Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

PROPUESTA DE UNA ARQUITECTURA DE APLICACIÓN UTILIZANDO HIBERNATE PARA LA PERSISTENCIA

Bryan Alexis Orellana Soberanis

Asesorado por el Ing. Rubén Darío Crespo Valenzuela

Guatemala, febrero de 2010

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

PROPUESTA DE UNA ARQUITECTURA DE APLICACIÓN UTILIZANDO HIBERNATE PARA LA PERSISTENCIA

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR:
BRYAN ALEXIS ORELLANA SOBERANIS

ASESORADO POR EL ING. RUBÉN DARÍO CRESPO VALENZUELA

AL CONFERÍRSELE EL TÍTULO DE
INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, FEBRERO DE 2010

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Inga. Glenda Patricia García Soria
VOCAL II	Inga. Alba Maritza Guerrero de López
VOCAL III	Ing. Miguel Ángel Dávila Calderón
VOCAL IV	Br. Luis Pedro Ortíz de León
VOCAL V	Br. José Alfredo Ortíz Herincx
SECRETARIA	Inga. Marcia Ivonne Véliz Vargas

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADORA	Inga. Virginia Victoria Tala Ayerdi
EXAMINADOR	Ing. Pedro Pablo Hernández Ramírez
EXAMINADOR	Ing. César Augusto Fernández Cáceres
SECRETARIA	Inga. Marcia Ivonne Véliz Vargas

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

PROPUESTA DE UNA ARQUITECTURA DE APLICACIÓN UTILIZANDO HIBERNATE PARA LA PERSISTENCIA,

tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, en julio de 2008.

Bryan Alexis Orellana Soberanis

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA
ESCUELA DE CIENCIAS Y SISTEMAS

Guatemala 03 de septiembre de 2009

Señores
Comisión de revisión de trabajos de graduación
Carrera de Ciencias y Sistemas
Facultad de Ingeniería
Universidad de San Carlos de Guatemala
Guatemala, Ciudad

Respetables Señores:

El motivo de la presente es informarles que como asesor del estudiante Bryan Alexis Orellana Soberanis he procedido a revisar el trabajo de graduación titulado **"Propuesta de una arquitectura de aplicación utilizando Hibernate para la persistencia"** y que de acuerdo a mi criterio el mismo se encuentra concluido y cumple con los objetivos definidos al inicio.

He sostenido reuniones periódicas con el estudiante y luego de haber revisado cuidadosamente el trabajo, considero que cumple con los requisitos de calidad y profesionalismo que deben caracterizar a un futuro profesional de Informática.

Sin otro particular me suscribo de ustedes,

Atentamente,

Rubén Darío Crespo Valenzuela

Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 30 de Septiembre de 2009

Ingeniero
Marlon Antonio Pérez Turk
Director de la Escuela de Ingeniería
En Ciencias y Sistemas

Respetable Ingeniero Pérez:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **BRYAN ALEXIS ORELLANA SOBERANIS,** titulado: **"PROPUESTA DE UNA ARQUITECTURA DE APLICACIÓN UTILIZANDO HIBERNATE PARA LA PERSISTENCIA",** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme,

Atentamente,

Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
INGENIERIA EN CIENCIAS Y SISTEMAS
COORDINACION TESIS

ESCUELA DE CIENCIAS Y SISTEMAS

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÌA
ESCUELA DE CIENCIAS Y SISTEMAS
TEL: 24767644

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, de trabajo de graduación titulado "PROPUESTA DE UNA ARQUITECTURA DE APLICACIÓN UTILIZANDO HIBERNATE PARA LA PERSISTENCIA", presentado por el estudiante BRYAN ALEXIS ORELLANA SOBERANIS, aprueba el presente trabajo y solicita la autorización del mismo.

"ID Y ENSEÑAD A TODOS"

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCIÓN DE BIBLIOTECAS EN CIENCIAS Y SISTEMAS

Ing. Marlon Antonio Pèrez Turk
Director, Escuela de Ingeniería en Ciencias y Sistemas

Guatemala, 15 de febrero 2010

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

Ref. DTG.050.2010

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al trabajo de graduación titulado: **PROPUESTA DE UNA ARQUITECTURA DE APLICACIÓN UTILIZANDO HIBERNATE PARA LA PERSISTENCIA,** presentado por el estudiante universitario **Bryan Alexis Orellana Soberanis,** autoriza la impresión del mismo.

IMPRÍMASE.

Ing. Murphy Olympo Paiz Recinos
DECANO

Guatemala, febrero 2010

/cc
c.c. archivo.

ACTO QUE DEDICO A:

Dios, presente en cada momento, en muchas personas y situaciones en que me dejó ver su acción en mi carrera estudiantil.

Mis padres, Juan Orellana y Norma Soberanis, sin ellos no habría sido posible alcanzar esta meta y lograrla sin ellos no tendría sentido.

Mis hermanas, Lucky, Carina y Vera Lucía, regalos de Dios para mi vida y compañeras incondicionales.

Mi cuñado y mis sobrinos, amigos por convicción.

Mis amigos, señales tangibles del amor de Dios en mi vida. En especial a mis amigos de los grupos juveniles que integré.

Mis compañeros de trabajo, de quienes he aprendido, no solo técnicamente, sino de sus cualidades humanas.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

GLOSARIO

Open Source	O código abierto. Con este nombre es conocido el software concebido para ser desarrollado libremente y que luego admite modificaciones, previa autorización de sus desarrolladores.
Java Data Objects	O JDO por sus siglas. Es un estándar para la persistencia de instancias de clases en Java, que tiene varias implementaciones comerciales.
PL/1	Acrónimo de Programming Language 1. El Lenguaje de Programación 1 fue propuesto por IBM, en su momento para suplir las necesidades. cada vez más diversas, de aplicaciones de software. Su incursión se remonta a inicios de los años 1970.
SQL	Lenguaje estructurado de consultas (*Structured Query Language*) diseñado para obtener acceso a los datos de una base de datos relacional. SQL es un lenguaje de cuarta generación.
DBMS	Sistemas de Gestión de bases de datos, por sus siglas en inglés (DataBase Management System). Su finalidad es brindar una interfaz que permita al usuario y a las aplicaciones interactuar con la base de datos.

RDBMS Acrónimo de *Relational DataBase Management System*. Referirse a DBMS. Existe un conjunto de reglas que se utiliza para evaluar si un DBMS es a su vez un RDBMS.

Tupla Con este nombre se conoce a los conjuntos de datos que generalmente se almacenan en una fila de una tabla.

ODBC Acrónimo de *Open DataBase Connectivity*. Este estándar de acceso a base de datos permite acceder a los datos en una base de datos desde una aplicación. Fue desarrollado por Microsoft Corporation.

URI Acrónimo de *Unified Resource Identifier.* Un URI es una cadena de caracteres que identifica a un servicio, página u otro recurso. Suelen ser de longitud corta.

API Acrónimo de *Application Programming Interface*, Interfaz de programación de aplicaciones. Es un conjunto de métodos provistos por una librería para proporcionar funciones de uso general.

JNDI Acrónimo de Java Naming Data Interface. Este API permite nombrar objetos, así mismo buscarlos a partir de ese nombre. La utilidad de JNDI va muy ligada con la integración entre aplicaciones.

Lookup En el contexto en que se utiliza se refiere a la búsqueda de un recurso, conociendo el nombre que se le da en

	ese entorno.
Cross-platform	El término se utiliza para indicar que una aplicación puede utilizarse independientemente de la plataforma con que se cuente, como en el caso de Java, corriendo sobre MS Windows y Linux.
Alta disponibilidad	Este término ha tomado gran relevancia y se utiliza para indicar que la continuidad de las prestaciones de un servicio o equipo puede garantizarse en un período de tiempo determinado.
Multiusuario	Se refiere a la capacidad que tiene un sistema para procesar las peticiones de varios usuarios simultáneamente.
J2EE	Acrónimo de *Java 2 Enterprise Edition*. De esta manera se conoció a la edición empresarial de Java hasta la versión 1.4. Se emplea para desarrollar aplicaciones en lenguaje Java, aprovechándose de una arquitectura madura y de componentes ampliamente probados.

Java Community Process	Conocida también como JCP. Literalmente, el Proceso de la Comunidad Java, es un proceso que permite a los distintos involucrados en las versiones de la plataforma Java tomar parte en sus definiciones. Este proceso se lleva a cabo a partir de JSR.
JSR	Acrónimo de *Java Specification Request*. Son documentos en los cuales se describe las especificaciones que serán añadidas a la plataforma Java y, de alguna manera, definen las versiones.
Herencia	De esta manera se conoce a la característica de algunos lenguajes que permite a una clase, utilizar la definición (atributos y métodos) de otra clase anterior, de la cual *hereda*.
Polimorfismo	Esta característica surgida de la Programación Orientada a Objetos (POO), permite obtener comportamientos distintos de acuerdo a los parámetros que se envíen. Los comportamientos comparten el nombre, pero actuarán distinto de acuerdo a lo que se requiera.
JavaBean	De esta manera se conoce a un tipo especial de componentes, que encapsulan varios objetos, y que se utilizan ampliamente para construir aplicaciones Java.

Struts	O Apache Struts, es una herramienta que asiste el diseño de aplicaciones, sobre todo aplicaciones web, que utilizan la arquitectura MVC.
JavaServer Faces	Este es el nombre de una tecnología ampliamente utilizada en aplicaciones Java en la web, para el desarrollo de la interfaz de usuario.
Serialización	La serialización es el proceso de convertir un Objeto en una secuencia de *bytes* (codificación), para transportar o hacer persistente dicho objeto.
JTA	Acrónimo de *Java Transaction API*, API para transacciones Java. Este API se utiliza para el manejo de transacciones en aplicaciones que lo requieren; llegando a utilizarse para manejar las transacciones en aplicaciones distribuidas.
HTTP	Acrónimo de *HyperText Transfer Protocol*, es el protocolo de transferencia de hipertexto. Ampliamente utilizado en la web, define la sintaxis y significado de los elementos que se utilizan en esa arquitectura, para la comunicación.
JPA	Acrónimo de *Java Persistence API*; es el API de persistencia Java. Este API provee de funcionalidades necesarias para conseguir el mapeo objeto relacional.

Interfaz Del inglés *interface*. Se refiere al punto en que dos sistemas se conectan; en este contexto es provista por uno de los dos sistemas, para que el otro se sirva de ella.

DAO layer Acrónimo de Data Access Object layer, capa de objetos de acceso a datos. Esta capa provee de funcionalidades de acceso a los datos a otras capas superiores, abstrayendo los detalles de implementación de esas funcionalidades.

Ant O Apache Ant. Es una herramienta que se utiliza para automatizar la tarea de compilación y construcción de un proyecto de desarrollo de software, aunque esto puede extenderse a otras tareas igualmente repetitivas.

Tipo wrapper Se refiere a un tipo de datos que encapsula en si mismo funcionalidad, por lo que provee un conjunto de métodos a los que posible acceder en las variables de ese tipo.

Multiplicidad La multiplicidad se refiere a las relaciones que se establecen entre las tablas de una base de datos y se establecen entre pares. La multiplicidad se escribe como un par de números, que señalan a cada uno de los lados de la misma, por ejemplo: 1:1 se lee como 1 a 1.

Patrón En este trabajo será usado para referirse a Patrón de diseño. Un patrón de diseño es una generalización de la solución a un problema, igualmente, de diseño.

Façade Literalmente fachada. En este trabajo se referirá al patrón façade. Este patrón se utiliza para generalizar una interfaz que se utilice para acceder a varios servicios, simultáneamente.

HTML Acrónimo de HyperText Markup Language. El Lenguaje de Marcas de Hipertexto es ampliamente utilizado para la construcción de páginas web, y provee tanto de estructura como de forma a los elementos de las páginas.

Proxy Se utiliza el término para referirse a un sistema o un dispositivo que hace de intermediario para permitir a otro cumplir algún objetivo. En este trabajo se usará en varias ocasiones para referirse al patrón de diseño proxy.

POJO Acrónimo de *Plain Old Java Object*, se refiere a clases planas de uso genérico y que no dependen de un *Framework* específico.

Bag Literalmente bolsa. En este trabajo se refiere a una de las colecciones que ofrece el lenguaje Java, y, que es de las más generales, registrando los pares: elemento y

	la frecuencia con que aparece en el *bag*.
Lazy	Literalmente perezoso. En el presente trabajo se usará para referirse a una de las formas que ofrece Hibernate para recuperar objetos de la base de datos, si la recuperación se hace *lazy*, significará que se hará por demanda.
Singleton	En este trabajo se usará para referirse al patrón Singleton. En este patrón se utiliza una única instancia de una clase, en el entorno de una aplicación, restringiendo la creación de otras instancias.
Escalabilidad	Es una característica de los sistemas, que le permite extender sus operaciones, o bien, crecer sin afectar los servicios que ofrece.
IDE	Acrónimo de *Integrated Development Environment*. El entorno de desarrollo integrado es un conjunto de herramientas que permite el desarrollo de aplicaciones de software.
Portabilidad	Es una característica que posee una aplicación de software para funcionar, independientemente de la plataforma en que se ejecute.
Mantenibilidad	Propiedad que tiene un sistema que indica cuantitativamente, el esfuerzo para realizar el mantenimiento de dicho sistema.

XML	Acrónimo de *eXtensible Markup Language*, es un metalenguaje de etiquetas extensible, desarrollado por el *World Wide Web Consortium*.
Plugin	Es una aplicación que le aporta funcionalidad a otra y que, generalmente, extiende la funcionalidad de esta última.
JAR	Archivo Java (del inglés *Java Archive*) que permite ejecutar una aplicación escrita en Java.
Framework	Un *Framework* es un conjunto de software que provee una estructura para asistir el desarrollo de aplicaciones de software.

RESUMEN

La persistencia de los datos ocupa buena parte del esfuerzo que se emplea en el desarrollo de las aplicaciones del software y por ello se designa recursos para llevar a cabo las tareas pertinentes para esa persistencia.

Han surgido varias alternativas para mejorar la persistencia de las aplicaciones, y se ha creado toda una disciplina alrededor del tema de la persistencia. Por otro lado, también ha habido un gran desarrollo en automatizar la generación de las rutinas de persistencia de una aplicación y de simplificar las tareas de persistencia.

Las herramientas de mapeo objeto-relacional son una alternativa para realizar la persistencia en las aplicaciones, resolviendo la mayoría de casos que se presentan y aplicando perfectamente para las aplicaciones de software grandes y medianas.

Por otro lado, cuando se trabaja con aplicaciones de software grandes y medianas, se busca reducir la complejidad y el acoplamiento en el código y permitir un mejor mantenimiento, ofreciendo agilidad en los cambios requeridos. Así mismo, para manejar muchos aspectos complejos en las aplicaciones medianas y grandes se busca aprovechar los beneficios de una arquitectura de aplicación.

La arquitectura de aplicación ofrece beneficios para las aplicaciones de software, tanto para su construcción, como para el resto de su ciclo de vida. Es

útil establecer si esos beneficios pueden llegar a representar también beneficios económicos para una empresa. Más útil resulta estimar el ahorro de recursos que podría representar un componente para persistencia como parte de la arquitectura de aplicación.

Los esfuerzos por implementar una arquitectura de aplicación y desarrollar componentes de software reutilizables para tareas comunes representa grandes costos para una empresa, así se trate de una empresa de software. Por esto es necesario que el costo por esta implementación se justifique por los beneficios en el desarrollo de las aplicaciones que se mencionaron anteriormente.

Un componente de software que permita agilizar el desarrollo de las operaciones de acceso a los datos (persistencia) será de gran utilidad en la mayoría de las aplicaciones y es interesante su desarrollo a partir de las herramientas de mapeo objeto-relacional, siendo una buena opción Hibernate para emprender ese desarrollo. Es este componente de software, empotrado en una arquitectura de aplicación y el análisis de los beneficios que ofrece su desarrollo el objeto de estudio de este trabajo.

OBJETIVOS

General:

Proponer una arquitectura de aplicación que permita la reutilización de un componente de persistencia y analizar el beneficio al realizarlo.

Específicos:

1. Definir el concepto de mapeo objeto-relacional.

2. Explicar las características de una herramienta de mapeo objeto-relacional, Hibernate.

3. Analizar la importancia de utilizar una arquitectura de aplicación en el desarrollo de aplicaciones de tamaño mediano o grande.

INTRODUCCIÓN

Hay muchos costos asociados al desarrollo de aplicaciones de software. Cuando se trata de grandes aplicaciones, estos costos se multiplican también; en algunas ocasiones los costos que implican el desarrollo de esas aplicaciones son despreciables para una organización si se toma en cuenta que es mayor el costo de no contar con la aplicación en el tiempo estipulado inicialmente.

Un departamento de desarrollo de software trata, por estas razones, de minimizar el esfuerzo de desarrollar las aplicaciones que le son requeridas. El problema no es sencillo, resolver requerimientos muy diversos, ofreciendo soluciones que se presuman óptimas y hacerlo en el menor tiempo posible. Un elemento fundamental para resolver ese problema se encuentra en el diseño de las aplicaciones, y a pesar de la naturaleza diversa de las mismas, debe establecerse directivas elementales a seguir para desarrollar las aplicaciones.

Como parte de esas directivas debe establecerse una arquitectura que rija a las aplicaciones que se desarrollan en un departamento, con las excepciones que se presenten para seguir la arquitectura elegida.

El activo más importante de muchas organizaciones es, en la actualidad, la información. Existe, por ello, una capa encargada del acceso a los datos en las aplicaciones construidas. Como se observará, el adecuado manejo de la información es vital para las organizaciones y por ende una constante preocupación en el diseño y desarrollo de las aplicaciones.

Hibernate se presenta como una herramienta de mapeo objeto/relacional con grandes capacidades en la construcción del acceso a datos de las aplicaciones, desde diversas plataformas, pero sobre todo, cuando se utiliza el estándar J2EE.

1. TRES ALTERNATIVAS PARA LA PERSISTENCIA EN APLICACIONES MEDIANAS Y GRANDES

1.1. Las herramientas de persistencia

La utilización de herramientas de persistencia en las grandes aplicaciones de software ha cobrado mayor relevancia por la necesidad de reducir el esfuerzo de desarrollar la capa de acceso a datos y aún así conservar, o aún, mejorar las prestaciones y características de la misma.

Es posible generalizar la utilización de bases de datos en el nivel más bajo de almacenamiento de información de las aplicaciones, mientras el acceso a los datos sigue siendo motivo de estudio y son aún muy diversas las opciones a considerar.

En cuanto a las bases de datos, siguen siendo relacionales en su mayoría, tanto los productos comerciales como los *Open Source* (código abierto). En tanto, las herramientas de persistencia se han multiplicado y diversificado su campo.

Las herramientas de persistencia cumplen con la tarea de proveer acceso a la información de la base de datos desde las aplicaciones construidas.

Entre algunas de estas herramientas se encuentran: Hibernate, iBatis, TopLink, Cocobase y FastObjects, Kodo, JDO Genie, LiDo, Exadel JDO, IntelliBO, JRelay JDO (todos ellos comerciales), TJDO y XORM (de código abierto). Específicamente, las herramientas mencionadas son utilizadas como herramientas de mapeo objeto relacional (ORM, por su acrónimo en inglés).

La necesidad de una especificación para estandarizar la forma de programar en lenguaje Java con estas herramientas dio origen a *Java Data Objects* (JDO).

1.1.1. Una definición de persistencia

"Persistencia es la capacidad que tiene el desarrollador para que los datos se conserven al finalizar la ejecución de un proceso, de forma que se puedan reutilizar en otros procesos."[1]

"Se dice que es necesaria la persistencia en un sistema cuando una transacción es confirmada, sus cambios deben ser grabados sobre la base de datos y no deben perderse debido a fallos de otras transacciones o del sistema."[2]

"La persistencia es un mecanismo para guardar datos u objetos en una memoria externa tal como archivos o bases de datos."[3]

1.2. Una definición de base de datos relacional

"Una base de datos relacional es una base de datos en donde todos los datos visibles al usuario están organizados estrictamente como tablas de valores, y en donde todas las operaciones de la base de datos operan sobre estas tablas.

[1] http://www.monografias.com/trabajos5/tipbases/tipbases.shtml
[2] http://html.rincondelvago.com/base-de-datos-relacional_1.html
[3] http://swik.net/persistence (Traducción)

Estas bases de datos son percibidas por los usuarios como una colección de relaciones normalizadas de diversos grados que varían con el tiempo.

El modelo relacional representa un sistema de bases de datos en un nivel de abstracción un tanto alejado de los detalles de la máquina subyacente, de la misma manera como, por ejemplo, un lenguaje del tipo de PL/1 representa un sistema de programación con un nivel de abstracción un tanto alejado de los detalles de la máquina subyacente. De hecho, el modelo relacional puede considerarse como un lenguaje de programación más bien abstracto, orientado de manera específica hacia las aplicaciones de bases de datos."[4]

"Una base de datos relacional es un conjunto de relaciones (o tablas) de dos dimensiones."[5]

"Un sistema de administración de base de datos relacional, RDBMS (acrónimo en inglés), es un tipo de sistema de gestión de base de datos que almacena información en la forma de tablas relacionadas. Las bases de datos relacionales son poderosas porque requieren de pocas asunciones respecto de cómo la información está relacionada o de cómo será extraída de la base de datos. Como resultado, la misma base de datos puede ser vista en muchas diferentes formas.

Una característica importante de los sistemas relacionales es que una sola base de datos puede ser dividida en varias tablas. Esto difiere de las bases de datos planas, en las que la base de datos está contenida en una sola tabla."[6]

[4] http://www.fismat.umich.mx/~elizalde/tesis/node15.html [Date, 1993]
[5] http://www.lcc.uma.es/~esc/docenciabd/Tema%202%20Relacional.pdf (Edgar Frank Codd)
[6] http://isp.webopedia.com/TERM/R/RDBMS.html (Traducción)

Para los fines de este trabajo se usará el término base de datos en su acepción más práctica, definiéndola como una organización de datos en tablas, en las que cada una de estas tablas tiene un significado para la organización, y esos datos “interesantes” se almacenan como filas (registros) de esas tablas.

Debido a las restricciones que presentan las herramientas de persistencia que son el objeto de este trabajo, el subconjunto de bases de datos interesante será el de aquellas que cumplen con ser relacionales y a cuales puede administrarse por algún subconjunto de instrucciones del lenguaje estructurado de consultas, SQL (acrónimo de *Structured Query Language*). Aunque se hará referencia a un único administrador de base de datos, DBMS (acrónimo de *DataBase Management System*), la aplicación de lo expuesto puede extenderse a otros DBMS en el subconjunto de base de datos mencionado.

Las bases de datos relacionales son ampliamente utilizadas por su gran flexibilidad, así como su robustez para el manejo de datos. Las bases de datos relacionales tienen una fuerte fundamentación teórica y por ello permiten garantizar la integridad, entre otras características.

1.2.1. Otras definiciones del modelo relacional

Una base de datos relacional, como se señaló en el apartado anterior, es un conjunto de una o más tablas que pueden vincularse entre sí, por medio de un campo.

En el modelo relacional una relación es un conjunto de tuplas o filas y, para simplificar la teoría de las bases de datos relacionales creada por Edgar F. Codd, una relación se piensa como una tabla.

Las tablas a su vez tienen una estructura interna en la que se distinguen las filas o tuplas y las columnas o atributos.

Una fila o tupla es un conjunto de valores para cada cada campo en un momento dado.

1.3. Panorama general de las herramientas de persistencia en las aplicaciones actuales

Como se mencionó en un principio el uso de las herramientas de persistencia ha cobrado una relevancia proporcional a la necesidad de la utilización de datos por parte de las aplicaciones construidas. Estas herramientas, en general, intentan minimizar el esfuerzo para desarrollar el código fuente para acceso a los datos, esté este confinado en una capa especializada para ello (que suele ocurrir en aplicaciones de gran dimensión) o sea este parte de una aplicación menos estructurada, que carezca de una arquitectura.

En concreto, el interés de este estudio es mostrar las propiedades de las herramientas de mapeo objeto relacional como herramientas de persistencia.

Una de las necesidades que las herramientas de persistencia buscan subsanar es la de desacoplar las capas de lógica o de interfaz de usuario de la base de datos. En aplicaciones pequeñas o medianas este esfuerzo podría parecer innecesario, sin embargo, en grandes aplicaciones un cambio del DBMS elegido, irremediablemente acarrearía un gran esfuerzo de codificación y configuración de la aplicación completa.

Desacoplar las distintas capas que constituyen la arquitectura de una aplicación de acuerdo a la finalidad que cada una de ellas cumple es un tema ya ampliamente tratado en muchos trabajos. Este trabajo, enfocado en la persistencia de las aplicaciones, constituida esta por una o más capas de la arquitectura, tratará sobre la necesidad de considerar esa misma arquitectura de forma tal que un cambio en las capas de acceso a datos no implique cambios en otras capas de la aplicación, cuyas funcionalidades persigan objetivos distintos al acceso a datos.

Se tratará ahora acerca de las alternativas que se tiene para la persistencia de los datos en aplicaciones medianas y grandes, no con ello desvirtuando la posibilidad de utilizar alguna de estas herramientas en aplicaciones pequeñas, pero sí afirmando que en esos casos probablemente no valga la pena hacer un esfuerzo tan grande como el que algunas de estas herramientas requieren:

1.4. JDBC

La primera alternativa es JDBC, que es utilizada con mucha frecuencia por la sencillez de su implementación y porque su API se apega a los pasos que un desarrollador comúnmente ha ejecutado en otros lenguajes para gestionar las bases de datos.

"**JDBC** es el acrónimo de *Java Database Connectivity*, un API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java independientemente del sistema de operación donde se ejecute o de la base de datos a la cual se accede utilizando el dialecto SQL del modelo de base de datos que se utilice.

El API JDBC se presenta como una colección de interfaces Java y métodos de gestión de manejadores de conexión hacia cada modelo específico de base de datos. Un manejador de conexiones hacia un modelo de base de datos en particular es un conjunto de clases que implementan las interfaces Java y que utilizan los métodos de registro para declarar los tipos de localizadores a base de datos que pueden manejar. Para utilizar una base de datos particular, el usuario ejecuta su programa junto con la librería de conexión apropiada al modelo de su base de datos, y accede a ella estableciendo una conexión, para ello provee en localizador a la base de datos y los parámetros de conexión específicos. A partir de allí puede realizar con cualquier tipo de tareas con la base de datos a las que tenga permiso: consultas, actualizaciones, creado modificado y borrado de tablas, ejecución de procedimientos almacenados en la base de datos, etc."[7]

"JDBC permite ejecutar sentencias SQL a los programas desarrollados en Java. Esto permite a los programas Java interactuar con cualquier base de datos que cumple con el estándar SQL. Dado que todos los DBMS relacionales soportan SQL, y que Java corre en muchas plataformas, JDBC hace posible escribir una aplicación de base de datos que corre en distintas plataformas y con diferentes DBMS. Haciendo una comparación, JDBC es similar a ODBC, pero fue diseñado específicamente para programas Java, mientras ODBC es independiente del lenguaje. JDBC fue desarrollado por JavaSoft, una subsidiaria de Sun Microsystems."[8]

1.4.1. Arquitectura de JDBC

JDBC trabaja encapsulando las operaciones en la base de datos por medio de una conexión. Estas conexiones se obtienen generalmente a través

[7] http://es.wikipedia.org/wiki/JDBC
[8] http://isp.webopedia.com/TERM/J/JDBC.html (Traducción)

de la interfaz *java.sql.Connection* y permiten la creación de sentencias para ser ejecutadas en la base de datos.

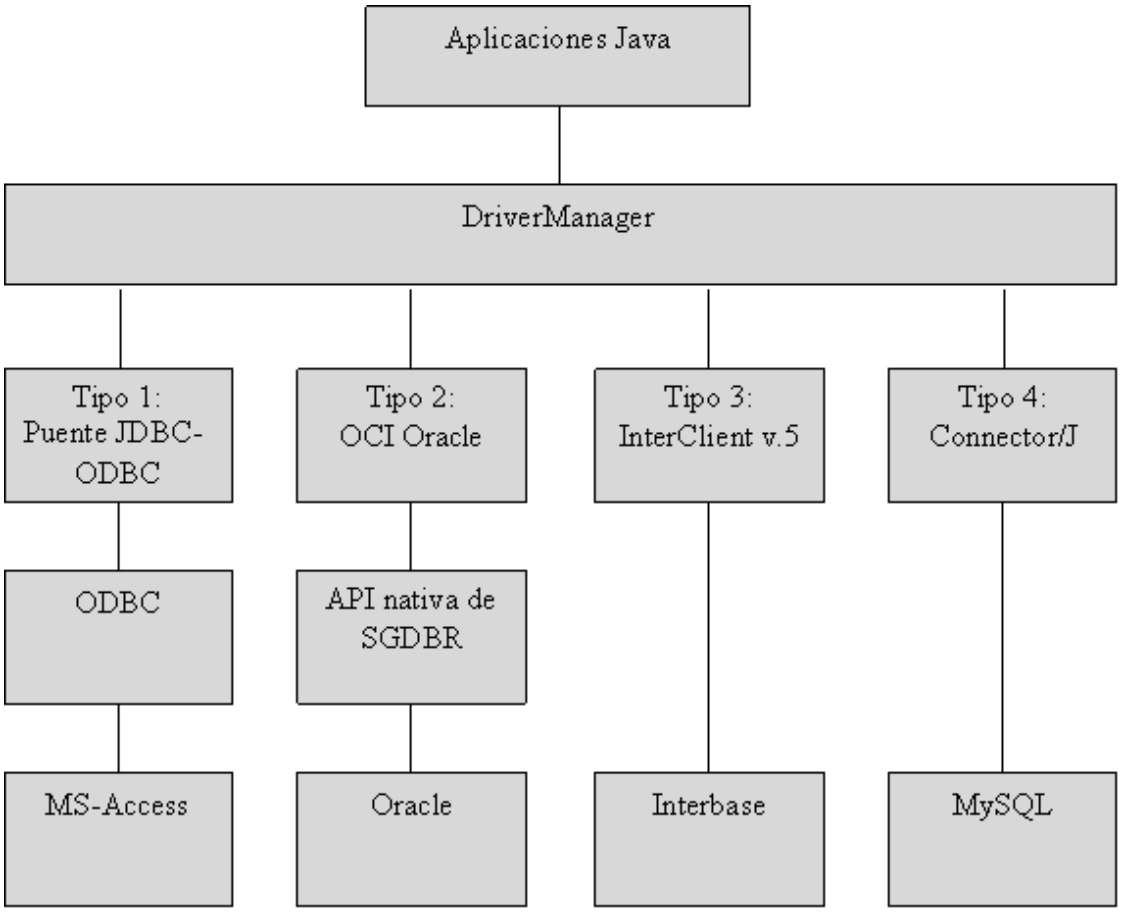
Existen varias formas para obtener una conexión de base de datos, la tradicional consiste en usar métodos estáticos definidos en la clase *java.sql.DriverManager* para establecer la comunicación con el DBMS.

Para obtener una conexión a la base de datos se utiliza regularmente la instrucción connection, método de la clase DriverManager. Generalmente se envían parámetros a esta instrucción:

- Una cadena de caracteres que representa a la base de datos, tanto el manejador de base de datos, como el nombre de la misma. Los nombres de la base de datos se conforman como URIs, y siguen la especificación: protocolo:manejador:nombre
- Una cadena de caracteres que representa al usuario de base de datos que requiere la conexión.
- Y por último, una cadena de caracteres para la contraseña del usuario a autenticar en la base de datos.

En un entorno de aplicaciones de empresa la obtención de las conexiones se hace por medio de objetos de la clase javax.sql.DataSource, los que pueden obtenerse por medio de JNDI, a través de un *lookup*.

Figura 1. Arquitectura de JDBC simple



Fuente: CLUB Developers – FAQ de Java

http://www.clubdevelopers.com/sources/filefaqs/archijdbc.gif

1.5. EJB

Los EJB tomaron un fuerte impulso a inicios de esta década, dado que representaban una solución integral para la mayoría de problemas que se presentaban en las aplicaciones grandes o empresariales. Para la parte de la persistencia su alternativa fueron los *Entity Beans* que se explicarán más adelante.

EJB es el acrónimo de *Enterprise Java Beans*, es decir, *Java Beans* para grandes aplicaciones (literalmente: aplicaciones empresariales). "EJB es un estándar para construir componentes que se despliegan en el servidor. Este estándar define un contrato entre componentes y servidores de aplicación que permite que un componente corra en un servidor de aplicaciones. Los componentes EJB son *desplegables*, lo que significa que pueden ser importados y cargados dentro de un servidor de aplicaciones, que hospeda esos componentes."[9]

"La especificación de EJB es pública y libre. Una de las ventajas de que EJB sea un estándar, es que no es necesario limitarse a productos de un vendedor en particular, sino de acuerdo a la conveniencia y factibilidad económica, entre otros factores a considerar."[10]

"Hasta el momento la arquitectura de EJB ha soportado únicamente al lenguaje Java. Aunque esto pueda parecer una gran restricción, Java es considerado uno de los lenguajes más potentes para construir componentes por varias razones, entre ellas las siguientes:

La separación entre interfaz e implementación, particularmente impulsada por Java. En las aplicaciones desarrolladas en este lenguaje es común encontrar esta práctica separación entre interfaz e implementación."[11]

"*Cross-platform*. Java corre sobre cualquier plataforma. Siendo EJB una aplicación de Java, EJB también debería correr fácilmente sobre cualquier plataforma. Esto adquiere un gran valor cuando en las empresas se ha hecho

[9] Mastering Enterprise Java, 3era. Edición página 11 (Traducción)
[10] Mastering Enterprise Java, 3era. Edición página 11 (Traducción)
[11] Mastering Enterprise Java, 3era. Edición página 12 (Traducción)

una gran inversión tanto en software como en hardware para contener las aplicaciones desarrolladas."[12]

Las aplicaciones corriendo en el lado del servidor tienen necesidades diferentes a las que tienen las aplicaciones corriendo en los clientes. Por ejemplo, los componentes de servidor requieren correr en un entorno de alta disponibilidad, tolerante a fallos, transaccional y multiusuario. Un servidor de aplicaciones es quien provee de este entorno para los EJB.

Los usos que EJB esperaba cubrir eran muy diversos, pero en esencia se esperaba que fueran utilizados para escribir la lógica que resuelve los problemas del negocio, es decir, directamente involucrarse en la solución que la aplicación brinda al negocio. "Entre las tareas típicas que EJB puede ejecutar se encuentran:

- Calcular el monto total en la aplicación de carretilla de compras, tan popular en compras desde Internet;
- Enviar una confirmación de la orden de compra (de la carretilla de compras) por e-mail, utilizando el API Java-Mail.
- Enviar una orden por libros, accediendo a la base de datos de libros.
- Transferir dinero entre dos cuentas bancarias, accediendo a las cuentas en una base de datos."[13]

1.5.1. EJB en el marco de J2EE

Los EJB constituyeron una parte medular del estándar J2EE (*Java 2 Enterprise Edition*).

[12] Mastering Enterprise Java, 3era. Edición página 13 (Traducción)
[13] Mastering Enterprise Java, 3era. Edición página 13 (Traducción Interpretada)

Como se mencionó en una sección anterior, el estándar EJB define cómo son escritos los componentes de servidor y establece reglas para indicar el tratamiento estándar entre componentes y los servidores de aplicación que los manejan.

Siendo EJB un ingrediente importante dentro del estándar J2EE se apoya en otros elementos (tecnologías y APIs) que cumplen con J2EE versión 1.4, entre ellas:

- JAX RPC (Java API for XML RPC)
- *Java Remote Method Invocation* (RMI) y RMI – IIOP
- *Java Naming and Directory Interface* (JNDI).
- *Java Database Connectivity* (JDBC)
- *Java Transaction API* (JTA) y *Java Transaction Service* (JTS)
- *Java Messagin Service* (JMS)
- *Java Servlets*
- *Java Server pages*
- Java IDL
- JavaMail
- J2EE Connector Architecture (JCA)
- El Java API for XML Parsing (JAXP)

1.5.2. Entity Beans

Los *Entity Beans* son la alternativa en aspectos de persistencia del estándar de EJB. Cuando se utiliza *Entity Bean* suele crearse una o más clases *Entity Bean* asociadas a una tabla o a un subconjunto de sus atributos. En tiempo de ejecución las tuplas de la tabla asociada al *Entity Bean* se convierten

en instancias de ese *Entity Bean* en el contexto del contenedor de EJB, de un servidor de aplicaciones.

Los *Entity Bean* han sido redefinidos en las distintas especificaciones de EJB tratando de disminuir las desventajas que presentan, sobre todo, su alto costo de implementación.

Al momento de escribir este trabajo hay gran expectativa de conocer si la especificación 3.0 de EJB presentará mejoras sustanciales en los *Entity Bean*, dado que estos han sido rediseñados por un grupo de expertos que incluyó a los creadores de Hibernate.

Las críticas sufridas por los *Entity Bean* están principalmente ligadas a los requerimientos para su puesta en producción, que suelen ser accesibles solo por grandes empresas. Otra crítica es sobre su rendimiento en la aplicación, una vez desplegados en un servidor, a lo que hay que agregar una curva de aprendizaje muy grande.

1.6. JDO

Para explicar JDO es necesario hacer una comparación con algo más conocido entre los desarrolladores: JDBC. El enfoque del API de JDBC son las bases de datos relacionales; el enfoque de JDO es la persistencia declarativa. Los desarrolladores crean objetos de datos o del negocio que serán hechos persistentes con la ayuda de una implementación concreta de JDO.

Con JDBC es posible especificar qué contenido debiera ser persistente (cadenas, enteros, fechas) y cómo debería hacerse (sentencias de SQL). Con JDO es posible definir la parte persistente de las clases. Cómo se hace esta

parte depende de la implementación de JDO elegida. JDO en sí mismo es independiente de alguna implementación particular.

Java Data Objects, JDO, provee de persistencia transparente de objetos Java en almacenamiento de datos transaccionales.

JDO usa clases Java para definir el esquema de datos, así como referencias o colecciones para navegar por los datos. JDO tiene un número muy pequeño de interfaces, por lo que el uso directo de esas interfaces por la aplicación es mínimo.

JDO permite un mapeo automático de objetos desde y hacia la base de datos.

JDO se convirtió en un estándar Java en marzo de 2002 a través de la *Java Community Process*(JSR-012). Craig Russell de Sun sirvió como líder especialista, además hubo representantes de grupos de expertos: vendedores de base de datos relacionales, vendedores de herramientas de mapeo objeto-relacional, vendedores de base de datos de objetos y otros expertos en persistencia de objetos.

Mientras JDBC usa el modelo relacional, JDO usa el modelo de objetos de Java.

JDBC soporta consultas a través de una rica gama de agregaciones provistas por SQL, aunque hay decir que todavía hay un problema con la portabilidad entre diversas implementaciones de SQL. Mientras tanto JDO provee soporte para transacciones y consultas.

Entre los beneficios en la productividad que JDO promete para los desarrolladores están:

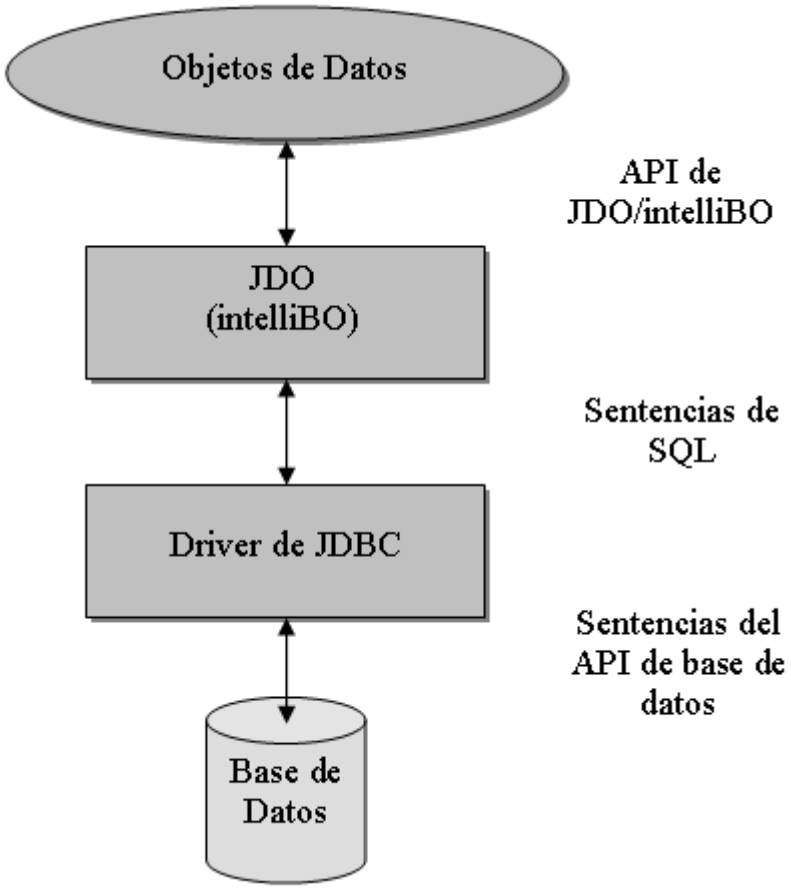
- Provee mapeo entre objetos Java y objetos de la base de datos.
- Los desarrolladores pueden mantener el enfoque solo en el modelo de objetos.
- No es necesario que los desarrolladores sean expertos en base de datos o en mapeo de modelos.

Entre los beneficios en la portabilidad:

- JDO provee compatibilidad binaria a través de base de datos relacionales, de objetos y aún de otras implementaciones.
- El lenguaje de consultas de JDO es consistente entre las distintas implementaciones.

1.6.1. Arquitectura de JDO

Figura 2. Arquitectura de JDO



Fuente: Documento jdbc2jdo - Signsoft intelliBO

2. HIBERNATE, UN PARADIGMA DISTINTO PARA LA PERSISTENCIA

2.1. Herramientas de mapeo objeto relacional

Las herramientas de mapeo objeto relacional u ORM (*Object Relational Mapping*) por sus siglas en inglés cumplen con la finalidad de permitir a los desarrolladores de software utilizar una base de datos relacional, como las mencionadas en el capítulo uno, como si se tratase de una base de datos orientada a objetos o bien objetos, típicamente utilizados en una aplicación común.

Específicamente, y simplificando el ejemplo, una tupla en la base de datos, es utilizada como si se tratase un objeto y esta transformación fila-objeto y viceversa constituye, en pocas palabras, el quehacer de un ORM.

2.1.1. Una definición de mapeo objeto relacional

Si en el desarrollo de una aplicación se opta por tratar directamente con las tablas de la base de datos y también manipular los datos por medio de *session beans* o cualquier otra manera de controlar el estado de la aplicación, se estará dejando de lado el uso de ORM, o al menos haciendo un esfuerzo que podría minimizarse con ORM.

Como se dijo en una sección anterior: las herramientas ORM permiten trabajar con objetos en memoria, y al final persistir los objetos en un medio de almacenamiento no volátil como parte de una base de datos relacional.

Cabe preguntarse si manipular directamente las tablas de la base de datos a la vez de usar session beans para la manipulación de los datos es práctico para proyectos con una alta complejidad; esa fue la razón para desarrollar Entity Beans como parte de la especificación EJB.

"El mapeo objeto-relacional (más conocido por su nombre en inglés, *Object-Relational mapping*, o sus siglas ORM y O/R *mapping*) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional. En la práctica esto crea una base de datos orientado a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo)."[14]

2.1.2. La diferencia objeto-relacional

En años recientes, la adopción de la programación en lenguaje Java ha puesto de relieve el paradigma orientado a objetos para el desarrollo de software. Los desarrolladores han tomado conciencia de los beneficios de la orientación a objetos. Sin embargo, la gran mayoría de compañías están todavía atadas a largas investigaciones sobre sistemas de bases de datos relacionales de alto costo.

La representación tabular de datos en un sistema relacional es fundamentalmente distinta a las redes de objetos usados en aplicaciones Java orientadas a objetos. Esta diferencia es la que ha sido llamada diferencia objeto/relacional.

[14] Wikipedia, http://es.wikipedia.org/wiki/Mapeo_objeto-relacional

2.1.3. Características de herramientas de mapeo objeto relacional

La principal característica de una herramienta ORM es mapear uno a uno objetos (entendiendo como objetos los componentes separables de la base de datos) de una base de datos relacional, incluyendo diferentes granularidades.

2.2. Hibernate

Como se ha mencionado anteriormente, Hibernate es una tecnología para Mapeo Objeto-Relacional, ORM. Es una tecnología *Open-Source* (de código abierto), desarrollada como proyecto del SourceForge.net. Ha habido ya un número considerable de este tipo de herramientas, por ejemplo TopLink, la cual fue subsecuentemente adoptada por Oracle.

"Gavin King ha sido el líder de este proyecto desde el inicio del mismo. Paralelamente Craig Russel y David Jordan lideraban el esfuerzo auspiciado por SUN, el proyecto JDO (*Java Data Objects*). Debido a algunos problemas técnicos y al tener que decidirse por alguno de los dos, la mayoría de los expertos en Java (específicamente en la *Java Community Process*, JCP) se inclinó por Hibernate en lugar de JDO. A primera vista, no hay mucha diferencia entre Hibernate y JDO. Aunque ambos sean parecidos en cuanto a opciones y sintaxis, la sintaxis de Hibernate es considerada más fácil de aprender."[15]

Como se mencionó en el primer capítulo, EJB3.0 es la versión más reciente de los EJB (*Enterprise Java Bean*) y está altamente influenciada por Hibernate. "Algunos expertos han considerado a EJB3.0 e Hibernate como la misma cosa. Oracle, por ejemplo, soporta EJB3.0, y siendo Oracle la principal compañía en base de datos en el mundo de J2EE, EJB3.0 parece tener un

[15] http://swik.net/itefforts/itefforts+blog/CMP-EJB+vs.+Hibernate/bmqzy (Traducción)

futuro asegurado. J2EE es una tecnología para un nivel empresarial (muy alto nivel) y, estando EJB en la esencia de las aplicaciones empresariales, por los servicios ofrecidos por un contenedor, la relevancia del emergente interés en Hibernate puede ser apreciado solo en asociación con EJB."[16]

"Las herramientas ORM han sido utilizadas algunas veces junto con *Session Beans*. El único problema ha sido que algunas de estas herramientas han empezado a ser propietarias, con las consecuentes alzas en los costos de su implementación. Pero hay muchas herramientas ORM de código abierto confiables, y aún Richard Monson Haefel (una autoridad conocida y autor de EJB) aprueba este método como una alternativa segura y productiva a los *Entity Beans*.

Por otro lado, los *Entity Beans* han tenido menos suerte. Los cambios de versiones EJB1.1, EJB2.0 y luego la EJB2.1 han significado un número de cambios en la especificación en lo relacionado a *Entity Beans*. El patrón de definir un JavaBean típico se ha convertido en un tema recurrente en Java. Lo mismo ocurría, hace algún tiempo, con *EJB Entity Beans*, Struts, *JavaServer Faces* (JSF) y ahora con Hibernate, *JavaServerFaces* y Spring. La definición de esta clase plana es muy importante en Java.

Los *Entity Beans*, por su parte, son de dos tipos CMP y BMP, de acuerdo a si la persistencia de los objetos es manejada por el contenedor (*Container Managed Persistence*, CMP), o bien manejada por el *Bean* (*Bean Managed Persitence*, BMP). Teóricamente, la especificación de EJB no dice nada acerca del método adoptado para persistir objetos para almacenamiento permanente y recuperación de información. La persistencia podría ser por simple serialización

[16] http://swik.net/itefforts/itefforts+blog/CMP-EJB+vs.+Hibernate/bmqzy (Traducción)

de objetos."[17] La base de datos podría ser tanto una orientada a objetos o bien una base de datos relacional como las mencionadas en el capítulo uno.

En una aplicación empresarial compleja, suele comenzarse el desarrollo con el modelo de la aplicación, incluyendo el modelo de datos. En lugar de tratar directamente con tablas y sus inter-relaciones, en ocasiones muy complejas, es más intuitivo tratar con clases y objetos o instancias de ellas. De esa forma se diseña un sistema, "pensando en objetos" y no en tablas. Al tratar con objetos en memoria o con objetos persistidos en un medio de almacenamiento persistente, se evitan esos problemas de complejidad, sin embargo se hace necesario tratar con la serialización de los objetos y la serialización de objetos complejos es un proceso lento.

Por otro lado, la tecnología de bases de datos relacionales tiene las ventajas de que es rápida, probada y bien conocida. El problema es, entonces, trabajar con objetos en memoria pero persistir los objetos en el disco duro como parte de una base de datos relacional.

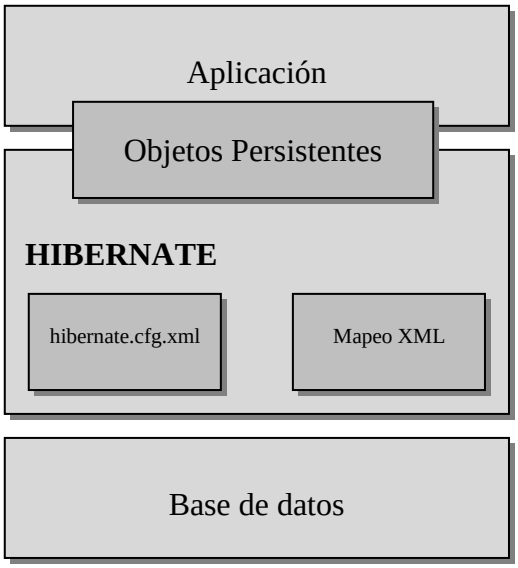
Por otro lado, la tecnología de bases de datos relacionales tiene las ventajas de que es rápida, probada y bien conocida. El problema es, entonces, trabajar con objetos en memoria pero persistir los objetos en el disco duro como parte de una base de datos relacional.

2.3. Arquitectura de Hibernate

Una vista de alto nivel de la arquitectura de Hibernate:

[17] http://swik.net/itefforts/itefforts+blog/CMP-EJB+vs.+Hibernate/bmqzy (Traducción)

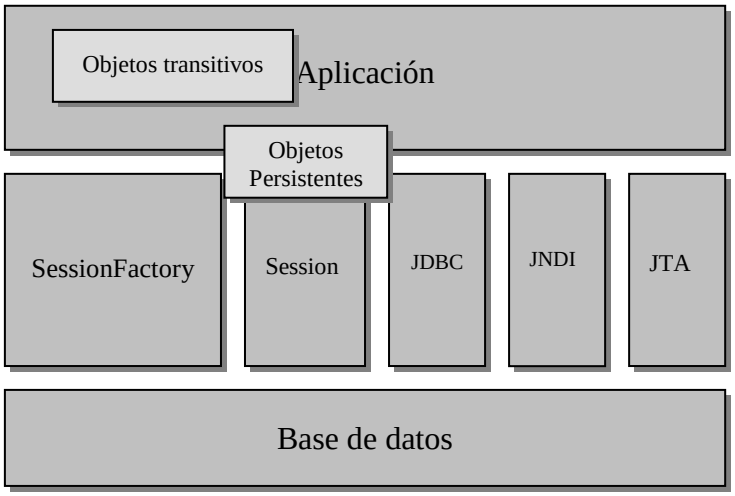
Figura 3. Arquitectura simple de Hibernate



Fuente: *Hibernate Reference Documentation*

Este diagrama muestra una arquitectura de Hibernate usando la base de datos y la configuración para proveer servicios de persistencia a la aplicación. Aunque podría mostrarse una vista mucho más detallada de la arquitectura de tiempo de corrida, pero la flexibilidad de Hibernate soporta varias opciones distintas. Por ello, se mostrarán dos arquitecturas, la considerada “lite” en la cual la aplicación maneja sus propias conexiones JDBC y maneja sus propias transacciones; en esta opción se utiliza un mínimo subconjunto de APIs de Hibernate:

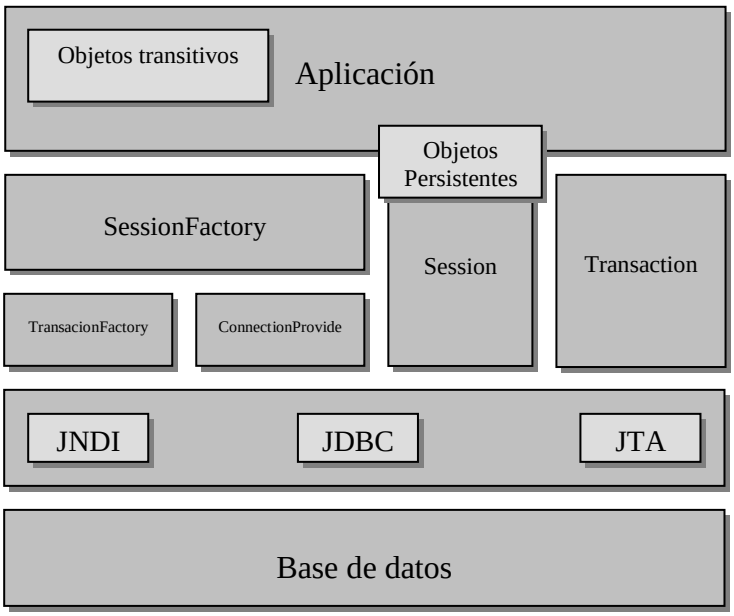
Figura 4. Arquitectura *lite* de Hibernate



Fuente: *Hibernate Reference Documentation*

En el otro extremo, una arquitectura que utiliza los APIs JDBC, JTA, y otros, lo que permite a Hibernate tomar control de los detalles, comúnmente es llamada: arquitectura “full cream”.

Figura 5. Arquitectura full cream de Hibernate



Fuente: *Hibernate Reference Documentation*

2.4. Principales APIs de Hibernate

2.4.1. API Configuration

Hibernate está diseñado para operar en muchos entornos distintos, hay un gran número de parámetros de configuración. Afortunadamente, muchos de estos tienen valores por defecto útiles en la mayoría de los casos, además Hibernate es distribuido con un archivo de configuración *hibernate.properties* que muestra las opciones disponibles. En la mayoría de los casos solo se requiere poner el archivo de ejemplo en el *classpath* de la aplicación y personalizarlo.

Una instancia de *org.hibernate.cfg.Configuration* representa un conjunto completo de mapeos entre clases Java y tablas de una base de datos compatible con SQL. Este objeto *Configuration* es usado para construir una fábrica de sesiones, *SessionFactory*.

Los mapeos son compilados desde varios archivos XML de mapeo.

Puede obtenerse una instancia de *Configuration* instanciándola directamente y especificar los documentos de mapeo. Si los archivos de mapeo están en el *classpath*, se usa *addResource()*.

2.4.2. API Session

Para comprender lo que el API *Session* de Hibernate es, es necesario olvidarse de la idea de sesión que en ámbitos asociados se tiene: sesión de base de datos, sesión HTTP, etcétera. Una sesión en Hibernate engloba varios conceptos en uno; una sesión es un objeto que se ejecuta por un solo hilo y que representa una unidad particular de trabajo con la base de datos. A lo interno una *Session* consiste de una cola de sentencias SQL que en algún momento se sincronizará con la base de datos, además la sesión contiene un conjunto de instancias persistentes de objetos que son manejadas por la misma.

2.4.3. API Criteria

Este API no forma parte del estándar de Java Persistente (JPA), está disponible únicamente en Hibernate.

La interfaz *Criteria* de Hibernate provee una manera de ejecutar consultas sin necesidad de manipulación de cadenas, como sucede con JDBC. Esta facilidad incluye también las consultas basadas en objetos usados como ejemplo (*Query By Example*).

"Un objeto *Criteria* es un árbol de instancias de otros objetos *Criterion*"[18], que serán mencionados en el siguiente apartado. "La clase *Restrictions* provee de una fábrica estática de objetos que retornan instancias de *Criterion*."[19]

[18] Java Persistence with Hibernate, Edición Revisada, Página 562 (Traducción)
[19] Java Persistence with Hibernate, Edición Revisada, Página 562 (Traducción)

Como se mencionó con anterioridad el API *Criteria* no es parte del estándar de persistencia Java. "En la práctica, *Criteria* será la extensión más común que se utiliza en una aplicación JPA."[20]

Las funcionalidades de búsquedas complejas multi-criterio son encontradas frecuentemente en aplicaciones de negocios habilitadas para la web. Las pantallas para búsquedas multi-criterio típicamente tienen un gran número de búsquedas por criterio, muchas de las cuales son opcionales.

El API *Criteria* de Hibernate provee una manera elegante de construir consultas dinámicas, al vuelo, sobre una base de datos. Usando esta técnica puede minimizarse en gran medida la cantidad de líneas para codificar una búsqueda, aunque a veces sacrificando la legibilidad de las mismas.

2.4.4. Framework Criterion

"Cuando se habla del API *Criteria*, la cosa mejor que de este puede mencionarse es el *Framework Criterion*. Este *framework* permite ser extendido por el usuario, lo que es más difícil en el caso de un lenguaje de consulta como HQL."[21]

Para una consulta realizada con *criteria*, se debe construir un objeto *Criterion* para expresar una condición o restricción. "La clase *Restrictions* provee métodos de fábrica estáticos para ir construyendo los distintos tipos de *Criterion*."[22]

[20] Java Persistence with Hibernate, Edición Revisada, Página 563 (Traducción)
[21] Java Persistence with Hibernate, Edición Revisada, Página 563 (Traducción)
[22] Java Persistence with Hibernate, Edición Revisada, Página 562 (Traducción)

2.5. HQL: el lenguaje de consulta de Hibernate

Hibernate tiene un lenguaje de consultas muy poderoso llamado HQL, *Hibernate Query Language*, (parecido al SQL); este parecido es completamente intencional, según la referencia de Hibernate[23]. HQL se utiliza en combinación con otro interfaz de Hibernate, *Query*; este interfaz permite la creación de consultas, envío de los argumentos a variables en la consulta, y ejecutar la consulta en varias formas.

Cuando se escribe una consulta utilizando HQL, esta consulta se traduce internamente a una sentencia SQL cuando se invoca a la instrucción *list()* del API *Query*.

2.6. Un ejemplo de mapeo-objeto relacional usando Hibernate

En este ejemplo se mostrará la configuración básica de Hibernate, para obtener el acceso a los datos almacenados en una base de datos relacional. La aplicación utiliza una arquitectura multicapas, se reserva una de estas capas para el acceso a los datos (*Data Access Object layer*).

2.6.1. Configuración de Hibernate

En algunos proyectos, el desarrollo de una aplicación es conducido por los desarrolladores involucrados analizando el dominio del negocio en términos orientados a objetos. En otros proyectos, el desarrollo es fuertemente influenciado por un modelo de datos relacional anterior, ya sea una base de datos heredada desde aplicaciones anteriores o un esquema diseñado por un profesional en el área. El ciclo de desarrollo cambia drásticamente de acuerdo a si hay una base de datos preexistente o no.

[23] Hibernate Reference Documentation versión 3.1.1

Una de las primeras cosas que debe hacerse al configurar Hibernate dentro de una aplicación es descargar la versión más reciente de Hibernate desde su sitio web[24]. Luego, es necesario configurar un directorio dentro del sistema donde se alojará los archivos de la aplicación, este directorio es comúnmente llamado directorio de trabajo.

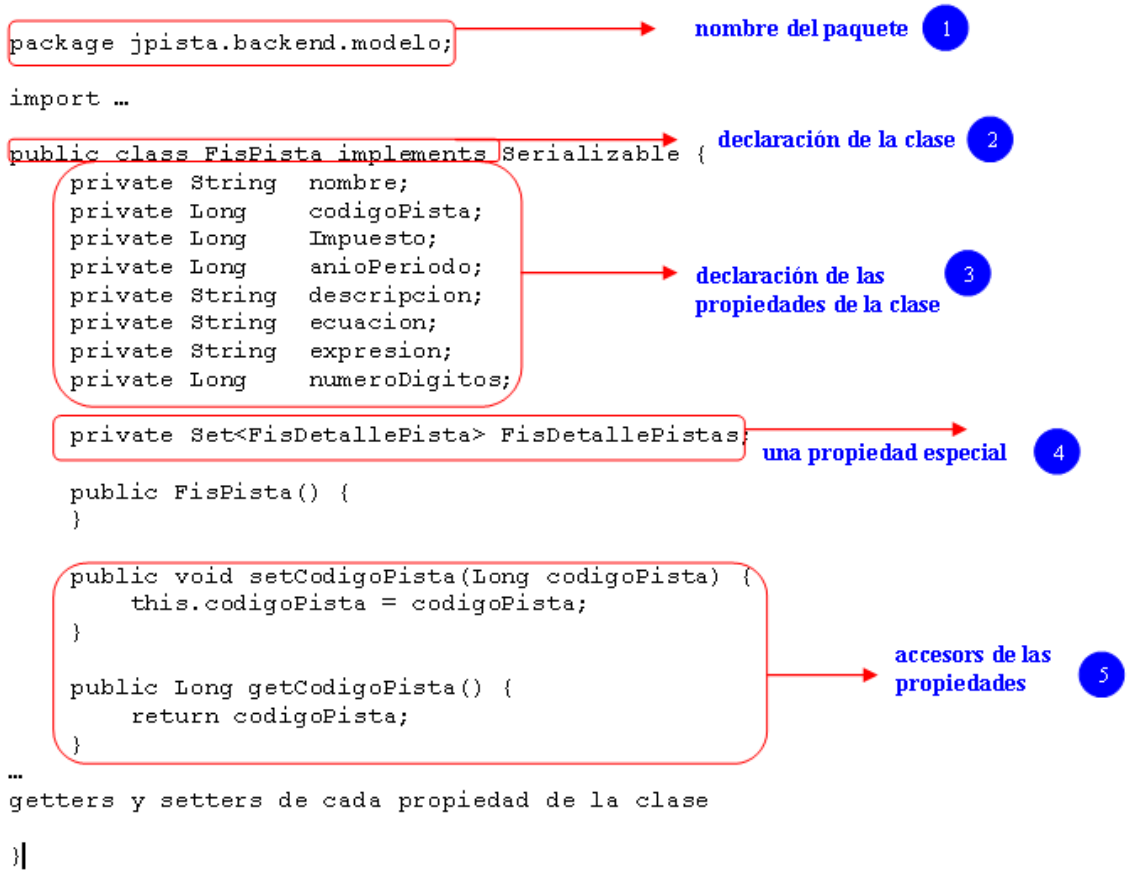
Dentro de la librería de Hibernate se encuentran otras librerías útiles para el funcionamiento de Hibernate, además de la propia distribución *hibernatev.jar* (donde *v* indica la versión de la librería); entre ellas:

- Una librería de *Ant*, utilizada para la automatización de algunas tareas rutinarias del desarrollo por medio de scripts.
- Una librería de *C3P0*, un sencillo algoritmo de código abierto, para el manejo de pool de conexiones.
- Una librería *CGLib* para la optimización de *bytecodes* generados en la compilación.
- Una librería JTA, para el manejo adecuado de transacciones, que bien puede requerir sentencias en una capa de negocio.
- Una librería *HSQLDB*, una sencilla distribución de manejador de base de datos, con fines propiamente didácticos en la construcción de prácticas, el cual puede reemplazarse con el JAR (*java archive*) del *driver* de la base de datos elegida.

[24] http://www.hibernate.org/

2.6.2. Creación de las clases planas asociadas

Figura 6. Una clase plana



La clase plana utilizada para la persistencia en Java que cumple apenas algunas características especiales:

Una clase plana, es decir, una clave que contiene en sí misma atributos y la manera de acceder a ellos (métodos de acceso).

Ser serializable, es decir, ofrecer la posibilidad de almacenar el estado del objeto con una serie de bytes; los objetos serializables pueden manipularse

con más propiedad y ofrecer la posibilidad de almacenar su estado y luego recuperarse el mismo cuando se requiera.

Como se mencionó anteriormente, es importante crear los métodos que permitan leer y escribir sobre cada uno de los atributos de manera individual (estos métodos son generalmente conocidos como *accessors*), los atributos de la clase se declaran, generalmente, de ámbito privado, por lo que la única manera de acceder a las propiedades de la clase es a través de los métodos de acceso, conservando la característica de ocultamiento de la programación orientada a objetos.

Entrando en detalle de la clase modelo, en el caso del nombre, para seguir la convención general de nombrado de clases en Java, esta es nombrada igual que la tabla, iniciando con mayúscula cada palabra en el nombre de la tabla e ignorando los signos de subrayado (*underscore*), así para la tabla FIS_PISTA el nombre de la clase modelo sería FisPista.

Los campos en la tabla son mapeados uno a uno con atributos de la clase Java, esto es, para cada campo en la tabla se crea un atributo en la clase modelo. Cada campo es mapeado de acuerdo al nombre y tipo en la tabla de base de datos.

Para el nombre de los atributos, similar a como se hizo con la clase, este se convierte en una sola palabra, iniciando con mayúscula a partir de cada palabra en el nombre de la tabla, exceptuando la primera, que inicia con minúscula, por ejemplo: para el campo CODIGO_PISTA en la tabla, el nombre del atributo en la clase será codigoPista y para el campo NOMBRE en la tabla, el atributo correspondiente será nombre.

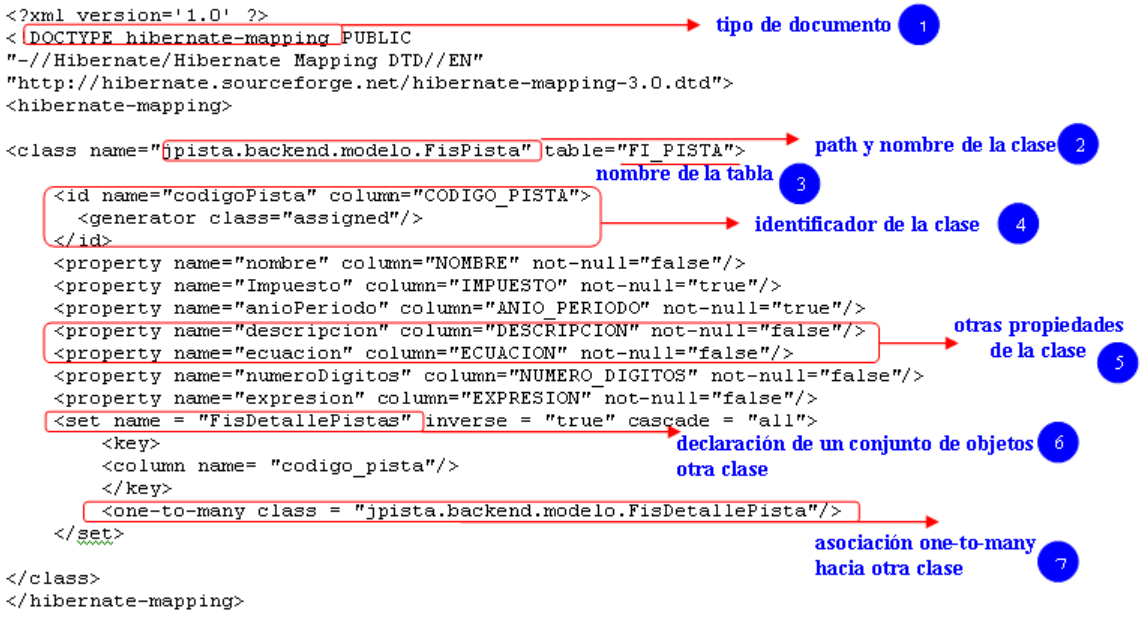
Para los tipos que se utilizarán en la clase para cada atributo se tiene una gran gama, ya que Java se caracteriza por tener un conjunto grande y especializado de tipos. Es importante, sin embargo, utilizar tipos *wrapper*, cuando sea conveniente, en lugar de tipos nativos del lenguaje, cuando se declara un atributo ya que los tipos *wrapper* son a su vez clases y poseen características deseables para las clases modelo y evitan tener que convertir constantemente el atributo de tipo nativo en una clase. En las versiones más recientes de Hibernate el componente *hbm2java*, genera tipos primitivos cuando el campo mapeado no admite nulos, pero esto ha presentado algunos inconvenientes menores. Por ejemplo, si en la tabla se tiene el campo NOMBRE del tipo VARCHAR2 (tipo de Oracle), el atributo nombre de la clase será de tipo String (java.lang.String) y el campo CODIGO_PISTA de tipo NUMBER en la tabla será mapeado como el atributo codigoPista de tipo Long (java.lang.Long).

2.6.3. Creación de la clase de mapeo

Las aplicaciones que utilizan Hibernate para la persistencia definen clases persistentes que son mapeadas a tablas de la base de datos. La definición de estas clases está basado en el análisis del negocio, previo a iniciar la codificación del sistema.

A continuación se explicará el contenido de un archivo de mapeo.

Figura 7. Un archivo de mapeo en Hibernate



Como se aprecia en la figura siete, el archivo de mapeo entre una tabla de base de datos y una clase java es un archivo XML. El nombre de la clase Java es referenciado desde el inicio, siguiendo además la convención del lenguaje para la ubicación de la clase entre los paquetes, en este caso la clase FisPista se encuentra en jpista.backend.modelo. Junto al nombre de la clase aparece el nombre de la tabla, en este caso FI_PISTA.

2.6.4. Particularidades en el mapeo de los objetos

Debido a la gran cantidad de estructuras o diseños que pueden presentarse en los modelos de datos y los resultantes modelos de objetos o modelos entidad-relación existe un gran número de mapeos cuya particularidad no permite hacer un mapeo manual sencillo, y, en ocasiones, existe más de una manera de realizar ese mapeo.

Como se verá más adelante, existen herramientas que, instalándose sobre los entornos de desarrollo, permiten realizar ingeniería inversa de un objeto o conjunto de objetos de la base de datos, asistiendo a los desarrolladores en esta minuciosa labor. Ahora bien, estas herramientas requerirán de que sean tomadas decisiones sobre lo que se desea quede como modelo de objetos.

2.6.4.1. Mapeo de relaciones entre tablas

Es común que las tablas en un modelo de datos estén relacionadas entre sí, y estas relaciones pueden ser de diferentes tipos. Uno de los factores que determina el tipo de relación entre dos tablas es la cardinalidad. De acuerdo a esta cardinalidad, la relación entre dos tablas puede ser de tres tipos:

1-1, uno a uno: a cada ocurrencia de una entidad, corresponde solo una fila de la otra y a la inversa.

1-N, uno a muchos: a cada ocurrencia de una de las entidades involucradas puede corresponderle varias filas de la otra y a ésta solo le corresponde una única ocurrencia de la primera.

N-N, muchos a muchos: a cada ocurrencia de una entidad le corresponde varias de la otra y viceversa.

3. PROPUESTA DE UNA ARQUITECTURA DE APLICACIÓN UTILIZANDO HIBERNATE

3.1. Una arquitectura de aplicación

Las aplicaciones hoy en día subsisten en un escenario muy distinto y más complejo que el de hace algunos años, por ejemplo, la necesidad de comunicación entre las aplicaciones se considera básica, mientras era un valor agregado hasta hace un tiempo.

Por otro lado, desarrollar aplicaciones desde un nivel muy bajo, utilizando lenguajes de bajo nivel, es cada vez menos común, haciendo uso de APIs en la mayoría de desarrollos, tanto como componentes ya existentes evitándose desarrollar aquellas rutinas que por su sencillez y versatilidad pueden reutilizarse o bien en algunos casos son incluidas dentro de los entornos de desarrollo de las aplicaciones.

Hay, además, una especialización de las aplicaciones tanto en su finalidad respecto a las necesidades de los usuarios finales como a lo interno, es decir, a la forma en que está construida la aplicación, desde su diseño hasta su implementación.

3.1.1. Los patrones de diseño en una arquitectura de aplicación

“Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características.

Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias."[25]

Como parte de la construcción de la arquitectura propuesta se utilizarán diferentes patrones de aplicación, a continuación se explica brevemente cada uno de ellos.

3.1.1.1. Patrón Modelo Vista Controlador MVC:

"El Modelo Vista Controlador es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario y la lógica de control en tres componentes distintos. El patrón MVC se ve frecuentemente en aplicaciones web, donde la vista es la página HTML y el código que provee de datos dinámicos a la página; el modelo es el Sistema de Gestión de Base de Datos y la Lógica de negocio; y el controlador es el responsable de recibir los eventos de entrada desde la vista."[26]

Este patrón de arquitectura es muy importante para la arquitectura que se propone, ya que la soporta desde su concepción de separación de funciones, de las cuales se ha hecho capas especializadas.

3.1.1.2. Patrón *Facade*

"El patrón de diseño *facade* o *façade* sirve para proveer de una interfaz unificada y sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas.

Facade puede hacer una biblioteca de software más fácil de usar y entender, ya que *facade* implementa métodos convenientes para tareas

[25] Wikipedia http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o
[26] Wikipedia http://es.wikipedia.org/wiki/Modelo_Vista_Controlador

comunes; puede reducir la dependencia de código externo en los trabajos internos de una biblioteca, ya que la mayoría del código lo usa *Facade*, permitiendo así más flexibilidad en el desarrollo de sistemas; y puede envolver una colección mal diseñada de APIs con un solo API bien diseñado."[27]

Este patrón se utiliza en la arquitectura propuesta al menos en dos ocasiones, en una de ellas es cuando se presentan los servicios que se prestan a la capa de presentación; es decir, la capa de negocio ofrece una gama de objetos que presentan cada uno métodos y aquellos que se definen como públicos son los que se ofrecerán a la capa de presentación y se crea una interfaz que es utilizada por las distintas instancias de clientes en la capa de presentación.

3.1.1.3. Patrón *Proxy*

"El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él."[28]

Aunque este patrón no es implementado por la arquitectura propuesta, de una manera declarativa, es importante mencionarlo dado que es de esta manera como se hace uso de una de las principales características de Hibernate, la recuperación perezosa de objetos. Esta es la manera como se recupera un objeto asociado al objeto examinado, el objeto examinado tiene un Proxy del objeto asociado, además de un método de acceso a este objeto. Si cuando se intenta acceder el objeto asociado éste no ha sido inicializado entonces se recupera desde la base de datos.

[27] Wikipedia http://es.wikipedia.org/wiki/Facade_(patrón_de_diseño)
[28] Wikipedia http://es.wikipedia.org/wiki/Proxy_(patrón_de_diseño)

3.2. Importancia de una arquitectura de aplicación en el desarrollo de aplicaciones medianas y grandes

Aunque la importancia de una arquitectura de aplicación podría no asociarse con las dimensiones de las aplicaciones, es cierto que el esfuerzo que implica el uso de una arquitectura no siempre es justificado en aplicaciones de tamaño pequeño.

En las aplicaciones medianas y grandes es determinante el uso de una arquitectura debido, entre otras cosas, a las prestaciones que de ella se esperan.

3.3. Una arquitectura multicapas

En la mayoría de aplicaciones es posible distinguir responsabilidades diversas a lo interno, es decir, el producto que los usuarios finales perciben como una interfaz, que cumple con una funcionalidad esperada, es, a lo interno, rutinas de diverso nivel; algunas de las cuales cumplen con acceder a los datos de la aplicación, mientras otras se encargan de operar esos datos con alguna lógica y otras de presentar los resultados en una interfaz de usuario, por ejemplo.

Entonces, es posible, agrupar aquellas rutinas que realicen labores de una naturaleza común, desde alguna perspectiva, en una capa, la cual abstrae una funcionalidad.

Una de las ventajas del uso de capas es que las diversas funcionalidades permanecen desacopladas, permitiendo que el cambio de alguna de ellas no implique cambios en el resto de capas.

Las capas generalmente, se acomodan verticalmente, una sobre otra, ubicándose en las capas inferiores el acceso a los datos, mientras en las capas superiores se ubica la interfaz con el usuario. Las capas suelen no saber de la existencia de la capa inferior, recibiendo servicios de la misma, pero abstrayendo los detalles de su implementación.

El esfuerzo en la implementación de una arquitectura de capas es grande y justificado solo por sus grandes prestaciones para medianas y grandes aplicaciones.

Este esfuerzo incluye a su vez un mayor esfuerzo en el mantenimiento de la aplicación, dado que un cambio en una capa “inferior” suele implicar cambios en otras capas superiores.

Es válido hacer la distinción entre distintos tipos de cambios en la aplicación. Podría esperarse que los cambios que no implican crear nuevos métodos o servicios entre capas impliquen cambios mínimos, mientras la introducción de nuevas funcionalidades sí requiera de cambios en distintas capas de la aplicación.

En este caso particular se sugiere el uso de una arquitectura de aplicación usando capas. Las ventajas de utilizar una arquitectura de capas pueden resumirse en:

3.3.1. Los cambios o mantenimiento a un segmento de código no se propagarían por todo el sistema

El uso de capas y la consecuente separación de responsabilidades en el código, hace independiente al código de una capa del código en otra, permitiendo de esa manera hacer más mantenible el código.

3.3.2. Las interfaces entre las capas se mantienen estables una vez definidas

Al usar capas se definen una vez las interfaces entre las capas y luego, si es necesario realizar cambios en la funcionalidad, estos, en muchos casos, no requerirán cambiar la interfaz, sino únicamente la lógica del código interno a la capa. Esto permite desacoplar las capas durante el desarrollo, el cual podría llevarse a cabo por equipos de especialistas en cada responsabilidad que, trabajarían únicamente conociendo las interfaces que ofrecen las capas adyacentes a la que desarrolla.

3.3.3. La reutilización de rutinas o lógica de una capa en aplicaciones posteriores

Ya que la arquitectura en capas separa eficientemente las responsabilidades en una aplicación se hace más sencillo encontrar código y hasta clases enteras cuya funcionalidad pueda reutilizarse en aplicaciones a desarrollarse en el futuro.

3.3.4. Subdivisión del trabajo

Una de las principales fortalezas de la utilización de una arquitectura de capas se basa en el antiguo principio de “divide y vencerás”; efectivamente, el trabajo de una aplicación concreta puede dividirse y de esa manera obtener

beneficios tales como una mayor especialización. Será más fácil con tareas complejas si estas pueden subdividirse.

Aunque esta arquitectura está pensada para ser resistente y robusta, esto debe aún someterse a análisis. Las principales pruebas a las que se deberá exponer la arquitectura para probar su resistencia están relacionadas con los cambios en los requerimientos técnicos. Estos cambios no deben afectar a la arquitectura, de tal forma que cambien la estructura.

Por ejemplo, un cambio en el motor de base de datos a utilizar no debe significar cuestionar la capa de acceso a datos, sino simplemente una re-configuración y adecuaciones mínimas.

Una de las principales características buscadas en esta arquitectura, ha sido la resistencia y existe más de una razón evidente para ello. La inversión que una empresa de grandes dimensiones hace en la construcción de sus aplicaciones de uso interno es muy fuerte, el esfuerzo de codificarlas involucra a muchos empleados, además se espera que estas aplicaciones resistan cambios durante mucho tiempo, tanto, en la funcionalidad concebida inicialmente, como en nuevas funcionalidades que se espera poder agregar, por cambios en los requerimientos.

Se espera que las aplicaciones tengan una vida útil larga, en ocasiones superando los cinco años; una de las razones por las que esto es deseable, es debido a otro tipo de resistencia, la de los usuarios finales, que una vez adecuados a una aplicación y su interfaz se resisten al cambio a nuevas aplicaciones, aunque estas estén mejor construidas o utilicen tecnologías que, con un mejor rendimiento, les permitan realizar sus tareas de manera más eficiente.

3.4. Capas de arquitectura con Hibernate

Las responsabilidades de las distintas capas en una aplicación pueden variar mucho de acuerdo a diversos factores, pero los criterios en los que se enunciará una propuesta de las capas a incluir se basa en la dimensión de la aplicación.

3.4.1. La capa de modelo

Esta capa bien podría incluirse dentro de la capa de acceso a datos, porque su funcionalidad está íntimamente ligada a esa capa, dicha relación puede enunciarse de la siguiente manera: “la capa de acceso a datos usa la capa de modelo”.

Esta capa diferenciará esta arquitectura de aplicación de otras arquitecturas que no utilicen mapeo objeto relacional.

El objetivo principal de esta capa consiste en crear una representación de objetos del modelo de datos asociado a la aplicación. Esto es, la capa de modelo se encarga de crear una representación de cada uno de los objetos de la base de datos (enfáticamente de las tablas), representándolos mediante clases, así mismo, son representadas las relaciones, mediante asociaciones.

3.4.1.1. Clases representando tablas

Las tablas del modelo en la base de datos suelen representarse mediante clases, generalmente el mapeo se hace uno a uno entre las clases y las tablas, aunque la variedad de configuraciones con que se cuenta es en ocasiones muy grande.

Las clases que representan objetos en la base de datos son generalmente clases planas, comúnmente llamadas POJOs (*Plain Old Java*

Object), cuya funcionalidad se limita a actualizar y recuperar valores hacia y desde la base de datos, respectivamente.

En un mapeo más fino, granularmente, los campos de las tablas son mapeados a propiedades de la clase, los cuales son accedidos por medio de métodos de acceso, llamados comúnmente *accessors*.

Una de las ventajas que brinda Hibernate, es poder utilizar tipos, tanto del lenguaje de programación, como propios, lo cual permite una mayor adecuación a los tipos utilizados por la base de datos.

En tiempo de ejecución de la aplicación las instancias de las clases pasan a representar tuplas de las tablas en la base de datos y las operaciones de manipulación de datos son automáticamente generadas por Hibernate. Es decir, una vez un objeto de la clase Maestro se asocia con una tupla de la tabla MAESTRO de la base de datos, las invocaciones a operaciones de alto nivel en la clase por ejemplo saveMaestro, se traducirá en una sentencia de SQL, por ejemplo INSERT INTO MAESTRO(ID_MAESTRO, NOMBRE, DIRECCION) VALUES (1, ‘nombre Maestro’, ‘direccion Maestro’).

3.4.1.2. Recursos de mapeo

Con este nombre se conoce a los archivos de mapeo, construidos con XML (*eXtensible Markup Language*), cada archivo contiene, a manera de *metadata*, la información de la correspondencia existente entre un campo de la base de datos y su atributo correspondiente en una clase plana de Java.

Este archivo es medular en la configuración de las herramientas de mapeo objeto-relacional y por tanto, en Hibernate, por lo que su construcción debe hacerse de forma cuidadosa.

En grandes sistemas, las distintas aplicaciones que conforman esos sistemas, pueden requerir utilizar los mismos recursos de mapeo y por ello las decisiones importantes respecto a esos recursos deben decidirse de manera de no afectar a ninguna de las aplicaciones.

Algunas de las determinaciones importantes que el recurso de mapeo contiene son:

- Los tipos de datos con que se mapeará los campos de las tablas de la base de datos
- Las asociaciones entre las clases
- Si la recuperación de las asociaciones (relaciones entre las tablas) se hace inmediata o “perezosa”.

A continuación se explicará esto a detalle.

3.4.1.3. Tipos de datos en Hibernate

Hibernate posee un gran conjunto de tipos de datos incorporados, los que permiten flexibilidad en la elección de los tipos a utilizar en una aplicación. Estos tipos de datos incorporados son integer, long, short, float, double, carácter, byte, boolean, yes_no, true_false, string, date, time_stamp, calendar, calendar_date, big_decimal, big_integer, locale, timezone, currency, class, binary, text, serializable, clob, blob.

Una de las grandes ventajas que Hibernate ofrece es la capacidad de extender el conjunto (ya bastante amplio) de tipos de datos para crear tipos de datos personalizados, que se adapten de mejor manera al negocio para el que se construye la solución de software. Uno de los ejemplos más gráficos del uso de estos tipos personalizados de datos es la creación de un tipo de dato nombre, que contenga internamente los campos nombre inicial y apellidos, en lugar de usar un String de una longitud determinada.

3.4.1.4. Asociaciones en Hibernate

El manejo de asociaciones en Hibernate constituye una de sus capacidades más útiles. Generalmente, lo que se persigue es transformar el entorno relacional de las bases de datos en uno orientado a objetos. Los objetos no poseen características relacionales como las tablas de una base de datos, sin embargo, puede emularse un comportamiento similar por medio de las asociaciones. Aún más, las asociaciones permiten abstraer la idea de bases de datos relacionales en la persistencia, de esa forma, los programadores de las capas superiores a la de acceso a datos no tendrán que tener ningún conocimiento de la base de datos, sus tablas y sus interacciones.

La manera en que esto se implementa es por medio de las colecciones que implementan los lenguajes de programación. En el caso particular de Java, las colecciones que se manejan son las listas, los *bags*, los conjuntos (*set*) y los mapas.

Al igual que en las relaciones del modelo relacional se presentan varios tipos de cardinalidad de las relaciones, como se indicó al final del capítulo anterior. La siguiente tabla ilustra las asociaciones y su correspondencia con las relaciones del modelo relacional.

Tabla I. Cardinalidad de las relaciones en Hibernate

En el modelo relacional (Relación-Cardinalidad)	En Hibernate	Parámetros en Hibernate
Uno a uno	*one-to-one*	El atributo de la clase a asociar
Uno a muchos	*one-to-many*	La colección que representa "muchos" y el atributo por el que se relacionan
Muchos a uno(No implementada)	*many-to-one*	La clase que representa a "uno"
Muchos a muchos (Implementada por 2 relaciones uno a muchos y una tabla asociativa)	*many-to-many*	Las colecciones que representan a ambos extremos "muchos"

3.4.1.5. Recuperación de las asociaciones (el atributo *lazy*)

Una de las más importantes decisiones, que incidirá profundamente en el rendimiento de la aplicación consiste en decidir si una asociación se recupera "inmediatamente" o por demanda, cuando la lógica de la aplicación requiere utilizarla.

Una recuperación inmediata se refiere a recuperar subsecuentemente las tablas asociadas a la tabla que se accede actualmente. Traducido a términos prácticos, la consulta SQL que recupera a la tabla actual dispara a su vez una consulta para la tabla asociada. Por ejemplo, si se consulta a la tabla *MAESTRO* y está tiene asociada la tabla *DETALLE* y se declara esta asociación como "no *lazy*", el acceder a la tabla *MAESTRO*, inmediatamente recuperará las tuplas de *DETALLE* asociadas; este proceso se lleva a cabo por medio de los objetos cuyas clases representan a las tablas en la base de datos.

Como se sospechará, en asociaciones en n niveles (Maestro-Detalle sucesivas) este comportamiento puede llegar a ser no deseado porque afectará directamente el rendimiento de la aplicación, colocando en memoria tantos objetos que la memoria disponible puede llegar a ser insuficiente, o afectando la transferencia de objetos entre servidores, en ambientes distribuidos.

Cuando se han hecho las consideraciones pertinentes y se ha determinado la necesidad de trabajar con recuperaciones inmediatas de las asociaciones, se contará sin embargo, con una prestación muy grande en cuanto al contenido que ya se encuentra en memoria, pues se evitará las rutinas de re-consulta a la base de datos. Es, sin embargo, como se ha dicho, una decisión importante que afectará el rendimiento de la aplicación y por ende el nivel de satisfacción de los usuarios finales, entre otros factores.

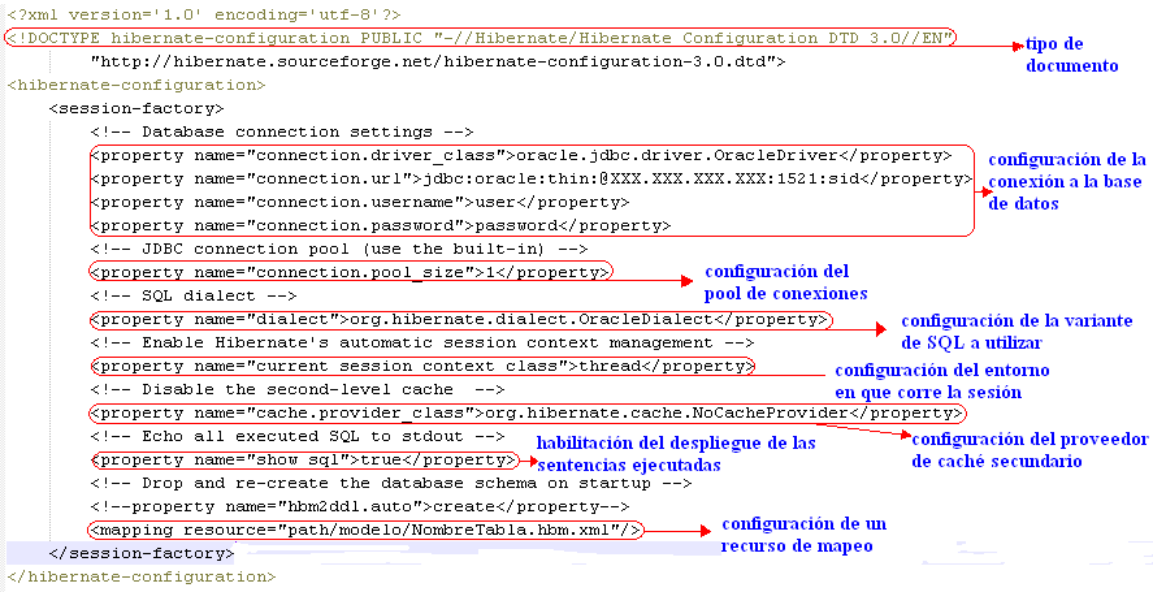
Entre las consideraciones que se realizan para tomar tan compleja decisión, está el factor entre las consultas y actualizaciones hechas sobre una tabla. Por ejemplo, en las tablas que suelen considerarse catálogos, en las que las inserciones son poco frecuentes en relación con las consultas, y cuya dimensión difícilmente alcanzará la de una tabla utilizada para transacciones, es deseable mantener todo el tiempo en memoria los objetos asociadas y ser recuperada inmediatamente que es requerida por su asociación desde otra tabla.

3.4.1.6. El archivo de configuración de Hibernate

Existen diversas maneras de enviar los parámetros de configuración de Hibernate, la más común de todas es utilizar un archivo con formato XML para la configuración. Este archivo comúnmente es llamado hibernate.cfg.xml; su ubicación acostumbrada es el directorio de archivos fuentes de la aplicación, de

esta manera se garantiza que este aparecerá en la raíz del *classpath* cuando los fuentes sean compilados.

Figura 8. Archivo de configuración de Hibernate



3.5. La capa de acceso a datos

Existen diferencias entre implementar el acceso a datos con diferentes APIs, y por ende existen diferencias al crear la capa de acceso a datos cuando esta se implementa en JDBC (*Java DataBase Connectivity*) y cuando se hace usando Hibernate.

Una de las principales ventajas de implementar la capa de acceso a datos con Hibernate es la abstracción del SQL que provee. Es decir, cuando se crea una capa de acceso a datos usando Hibernate puede prescindirse, en la mayoría de los casos, de un experto en SQL, pues operaciones del API, de alto nivel, generan automáticamente las sentencias de SQL.

Estas sentencias de Hibernate, de alto nivel, se colocan, generalmente, en una clase especial para ello, de servicios de Hibernate. Es esta clase la que se encarga, entonces, de hacer transparente las operaciones de persistencia para capas superiores en la arquitectura.

Esta clase suele implementarse utilizando un patrón *Singleton*, pues una sola clase es utilizada por los objetos de todas las clases para realizar las operaciones de persistencia. Una de las razones para utilizar este patrón es la convivencia de los objetos que representan a tuplas en la base de datos en una misma sesión de contenedor en el servidor de aplicaciones; otra razón para hacerlo es que en la mayoría de los casos la configuración inicial de persistencia no se modificará durante toda la ejecución de una aplicación.

3.6. La capa para transferencia de datos

Desacoplar unas capas de otras no es una tarea sencilla, sino se cuenta con las interfaces adecuadas, entre capas; además es necesario contar con objetos, con características deseadas, que puedan ser transportados de una capa a otra, tales como ser ligeros, serializables, empaquetables. Para ello se sugiere la utilización de una capa especial para este fin, es decir, una capa donde se alojen todas aquellas clases, cuyas instancias se transferirán entre las capas de la aplicación.

Estas clases son, generalmente, clases planas (POJOs en Java). Cuyos atributos son aquellos datos que se quieren transmitir entre capas y cuyos métodos son las formas de acceso hacia esos mismos atributos.

Se busca la adecuación de estos objetos de transferencia a las necesidades del negocio, es decir, se considera más deseable crear muchas

clases de transferencia de datos asociados a la misma tabla en la base de datos (todas con un subconjunto distinto de campos de la tabla), que transferir siempre objetos grandes de la misma clase (que poseen todos los campos de la tabla) en la que se utilicen solo unos pocos atributos a la vez.

4. ESCENARIOS DE APLICACIÓN INDICADOS PARA HIBERNATE

4.1. Criterios para la elección de una alternativa de persistencia

Los criterios para la elección de una herramienta para la persistencia dependen de los requerimientos específicos para la aplicación que se construya. Sin embargo, al referirse a aplicaciones grandes, en entornos empresariales puede esperarse encontrar que algunos de estos criterios sean comunes. Las características más comunes de los ORM se mencionaron en el capítulo 2. Algunos de estos criterios se presentan a continuación:

4.1.1. Independencia

En la mayoría de los casos se espera de una herramienta de persistencia que se desacople, tanto de las capas superiores como inferiores de la aplicación; por ejemplo, en una aplicación que tiene una capa de negocio, adyacente a la de persistencia es deseable que no existan datos, en la capa de persistencia, que sean directamente accedidos desde lógica de la capa de negocio, si no es a través de métodos provistos por la interfaz ofrecida por dicha capa.

Así mismo, es deseable que la capa de persistencia sea independiente de la base de datos. Aunque esto no es posible en un sentido literal, porque la capa de persistencia dependerá directamente de la estructura creada en una base de datos para un contexto específico.

4.1.2. Escalabilidad

Aunque esta característica es, más bien, regida por lo que la arquitectura señale, es también dependiente de la herramienta de persistencia que se utilice. Ya que está debe permitir que, a cambios en la magnitud o en la estructura de la base de datos, se haga una adaptación sencilla para escalar así mismo lo contenido en la capa de persistencia.

4.1.3. Flexibilidad

La herramienta encargada de la persistencia debe ser flexible, al permitir que los cambios estructurales en la base de datos y en los datos que se quieren persistir puedan reflejarse sin realizar cambios complejos o poco evidentes en la aplicación, esto es tanto una bondad de la arquitectura que se propone como de las herramientas que se utilicen para la persistencia de los datos.

4.1.4. Rendimiento

El rendimiento ha sido considerado como una de las características indispensables en las herramientas utilizadas para persistencia, el acceso a los datos suele ser en muchas aplicaciones el cuello de botella y llega a poner en duda la viabilidad de la puesta en marcha de una aplicación. Las herramientas de ORM han sido consideradas, inicialmente, como pesadas y demandantes de muchos recursos para su funcionamiento, pero estas han ido evolucionando y mejorando su desempeño.

4.1.5. Simplicidad

Las tareas inherentes a la persistencia de datos en sí mismas, como otras muchas, no aportan valor al proceso de un negocio, por lo mismo estas

tareas deben simplificarse y en la medida de lo posible debe dedicarse un esfuerzo mínimo en su implementación.

Se requiere, entonces, que una herramienta de persistencia brinde una manera práctica y transparente de realizar dichas tareas, es decir, es necesario proveer a los desarrolladores de herramientas que permitan simplificar la persistencia hasta el punto de convertir una, muchas veces compleja, sentencia de SQL en una sentencia más cercana a la realidad del negocio como solicitud.guardar(), en donde, en el contexto de la aplicación solicitud es un objeto de la clase Solicitud, que tiene sentido para la aplicación y que contiene un método guardar() el cual permite guardar en la base de datos la información asociada a la solicitud.

4.2. Escenarios que se adecuan para la utilización de Hibernate en el desarrollo de una aplicación

Aunque no existe un escenario contraindicado para utilizar Hibernate como herramienta de persistencia, también es cierto que hay escenarios en los que se adecúa mejor. Se presentarán a continuación algunas condiciones a considerar cuando se elige Hibernate.

4.2.1. Bases de datos heredadas

Algunos entornos integrados de desarrollo (IDEs) de Java como Eclipse contienen *plugins* o funcionalidades agregables que permiten, por ejemplo, aplicar ingeniería inversa a una base de datos existente para generar los recursos de mapeo, clases de modelo y hasta las clases para acceso a datos. Sin embargo, y muy a pesar de los excelentes resultados que se obtienen de esta automatización de generación de las capas de Modelo y de Acceso a

Datos, muchas veces es difícil y trabajosa la tarea de mapear un modelo de datos complejo en clases planas.

Por ello, en ocasiones, la utilización de un esquema de base de datos que antecede a la construcción de una aplicación con una arquitectura que utiliza Hibernate se vuelve una tarea que requiere un esfuerzo mayor a los beneficios que de esa aplicación pueden obtenerse.

4.2.2. Granularidad fina

En algunos casos, la lógica de la aplicación o alguna capa en especial, requiere el tratamiento indistinto de clases y atributos de esas clases de forma que su acceso a la persistencia requiere tratamiento especial en uno y otro caso (clases y atributos); por ejemplo, existen casos en que la cantidad de campos en una tabla, hace necesario considerar mapear subconjuntos de la tabla en clases distintas, sin embargo el API de Hibernate trabaja únicamente con clases y arreglos desordenados de objetos, lo que abre la discusión de si es o no conveniente la aplicación del API cuando en la generalidad se tratará únicamente con atributos de las clases, y no con objetos enteros como el API propone.

4.3. Un caso de estudio

Para examinar adecuadamente una arquitectura de software sería necesario someterla a la prueba de diversas implementaciones de la arquitectura para el diseño y desarrollo de aplicaciones y, yendo más allá, sería necesario que una vez desarrolladas las aplicaciones utilizando la arquitectura sugerida estas tengan un desempeño aceptable.

Llegan al departamento de desarrollo de sistemas diversas solicitudes para el desarrollo de aplicaciones que soporten las operaciones de las unidades administrativas que constituyen la institución y, de entre estas solicitudes se ha elegido algunas que enriquecerán este caso de estudio.

4.3.1. Entorno de la empresa

4.3.1.1. Naturaleza de la empresa

El caso de estudio se lleva a cabo en la Gerencia de Informática de la Superintendencia de Administración Tributaria (SAT). Como se comprenderá, la naturaleza de los problemas específicos de la institución tiene poco en común con otros negocios, sin embargo los problemas pueden generalizarse, de tal forma que pueden diseñarse soluciones de software que pueden adaptarse a varios negocios, con adaptaciones mínimas.

Simplificando, puede decirse, que la empresa (Gerencia de Informática) se dedica a realizar soluciones de software a la medida, que satisfagan las necesidades de los usuarios, siendo estos internos y externos.

Los usuarios internos, por su parte, demandan de esta gerencia soluciones de informática, aplicaciones de software, que les ayuden a automatizar actividades muy diversas, en un mayor porcentaje orientadas a la solución de problemas inherentes al negocio del área particular a la que se dedican.

Los usuarios externos se constituyen por la población en general del país, estos acceden a los servicios existentes, generalmente vía web, ofrecidos como alternativa a los métodos convencionales de realización de trámites para la tributación al Estado.

El quehacer de la Gerencia de Informática está, entonces, enfocado en la solución de los problemas de la Institución y la Institución es el ente encargado de la recaudación de impuestos en el país. Como se insinuó anteriormente, esta labor es realizada únicamente por esta Institución en el país, y por ello el quehacer del departamento de Informática es también único, por las tareas especializadas que realiza y otras veces diversificado porque trata con otras Instituciones tanto estatales como privadas, tal es el caso de los bancos.

4.3.1.2. Dimensión de la empresa

Siendo las funciones de la Superintendencia de Administración Tributaria únicas por su naturaleza, se centraliza en ella todas las operaciones relacionadas con la recaudación de impuestos del país.

Es por ello que la característica **dimensión**, de la Institución, adquirió desde sus inicios un carácter global-nacional.

La necesidad de información en esta Institución impulsa a la creación de un departamento de Informática.

La información adquiere a este nivel un carácter legal, pudiendo repercutir en implicaciones legales para los contribuyentes de cualquier categoría, es por ello indispensable la Integridad de la información que los sistemas manejan. Es necesario concentrar de alguna manera toda esa información por la interrelación existente entre los distintos tributos que un contribuyente debe cumplir.

Todas estas necesidades solo pueden ser paliadas por un robusto departamento de informática, conformado de forma que permita abarcar tanto el

mantenimiento de las aplicaciones actuales como el diseño e implantación de las nuevas, de acuerdo a las necesidades que se van presentando.

4.3.2. Situación actual del departamento de informática

Los departamentos de Informática intentan correr al ritmo que impone la cambiante tecnología, así como las necesidades de sus variados y exigentes clientes.

La Gerencia de Informática de SAT no escapa de esa visión. Mientras se mantienen las aplicaciones existentes, y se crean nuevas funcionalidades a las mismas, también se implementan nuevas aplicaciones, empujados por las crecientes necesidades de la Institución y una visión Institucional de crecimiento constante.

El departamento de informática está a su vez seccionado para atender las necesidades de acuerdo a las funciones que le son conferidas específicamente. De esta manera existe dentro del departamento un área dedicada a los desarrollos de nuevas aplicaciones y dentro de esta área se realizan las tareas propias del ciclo de desarrollo adoptado, y transversalmente existen también los roles para ejecutar las tareas mencionadas. Los roles más importantes para el desarrollo de nuevas aplicaciones son:

Arquitecto: Dedicado principalmente a las tareas de: proporcionar la arquitectura candidata para cada escenario que se presente, proporcionar los estándares de desarrollo, como estándares de codificación; y, especificar los documentos entregables en cada parte del ciclo y al final del mismo. De la misma manera, el Arquitecto es quien define la estructura de estos documentos.

Analista del Negocio: Sus tareas se suscriben a servir de enlace entre los clientes y el departamento de desarrollo; recibiendo los requisitos de los primeros, y convirtiéndolos en requerimientos que pueden ser atendidos por el área de desarrollo. El Analista del Negocio puede realizar esta tarea a partir de documentación, la cual depende de la naturaleza de la solicitud y de la solución en cuestión. Se encarga en ocasiones de generar un prototipo que acerque al cliente a la solución que finalmente será ofrecida. Además, es quien puede generar los test-case para la aplicación, pues conoce a profundidad el negocio y la utilidad que para los usuarios tiene la aplicación.

Analista/Desarrollador: Sus atribuciones están relacionadas con la interpretación de los artefactos generados por el Analista del Negocio y los estándares establecidos por el Arquitecto para construir la aplicación mediante los entornos de desarrollo y haciendo uso de los lenguajes de programación convenidos. Es decir, la tarea del analista/desarrollador es recibir los insumos de un primer análisis y diseño y a partir de esto hacer la codificación que construya la aplicación entregable.

Administrador de Proyectos: Las atribuciones de este rol son las de un gerente de primera línea, que suele mezclar su conocimiento del negocio con actividades administrativas. Debe administrar los recursos existentes de manera adecuada para obtener en presupuesto, en tiempo y con la calidad requerida las aplicaciones solicitadas.

4.3.3. El problema

La aplicación que se mostrará a continuación es una aplicación con características tanto transaccionales como de interacción con el usuario, por lo que puede considerarse que habrá diversas aplicaciones que requieran prestaciones similares.

La aplicación es para uso interno de un sector muy reducido de usuarios, dada la especialización requerida para ejercer el rol de negocio que se pretende asistir.

La aplicación se ha creado para sustituir a una aplicación pre-existente. Esta aplicación forma parte de un sistema global que ofrece una solución completa para los usuarios respecto a esta área específica del negocio. Por ello la aplicación deberá interactuar con este sistema global. Tanto la aplicación anterior como la actual son aplicaciones web, por lo que la percepción que el usuario final tendrá no deberá variar de forma drástica.

La necesidad de mantenimiento de la aplicación y de agregarle nuevas funcionalidades, conforme han ido cambiando las necesidades de los usuarios ha empujado a modificar, pensando inicialmente en hacerlo con la aplicación actual y posteriormente con una nueva aplicación.

Hubo factores que hicieron que no fuera factible el mantenimiento de la aplicación actual. Uno de los factores que no permitió hacerlo fue la falta de documentación tanto interna como externa de la aplicación. Los analistas que crearon la aplicación hicieron un producto funcional, y útil durante un período largo, sin embargo, una vez necesario el mantenimiento aparecieron grandes dificultades en mantener el código.

Aunque esta aplicación se hizo utilizando tecnologías adecuadas a cada una de las “capas”, estas “capas” no existen como tal. La aplicación fue realizada utilizando alguna metodología de desarrollo similar a *top-down*, donde el código se usa para generar, finalmente, HTML en la vista. El código está constituido por archivos que incluyen indistintamente lógica para el acceso a

datos, de tratamiento de esos datos (negocio) y de construcción de la vista de los resultados e interfaz de usuario.

Esta falta de distinción de responsabilidades del código incrementa la complejidad del mismo y dificulta su comprensión, de la misma forma agrava la dificultad del mantenimiento de la aplicación.

4.3.3.1. Problemas asociados con la falta de una arquitectura

La diversificación de las aplicaciones de software elaboradas por la Gerencia de Informática (algunas veces asociada a la distintas áreas de negocio) provocó la especialización de grupos en áreas distintas, esta sectorización trajo como consecuencia que al convenir algunas decisiones inherentes a la construcción de las aplicaciones, no se hiciera desde una perspectiva superior sino solo como decisiones grupales, con menor visión.

Los grupos de desarrollo tuvieron que tomar decisiones en cuanto a las herramientas con que se construirían las aplicaciones, así como los lenguajes en que se desarrollaría la funcionalidad de las mismas.

Las decisiones inherentes a la arquitectura de las aplicaciones se delegaron también al criterio de los analistas o programadores de cada aplicación. Esto originó una diversidad de criterios en la estructuración de las aplicaciones y se tuvieran desde aplicaciones de escritorio funcionando con dos capas hasta algunas utilizando "n" capas para propósitos similares.

El crecimiento constante al que se ha visto sometido el departamento ha obligado a reconsiderar estos criterios de una forma unificada. La dificultad en el mantenimiento de las aplicaciones ya existentes, elaboradas usando diversas

herramientas y utilizando arquitecturas muy diversas ha sido un factor muy importante para considerar la existencia de una arquitectura que abarque las consideraciones más frecuentes en las aplicaciones que se elaboran en la gerencia.

Esta dificultad en el mantenimiento ha estado asociada a varios factores, entre ellos:

- La falta de definición de los artefactos considerados entregables en la construcción de una aplicación. Esto origina que algunas aplicaciones carezcan, en su resultado final, del diseño por el cual fueron concebidas o de la documentación básica para saber qué ideas dieron origen al producto final.
- La ausencia de un patrón arquitectónico estándar para la elaboración del total de las aplicaciones, esto provoca que decisiones importantes para el funcionamiento global del sistema sean tomadas desde la perspectiva parcial de un analista y no desde una perspectiva global como la del arquitecto de las aplicaciones.

4.4. La solución propuesta

4.4.1. Suposiciones

Un alto porcentaje de las aplicaciones de software tienen un conjunto de entidades del negocio, unidades conceptuales que generalmente se traducen en entidades o tablas en una base de datos relacional.

Con fines de emitir la siguiente conclusión de este trabajo se dará por aceptado que por la frecuencia en que varía la información contenida en las tablas, éstas pueden clasificarse en dos grandes tipos:

4.4.1.1. Tablas catálogo

En ellas se almacena información relativamente menos variable que la de tablas operacionales, y generalmente son mantenidas por un administrador del sistema.

El concepto de catálogo es amplio, se utiliza para denotar desde valores casi completamente estáticos e inamovibles como el catálogo de países que puede encontrarse en una web para registrar la ubicación geográfica del usuario, el cual variará únicamente cuando se agregue un nuevo país; hasta otros que varían constantemente como el caso de un catálogo de productos en una web de venta de artículos de moda.

4.4.1.2. Tablas operacionales

Ellas son el resultado de combinaciones hechas con las tablas catálogo o bien operaciones que se realizan con los datos de esas tablas o entre tablas catálogo y operacionales. Un ejemplo de estas son los conjuntos de valores que conforman una compra y que suelen asociar un catálogo de Clientes con un catálogo de Artículos en una transacción.

4.4.2. Altas, bajas y cambios en las tablas

Cuando se habla de realizar un mantenimiento a un catálogo, generalmente se trata de realizar las operaciones de altas, bajas y cambios sobre la tabla en la base de datos, es decir, mantener actualizado el catálogo de acuerdo a las necesidades del negocio que subyace.

Como se ve, estas operaciones de mantenimiento son frecuentes en las aplicaciones y además poseen una característica de repetitividad para cada uno de los catálogos del sistema.

Lo anterior implica que las rutinas de acceso a la base de datos en el código de una aplicación pueden llegar a repetirse constantemente.

4.4.3. Rutinas reutilizables

Cuando una rutina se repite constantemente puede encontrarse un patrón o bien abstraer esa rutina de manera que la misma pueda llegar a ser reutilizable. Un conjunto de estas rutinas reutilizables puede llegar a constituir un componente sólido del que puedan esperarse grandes beneficios al construir aplicaciones semejantes tecnológicamente.

Finalmente la utilización de estos patrones de arquitectura, llega a constituir la arquitectura utilizada.

En el caso que ocupa este trabajo, las rutinas cuya funcionalidad busca abstraerse son esas que acceden a la base de datos.

El componente, parte de la arquitectura propuesta, se encarga de acceder a la base de datos, realizando las operaciones que son necesarias para dar mantenimiento a las tablas utilizadas por la aplicación.

Más específicamente, el componente se encarga de las operaciones de insertar, actualizar, eliminar y recuperar un registro o fila de la tabla, abarcando de esa manera, todas las operaciones que son necesarias realizar en la capa de acceso a datos.

4.5. Sin utilizar una arquitectura de aplicación

Cuando no se utiliza una arquitectura, estas operaciones se realizan para cada tabla particularmente, deteniéndose a hacer para cada caso, más o

menos, la misma codificación, lo que redunda en repetición de código fuente, de forma innecesaria, afectando la mantenibilidad de la aplicación e incrementando el costo de desarrollarla.

Figura 9. Recuperación de tuplas desde una clase.

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Connection;
import java.util.Hashtable;
import javax.naming.Context;

public class Clase1 {
    public TablaADAO[] metodoSeleccion(){
        Hashtable env = new Hashtable();
        int i = 0;
        Connection conn = null;
        Statement stmt = null;
        String sentenciaSQL = "select id, campo1, campo2, campo3" +
        " from tabla_A " +
        " where campo1 like 'variable1'";
        try{
            env.put(Context.INITIAL_CONTEXT_FACTORY, "com.evermind.server.rmi.RMIInitialContextFactory");
            stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
            ResultSet rset = stmt.executeQuery(sentenciaSQL);
            if (rset.next())
            {
                rset.last();
                TablaADAO[] vRegistrosTablaADAO = new TablaADAO[rset.getRow()];
                rset.first();
                while (rset.next())
                {
                    vRegistrosTablaADAO[i] = new TablaADAO();
                    vRegistrosTablaADAO[i].setId(rset.getString("id"));
                    vRegistrosTablaADAO[i].setCampo1(rset.getString("campo1"));
                    vRegistrosTablaADAO[i].setCampo2(rset.getString("campo2"));
                    vRegistrosTablaADAO[i].setCampo3(rset.getString("campo3"));
                }
                rset.close();
                stmt.close();
                return vRegistrosTablaADAO;
            }
            else
            {
                TablaADAO[] vRegistrosTablaDAO = new TablaADAO[0];
                rset.close();
                stmt.close();
                return vRegistrosTablaDAO;
            }
        }
        catch (SQLException e)
        {
            TablaADAO[] vRegistrosTablaADAO = new TablaADAO[1];
            vRegistrosTablaADAO[0].setCampo3("Excepcion de SQL - find " + e.getMessage());
            return vRegistrosTablaADAO;
        }
    }
}
```

La figura nueve muestra una clase que contiene un método para recuperar tuplas de información de una tabla específica de la base de datos.

De una manera resumida, es posible enumerar los pasos:

- Crear u obtener una conexión de la base de datos.
- Crear la sentencia que se desea ejecutar en la base de datos, en este caso una sentencia SELECT sobre cuatro campos de una tabla.
- Ejecutar la sentencia.
- Almacenar el resultado en un objeto con esa capacidad.
- Trasladar el resultado de la ejecución del objeto resultado la sentencia en objetos creados con esa finalidad.

4.6. Utilizando una arquitectura de aplicación

Al utilizar un componente que abstraiga esta funcionalidad, el desarrollador no se detendría en hacer constantemente el código que se muestra en la figura nueve, en lugar de ello utilizaría una función del componente.

Figura 10. Recuperación de tuplas desde una clase, utilizando un componente de persistencia.

```
import crud.Finder;
import crud.FinderException;
import java.rmi.RemoteException;
import java.util.ArrayList;
import java.util.Collection;

import org.apache.commons.lang.StringUtils;
import org.hibernate.criterion.Conjunction;
import org.hibernate.criterion.DetachedCriteria;
import org.hibernate.criterion.Expression;
import org.hibernate.criterion.Restrictions;

public class Clase1 {
    private Finder finder;

    public Collection metodoSeleccion(String variable1) throws FinderException {

        DetachedCriteria vCriteria = DetachedCriteria.forClass(TablaADAO.class);
        Conjunction vConjunction = Restrictions.conjunction();

        if (StringUtils.isNotEmpty(pIntercambio.getTipoDocImportador())) {
            vConjunction.add(Expression.eq("campo1", variable1));
        }
        vCriteria.add(vConjunction);
        return finder.findByCriteria(vCriteria);
    }
}
```

Como se muestra en la figura diez, el ahorro en líneas de código es sensible al utilizar un componente destinado a la recuperación de tuplas desde la base de datos (52 líneas al utilizar un API común para desarrollar cada clase y 12 líneas al utilizar la arquitectura y un componente). Puede redondearse a un 25% de líneas de código utilizando el componente de acceso a datos que se sugiere en esta arquitectura.

De la misma manera que se hace para la recuperación es posible abstraer las operaciones de insertar, actualizar y eliminar registros de una tabla de la base de datos, por medio de un componente.

4.7. Estimación del ahorro en un proyecto

Se realizará a continuación una estimación del ahorro en proyectos de desarrollo. La muestra de aplicaciones utilizada contiene características que comparten como: son aplicaciones accesibles vía web, para acceder de forma segura a información relacionada a los usuarios o clientes. Las aplicaciones se han desarrollado utilizando el lenguaje Java como base.

Se ha realizado la evaluación de tres aplicaciones con las características mencionadas.

4.7.1. Motivación del análisis

La viabilidad de la implementación de una arquitectura está influenciada por el retorno de la inversión que una empresa obtenga.

En la mayoría de los casos resulta difícil estimar el esfuerzo para desarrollar una aplicación desde la vista de los distintos roles involucrados. La estimación del tiempo de codificación de una aplicación es una tarea que comúnmente realiza el Project Manager, apoyándose en distintos aspectos incluyendo su propia experiencia y la opinión del desarrollador o desarrolladores a utilizar.

Inicialmente, se busca determinar la relación del esfuerzo para codificar una aplicación, desarrollada haciendo uso de la arquitectura de aplicación en cuestión versus el esfuerzo para codificar una aplicación desarrollada sin ella. Anteriormente ya se hizo un análisis del ahorro en líneas de código que se obtiene al utilizar un componente de acceso a datos en lugar de codificar una y otra vez esta funcionalidad.

A continuación se determinará la reducción en tiempo de desarrollo de una aplicación completa, específicamente en la etapa de codificación.

Se concluirá respecto al ahorro de tiempo, siendo que este suele traducirse a dinero al valorarse las horas que se contrata a un empleado, por ejemplo, las horas que un programador realizó tareas en una aplicación dada.

4.7.2. Metodología del análisis

En este trabajo se hará uso de una estimación pero invirtiendo el orden habitual, de forma que partiendo de información acerca de las líneas de código utilizadas en aplicaciones construidas previamente, haciendo uso de una arquitectura de aplicación, y con base en la estimación del ahorro de líneas presentado en la sección 4.6 de este capítulo, permita estimar el esfuerzo en líneas de código al no utilizar una arquitectura de aplicación.

Como se mencionó anteriormente, se cuenta con tres proyectos que han sido desarrollados siguiendo la arquitectura de aplicación propuesta y auxiliándose con un componente para la persistencia basado en Hibernate.

La manera en que se procederá se resume como sigue:

- Se cuantificará las líneas de código en cada uno de los tres proyectos.
- Se estimará las líneas que tendría cada uno de los proyectos de haberse desarrollado sin usar la arquitectura.
- Se estimará el ahorro promedio al utilizar la arquitectura y un componente para la persistencia.

4.7.3. El esfuerzo de una aplicación en líneas de código

Existen varios modelos para estimar el esfuerzo de realizar una aplicación, el tema ha sido agotado por la importancia que el tema ha tenido, sobre todo en los grandes proyectos.

La principal motivación de estos modelos es averiguar el costo del desarrollo de una aplicación, en términos de meses-persona o directamente en unidades monetarias.

El modelo de estimación de costos COCOMO I[29], que se utilizó ampliamente durante los años 80, usaba como parámetros principales el modo de desarrollo y las líneas de código de la aplicación, expresadas en miles de líneas físicas.

Aunque las dimensiones de una aplicación de software pueden expresarse en diversas formas, en esta comprobación se utilizará el número de líneas de código de la aplicación, aunque este factor depende a la vez de muchos otros.

4.7.4. Las aplicaciones a analizar

4.7.4.1. Aplicación 1

Esta es una aplicación empotrada en un sistema, cuya funcionalidad está bien delimitada y por ello posee en sí misma todas las capas de la arquitectura y hace uso del componente para persistencia de datos. La aplicación es pequeña y requirió, aproximadamente, de 320 horas de programación. Se ha hecho una estimación, basada en la apreciación de las personas involucradas y el cronograma de actividades, para ponderar las capas, generalizando al patrón

[29] COCOMO (Constructive Cost Model) Es un modelo matemático para la estimación de costos del software.

Modelo Vista Controlador; la estimación ha sido: 25% al Modelo, 45% al Negocio y 30% a la Vista.

4.7.4.2. Aplicación 2

Esta aplicación sirve una pantalla, que posee en sí misma toda una funcionalidad para el cliente. La estimación, haciendo la separación propuesta por el patrón Modelo Vista Controlador, ha sido nuevamente 25% de esfuerzo al Modelo, 45% al Negocio y 30% a la Vista.

4.7.4.3. Aplicación 3

Esta es una pequeña aplicación dentro de un proyecto mucho más grande, que alberga toda una parte del negocio. Esta aplicación aunque pequeña es muy compleja por las interacciones que forma hacia otras partes del negocio.

En este caso la estimación realizada, con el patrón Modelo Vista Controlador ha revelado: 20% del esfuerzo al Modelo, 50% al Negocio y 30% a la Vista, muy similar a las anteriores, aunque más cargada en la capa del Negocio.

4.7.5. Cantidad de líneas de código por aplicación

Existen en el mercado herramientas capaces de contar las líneas de código de una aplicación o parte de la misma, en varios lenguajes de programación. Estas herramientas suelen hacer la distinción entre líneas físicas y lógicas de código.

En este trabajo se ha utilizado la herramienta CLOC (la cual permite contar las líneas físicas, las de comentarios y las líneas en blanco) para contar las líneas de código en las aplicaciones, donde predomina el código Java.

Existen herramientas como SLOCCount que, igual a CLOC, toma como entrada el código fuente de una aplicación y usa el método de COCOMO para estimar el esfuerzo del proyecto y de allí mismo su costo estimado.

Los resultados de la aplicación de CLOC han sido los siguientes:
Para la aplicación 1:

Tabla II. Número total de líneas de código en aplicación 1

Language	*files*	*blank*	*comment*	*code*	*Scale*	*3rd gen equiv*
Java	52	1564	3177	5208	1.36	7082.88
XML	24	27	75	1141	1.9	2167.9

Para esta aplicación se muestra el resultado obtenido también para las líneas de XML, para evidenciar la gran cantidad de líneas que los IDEs generan automáticamente para la persistencia al generar los archivos de mapeo entre las tablas y las clases. Además, de estos 52 archivos, 37 son archivos planos que representan tablas en la base de datos y por ser generados automáticamente, se han descartado y dejado únicamente las clases de lógica:

Tabla III. Número de líneas de código en aplicación 1

Language	*files*	*blank*	*comment*	*code*	*Scale*	*3rd gen equiv*
Java	15	491	1136	2075	1.36	2822

Estos 15 archivos contienen la lógica de la aplicación.

Para la aplicación 2:

Tabla IV. Número de líneas de código en aplicación 2

Language	*files*	*blank*	*comment*	*code*	*scale*	*3rd gen equiv*
Java	13	223	455	1036	1.36	1377.68

De los 13 archivos que se consideraron de la aplicación, solamente uno es de una clase plana que representa a una tabla de la base de datos, los restantes 12 son de lógica propia del negocio.

Es posible hacer una aproximación de cuántas de las líneas, arrojadas por ***CLOC***, son de lógica del negocio usando proporciones.

Tabla V. Parámetros para estimación de líneas de código

	Número de archivos	**Porcentaje**
Total de clases	13	100
Clases de lógica	12	X
Clases planas	1	100-x

$$LíneasdeCo\ digodeLogi\ ca \approx TotaldeLin\ eas \times \left(ClasesdeLo\ gica \div TotaldeCla\ ses \right)$$

El total de líneas es de 1377.68

$$LíneasdeCo\ digodeLogi\ ca \approx 1377.68 \times \left(\frac{12}{13} \right)$$

El resultado es, aproximadamente, 1272 líneas de código de lógica del negocio.

Para la aplicación 3:

Tabla VI. Número de líneas de código en aplicación 3

Language	*files*	*blank*	*comment*	*code*	*scale*	*3rd gen equiv*
Java	3	329	548	1032	1.36	1403.52
XML	1	6	0	191	1.9	362.9
Total	4	335	548	1223		1766.42

En el caso de esta aplicación se agregó un archivo adicional de formato XML, que contiene consultas en lenguaje SQL que también fueron desarrolladas como parte de la lógica. Estas consultas deben codificarse sin asistencia de herramientas o *plugins* en los IDEs, por lo que las consultas en este archivo deben desarrollarse independientemente si se utiliza una arquitectura o no.

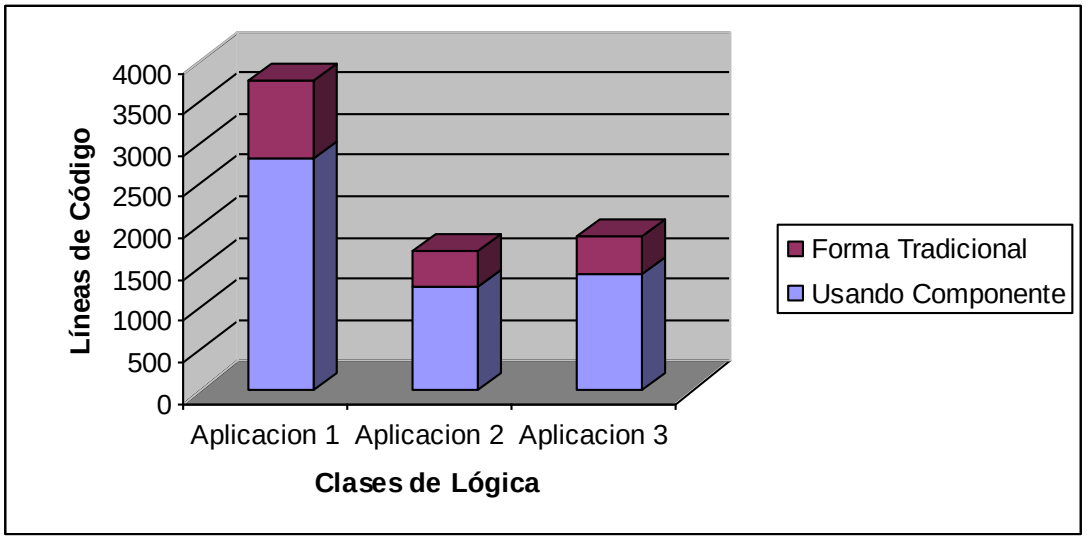
Las 1404 líneas en este caso pueden considerarse como de lógica ajena a las clases asociadas a las tablas en la base de datos.

Concluyendo, es posible afirmar que el ahorro que se estima para las líneas de código de clases de lógica es del 25 por ciento (25%), como se señaló previamente en este capítulo, lo anterior se muestra en las siguientes tabla y figura:

Tabla VII. Resumen de líneas de código usando un componente vs. forma tradicional

Aplicación	**Forma tradicional**	**Usando componente**
Aplicación 1	3763	2822
Aplicación 2	1696	1272
Aplicación 3	1872	1404

Figura 11. Número de líneas de código usando un componente versus la forma tradicional



4.7.6. Ahorro en el proyecto

El análisis se tornará ahora en cuanto al ahorro en el desarrollo de una aplicación completa, de tal manera que permita tomar la decisión de invertir en implementar una arquitectura y desarrollar componentes reutilizables.

Los insumos para este análisis son el ahorro estimado en las líneas de código para el acceso a datos, calculado previamente en la sección 4.6 y, el estimado del porcentaje de esfuerzo invertido en la capa de negocio (controlador, según el patrón MVC) en cada aplicación de las analizadas, de la sección 4.7.4.

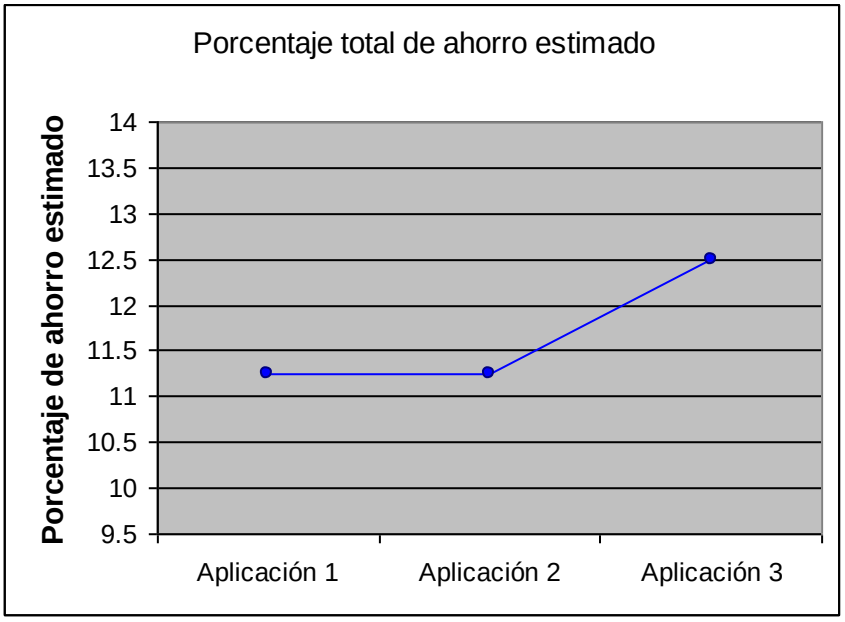
Es decir, basándose en las parejas que se forman (ahorro en líneas de código – porcentaje del esfuerzo invertido en esa capa), es posible determinar, aproximadamente, y para el tipo de aplicaciones al que se ha limitado el estudio, el porcentaje total de ahorro en esfuerzo de líneas de código. Luego se extiende a que ese porcentaje de ahorro se equivale con el de horas-

programador y finalmente, este resultado puede considerarse el ahorro en dinero que una organización paga a un empleado para elaborar la aplicación.

Tabla VIII. Porcentaje estimado de ahorro en las aplicaciones completas

Aplicación	Porcentaje de esfuerzo en clases de negocio	Porcentaje total de ahorro estimado
Aplicación 1	45	11.25
Aplicación 2	45	11.25
Aplicación 3	50	12.5

Figura 12. Porcentaje total de ahorro estimado en las aplicaciones



De acuerdo a lo que se observa en la tabla ocho y figura 12, se estima que el porcentaje de ahorro en el costo del desarrollo (codificación) del proyecto oscila entre 11.25% y 12.50%.

4.8. Análisis del retorno de la inversión

4.8.1. Costos

4.8.1.1. Contratación de personal

Este costo es potencial. Se requiere hacer un análisis profundo que desemboque en contratación de personal, debido a que suele representar un alto costo al punto de poner en riesgo la viabilidad del proyecto del diseño de una arquitectura y de los componentes que se consideren deseables para su implementación.

La potencialidad de la contratación de personal especializado, tal como un arquitecto de software está relacionada con la posibilidad de emplear capacitación para promover a algún programador o analista al rol de arquitecto.

4.8.2. Beneficios

4.8.2.1. Reducción del costo de personal

El tiempo de codificación de una aplicación disminuye al utilizar una arquitectura, y particularmente en este caso, las líneas de código empleadas en las operaciones relacionadas con la recuperación y persistencia de datos hacia la base de datos.

Esto permite llegar a una reducción en el costo del personal destinado a un proyecto.

Las horas de programador se reducen, siempre que la naturaleza de la aplicación implique operaciones a una base de datos.

La decisión que un jefe de desarrollo o administrador de proyectos puede tomar a partir de esta reducción de costos es la de reducir la cantidad de programadores al proyecto.

Siguiendo con la vista de desarrollo, para un programador será más sencillo y rápido integrar los sub-productos que componen una aplicación, y de esa manera se hará más efectiva esta tarea.

4.8.2.2. Capacidad de entregar a tiempo

La reducción del tiempo de codificación permite un ajuste en el cronograma de actividades del proyecto adelantando la fecha de entrega respecto a la misma planificación realizada considerando no utilizar una arquitectura.

Bajo la premisa de una buena planificación puede afirmarse que la utilización de una arquitectura también aumenta la capacidad de entregar a tiempo, al reducir el tiempo de codificación invertido por los programadores.

Al menos disminuye la incertidumbre relacionada con la parte de la codificación relacionada con el acceso a la base de datos, pues es más fácil determinar cuánto tiempo tomará desarrollar esa parte.

Esta capacidad en cualquier proyecto, pero sobre todo en los relacionados con desarrollo de aplicaciones de software es muy valiosa, y muchas veces los retrasos en las entregas representan incurrir en gastos o estar sujeto a sanciones cuando la aplicación se desarrolla para otra empresa.

4.8.2.3. Aumento en la productividad

Asociado a la reducción en el costo de personal, también es posible concluir que si el tiempo de codificación se reduce, con la misma cantidad de personal asociado a la programación, es factible tener un mayor volumen de producción. Es decir, el tiempo que se ahorra al utilizar un componente de la arquitectura para las operaciones con la base de datos, un programador puede dedicarlo a otras tareas, aumentando con ello la productividad.

4.8.2.4. Flexibilidad

Existen varios factores a considerar relacionados con la flexibilidad de una aplicación, específicamente, se da un beneficio cuando el esfuerzo es mínimo para realizar un cambio en una aplicación que ya se encuentra funcionando en un ambiente de producción. Los usuarios finales estarán altamente satisfechos cuando los cambios que se hacen a una aplicación no impactan las operaciones del negocio, permitiendo en todo tiempo mantener el servicio disponible.

CONCLUSIONES

1. Las herramientas de mapeo objeto-relacional permiten trabajar con una base de datos, y que el analista o programador se olvide de detalles de implementación de la base de datos, de esta manera permite enfocarse en las tareas de desarrollo haciendo uso lenguajes orientado a objetos, por lo cual desde el punto de vista de los jefes o administradores de proyectos la productividad aumenta con su utilización.

2. Hibernate es una excelente herramienta para el desarrollo de aplicaciones, haciendo uso del concepto de mapeo objeto-relacional, porque se apega fuertemente a este concepto, y además se ha integrado a los principales entornos de desarrollo de Java, tales como Eclipse, con *plugins* que permiten realizar, entre otras actividades, la ingeniería inversa, citada en la sección 2.6.3 del capítulo dos.

3. La utilización de una arquitectura de aplicación es de gran importancia cuando se crea aplicaciones grandes, generalmente empresariales y permite manejar la complejidad de estos grandes sistemas, mejorando su extensibilidad, testeabilidad (*testability*) y mantenibilidad, entre otros atributos de calidad de software.

4. El uso de una arquitectura de aplicación implica grandes costos para las empresas, y debe considerarse el retorno de la inversión que de las aplicaciones construidas a partir de esa arquitectura se obtendrán. Es decir, se hace necesario hacer el análisis de retorno de la inversión señalado,

pues de acuerdo a este, se debe o no considerar sí amerita la utilización de una arquitectura, dado el alto costo que representa. Como se mostró en la sección 4.7 el ahorro en el desarrollo de una aplicación se estima entre un 11.25% y un 12.50%.

RECOMENDACIONES

1. Las herramientas de mapeo objeto-relacional permiten trabajar con una base de datos, y que el analista o programador se olvide de detalles de implementación de la base de datos, de esta manera permite enfocarse en las tareas de desarrollo haciendo uso lenguajes orientado a objetos, por lo cual desde el punto de vista de los jefes o administradores de proyectos la productividad aumenta con su utilización.

2. Hibernate es una excelente herramienta para el desarrollo de aplicaciones, haciendo uso del concepto de mapeo objeto-relacional, porque se apega fuertemente a este concepto, y además se ha integrado a los principales entornos de desarrollo de Java, tales como Eclipse, con *plugins* que permiten realizar, entre otras actividades, la ingeniería inversa, citada en la sección 2.6.3 del capítulo dos.

3. La utilización de una arquitectura de aplicación es de gran importancia cuando se crea aplicaciones grandes, generalmente empresariales y permite manejar la complejidad de estos grandes sistemas, mejorando su extensibilidad, testeabilidad (*testability*) y mantenibilidad, entre otros atributos de calidad de software.

4. El uso de una arquitectura de aplicación implica grandes costos para las empresas, y debe considerarse el retorno de la inversión que de las aplicaciones construidas a partir de esa arquitectura se obtendrán. Es decir,

se hace necesario hacer el análisis de retorno de la inversión señalado, pues de acuerdo a este, se debe o no considerar si amerita la utilización de una arquitectura, dado el alto costo que representa. Como se mostró en la sección 4.7 el ahorro en el desarrollo de una aplicación se estima entre un 11.25% y un 12.50%.

BIBLIOGRAFÍA

1. Bauer, Christian y King, Gavin (2007). Java Persistence with Hibernate. 2a. Edición. New York: Manning Publications Co.

2. King, Gavin; Bauer, Christian; Andersen, Max Rydahl; Bernard, Emmanuel y Ebersole, Steve (2009). Hibernate Reference Documentation. Red Hat Middleware.

3. Date, C. J. (1990) Introducción a los Sistemas de Bases de Datos. 5ª. Edición. Addison-Wesley.

4. Monografías (febrero 2009)
   http://www.monografias.com/trabajos5/tipbases/tipbases.shtml

5. StarMedia – El Rincón del Vago (febrero 2009)
   http://html.rincondelvago.com/base-de-datos-relacional_1.html

6. Suárez Moreno, José Alfonso
   Chochurro (2004)
   http://www.chochurro.com/archivos/000052.html

7. Ridjanovic, Dzenan
   SWiK (febrero 2009)
   http://swik.net/persistence

8. Mi Tecnológico (abril 2009)
   http://www.mitecnologico.com/Main/LaArquitecturaDe41Vistas

9. Dolado Cosín, José Javier - Universidad del País Vasco (septiembre 2009)
   http://www.sc.ehu.es/jiwdocoj/mmis/externas.htm

10. Proyectos de Fin de Carrera (2004)
    http://libresoft.es/grex/pfcs

11. Wikipedia (septiembre 2009)
    http://es.wikipedia.org/wiki/COCOMO

12. Método COCOMO I para el cálculo del esfuerzo (septiembre 2009)
http://www.oei.eui.upm.es/Asignaturas/PInformaticos/ficheros/software/opcion3/CBasico.html

13. Análisis y Diseño de Sistemas (septiembre 2009)
http://www.galeon.com/jl_manuales/tutores/aydes/estimaciones.htm

14. Indicadores Económicos del Desarrollo del Software (2007)
http://www.kybeleconsulting.com/index.php?option=com_content&view=article&id=62&Itemid=61

15. Costes del desarrollo de software – TuFuncion (septiembre 2009)
http://www.tufuncion.com/desarrollo-software