S**tandards **I**nstitute: *http://www.ansi.org*

[13] **I**nternation Organization for Standarization : *http://www.iso.org*

[14] Hibernate: *http://www.hibernate.org*

[15] **J**ava **P**ersistence **A**PI: *http://java.sun.com/developer/technicalArticles/J2EE/jpa/*

[16] Spring Remoting: *http://www.springframework.org/docs/reference/remoting.html*

[17] Struts: *http://struts.apache.org/*

[18] **J**ava **S**erver **F**aces: *http://java.sun.com/javaee/javaserverfaces/*

[19] RFC-707: *http://tools.ietf.org/html/rfc707*

[20] XEROX. Courier: *The remote procedure call protocol. Xerox System Integration Standard, Xerox Corporation, Stamford, Conn.*,1981

[21] DCE Open Group: *http://www.opengroup.org/dce/*

[22] OMG, Object Management Group: *http://www.omg.org/*

[23] CORBA: *http://www.corba.org/*

[24] Rod Johnson*: Expert One-on-One: J2EE Design and Development*

[25] Caucho: *http://www.caucho.com/*

[26] Spring: *http://www.springframework.org/*

[27] JCP: Java Community Process: *http://www.jcp.org/en/home/index*

[28] JSR 220: *http://jcp.org/en/jsr/detail?id=220*

[29] Dynamic Proxy Classes**:** *http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html*