Universidad Autónoma de Querétaro

Facultad de Informática

Memorias de Trabajo

Desarrollo de Software Java y Scripting INFORMIX-4GL

Alberto Vázquez Ramírez

Generación 2002 - 2008

La presente obra está bajo la licencia:
https://creativecommons.org/licenses/by-nc-nd/4.0/deed.es

CC BY-NC-ND 4.0 DEED

Atribución-NoComercial-SinDerivadas 4.0 Internacional

Usted es libre de:

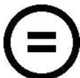
Compartir — copiar y redistribuir el material en cualquier medio o formato

La licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

Atribución — Usted debe dar crédito de manera adecuada , brindar un enlace a la licencia, e indicar si se han realizado cambios . Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciante.

NoComercial — Usted no puede hacer uso del material con propósitos comerciales .



SinDerivadas — Si remezcla, transforma o crea a partir del material, no podrá distribuir el material modificado.

No hay restricciones adicionales — No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia.

Avisos:

No tiene que cumplir con la licencia para elementos del material en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable.

No se dan garantías. La licencia podría no darle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

Introducción 5
1. Introducción a los patrones de diseño 10
1.1 ¿Qué es J2EE (Plataforma Edición Empresarial Java)? 10
1.2 Definición de patrón 10
1.3 Características comunes de los patrones: 10
1.4 Categorización de patrones. 10
1.5 Beneficios de usar patrones 11
1.5.1 Emplear una solución probada 11
1.5.2 Vocabulario común 11
1.5.3 Delimitar la solución 11
1.6 Identificando un patrón 11
1.7 Patrones contra estrategias 12
1.8 El enfoque de las capas 12
1. 9 Capa del cliente (Client Tier) 13
1.10 Capa de presentación (Presentation Tier) 13
1.11 Capa de negocio (Business Tier) 13
1.12 Capa de integración (Integration Tier) 13
1.13 Capa de recursos (Resource Tier) 13
2. El Modelo Vista Controlador 13
2.1 Contexto 13
2.2 El problema 14
2.3 Las obligaciones 14
2.4 La solución 15
2.5 Estructura 15
2.6 Participantes y responsabilidades 15
3 Instalación de Eclipse IDE 16
3.1 Descargas necesarias 16
3.2 Configuración de Tomcat 18
3.3 Creación de un Proyecto Web Dinámico 19
4. Modelo de arquitectura de 4 capas 22
4.1 Capa de presentación 22
4.1.1 Servicios de Aplicación 23
4.1.2 Direct Web Remoting (DWR) 23
4.1.2.1 Acerca de RPC 23
4.1.2.2 Resumen de la arquitectura DWR 24
4.1.3 Preparando el ambiente DWR 25
4.1.4 Ejemplo de actualización de texto dinámicamente 26
4.1.5 Resumen del ejemplo presentado. 28
4.1.6 Ext JS 29
4.1.6.1 Componentes de Ext (widgets) 29
4.1.6.2 Plantilla HTML básica para agregar componentes Ext JS 30
4.1.6.3 Widgets Ext JS 30
4.2 Capa de Servicio o Negocio 38
4.2.1 Mapeo Relacional de Objetos con HIBERNATE (ORM-Object Relational Mapping) 39
4.2.2 Introducción a HIBERNATE 40
4.2.3 Estados de las instancias 42
4.2.4 Cómo funciona Hibernate 42
4.2.5 La session Hibernate 43
4.2.6 La fábrica de la sesión 43
4.2.7 El mapeo de objetos con Hibernate 43
4.2.8 El ciclo de vida de la persistencia 44
4.2.9 El funcionamiento del enfoque POJO (Objeto Java Plano) 45
4.2.10 Obtener Hibernate 45
4.2.11 Configuración del ambiente POJO 45
4.2.12 Configuración en Eclipse 47
4.2.13 Ejemplo de un mapeo con Hibernate 47

4.2.13.1 Código POJO Direccion ... 47
4.2.13.2 Mapeo de una clase con el archivo HBM.XML ... 49
4.2.13.3 Creación de la tabla en MySQL ... 50
4.2.13.4 Configuración de la base de datos MySQL ... 51
4.2.13.5 Generación de la Llave Primaria ... 51
4.2.13.6 Guardar un objeto ... 52
4.2.13.7 Prueba simple de Hibernate para guardar un objeto en la base de datos ... 54
4.2.13.8 El archivo de configuración hibernate.cfg.xml ... 56
4.2.14 Mapeo Hibernate para relaciones entre tablas ... 58
4.2.14.1 POJO para la entidad CONFERENCIA ... 59
4.2.14.2 Mapeo Uno-A-Muchos usando una colección de tipo Set ... 61
4.2.14.3 POJO para la clase Tema. ... 62
4.2.14.4 Configuración HBM para la entidad Tema ... 63
4.2.14.5 Mapeo Mucho-A-Uno ... 64
4.2.14.6 POJO para la entidad Sede ... 64
4.2.14.7 Configuración HBM para la entidad Sede ... 65
4.2.14.8 Prueba Java para comprobar el mapeo. ... 66
4.2.15 El lenguaje Hibernate de Sentencias SQL (Hibernate Query Language) ... 68
4.2.15.1 Ejecutando consultas ... 69
4.2.15.2 Las interfaces de Query ... 69
4.2.15.3 Paginando los resultados ... 69
4.2.15.4 Uso de consultas nombradas (named queries) ... 70
4.2.15.5 Usando alias ... 70
4.2.15.6 Operadores de comparación ... 71
4.2.15.7 Búsqueda de patrones ... 71
4.2.15.8 Usando inicialización dinámica ... 71
4.2.15.9 Llamadas a funciones SQL ... 72
4.3 Capa de Persistencia ... 72
4.3.1 Manejo de la sesión en Hibernate ... 73
4.4 El Framework Spring ... 73
4.4.1 Inyección de dependencias ... 74
4.4.2 Arquitectura de Spring ... 75
4.4.3 Spring Core (Núcleo) ... 75
4.4.4 Bean Factory (Fábrica de beans) ... 75
4.4.5 Spring Context (Contexto Spring) ... 76
4.4.6 Application Context (Contexto de la aplicación) ... 76
4.4.7 Spring AOP (Programación Orientada a Aspectos) ... 77
4.4.8 Spring ORM (Mapeo de Objetos Relacional) ... 77
4.4.9 Spring DAO (Objeto de Acceso a Datos) ... 78
4.4.10 Spring Web ... 78
4.4.11 Spring Web MVC ... 78
4.4.12 Instalación y configuración de Spring ... 79
4.4.13 Trabajando con la fábrica de Spring Bean ... 79
4.4.14 Interfaz para el servicio ... 79
4.4.15 Implementación del servicio ... 79
4.5 El Patrón de diseño DAO (Data Access Object - Objeto de Acceso a Datos) ... 81
4.5.1 Características del diseño DAO ... 81
4.5.2 Archivos de configuración Spring y Hibernate ... 82
4.5.3 Archivo de configuracón applicationContext-jdbc.xml ... 82
4.5.4 Archivo jdbc.properties ... 83
4.5.5 Archivo applicationContext-resources.xml ... 83
4.5.6 Archivo applicationContext-hibernate.xml ... 84
4.6 Implementación del Patrón DAO con Spring y Hibernate ... 86
4.6.1 Clase Base Abstracta DAO ... 86
4.6.2 Ejemplo cómo crear un DAO ... 87
4.6.3 Interfaz para crear un DAO de la entidad Conferencia ... 87

4.6.4 Implementación del DAO 87
4.6.5 Clase Negocio para probar el DAO Conferencia 88
5 Pruebas unitarias con el framework JUnit 89
5.1 Características principales 89
5.2 Estructura de una prueba unitaria 90
5.3 Preparación del contexto 90
5.4 Invocación de funcionalidad a probar 90
5.5 Comprobación de resultados obtenidos 90
5.6 Restablecimiento de contexto 90
5.7 Introducción a JUnit 91
5.7.1 Elementos básicos de JUnit 91
5.7.2 Las validaciones Assert 91
5.7.3 Descargar e instalar JUnit 92
5.7.4 Ejemplo estructurado de una prueba unitaria 92
5.7.5 Diferencia entre falla y error (fail, error) 95
6 Introducción a INFORMIX-4GL 95
6.1 Qué son los lenguajes de 4ª generación 95
6.2 Estructura de un archivo 4gl para ser compilado 95
6.3 Principales sentencias del lenguaje 4gl 96
6.3.1 Sentencia DATABASE *nombre-baseDatos* 96
6.3.2 Sentencia DEFINE lista-de-variables tipo-de-dato 97
6.3.3 Tipos de dato 97
6.3.4 Sentencia LET 97
6.3.5 Definiendo un RECORD 98
6.3.6 Usando un RECORD en una sentencia SELECT 98
6.3.7 Definiendo un CURSOR 98
6.3.8 Sentencia FOREACH 99
6.3.10 Valores de retorno de una función 99
6.4 Compilación de un archivo 4gl 100
Conclusiones 102
Glosario 104
Fuentes bibliográficas 105

Introducción

El contenido de las siguientes memorias de trabajo se basa en conceptos generales de desarrollo de software y cómo construir en el lenguaje de programación Java. Adicionalmente se presentará un tema básico acerca de la programación en el lenguaje 4gl del manejador de base de datos Informix. El objetivo es mostrar las principales tecnologías, métodos de diseño y arquitectura de software con las que tuve contacto al trabajar en una fábrica de software. Es objetivo de igual manera introducir al lector a los temas de estudio necesarios para adquirir un contexto de la creación de aplicaciones de software empresarial. Va dirigido a desarrolladores y estudiantes que elaboran software en Java, se requiere de conocimientos básicos en programación orientada a objetos conceptos tales como; clase, objeto, variables de instancia, métodos, interfases, polimorfismo, herencia y cierto manejo básico del lenguaje de programación Java.

En el capítulo 1 se introduce de manera general el concepto de Patrón de diseño, y el modelo arquitectónico de capas. Lo que ha surgido a raíz del desarrollo de software empresarial a lo largo de los últimos años, es la implementación de modelos y patrones de diseño estructural de aplicaciones. Es la suma de las mejores prácticas y soluciones comunes para resolver problemas específicos en el ámbito de la ingeniería de software. Tales conocimientos se presentan a la comunidad como patrones estandarizados para la resolución de un problema en particular. Exactamente se demuestra en el Modelo-Vista-Controlador implementado hace varios años pero que es de gran aceptación y está generalizado hasta nuestros días. Se atribuye la invención del Modelo-Vista-Controlador a Trygve Reenskaug durante la década de los 70, quien trabajaba en Xerox Parc en ese tiempo. La idea básica es dividir una aplicación en tres partes:

Un modelo que mantenga la parte del estado actual de la aplicación. En este trabajo el estado de un modelo se representa de un conjunto de propiedades. Una propiedad se implementa como un método getter y setter, como en el concepto de JavaBean. ***Una vista*** que está por cualquier razón, interesada en conocer el estado actual de la aplicación. Una vista se registra con un modelo de escuchador. En el momento que cambia el estado del *modelo*, el modelo notifica a sus escuchadores registrados. En una interfaz de usuario, una vista comúnmente no hace más que desplegar el estado del modelo a través de un componente gráfico. ***Un controlador***, que cambia las propiedades del *modelo*. Cuándo y cómo cambia el controlador una propiedad es a la discreción del controlador. En programación podemos decir que un controlador es comúnmente algo que el usuario toma como entrada, por ejemplo, un botón o un campo de texto.

En el capítulo 2 se expone el modelo de arquitectura de software con el que trabajé, es básicamente una adecuación del modelo MVC por eso la importancia de presentarlo en el capítulo previo. Se basa en tres capas: capa de presentación, capa de negocio, capa de persistencia. La capa de presentación que no es más que la vista en el modelo MVC, es acerca del framework Ext JS. Una aplicación Web de internet enriquecida es diseñada para proporcionar las mismas características y funcionalidad de una aplicación de tipo escritorio. Generalmente este tipo de aplicaciones dividen el procesamiento de la aplicación por medio de la colocación de las interfaces de usuario, sus usos y capacidades del lado del cliente y las operaciones o manipulaciones de los datos del lado del servidor de aplicaciones. Ext JS, contiene la característica de crear interfaces muy dinámicas y de fácil mantenimiento debido a que ya cuenta con una gama de controles sobre Javascript que dan la apariencia de una aplicación Cliente/Servidor pero contenida en un browser.

También para facilitar el desarrollo con Ext JS se establece el uso de DWR, framework cuya característica principal es la exposición de los componentes de servicios desarrollados en Java y que radican en el servidor de aplicaciones, como objetos que puede ser accedidos por Ext JS como Javascript nativo. Como subtema del capítulo 2 se expone el enfoque de la capa de negocio en esta capa los servicios de negocio se muestran a través de Interfaces e Implementaciones (POJOs) con la ayuda de Spring, de esta forma el cliente de los servicios no conoce la implementación e inclusive no conoce si el servicio es local o remoto a su contexto solo conoce un contrato (Interfaz) que debe de cumplir la implementación, esto permite desacoplar esta capa con las demás capas así como también para que sean consumidos de manera externa y ser mas ágiles en el mantenimiento sin que se vean afectadas las demás capas por aplicar nuevas fórmulas o reglas de negocio. Algo que cabe remarcar es que para el completo éxito de esta capa en su implementación a POJOs se debe de contar con un buen modelo de negocio con orientación a objetos. Como parte del capítulo 2 se desarrolla el tema de la capa de persistencia la cual es accedida mediante el patrón de diseño Data Access Object o DAO. Los DAO son inyectados mediante Spring a los servicios de negocio y aplicación que lo requieran. Cada entidad persistente debe contar con un DAO, quien encapsula el comportamiento de persistencia de cada entidad. Cada DAO heredará de una clase abstracta que ya implementa las operaciones generales de persistencia, siendo necesario definir nuevas operaciones para casos específicos de cada entidad. En el capítulo 3 se trata el tema de Hibernate para el manejo de sesiones. El manejo de la sesión de Hibernate debe de ser transparente para el desarrollador, es decir que el desarrollador no tiene que preocuparse por abrir o cerrar la sesión en el código. Esto se logra mediante un filtro en la capa de presentación que abre una sesión de Hibernate nueva por cada petición del cliente (patrón de diseño Open Session in View). Los DAO de

la aplicación, al extender de un DAO base abstracto y mediante configuración de Spring que le inyecta una referencia al SessionFactory de Hibernate, no tienen que hace más que invocar al método getSession() de la clase padre.

A pesar de abrir la sesión de Hibernate desde la capa de presentación, esto no representa ningún problema desde el punto de vista de recursos pues Hibernate no obtiene una conexión a la base de datos sino hasta que se hace uso de la sesión. Esto quiere decir que la conexión a la base de datos se obtendrá del pool de conexiones hasta que se realice una petición a la capa de persistencia, y si alguna petición ni siquiera requiere acceder a la capa de persistencia nunca se hará una petición a una conexión de base de datos. En el capítulo 4 se aborda el tema de Spring. La inyección de dependencias consiste en proporcionar a los objetos las referencias a otros objetos de los cuales depende su funcionamiento sin que el objeto dependiente sepa cómo se obtiene dicha referencia. El modelo de inyección de dependencias tiene las siguientes ventajas: desacoplamiento de componentes, desacoplamiento de tecnologías (por ejemplo el objeto dependiente no sabe si la referencia es un EJB o un POJO). Código más limpio con menos código de infraestructura, facilita el desarrollo de pruebas unitarias, permite gran flexibilidad al permitir agregar nueva funcionalidad mediante aspectos, evitar que los desarrolladores de código repetitivo. La arquitectura se basa en el framework llamado Spring para el manejo de inyección de dependencias. Esto no quiere decir que todas las relaciones entre objetos deban de ser establecidas mediante inyección de dependencias. Los componentes que deben de ser inyectados son aquellos que atraviesan capas de la arquitectura, por ejemplo, la relación entre un BusinessService y un DAO.

El capítulo 5 es el tema de las pruebas unitarias, algunos desarrolladores sienten que las pruebas automatizadas son una parte esencial del proceso de construcción. Un componente

no puede considerarse probado hasta que haya pasado una serie de pruebas. Una prueba unitaria examina el comportamiento de una unidad de trabajo distinta. En una aplicación Java, “una unidad de trabajo es usualmente (pero no siempre) un simple método. Probar aisladamente los métodos de una clase asegura el buen funcionamiento en general de la clase pero lo más importante es asegurar una correcta interacción de todas las clases a nivel macro de una aplicación. El objetivo de este tema es demostrar con ejemplos básicos la creación de pruebas a métodos de una clase, utilizando el framework Junit. Por último en el capítulo 5 se verá el lenguaje de programación 4gl de Informix, Informix desarrolló 4GL (Fourth-Generation Application Development Language) para diseñadores de bases de datos que buscan crear aplicaciones de administración de base de datos hechas a la medida. 4GL ofrece grandes ventajas al llevar a cabo voluminosas tareas de procesamiento lógico en el servidor (al contrario de los componentes de procedimiento simples almacenados) en un lenguaje enriquecido y depurable que promueve la eficiencia del programador. Las principales características son: ofrece un elevado rendimiento en el entorno de producción, integra toda la funcionalidad necesaria para crear incluso las aplicaciones más complejas, no requiere el uso de ningún lenguaje de tercera generación, permite un mantenimiento fácil de las aplicaciones, basado en el lenguaje SQL estándar. El objetivo del tema será conocer las características de 4GL para realizar scripts en un lenguaje estructurado.

1. Introducción a los patrones de diseño

1.1 ¿Qué es J2EE (Plataforma Edición Empresarial Java)?

J2EE es una plataforma para desarrollar aplicaciones de software distribuido empresarial. La plataforma J2EE ofrece numerosas ventajas en el ámbito empresarial.

- J2EE establece estándares necesarios para las áreas de cómputo empresarial tales como, conectividad de base de datos, componentes de negocio empresarial, software intermediario orientado a mensajes (MOM), componentes Web, protocolos de comunicación e interoperabilidad.
- J2EE promueve las mejores implementaciones basadas en estándares abiertos, protege contra la evolución tecnológica.
- J2EE provee una plataforma estándar para construir componentes de software que sean portables a través de las implementaciones de los proveedores.
- J2EE incrementa la productividad de los programadores, ya que es relativamente fácil aprender las tecnologías J2EE que están basadas en el lenguaje de programación Java.
- J2EE promueve la interoperabilidad entre ambientes heterogéneos.

1.2 Definición de patrón

Los patrones se refieren a la comunicación de los problemas y sus soluciones. Los patrones nos permiten documentar acerca de un problema recurrente y su solución en un contexto en particular, y comunicar este conocimiento a los demás. El objetivo del patrón es la reusabilidad.
Cada patrón es una regla tripartita, la cual expresa una relación entre un cierto contexto, un sistema de fuerzas que ocurren repetidamente en dicho contexto, y una configuración de software que permite que esas fuerzas se resuelvan.

1.3 Características comunes de los patrones:

- Los patrones son observables a través de la experiencia.
- Los patrones están escritos típicamente en un formato estructurado.
- Los patrones previenen buscar el hilo negro.
- Los patrones existen en diferentes niveles de abstracción.
- Los patrones son artefactos reutilizables.
- Los patrones comunican los diseños y mejores prácticas.
- Los patrones pueden ser usados en conjunto para resolver grandes problemas.

1.4 Categorización de patrones.

J2EE clasifica los patrones en las siguientes tres capas arquitectónicas lógicas:
Capa de presentación.
Capa de negocio.
Capa de integración.

Patrones en el catálogo de patrones J2EE

Capa	Nombre del patrón
Presentación	Front Controller
Negocio	Business
Integración	DAO (Data Access Object)

Tabla 1.4

1.5 Beneficios de usar patrones

1.5.1 Emplear una solución probada

Un patrón es un documento basado en el hecho de que la solución ofrecida ha sido usada una y otra vez para resolver problemas similares en diferentes proyectos. De este modo, los patrones proveen un mecanismo poderoso para reutilizar, lo cual ayuda a los desarrolladores y arquitectos a evitar descubrir el hilo negro.

1.5.2 Vocabulario común

Los patrones proveen diseños de software con un vocabulario común, esto sirve para duplicar satisfactoriamente los diseños y además ayuda a transmitir un formato común entre los desarrolladores.

Un diseñador que no confía en la necesidad de un patrón, invierte más esfuerzo en comunicar su diseño a otros diseñadores y desarrolladores.

1.5.3 Delimitar la solución

La aplicación del patrón emplea un componente de diseño – los delimitadores. Usar un patrón delimitador o establecer los alcances de la solución de tal manera que sugiera al desarrollador los alcances de la implementación. Salirse de este contexto rompe con la apropiada adherencia del patrón y el diseño, lo cual desencadena una mala práctica de desarrollo.

1.6 Identificando un patrón

Cuando vemos que un problema y una solución determinada son recurrentes, se trata de identificar y documentar sus características usando una plantilla. En primera instancia, consideramos esos documentos como primer candidato de patrón a seguir, no se agregan al catálogo de patrones hasta que se observa y documenta su uso en múltiples proyectos. Como parte de validación de proceso, se usa una regla de tres. De acuerdo con esta regla, una solución es candidata como patrón de diseño hasta que ha sido verificada al menos en tres sistemas diferentes.

Comúnmente, las soluciones similares pueden representar un patrón. Cuando decidimos como formar el patrón, es muy importante considerar la mejor manera de comunicar la solución.

1.7 Patrones contra estrategias

Algunas de las relaciones entre las estrategias y los patrones son las siguientes:

- Los patrones existen a un mayor nivel de abstracción que las estrategias.
- Los patrones incluyen las mejores recomendaciones o las implementaciones más comunes que las estrategias.
- Las estrategias proveen un punto de extensibilidad para cada punto de cada patrón. Los desarrolladores descubren e inventan nuevos caminos para implementar los patrones, produciendo nuevas estrategias para patrones conocidos.
- Las estrategias promueven una mejor comunicación al proveer aspectos de nivel bajo a una solución en particular.

1.8 El enfoque de las capas

Debido a que los patrones descritos que ayudan a construir aplicaciones se ejecutan en la plataforma J2EE, y dado que la plataforma J2EE es un sistema multicapa, veremos el sistema en términos de capas. Una capa es una partición lógica de la separación de contextos. Cada capa tiene asignada su responsabilidad única en el sistema. Veremos cada capa lógicamente separada de cada una. Cada capa es de bajo acoplamiento con la capa adyacente. La figura 1.8 muestra el sistema completo de capas.

Five Tier Model for logical separation of concerns
Client Tier
Application clients, applets, apps, and other GUIs
User interaction, UI presentation, devices
Presentation Tier
JSP, Servlets and other UI elements
Single sign-on, session management, content creation, format and delivery
Business Tier
EJBs and other Business Objects
Business logic, transactions, data, services
Integration Tier
JMS, JDBC, Connectors, and Legacy
Resource adapters, legacy, external systems, rules engines, workflow
Resource Tier
Databases, external systems, and legacy resources
Resources, data and external services
J2EE Patterns Catalog addresses these tiers

Fig 1.8 Modelo de capas

1. 9 Capa del cliente (Client Tier)

Esta capa representa todos los dispositivos o sistemas cliente accediendo al sistema o a la aplicación. Un cliente puede ser un *Web browser,* una aplicación Java, un applet Java, una aplicación de red, puede ser incluso un proceso batch.

1.10 Capa de presentación (Presentation Tier)

Esta capa encapsula todo la presentación lógica requerida para servir a los clientes que accedan al sistema. La capa de presentación intercepta las peticiones del cliente, provee una firma simple a través de la administración de una sesión, controla el acceso a los servicios de negocio, construye las respuestas, y entrega dichas respuestas al cliente. Servlets y JSPs residen en esta capa.

1.11 Capa de negocio (Business Tier)

Esta capa provee los servicios de negocio requeridos por las aplicaciones cliente. La capa contiene la lógica de negocio de datos. Típicamente, la mayor parte del procesamiento de negocio de la aplicación se centraliza en esta capa. Es posible que, debido al comportamiento del sistema, alguna parte del procesamiento de negocio pueda ocurrir en la capa de recursos. Los beans de negocio (Enterprise beans) son los componentes que usualmente se usan para implementar el negocio entre objetos en la capa de negocio.

1.12 Capa de integración (Integration Tier)

Esta capa es responsable de la comunicación con recursos externos y sistemas tales como datos almacenados y aplicaciones. La capa de negocio esta acoplada con la de integración una vez que los objetos de negocio requieren de datos y servicios que residen en la capa de recursos. Los componentes en esta capa pueden usar JDBC, que son conectores de la tecnología J2EE, o algún software intermediario para trabajar con la capa de recursos.

1.13 Capa de recursos (Resource Tier)

Es la capa que contiene los datos de negocio y recursos externos tales como mainframes, sistemas negocio-a-negocio (business-to-business) y servicios tales como autorización de tarjetas de crédito.

2. El Modelo Vista Controlador

2.1 Contexto

Las aplicaciones presentan contenido para usuarios en numerosas páginas que contienen información. Además, el equipo de ingeniería responsable de diseñar, implementar y

mantener las aplicaciones está compuesto de individuos con un conjunto de habilidades diferentes.

2.2 El problema

Ahora bien, hoy más que nunca, las aplicaciones empresariales necesitan soporte para múltiples tipos de usuarios con múltiples tipos de interfaces. Por ejemplo, una tienda en línea podría requerir de un frente HTML para clientes Web, un frente WML para clientes inalámbricos, una interfase Java JFC / Swing para administradores, y un servicio basado en XML Web para proveedores. La siguiente figura 2.2 muestra un bosquejo del problema presentado.

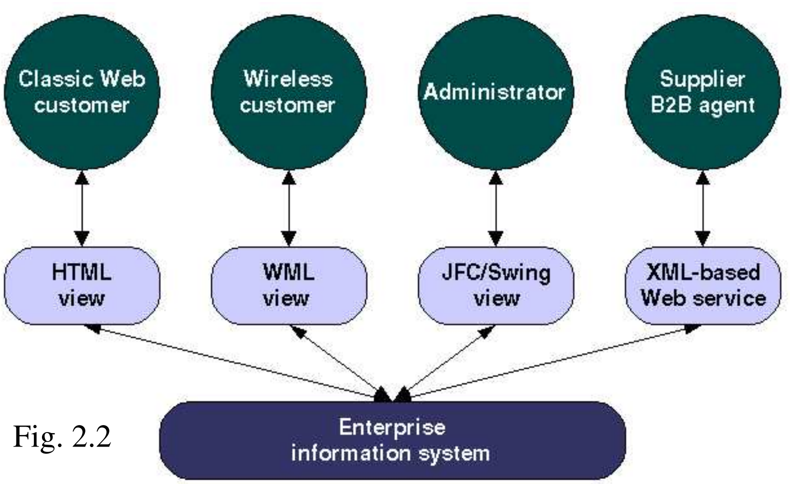


Fig. 2.2

Cuando se desarrolla una aplicación para ayudar a un solo tipo de cliente, algunas veces es benéfico intercambiar el acceso a los datos y la lógica de las reglas de negocio con lógica específica de interfase para la presentación y el control. Tal enfoque, sin embargo, es inadecuado cuando se aplica a sistemas empresariales que necesitan soporte para múltiples tipos de clientes. Diferentes aplicaciones se necesitan desarrollar, una para soportar cada tipo de la interfaz del cliente. El código específico de la interfaz se duplica en cada aplicación, lo cual resulta en duplicar el esfuerzo en la implementación, así como las pruebas y el mantenimiento. La tarea de determinar que es lo que se va a duplicar es extensiva en si misma.

2.3 Las obligaciones

La misma información empresarial debe ser accedida aún presentada en diferentes vistas, por ejemplo. HTML, WML, JFC/Swing, XML.
La misma información empresarial debe ser actualizada a través de diferentes interacciones: por ejemplo, las selecciones de ligas en una página HTML o una página WML, presionando botones en una aplicación JFC/Swing o mediante mensajes SOAP escritos en XML.
Soportar múltiples tipos, vistas e interacciones no debería impactar los componentes que proveen la funcionalidad principal de la aplicación empresarial.

2.4 La solución

Aplicando el diseño Modelo-Vista-Controlador en una plataforma de aplicación J2EE, se separa el núcleo de la funcionalidad del modelo de negocio del de presentación y lógica de control. Dicha separación permite múltiples vistas para compartir el mismo modelo de datos empresarial, lo que hace que sean soportados diversos clientes para implementar, probar y mantener.

2.5 Estructura

La siguiente figura 2.5 Representa el diagrama del Modelo Vista Controlador, muestra cómo es que los objetos en MVC hacen referencia entre sí. La figura del diamante representa una relación, en la cual un objeto almacena una instancia de otro objeto.

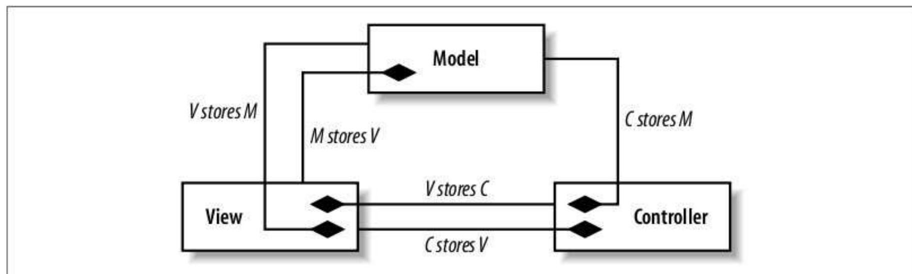


Fig. 2.5 Modelo Vista Controlador

El flujo de la comunicación es en el sentido que muestra la figura anterior, la cual es de la siguiente manera:

- La vista recibe la entrada del usuario y la pasa al controlador.
- El controlador recibe la entrada del usuario proveniente de la vista.
- El controlador modifica el modelo en respuesta a la entrada del usuario (o, en algunos casos, el controlador modifica la vista directamente y no actualiza el modelo en nada).
- El modelo cambia en base a una actualización hecha por el controlador.
- El modelo notifica el cambio de la vista.
- La vista actualiza la interfaz del usuario, (presenta los datos de determinada manera, tal vez reconstruyendo el componente visual).

2.6 Participantes y responsabilidades

La arquitectura MVC tiene sus raíces en Smalltalk, donde fue aplicada originalmente para mapear la entrada tradicional, procesar y mostrar la salida mediante el modelo de interacción gráfica del usuario. Sin embargo, es muy importante que esos conceptos de sean enfocados dentro del contexto de las aplicaciones multicapa.

3 Instalación de Eclipse IDE

Para dar paso a los temas en los cuales es necesario construir en Java, se presenta a continuación los pasos para configurar un entorno de desarrollo integrado (IDE) en este caso eclipse.

Eclipse es principalmente una plataforma de programación, usada para crear entornos integrados de desarrollo. Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto.

3.1 Descargas necesarias

Eclipse IDE. http://www.eclipse.org 13/06/2009
Java Virtual Machine. http://java.com/es/download/index.jsp 13/06/2009
Apache Tomcat. http://tomcat.apache.org/ 13/06/2009
Configuración inicial de eclipse.

NOTA IMPORTANTE: Previamente se debió de haber instalado el Java SDK y JRE en el mismo directorio, se recomienda que sea uno ubicado directamente en raíz y de nombre corto (p.e. C:\jdk1.5.x)

Crear las siguientes 2 variables de entorno:
El path de la variable debe estar apuntando a la carpeta donde se instaló la máquina virtual java JAVA_HOME = {PATH}/jdk1.5.0_06
El path de esta segunda variable debe apuntar a la carpeta donde se descomprimió tomcat. CATALINA_HOME = {PATH}/apache-tomcat-5.5.16.

Configuración Java SDK.

Los siguientes pasos se ejemplifican mejor en la fig 3.1.1

- Seleccionar *Window/Preferences…*
- Seleccionar *Java/Installed JREs*
- Dar click en *Add…*
- Dar click en *Browser…* de la pantalla emergente.
- Seleccionar la ruta donde se descomprimió el zip del Java jdk.
- Dar clic en *Aceptar…* (Esto puede demorar algunos segundos en lo que se cargan las librerías.

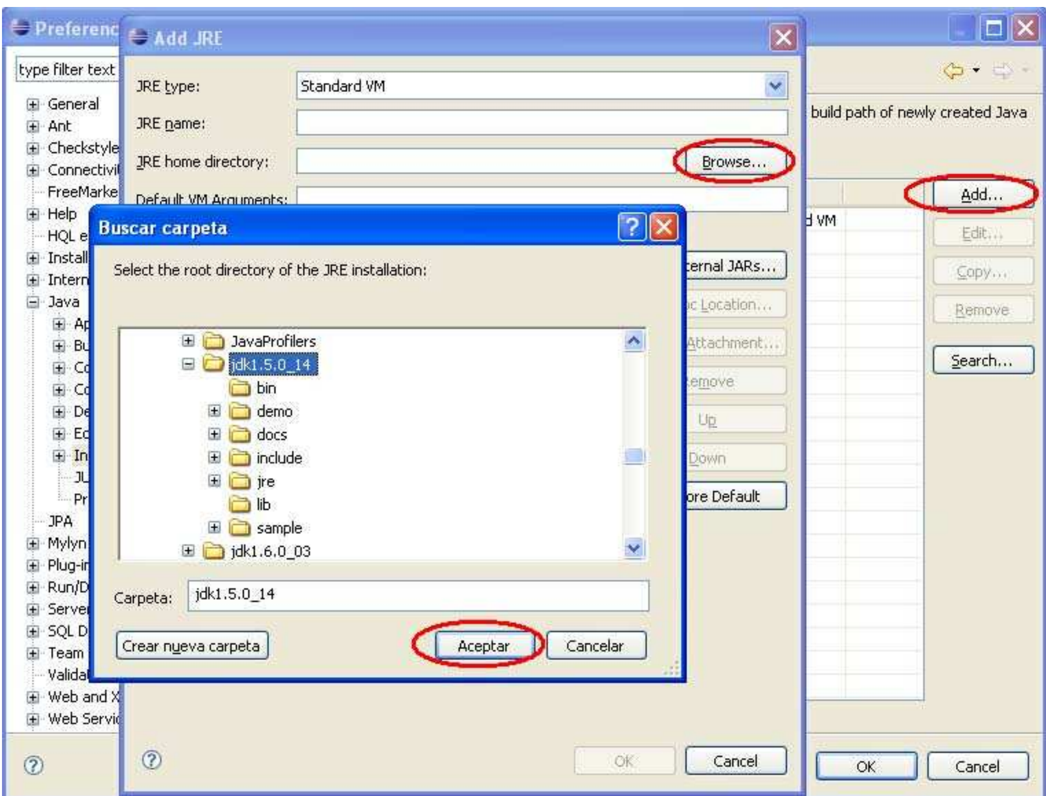


Fig. 3.1.1

- Una vez cargadas las librerías, dar click en *OK*
- Seleccionar el SDK que se acaba de agregar.
- Dar click en *OK* como se ve en la fig.3.1.2

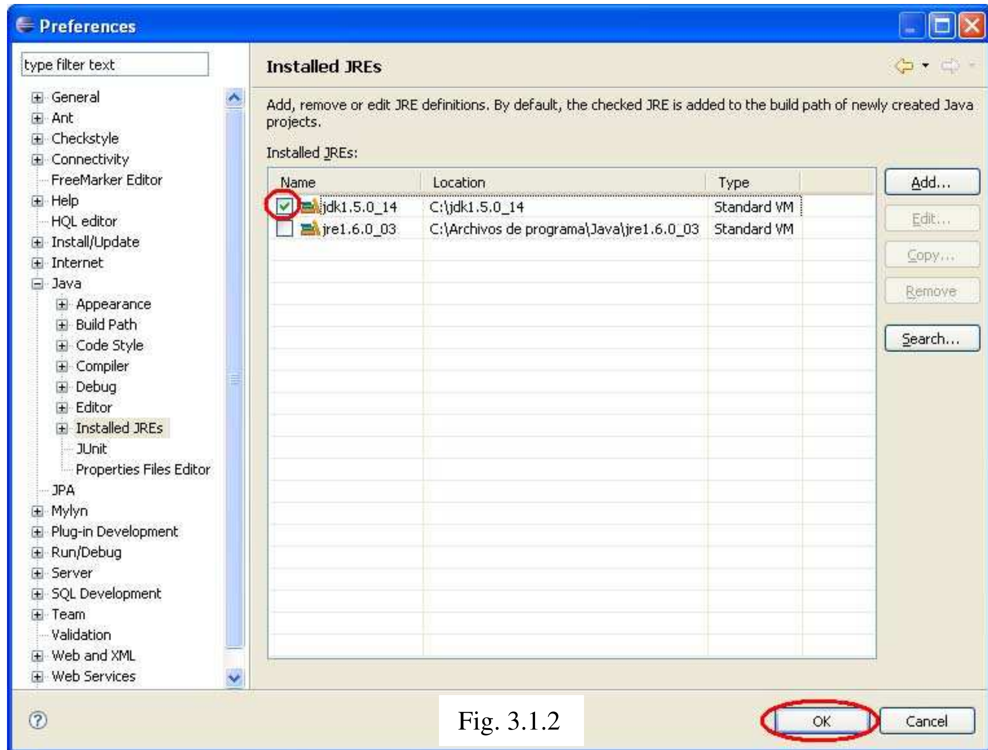


Fig. 3.1.2

3.2 Configuración de Tomcat.

- Seleccionar *Window/Preferences…*
- Seleccionar *Server/Installed Runtimes*
- Dar click en *Add…*
- Seleccionar Apache/Apache Tomcat v5.5
- Dar click en *Next*, como se muestra en la figura 3.2.1

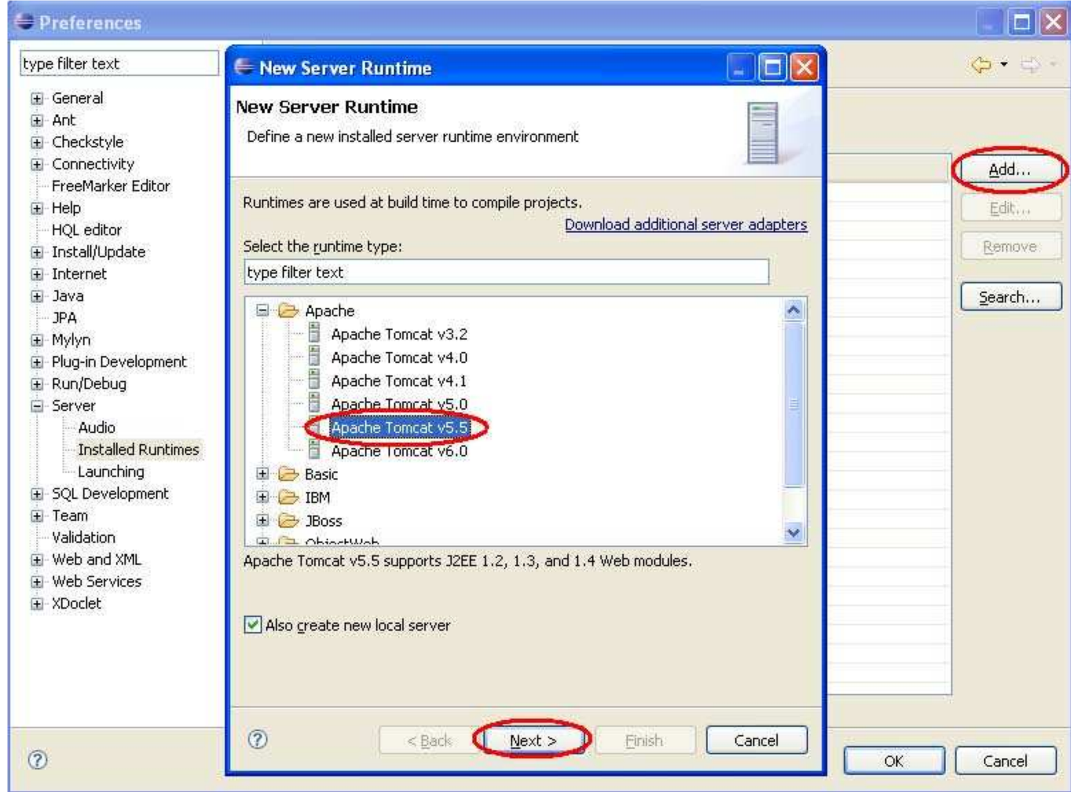


Fig. 3.2.1

- Dar click en *Browse…*
- Seleccionar la carpeta donde se encuentra instalado tomcat.
- Dar click en *Aceptar*
- *Seleccionar el JRE que se configuró en la sección Configuración del Java SDK.*
- Dar click en *Finish.*
- Dar click en *OK*, así como se muestra en la fig 3.2.2

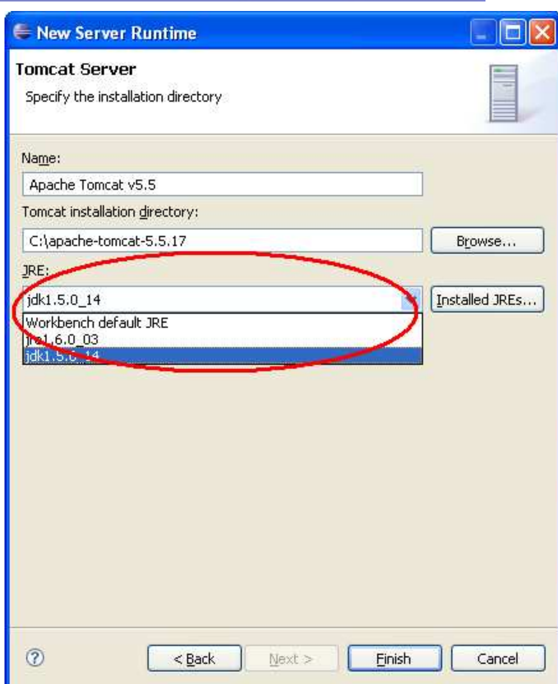


Fig. 3.2.2

3.3 Creación de un Proyecto Web Dinámico

Antes de crear el proyecto, necesita seleccionar o crear alguna carpeta dentro de su sistema. Se sugiere crear una carpeta a nivel raíz, la cual contenga todos los proyectos java. P. ej. C:\Proyectos Java.
A continuación crear la carpeta que contendrá el proyecto, **p. ej.** C:\Proyectos Java**sise12beta**. Tomar en cuenta que es el mismo nombre que se le dará en Eclipse y el mismo nombre que se utilizará en la URL para acceder a él desde el navegador.

Desde Eclipse.

- Para crear el proyecto, realizar lo siguiente:
- Seleccionar *File/New/Other.*
- De la carpeta *Web* Seleccionar *Dynamic/Web/Project.*
- Dar clic en el botón Next. , tal como se muestra en la figura 3.3.1.

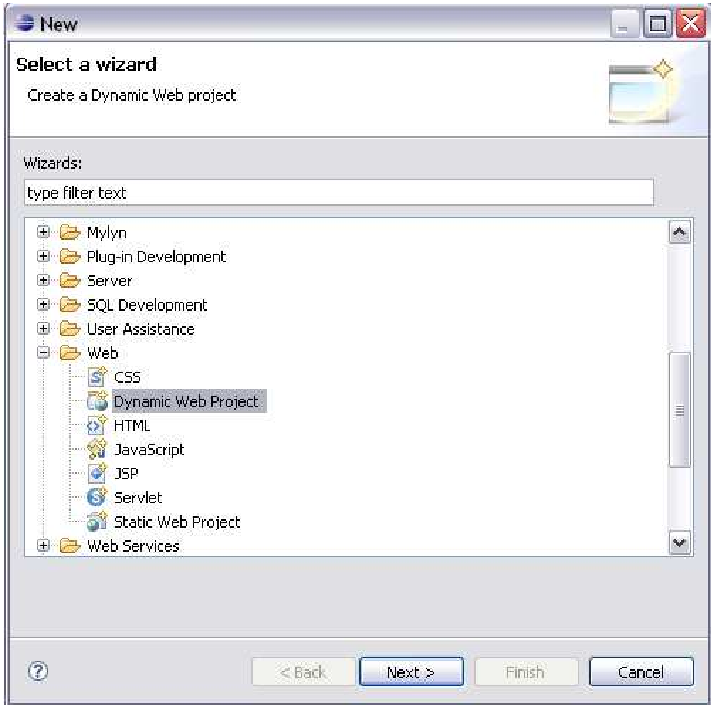


Fig. 3.3.1

En la pantalla de *New Dynamic Web Project*, configurar lo siguiente:

- Como *Project Name* indicar el nombre del proyecto
- En la sección de *Project Contents*: Deseleccionar el check Box de *Use Default*, seleccionar el *directorio base*[1] donde se encuentra el proyecto a configurar.
- En Target Runtime, Eclipse selecciona automáticamente la configuración que se realizó del tomcat.
- Dar click en *Next* como lo ilustra la figura 3.3.2

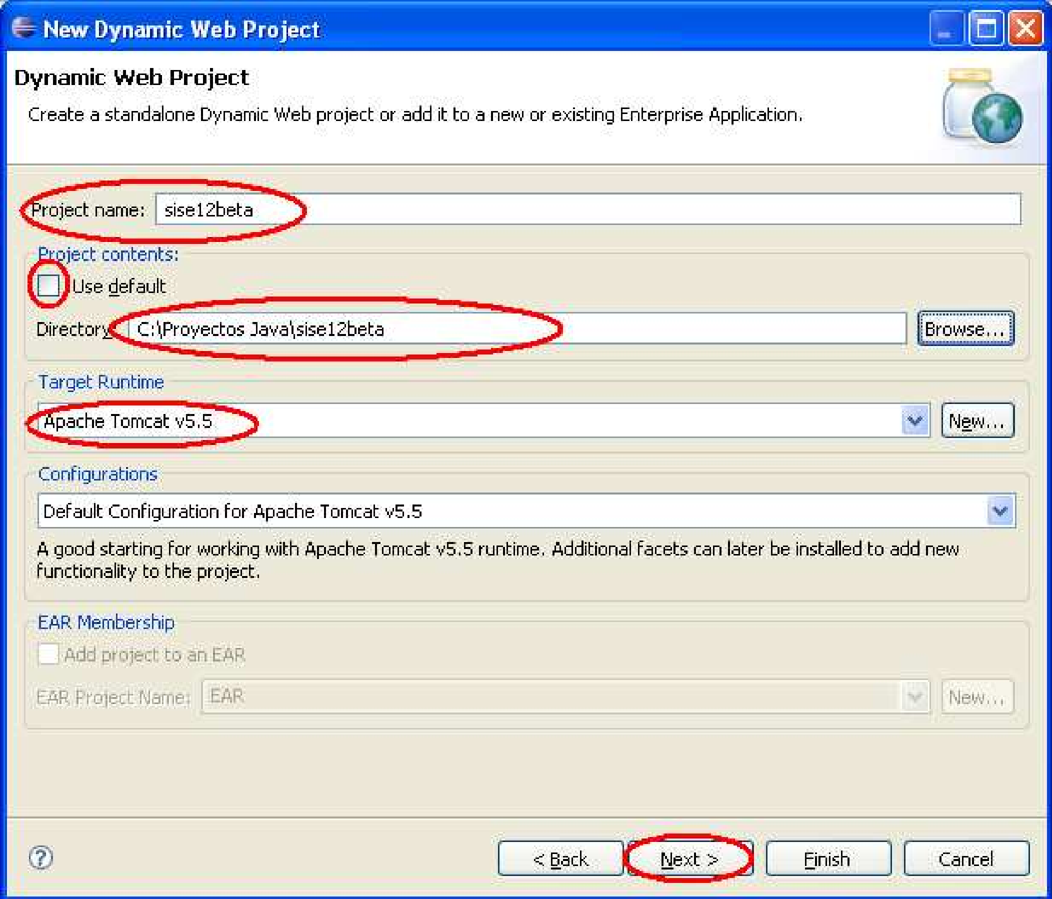


Fig. 3.3.2

- *Dar click en Next.* Como se muestra en la figura 3.3.3

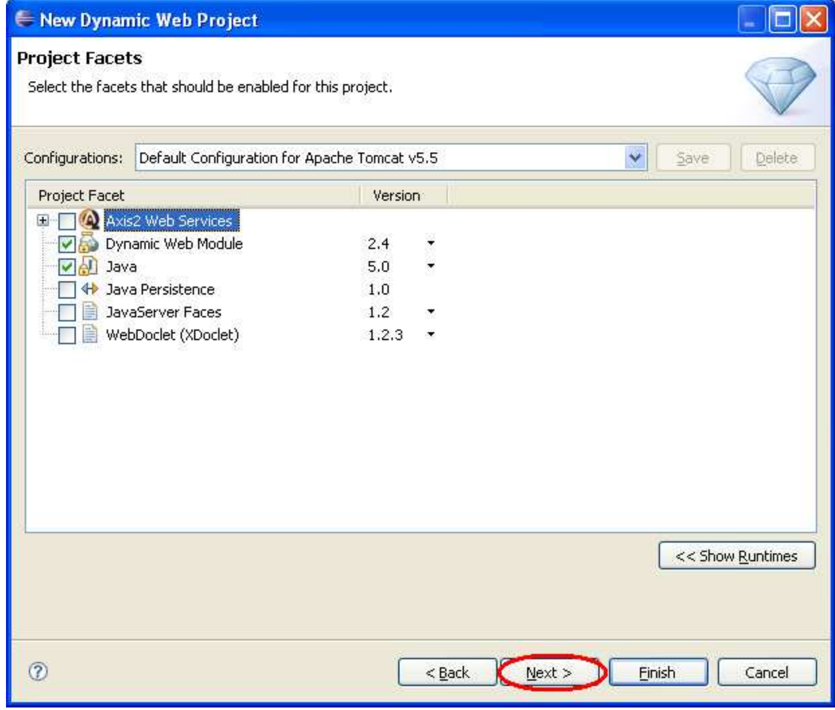


Fig 3.3.3

- Escribir **web** en el campo *Content Directory*.
- *Dar click en Finish.* Así como se muestra en la siguiente figura 3.3.4

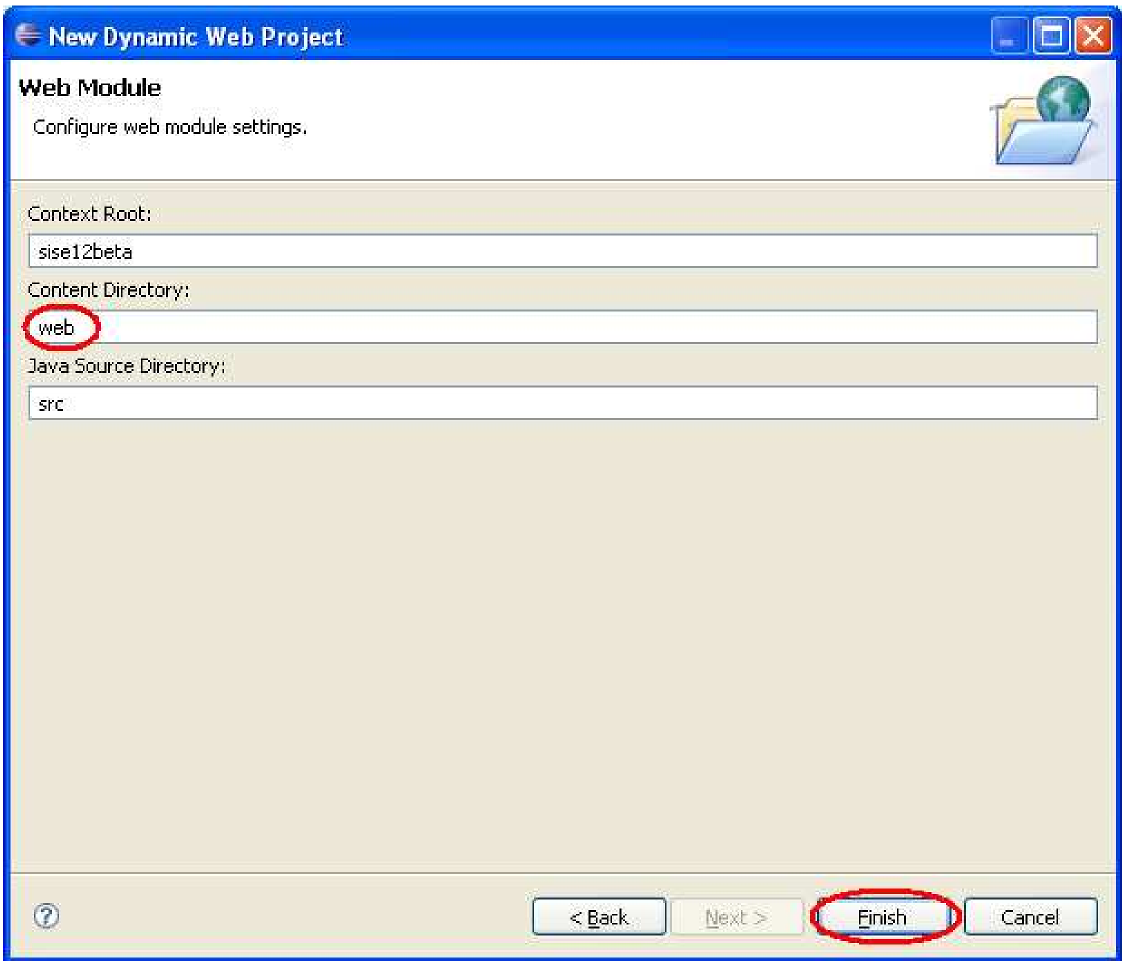


Fig. 3.3.4

4. Modelo de arquitectura de 4 capas

El modelo de diseño de aplicaciones distribuidas en el cuál participé como programador, se basa en múltiples capas, consiste en dividir la lógica de la aplicación en componentes de acuerdo a la función que realizan. En esta sección se desarrolla cada una de las capas de este modelo de 4 capas. La siguiente figura 4 ejemplifica de manera gráfica como está constituido el modelo.

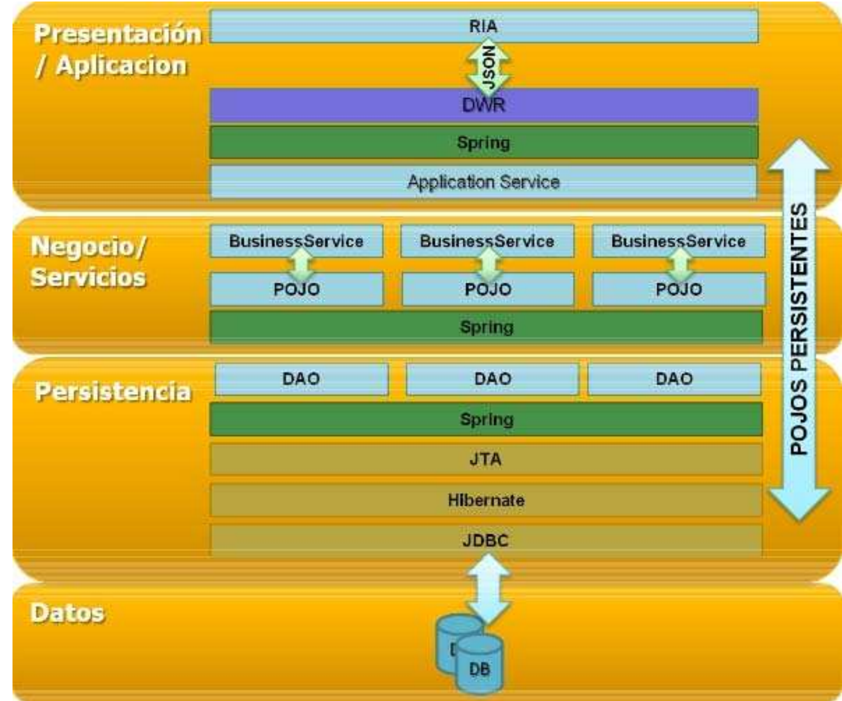


Fig. 4 Modelo de arquitectura de 4 Capas.

4.1 Capa de presentación

En la capa de aplicación RIA (Rich Internet Application) de una aplicación empresarial distribuida Web se toman en cuenta varios factores como:

Los usuarios deben operar en muchas ocasiones bajo marcos de tiempo.
Las condiciones de los equipos (computadoras) con las que cuentan los usuarios no son muy actualizadas.
Los anchos de banda de la red pueden variar.

Una aplicación RIA es una aplicación Web diseñada para proporcionar las mismas características y funcionalidad de una aplicación de tipo escritorio. Generalmente este tipo de aplicaciones dividen el procesamiento de la aplicación por medio de la colocación de las interfaces de usuario, sus usos y capacidades del lado del cliente y las operaciones o manipulaciones de los datos del lado del servidor de aplicaciones.
Con lo anterior se establece el uso de Ext JS, este framework contiene la característica de crear interfaces muy dinámicas y de fácil mantenimiento debido a que ya cuenta con una

gama de controles sobre Javascript que dan la apariencia de una aplicación Cliente/Servidor pero contenida en un browser.
También para facilitar el desarrollo con Ext JS se establece el uso de DWR, framework cuya característica principal es la exposición de los componentes de servicios desarrollados en Java y que radican en el servidor de aplicaciones, como objetos que puede ser accedidos por Ext JS como Javascript nativo. Adicionalmente DWR provee mecanismos para convertir de manera automatizada los objetos de dominio (por ejemplo, objetos persistentes que se obtuvieron de la capa de persistencia) a objetos de javascript (mediante JSON) permitiendo a Ext obtener información directamente del modelo de dominio sin necesidad de crear value objects.

4.1.1 Servicios de Aplicación

Dentro de la capa de presentación se incluyen los llamados **servicios de aplicación**, los cuales son servicios responsables de atender las peticiones del front end. Éstos son los que se exponen para ser consumidos por EXT JS. La responsabilidad de los servicios de aplicación es preparar la información requerida por las vistas.
Los servicios de aplicación no deben contener lógica de negocio (para eso son los servicios de negocio de la capa correspondiente). Los servicios de negocio deben de ser más generales y reutilizables, a diferencia de los servicios de aplicación aquí descritos; más de una aplicación puede interactuar con el mismo modelo de negocio.
Los servicios de aplicación pueden devolver al browser tanto objetos persistentes de dominio directamente, o bien objetos especializados para el despliegue de cierta información como por ejemplo árboles. Los servicios de aplicación deben de extender de una clase abstracta predefinida la cual encapsula funcionalidad genérica para el manejo de objetos de dominio.

4.1.2 Direct Web Remoting (DWR)

DWR, por sus siglas en inglés Direct Web Remoting, es un producto de código abierto. DWR es un interesante enfoque para Ajax en el punto en que permite tratar las clases Java que se ejecuten en el servidor como si fuesen locales, lo cual significa, que se ejecutan en el Web browser. DWR es una solución Java-céntrica, DWR es, esencialmente, una forma de RPC, como lo dice su nombre Direct Web **Remoting.**

4.1.2.1 Acerca de RPC

RPC (Remote Procedure Call) es un mecanismo en donde el código se ejecuta en un espacio de direcciones, típicamente un dispositivo físico de cómputo, este ejecuta una subrutina o un procedimiento en otro o en una red compartida. Esto se lleva a cabo sin que el programador del código, codifique explícitamente para obtener detalles de la interacción remota. De otra manera, el programador escribe en esencia el mismo código si la función que se llama está en un sistema local o un sistema remoto. La invocación remota o invocación del método remoto significa lo mismo como RPC.

En el modelo DWR, lo que llamamos "sistema" es Javascript ejecutándose en el browser del usuario, y el sistema que se está llamando son las clases Java corriendo en el servidor. Justo como otras formas de RPC, el código escrito en Javascript es muy similar al código que se escribe en el servidor para ejecutar el código.

El siguiente es un pequeño ejemplo de lo descrito anteriormente, considera la definición de la siguiente clase.

```
Package com.memorias.app;
public class MiClase {
     String saludo (String name) {
     return "Hello, " + name;
     }
}
```

Si se desea llamar este código vía DWR desde Javascript ejecutándose en el browser y asumiendo que ya se llevo a cabo toda la configuración necesaria, el siguiente código sería todo lo necesario.

```
MiClase.saludo ("Memorias de Trabajo", saludoHandler);
var saludoHandler = function (data) {
alert (data);
}
```

La sintaxis de la llamada es bastante similar a lo que se hace en Java, con la adición del segundo parámetro, es una función que será ejecutada cuando la invocación se complete.

DWR consiste de dos partes principales:

- Un servlet Java ejecutándose del lado del servidor que procesa las peticiones y manda las respuestas de regreso al browser.
- La otra parte Javascript ejecutándose en el browser que manda las peticiones y puede actualizar dinámicamente la página Web.

4.1.2.2 Resumen de la arquitectura DWR

La figura 4.1.2.2 muestra concisamente cómo funciona DWR. Se puede ver del lado izquierdo código JavaScript, una función llamada eventHandler(), en la cual se hace una llamada al método getOptions() en el objeto AjaxService. Este código luce como que el método está siendo ejecutado en un objeto local JavaScript, pero de hecho está llamando al método en un objeto Java que se ejecuta en el servidor. No se sabría esto viendo simplemente en el código, lo cual es una ventaja de DWR. El parámetro que se pasa al método, populateList(), es el nombre de otra función Javascript, y esta es la llamada de regreso. Será ejecutada cuando la llamada al objeto remoto regrese; de ahí la razón del círculo con las flechas que indican el flujo en ambas direcciones.
La principal parte de DWR, lo que se puede definir como el núcleo, es la clase DWRServlet. Es básicamente un servlet Java ordinario, pero realiza diversas tareas por

nosotros, de una de ellas es generar automáticamente objetos Javascript basados en clases Java. Como se verá a continuación, una de las primeras cosas que se necesitan hacer es incluir en la página web el link al archivo Javascript:

```
<script type="text/javascript"
src="myApp/dwr/interface/AjaxService.js"> </script>
```

Nótese que el nombre del archivo Javascript que se está importando. Es el nombre de la clase java en el diagrama. Lo que pasa cuando la página que tiene esta línea es interpretada en el browser es que se hace una petición al recurso AjaxService.js en la ubicación miApp/dwr/interface. Esta URI se mapea al DWRServlet. El servlet entonces inspecciona dinámicamente la clase Java AjaxService y genera Javascript representándola y regresa el código Javascript.

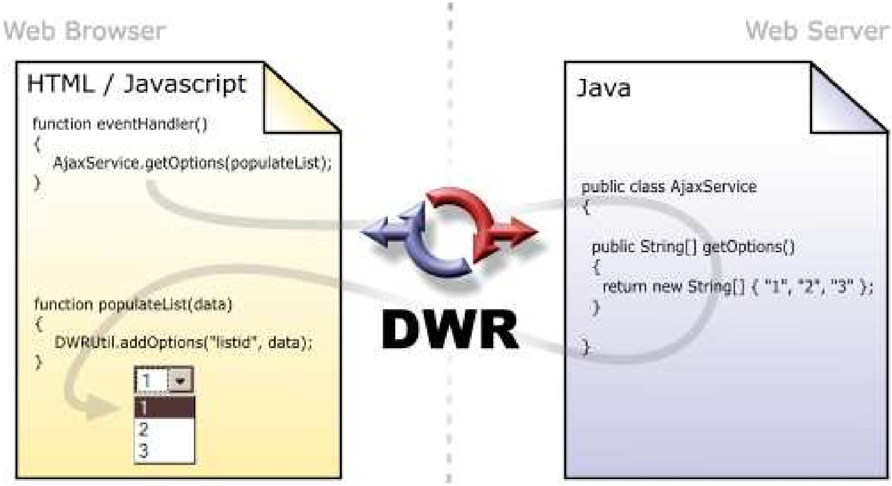


Fig. 4.1.2.2

4.1.3 Preparando el ambiente DWR

Los siguientes 3 pasos son recomendados:

1 Instalar el archivo DWR jar

Descargar el archivo dwr.jar de la ruta http://directwebremoting.org/dwr/download.html 06/14/2009, colocarlo en el directorio WEB-INF/lib de la aplicación web. Desde la versión 2.0, DWR requiere además agregar el archivo commons-logging http://commons.apache.org/downloads/ 06/14/2009.

2 Editar los archivos de configuración

La siguientes líneas se deben agregar al archivo WEB-INF/web.xml. La sección <servlet> debe ir con las otras secciones <servlet>, así como con las secciones <servlet-mapping>.

```
<servlet>
```

```
  <servlet-name>dwr-invoker</servlet-name>
  <display-name>DWR Servlet</display-name>
  <servlet-
class>org.directwebremoting.servlet.DwrServlet</servlet-
class>
  <init-param>
     <param-name>debug</param-name>
     <param-value>true</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

3. Revisar la siguiente URL http://localhost:8080/nombre_de_la_aplicacion/dwr/. Debería ser visible una página que muestra las clases que se han seleccionado en el paso 2.

4.1.4 Ejemplo de actualización de texto dinámicamente

Esto es una simple demostración de cómo actualizar dinámicamente una página-web, con texto traído desde el servidor web.

Código html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Memorias de trabajo</title>
<script type='text/javascript' src='../dwr/engine.js'>
</script>
<script type='text/javascript' src='../dwr/util.js'>
</script>
<script type='text/javascript'
src='../dwr/interface/Texto.js'> </script>
<script type="text/javascript" src='../js/text.js'> </script>
</head>
<body>
<p>
Nombre:
<input type="text" id="campoTexto"/>
<input value="Enviar" type="button" onclick="saludar()"/>
```

```
<br/>
Respuesta: <span id="respuesta"></span>
</p>

</body>
</html>
```

Código Javascript

```
function saludar() {
var name = dwr.util.getValue("campoTexto");
Texto.saludo(name, function(data) {
dwr.util.setValue("respuesta", data);
});
}
```

Código Java

```
package com.dwr.prueba;

public class Texto {
public String saludo(String nombre) {
return "Hola, " + nombre;
}
}
```

Configuración dwr.xml

```
<!DOCTYPE dwr PUBLIC
"-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
"http://getahead.org/dwr/dwr20.dtd">

<dwr>
<allow>
<create creator="new" javascript="Texto">
<param name="class" value="com.dwr.prueba.Texto"/>
</create>
</allow>
</dwr>
```

En la figura 4.1.4 se muestra el árbol jerárquico de paquetes, es la configuración adecuada para ejecutar la aplicación desde eclipse.

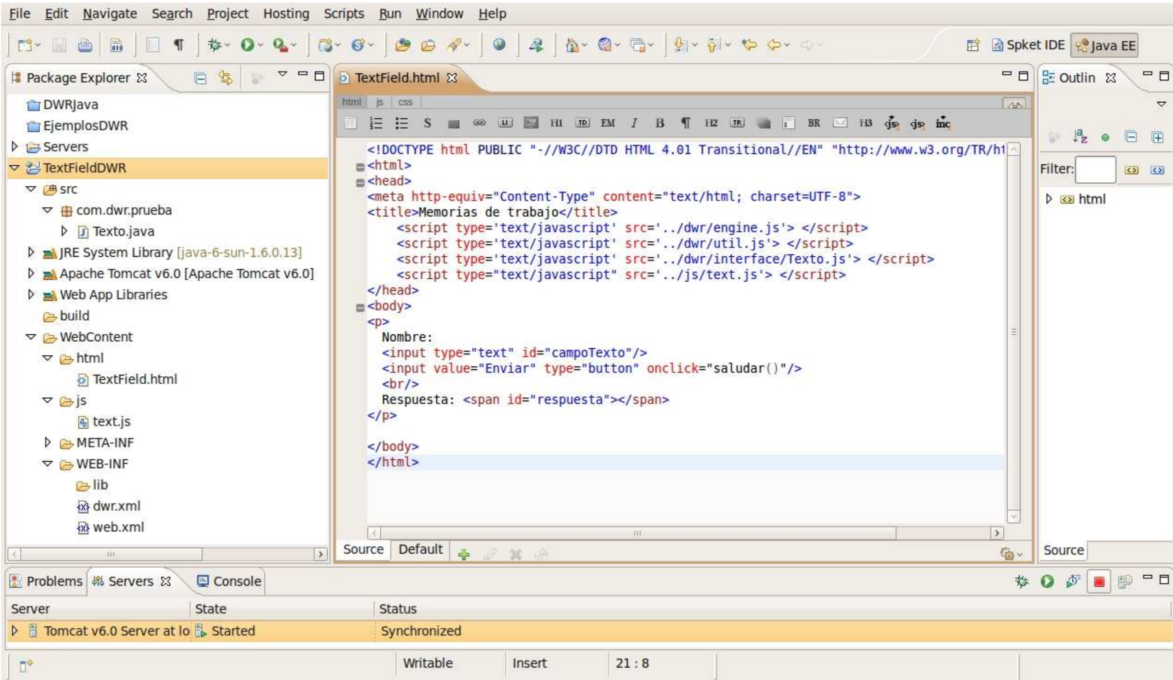


Fig. 4.1.4

4.1.5 Resumen del ejemplo presentado.

Cuando se da un click en el botón “Enviar” el browser llama el evento onclick, que a su vez llama a la función saludar(), en esta función se usa dwr.util.getValue() la cual es una utilería para obtener el valor de cualquier elemento, en este caso un campo de texto, pero podría ser un div o una caja de selección. DWR es asíncrono debido a la manera en que Javascript trabaja el web browser no se detendrá mientras se espera a que la respuesta HTTP regrese. Entonces el segundo parámetro nombra a una función para ser llamada precisamente cuando la llamada regrese. En el servidor, DWR llama el método Java Texto.saludo(), cuando este método regresa, DWR llama la función que actualiza el texto. Haciendo uso de la utilería dwr.util.setValue() la cual toma el dato pasado como segundo parámetro y trabaja para colocarlo en el elemento HTML especificado como primer parámetro. Esta función es una de las diversas utilidades nítidas de Javascript que hacen trabajar más fácil con DWR.

4.1.6 Ext JS

La librería Ext comienza como una extensión de la moderadamente popular, pero poderosa librería de Interfaz de Usuario Yahoo, la cual es una API (Application Programming Interface) de componentes comunes (widgets). Ext provee una interfaz de usuario enriquecida, más parecido a lo que se puede encontrar en una aplicación de escritorio. Esto permite a los desarrolladores web concentrarse en la funcionalidad de la aplicación web en vez de lidiar con problemas para construir componentes en Javascript. Ext no es solo otra librería Javascript, de hecho, Ext puede trabajar de la misma forma con librerías Javascript haciendo uso de adaptadores. Típicamente, se usa Ext en sitios web que requieren un nivel superior de interacción con el usuario. Un sitio web que requiere procesamiento y un flujo de trabajo sería un ejemplo perfecto. Ext tiene las siguientes características:

2. Provee un fácil uso a través de la compatibilidad entre browsers para crear widgets tales como ventanas, tablas y formas. Los widgets están bien configurados para manejar todos los posibles problemas de cada web browser en el mercado, sin tener que cambiar nada.
3. Interactúa con el usuario y el browser mediante el EventManager(Administrador de eventos), respondiendo a el presionado de teclas, clicks del ratón, también a los eventos de monitoreo del browser como redimensionar una ventana o cambiar el tamaño de la letra.
4. La comunicación con el servidor en segundo plano sin la necesidad de refrescar la página. Esto permite pedir o enviar información desde o hacia el web server usando AJAX y procesar la respuesta en tiempo real.

En la **capa de presentación** es importante destacar que el modelo Ext es un tanto diferente al modelo tradicional de aplicaciones web basadas en JSP, Struts y/o JSF en donde cada acción del usuario se traduce en una petición al servidor el cual genera una nueva vista completa que se envía al navegador, el cual simplemente "dibuja" dicha vista. El modelo utilizado por Ext JS está basado en el concepto de Rich Client Application (RIA) el cual es un modelo híbrido que integra lo mejor del modelo cliente servidor con lo mejor del modelo web.
Esto significa que el navegador se utiliza como una plataforma de ejecución mucho más poderosa que no simplemente "dibuja" una página de HTML, sino que en realidad toda la parte visual se ejecuta en el mismo navegador. La comunicación con el servidor se limita al envío y recepción de datos (a diferencia del modelo "tradicional" en el que viajan tanto datos como la parte visual en forma de HTML). Este modelo presenta enormes ventajas, tanto de desempeño debido al menor uso de ancho de banda como a una experiencia de usuario extraordinaria debido tanto a un modelo visual más maduro y consistente.

4.1.6.1 Componentes de Ext (widgets)

Existe una extensa gama de widgets disponibles dentro de ExtJS, en esta sección se nombran aquellos widgets más importantes y de mayor uso dentro de una aplicación de desarrollo web, además se presenta el código de construcción y una imagen de muestra.

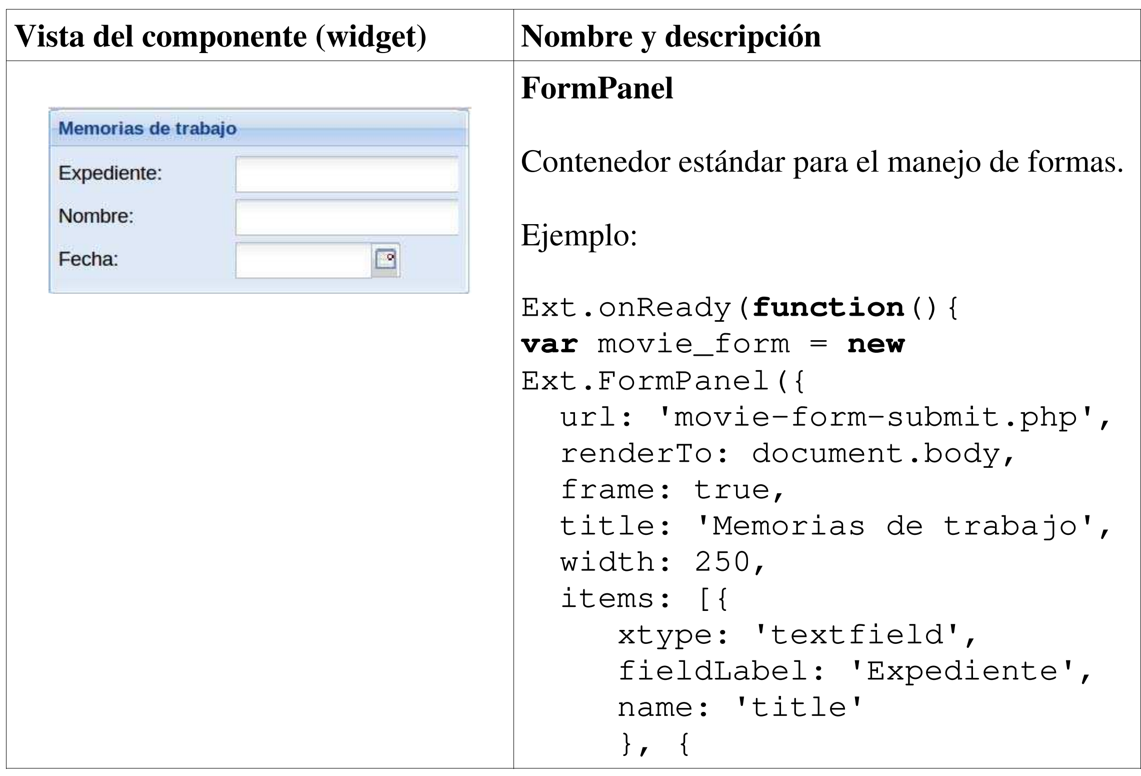
4.1.6.2 Plantilla HTML básica para agregar componentes Ext JS

Debe agregarse en la ruta *WebContent > html,* nótese que los componentes deben ser incluidos dentro de las etiquetas <script> </script>

```
<html>
<head>
<title>Plantilla básica de componentes Ext JS</title>
<link rel="stylesheet" type="text/css"
       href="../lib/ext/resources/css/ext-all.css" />
<script src="../lib/ext/adapter/ext/ext-base.js"></script>
<script src="../lib/ext/ext-all-debug.js"></script>
   <script>
       Ext.onReady(function(){
           Ext.Msg.alert('Hola', 'Mundo');
       });
   </script>
</head>
<body>
   <!-- Body vacío -->
</body>
</html>
```

4.1.6.3 Widgets Ext JS

Vista del componente (widget)	Nombre y descripción


 	
	FormPanel
	
	Contenedor estándar para el manejo de formas.
	
	Ejemplo:
	
	`Ext.onReady(function(){`
	`var movie_form = new`
	`Ext.FormPanel({`
	`  url: 'movie-form-submit.php',`
	`  renderTo: document.body,`
	`  frame: true,`
	`  title: 'Memorias de trabajo',`
	`  width: 250,`
	`  items: [{`
	`     xtype: 'textfield',`
	`     fieldLabel: 'Expediente',`
	`     name: 'title'`
	`     }, {`

```
      xtype: 'textfield',
      fieldLabel: 'Nombre',
      name: 'director'
      }, {
      xtype: 'datefield',
      fieldLabel: 'Fecha',
      name: 'released'
      }]
  });
});
```

Campo de texto:
Campo numérico:

TextField, NumberField

Campo de texto básico, es usado como reemplazo al tradicional campo de captura de texto en HTML.
En el caso del NumberField, este widget permite el filtrado automático de solo valores numéricos en su captura.

Ejemplo:

```
items: [{
  xtype: 'textfield',
  fieldLabel: 'Campo de texto',
  name: 'texto'
  }, {
  xtype: 'numberfield',
  fieldLabel: 'Campo
num&eacute;rico',
  name: 'numero'
}]
```

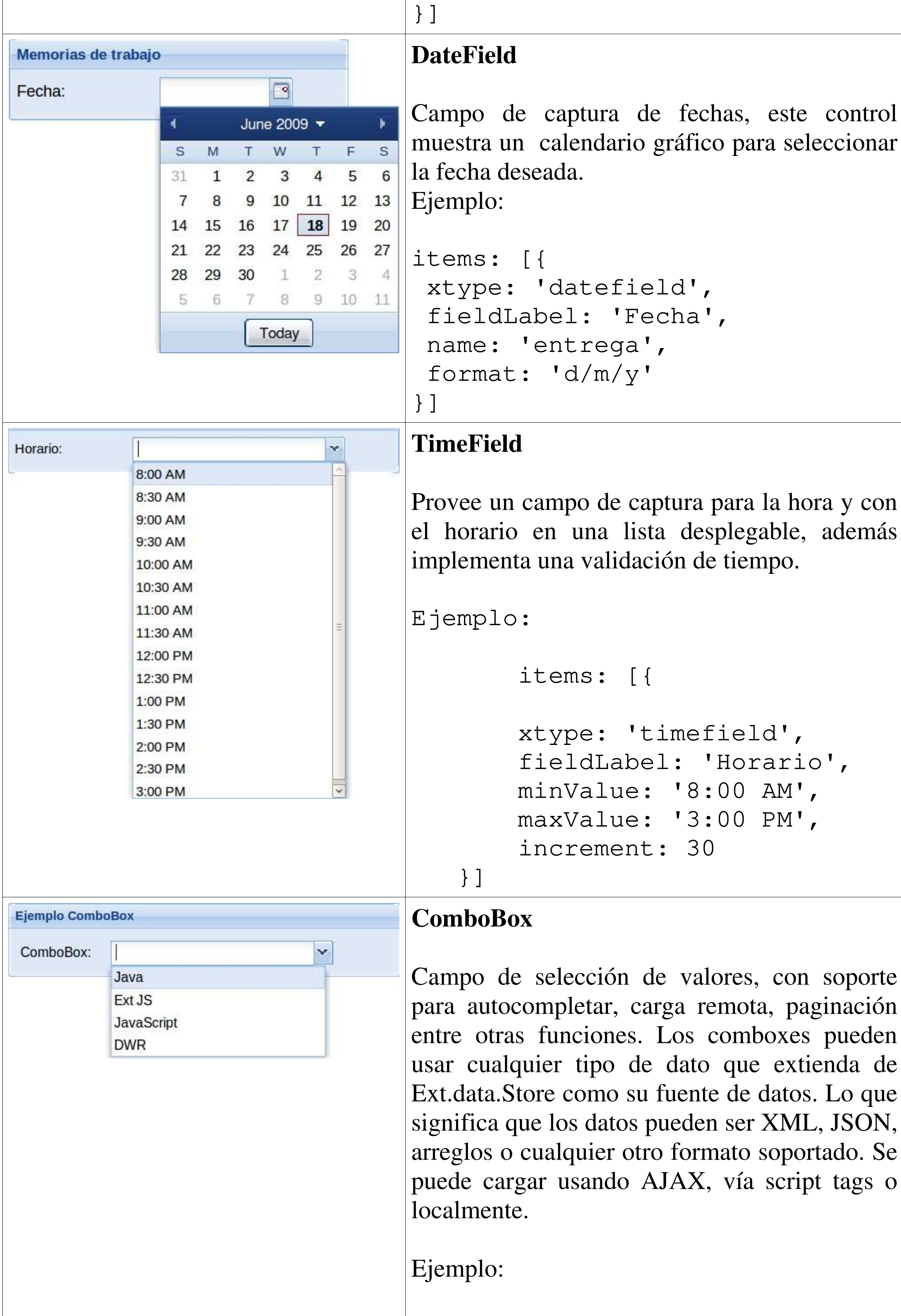
Area de texto:
Memorias de trabajo

TextArea

Campo de área de texto básico, es usado como reemplazo al tradicional campo de captura de área de texto en HTML.

Ejemplo:

```
items: [{
 xtype: 'textarea',
 fieldLabel: 'Area de texto',
 name: 'descmotivo',
 disabled: false,
 width:300,
 height: 40
```




```
}]
```

DateField

Campo de captura de fechas, este control muestra un calendario gráfico para seleccionar la fecha deseada.
Ejemplo:

```
items: [{
 xtype: 'datefield',
 fieldLabel: 'Fecha',
 name: 'entrega',
 format: 'd/m/y'
}]
```

TimeField

Provee un campo de captura para la hora y con el horario en una lista desplegable, además implementa una validación de tiempo.

```
Ejemplo:

       items: [{

       xtype: 'timefield',
       fieldLabel: 'Horario',
       minValue: '8:00 AM',
       maxValue: '3:00 PM',
       increment: 30
   }]
```

ComboBox

Campo de selección de valores, con soporte para autocompletar, carga remota, paginación entre otras funciones. Los comboxes pueden usar cualquier tipo de dato que extienda de Ext.data.Store como su fuente de datos. Lo que significa que los datos pueden ser XML, JSON, arreglos o cualquier otro formato soportado. Se puede cargar usando AJAX, vía script tags o localmente.

Ejemplo:

```
var store = new
```

```
Ext.data.SimpleStore({
 fields: ['nombreDato',
'nombreElemento'],
 data: [['J', 'Java'], ['E',
'Ext JS'], ['JS',
'JavaScript'], ['D', 'DWR']],
 autoLoad: false
});

items: [{
   xtype: 'combo',
   store: store,
   fieldLabel: 'ComboBox',
   displayField:
'nombreElemento',
   valueField: 'nombreDato',
   typeAhead: true,
   emptyText: "Seleccione una
opcion ...",
   forceSelection: true,
   mode: 'local',
   triggerAction: 'all',
   selectOnFocus: true,
   editable: true
}]
```

Memorias de trabajo
Componente de agrupación
Seleccione: Java
Ext JS
DWR

FieldSet / CheckBox

Es un componente estándar usado para agrupar otros componentes en este ejemplo se muestra la agrupación de checkboxes.

Ejemplo

```
items: {
   xtype:'fieldset',
   title: 'Componente de
agrupaci&oacute;n',
   autoHeight: true,
   defaultType: 'checkbox',
   items: [{
      fieldLabel: 'Seleccione',
      boxLabel: 'Java',
      name: 'java'
   },{
      fieldLabel: '',
```

<table>
<tr><td></td><td>

```
      labelSeparator: '',
      boxLabel: 'Ext JS',
      name: 'ext'
   },{
      checked: true,
      fieldLabel: '',
      labelSeparator: '',
      boxLabel: 'DWR',
      name: 'dwr'
   }]
}
```

</td></tr>
<tr><td>
</td><td>

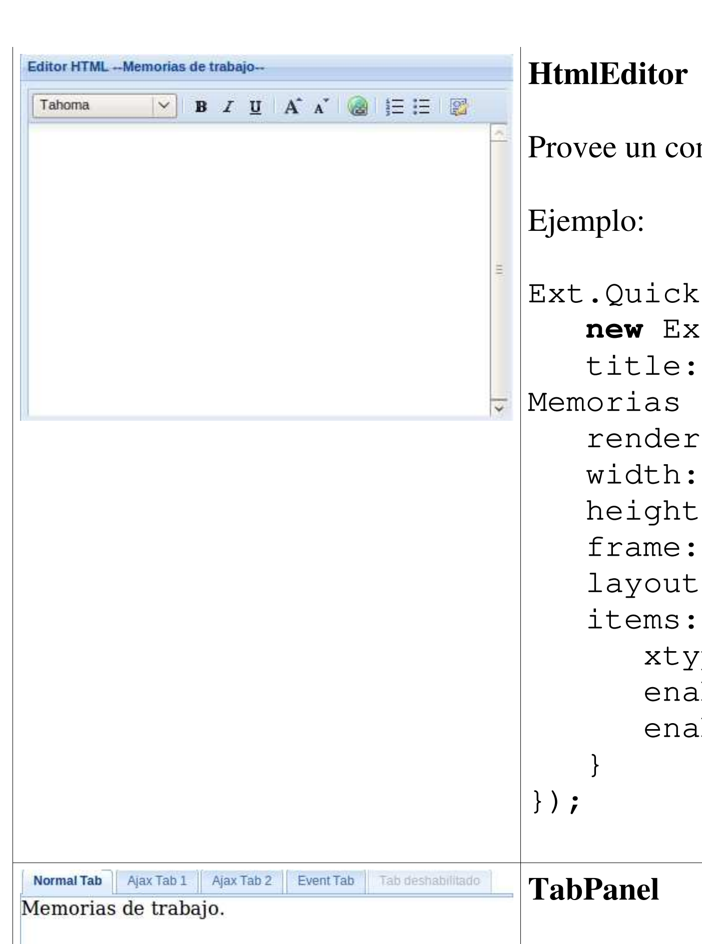
Radio

Campo de radio simple. Es igual que el componente checkbox, pero provee el ajuste automático del tipo de entrada. La agrupación de radios se maneja automáticamente por el browser si se le da a cada radio en un grupo el mismo nombre.

Ejemplo:

```
items: [{
   xtype: 'radio',
   boxLabel: 'Radio 1',
   inputValue: 'radio1',
   name: 'radio',
   checked: false,
   id:'radio1',
   valueField: 'radio1'
   }, {
   xtype: 'radio',
   boxLabel: 'Radio 2',
   inputValue: 'radio2',
   name: 'radio',
   checked: false,
   id:'radio2',
   valueField: 'radio2'
}]
```

</td></tr>
</table>



HtmlEditor

Provee un componente ligero para editar.

Ejemplo:

```
Ext.QuickTips.init();
   new Ext.Panel({
   title: 'Editor HTML --
Memorias de trabajo--',
   renderTo: Ext.getBody(),
   width: 400,
   height: 300,
   frame: true,
   layout: 'fit',
   items: {
      xtype: 'htmleditor',
      enableColors: false,
      enableAlignments: false
   }
});
```

Normal Tab | Ajax Tab 1 | Ajax Tab 2 | Event Tab | Tab deshabilitado

Memorias de trabajo.

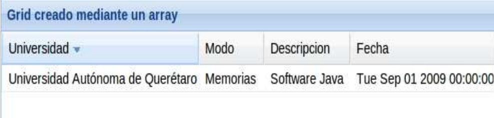
TabPanel

Es un contenedor básico de tabs, con propósitos de distribución de componentes de la misma manera que el panel. Aunque suministra soporte adicional para componentes hijos que se desplegarán en cada tab.

Ejemplo:

```
new Ext.TabPanel({
   renderTo: document.body,
   activeTab: 0,
   width:430,
   height:250,
   plain:true,
   defaults:{autoScroll: true},
   items:[{
      title: 'Normal Tab',
      html: "Memorias de
trabajo."
   },{
      title: 'Ajax Tab 1',
      autoLoad:'ajax1.htm'
```

```
    },{
        title: 'Ajax Tab 2',
        autoLoad: {url:
'ajax2.htm', params:
'foo=bar&wtf=1'}
    },{
        title: 'Event Tab',
        listeners: {activate:
handleActivate},
        html: "Tabulacion de
prueba"
    },{
        title: 'Tab
deshabilitado',
        disabled:true,
        html: "Deshabilitado"
    }]
});
```



Grid

Un grid es un componente para mostrar información en formato de tabla. El siguiente ejemplo muestra un grid creado en base a un array.

```
//Array de información a
mostrar.
var informacion =
[['Universidad Aut&oacute;noma
de Quer&eacute;taro',
'Memorias','Software
Java',0.03,'9/1 12:00am']
];
//Configuración de las columnas
grid
var store = new
Ext.data.SimpleStore({
    fields: [
        {name: 'universidad'},
        {name: 'modo', type:
'string'},
        {name: 'descripcion',
type: 'string'},
        {name: 'fecha', type:
'date', dateFormat: 'n/j h:ia'}
```

	``
	    ]
	});
	
	store.loadData(informacion);
	//Creación del grid.
	   var grid = new
	Ext.grid.GridPanel({
	      store: store,
	      columns: [
	
	{id:'universidad',header:
	"Universidad", width: 160,
	sortable: true, dataIndex:
	'universidad'},
	         {header: "Modo",
	width: 75, sortable: true,
	dataIndex: 'modo'},
	         {header:
	"Descripcion", width: 75,
	sortable: true, dataIndex:
	'descripcion'},
	         {header: "Fecha",
	width: 85, sortable: true,
	dataIndex: 'fecha'}
	],
	     stripeRows: true,
	     autoExpandColumn:
	'universidad',
	     height:100,
	     width:600,
	     title:'Grid creado
	mediante un array'
	});
	
	    grid.render(document.body);
	``

4.2 Capa de Servicio o Negocio

Esta capa es la responsable de atender las peticiones de la capa de presentación, ejecutar las reglas de negocio, consulta y actualización de la capa de datos y generación de la respuesta requerida por el cliente.

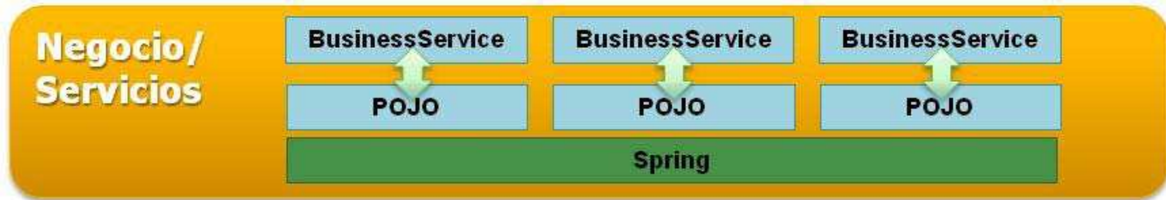


Fig 4.2 Capa de negocio.

En esta capa se plantea cambiar su enfoque para exponer servicios de negocio a través de Interfaces e Implementaciones (POJOs) con la ayuda de Spring, de esta forma el cliente de los servicios no conoce la implementación e inclusive no conoce si el servicio es local o remoto a su contexto solo conoce un contrato (Interfase) que debe de cumplir la implementación, esto permite desacoplar esta capa con las demás capas así como también para que sean consumidos de manera externa y ser mas ágiles en el mantenimiento sin que se vean afectadas las demás capas por aplicar nuevas fórmulas o reglas de negocio.
Con ello también se permite desacoplar la capa de negocio con la de persistencia ya que ahora el negocio no tendrá que declarar la sesión de Hibernate o conexión de JDBC como se venía haciendo anteriormente.
Por la misma característica que nos provee Spring de desacoplamiento de capas con alta cohesión.
Algo que cabe mencionar es que para el completo éxito de esta capa en su implementación a POJOs se debe de contar con un buen modelo de negocio con orientación a objetos. La capa de servicio define los limites de una aplicación y proporciona una serie de operaciones disponibles desde la perspectiva de comunicación con la capa cliente (presentación, interfaz con otro sistema etc...). Esta encapsula la lógica de negocios de una aplicación, controlando las transacciones y coordinando la respuesta en la implementación de estas operaciones. Utilizando Spring como BeanFactory, esta capa de servicios puede implementarse fácilmente, diseñando estos servicios como Beans.

4.2.1 Mapeo Relacional de Objetos con HIBERNATE (ORM-Object Relational Mapping)

ORM es el nombre dado a las tecnologías, herramientas y técnicas usadas como un puente entre la división de los objetos y las bases de datos relacionales. Las herramientas ORM permiten mapear declarativamente los objetos de datos con los datos y relaciones en una base de datos relacional. Lo cual brinda a los programadores trabajar con la información de una base de datos relacional en la forma de objetos Java. Esta clase de herramientas generan todo el código SQL necesario para interactuar con la base de datos. Los programadores trabajan a un nivel de objetos, y el concepto de queries y transacciones son aplicadas a los objetos Java en lugar de los objetos de base de datos.

La mayoría de las herramientas de mapeo no son del todo transparentes, proveen una buena separación de contextos aislando la persistencia de los objetos tras una construcción orientada a objetos muy simple. En HIBERNATE por ejemplo, la transparencia significa que los Objetos Java (POJOs Plain Old Java Objects) que se usan para el mapeo no tienen noción de que son objetos persistentes. El tema con la persistencia en Java es acerca de opciones alternas, como con otras cosas en Java. Ninguna herramienta es perfecta para cada escenario de persistencia. Algunas herramientas cambian la transparencia por desempeño o simplicidad de construcción. Algunas preguntas que emergen durante la selección de un modelo objeto-relacional son las siguientes:

- Cómo convertir valores de columnas a Objetos Java y tipos primitivos. Por ejemplo, un objeto Java de fecha puede ser mapeado a muchos tipos de dato en una base de datos.

- Cómo modelar las relaciones de objetos (tales como herencia, agregación y composición) en un esquema de base de datos o cómo modelar las relaciones entre tablas en un objeto o grupo de objetos.

- Cómo lidiar con llaves primarias de base de datos y la identidad del objeto, que pudiera no existir en el modelo de objetos.

- Cómo optimizar los resultados de las llamadas SQL.

- Cómo tomar ventaja de las características de las bases de datos tales como la actualización de vistas y procedimientos almacenados.

- Cómo garantizar la integridad referencial sin limitar el comportamiento del modelo de objetos.

- Cómo lidiar con las transacciones cuando la base de datos está siendo accedida concurrentemente desde múltiples fuentes. Desde un punto de una vista relacional, un objeto no es nada más que caché en memoria de un dato de la base de datos que debe ser invalidado y refrescado cuando sea apropiado.

- Cómo lidiar con operaciones costosas como cargar muchos objetos hijo en una relación uno-a-muchos (cargas lentas "lazy loading", objetos Proxy, o caché.)

4.2.2 Introducción a HIBERNATE

Hibernate es una herramienta ORM programada como un servicio de persistencia relacional para el lenguaje Java, lo que puede ser traducido al decir que es "un mecanismo transparente de persistencia orientado a objetos". La transparencia en este punto significa que los objetos que son persistidos no tienen ningún código en ellos que exponga su habilidad para ser persistidos. Hibernate provee persistencia y capacidades de consulta SQL con objetos, todo ello en un ambiente de modelo orientado a objetos enriquecido. Provee una programación declarativa orientada a objetos que no depende de alguna generación de código o modificaciones en tiempo de construcción. Hibernate no forza a implementar interfaces especiales o a extender de una clase en particular; se puede trabajar con simples y limpios POJOs.
En Hibernate, el mapeo entre objetos y tablas puede ser definido en documentos XML, programáticamente en Código Java o vía anotaciones JSR-220. Además, proyectos como XDoclet posibilita crear anotaciones a partir de aplicaciones Java Estándar 1.5 con comentarios Javadoc los cuales se usan para generar los mapeos XML.

La siguiente figura 4.2.2 provee una vista de alto nivel de la arquitectura Hibernate:

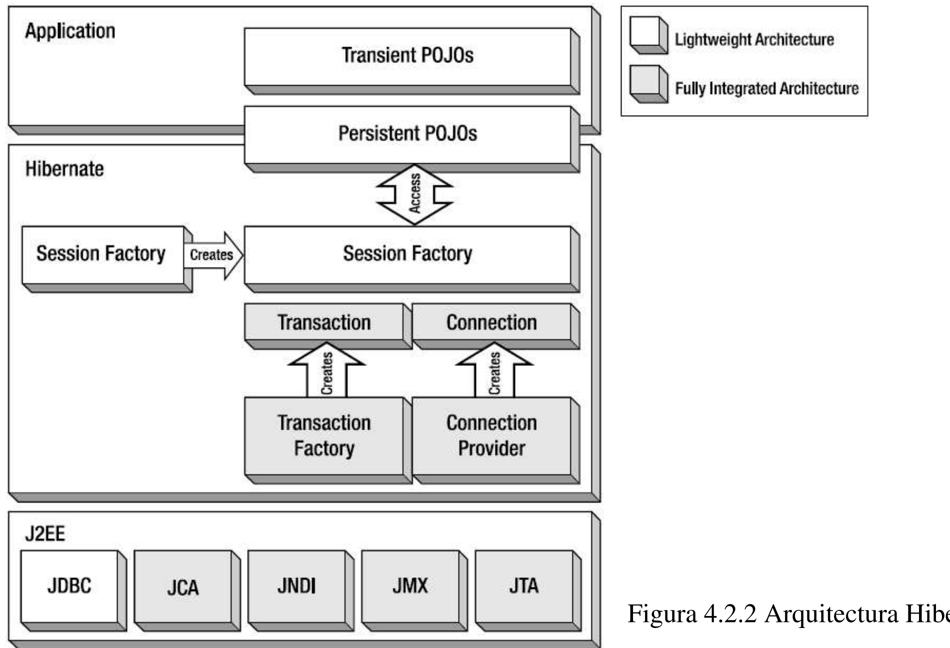


Figura 4.2.2 Arquitectura Hibernate

A continuación se definen los objetos que se mencionan en el diagrama:

`SessionFactory (org.hibernate.SessionFactory)`

Es un hilo de caché inmutable de mapeos compilados para una sola base de datos. Una fábrica para `Session` y un cliente de `ConnectionProvider`, `SessionFactory` pueden mantener un caché opcional (de segundo nivel) de datos que es reutilizable entre transacciones en un proceso o cluster.

`Session (org.hibernate.Session)`

Es un hilo, un objeto de vida-corta que representa una conversación entre la aplicación y el almacén de persistencia. Envuelve una conexión JDBC y es una fábrica para `Transaction`. `Session` que mantiene un caché persistente de objetos que son usados cuando se navega por el grafo de objetos o se busca objetos por su identificador.

Objetos persistentes POJOs y colecciones

Objetos de vida-corta, de un hilo que contienen estado persistente y funcionalidad de negocio. Estos pueden ser ordinariamente JavaBeans/POJOs. Están asociados exactamente con una `Sesión (Session)`. Una vez que se cierra la sesión, los objetos serán desasociados y libres para usarse en cualquier capa de la aplicación (por ejemplo, directamente como objetos de transferencia de datos para y desde la capa de presentación.)

Objetos transitorios (`transient`) y separados.

Son instancias de clases persistentes que no están actualmente asociadas con una sesión (`Session`). Pudieron haber sido inicializadas por la aplicación y no persistidas aún, o inicializadas por una sesión cerrada.

Transaction (`org.hibernate.Transaction`)

(Opcional) De trato hilo-simple, objeto de vida-corta usado por la aplicación para especificar una unidad de trabajo atómica. Abstrae la aplicación de la transacción subyacente JDBC, JTA o CORBA. Una sesión (`Session`) puede producir varias transacciones (`Transaction`) en algunos casos. Sin embargo, la demarcación de la transacción, aún usando la API o `Transaction`, nunca es opcional.

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)

(Opcional) Fábrica de conexiones JDBC. Abstrae la aplicación de las clases subyacentes `Datasource` o `DriverManager`. No está expuesta a la aplicación, pero puede ser extendida y/o implementada por el programador.

TransactionFactory (`org.hibernate.TransactionFactory`)

(Opcional) Es una fábrica de instancias Transaction. No está expuesta en la aplicación, pero puede ser extendida y/o implementada por el programador.

4.2.3 Estados de las instancias

Una instancia de una clase persistente puede estar en uno de los tres diferentes estados. Estos estados están definidos en un contexto relacional persistente. El objeto **`Session`** es el contexto de persistencia. Los tres diferentes estados son como sigue:

Transitorio (Transient)
La instancia (objeto) no está asociada con algún contexto persistente. No tiene valor de llave primaria persistente.

Persistente
La instancia está actualmente asociada con un contexto persistente. Tiene una llave primaria persistente y puede tener una fila correspondiente en la base de datos. Para un contexto persistente en particular, Hibernate garantiza que la identidad persistente es equivalente a una identidad de un objeto Java localizado en memoria.

Desasociado
La instancia fue una vez asociada con un contexto persistente, pero el contexto fue cerrado, o la instancia fue serializada por otro proceso. Tiene una identidad persistente y puede corresponder a una fila en la base de datos. Para las instancias desasociadas, Hibernate no garantiza la relación entre la identidad persistente y la identidad Java.

4.2.4 Cómo funciona Hibernate

El éxito de Hibernate reside en la simplicidad de sus conceptos centrales. En el corazón de cada interacción entre el código y la base de datos se centra en la sesión Hibernate. La figura 4.2.4 provee un resumen de cómo funciona Hibernate.

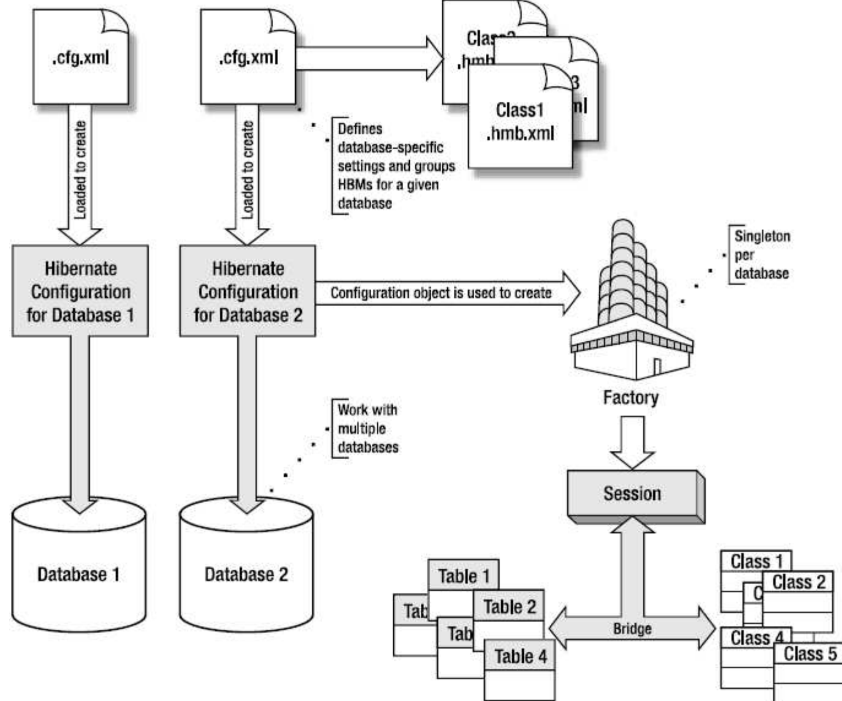


Fig. 4.2.4 Flujo Hibernate

4.2.5 La session Hibernate

La sesión Hibernate engloba el concepto de un servicio de persistencia (o administrador de persistencia) que puede ser usado para realizar sentencias SQL de inserción, actualización y borrado en instancias de una clase mapeada por Hibernate. En una herramienta ORM se realizan todas las sentencias mencionadas usando semántica orientada a objetos, lo cual es, que no se refiere directamente a las columnas o tablas, sino que se usan clases Java y propiedades del objeto. La **sesión** es un objeto ligero de vida-corta que es usado como puente enlazador durante una conversación entre la aplicación y la base de datos. La sesión envuelve la conexión JDBC o la fuente J2EE, y sirve como un caché de primer nivel para objetos persistentes ligados a la sesión.

4.2.6 La fábrica de la sesión

Hibernate requiere que se especifique la información requerida para conectarse a cada base de datos que será usada por la aplicación así como también qué clases están mapeadas para una base de datos.
Cada uno de estos archivos de configuración de especificaciones de la base de datos, y las clases mapeadas asociadas están compilados y se mantienen en la fábrica de la sesión `SessionFactory` la cual se usa para recuperar las sesiones Hibernate. La fábrica de la sesión es un objeto muy pesado que idealmente debería ser creado solo una vez (dado que es costoso y de operación lenta) y puesto a disponibilidad del código de la aplicación que necesite realizar operaciones de persistencia.
Cada `SessionFactory` se configura para trabajar con una cierta plataforma de base de datos usando uno de los dialectos provistos por Hibernate. Hibernate puede portar la aplicación de base de datos a otra diferente, simplemente cambiando unos pocos parámetros en un archivo XML.

4.2.7 El mapeo de objetos con Hibernate

Hibernate especifica cómo se recupera y almacena el estado de cada objeto en la base de datos vía un archivo de configuración XML. Los mapeos Hibernate se cargan, inicializan y son contenidos en el objeto SessionFactory. Cada mapeo especifica una variedad de parámetros relacionados al ciclo de vida de persistencia de las instancias de las clases mapeadas tales como:

- Mapeo de llaves primarias y generación de esquemas.
- Mapeo de propiedades del objeto hacia columnas de una tabla.
- Asociaciones.
- Configuración de caché.
- Sentencias SQL, llamadas a procedimientos remotos, queries parametrizados y más.

4.2.8 El ciclo de vida de la persistencia

Este siguiente tema de información acerca de Hibernate con el cual trabajé en mi estancia laboral es el entendimiento del ciclo de vida de un objeto perteneciente en Hibernate.

Son tres los posibles estados de un objeto mapeado en Hibernate. Entender estos estados y las acciones que causan las transiciones de estado se convierte en un punto importante cuando se lidia con problemas más complicados con Hibernate. La siguiente figura 4.2.8 muestra la transición de estados de un objeto mapeado con Hibernate.

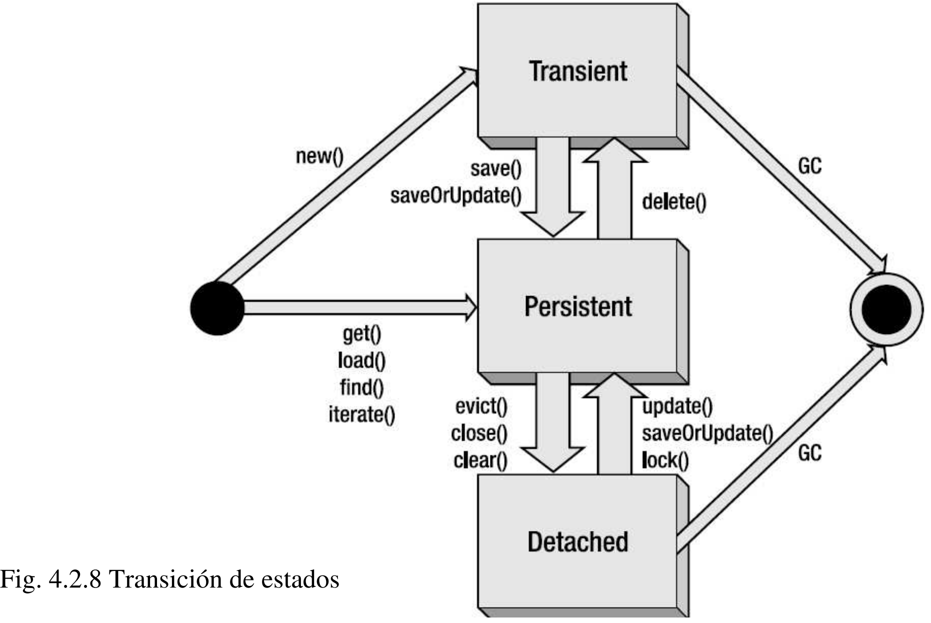


Fig. 4.2.8 Transición de estados

En el estado **transitorio** (`transient`), el objeto no está asociado con una tabla de la base de datos. Esto es, que su estado no ha sido guardado en una tabla, y el objeto no tiene asociada una identidad de base de datos (no se ha asignado una llave primaria). Los objetos en el estado transitorio no son transaccionales, lo que significa que no participan en el contexto de alguna transacción limitada dentro de la sesión Hibernate. Después de una invocación exitosa de los métodos `save` o `saveOrUpdate` un objeto cesa de ser transitorio y se convierte en persistente. El método `delete` (sentencia SQL delete) produce el efecto inverso haciendo que un objeto persistente se vuelva transitorio.

Los objetos **persistentes** (`persistent`) son objetos con un identificador de base de datos. (Si se les ha asignado una llave primaria pero no han sido guardados en la base de datos, se les considera que están en un estado “nuevo”.) Los objetos persistentes son transaccionales, lo que significa que participan en las transacciones asociadas con la sesión (al final de la transacción el estado del objeto será sincronizado con la base de datos.)

Un objeto persistente que no está en el caché de la sesión (asociado a la sesión) se convierte en un objeto **desasociado** (`detached`). Esto ocurre después de que se completa una transacción, cuando se cierra la sesión, se limpia o si el objeto es explícitamente desalojado del caché de la sesión.

4.2.9 El funcionamiento del enfoque POJO (Objeto Java Plano)

Una de las ventajas de la herramienta Hibernate ORM de POJO-Céntrico es que realmente ayuda a separar contextos. Cuando se desarrolla es posible trabajar en el contexto de los POJOs sin preocuparse mucho acerca de las complejidades que agrega las características de la aplicación (como la distribución, partición de las capas, entre otras.) Como se verá con Hibernate, se puede concentrar en el desarrollo de un modelo sólido de POJO, diseñar y probar las necesidades de persistencia del modelo antes de implementarlo en una aplicación J2EE.

4.2.10 Obtener Hibernate

Se puede descargar Hibernate de la página www.hibernate.org en la distribución binaria o fuente. La cual se puede descomprimir y guardad en cualquier ubicación, por ejemplo c:\java\hibernate. La distribución de Hibernate contiene el JAR Hibernate (hibernate3.jar) además las dependencias extra.

4.2.11 Configuración del ambiente POJO

En la tabla 4.2.11 se enlistan los archivos JAR que se necesitan para el siguiente ejemplo. Esos archivos JAR se pueden encontrar en la distribución Hibernate, se ubicarán bajo el directorio de la distribución en el directorio **lib**. Se agregarán los JARs a la librería de la aplicación. Para hacer esto se crea un directorio con el nombre hibernate en el directorio lib de la aplicación MemoriasHibernate y se copian los archivos JAR (hibernate3.jar, dom4j-1.6.1.jar, cglib-2.1.1.jar, asm.jar, ehcache-1.1.jar) listados en la figura. Dado que las librerías Yakarta Commons se comparten por algunas otras librerías y herramientas, se decidió para este ejercicio ubicarlas en su propio conjunto de directorios. Los archivos commons-logging-1.0.4 y commons-lang-2.0.jar se ubican el los directorios lib/commons-logging y lib/commons-lang respectivamente. Finalmente, se necesita agregar el archivo mysql-connector-java-3.0.14-production-bin.jar, el cual contiene el controlador JDBC para la instancia MySQL. La figura 4.2.11 muestra la estructura del directorio lib del proyecto en eclipse MemoriasHibernate después de las configuraciones necesarias para trabajar con Hibernate.

Tabla 4.2.11 Dependencias Hibernate

Nombre del archivo	**Descripción**
hibernate3.jar	Clases principales de Hibernate
dom4j-1.6.1.jar	Librería XML Dom4j (usada para cargas todas las definiciones XML tales como los archivos .cfg.xml y .hbm.xml).
antlr-2.7.6.jar	Es una herramienta que provee un framework para construir intérpretes compiladores y traductores de descripciones gramaticales contenidas en una variedad de lenguajes (entre ellos java).

asm-3.2.jar	Framework de manipulación de código-byte Java.
cglib-2.2.jar	Librería de generación de código en tiempo de ejecución.
commons-collections-3.1.jar	Ayuda a construir estructuras de datos que aceleran el desarrollo de la mayoría de las aplicaciones Java.
ehcache-1.2.3.jar	Implementación de caché simple no distribuido.
slf4j-api-1.5.8-sources.jar	Para abstraer la utilización de frameworks logging como Log4j.
mysql-connector-java-5.1.7-bin.jar	Controlador JDBC usado para utilizar MySQL.
commons-logging-1.0.4.jar	Clases comunes de Jakarta usadas como un envolvente ligero.
commons-lang-2.4.jar	Usado para simplificar la implementación POJO.
log4j-1.2.1.5.jar	Sirve para mandar mensajes a la consola que ayudan a debuguear una aplicación.

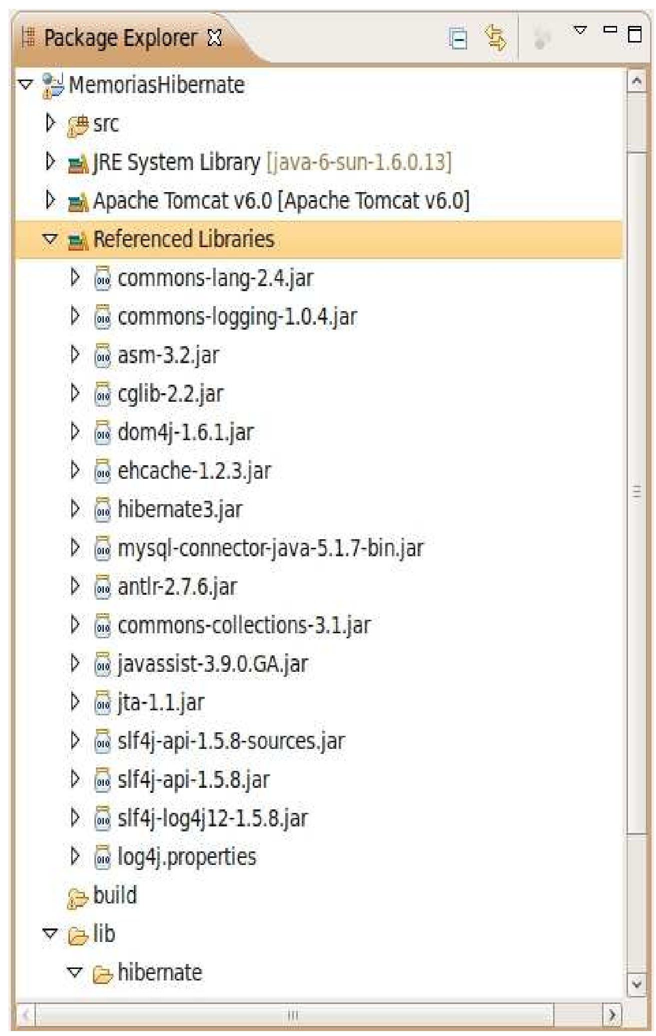


Fig. 4.2.11 Librerías Hibernate.

4.2.12 Configuración en Eclipse

Para realizar los ejemplos de este capítulo se necesita agregar los JARs Hibernate a la ruta de construcción del proyecto MemoriasHibernate. Para hacer eso, es necesario ir a las Propiedades del Proyecto seleccionando el nodo del proyecto en el explorador de paquetes de Eclipse, seleccionar las propiedades (dando clic-derecho en el nodo o vía menú Project – Properties) y seleccionar Java Build Path. Presionar clic en el botón Add Jars para navegar por el directorio de librerías y seleccionar los JARs que se muestran en la figura 4.2.12.

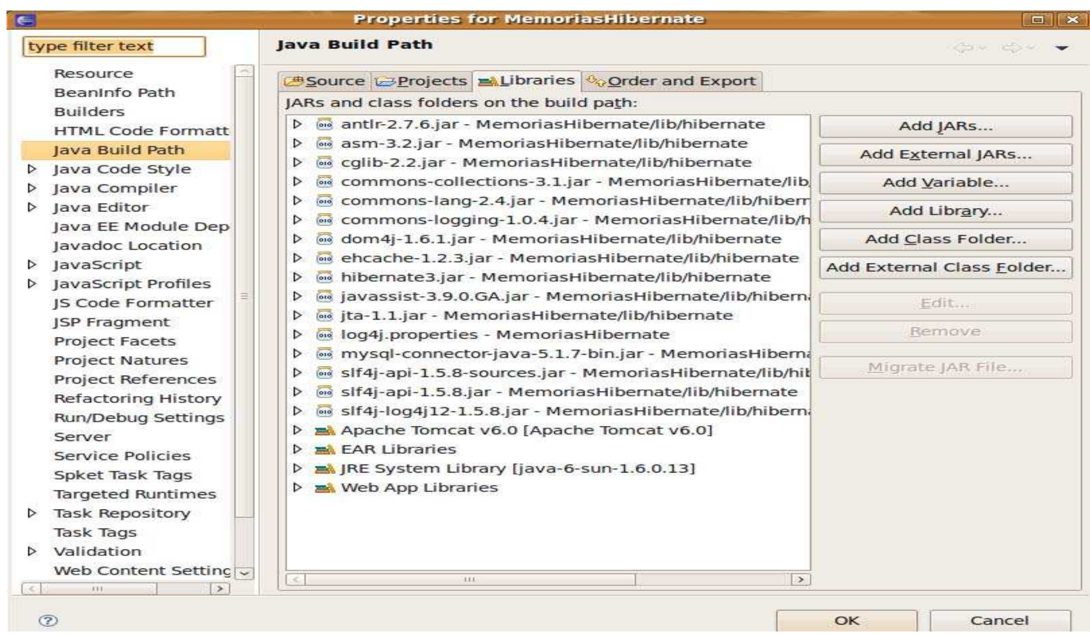


Fig. 4.2.12 Java build path en eclipse.

4.2.13 Ejemplo de un mapeo con Hibernate

El POJO **Direccion** sigue las conocidas convenciones de una JavaBean, campos privados expuestos vía getters y setters. Esa es la parte sencilla, de la importancia de cómo Hibernate manipula los objetos persistentes mediante la implementación de los métodos equals y hasCode. La implementación de dichos métodos no es un requerimiento específico de Hibernate, pero debería ser parte del negocio de cualquier objeto Java. En el caso del POJO Direccion la combinación de las propiedades calle, estado, codigoPost, ciudad, numero identifican una única dirección como se muestra en el siguiente código.

4.2.13.1 Código POJO Direccion

```
package com.memorias.hibernate.pojo;

import java.io.Serializable;

import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
```

```java
import org.apache.commons.lang.builder.ToStringBuilder;

public class Direccion implements Serializable {
    // Llave primaria
    private Integer cveDir;
    // columnas
    private String calle;
    private String estado;
    private String codigoPost;
    private String ciudad;
    private String numero;
    // constructores
    public Direccion () {}

    // getters y setters
     public Integer getCveDir() {
          return cveDir;
     }
     public void setCveDir(Integer cveDir) {
          this.cveDir = cveDir;
     }
     public String getCalle() {
          return calle;
     }
     public void setCalle(String calle) {
          this.calle = calle;
     }
     public String getEstado() {
          return estado;
     }
     public void setEstado(String estado) {
          this.estado = estado;
     }
     public String getCodigoPost() {
          return codigoPost;
     }
     public void setCodigoPost(String codigoPost) {
          this.codigoPost = codigoPost;
     }
     public String getCiudad() {
          return ciudad;
     }
     public void setCiudad(String ciudad) {
          this.ciudad = ciudad;
     }
     public String getNumero() {
          return numero;
```

```
    }
    public void setNumero(String numero) {
         this.numero = numero;
    }
      public boolean equals (Object object) {
          // short circuits
          if (object == null) return false;
          if (this == object) return true;
          if (!(object instanceof Direccion)) return false;
          final Direccion direccion = (Direccion) object;
          return new EqualsBuilder().
              append(calle, direccion.getCalle()).
              append(numero, direccion.getNumero()).
              append(ciudad, direccion.getCiudad()).
              append(estado, direccion.getEstado()).
              append(codigoPost, direccion.getCodigoPost()).
              isEquals();
      }
      public int hashCode () {
          // Seleccionar un valor aleatorio,
           // no-cero, impar.
          // Idealmente diferente para cada clase
          return new HashCodeBuilder(17, 37).
              append(calle).
              append(numero).
              append(ciudad).
              append(estado).
              append(codigoPost).
              toHashCode();
      }
      public String toString () {
          return new ToStringBuilder(this).
              append("Calle", calle).
              append("Numero", numero).
              append("Ciudad", ciudad).
              append("Estado", estado).
              append("Codigo Postal", codigoPost).
              toString();
      }
}
```

4.2.13.2 Mapeo de una clase con el archivo HBM.XML

Este es el mapeo para la entidad Direccion.java es un mapeo directo objeto-a-tabla sin asociaciones.

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping
    package="com.memorias.hibernate.pojo">
    <class name="Direccion" table="DIRECCION" >
        <id name="cveDir" column="cve_dir" type="integer">
            <generator class="identity" />
        </id>
        <property name="calle" column="calle" />
        <property name="estado" column="estado" />
        <property name="codigoPost" column="codigo_post" />
        <property name="ciudad" column="ciudad" />
        <property name="numero" column="numero" />
    </class>
</hibernate-mapping>
```

Examinemos el archivo HBM en detalle. Los mapeos Hibernate son documentos XML cuya raíz es el mapeo de elementos hibernate. Son típicamente nombrados como las clases a las que mapean, seguido de la extensión hbm.xml. En el caso de este ejemplo Direccion.hbm.xml.

El atributo de paquete (package) especifica donde buscar por el POJO “Direccion” en la ruta de las clases en tiempo de ejecución. Hibernate asumirá que la tabla que se está mapeando de la clase Direccion se llama de la misma forma, a menos que se le indique en el atributo tabla. El elemento **id** denota el mapeo de la base de datos a la columna de llave primaria en la tabla especificada. En este caso se mapea el campo Integer **Id** a la columna de base de datos **cve_dir**, con ello le decimos a Hibernate que use la estrategia “identity” para generar un nuevo valor para la llave primaria cada vez que se inserta un nuevo registro en la base de datos.

Los elementos **property** mapean las propiedades del POJO calle, estado, codigoPost, ciudad y numero a cada columna de la base de datos que se especifique en el atributo **column**. Hibernate supondrá que las propiedades son del tipo String en el POJO y las columnas de tipo VARCHAR en la base de datos. Se puede especificar más precisamente en el mapeo y proveer la longitud exacta y los tipos de datos de la clase en particular.

4.2.13.3 Creación de la tabla en MySQL

Ahora que se tiene el POJO Direccion y el mapeo correspondiente, solo es necesario crear una nueva tabla llamada DIRECCION en la base de datos.

4.2.13.4 Configuración de la base de datos MySQL

Una vez instalado el manejador de base de datos MySQL el cual se puede obtener de la página http://dev.mysql.com/downloads/, se realizan los siguientes pasos para crear la tabla DIRECCION en la base de datos de nombre MySQL_Hibernate.

```
root@unix-ubuntu:~# mysql -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 34
Server version: 5.0.75-0ubuntu10.2 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> create database MySQL_Hibernate;
Query OK, 1 row affected (0.03 sec)

mysql> use MySQL_Hibernate;
Database changed

mysql> CREATE TABLE DIRECCION (
    ->          cve_dir   INTEGER   NOT   NULL   PRIMARY   KEY
AUTO_INCREMENT,
    ->    calle VARCHAR(64),
    ->    numero VARCHAR(32),
    ->    ciudad VARCHAR(32),
    ->    estado CHAR(3),
    ->    codigo_post VARCHAR(10)
    -> );
Query OK, 0 rows affected (0.06 sec)

mysql>  grant  all  privileges  on  MySQL_Hibernate.*  to
'unix'@'localhost'  identified  by  'avr102894'  with  grant
option;
Query OK, 0 rows affected (0.04 sec)
```

4.2.13.5 Generación de la Llave Primaria

Nótese que se creó la llave primaria como un campo MySQL_AUTO_INCREMENT. Este es el caso de un mapeo en donde la propiedad objeto es opcional pero la propiedad de la columna de la tabla no. Es altamente recomendable que se cree el campo ID en el mapeo, ya que algunas características avanzadas de Hibernate, tales como la habilidad de usar objetos "**desasociados**" y los útiles métodos saveOrUpdate y merge del objeto Session, dependiendo de tener el identificador de base de datos reflejado en los POJOs. Se puede minimizar el imparto del diseño POJO usando inteligentemente los miembros de la clase Java cuando se declaran los identificadores de los campos.

En una herramienta ORM es importante saber el estado de persistencia del objeto. Al tener una llave primaria que puede ser nula, la llave primaria no primitiva hace el trabajo de reconocer si el objeto necesita ser guardado o actualizado.

4.2.13.6 Guardar un objeto

Ahora es necesario decirle a Hibernate como conectarse a la base de datos y cargar el archivo Direccion.hbm.xml en el SessionFactory. Para este ejemplo se hará directo en código java. Después se presentará el archivo de configuración XML.

En el código descrito a continuación se muestra cómo crear una configuración Hibernate en java, agregar una clase mapeada y recuperar una instancia del tipo SessionFactory de la configuración. Cuando se invoca al método addClass de la clase Configuration, pasando el objeto de la clase para Direccion (Direccion.class), Hibernate busca el archivo nombrado "Direccion.hbm.xml" en la ubicación de las clases (classpath). Es entonces que la forma sencilla de asegurar que Hibernate encontrará los mapeos es colocarlos en la misma ubicación que la clase que mapean. Una vez que se tiene la SessionFactory, trabajar con los objetos persistentes es tan simple como obtener una Session e invocar los métodos sobre ella. Reemplace el nombre de usuario y password con los requeridos para su base de datos.

Por ejemplo para crear un objeto Direccion y guardarlo en la base de datos se deben seguir los siguientes pasos:

5. Se crea un POJO y se asignan los valores.
6. Se crea una SessionFactory usada para crear una sesión (Session).
7. Una transacción Hibernate limitada a la sesión es comenzada.
8. Se guarda el objeto usando el método de sesión persist.
9. Se aplica un comit a la transacción.
10. Se cierra la sesión.

```
Address address = new Address();
...
Session session = null;
Transaction tx = null;
try {
    session = factory.openSession();
    tx = session.beginTransaction();
    session.persist(address);
    tx.commit();
} catch (Exception e) {
    if (tx != null) tx.rollback();
} finally {
    session.close();
}
```

El código anterior sigue el patrón que se puede usar cuando se interactúa con objetos mapeados en Hibernate en una aplicación simple. Detrás de escena la Sesión Hibernate solo

abre una conexión JDBC dada para la base de datos especificada en la configuración SessionFactory (esta conexión puede ser una nueva conexión o puede venir de un conjunto de conexiones existentes en Hibernate. Una vez que la sesión es abierta actúa como un caché de entidades de objetos mapeados, es decir, cualquier objeto que se obtenga o guarde vía Session estará en el caché de sesión. Entonces, si se carga el mismo objeto varias veces, en la primer llamada Hibernate se realiza una sentencia SQL para cargar el estado del objeto, pero en llamadas subsecuentes usando la misma Sesión, el objeto será obtenido del caché interno Session sin acceder a la base de datos.

Si el código se ejecuta exitosamente, la instancia Direccion que fue pasada al método persist tendrá ahora su campo ID con el identificador generado por la base de datos y será un nuevo renglón en la base de datos. El objeto Session de Hibernate provee muchos métodos que persisten el estado de un objeto en la base de datos, incluyendo persist, save, update y saveOrUpdate. Si el atributo de llave primaria de un objeto no está asignado. Hibernate detectará esto y generará automáticamente una llave primaria para el objeto basado en la estrategia seleccionada en el elemento ID.
Lo cual se traduce en un INSERT de SQL.

En el caso de un método saveOrUpdate, si la llave primaria es asignada entonces la operación se convierte en un UPDATE de SQL si el objeto existe y un INSERT SQL sino, mientras que en el caso de los métodos persist, save, update, la semántica está claramente definida; una llamada a persist o save siempre resultará en un INSERT de SQL mientras que una llamada a update resultará en un UPDATE SQL.

Otra importante pieza de información acerca del funcionamiento de Hibernate es que Hibernate interactúa con la base de datos en una forma transparente. Lo cual significa que no es posible ver sentencias SQL mientras son ejecutadas en la base de datos al momento de que el método persist se ejecuta pero posiblemente después. Esto es una técnica común empleada por las herramientas ORM que permite optimizaciones en tiempo de ejecución cuando se interactúa con la base de datos. Por ejemplo, las acciones en el código pueden resultar en muchas sentencias SQL UPDATE con las herramientas ORM se pueden combinar en un simple SQL UPDATE. En Hibernate, al momento en el cual la Session se sincroniza con la base de datos se refiere a un `flushing` (liberación). La clase Hibernate provee de un método explícito flush que puede ser invocado. Típicamente no es necesario liberar manualmente la Session, pero en escenarios complejos puede ser útil para propósitos de debugueo.
Finalmente, nótese que el código que realiza el guardado del objeto Direccion está envuelto en un bloque `try-cath-finally`. En la clausula `finally` se asegura que se invoque al método close de la sesión. Lo cual respalda que cualquier recurso (como conexiones de base de datos) se desvincule.

En la cláusula `catch` la transacción Hibernate se revierte en caso de que ocurra una excepción para asegurar la integridad de la operación.

4.2.13.7 Prueba simple de Hibernate para guardar un objeto en la base de datos

```
package com.memorias.negocio;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.memorias.hibernate.pojo.Direccion;

public class PruebaDireccion {
     public static void main(String[] args) {
             Configuration config = new Configuration().
             setProperty("hibernate.dialect",

     "org.hibernate.dialect.MySQLDialect").
             setProperty("hibernate.connection.driver_class",
                         "com.mysql.jdbc.Driver").
             setProperty("hibernate.connection.url",

     "jdbc:mysql://localhost/MySQL_Hibernate").
             setProperty("hibernate.connection.username",
     "unix").
             setProperty("hibernate.connection.password",
                         "avr102894").
             setProperty("hibernate.show_sql", "true");
             config.addClass(Direccion.class);
             SessionFactory factory =
config.buildSessionFactory();
             Direccion address = new Direccion();
             address.setCalle("Av. Universidad");
             address.setCiudad("Santiago de Queretaro");
             address.setNumero("220");
             address.setEstado("QRO");
             address.setCodigoPost("76800");
             Session session = null;
             Transaction tx = null;
             try {
                 session = factory.openSession();
                 tx = session.beginTransaction();
                 session.persist(address);
                 tx.commit();
             } catch (Exception e) {
                 if (tx != null) tx.rollback();
             } finally {
```

```
                session.close();
            }

        }
}
```

Ejecutando el ejemplo debería producir una salida en consola similar a la mostrada en el siguiente ejemplo. Primero, nótese que Hibernate ubica el mapeo de la clase Direccion en el archivo Adress.hbm.xml, establece la conexión al manejador de base de datos MySQL usando el dialecto Hibernate MySQL, establece también las transacciones JDBC en el mecanismo Hibernate transactions, y finalmente realiza una sentencia SQL INSERT (en consecuencia de la llamada directa al método persist.)

Salida en consola

```
INFO  Environment - Hibernate 3.3.2.GA
INFO  Environment - hibernate.properties not found
INFO  Environment - Bytecode provider name : javassist
INFO  Environment - using JDK 1.4 java.sql.Timestamp handling
INFO  Configuration - Reading mappings from resource:
com/memorias/hibernate/pojo/Direccion.hbm.xml
...
INFO  HbmBinder - Mapping class:
com.memorias.hibernate.pojo.Direccion -> DIRECCION
...
INFO  DriverManagerConnectionProvider - using driver:
com.mysql.jdbc.Driver at URL:
jdbc:mysql://localhost/MySQL_Hibernate
INFO  DriverManagerConnectionProvider - connection
properties: {user=unix, password=avr102894}
DEBUG DriverManagerConnectionProvider - opening new JDBC
connection
DEBUG DriverManagerConnectionProvider - created connection
to: jdbc:mysql://localhost/MySQL_Hibernate, Isolation Level:
4
INFO  SettingsFactory - JDBC driver: MySQL-AB JDBC Driver,
version: mysql-connector-java-5.1.7 ( Revision:
${svn.Revision} )
INFO  Dialect - Using dialect:
org.hibernate.dialect.MySQLDialect
INFO  TransactionFactoryFactory - Using default transaction
strategy (direct JDBC transactions)
...
INFO  SessionFactoryImpl - building session factory
...
Hibernate: insert into DIRECCION (calle, estado, codigo_post,
ciudad, numero) values (?, ?, ?, ?, ?)
```

```
DEBUG Printer - listing entities:
DEBUG Printer -
com.memorias.hibernate.pojo.Direccion{ciudad=Santiago de
Queretaro, estado=QRO, cveDir=1, codigoPost=76800, calle=Av.
Universidad, numero=220}
DEBUG JDBCTransaction - committed JDBC Connection
```

Si se usa la herramienta de líneas de comando SQL para revisar el contenido de la tabla, debería verse el registro insertado como se muestra en la figura 4.2.13.7:

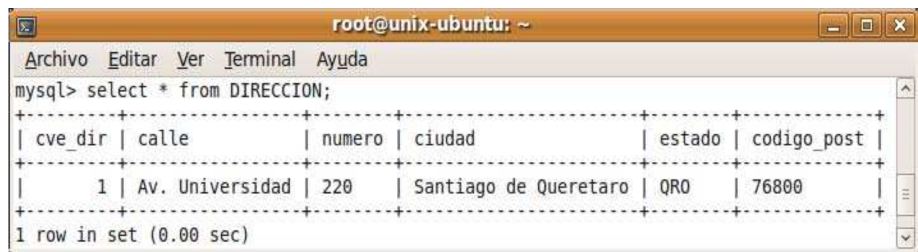


Figura 4.2.13.7 Entidad persistida.

4.2.13.8 El archivo de configuración hibernate.cfg.xml

La mayoría de las aplicaciones usan el formato XML para sus configuraciones, y es la forma recomendada para configurar una aplicación. Para una simple aplicación de base de datos, el archivo es nombrado como hibernate.cfg.xml. El siguiente archivo de configuración realiza el equivalente de lo que se hizo en el ejercicio anterior en código Java. El archivo muestra el mapeo para la aplicación. Se puede colocar en la raíz del proyecto.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
  <!-- ================================================ -->
  <!-- SessionFactory usada para probar MySQL_Hibernate  -->
  <!-- ================================================ -->
  <session-factory>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/MySQL_Hibernate</property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.username">
     unix</property>
    <property name="hibernate.connection.password">
     avr102894</property>
```

```
    <property
    name="dialect">org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.show_sql">true</property>
    <!-- =============================================== -->
    <!-- Mapeos -->
    <!-- =============================================== -->
    <mapping
resource="com/memorias/hibernate/pojo/Direccion.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Para cargar el XML de configuración se necesita código parecido al siguiente:

```
public class PruebaDireccion {
     public static void main(String[] args) {

          File configFile = new File("hibernate.cfg.xml");
          Configuration configuration = new Configuration().
               configure(configFile);
          SessionFactory factory =  configuration.
               buildSessionFactory();
          Session session = null;
          Transaction tx = null;
          try {
               Direccion address = new Direccion();
               address.setCalle("Av. Zaragoza");
               address.setCiudad("Santiago de Queretaro");
               address.setNumero("550");
               address.setEstado("QRO");
               address.setCodigoPost("76880");
               session = factory.openSession();
               tx = session.beginTransaction();
               session.persist(address);
               tx.commit();
          } catch (Exception e) {
               if (tx != null)
                    tx.rollback();
          } finally {
               session.close();
          }

     }
}
```

4.2.14 Mapeo Hibernate para relaciones entre tablas.

En una aplicación real no habrá solo mapeos simples sino situaciones complejas que envuelvan conceptos avanzados de Mapeo Relacional de Objetos (ORM). A continuación se mostrará un ejemplo de un modelo de base de datos que implica varias tablas y diferentes relaciones entre si. Se trata de un sistema para registrar conferencias las cuales pueden tener un o más temas, una sede asignada y su respectiva dirección. La siguiente figura XXX muestra claramente el modelo relacional de entidades.

Código MySQL para crear las tablas SEDE, CONFERENCIA, TEMA

```
CREATE TABLE SEDE (
cve_sede INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
nombre VARCHAR(32),
tel VARCHAR(12),
fax VARCHAR(12),
fk_cve_dir INTEGER NOT NULL,
CONSTRAINT SEDE_DIRECCION_FK FOREIGN KEY(fk_cve_dir)
REFERENCES DIRECCION(cve_dir),
CONSTRAINT UNIQUE_NOMBRE_SEDE UNIQUE(nombre)
);

CREATE TABLE CONFERENCIA (
cve_conf INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
nombre VARCHAR(64) NOT NULL,
descripcion LONGTEXT NOT NULL,
fech_ini DATETIME NOT NULL,
fech_fin DATETIME NOT NULL,
fk_cve_sede INTEGER,
CONSTRAINT CONFERENCIA_SEDE_FK FOREIGN KEY(fk_cve_sede)
REFERENCES SEDE(cve_sede),
CONSTRAINT UNIQUE_NOMBRE_CONFERENCIA UNIQUE(nombre)
);

CREATE TABLE TEMA (
cve_tema INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
titulo VARCHAR(32) NOT NULL,
subtitulo VARCHAR(32),
descripcion LONGTEXT,
fk_cve_conf INTEGER NOT NULL,
CONSTRAINT TEMA_CONFERENCIA_FK FOREIGN KEY(fk_cve_conf)
REFERENCES CONFERENCIA(cve_conf),
CONSTRAINT UNIQUE_TITULO_TEMA UNIQUE(titulo)
);
```

4.2.14.1 POJO para la entidad CONFERENCIA

Como se puede ver en el siguiente código, la clase Conferencia.java tiene una propiedad de java.util.Set de tipo Tema y otra de tipo Sede además otros campos que representa el nombre, descripción y otros datos.

```
package com.memorias.hibernate.pojo;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

public class Conferencia implements Serializable{
//Llave primaria.
private Integer cveConf;
//Columnas.
private String nombre;
private String descripcion;
private Date fechaIni;
private Date fechaFin;
//Relación muchos-a-uno.
private Sede sede;
//Colecciones uno-a-muchos.
private Set <Tema> temas;
//Constructor
public Conferencia() {
}
// Getters y setters
public Integer getCveConf() {
return cveConf;
}

public void setCveConf(Integer cveConf) {
this.cveConf = cveConf;
}

public String getNombre() {
return nombre;
}

public void setNombre(String nombre) {
this.nombre = nombre;
}

public String getDescripcion() {
return descripcion;
```

```
}

public void setDescripcion(String descripcion) {
this.descripcion = descripcion;
}

public Date getFechaIni() {
return fechaIni;
}

public void setFechaIni(Date fechaIni) {
this.fechaIni = fechaIni;
}

public Date getFechaFin() {
return fechaFin;
}

public void setFechaFin(Date fechaFin) {
this.fechaFin = fechaFin;
}

public Sede getSede() {
return sede;
}

public void setSede(Sede sede) {
this.sede = sede;
}

public Set<Tema> getTemas() {
return temas;
}

public void setTemas(Set<Tema> temas) {
this.temas = temas;
}
public void addTema(Tema tema) {
if (null == this.temas)
this.temas = new HashSet<Tema>();
tema.setConferencia(this);
temas.add(tema);
}
}
```

4.2.14.2 Mapeo Uno-A-Muchos usando una colección de tipo Set

Primero es necesario mapear la relación uno-a-muchos entre los POJOs Conferencia y Tema. En seguida se muestra el archivo de configuración `conferencia.hbm.xml` con los mapeos de las propiedades-a-columnas:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class table="CONFERENCIA"
name="com.memorias.hibernate.pojo.Conferencia" >
<id column="cve_conf" name="cveConf" type="integer">
<generator class="identity"/>
</id>
<property name="nombre" length="64" not-null="true"
type="string"/>
<property name="descripcion" not-null="true" type="string"/>
<property column="fech_ini" name="fechaIni"
length="19" not-null="true" type="timestamp"/>
<property column="fech_fin" name="fechaFin"
length="19" not-null="true" type="timestamp"/>
<set inverse="true" name="Temas" cascade="all" lazy="false">
<key column="fk_cve_conf"/>
<one-to-many class="com.memorias.hibernate.pojo.Tema"/>
</set>
<many-to-one class="com.memorias.hibernate.pojo.Sede"
name="sede" not-null="true" cascade="persist,save-update">
<column name="fk_cve_sede" />
</many-to-one>
</class>
</hibernate-mapping>
```

Para mapear la relación uno-a-muchos entre los objetos Conferencia y Tema, se usa la sentencia `<set></set>`. El elemento java.util.Set en el POJO, permite trabajar con una colección simple Java usando semántica típica Java sin preocuparse si la clase es una clase persistente o no. La propiedad `temas` es mapeada en la clase Tema (en el mismo paquete) la cual es referencia a la tabla TEMA. La columna de llave primaria le dice a Hibernate que hay un constraint de llave foránea en la tabla TEMA hacia la tabla CONFERENCIA vía la columna `fk_cve_conf`:

```
CONSTRAINT TEMA_CONFERENCIA_FK FOREIGN KEY(fk_cve_conf)
REFERENCES CONFERENCIA(cve_conf),
```

Se requiere hacer una relación bi-direccional uno-a-muchos entre Conferencia y Tema. En el POJO se representa con la colección `java.util.Set` "temas" para el lado "uno" de la relación y con la propiedad simple Conferencia para el lado "muchos". Este tipo de asociación bi-direccional puede ser muy conveniente, y permitirá, dado un `Tema`, determinar a que conferencia pertenece accesando al objeto `Conferencia` vía su setter `getConferencia()`. El atributo `inverse="true"` mostrado en el elemento set le dice a Hibernate que ese es una relación bidireccional o inversa.

Nótese que en la implementación de la clase Conferencia, se incluyó un método conveniente, el método `addTema`(), para asignar temas a una conferencia dada. Nótese que el método `addTema` mantiene ambos el final de la relación agregando el objeto `Tema` en la colección `set` de temas en el objeto `Conferencia` y además asignando la propiedad conferencia en el `Tema` que se agregó. El atributo "lazy" en el mapeo, le dice a Hibernate que cargue los objetos Tema asociados al mismo tiempo de cargar los datos del objeto Conferencia.

En la clase Conferencia se creó una asociación para la clase Tema en la forma de una colección Set. Para completar el otro lado de la asociación entre conferencias y temas, se necesita agregar una asociación a la clase Tema lo que permitirá recuperar el objeto conferencia asociado. La identidad de objeto conferencia debe corresponder al valor de la columna fk_cve_conf. Se utiliza el elemento `<many-to-one> </many-to-one>` para proveer tal enlace, a continuación la implantación de la configuración.

4.2.14.3 POJO para la clase Tema.

```
package com.memorias.hibernate.pojo;

import java.io.Serializable;

public class Tema implements Serializable{
//Llave primaria
private Integer cveTema;
//Columnas Tema.
private String titulo;
private String subtitulo;
private String descripcion;
//Temas que perteneces a una conferencia.
private Conferencia conferencia;

//Constructor.
public Tema() {
}

//getters y setters
public Integer getCveTema() {
     return cveTema;
```

```
}

public void setCveTema(Integer cveTema) {
     this.cveTema = cveTema;
}

public String getTitulo() {
     return titulo;
}

public void setTitulo(String titulo) {
     this.titulo = titulo;
}

public String getSubtitulo() {
     return subtitulo;
}

public void setSubtitulo(String subtitulo) {
     this.subtitulo = subtitulo;
}

public String getDescripcion() {
     return descripcion;
}

public void setDescripcion(String descripcion) {
     this.descripcion = descripcion;
}

public Conferencia getConferencia() {
     return conferencia;
}

public void setConferencia(Conferencia conferencia) {
     this.conferencia = conferencia;
}
}
```

4.2.14.4 Configuración HBM para la entidad Tema

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class table="TEMA" name="com.memorias.hibernate.pojo.Tema" >
```

```
<id column="cve_tema" name="cveTema" type="integer">
<generator class="identity"/>
</id>
<property name="titulo" length="32" not-null="true"
type="string"/>
<property name="subtitulo" length="32" type="string"/>
<property name="descripcion" type="string"/>
<many-to-one class="com.memorias.hibernate.pojo.Conferencia"
name="conferencia" not-null="true">
<column name="fk_cve_conf" />
</many-to-one>
</class>
</hibernate-mapping>
```

4.2.14.5 Mapeo Mucho-A-Uno

En el ejemplo previo se tuvo una relación donde uno o más temas pertenecían a una conferencia, lo cuál es una relación padre-hijo debido a que no se necesita la existencia de temas sin su propia conferencia. En el caso de una conferencia y una sede, se tiene una asociación donde una sede puede existir en la base de datos sin una conferencia asociada y lo mismo aplica para una conferencia (ya que se podría crear todo el contenido de las conferencias en primera instancia antes de escoger una sede apropiada.), y obviamente muchas conferencias pueden compartir el mismo valor (muy probablemente no al mismo tiempo).
Para satisfacer esta relación se puede usar un mapeo muchos-a-uno en el archivo `conferencia.hbm.xml` se especifica con el elemento </many-to-one>.

Dado que no se desea eliminar sedes cuando una conferencia es eliminada, pero es necesario actualizar o insertar la información de la sede en la base de datos cuando actualizamos o creamos una conferencia, por ello se asigna un atributo `cascada = "persist, save-update"`. Luego entonces en cada llamada a `persist`, `save` o `update` contra un objeto Conferencia, Hibernate determinará si el objeto Sede asociado necesita ser guardado o actualizado.

4.2.14.6 POJO para la entidad Sede

```
package com.memorias.hibernate.pojo;

import java.io.Serializable;

public class Sede implements Serializable{
private Integer cveSede;
private String nombre;
private String tel;
private String fax;
//Constructor.
```

```java
public Sede() {
}
//Getters y setters.
public Integer getCveSede() {
return cveSede;
}
public void setCveSede(Integer cveSede) {
this.cveSede = cveSede;
}
public String getNombre() {
return nombre;
}
public void setNombre(String nombre) {
this.nombre = nombre;
}
public String getTel() {
return tel;
}
public void setTel(String tel) {
this.tel = tel;
}
public String getFax() {
return fax;
}
public void setFax(String fax) {
this.fax = fax;
}
}
```

4.2.14.7 Configuración HBM para la entidad Sede

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class table="SEDE" name="com.memorias.hibernate.pojo.Sede" >
<id column="cve_sede" name="cveSede" type="integer">
<generator class="identity"/>
</id>
<property name="nombre" length="32" type="string"/>
<property name="tel" length="12" type="string"/>
<property name="fax" length="12" type="string"/>
</class>
</hibernate-mapping>
```

4.2.14.8 Prueba Java para comprobar el mapeo.

El código que a continuación se muestra crea un objeto Conferencia con sus Temas asociados y asigna una sede a la conferencia, después guarda la conferencia en la base de datos.

```
package com.memorias.negocio;

import java.io.File;
import java.util.Date;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import com.memorias.hibernate.pojo.Conferencia;
import com.memorias.hibernate.pojo.Sede;
import com.memorias.hibernate.pojo.Tema;

public class MapeoOneToMany {

public static void main(String[] args) {
     File configFile = new File("hibernate.cfg.xml");
     Configuration configuration = new
     Configuration().configure(configFile);
     SessionFactory factory =
     configuration.buildSessionFactory();
     Session session = null;
     Transaction tx = null;
     try {
          // Creacion de una conferencia.
          Conferencia conferencia = new Conferencia();
          conferencia.setDescripcion("Mapeo de entidades con
          Hibernate3");
          conferencia.setFechaFin(new Date());
          conferencia.setNombre("Memorias de Trabajo");
          conferencia.setFechaIni(new Date());
          //Creación de una sede.
          Sede sede = new Sede();
          sede.setFax("Fax");
          sede.setNombre("Auditorio Pablo Neruda");
          sede.setTel("4422335588");
          //Se asigna una sede a la conferencia.
          conferencia.setSede(sede);
          // create some tracks
          Tema tema1 = new Tema();
          tema1.setDescripcion(
```

```
        "Como mapear objetos con Hinernate");
        tema1.setSubtitulo("Java y Hibernate");
        tema1.setTitulo("JSE");
        Tema tema2 = new Tema();
        tema2.setDescripcion(
        "Configurar Hibernate desde cero");
        tema2.setSubtitulo("Hinernate3");
        tema2.setTitulo("JEE");
        // Agregar temas a la conferencia.
        conferencia.addTema(tema1);
        conferencia.addTema(tema2);
        // Guardar la conferencia
        session = factory.openSession();
        tx = session.beginTransaction();
        session.persist(conferencia);
        tx.commit();
    } catch (Exception e) {
        if (tx != null) {
            tx.rollback();
        }
    } finally {
        session.close();
      }
    }
}
```

Lo cual produciría una salida en consola similar a la que se muestra:

```
Hibernate: insert into SEDE (nombre, tel, fax) values (?, ?,
?)
Hibernate: insert into CONFERENCIA (nombre, descripcion,
fech_ini, fech_fin, fk_cve_sede) values (?, ?, ?, ?, ?)
Hibernate: insert into TEMA (titulo, subtitulo, descripcion,
fk_cve_conf) values (?, ?, ?, ?)
Hibernate: insert into TEMA (titulo, subtitulo, descripcion,
fk_cve_conf) values (?, ?, ?, ?)
```

Como se puede ver del ejemplo de salida, Hibernate produjo INSERTS SQL, uno para el objeto Conferencia, dos para los objetos Tema y otro más para la Sede. Si se observa de vuelta en el mapeo `conferencia.hbm.xml` se puede observar que hay un atributo `cascade` con un valor asignado all. En Hibernate cada asociación puede tener un estilo en cascada. La opción en cascada le dice a Hibernate como tratar los objetos asociados en el contexto de las operaciones de persistencia. Por ejemplo, si se le asigna el valor de cascada `save-update` en una asociación de uno-a-muchos, Hibernate determinará si alguno de los objetos asociados se encuentra modificado y generará los UPDATES SQL para dichos objetos y de manera similar generará INSERTS SQL para objetos nuevos. Existen otros valores para el atributo como: `none`, `persist`, `merge`, `delete`, `evict`, `replicate`,

lock, refresh, delete-orphan y all se pueden combinar separados por coma y están relacionados con las operaciones disponibles por la clase Session.

4.2.15 El lenguaje Hibernate de Sentencias SQL (Hibernate Query Language)

HQL es un lenguaje de dialecto orientado a objetos de la familia relacional del lenguaje SQL. Es un lenguaje poderoso, elegante y fácil de aprender si se tiene conocimiento de SQL. HQL es usado comúnmente para consultar objetos y no para actualizar, insertar o borrar datos. Sin embargo, HQL soporta operaciones directas en volumen para actualizar, borrar e insertar si se requiere de efectuar operaciones en masa. La mayor parte del tiempo, solo se necesita consultar objetos de una clase en particular y restringir por las propiedades de esa clase. Por ejemplo, el siguiente query regresa las *direcciones* filtradas por la columna *ciudad.*

```
// Código para abrir la sesión Hibernate
Session session = null;
File configFile = new File("hibernate.cfg.xml");
Configuration configuration = new
Configuration().configure(configFile);
SessionFactory factory = configuration.buildSessionFactory();
session = factory.openSession();

Query q = session.createQuery("from Direccion d where
d.ciudad = :columnaCiudad");
q.setString("columnaCiudad", "Santiago de Queretaro");
```

Donde:

Query: Una consulta de base de datos puede ser creada en el lenguaje Hibernate (HQL) o en SQL estándar. Esta interfaz permite crear consultas, agregar parámetros al query y ejecutar consultas de diversas maneras.

setString: Método para pasar parámetros a la consulta.

session: Es el objeto que guarda la sesión Hibernate.

HQL soporta lo siguiente:

1. Tiene la habilidad para aplicar restricciones a propiedades de objetos asociados relacionados por referencia o mantenerlos en colecciones (para navegar el grafo de objetos usando SQL).
2. La habilidad para recuperar solo propiedades de una o muchas entidades, sin el costo de cargar la entidad completa en un contexto transaccional. Lo que es llamado proyección (projection).
3. La habilidad para ordenar el resultado de una consulta.
4. La habilidad de paginar el resultado de la consulta.
5. Utilizar cláusulas como group by, having y funciones como sum, min, max, etc.

6. Utilizar uniones (joins) para recuperar múltiples objetos por renglón.
7. La habilidad para llamar funciones definidas en SQL estándar.
8. Formar subqueries (consultas anidadas).

4.2.15.1 Ejecutando consultas

La interfaz Query define diversos métodos para controlar la ejecución de una consulta SQL. En adición, Query provee métodos para asociar valores concretos a parámetros en una consulta. Para ejecutar consultas en una aplicación, se necesita obtener una instancia de una de las interfaces mencionadas usando el objeto Session.

4.2.15.2 Las interfaces de Query

Para crear una nueva instancia de Query, se llama a `createQuery()` o `createSQLQuery()`. El método `createQuery()` prepara una consulta HQL:

```
Query hqlQuery = session.createQuery("from Direccion");
```

El método `createSQLQuery()` es usado para crear una consulta SQL, usando una sintaxis nativa del manejador de base de datos especificado.

```
Query sqlQuery = session.createSQLQuery( "select * from
DIRECCION");
```

En ambos casos, Hibernate regresa una nueva instancia del objeto `Query` que puede ser usada para especificar exactamente cómo debería ser ejecutada una consulta en particular.

4.2.15.3 Paginando los resultados

La paginación es una técnica comúnmente usada. Los usuarios pudieran ver el resultado de su petición de búsqueda como una página. Esta página muestra solo un subconjunto limitado (digamos 10 elementos) a la vez, y los usuarios pueden navegar a la siguiente y previa página manualmente. La interfaz Query soporta paginación de las consultas de la forma:

```
Query query = session.createQuery("from Direccion d order by
d.ciudad asc");
query.setFirstResult(0);
query.setMaxResults(10);
```

La llamada a `setMaxResults(10)` limita el resultado de la consulta a los primeros 10 objetos seleccionados de la base de datos.

4.2.15.4 Uso de consultas nombradas (named queries)

No es muy conveniente ver cadenas HQL literalmente diseminadas por todo el código Java a menos que sea necesario. Hibernate permite externalizar los strings de consulta al mapeo de metadatos, usando una técnica llamada **named queries**. Esto permite almacenar y encapsular todos los queries relacionados a una clase persistente en particular (o conjunto de clases) con los otros meta datos de la clase en un archivo de mapeo XML. Se usa el nombre del query para llamarlo desde la aplicación.
El método `getNamedQuery()` obtiene una instancia de `Query` para relacionar una consulta nombrada:

```
Query q = session.getNamedQuery("buscarElementosPorEstado");
          q.setString("ciudad", "%Queretaro%");
        .list();
```

En este ejemplo, ejecutamos el query nombrado buscarElementosPorEstado después de especificar los strings de argumentos a los parámetros nombrados. El query nombrado se define en el mapeo XML, en este caso Direccion.hbm.xml archivo mapeado en un ejemplo anterior, usando el elemento `<query>`:

```
<query name="buscarElementosPorEstado"><![CDATA[
        from Direccion d where d.ciudad like
:cuidad]]></query>
```

Las consultas nombradas no necesitan ser strings HQL; pueden ser incluso queries SQL, y el código Java no necesita saber la diferencia:

```
<sql-query name="buscarElementosPorEstado"><![CDATA[
        select {d.*} from DIRECCION {d} where ciudad like
:ciudad]]>
<return alias="d" class="Direccion"/>
</sql-query>
```

4.2.15.5 Usando alias

Usualmente, cuando se consulta una clase usando HQL, es necesario asignar un alias a la clase consultada para usarle como referencia en otras partes del query:

from Direccion as d

La palabra reservada as es siempre opcional. El equivalente es algo como:

from Direccion d

4.2.15.6 Operadores de comparación

HQL soporta los operadores básicos como en SQL: =, <>, <, >, >=, <=, between, not between, in and not in por ejemplo:

from Direccion d where d.numero between 200 and 2000
from Direccion d where d.numero > 100
from Direccion d where d.ciudad in (“Queretaro”, “Guanajuato”)

4.2.15.7 Búsqueda de patrones

El operador like permite búsquedas de patrones, donde los operadores % y _, funcionan igual que en SQL:

from Direccion d where d.ciudad like “%Quere%”

4.2.15.8 Usando inicialización dinámica

Se puede definir una clase para representar cada renglón de los resultados y usar el constructor HQL select new:

```
<!-- Query nombrado que utiliza el constructor select new -->
<query name="buscarElementosPorEstado"><![CDATA[select new
com.memorias.hibernate.pojo.ColumnasSeleccionadas(d.calle,
d.estado) from Direccion d]]></query>
```

La clase contenedora para la cláusula `SELECT` del query (**ColumnasSeleccionadas.java)** debe tener un constructor apropiado como se muestra en el siguiente código:

```
package com.memorias.hibernate.pojo;

public class ColumnasSeleccionadas {
     public ColumnasSeleccionadas(String calle, String
     estado) {
          super();
          this.calle = calle;
          this.estado = estado;
     }
     private String calle;
     private String estado;
     public String getCalle() {
          return calle;
     }
     public void setCalle(String calle) {
          this.calle = calle;
     }
```

```
	public String getEstado() {
		return estado;
	}
	public void setEstado(String estado) {
		this.estado = estado;
	}
}
```

La clase contenedora ColumnasSeleccionadas no tiene que ser una clase persistente, no tiene que estar mapeada a la base de datos o incluso saber de Hibernate. La clase ColumnasSeleccionadas es solo una clase de transferencia de datos, útil para generar reportes.

4.2.15.9 Llamadas a funciones SQL

Es posible además (para algunos dialectos SQL de Hibernate) llamar a funciones específicas SQL desde las clausulas `select` y `where`. Por ejemplo,

```
<query name="funcionMySQL"><![CDATA[select SYSDATE(),
d.ciudad from Direccion d]]> </query>
```

La técnica de las funciones de la base de datos en la cláusula `select` no está limitada desde luego a funciones dependientes de la base de datos que se utiliza, se pueden llamar funciones SQL estándar también:

```
<query name="funcionUpperMySQL"><![CDATA[select
upper(d.ciudad) from Direccion d]]> </query>
```

4.3 Capa de Persistencia

Esta capa es la responsable de obtener, actualizar y en su caso eliminar la información contenida en los objetos persistentes de dominio. Para esta capa se tiene pensado continuar utilizando el framework de Hibernate más las características que tiene Spring sobre este.
La primera ventaja que nos otorga es que la configuración de Hibernate se vuelve más sencilla ya que también entra como un objeto adicional dentro de la configuración de Spring. Con ello también Spring se encarga de administrar las sesiones de Hibernate sin que esta tenga que ser creada o declarada por el programador, mejora la administración de transacciones vía aspectos lo cual se traduce en que el desarrollador ya no manejara dentro de su código el control de transacciones sino por declaración o demarcación de la transacción.
Por otra parte, se pretende dar un mejor uso a los mapeos entre la orientación a objetos y las tablas de la base de datos, aprovechar aun más el manejo de cache de 1er y 2o nivel lo cual permite optimizar la operación y consumo de la base de datos al no realizar viajes innecesarios a esta.

Por último si en algún momento fuera necesario un cambio de base de datos solo impactaría en los parámetros de conexión y en mapeos. Es importante resaltar que el modelo de dominio, por ejemplo los POJOs persistentes, pueden viajar a otras capas de la aplicación evitando así la proliferación de objetos en memoria.
La capa de persistencia es accedida mediante el patrón de diseño Data Access Objet o DAO. Los DAO son inyectados mediante Spring a los servicios de negocio y aplicación que lo requieran. Cada entidad persistente debe contar con un DAO, quien encapsula el comportamiento de persistencia de cada entidad. Cada DAO heredará de una clase abstracta que ya implementa las operaciones generales de persistencia, siendo necesario definir nuevas operaciones para casos específicos de cada entidad.
Los DAOs que se desarrollen para la aplicación no requieren abrir sesiones e Hibernate ni transacciones. Esto se lleva a cabo con Spring implementando una clase Abstracta que encapsula funciones comunes. Para obtener la sesión de Hibernate, lo único que el desarrollador tiene que hacer es invocar un método para recuperar la sesión. Para que esto funcione, es necesario que esté configurado correctamente el contexto de Spring de tal forma que el DAO tenga una referencia al `SessionFactory` y que además se encuentre dentro del contexto de una transacción.

4.3.1 Manejo de la sesión en Hibernate

El manejo de la sesión de Hibernate debe de ser transparente para el desarrollador, es decir que el desarrollador no tiene que preocuparse por abrir o cerrar la sesión en el código, aunque es importante que el desarrollador entienda el manejo para que pueda probar el código y resolver problemas que se puedan presentar.
Los DAO de la aplicación, al extender de un DAO base abstracto y mediante configuración de Spring que le inyecta una referencia al `SessionFactory` de Hibernate, no tienen que hace más que invocar al método que obtiene la sesión de la clase padre.
A pesar de abrir la sesión de Hibernate desde la capa de presentación, esto no representa ningún problema desde el punto de vista de recursos pues Hibernate no obtiene una conexión a la base de datos sino hasta que se hace uso de la sesión. Esto quiere decir que la conexión a la base de datos se obtendrá del pool de conexiones hasta que se realice una petición a la capa de persistencia, y si alguna petición ni siquiera requiere acceder a la capa de persistencia nunca se hará una petición a una conexión de base de datos.

4.4 El Framework Spring

Spring es un framework creado para contrarrestar la complejidad del desarrollo de aplicaciones empresariales. Spring hace posible hacer uso de POJOs (Plain Old Java Objects) para alcanzar objetivos que eran imposibles con la arquitectura anterior. Sin embargo, la utilidad de Spring no se limita al desarrollo del lado del servidor. Cualquier aplicación Java puede beneficiarse de Spring en términos de simplicidad, pruebas unitarias, y bajo acoplamiento.

Spring hace muchas cosas, pero las partes medulares son lo ligereza, la inyección de dependencias, y que es un framework contenedor orientado a aspectos. Para poner en contexto estos términos se describirán a continuación.

Ligereza – Spring es ligero en términos de tamaño y robusto en extensión de soluciones. El contenido del Framework Spring se distribuye en un archivo JAR y el procesamiento requerido es insignificante. Además, Spring es desacoplado, es decir, los objetos en una aplicación Spring no tienen dependencias con otras clases.

Inyección de dependencias – Spring promueve el bajo acoplamiento bajo una técnica llamada inyección de dependencias. Cuando se aplica la ID, los objetos reciben pasivamente sus dependencias en vez de crearlas o buscarlas por si mismos. Es decir el contenedor (spring) asigna las dependencias a los objetos sin esperar a que los objetos las pidan.

Orientado a aspectos – Spring soporta la programación orientada a aspectos (AOP) la cual provee un desarrollo cohesivo separando la lógica de negocio de la aplicación de los servicios del sistema (tales como, auditoria y administración de transacciones). Los objetos de la aplicación hacen lo que se supone debieran hacer – realizar lógica de negocio – y nada más. No son responsables (incluso no se preocupan) de otros contextos del sistema, como las transacciones, seguridad, administración de sesiones, etc.

Contenedor: Spring es un contenedor en el sentido que contiene y administra el ciclo de vida y configuración de los objetos de la aplicación. In Spring, se puede declarar cómo cada uno de los objetos de la aplicación serán creados, cómo serán configurados y cómo deberían ser asociados entre si.

Framework – Spring hace posible configurar y crear aplicaciones complejas con componentes simples. En Spring, los objetos de la aplicación son creados declarativamente, en archivos XML. Spring provee además muchas funcionalidades de infraestructura (administración de transacciones, integración de persistencia, etc.)

4.4.1 Inyección de dependencias

La inyección de dependencias consiste en proporcionar a los objetos las referencias a otros objetos de los cuales depende su funcionamiento sin que el objeto dependiente sepa cómo se obtiene dicha referencia. El modelo de inyección de dependencias tiene las siguientes ventajas:

- Desacoplamiento de componentes.
- Desacoplamiento de tecnologías (por ejemplo el objeto dependiente no sabe si la referencia es un EJB o un POJO).
- Código más limpio con menos código de infraestructura.
- Facilita el desarrollo de pruebas unitarias.
- Permite gran flexibilidad al permitir agregar nueva funcionalidad mediante aspectos.

Esto no quiere decir que todas las relaciones entre objetos deban de ser establecidas mediante inyección de dependencias. Los componentes que deben de ser inyectados son

aquellos que atraviesan capas de la arquitectura., por ejemplo, la relación entre un servicio y un DAO.

Consideraciones:

- Diseñar usando interfases
- Los objetos deben de referirse a sus dependencias únicamente mediante sus interfases
- Es mediante Spring donde se inyecta la implementación de la interfase al objeto dependiente.

4.4.2 Arquitectura de Spring

Spring es un framework modular que cuenta con una arquitectura divida en siete capas o módulos, como se muestra en la Figura 4.4.2, lo cual permite tomar y ocupar únicamente las partes que interesen para el proyecto y juntarlas con gran libertad.

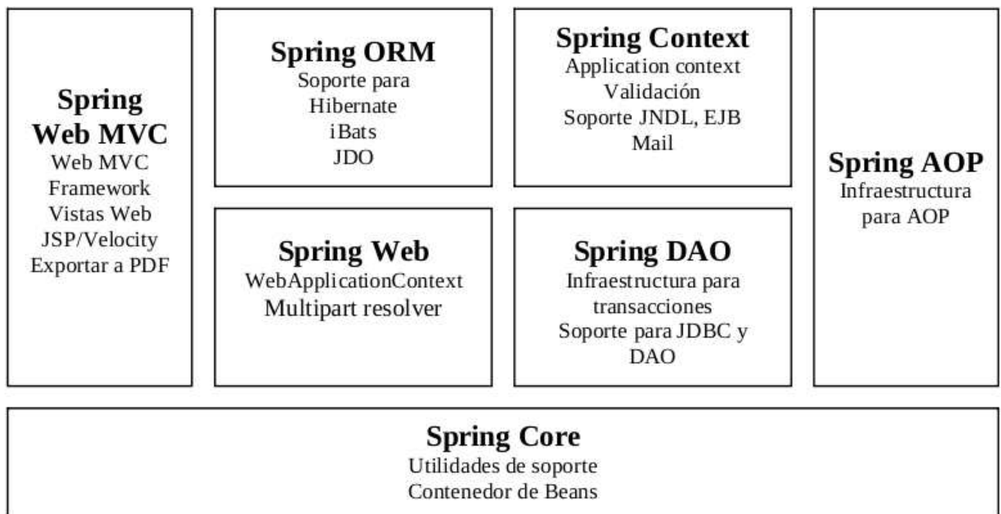


Figura 4.4.2 Arquitectura Spring

4.4.3 Spring Core (Núcleo)

Esta parte es la que provee la funcionalidad esencial del framework, está compuesta por el `BeanFactory`, el cual utiliza el patrón de Inversión de Control (Inversion of Control) y configura los objetos a través de Inyección de Dependencia (Dependency Injection). El núcleo de Spring es el paquete `org.springframework.beans` el cual está diseñado para trabajar con JavaBeans.

4.4.4 Bean Factory (Fábrica de beans)

Es uno de los componentes principales del núcleo de Spring. Es una implementación del patrón Factory, pero a diferencia de las demás implementaciones de este patrón, que muchas veces sólo producen un tipo de objeto, `BeanFactory` es de propósito general, ya que puede crear muchos tipos diferentes de Beans. Los Beans pueden ser llamados por

nombre y se encargan de manejar las relaciones entre objetos. Todas las Bean Factories implementan la interfase `org.springframework.beans.factory.BeanFactory`, con instancias que pueden ser accedidas a través de esta interfaz. Además también soportan objetos de dos modos diferentes:

- *Singleton*: Existe únicamente una instancia compartida de un objeto con un nombre particular, que puede ser regresado o llamado cada vez que se necesite. Este es el método más común y el más usado. Este modo está basado en el patrón de diseño que lleva el mismo nombre.
- *Prototype*: también conocido como non-singleton, en este método cada vez que se realiza un regreso o una llamada, se crea un nuevo objeto independiente.

La implementación de BeanFactory más usada es `org.springframework.beans.factory.xml.XmlBeanFactory` que se encarga de cargar la definición de cada Bean que se encuentra guardada en un archivo XML y que consta de: id (que será el nombre que el que se le conocerá en las clases), clase (tipo de Bean), Singleton o Prototype (modos del Bean antes mencionados), propiedades, con sus atributos `name`, `value` y `ref`, además argumentos del constructor, método de inicialización y método de destrucción. A continuación se muestra un ejemplo de un Bean:

```
<beans>
<bean id=”exampleBean” class=”eg.ExampleBean”
singleton=”true”/>
<property name="driverClassName"
value="com.mysql.jdbc.Driver" />
</beans>
```

Como se muestra en el ejemplo la base de este documento XML es el elemento `<beans>`, y adentro puede contener uno o más elementos de tipo `<bean>`, para cada una de las diferentes funciones que se requieran.

4.4.5 Spring Context (Contexto Spring)

El módulo `BeanFactory` del núcleo de Spring es lo que lo hace un contenedor, y el módulo de contexto es lo que hace un framework. En sí `Spring Context` es un archivo de configuración que provee de información contextual al framework general. Además provee servicios enterprise como JNDI, EJB, e- mail, validación y funcionalidad de agenda.

4.4.6 Application Context (Contexto de la aplicación)

Es una subinterfaz de `BeanFactory`, ya que `ApplicationContext` `org.springframework.context.ApplicationContext` es una subclase de `BeanFactory`. En si todo lo que puede realizar una `BeanFactory` también lo puede realizar `ApplicationContext`. En sí agrega información de la aplicación que puede ser utilizada por todos los componentes. Además brinda las siguientes funcionalidades extra:

- Localización y reconocimiento automático de las definiciones de los Beans
- Cargar múltiples contextos
- Contextos de herencia
- Búsqueda de mensajes para encontrar su origen.
- Acceso a recursos
- Propagación de eventos, para permitir que los objetos de la aplicación puedan publicar y opcionalmente registrarse para ser notificados de los eventos.
- Agrega soporte para internacionalización (i18n)
- En algunos casos es mejor utilizar ApplicationContext ya que obtienes más funciones a un costo muy bajo, en cuanto a recursos se refiere.

Ejemplo de un ApplicationContext:

```
ApplicationContext ct =
new FileSystemXmlApplicationContext ("c:\bean.xml");
ExampleBean eb = (ExampleBean) ct.getBean("exampleBean");
```

4.4.7 Spring AOP (Programación Orientada a Aspectos)

La programación orientada a Aspectos AOP, es una técnica que permite a los programadores modularizar contextos (crosscutting), o el comportamiento a través de las divisiones de responsabilidad, como logging, y manejo de transacciones. El núcleo de construcción es el aspect, que encapsula comportamiento que afectan a diferentes clases, en módulos que pueden ser reutilizados.

Spring AOP soporta las siguientes funcionalidades:

Intercepción: se puede insertar comportamiento personalizado antes o después de invocar a un método en cualquier clase o interfaz.
Introducción: Especificando que un advice (acción tomada en un punto particular durante la ejecución de un programa) debe causar que un objeto implemente interfaces adicionales.
Pointcuts dinámicos y estáticos: para especificar los puntos en la ejecución del programa donde debe de haber intercepción.

4.4.8 Spring ORM (Mapeo de Objetos Relacional)

En lugar de que Spring proponga su propio módulo ORM (Object-Relational Mapping), para los usuarios que no se sientan confiados en utilizar simplemente JDBC, propone un módulo que soporta los frameworks ORM más populares del mercado, entre ellos:

Hibernate: herramienta de mapeo O/R open source muy popular, que utiliza su propio lenguaje de query llamada HQL.

Todo esto se puede utilizar en conjunto con las transacciones estándar del framework. Spring y Hibernate es una combinación muy popular. Algunas de las ventajas que brinda Spring al combinarse con alguna herramienta ORM son:

Manejo de sesión: Spring hace de una forma más eficiente, sencilla y segura la forma en que se manejan las sesiones de cualquier herramienta ORM que se quiera utilizar.

Manejo de recursos: se puede manejar la localización y configuración de los SessionFactories de Hibernate o las fuentes de datos de JDBC por ejemplo. Haciendo que estos valores sean más fáciles de modificar.

Manejo de transacciones integrado: se puede utilizar una plantilla de Spring para las diferentes transacciones ORM.

Envolver excepciones: con esta opción se pueden envolver todas las excepciones para evitar las molestas declaraciones y los catch en cada segmento de código necesarios.

Evita limitarse a un solo producto: Si se desea migrar o actualizar a otra versión de un ORM distinto o del mismo, Spring trata de no crear una dependencia entre la herramienta ORM, el mismo Spring y el código de la aplicación, para que cuando sea necesario migrar a un nuevo ORM no sea necesario realizar tantos cambios.

Facilidad de prueba: Spring trata de crear pequeños pedazos que se puedan aislar y probar por separado, ya sean sesiones o una fuente de datos (datasource).

4.4.9 Spring DAO (Objeto de Acceso a Datos)

El patrón DAO (Data Access Object) es uno de los patrones más importantes y usados en aplicaciones J2EE, y la arquitectura de acceso a los datos de Spring provee un buen soporte para este patrón.

4.4.10 Spring Web

El módulo web de Spring se encuentra en la parte superior del módulo de contexto, y provee el contexto para las aplicaciones web. Este módulo también provee el soporte necesario para la integración con el framework Struts de Yakarta.
Este módulo también se encarga de diversas operaciones web como por ejemplo: las peticiones multi-parte que puedan ocurrir al realizar cargas de archivos y la relación de los parámetros de las peticiones con los objetos correspondientes (domain objetcs o business objects).

4.4.11 Spring Web MVC

Spring brinda un MVC (Model View Controller) para Web bastante flexible y altamente configurable, pero esta flexibilidad no le quita sencillez, ya que se pueden desarrollar aplicaciones sencillas sin tener que configurar muchas opciones.

4.4.12 Instalación y configuración de Spring

El framework Spring se distribuye en un archivo ZIP el cual contiene la fuente de archivos JAR con todas las dependencias. Está disponible en la dirección: http://www.springsource.org/download julio 2009.

Y basta con colocar los siguientes archivos JAR en la dirección WEB-INF/lib del proyecto en eclipse.

`Antlr-2.7.6.jar, antlr-runtime-3.1.3.jar, commons-logging-1.0.4.jar, spring.jar`

4.4.13 Trabajando con la fábrica de Spring Bean

La implementación de la fábrica Spring Bean del patrón de diseño `factory` proporciona beans u objetos de un tipo específico y los configura usando metadatos en forma de propiedades en documentos XML. La parte más usada de la fábrica Spring Bean es `XMLBeanFactory`, la cual puede inicializa una implementación de una interfaz desde la definición XML que describe qué clase en concreto utilizar y cómo configurarla. A continuación se desarrolla un ejemplo. Se define una interfaz para definir un servicio simple el caul escribe un mensaje, se llama ServicioSaludar.java.

4.4.14 Interfaz para el servicio

```
package mx.uaq.memorias.spring.servicios;

public interface ServicioSaludar {
String saludo();
String saludo(String contenido);
}
```

4.4.15 Implementación del servicio

```
package mx.uaq.memorias.spring.serviciosImpl;
import mx.uaq.memorias.spring.servicios.ServicioSaludar;

public class ServicioSaludarImpl implements ServicioSaludar {

private String saludo;

// Constructor
public ServicioSaludarImpl() {
}

/**
* Constructor usado para la inyeccion de dependencias.
```

```
*
* @param saludo
*/
public ServicioSaludarImpl(String saludo) {
this.saludo = saludo;
}

// Metodos del servicio saludar
public String saludo() {
return saludo;
}

public String saludo(String contenido) {
return saludo + " " + contenido;
}

// Setters para la inyeccion de dependencias.
public void setSaludo(String saludo) {
this.saludo = saludo;
}

}
```

El siguiente código es la configuración del archivo `beans.xml` Spring. El elemento bean define el `bean` de la clase que implementa la interfaz `ServicioSaludarImpl` y asigna la propiedad saludo vía el setter `setSaludo()` con el valor 'Memorias de trabajo SPRING'.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<!-- Configuracion simple Spring Bean -->
<beans>
<!--Configuracion del Servicio Saludar-->
<bean id="servicioSaludar"
class="mx.uaq.memorias.spring.serviciosImpl.ServicioSaludarIm
pl">
<property name="saludo">
<value></value>
</property>
</bean>
</beans>
```

Colocar el archivo XML en el mismo paquete (). Para usar el bean `servicioSaludar` que se inicializó con la fábrica Spring Bean se lee el archivo de configuración (usando `ClassPathResource`) y se usa el método `getBean` para recuperar la instancia del bean por su `id`, como se muestra:

```
package mx.uaq.memorias.spring.pruebas;
import mx.uaq.memorias.spring.servicios.ServicioSaludar;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
public class ServicioSaludarCliente {
public static void main(String[] args) {
Resource configuration = new
ClassPathResource("mx/uaq/memorias/spring/beans.xml");
BeanFactory factory = new XmlBeanFactory(configuration);
ServicioSaludar bean =
(ServicioSaludar)factory.getBean("servicioSaludar");
System.out.println(bean.saludo("Memorias de trabajo
SPRING"));
}
}
```

Con lo cual se obtiene la siguiente salida en consola.

```
INFO XmlBeanDefinitionReader - Loading XML bean definitions
from class path resource [mx/uaq/memorias/spring/beans.xml]
Memorias de trabajo SPRING
```

4.5 El Patrón de diseño DAO (Data Access Object - Objeto de Acceso a Datos)

Se utiliza el patrón DAO por algunas razones una de ellas es para poder abstraer y encapsular uno o todos los accesos a la fuente de datos de una aplicación. En una aplicación que utiliza múltiples fuentes de datos, el patrón DAO hace creer al código que todas las fuentes de datos son iguales, proporcionando una interfaz unificada y consistente para todas las operaciones de datos relacionadas. Una capa de diseño DAO debe esconder todos los detalles específicos de la tecnología de acceso a datos.

4.5.1 Características del diseño DAO

Codificar con interfaces: Los DAOs se definen como interfaces. Esto provee la flexibilidad para agregarse a diferentes implementaciones en tiempo de ejecución, así como para simplificar las pruebas unitarias utilizando objetos “mock” en ausencia de base de datos.

Provee transparencia: Los objetos de negocio (capa de negocio) pueden usar la fuente de datos sin saber los detalles específicos de la implementación. El acceso es transparente porque los detalles de la implementación son escondidos dentro del DAO.

Fácil migración: Una capa de DAOs es fácil para una aplicación en migrar a una implementación de base de datos diferente. Los objetos del negocio no saben de la implementación de dicha capa. Es por ello, que la migración involucra solo cambios en la capa DAO.

Reduce la complejidad del código en los objetos negocio: Porque los DAOs manejan todas las complejidades del acceso a los datos, simplificando el código en los objetos de negocio y otros clientes que usen DAOs. Todo el código de implementación relacionado con sentencias SQL está contenido en los DAOs y no en objetos de negocio. Esto provee legibilidad de código y productividad en el desarrollo.

Centraliza todos los accesos a datos en una capa separada. Todas las operaciones de acceso a los datos están delegadas a los DAOs, la capa que separa el acceso a los datos es vista como una capa que puede ser aislada del resto de la aplicación. Esta centralización hace que la aplicación sea más fácil de mantener y administrar.

4.5.2 Archivos de configuración Spring y Hibernate

A continuación se mostraran los archivos de configuración necesarios para levantar un ambiente Spring y Hibernate, la figura XXX muestra la jerarquía para colocar los archivos en un proyecto en eclipse.

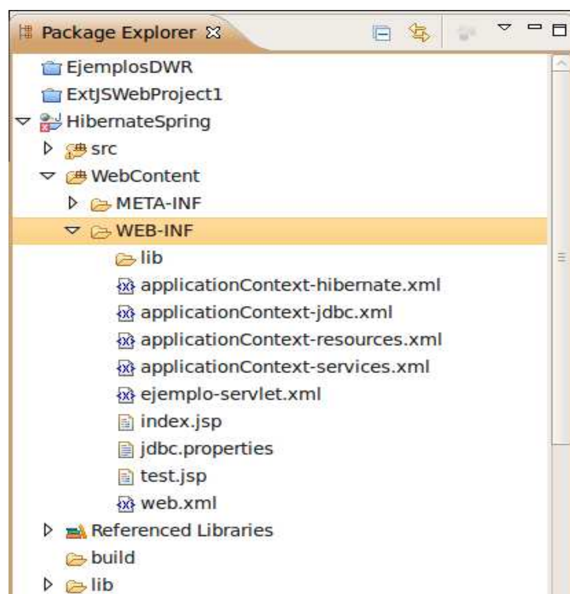


Fig. 4.5.2 Jerarquía de archivos de configuración

4.5.3 Archivo de configuracón applicationContext-jdbc.xml

Para poder configurar beans de tipo `SessionFactory`, es necesario ajustar la fuente de datos JDBC.

El siguiente archivo de configuración `applicationContext-jdbc.xml` muestra la configuración JDBC para la fuente de datos. Los valores están contenidos en el archivo llamado `jdbc.properties`, los cuales se localizan en el directorio del proyecto `WEB-INF/`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!-- JDBC Template -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
<property name="exceptionTranslator"
ref="jdbcExceptionTranslator" />
<property name="dataSource" ref="dataSource" />
</bean>
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransact
ionManager">
<property name="dataSource" ref="dataSource" />
</bean>
</beans>
```

4.5.4 Archivo jdbc.properties

```
# Archivo properties con la configuración JDBC relacionada.
# Aplicado para PropertyPlaceholderConfigurer de los
"applicationContext-*.xml".

jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/mysql_hibernate
jdbc.username=unix
jdbc.password=avr102894

# Propiedades que determinan el dialecto Hibernate
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.cache.provider_class=org.hibernate.cache.HashtableC
acheProvider
```

4.5.5 Archivo applicationContext-resources.xml

Para especificar la configuración de los recursos del archivo jdbc se crea el archivo applicationContext-resources.xml,

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!-- ============ Definicion de recursos ============ -->
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlace
holderConfigurer">
```

```
<property name="location" value="/WEB-INF/jdbc.properties" />
</bean>
<!-- DataSource Definition -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
<property name="driverClassName"
value="${jdbc.driverClassName}" />
<property name="url" value="${jdbc.url}" />
<property name="username" value="${jdbc.username}" />
<property name="password" value="${jdbc.password}" />
</bean>
</beans>
```

4.5.6 Archivo applicationContext-hibernate.xml

Con la configuración de la fuente de datos en su lugar, se debe configurar también el SessionFactory como se muestra a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

<!-- Definicion de la Fuente de datos -->
<bean class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close" id="dataSource">
<property name="driverClassName"
value="${jdbc.driverClassName}"/>
<property name="url" value="${jdbc.url}"/>
<property name="username" value="${jdbc.username}"/>
<property name="password" value="${jdbc.password}"/>
</bean>

<!-- Definicion de Hibernate SessionFactory -->
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactory
Bean">
<property name="dataSource"><ref
local="dataSource"/></property>
<property name="mappingResources">
<list>
<value>com/memorias/hibernate/pojo/Conferencia.hbm.xml
</value>
<value>com/memorias/hibernate/pojo/Direccion.hbm.xml</value>
<value>com/memorias/hibernate/pojo/Tema.hbm.xml</value>
```

```
<value>com/memorias/hibernate/pojo/Sede.hbm.xml</value>
</list>
</property>
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">
org.hibernate.dialect.MySQLDialect</prop>
<prop key="hibernate.show_sql">true</prop>
<prop key="hibernate.hbm2ddl.auto">create</prop>
<prop key="hibernate.cglib.use_reflection_optimizer">
true</prop>
<prop key="hibernate.cache.provider_class">
org.hibernate.cache.HashtableCacheProvider</prop>
</props>
</property>
</bean>

<!-- Definición Hibernate -->
<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
<property name="sessionFactory" ref="sessionFactory" />
<property name="jdbcExceptionTranslator"
ref="jdbcExceptionTranslator" />
</bean>
<!--Objeto DAO:Template de implementacion Hibernate-->
<bean id="conferenciaDAO"
class="com.memorias.daoImpl.ConferenciaDAOImpl">
<property name="hibernateTemplate">
<ref bean="hibernateTemplate"/>
</property>
</bean>

</beans>
```

Se usa la clase de integración `LocalSessionFactoryBean` para usarse como referencia a beans, se comporta de la misma manera que SessionFactory en Hibernate. El uso de esa clase es recomendada para la mayoría de las aplicaciones Spring-Hibernate.

La configuración `mappingResources` en el archivo señala la ubicación de los mapeos ORM con hibernate

Finalmente en este archivo se configuran las propiedades de Hibernate mediante `hibernateProperties` la cual provee los parámetros para configurar el `SessionFactory`. En este caso las propiedades `hibernate.dialect` y `hibernate.show_sql` se definen en el archivo `jdbc.properties` por conveniencia.

De esta manera la sesión Hibernate estará disponible para cualquier bean en la aplicación.

4.6 Implementación del Patrón DAO con Spring y Hibernate

Spring provee una clase de utilería llamada HibernateTemplate para lidear con la administración de las instancias `Session` de Hibernate en un contexto transaccional.

HibernateTemplate implementa la interfaz Hibernate Session envolviendo la sesión Hibernate o creándola cuando es necesario. Al usar el esquema Spring los objetos se reducirán en código requerido para crear DAOs. Ya que eliminan la necesidad de cerrar la sesión y capturar las excepciones con código (try, catch, finally).

Para configurar el esquema Hibernate se inyectó el SessionFactory en el archivo de configuración `applicationContext-hibernate.xml` con el `bean id="hibernateTemplate"`.

HibernateTemplate será inyectado en las implementaciones de los DAOs, los cuales extenderán la clase base Hibernate.DAOSupport la cual provee métodos convenientes para transacciones.

Para simplificar la implementación concreta de las interfaces de los DAOs es común utilizar una clase base abstracta como la que se muestra a continuación, la cual usa HibernateTemplate con la clase base HibernateDaoSupport.

4.6.1 Clase Base Abstracta DAO

```
package com.memorias.dao;

import org.springframework.orm.hibernate3.support.
     HibernateDaoSupport;

public class BaseAbstractDAO extends HibernateDaoSupport {

protected void saveEntidad(Object entidad) {
getHibernateTemplate().persist(entidad);
}
protected void deleteEntidad(Object entidad) {
getHibernateTemplate().delete(entidad);
}

protected void updateEntidad(Object entidad) {
getHibernateTemplate().update(entidad);
}
}
```

Con una clase abstracta base se pretende encapsular los métodos comunes a todos los DAOs de una aplicación como son el guardar, eliminar, actualizar y buscar registros de la base de datos.

4.6.2 Ejemplo cómo crear un DAO

Finalmente, implementar un DAO en concreto se convierte en algo casi trivial. A continuación se muestra un DAO relacionado con la interfase para el objeto de dominio `Conferencia`.

4.6.3 Interfaz para crear un DAO de la entidad Conferencia

```
package com.memorias.dao;

import com.memorias.hibernate.pojo.Conferencia;

public interface ConferenciaDAO {
Conferencia obtenConferencia(int conferenciaId);
void save(Conferencia conferencia);
Conferencia obtenConferenciaPorNombre(String nombre);
void delete(Conferencia conferencia);
void update(Conferencia conferencia);
}
```

4.6.4 Implementación del DAO

```
package com.memorias.daoImpl;
import org.hibernate.Query;
import com.memorias.dao.BaseAbstractDAO;
import com.memorias.dao.ConferenciaDAO;
import com.memorias.hibernate.pojo.Conferencia;

public class ConferenciaDAOImpl extends BaseAbstractDAO
implements ConferenciaDAO {

public Conferencia obtenConferencia(int conferenciaId) {
Query query = getSession().createQuery("from
Conferencia c where c.cveConf = :conferenciaId");
query.setParameter("conferenciaId", conferenciaId);
return (Conferencia) query.uniqueResult();
}
public Conferencia obtenConferenciaPorNombre(
String nombre) {
Query query = getSession().createQuery("from
Conferencia c where c.nombre = :nombre");
query.setParameter("nombre", nombre);
return (Conferencia) query.uniqueResult();
}
public void save(Conferencia conferencia) {
saveEntidad(conferencia);
}
```

```
public void delete(Conferencia conferencia) {
deleteEntidad(conferencia);
}
public void update(Conferencia conferencia) {
updateEntidad(conferencia);
}
}
```

La siguente es una clase de Negocio para probar el DAO Conferencia los resultados son mostrados en la consola de eclipse.

4.6.5 Clase Negocio para probar el DAO Conferencia

```
package com.memorias.negocio;

import java.io.IOException;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import org.springframework.context.support.
ClassPathXmlApplicationContext;
import com.memorias.dao.ConferenciaDAO;
import com.memorias.hibernate.pojo.Conferencia;
import com.memorias.hibernate.pojo.Sede;
import com.memorias.hibernate.pojo.Tema;
public class ConferenciaNegocio {
//Definición de constantes
public static final String ID_BEAN_CONFERENCIA_DAO =
     "conferenciaDAO";
public static final String NOMBRE_CONFERENCIA =
     "Hibernate";
public static final Integer CVE_CONFERENCIA = 1;
public static final String NOMBRE_CONFERENCIA_UPDATE =
     "Nombre Actualizado";
public static void main (String ...strings) throws
IOException {

Conferencia conferencia = new Conferencia();
Tema tema = new Tema();
Set<Tema> temas = new HashSet<Tema>();
Sede sede = new Sede();
//Creación de los objetos persistentes.
sede.setNombre("Auditorio UAQ CU");
tema.setDescripcion("Configuración Spring");
tema.setSubtitulo("Memorias Hibernate-Spring");
tema.setTitulo("Frameworks Open Source ORM");
temas.add(tema);
```

```
conferencia.setDescripcion("Tecnologías ORM");
conferencia.setFechaIni(new Date());
conferencia.setFechaFin(new Date());
conferencia.setNombre("Hibernate");
conferencia.setTemas(temas);
conferencia.setSede(sede);
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
new String[] {
// Se genera el contexto o ambiente
// Spring-Hibernate a partir de los archivos de //
configuración
"/WEB-INF/applicationContext-hibernate.xml",
"/WEB-INF/applicationContext-resources.xml",
"/WEB-INF/applicationContext-services.xml",
"/WEB-INF/applicationContext-jdbc.xml"
}
);
ConferenciaDAO confeDAO = (ConferenciaDAO) context.getBean(
ID_BEAN_CONFERENCIA_DAO);
confeDAO.save(conferencia);
confeDAO.obtenConferencia(CVE_CONFERENCIA);
confeDAO.obtenConferenciaPorNombre(
NOMBRE_CONFERENCIA);
//Actualizacion del nombre de la conferencia.
conferencia.setNombre(NOMBRE_CONFERENCIA_UPDATE);
confeDAO.update(conferencia);
confeDAO.delete(conferencia);
}
}
```

Nótese, que en el context Spring y Hibernate applicationContext-hibernate.xml) quedó declarado el bean conferenceDAO y de esta manera se trabajan los DAOs con las interfaces.

5 Pruebas unitarias con el framework JUnit

Una prueba unitaria es el procedimiento usado para validar que un módulo particular de código fuente trabaje apropiadamente. Generalmente, consiste en escribir casos de prueba para las funciones y métodos. Cada caso de prueba ejecuta un método en particular, en un contexto en particular, validando que la ejecución sea la adecuada.

5.1 Características principales

- Se hace antes de declarar como terminado el código que se elabora.
- Es realizada por el programador, no por un área de pruebas ni por el usuario final.
- Se enfoca en probar un solo componente a la vez, no varios.

- Sirve para que el programador se asegure que su componente haga lo que se supone que tiene que hacer.

5.2 Estructura de una prueba unitaria

- Preparación del contexto
- Invocación de la funcionalidad a probar
- Comprobación de resultados obtenidos
- Restablecimiento de contexto

5.3 Preparación del contexto

El contexto de una prueba son todos los elementos del ambiente que se necesitan para poder ejecutarla.

Ejemplos de elementos de ambiente:

- Que existan registros en una BD
- Que existan las variables específicas de ambiente
- Que estén disponibles archivos de configuración

La preparación del contexto también incluye el crear el componente, y lograr que adquiera el estado necesario para ejecutar la prueba.

5.4 Invocación de funcionalidad a probar

Consiste simplemente en invocar al método que contiene la funcionalidad a probar, mandando los parámetros con los valores adecuados para que se ejecute dicha funcionalidad. Es posterior a la preparación del contexto.

5.5 Comprobación de resultados obtenidos

Consiste en confrontar los resultados esperados contra los obtenidos por la ejecución del método

Estos resultados pueden ser:

- Valores de retorno
- Modificación de parámetros de salida
- Cambios en el estado del objeto
- Excepciones o errores disparados
- Cambios en elementos del ambiente (BD, bitácoras, variables de ambiente, etc.)

5.6 Restablecimiento de contexto

Consiste en regresar los elementos utilizados para ejecutar la prueba o afectados por la prueba al estado en que estaban antes de su ejecución. La idea es dejar todo listo para poder repetir la prueba.

Ejemplos de elementos utilizados o afectados por la prueba son BD, variables de ambiente, bitácoras, etc. A diferencia de los pasos anteriores, este paso se debe realizar sólo si es necesario.

5.7 Introducción a JUnit

JUnit es un framework para realizar pruebas de componentes Java. Está estructurado de modo tal que sea fácil de usar y/o extender. Actualmente es el framework más utilizado, el que cuenta con mayor cantidad de especializaciones, y el más documentado.
Cuenta con la ventaja de que puede ser integrado a múltiples IDEs (Eclipse) y herramientas de desarrollo.

5.7.1 Elementos básicos de JUnit

`TestCase` - clase proporcionada por JUnit para derivar las clases que contendrá el código de las pruebas.

`TestSuite` - establece un mecanismo para agrupar pruebas. Si no se define uno, por defecto JUnit genera uno con todas las pruebas contenidas en la clase TestCase.

`TestRunner` - disparador de TestSuites. JUnit proporciona varios tipos, es opcional la posibilidad de elegir el que más sea conveniente.

5.7.2 Las validaciones Assert

La clase `TestCase` extiende una clase de utilería llamada `Assert` en el framework JUnit. La clase `Assert` provee los métodos necesarios para hacer validaciones a cerca del estado de los objetos que se prueban. `TestCase` extiende `Assert` es por eso que se pueden hacer validaciones para comprobar sin hacer referencia a una clase externa. Los métodos básicos de validación `Assert` se describen a continuación en la tabla 4.7.6.2.

Método	Descripción
`assertTrue`	Valida que una condición sea true. Si es false, el método lanza una excepción del tipo AssertionFailedError con el mensaje especificado (si es que existe.)
`assertFalse`	Valida que una condición sea false. Si es true el método lanza una excepción del tipo AssertionFailedError con el mensaje especificado (si es que existe.)
`assertEquals`	Valida que dos objetos sean iguales. Si no lo son, el método lanza una excepción del tipo AssertionFailedError con el mensaje especificado (si es que existe.)
`assertNotNull`	Valida que un objeto no sea nulo. Si es nulo, el método lanza una excepción del tipo AssertionFailedError con el mensaje especificado.

`assertNull`	Valida que un objeto sea nulo. Si no es nulo, el método lanza una excepción del tipo AssertionFailedError con el mensaje especificado (si es que existe.)
`assertSame`	Valida que dos objetos tengan la misma referencia al objeto. Si no la tienen, el método lanza una excepción del tipo AssertionFailedError con el mensaje especificado (si es que existe.)
`assertNotSame`	Valida que dos objetos no hagan referencia al mismo objeto. Si tienen la misma referencia, el método lanza una excepción del tipo AssertionFailedError con el mensaje especificado (si es que existe.)
`fail`	Provoca el fallo de una prueba con el mensaje especificado.

5.7.3 Descargar e instalar JUnit

JUnit es fácil de instalar y usar. Para tener JUnit configurado y ejecutándose, se debe:

1. Descargar el paquete JUnit de la dirección http://www.junit.org/ julio de 2009
2. Descomprimir el archivo al sistema.
3. Incluir el archivo JAR al class path de la aplicación Eclipse para poder construir o ejecutar pruebas.

5.7.4 Ejemplo estructurado de una prueba unitaria

El siguiente ejemplo es una clase que realiza operaciones aritméticas básicas como las de una calculadora. Primero se presenta la clase que implementa los métodos y en seguida la clase JUnit que realiza las pruebas.

Clase Calculadora.java

```
package com.memorias.junit.ejemplos;

public class Calculadora {
     public Double sumar (Double a, Double b) {
          return a + b;
     }

     public Double restar (Double a, Double b) {
          return a - b;
     }

     public Double multiplicar (Double a, Double b) {
          return a * b;
     }

     public Integer dividir(Integer a, Integer b) throws
          ArithmeticException {
```

```
                return (a / b);
        }
}
```

Clase JUnit que realiza las pruebas unitarias

```
package com.memorias.junit.ejemplos;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class TestCalculadora extends TestCase {
          protected Double valor1;
          protected Double valor2;
          protected Integer int1;
          protected Integer int2;
          protected Integer int3;
          public Calculadora calc;

          public TestCalculadora(String name) {
               super(name);
          }
          protected void setUp() {
               calc = new Calculadora();
               valor1= 2d;
               valor2= 7d;
               int1 = 25;
               int2 = 0;
               int3 = 5;
          }
           public static Test suite() {
                  TestSuite suite = new TestSuite();
                  suite.addTest(new
                         TestCalculadora("testSumar"));
                  suite.addTest(new
                         TestCalculadora("testRestar"));
                  suite.addTest(new
                         TestCalculadora("testMultiplicar"));
                  suite.addTest(new
                         TestCalculadora("testDivision"));
                  suite.addTest(new
                         TestCalculadora(
                              "testDivisionEntreCero"));
                  return suite;
              }
          public void testSumar() {
```

```
                Double resultado= calc.sumar(valor1, valor2);
                assertEquals("Suma", 9d, resultado);
        }

        public void testRestar() {
                Double resultado= calc.restar(valor1, valor2);
                assertEquals("Resta", -5d, resultado);
        }

        public void testMultiplicar() {
                Double resultado= calc.multiplicar(valor1,
                        valor2);
                assertEquals("Multiplicación", 14d,
                        resultado);
        }

        public void testDivision() {
                Integer resultado = calc.dividir(int1, int3);
                assertEquals("Division", (Integer)5,
                         resultado);
        }
        public void testDivisionEntreCero() {
                try {
                        calc.dividir(2, 5);
                fail("El método no lanzó la excepción
                        esperada");
                } catch (ArithmeticException ae) {}
        }
}
```

La figura 5.7.4 muestra el resultado de las pruebas en la perspectiva JUnit (pestaña JUnit en la imagen) del IDE Eclipse.

La prueba `testDivisionEntreCero` es una prueba para verificar que se lance una excepción por una división entre cero (`ArithmeticException`) y falla pues el resultado de la llamada al método con parámetros `dividir(2,5)` no lanza ninguna excepción.

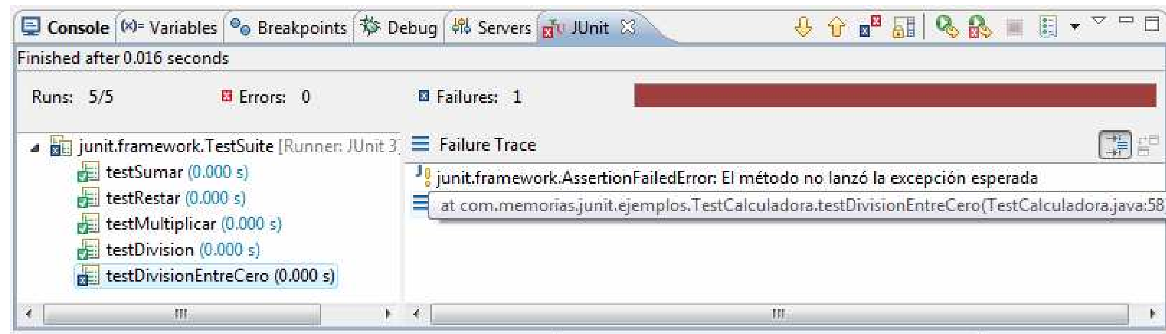


Fig. 5.7.4 Perspectiva JUnit en Eclipse

5.7.5 Diferencia entre falla y error (fail, error)

Cuando la prueba contiene validaciones o métodos `fail()`, JUnit cuenta como correctas las verificaciones del método, pues el programador orientó como correcto que el método fallara. Pero cuando la prueba lanza una excepción (y no se captura). JUnit asume que es un error, es decir, existe un problema con la prueba en si o con el ambiente en el que se ejecuta.

6 Introducción a INFORMIX-4GL

Informix 4gl es un lenguaje de cuarta generación desarrollado por Informix Software y diseñado específicamente para aplicaciones de bases de datos.

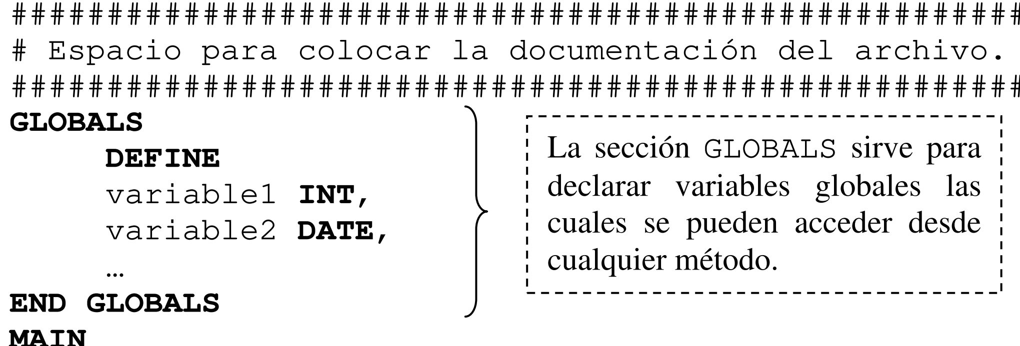
6.1 Qué son los lenguajes de 4ª generación

Los lenguajes de 4ª generación como 4gl son diseñados para una clase de aplicación en particular. Son menos complejos que lenguajes como C, Java, COBOL pero son muy poderosos una simple sentencia genera código máquina complejo. Algunas de las ventajas de los lenguajes de 4ª generación se describen a continuación.

- Son simples, lo cual aumenta la velocidad de construcción y mantenimiento de las aplicaciones.
- Son interactivos, lo cual simplifica el proceso de debugueo.
- Llega a una amplia audiencia porque no requiere de entrenamiento especial.
- Las aplicaciones resultantes son fáciles de usar y pueden resolver problemas eficientemente.

A pesar de que 4gl está diseñado para aplicaciones de bases de datos, tiene aún muchas características de un lenguaje de programación. En adición a las sentencias específicas de base de datos, 4gl soporta sentencias similares a aquellas encontradas en otros lenguajes, tales como, sentencias de control, condicionales, asignación de variables, etc. En este capítulo se muestran las más útiles a través de ejemplos sencillos.

6.2 Estructura de un archivo 4gl para ser compilado




```
######################################################
# Espacio para colocar la documentación del archivo.
######################################################
GLOBALS
     DEFINE
     variable1 INT,
     variable2 DATE,
     …
END GLOBALS
MAIN
```

```
DEFINE
     variableLocal1 CHAR(10),
     variableLocal2 CHAR(5),
     …
     CALL metodo1(variableLocal1,
           variableLocal2)
     CALL …

END MAIN
```

La sección MAIN es usada para hacer las llamadas a los métodos del programa a ejecutarse, seleccionar la base de datos y declarar variables locales.

```
###################################
# Documentación del método.
###################################
FUNCTION metodo1(variableLocal1,
               variableLocal2)
DEFINE
     variableLocal1 CHAR(10),
     variableLocal2 CHAR(5),
     resultado INT

     # Sentencias SQL y algoritmos

     RETURN resultado
END FUNCTION
```

Esta sección restante del archivo es utilizada para declarar todos los métodos (FUNCTION) que conforma la aplicación.

6.3 Principales sentencias del lenguaje 4gl

Una vez que se mostró cómo declarar un programa en funciones se da paso a explicar algunas de las sentencias que se utilizan en una aplicación y que sirven como introducción a 4gl.

6.3.1 Sentencia DATABASE *nombre-baseDatos*

Sirve para seleccionar la base de datos sobre la cual se trabajará.

Ejemplo:

```
MAIN
DEFINE
     baseDeDatos CHAR(50),
     LET baseDeDatos = ARG_VAL(1)
     DATABASE baseDeDatos
END MAIN
```

6.3.2 Sentencia DEFINE lista-de-variables tipo-de-dato

Se debe usar la sentencia `DEFINE` para identificar las variables del programa. Donde `lista-de-variables` es la lista de uno o más nombres de variables, separadas por coma y `tipo-de-dato` es el correspondiente tipo de dato.
Se coloca la sentencia `DEFINE` inmediatamente después de las siguientes secciones `GLOBALS, MAIN, FUNCTION,`

```
FUNCTION metodo1(variableLocal1)
DEFINE
     variableLocal1 CHAR(10)
END FUNCTION
```

6.3.3 Tipos de dato

TIPO	Descripción
CHAR [(*n*)]	Cadena de caracteres de longitud *n*.
DECIMAL [(*m*[,n])]	Números con dígitos significativos *m* <= 32 y dígitos n hacia la derecha del punto decimal.
SMALLINT	Números enteros del -32,767 a +32,767.
INTEGER	Números enteros del -2,147,483,647 a +2,147,483,647.
SMALLFLOAT	De precisión simple, números de punto flotante con números significativos de 7 dígitos.
FLOAT [(*n*)]	De doble precisión, números de punto flotante con números significativos de 14 dígitos.
DATE	Cadena de caracteres que contiene fechas del calendario.
DATETIME	Números que contienen fechas y la hora del día.
INTERVAL	Números para especificar periodos de tiempo.
MONEY [(*m*[,*n*])]	Números decimales con un número de dígitos establecidos a la derecha del punto decimal. MONEY (m) = DECIMAL (m,2) y del tipo MONEY = DECIMAL (16,2)

6.3.4 Sentencia LET

Una vez que se ha definido una variable, se le puede asignar un valor con la sentencia LET
Ejemplo:

```
DEFINE altura INTEGER
LET altura = 3
```

6.3.5 Definiendo un RECORD

Un record es una colección de variables posiblemente de diferentes tipos de datos. Y se utiliza principalmente para guardar columnas de una tabla en combinación con los cursores. El siguiente ejemplo muestra cómo declarar un record usando DEFINE:

```
DEFINE cur_conferencia
     RECORD
          nombre CHAR(25),
          descripcion CHAR(25),
          fech_ini DATE,
          fech_fin DATE
     END RECORD
```

6.3.6 Usando un RECORD en una sentencia SELECT

Se puede usar un record después de la palabra INTO de una sentencia SELECT para copiar la información de las columnas de la base de datos. Se debe estar seguro de que las columnas en la cláusula SELECT aparezcan en el mismo orden que las variables del RECORD, también se debe asegurar que los tipos de datos de las columnas sean iguales a las variables.
Ejemplo:

```
DEFINE cur_conferencia
     RECORD
          nombre CHAR(25),
          descripcion CHAR(25),
          fech_ini DATE,
          fech_fin DATE
     END RECORD

SELECT nombre, descripcion, fech_ini, fech_fin
     INTO cur_conferencia.*
FROM conferencia
WHERE cve_conf = 3
```

6.3.7 Definiendo un CURSOR

Un cursor sirve para almacenar la información de una sentencia SELECT que regresa más de un renglón.
Ejemplo:

```
DECLARE q_curs CURSOR FOR
SELECT * FROM customer
```

6.3.8 Sentencia FOREACH

Una vez que se ha declarado un cursor para una sentencia SELECT, se pueden seleccionar los renglones y ejecutar una secuencia de sentencias para cada renglón iterado utilizando la sentencia FOREACH. Se puede declarar la iteración secuencial o aleatoria.
Ejemplo:

```
DEFINE rec_tema RECORD LIKE tema.*
DECLARE cur_tema CURSOR FOR
     SELECT * FROM tema
FOREACH cur_tema INTO rec_tema.*
     DISPLAY rec_tema.titulo
     DISPLAY rec_tema.subtitulo CLIPPED
     DISPLAY rec_tema.descripcion
END FOREACH
```

El ejemplo anterior muestra en pantalla todo el contenido de la tabla `TEMA`. Nótese lo siguiente: la sentencia **`LIKE`** en el `RECORD` sirve para crear un record con la misma estructura de la tabla `TEMA`, otra sentencia usada fue **`CLIPPED`** la cual suprime los caracteres en blanco si es que el contenido de la columna no ocupa toda la longitud. **`DISPLAY`** como puede intuirse sirve para mandar como salida a pantalla una cadena.

6.3.9 Sentencia IF

Esta es una sentencia de control y la sintaxis es como se muestra a continuación.

```
IF rec_tema.titulo = "Hiberbate" THEN
     DISPLAY "Herramienta OpenSource"
END IF
```

6.3.10 Valores de retorno de una función

Esta sección muestra como pasar información de una función que tiene valor de retorno a la sección MAIN incluyendo la sentencia RETURN en la función asociada con la cláusula CALL y RETURNING. Ejemplo:

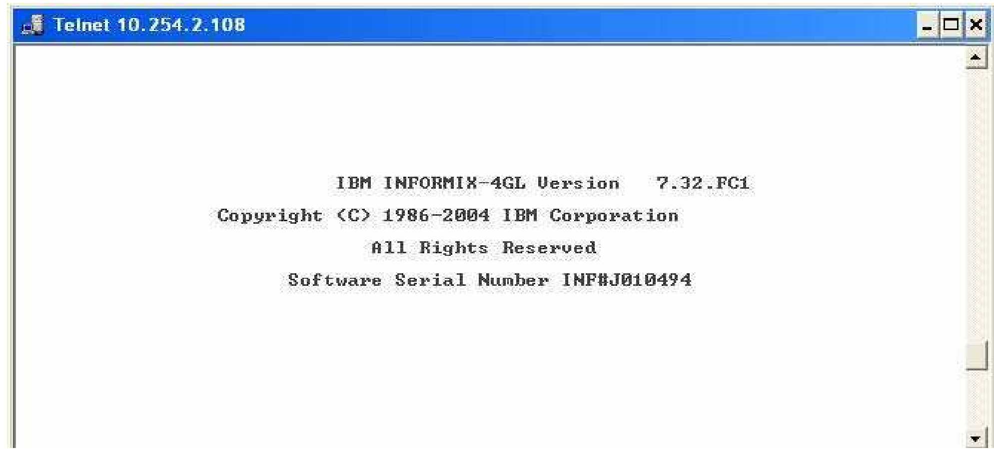
```
MAIN
     DEFINE fecha DATE
CALL obtenerFechaInicioConferencia()
     RETURNING fecha
     DISPLAY "Fecha de inicio de la Conferencia: ", fecha
END MAIN

FUNCTION obtenerFechaInicioConferencia( )
DEFINE fechaIni DATE
          SELECT fech_ini INTO fechaIni
          FROM conferencia
          WHERE nombre = "Scripting Informix-4GL"
```

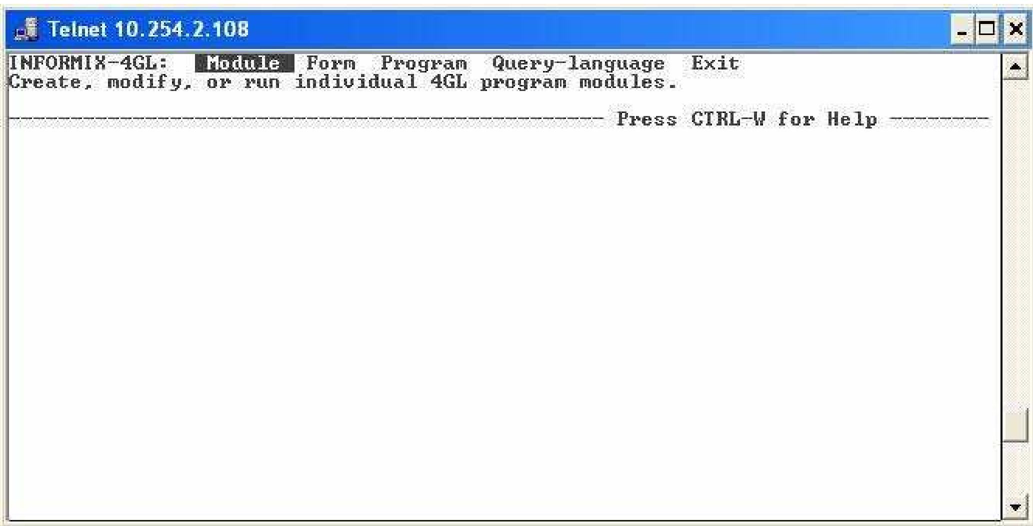
```
    RETURN fechaIni
END FUNCTION
```

6.4 Compilación de un archivo 4gl

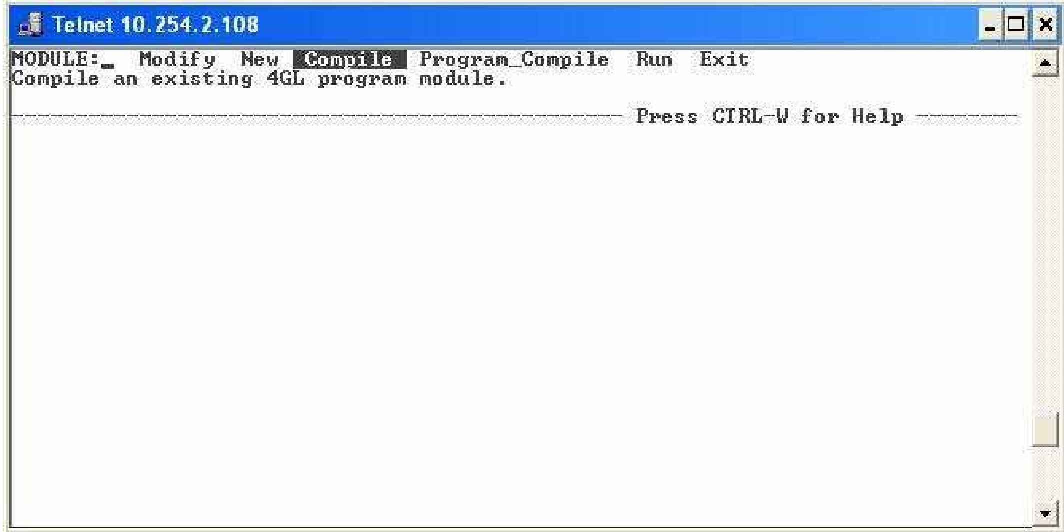
Solo basta con ingresar el comando `i4gl` una vez que se encuentra conectado al servidor unix donde resida alguna base de datos Informix e i4gl configurado. La primera ventana de bienvenida que aparece es la siguiente.



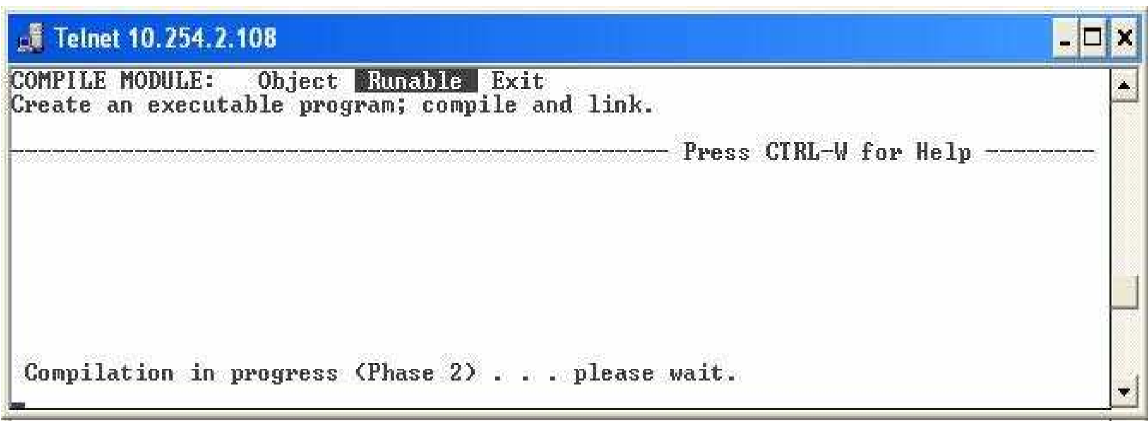
Se debe estar posicionado en la ruta donde se tienen guardados los archivos con extensión .4gl pues el siguiente paso es seleccionar del menú la opción MODULE como se muestra en la figura XXX.



Lo siguiente es elegir la opción COMPILE del menú desplegado. Como lo ilustra la figura XXX.



El siguiente paso es seleccionar RUNABLE para generar el archivo ejecutable 4ge



Una vez compilado es necesario salir del entorno 4gl y escribir el nombre del archivo con extensión 4ge.

Conclusiones

Como conclusiones finales quiero hablar de los Frameworks de código abierto, a lo largo del desarrollo de este trabajo se presentaron temas relacionados con múltiples tecnologías utilizadas para desarrollar aplicaciones principalmente en Java.

El framework Spring tiene como principal ventaja que proporciona la posibilidad de integrar al *framework* con otras herramientas o incluso otros *frameworks* con el fin de obtener los beneficios que el desarrollador desea de cada una de ellas. Para esto brinda diferentes módulos, según el tipo de la herramienta o *framework* a integrar. Por lo que se puede decir que Spring intenta integrarse con otras tecnologías y no competir contra ellas. Otro de los aspectos más interesantes de Spring es que la complejidad de la aplicación es proporcional a la complejidad del problema que se está resolviendo.

Otras de las ventajas de Spring es que esta diseñado con interfaces para que el desarrollador pueda utilizarlas, promoviendo así la reutilización de código y un estándar del paradigma orientado a objetos. Una ventaja práctica de Spring, para intentar hacer el código menos repetitivo al momento de hacer una lectura y/o escritura de datos en una base de datos, es que proporciona una serie de clases e interfaces para simplificar estas acciones. Además de asegurar de que se realicen de manera segura y no queden abiertas conexiones a la base de datos y que las excepciones que se den sean capturadas y traducidas a excepciones del mismo Spring.
Otro de los enfoques bajo los cuales fue construido y diseñado este *framework* es que una clase, en este caso un controlador, tenga una única función. Todo esto con el propósito de que se enfoque a resolver su tarea de manera más centrada y el desarrollador no tenga que preocuparse por otras cuestiones. Además de que este tipo de filosofía también se apoya en la teoría de que una clase con un solo propósito es más fácil de probar y corregir en caso de ser necesario.

Hibernate es una poderosa herramienta ORM muy popular de alto rendimiento y con características enriquecidas para soluciones Java. Hibernate facilita el desarrollo de objetos persistentes basados en el modelo común de objetos Java para interpretar la estructura de base de datos que se utiliza. Además del mapeo de objetos a una base de datos, Hibernate provee consultas avanzadas y recepción de servicios a través de HQL, proporciona también caché eficiente lo cual evita realizar consultas inecesarias. Por todo esto Hibernete sobresale como una de las principales tecnologías de mapeo relacional.

Las aplicaciones empresariales de hoy en día están orientadas a ser presentadas al usuario final en un ambiente Web. Uno de los frameworks de código libre que han crecido en popularidad es ExtJS el cual permite crear interfaces de usuario como si fuesen de escritorio. ExtJS usa un enfoque orientado a objetos y está muy bien estructurado. Y se convierte en lider de para la creación de interfaces JavaScript.

Parte importante del desarrollo de aplicaciones es realizar pruebas unitarias a cada uno de los componenetes que se elaboran y que componen una aplicación, el framework de código abierto más utilizado en construcción Java es JUnit sirve fundamentalmente para probar el funcionamiento de cada componente de manera aislada del resto de la aplicación.

Así es como concluye el trabajo presentado y sirve para introducir al lector en el desarrollo de software con el lenguaje de programación Java. Fueron abordados termas de las principales tecnologías de código abierto. Es la muestra de las vivencias y experiencias que tuve al trabajar en una fábrica de software que implementa estrictas normal de calidad en la elaboración de software.

Glosario

AJAX - Tecnología de JavaScript Asíncrona y XML.

API - Una interfaz de programación de aplicaciones.

DAO – Objeto de Acceso a Datos.

Debugger/Debuguear - Un depurador (en inglés, debugger), es un programa que permite depurar o limpiar los errores en el código.

DWR - Es una librería de Java que provee la interacción de llamadas a procedimientos de Java del lado del servidor y JavaScript del lado del navegador de la manera más simple posible.

Excepción - Una excepción es un evento inesperado que ocurre durabte la ejecución de un programa, y que interrumpe el flujo normal de las instrucciones del programa.

Framework - En el desarrollo de software, es una estructura de soporte definido, mediante la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

HQL – Lenguaje Hibernate de Consultas SQL

IDE - Un ambiente integrado de desarrollo conocido también como IDE es una aplicación de software que facilita la construcción de software. Provee un editor, compilador, debugger.

Interfaz - Es la forma en que los objectos definen su interacción con el exterior de la clase a través de los métodos que exponen.

JSP - Servidor de páginas Java.

MVC – Modelo Vista Controlador.

ORM – Mapeo Relacional de Objetos (Object Relational Mapping)

POJO – Objeto Plano construido en Java únicamente con propiedades y sus respectivos setters y getters. Carente de métodos que implementan algoritmos.

RPC - Llamada a procedimiento remoto.

Servlet - Los servlets son objetos que corren dentro del contexto de un contenedor de servlets, por ejemplo Tomcat y extienden su funcionalidad.

Widget - Es una pequeña aplicación o programa, usualmente presentado en archivos o ficheros pequeños que son ejecutados por un motor de widgets o. Entre sus objetivos están los de dar fácil acceso a funciones frecuentemente usadas y proveer de información visual

XML - Es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C).

Fuentes bibliográficas

Bauer Christian y King Gavin, *Java Persistence with Hibernate*, United States of America, Manning Publications Co. 2007.

Sierra Kathy y Bates Bert, *Sun Certified Programmer for Java 6 Study Guide,* United States of America, Mc Graw Hill, 2008.

Massol Vincent y Husted Ted, *JUnit in Action,* United States of America, Manning Publications Co. 2004.

Shea Frederick, Colin Ramsay y Blades Steve, *Learning Ext JS*, Birmingham-Mumbai, Packt Publishing, 2008.

Walls Craig y Breidenbach Ryan, *Spring in Action*, Segunda Edición, United States of America, Manning Publications Co. 2008.

Informix-4gl Reference Manual, Menlo Park, CA., AT&T, 1990.

Alur Deepak, Cupri John, Malks Dan, *Core J2EE Patterns best practices and design strategies*, United States of America, Sun Microsystems.

Frank W. Zammetti, *Practical DWR 2 Projects*, United States of America, 2008

Sam-Bodden Brian, *Beggining POJOs From Novice to Professional,* United States of America, Apress. 2006.

Modelo Vista Controlador http://java.sun.com/blueprints/patterns/MVC-detailed.html julio 2009

Objeto de Acceso a Datos
http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html julio 2009

DWR http://directwebremoting.org/dwr/introduction/index.html julio 2009