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

SPRING FRAMEWORK COMO BASE DE UNA ARQUITECTURA PARA APLICACIONES *WEB*

Nelson Moris Larin Reyes

Asesorado por el Ing. Bryan Orellana Soberanis

Guatemala, junio de 2011

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

SPRING FRAMEWORK COMO BASE DE UNA ARQUITECTURA PARA APLICACIONES *WEB*

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

NELSON MORIS LARIN REYES
ASESORADO POR EL ING. BRYAN ORELLANA SOBERANIS

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, JUNIO DE 2011

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Ing. Alfredo Enrique Beber Aceituno
VOCAL II	Ing. Pedro Antonio Aguilar Polanco
VOCAL III	Ing. Miguel Ángel Dávila Calderón
VOCAL IV	Br. Juan Carlos Molina Jímenez
VOCAL V	Br. Mario Maldonado Muralles
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Murphy Olympo Paiz Recinos
EXAMINADORA	Inga. Virginia Victoria Tala Ayerdi
EXAMINADOR	Ing. Edgar Estuardo Santos
EXAMINADOR	Ing. Manuel Aroldo Castillo
SECRETARIA	Inga. Marcia Ivónne Véliz Vargas

HONORABLE TRIBUNAL EXAMINADOR

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

SPRING FRAMEWORK COMO BASE DE UNA ARQUITECTURA PARA APLICACIONES _WEB_

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha noviembre de 2010.

Nelson Moris Larin Reyes

Guatemala, 28 de Marzo de 2011

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

Respetable Ingeniero:

Por medio de la presente hago de su conocimiento que he revisado a detalle y apruebo el trabajo de graduación realizado por el estudiante **Nelson Moris Larin Reyes**, quien se identifica con el carné número 200412591, y cuyo título es **"Spring Framework como base de una arquitectura para aplicaciones web".**

Sin otro particular, me suscribo atentamente,

Bryan Alexis Orellana Soberanis
Ingeniero en Ciencias y Sistemas
Colegiado No. 10394

Ing. Bryan Orellana Soberanis
Colegiado No. 10394

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

Guatemala, 27 de Abril de 2011

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

Respetable Ingeniero Pérez:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación del estudiante **NELSON MORIS LARIN REYES,** carné **2004-12591,** titulado: **"SPRING FRAMEWORK COMO BASE DE UNA ARQUITECTURA PARA APLICACIONES WEB",** y a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

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

Atentamente,

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

ESCUELA DE CIENCIAS Y SISTEMAS

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÌA
ESCUELA DE CIENCIAS Y SISTEMAS
TEL: 24767644

El Director de la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor y del Licenciado en Letras, de trabajo de graduación titulado ***"SPRING FRAMEWORK COMO BASE DE UNA ARQUITECTURA PARA APLICACIONES WEB",*** *presentado por el estudiante NELSON MORIS LARIN REYES, aprueba el presente trabajo y solicita la autorización del mismo.*

"ID Y ENSEÑAD A TODOS"

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION DE INGENIERIA EN CIENCIAS Y SISTEMAS

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

Guatemala, 10 de junio 2011

Universidad de San Carlos
de Guatemala

Facultad de Ingeniería
Decanato

DTG. 186.2010

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería en Ciencias y Sistemas, al trabajo de graduación titulado: ***SPRING FRAMEWORK* COMO BASE DE UNA ARQUITECTURA PARA APLICACIONES *WEB*,** presentado por el estudiante universitario **Nelson Moris Larin Reyes,** autoriza la impresión del mismo.

IMPRÍMASE:

Ing. Murphy Olympo Paiz Recinos
Decano

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANO
FACULTAD DE INGENIERIA

Guatemala, 13 de junio de 2011.

/gdech

ACTO QUE DEDICO A:

Dios Por ser el centro de mi vida y guiarme en cada paso que he dado.

Mis padres Nelson Larin y Claudia Reyes de Larin, por darme la vida, su amor, su apoyo y por la buena educación brindada; que me ha llevado a plantearme y cumplir con esta meta.

Mi hermana Claudia Steffi Larin, porque es parte importante de mi vida y quiero compartir este logro con ella.

Mi abuela Amelia Beatriz Cabrera, por su inmenso cariño y apoyo incondicional en todos los momentos de mi vida.

Mis amigos María José Recinos, Hesler Solares, Otto Santizo, Luis Carlos Pérez y Kenny Aguilar, por todas las experiencias compartidas que fortalecieron nuestra amistad, sé que puedo contar con ustedes. Espero que nuestra amistad prevalezca.

AGRADECIMIENTOS A:

Dios	Por permitirme cumplir las metas que me he propuesto y compartir este logro con las personas que quiero.
Mis padres	Por todo su apoyo, por invertir en mis estudios y darme todo lo necesario para alcanzar este día.
Mis amigos	Por todos los desvelos, risas, éxitos y fracasos compartidos; que hicieron que el duro camino se sintiera corto. También gracias a sus familias que me recibieron en sus hogares mientras trabajábamos en tareas y proyectos**.**
María José Marroquín Sarti	Por su apoyo, su compañía y gran cariño brindado durante todo este tiempo.
Bryan Orellana	Por todo su apoyo y asesoría en la realización del presente trabajo de graduación.
María José Recinos	Por su apoyo, ayuda y confianza en mí para terminar este proyecto.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

GLOSARIO

AOP	*Aspect-oriented programming* (programación orienta aspectos) es un paradigma de programación que tiene como objetivo aumentar la modularidad, permitiendo la separación de preocupaciones transversales. *AOP* constituye una base para el desarrollo de *software* orientado a aspectos.
AJAX	*Asynchronous JavaScript And XML (JavaScript* asincrónico y *XML)*, es una técnica de desarrollo *web* para crear aplicaciones interactivas o RIA
HTML	*HyperText Markup Language* (Lenguaje de Marcado de Hipertexto), es el lenguaje de marcado predominante para la elaboración de páginas *web*.
iBATIS	Es un marco de código abierto basado en capas desarrollado por *Apache Software Foundation*, se ocupa de la capa persistencia. Puede ser implementado en Java y .NET
Informática	Es la ciencia que trata la gestión automática de la información, utilizando dispositivos electrónicos y sistemas computacionales.

Internet	Es un conjunto de redes interconectadas por un protocolo TCP/IP que funciona como una red lógica única de alcance mundial.
IOC* / *IoC	*Inversion of control* (inversión de control) es un principio abstracto que describe un aspecto de algunos diseños de arquitectura de *software* en el que se invierte el flujo de control de un sistema en comparación con la programación procedimental.
JDBC	*DataBase Connectivity*, Controlador de bases de datos para lenguaje de programación Java.
POJO	*Plain Old Object* (objeto java plano viejo) es un acrónimo creado por Martin Fowler, Rebecca Parsons y Josh McKenzie en septiembre de 2000 y utilizada por programadores Java para enfatizar el uso de clases simples y que no dependen de un *framework* en especial.
POO	Programación orientada a objetos.
RIA	*Rich Internet Application* (Aplicaciones ricas de *internet*) es una aplicación *web* que tiene muchas de las características de las aplicaciones de escritorio, por lo general entregados ya sea a

	través de un navegador de sitios específicos, a través de un navegador *plug-in*, areneros independientes o máquinas virtuales.
SQL	*Structured Query Languaje*, (lenguaje de consulta estructurado) lenguaje de consulta de base de datos.
WWW	El *World Wide Web* comúnmente conocido como la *Web*, es un sistema de hipertexto enlazado y accesibles a través de internet.

RESUMEN

Una aplicación de *software* se puede definir como una herramienta que permite llevar a cabo uno o diversos trabajos, se puede decir que es un conjunto de utilitarios que brindan apoyo en tareas asociadas a un negocio, originalmente estas solo existían para ser ejecutadas en un computador, actualmente el paradigma se está migrando a la *web*.

La tendencia de las aplicaciones a nivel mundial indica que la accesibilidad es un activo de gran valor cuando se trata de aplicaciones, es por esto que las empresas productoras de *software* ahora se convierten en proveedoras de servicios vendiendo sus aplicaciones de forma *on-line*, brindando el servicio en cualquier lugar por medio de la *Web*; ya las aplicaciones de escritorio son un paradigma poco utilizado; como un caso de estudio tenemos a la Superintendencia de Administración Tributaria (SAT), en la cual el 80% de sus aplicaciones son orientadas a la *web*, debido a la facilidad de actualización y centralización de la información que proveen.

Este trabajo de investigación busca proponer una arquitectura para desarrollo de aplicaciones *web*, la cual pueda servir como base para la creación de múltiples aplicaciones, aportando a la aplicación de fácil mantenibilidad, poco acoplamiento y configuración sencilla. Estas características las brinda el *framework Spring*, por lo cual se adoptará como la base para la arquitectura, interactuando con el *framework Server Faces* como la cara de las aplicaciones, los *faces* brindan gran cantidad de funcionalidad. Esta arquitectura será una plantilla para desarrollos rápidos de aplicaciones

transaccionales dentro de la *WEB*, brindando una guía para los nuevos desarrolladores en el paradigma *web.*

OBJETIVOS

General

Proponer una arquitectura para desarrollo de aplicaciones *Web*, integrando atributos de calidad, teniendo como base el *framework Spring*, brindando una guía de desarrollo para el programador.

Específicos

1. Mostrar en detalle una arquitectura para aplicaciones *Web*, implementando la inversión de control y el modelo vista controlador, buscando poco acoplamiento entre capas, fácil configuración y mantenibilidad de la aplicación.

2. Realizar el análisis de una aplicación *web* de la Superintendencia de Administración Tributaria e identificar las características que le brinda el utilizar inversión de control y el modelo vista controlador.

3. Establecer los criterios que pueden servir para la elección de *Spring Framework* como base de una arquitectura, así como la adopción de la arquitectura propuesta como propia de la aplicación.

4. Proveer un manual/guía para el desarrollo de aplicaciones *web*, utilizando la arquitectura propuesta, que provee de atributos de calidad esenciales para una aplicación óptima, orientado a desarrolladores de aplicaciones *web*.

INTRODUCCIÓN

La Arquitectura de *Software* es el más alto nivel de diseño de un sistema, que consiste en un grupo de patrones y abstracciones que brindan un marco de trabajo y referencia, que funge como guía en la construcción. Se establecen fundamentos de importancia para diseñadores, analistas, programadores. La selección de una arquitectura se basa en los objetivos y restricciones que se poseen para un sistema. En este trabajo de graduación se busca una arquitectura que cumple con atributos de calidad.

Principalmente se busca separar, promover el poco acoplamiento entre capas, de la forma más simple y limpia posible, con especial atención en la facilidad de mantenimiento y evolución de las aplicaciones. Además que la misma aplicación pueda ser de fácil implantación, utilizar tecnologías que brinden sistemas ligeros y desacoplados.

Dentro de las aplicaciones *web* en su mayoría tiene una presentación, una lógica detrás de ésta y una base de datos, ya que el manejo de datos e información es importante hoy en día por lo cual, en el momento de decidir que capas utilizaremos, se debe de analizar ¿Qué tipo de negocio tiene? ¿Qué tanta lógica posee? ¿Los accesos a datos son muchos o pocos? ¿La presentación es muy complicada? ¿Es necesario crear componentes específicos? ¿Es necesario agregar seguridad?, de esta manera se deciden las capas que deben utilizarse.

Esta arquitectura toma como base tres capas, haciendo semejanza al modelo vista controlador, que utiliza tres categorías para la distinción de los

elementos vista, modelo y controlador, para fines de la arquitectura se utilizarán la capa de negocio, capa de persistencia y la capa de presentación.

1. MARCO CONCEPTUAL

1.1. Introducción a la programación orientada aspectos

1.1.1. Evolución de la programación

En los inicios de la informática, los desarrolladores y programadores escribían programas directamente en código de máquina, desafortunadamente esto conllevaba a pasar más tiempo pensando en cómo escribir el código, que en el problema en sí. Poco a poco se ha migrado a lenguajes de alto nivel en el cual tomamos solo una abstracción del código. Luego se enfocó en los lenguajes estructurados, estos permiten descomponer los problemas en los procesos necesarios para resolverlos, lo que conllevó a una mayor complejidad, ahora se necesitaba mejores técnicas para resolverlos.

La programación orientada a objetos deja ver a un sistema como un conjunto de objetos que interactúan entre sí, las clases que permiten ocultar implementaciones por debajo de las interfaces. El polimorfismo provee un comportamiento e interfaz común para objetos relacionados y permite mayor detalle en componentes sin la necesidad de cambiar el comportamiento particular de cada uno de los objetos.

Las metodologías de programación y los lenguajes definen la manera de comunicarse con las máquinas, cada nueva metodología presenta una manera distinta o nueva de afrontar los problemas a solucionar, es posible hablar de código de máquina, código independiente de la máquina, procedimientos, clases y así podemos mencionar muchas más. Cada metodología hace un

acercamiento más natural en la forma de solucionar las problemáticas, lo cual nos permite cada vez crear sistemas con mayor complejidad.

Actualmente, la programación orientada a objetos sirve como la metodología que mayormente se utiliza para el desarrollo de proyectos, este tipo de programación ha demostrado que posee gran fortaleza para modelar comportamiento común de los sistemas. Sin embargo, como se verá en el desarrollo de este trabajo, no es suficiente para abordar todos los comportamientos que poseen los sistemas, *AOP,* programación orientada a aspectos cubre esos deficiencias que tiene el POO, lo cual nos lleva al siguiente gran paso en la evolución de las metodologías de programación.

1.1.2. Programación orientada a aspectos

En un inicio con programación OO se tuvo un gran avance en el desarrollo de proyectos, los desarrolladores podían visualizar el sistema como un grupo de entidades y las interacciones entre ellas, lo que permite resolver sistemas muy complicados, en menor tiempo de desarrollo. El único problema para esta metodología es que esencialmente es estática, un cambio en los requerimientos puede ser un gran impacto en desarrollo.

La programación orientada a aspectos complementa a la programación OO ya que permite al desarrollador modificar dinámicamente el modelo planteado, así el sistema puede crecer en requerimientos nuevos o la modificación de los iniciales. Justamente, como los objetos en la vida real cambian sus estados durante su ciclo de vida, una aplicación puede adoptar nuevas características en su desarrollo.

Considérese un ejemplo: casi todos los desarrolladores han creado una aplicación *Web* sencilla, utilizando *servlets* como el punto de entrada, este recibe valores de un formulario *HTML*, relacionando el formulario con un objeto, pasando la información a la aplicación para ser procesada, y esta devuelve una repuesta al usuario.

La primera versión del *servlet* debe ser muy simple, solamente cumpliendo con lo que el caso de uso necesita para ser modelado. El código se triplica o hasta cuadruplica de su cantidad inicial debido a los requerimientos secundarios como manejo de excepciones, seguridad, *logging*, etcétera, ¿Por qué requerimientos secundarios? Debido a que el *servlet* no tiene necesidad de poseer estos mecanismo, solamente su función principal que es la de obtener información y procesarla.

AOP permite dinámicamente modificar el modelo estático, para poder cumplir con los requerimientos secundarios, sin la necesidad de modificar el modelo inicial. Aun mejor, podemos tener este código centralizado en una sola implementación en lugar de en cada uno de los elementos como fuese necesario en OO. Es posible definir la programación orientada a aspectos en los siguientes conceptos:

- ✓ *Advice (consejo)*: son todos los aspectos que no deben estar dentro de la base del sistema, hablamos de *Logging* y acciones extras fuera de la lógica de la aplicación.

- ✓ *Cross-cutting concerns* (problema cruzado): son todas aquellas funciones secundarias que posee una aplicación, en las cuales es necesario realizar acciones que no son propias de cada opción, podemos hablar de *loggeo*,

seguridad, etcétera, donde el código es muy similar o el mismo para las distintas opciones.

- ✓ *Point-cut* (punto de corte): es en el punto en el cual es necesario la utilización de los consejos, donde se inicia a utilizar la codificación que no es única para cada opción, sino es utilizada para toda la aplicación.

- ✓ *Aspect* (aspecto): combinar *Point-cut* y *Advice* se conoce como un aspecto, es ya la utilización de los elementos para realizar las tareas secundarias.

Estos conceptos brindan una primera vista de *AOP*.

1.2. Introducción a inversión de control

1.2.1. Patrones de diseño

Una definición muy básica de un patrón de diseño es “Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de *software*”, ¿a qué se refiere con esto?, generalmente cuando se crea una aplicación se cae en la cuenta que se encuentra problemas recurrentes, por lo cual se utiliza un patrón que solucione estos problemas, la solución que proveen está debidamente probada y documentada, lo que agiliza el desarrollo e implementación.

Existen 3 tipos de patrones:

- ✓ Patrones creacionales: configuración e inicialización de objetos (ej. *Abstract Factory, Builder, Factory Method*).

- ✓ Patrones estructurales: realizan una separación de interfaz e implementación, para formar estructuras más grandes (ej. *Adapter, Bridge, Decorator*).

- ✓ Patrones de comportamiento: buscan una comunicación adecuada entre objetos, (ej. *Interpreter, Iterator, Observer*).

1.2.2. Inyección de dependencia

Tradicionalmente una clase se encarga de crear instancias de los objetos para utilizarlos, en la inyección de dependencias ocurre lo contrario, a la clase se le suministran los objetos que utilizará, esto se hace por medio de un contenedor de inyección de dependencias, este le inyecta a cada objeto lo necesario para realizar su función, según lo definido en archivos de configuración previamente estipulados.

1.2.3. Inversión de control

Basado en la inyección de dependencias, este principio permite crear aplicaciones con poco acoplamiento, el beneficio de este es la no necesidad de hacer muchos cambios para adaptar una nueva funcionalidad, un cambio de esta o la relación entre una aplicación y su utilización en otra.

¿Cómo funciona?, la inversión de control consiste en, como su nombre lo indica, invertir como una clase obtiene las referencias o configuraciones de objetos externos, tradicionalmente una clase utilizaba los objetos externos creando instancias de ellos y apuntando directamente al objeto, ocasionando una gran dependencia entre los objetos, altos costos por modificaciones o inclusión de nuevas funcionalidades, utilizando *IoC* solo tiene referencias a interfaces y los objetos concretos son proveídos externamente, lo cual se traduce en poca dependencia y poco acoplamiento.

De esta manera se especifican solicitudes de datos o respuestas deseadas durante el tiempo de ejecución de la aplicación, dejando en manos de entidades externas el control de las acciones que deban de suceder, esta entidad externa es llamada contenedor, entre estos tenemos PicoContainer[1] y *Spring*.

1.3. Breve descripción de la arquitectura *Spring Framework*

Un marco de trabajo según Wikipedia es “un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular, que sirve como referencia para enfrentar y resolver nuevos problemas de índole similar”[2], y si se enfoca hacia el desarrollo de *software* nos dice “una estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de *software* concretos, con base en la cual, otro proyecto de *software* puede ser organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros programas para ayudar a desarrollar y unir los diferentes componentes de un proyecto”.

[1] PicoContainer - Marco de trabajo, que implementa inyección de dependencias. http://picocontainer.org/

[2] http://es.wikipedia.org/wiki/Framework

Spring es un marco de trabajo que promueve las mejores prácticas de programación, proporciona una gestión adecuada de los recursos y las excepciones, además de ejemplos de cómo utilizar con eficacia *IoC*, *AOP* y la programación orientada a objetos. Se puede catalogar como un contenedor de *IoC* y un marco de *AOP*, pero este va más allá, ya que busca simplificar el desarrollo en J2EE.

Aspectos importantes, que muchos marcos utilizados comúnmente no toman en cuenta, *Spring* lo hace; se centra en buscar la forma más adecuada de administrar los objetos de negocio, provee de una arquitectura en capas, lo cual brinda la posibilidad de utilizar cada una de forma aislada, sin crear conflictos en su comunicación, creando una arquitectura internamente consistente, ¿Qué ventajas trae esto? Fácilmente se puede adaptar parte de este marco de trabajo a nuestros proyectos, puede utilizarse solo para la persistencia de los datos, el acceso a ellos, o solo para manejar la lógica del negocio del mismo.

El aprendizaje de esta arquitectura se hace muy fácil, debido a que se toma cada parte por separado, conociendo de a poco como gestionar las capas, no es necesario conocer todo desde un inicio. Así, la migración de un proyecto hacia esta arquitectura se hace gradual, haciéndola hasta invisible para los usuarios finales.

Este marco de trabajo se construye desde cero, así se logra facilidad para realizar pruebas del mismo, es ideal para proyectos en los cuales se realizan pruebas unitarias y de negocio. *Spring* es una mezcla de tecnologías y cada versión hace una fusión de nuevas tecnologías buscando la optimización de las aplicaciones.

1.3.1. Beneficios de la arquitectura *Spring*

Para optar por un patrón de diseño o un marco de trabajo de *software*, siempre es necesario saber en qué ayudará, por lo cual estos son algunos de los beneficios que provee:

- ✓ Evita el uso del patrón de diseño *Singlenton*, ya que este reduce la capacidad de realizar pruebas.

- ✓ Organiza efectivamente las capas de nivel medio, así como su conexión con las demás, tomando importancia en su configuración, los servicios que estas capas pueden brindar, pueden ser utilizados en cualquiera de las otras capas del sistema.

- ✓ Reduce los costos de programación a interfaces en lugar de a clases en concreto, promoviendo las buenas prácticas de programación.

- ✓ Puede eliminar la necesidad de utilizar varios y distintos formatos de archivos de propiedades, creando un manejo de configuración constante a través de proyectos y aplicaciones, ya no es necesario buscar por todas partes de un proyecto como funciona o como se configura una clase, simplemente con leer las propiedades de los JavaBeans o los argumentos de un constructor se puede saber. El uso de inversión de control y la inyección de dependencias ayuda a lograr esta simplificación.

- ✓ Facilidad de realizar pruebas unitarias.

- ✓ Está diseñado para no depender de sus propias librerías.

- ✓ Tiene independencia de implementación, ya que es posible usar EJB, POJOs, interfaces, sin la necesidad de modificar la el código de llamada de los mismos.

- ✓ Provee un marco de trabajo consistente para el acceso a datos, con la utilización de *JDBC* o cualquier manejador de objetos relacionales como *Hibernate*, TopLink u objetos de datos Java.

- ✓ Proporciona un modelo de programación coherente y simple, por lo cual es considerado como un pegamento arquitectónico ideal.

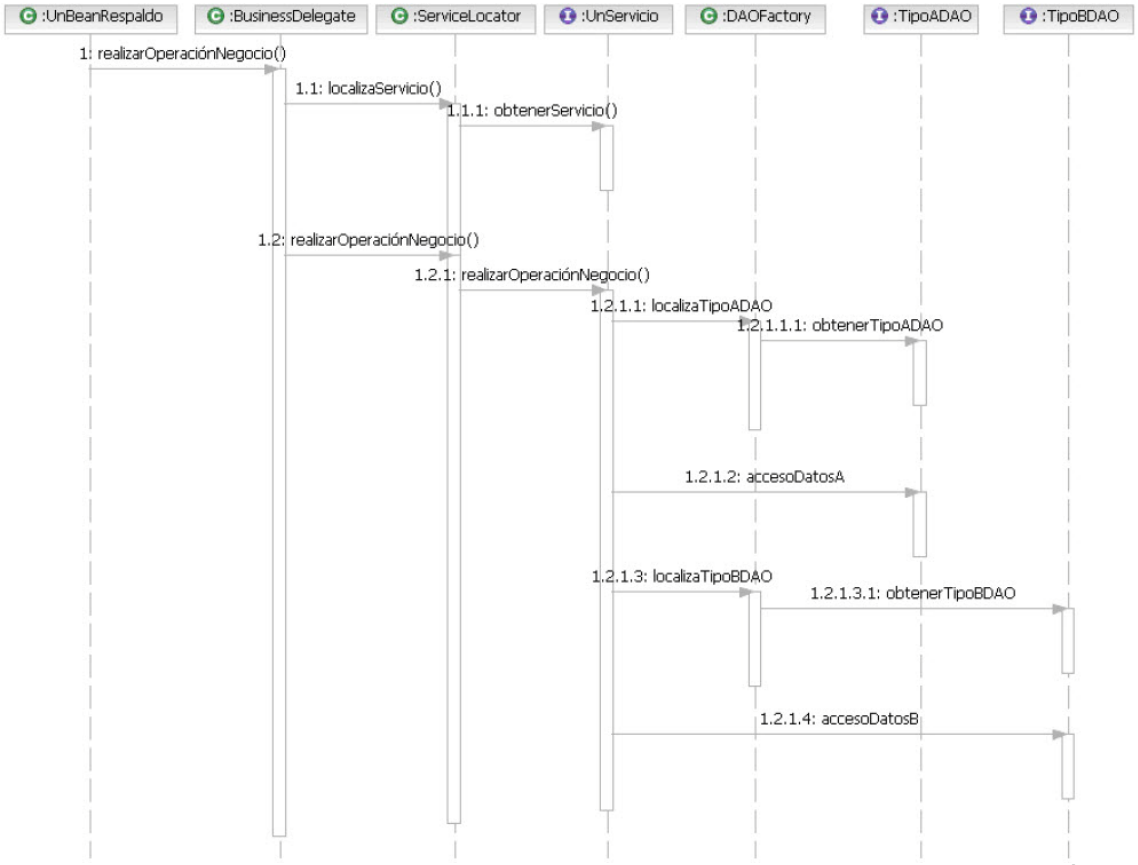
1.3.2. ¿Qué hace *Spring*?

Como objetivo principal se tiene: hacer más fácil la utilización de J2EE y promover las buenas prácticas de programación, asimismo no intenta crear la rueda, por lo mismo dentro de *Spring* no se encontrarán paquetes para el *logging*, conexiones de base de datos o coordinadores de transacciones distribuidas. Todas estas cosas son provistas por soluciones de código abierto o por el servidor de aplicaciones. Ya que existen buenas soluciones para estas problemáticas o requerimientos secundarios.

Spring busca hacer más fácil la utilización de las tecnologías existentes, como un ejemplo puede verse que no se enfoca en la coordinación de transacciones, sin embargo proporciona una capa de abstracción sobre *JTA*. No busca la competencia con otros marcos de trabajos de código abierto, simplemente toma lo que en muchas ocasiones los desarrolladores expresan, como ejemplo puede verse que la evolución hacia un contendor de *IoC* liviano, fue gracias a que en *Struts* se veía como un conflicto tener que montar un contenedor tan pesado.

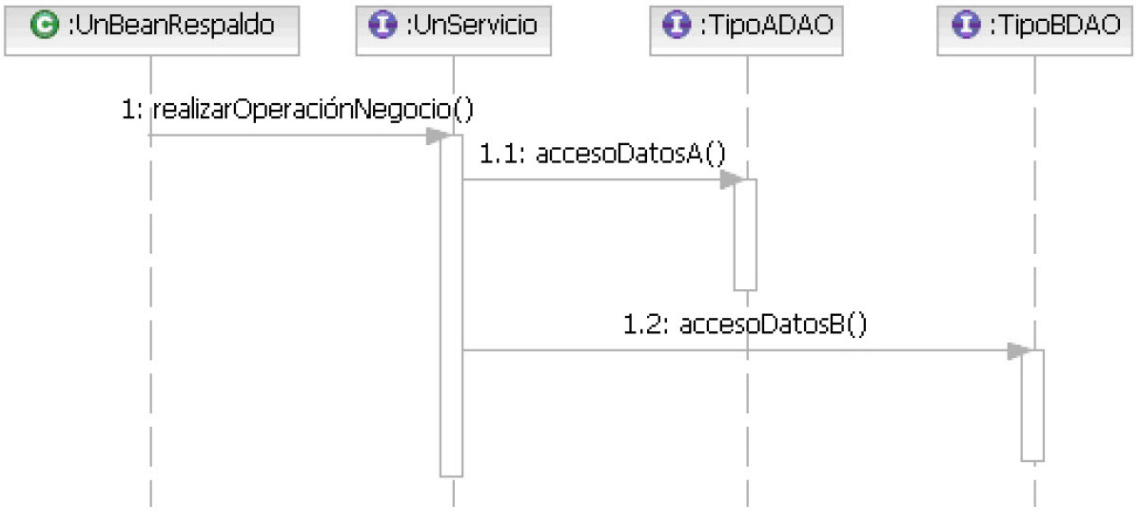
La portabilidad que posee entre servidores de aplicación, lo hace un marco de trabajo muy adaptable no solo a recursos de hardware, sino recursos monetarios, asegurar la portabilidad es un reto, pero este busca evitar cualquier dependencia de alguna plataforma en específico o vistas no estándares para los desarrolladores.

Figura 1. **Invocación operación a negocio en arquitectura clásica**



Fuente: hacia una arquitectura con *JavaServer* faces, Spring, Hibernate y otros, “Invocación de operación de negocio en arquitectura clásica.”, mayo 2006.

Figura 2. **Invocación de operación en arquitectura con *Spring***



Fuente: hacia una arquitectura con *JavaServer* faces, Spring, Hibernate y otros, "Invocación de operación de negocio en arquitectura clásica.", mayo 2006.

1.4. Frameworks para presentación (*JavaServer Faces*)

La capa de presentación o interfaz de usuario generalmente es la última parte de un proyecto que se realiza y tiende a ser muy dependiente de las herramientas de desarrollo que se han utilizado, sin embargo, suele suceder que es necesario desconectar la interfaz de usuario y reemplazarla por otra que cumple con las expectativas del cliente, la usabilidad o los estándares.

La lógica de presentación brinda la capacidad de tomar datos, transformarlos en entradas para acciones del negocio o para hacer uso de un servicio, algunas de las responsabilidades que se pueden ver para esta capa están:

- ✓ Validación de datos
- ✓ Formateo
- ✓ Usabilidad

- ✓ Estilo
- ✓ Transformación

1.4.1. Introducción a *JavaServer Faces*

Marco de trabajo dedicado a la interface de usuario del lado del servidor para crear aplicaciones *web* con base en *Java*. *JSF* como se le denomina, contiene una interfaz de programación de aplicaciones (*API*) para representar componentes en la interfaz de usuario (UI) y manejar eventos, estados, validación de datos y transacciones del lado del servidor, conversión y transformación de datos, definir la navegación entre las páginas de la aplicación, internacionalización y así mismo brindar extensibilidad.

Entre sus características principales se encuentran:

- ✓ *JSP* es la base de todas sus vistas, añade etiquetas a la biblioteca para crear elementos en los formularios.

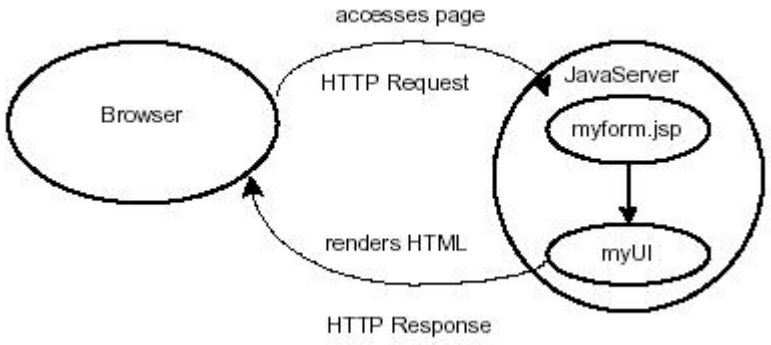
- ✓ Utiliza un archivo de tipo *XML* para la configuración del controlador.

- ✓ Es posible la agregación de nuevos elementos para la interfaz o modificar los ya existentes, agregarles funcionalidad o estilo.

- ✓ Introduce etapas al proceso de petición, creando un ciclo de refrescamiento de la vista.

- ✓ Asocia a cada una de las páginas o vistas con forms[3], estos son enlazados con controladores (*managed beans*) que facilitan la toma de datos, manipulación y visualización de datos.}

Figura 3. **Ciclo de acceso a una página con *JSF***



Fuente: propia.

Este marco de trabajo permite el desarrollo ágil de aplicaciones con negocios dinámicos, en las que la lógica detrás de la interfaz se maneja en *Java*. Por lo cual permite utilizar el *JavaScript* para hacer la página o la vista que se está presentando más rápida y ágil, evitando peticiones al servidor.

1.4.2. Como funciona *JSF*

Para cada aplicación *web* que se construye es necesario un conjunto de pantallas que son la cara al usuario de la aplicación, estas son con las cuales el usuario final interactúa. Estas pantallas contienen todos los componentes para poder visualizar datos, imágenes, ingreso o selección de datos, tablas, Etc. Todos estos elementos y componentes se encuentran en *HTML*[4], para ser más

[3] Form: Formulario utilizado en las java server pages (*JSP*).
[4] *HTML*: lenguaje de marcado de hipertexto.

específico en formularios, de esta forma es como se maneja él envió de la información hacia el servidor.

Como función principal del controlador *JSF* se busca la asociación de pantallas (páginas) y las clases de *Java* que contiene la información introducida por el usuario, así como los métodos que se ejecutan con las acciones del usuario. *JSF* brinda el soporte para mostrar datos a usuario en tablas, cajas de texto, realizar la recolección de datos, controlar el estado de cada uno de los componentes según el estado que la aplicación se encuentre, renderizando los controles adecuados, realizar validación y conversiones de los datos que el usuario ingresa.

Básicamente *Server Faces* utiliza:

- ✓ *Server Pages* como base, incluyendo los formularios de *JSF.*
- ✓ Clases nativas de *Java* para la implementación de la lógica de negocio.
- ✓ Java *beans* para conectar los formularios *JSF.*
- ✓ Archivos para la configuración de la aplicación.

1.4.3. Implementaciones de *JSF*

Existe una gran variedad de librerías de componentes para implementación con *JSF*, estas buscan brindarle más riqueza a la interfaz de usuario, funcionalidad, usabilidad y además ayudar al desarrollador a tener un entorno de desarrollo más adecuado y fácil.

Entre estas tenemos:

- ✓ *Rich Faces*: es una librería de componentes de código abierto, organizada por Jboss.org, brinda fácil integración con *AJAX* para el desarrollo de aplicaciones empresariales.

- ✓ *Ice Faces*: es un marco de trabajo de *AJAX* de código abierto, que permite crear aplicaciones *Rich Internet Applications (RIA),* las cuales poseen capacidades de una aplicación de escritorio en el internet.

- ✓ *MyFaces*: proyecto de Apache Software Foundation, es su implementación más básica, con varias ramas.

    - ➢ *Tomahawk*: componentes creados por *MyFaces*.
    - ➢ *Trinidad*: componentes desarrollados y aportados por *Oracle*[5], antiguamente ADF.
    - ➢ *Tobago*: componentes contribuidos por Atannion GmbH[6].

Rich Faces será la implementación que se utilizará para el desarrollo de la guía, por lo cual el enfoque será conocer más de ella.

1.4.4. Rich Faces

Esta es una implementación de componentes visuales para *JSF*, escrita por Exadel originalmente quien fuera adquirida por *JBoss*, posee un marco de trabajo para la integración con *AJAX*, por soporte de la librería *Ajax4JSF*[7].

[5] Oracle: empresa centralizada en el desarrollo de manejadores de base de datos, y software. http://www.oracle.com/us/index.html

[6] Atanion GmbH: empresa desarrollo de software alemana.

Rich Faces se caracteriza por integrarse perfectamente con el ciclo de vida de *JSF*, sin crear saltos o comportamientos extraños en las peticiones, incluye la funcionalidad ya mencionada con *AJAX*, siendo transparente para el desarrollador el código *JavaScript* que se ejecuta, ya que utiliza la librería *Ajax4JSF* la integración es transparente, soporta la implementación de temas y máscaras por medio de archivos de estilo CSS.

Contiene un set de componentes visuales bastante amplio con 212 componentes, los más utilizados para el desarrollo de aplicaciones RIA, cumpliendo casi todas las necesidades de un desarrollador. Debido a su carácter *open source* posee un crecimiento bastante aceptable en la comunidad. Y como bien se sabe cada una de las tecnologías debe tener algún inconveniente, por lo que para *Rich Faces* es la necesidad de indicar al momento de utilizar *AJAX* que se repintará.

Especificaciones técnicas de *Rich Faces*:

- ✓ Versión de java soportada: JDK 1.5 o mayor.

- ✓ Implementaciones y marcos de trabajo de *JSF* soportadas
  - *Sun JSF*-RI – 1.2_x
  - *MyFaces* 1.2.x, 2.x
  - *Facelets* 1.1.x
  - *Seam* 2.x

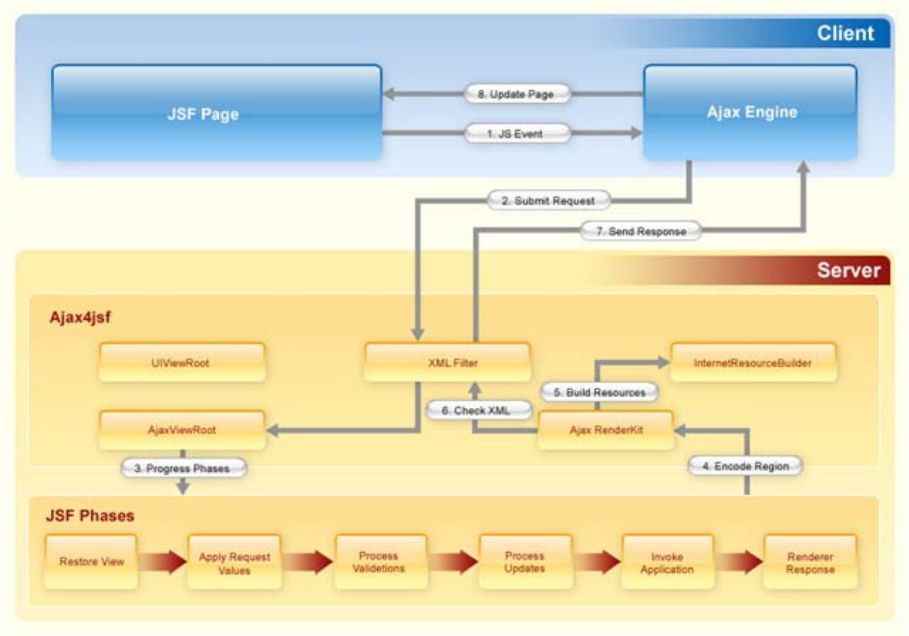
[7] Ajax4jsf: “es una librería open source que se integra totalmente en la arquitectura de JSF y extiende la funcionalidad de sus etiquetas dotándolas con tecnología Ajax de forma limpia y sin añadir código JavaScript.” Fuente http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=Ajax4Jsf

- ✓ Servidores soportados
  - ➢ *Apache Tomcat 5.5 – 6.0*
  - ➢ *Jetty* 6.1.x
  - ➢ *Sun Application Server 9*
  - ➢ *Glassfish V2, V3*
  - ➢ *JBoss 4.2.x – 5*
  - ➢ *Websphere 7.0**
  - ➢ *Gerónimo 2.0**

- ✓ Exploradores soportados
  - ➢ Ambiente Windows
    - ▪ *Firefox 3.0**
    - ▪ *Google Chrome*
    - ▪ *Internet Explorer 6.0**
    - ▪ *Opera 9.5**
    - ▪ *Safari 3.0**
  - ➢ Ambiente Linux
    - ▪ *Firefox 3.0**
    - ▪ *Opera 9.5**
  - ➢ Ambiente Mac OS
    - ▪ *Safari 3.0**
    - ▪ *Firefox 3.5**

Nota: * versión descrita o superior.

Figura 4. **Funcionamiento de *Rich Faces***



Fuente: propia.

1.5. Herramientas para persistencia (*JPA*, *ORM*)

Toda aplicación sea ambiente *web* o de escritorio tiene la necesidad de mantener información en una base de datos, por lo cual se ve, a su vez, en la necesidad de utilizar herramientas para esto; en un principio se utilizaron las conexiones nativas, sentencias nativas de SQL para persistir y consultar los datos, sin embargo esto se hacía muy tedioso, largo y poco reutilizable, por lo cual se tuvo la necesidad de crear herramientas que manejen la persistencia.

Entre algunas de las herramientas existentes: *TopLink, Cocobase y FastObjects, Kodo, JDO Genie, Exdel JDO, TJDO, iBatis, Hibernate* y *JPA*. Todas estas específicamente son utilizadas para el mapeo objeto relacional.

1.5.1. Mapeo objeto-relacional (*ORM - Object Relational Mapping*)

Técnica utilizada en programación para convertir datos de la base de datos a objetos, esto en búsqueda de una facilidad en el manejo de los datos y acoplar la misma a la programación orientada a objetos, creando una base de datos virtual, la cual puede ser usada por el lenguaje de programación.

1.5.2. Java Persistence API (JPA)

La programación en Java permite la solución de problemas de negocio por medio de objetos, que por consiguiente poseen estado y comportamiento, sin embargo, los modelos de datos relacionales poseen una estructura distinta, mediante tablas, filas y columnas, de esta manera es necesario crear un mapeo entre el modelo de datos y objetos de java. *JPA* es una abstracción de *JDBC* que da la facilidad de crear el mapeo objeto relacional, se configura por medio de metadatos (archivos *XML* o anotaciones). Este *API* establece un medio o interfaz que es utilizado por los demás *frameworks* de persistencia.

1.6. Modelo vista controlador (MVC)

Es un patrón de diseño usado generalmente en aplicaciones *Web*, el cual busca separar los objetos que conforman la aplicación en tres categorías modelo, vista y controlador, debido a esta separación varias vistas y controladores pueden interactuar con el mismo modelo, el desacoplamiento que brinda este patrón es una de las características de mayor aprovechamiento, además de la reutilización que brinda de componentes.

1.6.1. Modelo

Esta categoría o capa del patrón se encarga de:

- ✓ El acceso a la capa de almacenamiento, idealmente se busca que sea complemente independiente del sistema de almacenamiento.

- ✓ Define el negocio en sí de la aplicación, tiene las validaciones, y procesos internos de los datos.

- ✓ Registro de vistas y controladores.

1.6.2. Controlador

Esta capa del patrón es el responsable de:

- ✓ Recibir los eventos de entrada, clicks, cambios en texto, selección en un *check box*, Etc.

- ✓ Posee las acciones luego que suceda un evento; aquí es donde se indica que se debe hacer en el momento que suceda un evento X, estas pueden ser peticiones al modelo o a la vista. Una llamada a una vista puede ser la invocación de un método como Actualizar(), que puede ser el cambio de estado de algún componente, una invocación al modelo significa un proceso más largo, la búsqueda de información, inserción de datos, etcétera.

1.6.3. Vista

La vista como su nombre lo indica es lo que un usuario final observa, con lo que interactúa, por lo cual es responsable de:

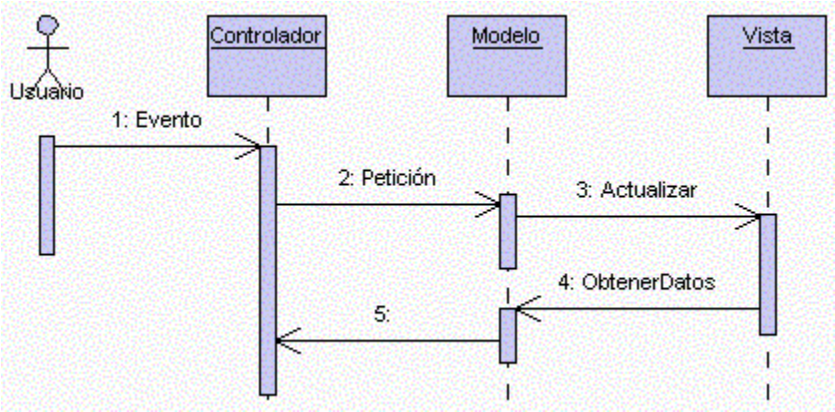
- ✓ Recibir datos del modelo y mostrarlos al usuario.
- ✓ Registro de controlador asociado.

1.6.4. ¿Cómo funciona MVC?

El funcionamiento de este se basa en los eventos producidos por el usuario, generalmente existe una reacción debida a una acción del cliente. Los pasos que sigue este modelo son:

a. Introducción de un evento por parte del usuario.
b. Controlador recibe el evento y traduce este a la petición o acción previamente programada, esta puede ser al modelo o vista.
c. Modelo si fue invocado, llama a la vista para actualización.
d. La vista depende de si la acción solicita datos al modelo.
e. El controlador recibe el control.

Figura 5. **Funcionamiento de modelo vista controlador (MVC)**



Fuente: funcionamiento de modelo vista controlador, http://www.proactiva-calidad.com/java/patrones/imagenes/mvc_secuencia.gif

Esta forma desacoplada de interactuar es lo que permite realizar cambios en las distintas capas sin la necesidad de alterar las demás.

1.7. Maven

Es una herramienta que brinda la facilidad de gestionar y construir proyectos Java, es muy similar a la función del Apache Ant, posee un modelo de configuración muy simple, basada en un archivo *XML*.

Project Object Model (POM) es el artefacto con el cual se describe el proyecto a construir, las dependencias, componentes externos, orden de construcción de cada elemento, además posee objetivos predefinidos. Una característica muy importante es que tiene la capacidad de trabajar en red, el motor que posee dinámicamente realiza descargas de *plugins*, actualizaciones directamente de un repositorio, se ha vuelto la forma en la cual se distribuyen las aplicaciones de Java.

Posee 2 fases para la obtención de las dependencias, una local en la cual se poseen las librerías y proyectos que se trabajan por la misma empresa, y una remota en la cual se suben los artefactos terminados para el acceso generalizado, actualmente la mayoría de proveedores de *software* brindan repositorios de *Maven* para el acceso al público.

1.7.1. El POM

Este es la unidad fundamental para trabajar con *Maven*, con estructura *XML*, contiene toda la información sobre el proyecto y configuraciones detalladas del mismo. Este fue la evolución de project.xml en *Maven* 1, ahora en *Maven* 2 toma el nombre de pom.xml, además se le agregó lo que antes se definía en el archivo maven.xml, como los objetivos de ejecución y compilación.

Ejemplo de un POM mínimo:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

2. DISEÑO DE ARQUITECTURA BASADA EN ATRIBUTOS DE CALIDAD

2.1. Atributos de calidad

Estas son características que permiten verificar y medir el grado de satisfacción de los usuarios y/o diseñadores con respecto al sistema de *software*, esta es la definición que nos da la ISO 9126.

Estos como su nombre lo indica, permiten medir y controlar la calidad de un sistema. Pueden ser vistos en el sistema como cualidades acorde a lo que el usuario solicite, estas cualidades son disponibilidad, modificabilidad, desempeño, seguridad, facilidad de pruebas y usabilidad.

2.1.1. Disponibilidad

Es la probabilidad que un sistema tenga la capacidad de operar en el momento que sea requerido, bajo circunstancias específicas. Las fallas ocurren cuando el sistema no brinda el servicio en el tiempo especificado, una falla como tal es observada por los usuarios finales del sistema.

Para tener un control de este atributo es necesario contar con un manejo apropiado de las fallas que el sistema puede presentar, se debe de identificar varios aspectos como: ¿Quién o como se detecta una falla? ¿Qué tan frecuente puede ocurrir? ¿Qué sucede en el sistema cuando la falla ocurre? ¿Qué acciones tomar en caso de falla? etc. Debido a esto es necesario la diferenciación entre fallas y defectos, un defecto puede ser una falla si este no

se corrige. Esto nos lleva a que un defecto no es observable por el usuario, las fallas sí.

2.1.2. Modificabilidad

La modificabilidad en un sistema se refiere al costo en que se incurre para realizar un cambio. Cuando se habla de costo, se refiere al desembolso monetario, tiempo de ejecución del cambio y recursos humanos necesarios para realizarlo. Cuando se habla de alta capacidad de modificabilidad, se refiere a que tan fácil es hacer un cambio, sustituir un componente sin que este afecte a los demás que están relacionados con él.

Se debe tomar en cuenta ciertas consideraciones al momento de realizar un cambio, como: ¿Qué elementos pueden cambiarse? En un sistema seguramente se harán cambios, continuamente surgen nuevos requerimientos, nuevas opciones, modificación de requerimientos, cambio de necesidades, lo que hace que cualquier elemento del sistema pueda cambiarse. Como consecuencia de un cambio es necesario evaluar los efectos que este tiene, que elementos son afectados directa e indirectamente.

¿Quién debe hacer el cambio? y ¿Cuándo se debe hacer? En vista que los sistemas tienen gran amplitud, todo dependerá de que es lo que se desea cambiar, con esto se define quien y cuando realizarlo, al decir cuándo se refiere básicamente en qué fase del sistema se realiza el cambio.

2.1.3. Desempeño

Se referente a los eventos que ocurren y a los cuales el sistema debe responder. Se puede definir como el tiempo que le toma al sistema responder a

un evento o suceso. Se vuelve complicado por la cantidad de eventos que pueden existir en la ejecución de una aplicación, así como las distintas fuentes de eventos. Otro factor importante es como los recursos son administrados, de esta parte depende en gran proporción el tiempo y la forma en que el sistema responda a una petición.

2.1.4. Seguridad

Los usuarios no autorizados son quienes ponen a prueba la seguridad de un sistema, la habilidad de resistir estas peticiones y aun así proporcionar servicios a los usuarios autorizados. Un intento de violar la seguridad se le denomina como ataque o amenaza y este puede darse en varias formas, con objetivos distintos.

La seguridad en sistema se caracteriza por proveer cualidades como:

- ✓ El no rechazo: se refiere a que un sistema no puede negar una función o transacción por otra función.

- ✓ Confidencialidad: es la protección de los datos y servicios y el acceso adecuado a los mismos, mediante los accesos autorizados para un usuario específico.

- ✓ Integridad: hace referencia a que los datos no sean alterados, ni corrompidos en el funcionamiento o envío.

- ✓ Confianza: es asegurar que las partes involucradas en una transacción reciban exactamente la misma información.

- ✓ Disponibilidad: especifica condiciones en las que el sistema estará disponible a los usuarios.

2.1.5. Facilidad de pruebas

Para que exista facilidad de probar un sistema, este debe poder controlar los estados internos de cada componente, las entradas y así poder observar las salidas. Esto se realiza utilizando algún *software* el cual nos permita saber si los componentes hacen lo que deben hacer y responden como se espera.

2.1.6. Usabilidad

¿Qué tan fácil es usar un sistema? Es a lo que se refiere la usabilidad, con que simplicidad el usuario final puede realizar la tarea deseada, se ha enfatizado en tres puntos de usabilidad.

- ✓ Uso eficiente de un sistema: las características que un sistema debe tener para lograr que los usuarios sean más eficientes en sus operaciones.

- ✓ Incremento de satisfacción y confianza: que hace el sistema para brindar confianza, para que el usuario tenga confianza en el sistema.

- ✓ Minimización del impacto de error: que puede hacer un sistema para que un error de usuario tenga mínimo impacto, o es más que no lo cometa.

2.2. Búsqueda de las capas adecuadas

Principalmente se busca separar, promover el poco acoplamiento entre capas de la forma más simple y limpia posible, con especial atención en la

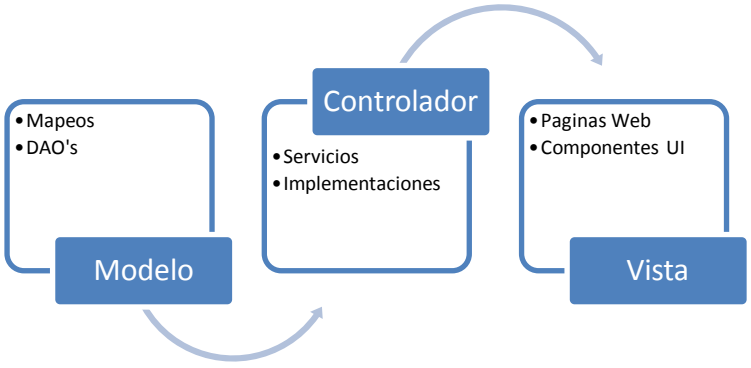
facilidad de mantenimiento y evolución de las aplicaciones. Además que la misma aplicación pueda ser de fácil implantación utilizando tecnologías que nos brinden sistemas ligeros y desacoplados.

Las aplicaciones *web* en su mayoría tiene una presentación, una lógica detrás de esta y una base de datos, tomando en cuenta que el manejo de datos e información es importante hoy en día, al momento de decidir que capas utilizaremos, se debe de analizar ¿Qué tipo de negocio tiene? ¿Qué tanta lógica posee? ¿Los accesos a datos son muchos o pocos? ¿La presentación es muy complicada? ¿Se necesitan componentes específicos? ¿Qué nivel de seguridad requiere? de esta manera se deciden las capas que deben utilizarse.

Esta arquitectura toma como base tres capas, haciendo semejanza al modelo vista controlador, que utiliza tres categorías para la distinción de los elementos vista, modelo y controlador, para fines de la arquitectura se utilizarán la capa de negocio, capa de persistencia y la capa de presentación.

2.3. Bosquejo de arquitectura

Figura 6. **Bosquejo de arquitectura MVC**



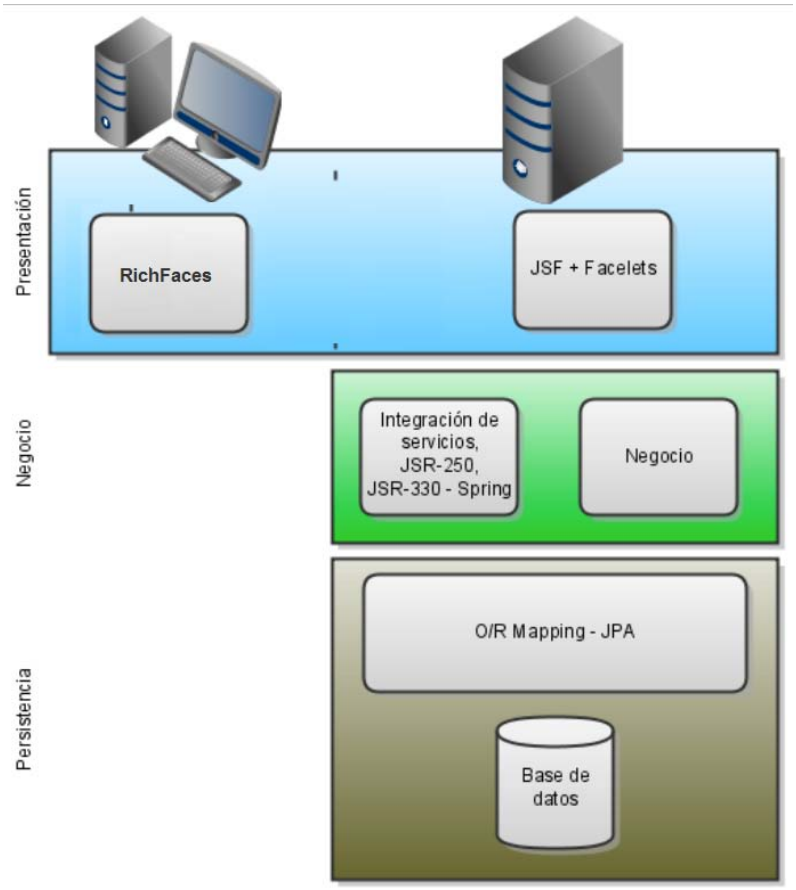
Fuente: propia.

2.4. Propuesta de arquitectura para aplicación *web*

Se propone una arquitectura basada en las tecnologías *JSF* y *RichFaces*, integración de servicios con *Spring* y persistencia con *JPA* teniendo el soporte de *Hibernate*. Estas tecnologías son de código abierto por lo cual su utilización no incurre en costos, son de fácil acceso y además de amplia utilización, por lo cual se consideran como adecuadas para el desarrollo de aplicación *web*.

En la imagen que se presenta a continuación se muestra como cada una de las tecnologías se utilizan en las capas adecuadas:

Figura 7. **Integración de las capas y sus elementos**



Fuente: propia.

2.4.1. Capa de presentación

En esta capa se recopilan los datos de entrada de los usuarios, presenta datos, controla la navegación por las páginas y delega la entrada del usuario a la siguiente capa. Esta se define como la cara de la aplicación, valida la entrada del usuario y mantiene la sesión de la aplicación.

Se debe implementar utilizando el patrón de diseño MVC debido a su adaptabilidad para aplicaciones interactivas en Java, logra separar los conceptos de diseño, por lo que reduce la codificación redundante, además permite a la aplicación ser más extensible. MVC ayuda a que los desarrolladores se enfoquen en su mayor habilidad, ya que divide el trabajo en 3 partes, el modelo, la vista y el controlador. Este es el diseño que se utilizará para la capa de presentación.

JSF como marco de trabajo, provee de componentes de interface de usuario los cuales son manejables por Java, brinda administración de estados, eventos, validaciones, internacionalización. La utilización de esta tecnología se debe a que encaja perfectamente en el patrón de diseño seleccionado, ya que hace una clara separación entre el comportamiento y la presentación. También une los componentes de la vista con los conceptos del modelo, sin limitar a la utilización de script o lenguaje de marcas.

Los *beans* que utiliza *JSF* hacen la función del modelo en el MVC, estos contiene acciones que son un agregado o extensión del controlador, permiten delegar peticiones de usuario a la capa de negocio. Cabe mencionar que la capa de negocio en ocasiones puede fungir como el modelo de la capa de presentación. Algunas de las características del por qué se utiliza esta tecnología son:

- ✓ Utilización de *beans* de respaldo, los cuales separan la definición de los objetos componentes del UI de los objetos que realizan el procesamiento.

- ✓ Configuración a la medida en los componentes existentes.

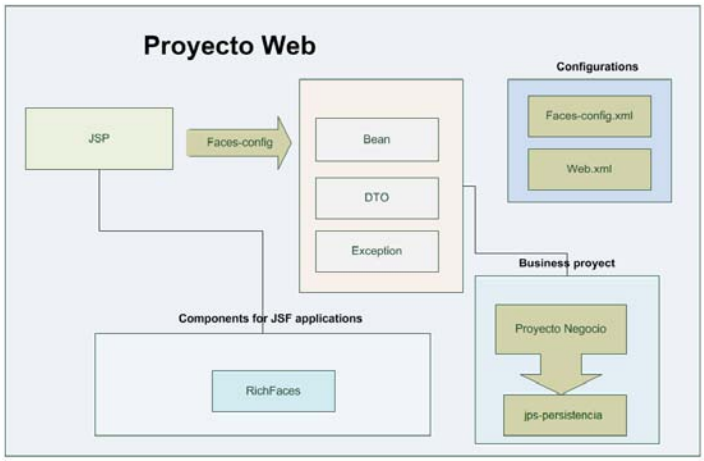
- ✓ Modelo de renderizado flexible, lo que permite hacer renderizaciones parciales y crear apariencias distintas de un mismo componente.

- ✓ Validación y conversión extensible.

Dentro de la arquitectura se maneja un *bean* de respaldo, el cual define las propiedades y la lógica de manejo de componentes utilizados en una página. Cada propiedad en este está unida a un componente o a su valor. Este también posee los métodos que realizan acciones y funciones para el componente, como pueden ser validaciones de datos y manejo de eventos.

En la siguiente figura se muestra cómo funcionan todos los componentes en la capa de presentación.

Figura 8. **Vista de la capa de presentación**



Fuente: propia.

2.4.2. Capa de negocio o modelo

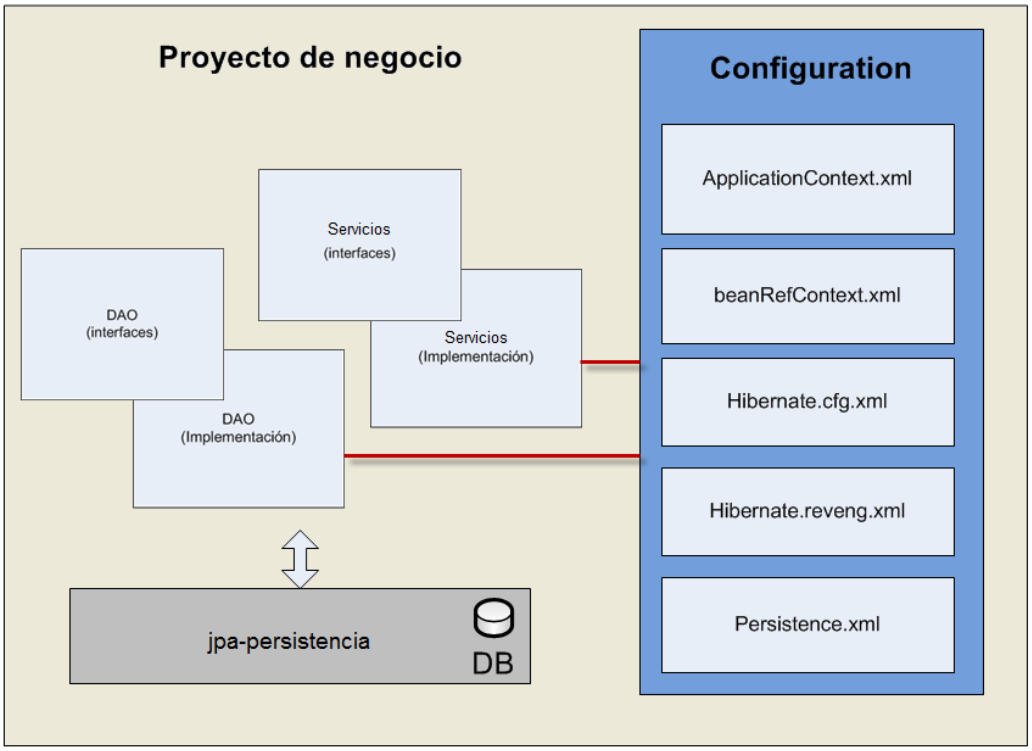
Los servicios, implementaciones y objetos que contiene la capa de negocio generalmente son la lógica detrás de la aplicación, un objeto además contiene datos. Los servicios de negocio se comunican con otros objetos de negocio, que proporcionan una lógica a un nivel más alto. Par lo cual se debería definir una capa de interfaz formal de negocio, la cual incluya las interfaces de servicio que el cliente usara directamente.

Spring organiza de forma efectiva los objetos que se encuentran en esta capa, por lo cual se implementa para aprovechar los recursos de mejor manera, se busca la eliminación de instancias únicas y utilizar buenas prácticas en la programación orientada a objetos.

Un manejo óptimo de las transacciones se implementa a través del manejo declarativo de *Hibernate*, la aplicación de *AOP* ofrece un control de transacciones sin la utilización de un contenedor EJB, *Spring* provee diversidad en el manejo de las transacciones sin embargo se utilizará *JTA* para las mismas.

Esta capa debe estar compuesta con todos los servicios e implementaciones que permitan el acceso y manipulación de los objetos modelados, aquí se define como se comportan los objetos o datos ingresados por el usuario, se generan transformaciones, se ejecutan transacciones, validaciones de datos y toda la lógica necesaria para que las reglas del negocio se cumplan. Esta capa puede ser accedida desde la vista para realizar transacciones en la misma, como en la capa de persistencia.

Figura 9. **Vista de la capa de negocio**



Fuente: propia.

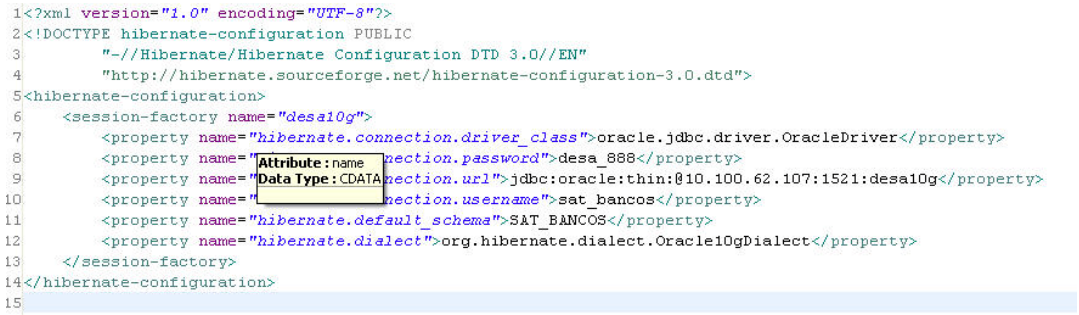
Para realizar mapeo de las tablas, se puede hacer por medio de una ingeniería inversa, la cual automáticamente nos crea las anotaciones necesarias dentro de la clase para conectarla a la base de datos, es necesaria la configuración de los siguientes archivos:

- ✓ *Hibernate*.cfg.xml
- ✓ *Hibernate*.reveng.xml
- ✓ Persistence.xml

El archivo de configuración *Hibernate*.cfg.xml se utiliza para enviarle parámetros al motor de *Hibernate*, casi siempre el archivo se encuentra en el directorio de las fuentes del sistema, de esta manera garantiza que aparezca en

la raíz del *classpath* cuando sea compilada, aquí se define la conexión que se utilizará para realizar la ingeniería inversa.

Figura 10. **Archivo de configuración hibernate.cfg.xml**

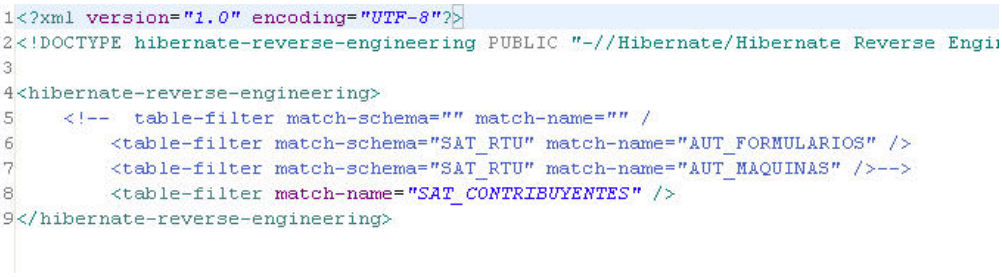



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="desa10g">
        <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <property name="hibernate.connection.password">desa_888</property>
        <property name="hibernate.connection.url">jdbc:oracle:thin:@10.100.62.107:1521:desa10g</property>
        <property name="hibernate.connection.username">sat_bancos</property>
        <property name="hibernate.default_schema">SAT_BANCOS</property>
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
    </session-factory>
</hibernate-configuration>
```

Fuente: propia.

Además se debe de definir qué tablas y con qué clases se conectará por lo cual se deben de configurar dos archivos más hibernate.reveng.xml y el persistence.xml, en el primero se coloca el nombre de las tablas y el *schema* al que pertenecen, en el segundo se colocan las clases con las cuales se amarran.

Figura 11. **Archivo hibernate.reveng.xml**



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse Engi

<hibernate-reverse-engineering>
    <!--  table-filter match-schema="" match-name="" /
        <table-filter match-schema="SAT_RTU" match-name="AUT_FORMULARIOS" />
        <table-filter match-schema="SAT_RTU" match-name="AUT_MAQUINAS" />-->
        <table-filter match-name="SAT_CONTRIBUYENTES" />
</hibernate-reverse-engineering>
```

Fuente: propia.

Figura 12. **Archivo persistece.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
    <persistence-unit name="persistencia" transaction-type="RESOURCE_LOCAL">
        <class>gt.gob.sat.RegistroTributario.modelo.AutFormularios</class>
        <class>gt.gob.sat.RegistroTributario.modelo.AutMaquinas</class>
    </persistence-unit>

</persistence>

```

Fuente: propia.

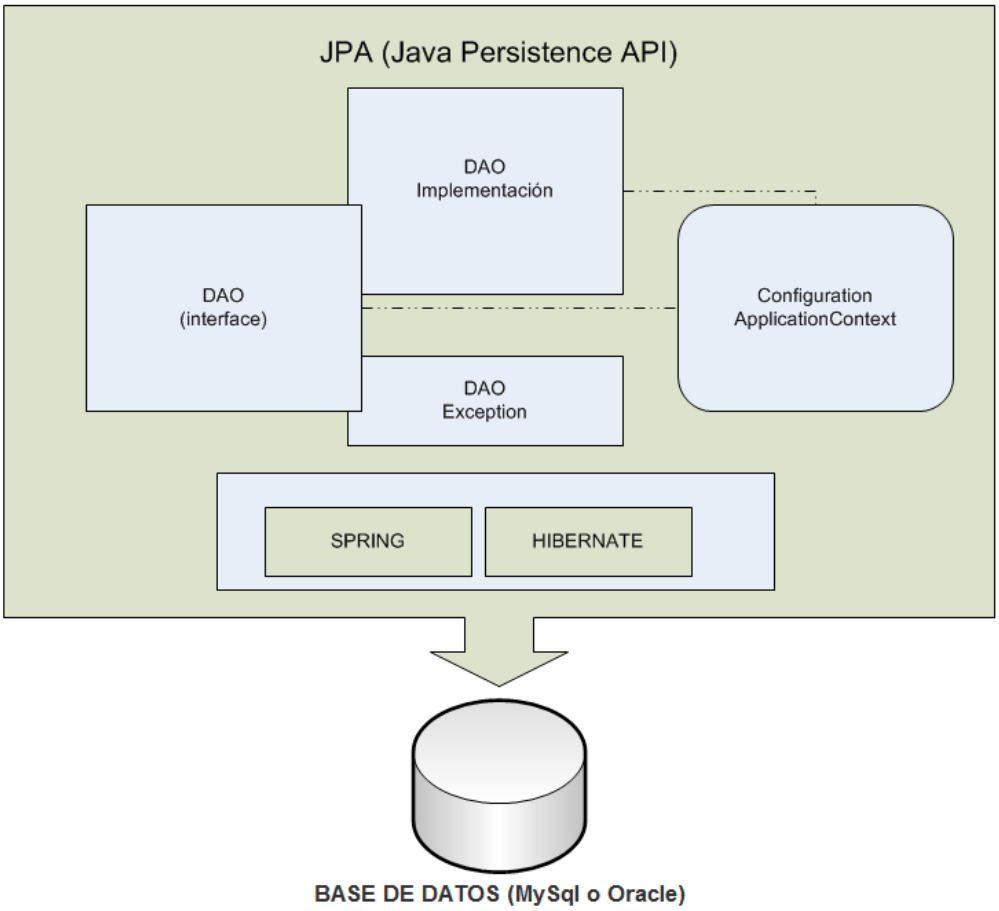
2.4.3. Capa de persistencia

En esta capa se busca unificar la manera como se acceden a los datos, generalmente se define un tipo de persistencia por proyecto, se busca que se la óptima para cada aplicación, esta capa trata la manera de ser lo más general posible para promover la reutilización de la misma. Esta capa cuenta con una clase *DAO* la cual contiene los servicios necesarios para la generar transacciones en base de datos. Entre estos están:

- ✓ Persist (Entity)
- ✓ Remove(Entity)
- ✓ Update(Entity)
- ✓ findAll ()
- ✓ findAllByQuery(query)

La arquitectura para esta capa se muestra en la siguiente gráfica, se hace la utilización de *Hibernate* como puente hacia la base de datos, así como el soporte brindado por *Spring* para *ORM*, que se comunica de muy buen manera con *JPA*, estos servicios son puestos a disposición del *Application Context*.

Figura 13. **Vista de la capa de persistencia**



Fuente: propia.

2.5. Integración de capas mediante archivos de configuración

La inversión de control o inyección de dependencias como se menciona en el capítulo anterior brinda poco acoplamiento, por lo cual es parte de la arquitectura propuesta, ¿Cómo se utiliza? ¿Qué se debe de configurar? ¿Qué archivos mapean la *IoC*? prácticamente viene inferido en el *framework* de *Spring*.

Existen dos formas en las cuales *Spring* provee la *IoC*, una de ellas se basa en la utilización del *BeanFactory,* este provee un soporte básico para la inyección de dependencias, es responsable de crear y dispensar los *beans*, manejar las asociaciones entre los mismos, se considera un contenedor muy liviano por lo que se recomienda en situaciones de escasos recursos, la utilización más común es en base a *XMLBeanFactory*. Generalmente para aplicaciones sencillas.

En la arquitectura se utilizará el *contexto de aplicación (Application Context),* tiene una aplicación en sistemas empresariales y de mayor complejidad, ya que nos brinda más funcionalidad que el *BeanFactory.*

Brinda un medio para la resolución de mensajes de texto, que abarca la internacionalización de los mensajes, una forma genérica para la carga de recursos de archivo, como imágenes. Permite publicar eventos hacia las *beans* que se registran como *listeners*.

2.5.1. *Web*.xml

Este archivo provee información para configuración y despliegue de los componentes *web*, que compromete una aplicación *web*. Este archivo debe estar en el directorio *WEB*-INF bajo el contexto de la aplicación.

Para la arquitectura es necesario configurar algunos parámetros que servirán para configurar la inyección de dependencia, *AOP* y otros filtros de seguridad por lo cual, a continuación se muestras estas configuraciones.

Configurar la utilización de *Application Context*,

<listener><listener-class>org.springframework.web.context.ContextLoaderListener</listener-class></listener>

Así como la ubicación del archivo de definición del mismo,

<context-param><param-name>contextConfigLocation</param-name><param-value>/WEB-INF/training-service.xml,/WEB-INF/training-data.xml</param-name></context-param>

Figura 14. **Configurar utilización de la implementación *RichFaces***

```
<!-- Making the RichFaces skin spread to standard HTML controls -->
<context-param>
    <param-name>org.richfaces.CONTROL_SKINNING</param-name>
    <param-value>enable</param-value>
</context-param>
<context-param>
    <param-name>org.ajax4jsf.SKIN</param-name>
    <param-value>classic</param-value>
</context-param>
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml</param-value>
</context-param>
```

Fuente: propia.

Figura 15. **Configuración del *listener* para el *JSF***

```
<listener>
    <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
```

Fuente: propia.

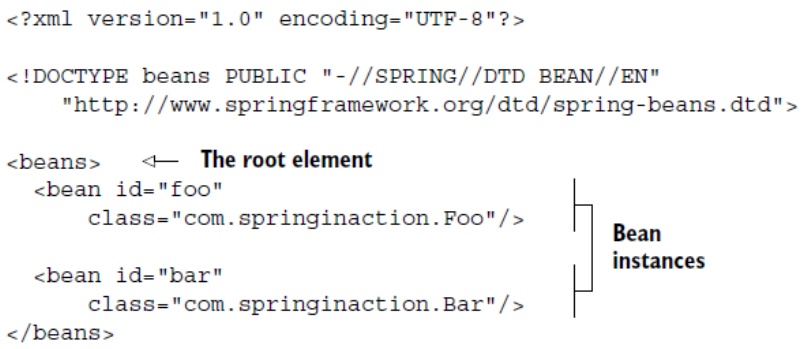
2.5.2. ApplicationContext.xml

En este archivo se debe configurar todos los *beans* que serán utilizados y expuestos para utilización de la aplicación, debido a que se utilizará el *Application Context* de *Spring* para la inyección de dependencias, es necesario

plasmar aquí que clases están expuestas, cual es el tipo de *bean* que se manejará y el método de inicialización y destrucción del *bean*. Previamente en el archivo web.xml está definido que se utilizará el *Application Context*, ahora se define el propio archivo que configura los *beans*.

Cada *bean* se mapea de la siguiente manera, en el archivo *XML* se coloca como inicial el tag <*beans*> y luego el <*bean*> individual, también se debe definir el *id* y la clase que se está utilizando de esa manera se instancia un *bean*.

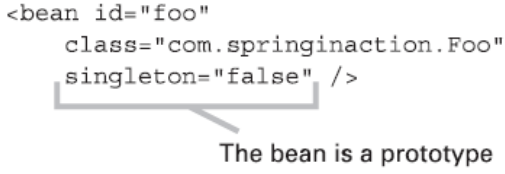
Figura 16. **Definición de un *bean***



Fuente: propia.

Se puede definir como el contenedor manejará los *beans*, si brindara el mismo *bean* cada vez que se solicite o desea una nueva instancia para cada solicitud esto se hacer por medio de la propiedad Singlenton.

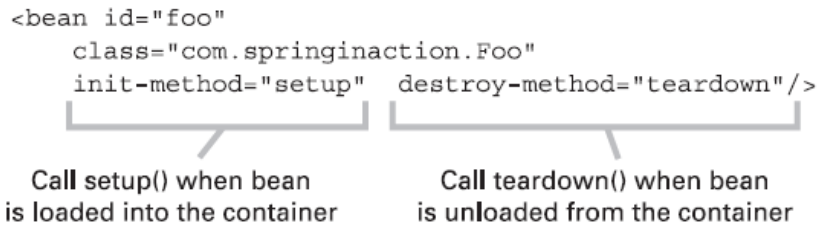
Figura 17. **Definición del tipo de *bean***



Fuente: propia.

Al momento de solicitar un *bean*, se puede configurar que inicie con un método, con el cual hará que el *bean* pueda tener el estado adecuado para utilizarlo, así como la eliminación o cambio de estado luego de que se elimine del contenedor. Propiedades *init-method* y *destroy-method*.

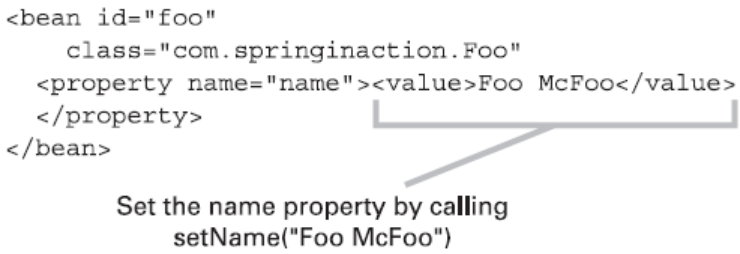
Figura 18. **Métodos de inicio y finalización**



Fuente: Spring in Action, 2006.

Se pueden configurar propiedades nativas para su utilización por medio del tag <property>.

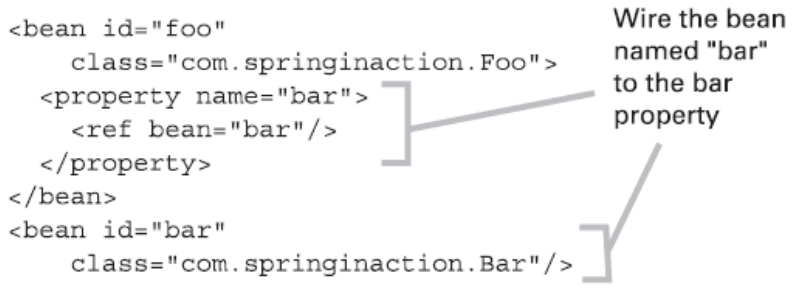
Figura 19. **Configuración de propiedades**



Fuente: Spring in Action, 2006.

De igual forma que las propiedades nativas, se puede hacer la utilización de otros *bean*, esto se hace de la misma manera por el tag <property>, con la diferencia que se debe hacer una referencia a otro *bean*.

Figura 20. **Configuración de propiedad vinculada**



Fuente: Spring in Action, 2006.

2.6. Criterios de evaluación para optar por una arquitectura

Criterios evaluados para adoptar *Spring* como base de la arquitectura:

2.6.1. Manejo de transacciones

Pueden utilizarse diferentes *ORM* como el propuesto que es *Hibernate*, además existen alternativas como *JDO*, *JDBC* y ODBC. También soporta *JTA*.

2.6.2. Oportunidad de transacciones

Puede soportar diferentes niveles de encapsulamiento y atributos de transacción. Las transacciones anidadas pueden ser soportadas si las implementa el manejador de transacciones.

2.6.3. Persistencia de entidades

Permite el uso de diversas de alternativas para el manejo de persistencia, así como la implementación de cualquier tecnología *ORM*.

2.6.4. Programación orientada a aspectos

Incluye aspectos personalizados que pueden definirse y también servicios de aplicación de forma declarativa.

2.6.5. Configuración de la aplicación

Permite la utilización de archivos *XML* en donde se definen los atributos de calidad y la configuración de los recursos. También programación mediante la *API* y un estándar JSR.

2.6.6. Seguridad

Proporciona integración con la solución de código abierto *Acegi*, la cual permite dar soporte tanto a la seguridad declarativa mediante el uso de *IoC* y *AOP*.

2.6.7. Flexibilidad de servicios

Puede ser acoplado cualquier servicio utilizando un archivo *XML* de configuración.

2.6.8. Integración de servicios

Por ser un *framework* abierto puede ser integrado fácilmente con otros *framework*s y además una aplicación *Spring* puede ser publicada en cualquier servidor de aplicaciones o incluso un contenedor de servlets.

2.7. Extendiendo la arquitectura

2.7.1. Seguridad

Se integrará con el gestor de seguridad *Acegi*, el cual está diseñado principalmente para ser usado con *Spring*. *Acegi* brinda una capa que incluye diversos estándares de seguridad incluidos en java como también ofrece una manera unificada para la configuración mediante descriptores *XML*.

Acegi resguarda la capa *web* como también la del negocio, en la parte *Web* captura las solicitudes por medio de la implementación de un filtro y para los métodos por medio de la contención a través de *AOP*. Para ambos escenarios permite utilizar los criterios de seguridad que incluye o bien agregar nuevas funcionalidades fácilmente implementando interfaces.

2.7.2. Loggeo

LOG4J es una biblioteca de código abierto que permite administrar los mensajes y avisos de una aplicación. Esta *API* es completamente configurable por medio de la utilización de archivos *XML*, también permite eliminar algunos mensajes o exportarlos a un archivo o a la base de datos.

Los niveles de prioridad que contiene Log4J se detallan a continuación por orden de menor a mayor detalle:

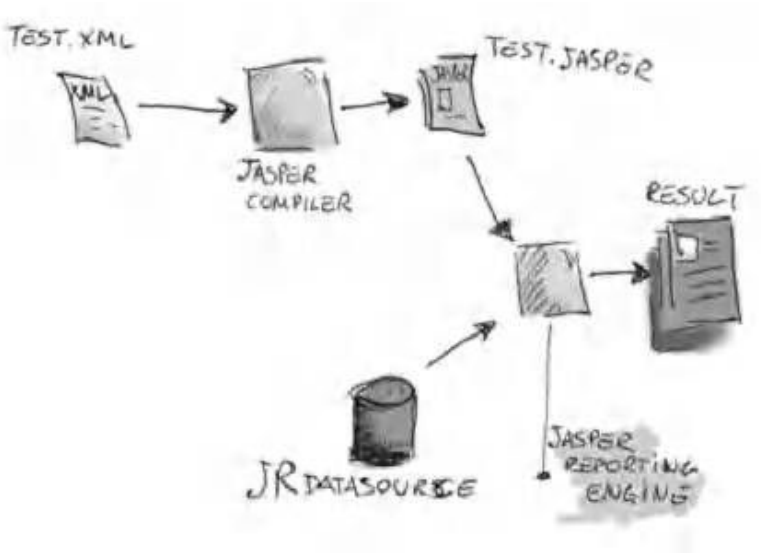
- ✓ OFF: deshabilita todos los logs.
- ✓ FATAL: se utiliza para los mensajes críticos del sistema.
- ✓ ERROR: se utiliza para los eventos que afectan al programa mas no interrumpen su disponibilidad.

- ✓ WARN: se utiliza para mensajes de alerta, y no afectan el correcto funcionamiento del sistema.
- ✓ INFO: se utiliza para informar sobre eventos dentro de la aplicación que son de cierto interés.
- ✓ DEBUG: se utiliza para mensajes útiles para depurar la aplicación.
- ✓ TRACE: incluye mayor detalle que el debug y también se utiliza para depurar la aplicación.
- ✓ ALL. es el nivel máximo de detalle y habilita todos los logs.

2.7.3. Reportería

Se utilizará la librería *Jasper Report*s la cual permite la generación de informes preparados para imprimir de forma fácil y sencilla. *Jasper Report* está escrito en Java y es libre, su funcionamiento consiste en recibir archivos *XML* que contienen los detalles del informe y como resultado se puede obtener mediante las clases de Jasper un *PDF*, *XML*, *HTML*, *CSV*, *XLS*, *RTF* o un TXT.

Figura 21. **Flujo de *Jasper Report***



Fuente: Spring in Action, 2006.

Como se puede notar *Jasper Report*s trabaja muy similar a un compilador y un intérprete, el usuario diseña el *XML* de acuerdo a las etiquetas y/o atributos, este archivo se compila para obtener un informe real el cual se convierte en el archivo fuente. Luego de esto se puede obtener datos para completar el informe, esto se realiza por medio del JRDataSource y al finalizar puede exportarse en los formatos ya descritos.[8]

2.7.4. *Web* services

Actualmente *Spring* brinda soporte completo para los servicios *web* incluidos en el *API* de Java. Con esto se logra exponer y acceder a los servicios *web* con la utilización de:

- ✓ JAX-RPC: *Spring* proporciona apoyo para la comunicación remota a través de *web* services, por medio de este *API* el cual se utiliza para la creación de servicios *web* y clientes que utilizan llamadas a procedimientos remotos (RPC) y *XML*. Se puede decir que se centra en los enlaces RPC.

- ✓ JAX-WS: es el sucesor del *API* anterior, para servicios *web* basados en *XML*. Este *API* es más flexible a la hora de consolidar la exposición y el acceso a los servicios.

[8]http://mygnet.net/articulos/java/introduccion_a_jasperreports_e_ireport_primera_parte.301

3. GUÍA PARA EL DESARROLLO CON LA ARQUITECTURA PROPUESTA

3.1. ¿Qué se necesita para trabajar?

Como entorno de desarrollo es necesario poseer:

- ✓ *IDE Eclipse Galileo**
- ✓ *MySql server versión 5.0* o Oracle 10g*
- ✓ *MySql Workbench 5.2 CE o SqlDeveloper*
- ✓ *Tomcat Apache 6.0**
- ✓ *Maven 2.0**

Todas estas herramientas son de código abierto, pueden conseguirse en sus páginas oficiales.

Esta arquitectura está basada en las *JSF*, *RichFaces*, *JPA* como partes fundamentales del mismo, por lo cual para el desarrollo es necesario poseer las librerías adecuadas:

- ✓ org.spring*framework* versión 2.5.* este *framework* de *Spring* provee integración entre otros con *Hibernate* y *jpa*, soporte para *DAO* y transacción. (http://www.springsource.org/download).
- ✓ org.*hibernate*.ejb3-persistence-1.0.2.GA se utiliza para persistir los datos de mapeos objeto/relación en conjunto con anotaciones propias de *Hibernate* (http://www.*hibernate*.org/downloads.html).
- ✓ jUnit 4.4 o superior, esta librería es utilizada para realizar pruebas unitarias.

- ✓ org.*richfaces* librería para la vista, implementación de *JSF* que se utilizará.
- ✓ org.aspectj.aspectjweaver-1.6.1 se utiliza para trabajar con programación orientada a aspectos.

3.2. Acceso a Datos

Dentro del servidor de aplicaciones, que para esta guía se utilizará Tomcat apache 6, se debe de crear un DataSource, el cual será la fuente de datos para el ejemplo.

Para esto es necesario modificar el archivo context.xml, el cual contiene la información de acceso a datos para el Tomcat, este se encuentra en TOMCAT_HOME/conf/ context.xml.

Se debe configurar un Resource, que está formado por las credenciales, driver, URL y parámetros de la conexión, así como un ResourceLink que es quien da el puente entre el recurso y la aplicación a éste se le configura el nombre del link, tipo y el entorno.

Figura 22. **Configuración de *DataSource Tomcat***

```
<Context>

    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

                <Resource name="jdbc/JPAPersistencia" auth="Container" type="javax.sql.DataSource"
                          driverClassName="oracle.jdbc.driver.OracleDriver"
                          url="jdbc:oracle:thin:@(DESCRIPTION_LIST =(DESCRIPTION = (ADDRESS = (PROTOCOL = tcp)(HOST = sa
                          username="AP_PERSISTENCIA"
                          password="AP_PERSISTENCIA_888"
                          maxActive="5"
                           maxIdle="3"
                          maxWait="10000"
                          useNaming="true"
                          removeAbandoned="true"
                          logAbandoned="true"/>

            <ResourceLink name="jdbc/JPAPersistencia" type="javax.sql.DataSource" global="jdbc/JPAPersistencia"/>
```

Fuente: propia

3.3. Creando las capas

Para la implementación del sistema es necesario crear tres proyectos, el proyecto *web* como la parte de presentación, manejo de *beans* de respaldo, archivos de mensajes, configuración *web*, Etc. El proyecto negocio con el que se tiene toda la lógica de negocio acceso a datos, mapeo de las tablas y además un proyecto de persistencia el cual maneja más a fondo las transacciones con la base de datos.

3.3.1. Creando los proyectos

Para crea un proyecto *Maven* con la arquitectura propuesta, se debe crear un proyecto desde línea de comando en el workspace de *Eclipse* con la siguiente instrucción:

mvn archetype:<tipo de ejecución> –DgroupId=<paquete> –DartifactId= <nombre del proyecto>

En donde:

- ✓ Tipo de ejecución: create (creación del proyecto).
- ✓ Paquete: mi.proyecto (paquete donde pondremos nuestras clases).
- ✓ Nombre del proyecto: nombre que quiere darle a su proyecto.

Estas sentencias crean los tres proyectos necesarios:

- ✓ mvn archetype:create –DgroupId=com.miempresa –DartifactId= *jpa*-persistencia

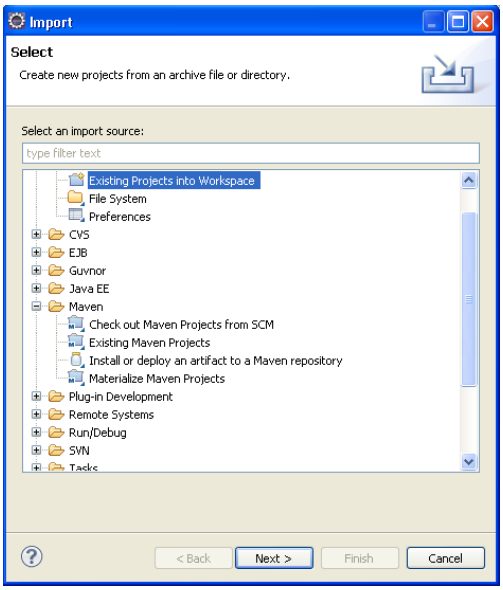
- ✓ mvn archetype:create –DgroupId=com.miempresa –DartifactId= *jpa-negocio*
- ✓ mvn archetype:create –DgroupId=com.miempresa –DartifactId= *jpa-web*

3.3.2. Importar proyectos a *Eclipse*

Para importar los proyectos creados con *Maven* a *Eclipse* se debe de seguir los siguientes pasos:

a. Menu File-> Import, seleccionar Existing Project into Workspace

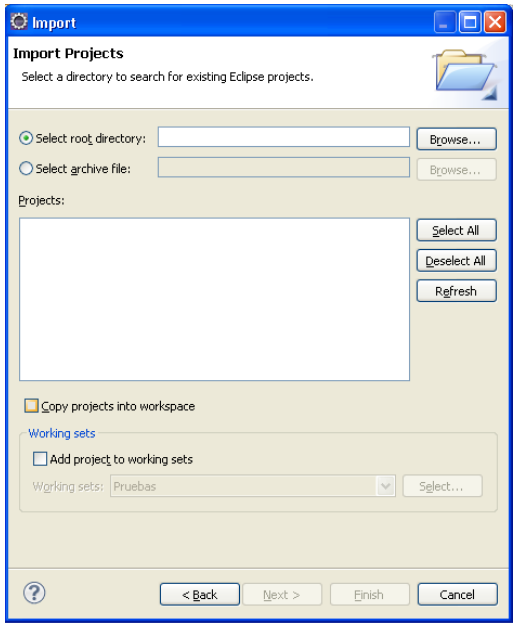
Figura 23. **Importar proyecto a *eclipse***



Fuente: propia.

b. Se busca el proyecto creado y se importa

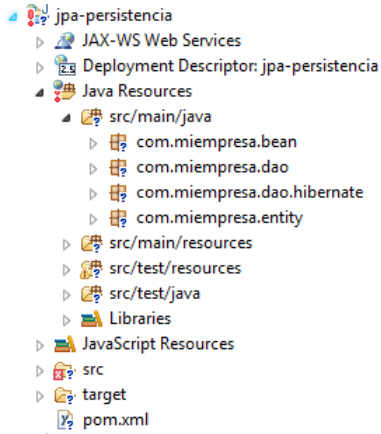
Figura 24. **Selección de archivo**



Fuente: propia.

c. Se hace un *Maven eclipse* para visualizar el proyecto en conjunto de paquetes y directorios.

Figura 25. **Aplicación en desarrollo**



Fuente: propia.

3.3.3. Creando el proyecto de persistencia

Proyecto que proporciona acceso a la base de datos utilizando *JPA* (Java *Persistence API*), que busca unificar la manera en que funcionan las utilidades que proveen un mapeo objeto-relacional utilizando *Hibernate* para persistir.

Se crea el proyecto con *Maven* y se importa hacia *Eclipse*. Se modifica el archivo pom.xml, se deben de poner las dependencias a continuación mostradas:

Figura 26. **Dependencias de proyecto persistencia**

```
<modelVersion>4.0.0</modelVersion>
<groupId>mi.empresa</groupId>
<artifactId>jpa-persistencia</artifactId>
<packaging>jar</packaging>
<version>1.0.0</version>
<name>jpa-persistencia</name>
```

Fuente: propia.

Figura 27. **Dependencias de *Spring***

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>2.5.6</version>
    <exclusions>
        <exclusion>
            Element : exclusion
            Content Model : all(artifactId?, groupId?)
            Press 'F2' for focus
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

Fuente: propia.

Figura 28. **Dependencia de *Hibernate***

```
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>ejb3-persistence</artifactId>
<version>1.0.2.GA</version>
<scope>provided</scope>
</dependency>
```

Fuente: propia.

Figura 29. **Dependencias de *AOP***

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.6.1</version>
    <scope>provided</scope>
</dependency>
```

Fuente: propia.

3.3.4. Creando el proyecto de negocio

El proyecto de negocio utiliza persistencia *JPA* con provider *Hibernate* y anotaciones para las entidades de negocio en la capa *DAO*, para ello es necesario utilizar dependencias de *Spring* e *Hibernate*. Se debe importar el proyecto al IDE *Eclipse*, luego de esto, se hace la modificación del archivo pom.xml, para incluir las dependencias mostradas.

Figura 30. **Dependencia de *jUnit* y *ojdbc***

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.4</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>ojdbc</groupId>
    <artifactId>ojdbc</artifactId>
    <version>14</version>
    <scope>test</scope>
</dependency>
```

Fuente: propia.

Figura 31. **Dependencia de *Spring***

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>2.5.6</version>
    <exclusions>
        <exclusion>
            <artifactId>aopalliance</artifactId>
            <groupId>aopalliance</groupId>
        </exclusion>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>2.5.6</version>
    <exclusions>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>2.5.6</version>
```

Fuente: propia.

Figura 32. **Dependencias de *Hibernate***

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
    <version>3.4.0.GA</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>3.4.0.GA</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>3.1.0.GA</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.5.2</version>
    <scope>provided</scope>
</dependency>
```

Fuente: propia.

3.3.5. Creando el proyecto de presentación

Utiliza el proyecto de negocio que creamos de acuerdo a nuestros requerimientos y los *framework*s *RichFaces* que nos permiten dibujar nuestras páginas en la fase de desarrollo, utilizando *beans* y dependencias de *Spring* para construir un proyecto de fácil mantenibilidad. Luego de importar el proyecto, creado previamente se debe de modificar el pom para incluir las dependencias que se utilizarán.

Y se colocan las dependencias descritas a continuación:

Se agregan las dependencias de *RichFaces*, excluyendo *beanutils*, *common-collecion* y *commons-logging*, se utilizan nativos de *Spring*.

Figura 33. **Dependencias de *RichFaces***

```
<dependency>
    <groupId>org.richfaces.framework</groupId>
    <artifactId>richfaces-api</artifactId>
    <version>3.3.2.SR1</version>
    <exclusions>
        <exclusion>
            <artifactId>commons-beanutils</artifactId>
            <groupId>commons-beanutils</groupId>
        </exclusion>
        <exclusion>
            <artifactId>commons-collections</artifactId>
            <groupId>commons-collections</groupId>
        </exclusion>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.richfaces.framework</groupId>
    <artifactId>richfaces-impl</artifactId>
    <version>3.3.2.SR1</version>
    <exclusions>
        <exclusion>
            <artifactId>commons-digester</artifactId>
            <groupId>commons-digester</groupId>
        </exclusion>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.richfaces.ui</groupId>
    <artifactId>richfaces-ui</artifactId>
    <version>3.3.2.SR1</version>
</dependency>
```

Fuente: propia.

Figura 34. **Dependencias de *JSF***

```
<dependency>
    <groupId>javax.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>1.2_12</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>1.2_12</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.el</groupId>
    <artifactId>el-api</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>el-impl</groupId>
    <artifactId>el-impl</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
</dependency>
```

Fuente: propia.

Figura 35. **Dependencias de *Spring***

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>2.5.6</version>
    <exclusions>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
        <exclusion>
            <artifactId>aopalliance</artifactId>
            <groupId>aopalliance</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

Fuente: propia.

Figura 36. Dependencia del negocio

```
<dependency>
    <groupId>mi.empresa</groupId>
    <artifactId>jpa-negocio</artifactId>
    <version>1.0.0</version>
    <scope>compile</scope>
</dependency>
```

Fuente: propia.

Figura 37. **Configuración para la construcción de la aplicación**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.0.2</version>

            <configuration>
                <dependentWarExcludes>
                    META-INF/**,WEB-INF/*.txt
                </dependentWarExcludes>
                <archiveClasses>true</archiveClasses>
                <warSourceDirectory>
                    src/main/webapp
                </warSourceDirectory>
                <warSourceExcludes>
                    **/*.mex,WEB-INF/*.tld,WEB-INF/classes/**
                </warSourceExcludes>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Fuente: propia.

Note que varias de las dependencias tienen scope “provided”, esto significa que están en el servidor y en nuestro proyecto “war” no serán incluidas. Una vez modificados los archivos pom.xml de los proyectos es necesario obtener las dependencias por medio de un mvn *eclipse*, con el cual se descargan del repositorio de *maven*.

3.4. Conectando las capas con *Spring*

Para la configuración de *Spring* dentro de esta aplicación es necesario agregar algunas líneas en el archivo *web*.xml, que se encuentra en la capa de presentación carpeta *WEB*-INF, las configuraciones que se deben de poner son las siguientes:

Se elige el lugar para guardar los estados de la página, por lo cual se setea en el servidor

Figura 38. **Método de preservar estados**

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
</context-param>
```

Fuente: propia.

Se escoge el *skin* clásico para *RichFaces*

Figura 39. **Configuración de vista de *RichFaces***

```
<context-param>
    <param-name>org.richfaces.CONTROL_SKINNING</param-name>
    <param-value>enable</param-value>
</context-param>
<context-param>
    <param-name>org.ajax4jsf.SKIN</param-name>
    <param-value>classic</param-value>
</context-param>
```

Fuente: propia.

Este parámetro define en qué lugar se encuentra la definición de los UI que manejara el *JSF*

Figura 40. **Configuración de archivo faces-config.xml**

```
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml</param-value>
</context-param>
```

Fuente: propia.

Spring ContextLoaderListener estos parámetros son para saber en qué lugar se encuentra el archivo applicationContext que contiene la definición de los *beans*

Figura 41. **Configuración de applicationContext.xml**

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath*:META-INF/applicationContext.xml</param-value>
</context-param>
<context-param>
    <param-name>locatorFactorySelector</param-name>
    <param-value>classpath*:META-INF/beanRefContext.xml</param-value>
</context-param>
<context-param>
    <param-name>parentContextKey</param-name>
    <param-value>mainApplicationContext</param-value>
</context-param>
```

Fuente: propia.

Se indica que se utilizará el listener para el *Application Context*

Figura 42. ***Listeners*** **para activar el** ***Application Context***

```
<listener>
    <listener-class>com.sun.faces.config.
    ConfigureListener</listener-class>
</listener>
<listener>
    <listener-class>org.springframework.web.context.
    request.RequestContextListener</listener-class>
</listener>
<listener>
    <listener-class>org.springframework.web.context.
    ContextLoaderListener</listener-class>
</listener>
```

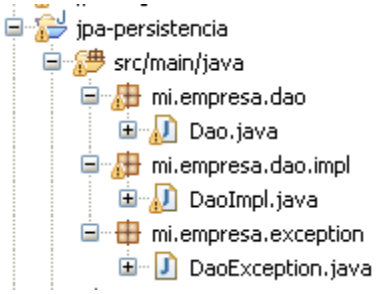
Fuente: propio.

Estas son las configuraciones que más resaltan en este archivo, tiempo de *session*, páginas de bienvenida y todos estos atributos pueden ser definidos a conveniencia.

3.5. Creando la persistencia

Dentro del proyecto de persistencia se debe de crear una interfaz y una clase en las cuales se implementara los accesos a datos estas clases deben de tener el nombre de: *DAO* y *DAOImpl*.

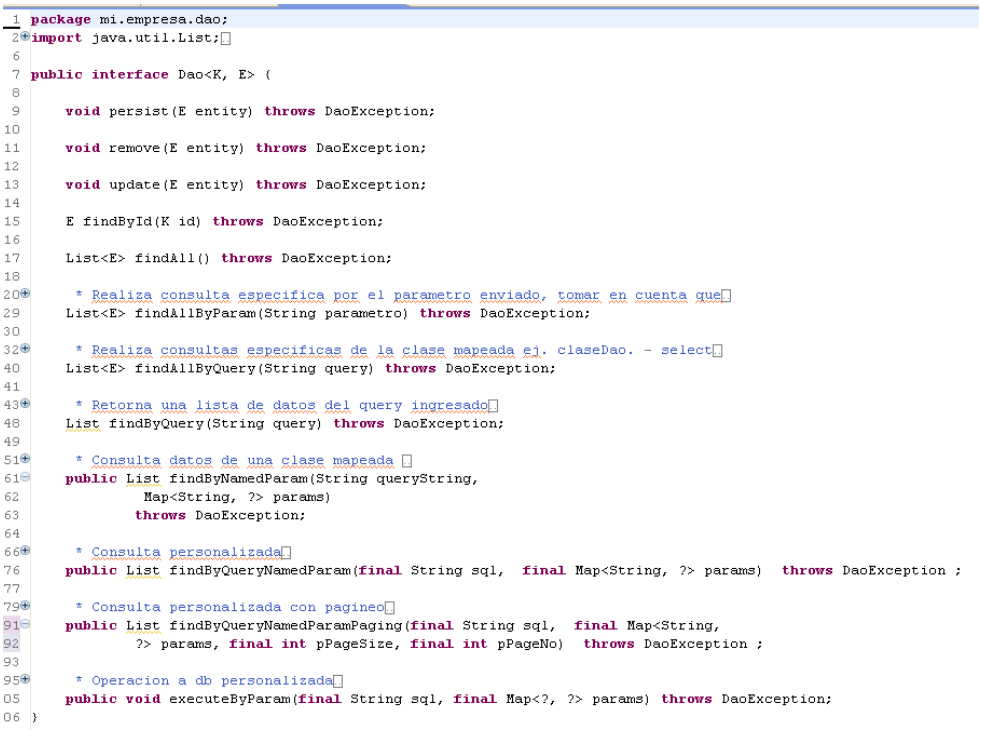
Figura 43. **Estructura de proyecto persistencia**



Fuente: propia.

Estas clases contienen los siguientes métodos para el *DAO* interfaz.

Figura 44. **Clases *DAO* servicios**



```
1  package mi.empresa.dao;
2  import java.util.List;
6
7  public interface Dao<K, E> {
8
9      void persist(E entity) throws DaoException;
10
11     void remove(E entity) throws DaoException;
12
13     void update(E entity) throws DaoException;
14
15     E findById(K id) throws DaoException;
16
17     List<E> findAll() throws DaoException;
18
20      * Realiza consulta especifica por el parametro enviado, tomar en cuenta que
29     List<E> findAllByParam(String parametro) throws DaoException;
30
32      * Realiza consultas especificas de la clase mapeada ej. claseDao. - select
40     List<E> findAllByQuery(String query) throws DaoException;
41
43      * Retorna una lista de datos del query ingresado
48     List findByQuery(String query) throws DaoException;
49
51      * Consulta datos de una clase mapeada
61     public List findByNamedParam(String queryString,
62             Map<String, ?> params)
63            throws DaoException;
64
66      * Consulta personalizada
76     public List findByQueryNamedParam(final String sql,  final Map<String, ?> params)  throws DaoException ;
77
79      * Consulta personalizada con pagineo
91     public List findByQueryNamedParamPaging(final String sql,  final Map<String,
92             ?> params, final int pPageSize, final int pPageNo)  throws DaoException ;
93
95      * Operacion a db personalizada
05     public void executeByParam(final String sql, final Map<?, ?> params) throws DaoException;
06 }
```

Fuente: propia.

En la implementación de esta clase se tienen los siguientes métodos:

- ✓ Persist
- ✓ Remove
- ✓ Update

Para consultas se utilizarán los siguientes métodos:

Figura 45. **Métodos para consulta**

```
@SuppressWarnings("unchecked")
public List<E> findAll() throws DaoException {
    return getJpaTemplate().find("from " + entityClass.getName());
}

@SuppressWarnings("unchecked")
public List<E> findAllByParam(String parametro) throws DaoException {
    if (parametro.equals("") || parametro == null)
        return getJpaTemplate().find("from " + entityClass.getName());
    else
        return getJpaTemplate().find(
                "select e from " + entityClass.getName() + " e where e."
                        + parametro);
}

@SuppressWarnings("unchecked")
public List<E> findAllByQuery(String query) throws DaoException {
    return getJpaTemplate().find(query);
}

@Deprecated
public List findByQuery(final String queryString) throws DaoException {
    return getJpaTemplate().executeFind(new JpaCallback() {
        public Object doInJpa(EntityManager em) throws PersistenceException {
            Query queryObject = em.createNativeQuery(queryString);
            return queryObject.getResultList();
        }
    });
}
```

Fuente: propia.

Estos métodos brindan consultas por *named queries*, consultas *SQL* puro y consultas por entidad.

Ahora se debe configurar el archivo ApplicationContext.xml, como primer punto se indica que el archivo el *DataSource* a utilizar será el proporcionado por el negocio:

Figura 46. **Configuración de *DataSource***

```
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" destroy-method="finalize">
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
            <property name="showSql" value="true" />
            <property name="generateDdl" value="false" />
            <property name="databasePlatform" value="org.hibernate.dialect.Oracle10gDialect" />
        </bean>
    </property>
</bean>

<bean id="jndiTemplate" class="org.springframework.jndi.JndiTemplate" destroy-method="finalize">
    <property name="environment">
        <props>
            <prop key="java.naming.factory.initial">org.apache.naming.java.javaURLContextFactory</prop>
            <prop key="java.naming.provider.url">org.apache.naming</prop>
            <prop key="java.naming.factory.url.pkgs">org.apache.naming</prop>
        </props>
    </property>
</bean>
```

Fuente: propia.

Ahora se hace la configuración del *transaction manager*, así como la definición del punto de corte de *AOP* y el control del servicio:

Figura 47. **Configuración de *transaction manager***

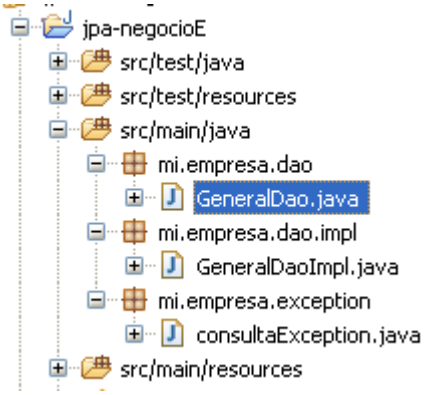
```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"
    destroy-method="finalize">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
    <property name="dataSource" ref="dataSource" />
</bean>

<aop:config>
    <aop:pointcut id="defaultAOP" expression="execution(* mi.empresa.dao.*.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="defaultAOP" />
</aop:config>

<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="*" propagation="REQUIRED" />
    </tx:attributes>
</tx:advice>
```

Fuente: propia.

3.6. Agregando lógica a la aplicación

Para la parte de negocio se debe definir como buena práctica una interfaz y además la implementación de la misma, esto para aprovechar el encapsulamiento y la publicación de servicios, de esta manera se crea un objeto tipo interface con el nombre que deseamos para ejemplificar se utilizará un objeto de nombre GeneralDao la cual extiende del *DAO* del proyecto de *jpa*-persistencia y su implementación GeneralDaoImpl.

Figura 48. **Estructura capa de negocio**



Fuente: propia.

Este servicio se debe de colocar en archivo ApplicationContext.xml de esta capa de la siguiente manera:

Figura 49. **Definición de *bean***

```
<bean id="generalDao" class="mi.empresa.dao.impl.GeneralDaoImpl">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Fuente: propia.

Donde el *id* es nombre que este ocupara en el contenedor de *IoC* y *class* es el objeto al cual está apuntando. Con estos pasos se tiene una clase que

brinda el servicio a través del *Application Context*, la cual extiende del proyecto de *jpa*-persistencia. Aquí podemos agregar los métodos que se deseen para ejecutar reglas de negocio, agregar más servicios, hacer utilización de un servicio en otro siempre con el mapeo en el *Application Context*.

3.7. La interfaz

En esta parte de la aplicación se generan las páginas que mostrarán los datos al usuario final, con la cual interactúa y la forma de entrada de los datos, se deben de configurar el archivo faces-context.xml, en el cual se hace la conexión lógica una página *JSP* y un *bean* de respaldo.

¿Cómo funciona la lógica? Se tiene una página *JSP* con la implementación de algún *JSF* en el caso de la arquitectura se utilizará *RichFaces*, se define un *bean* de respaldo el cual posee las propiedades que se conectarán con la página, y los métodos que se invocarán para generar todo tipo de transacciones, validaciones, Etc.

Se crea el *bean* de respaldo con nombre ConsultaEjemploUI y la página *JSP* estos se mapean en el archivo faces-context.xml de la siguiente manera:

Figura 50. **Definición de *managed bean***

```
<managed-bean>
        <managed-bean-name>bkn_ConsultaEjemplo</managed-bean-name>
        <managed-bean-class>mi.empresa.web.ConsultaUI</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
                <property-name>generalDao</property-name>
                <value>#{generalDao}</value>
        </managed-property>
</managed-bean>
```

Fuente: propia.

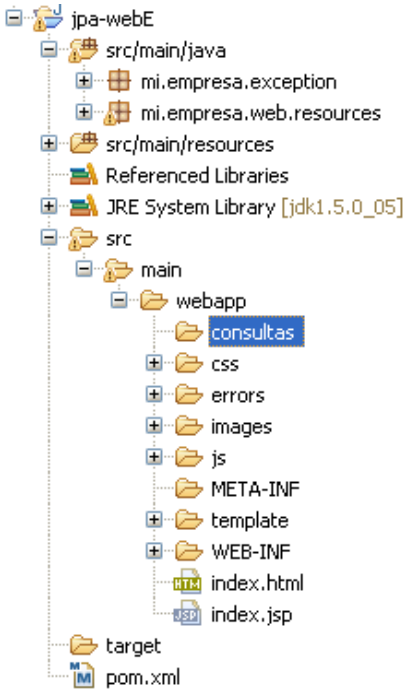
Aquí se define el nombre del *bean* como se hará el llamado en la *JSP* para referenciar a este objeto, se define que *bean* es el que formara el respaldo de la página, que tipo de ámbito tiene puede ser *request* o *session* y además se configura las propiedades que se utilizan del negocio o modelo en el *bean*, para el caso de ejemplo se hace referencia al servicio generalDao.

Los objetos se colocan en distintas partes del proyecto:

- ✓ Para las páginas *Web* se deben colocar en src/main/webapp/*
- ✓ Los *bean* de respaldo se colocan en src/java/main/*
- ✓ El archivo face-context.xml en src/main/webapp/*WEB*-INF

Esta es la estructura del proyecto de presentación.

Figura 51. **Estructura de capa de presentación**



Fuente: propia.

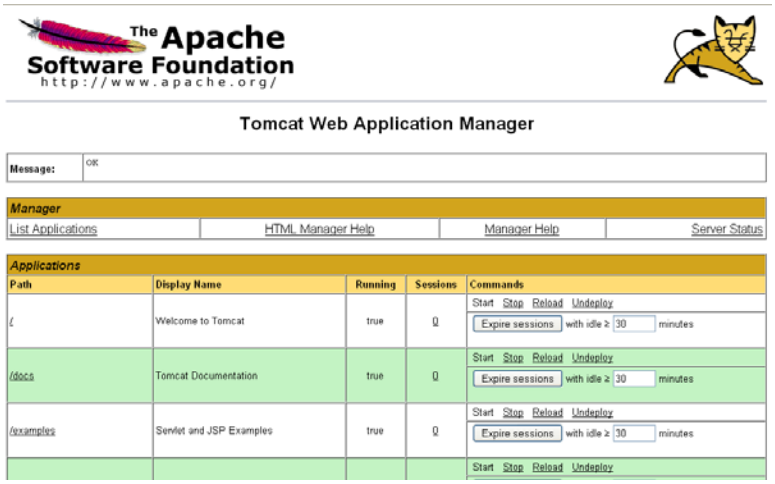
3.8. Implantación de la aplicación

Otra característica de esta arquitectura es su fácil implantación, esto debido a que el contenedor que soporta la aplicación es muy ágil y liviano, permite un crecimiento, utiliza pocos recursos y es de código abierto, Tomcat 6 es la versión que se destina para el ejemplo.

Para poder realizar el despliegue de la aplicación se debe de formar un paquete tipo War, el cual es leído y descomprimido por el servidor de aplicaciones para posteriormente realizar el despliegue del mismo en el servidor, levantar los servicios necesarios, crear el contexto de aplicación y brindar la misma a utilización del usuario final.

La aplicación se debe desplegar en el servidor de Tomcat, como se muestra a continuación:

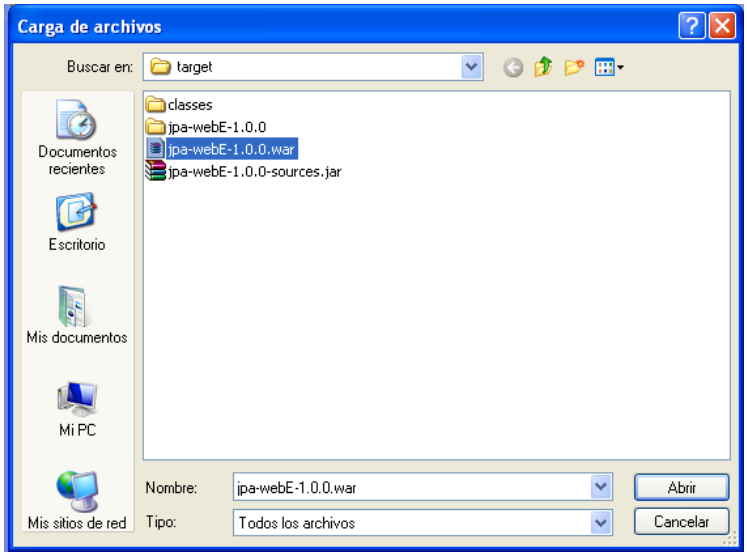
Figura 52. **Servidor de aplicación**



Fuente: propia.

Se debe seleccionar el archivo War que se debe de desplegar.

Figura 53. **Selección de archivo a desplegar**



Fuente: propia.

4. IMPLANTACIÓN DE LA ARQUITECTURA

4.1. Definición de aplicación

La empresa Impresos Sergrafic es una imprenta litografía la cual se dedica a la impresión de papelería comercial y publicitaria, por lo cual poseen una bodega en la que almacena todas sus materias primas, entre las materias primas que poseen se encuentra tintas, espátulas, papel, papel de empaque.

Por lo cual se hacen de la necesidad de mantener un control de los insumos que poseen y de los gastos de los mismos, por lo cual han solicitado una aplicación o herramienta que les permita llevar un control de estos.

Se define que la aplicación contará con un solo módulo, en el cual se tendrá un catálogo de productos, así como reportes de los mismos, la aplicación se debe manejar los productos, así como tipos y existencias de los mismos.

Además se ha solicita la utilización de roles dentro de la aplicación ya que no todos los usuarios podrán cargar o descargar productos, se definen los siguientes roles:

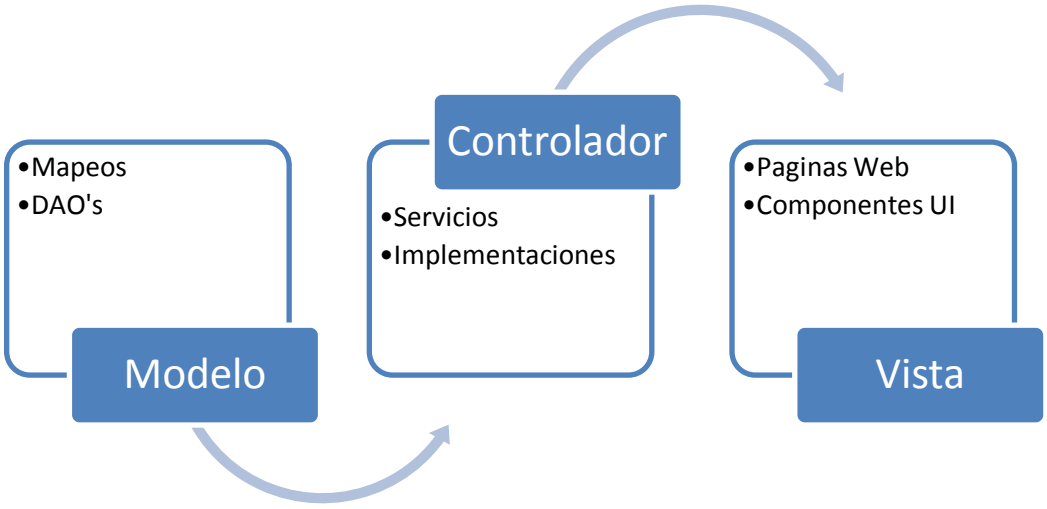
- ✓ Usuario administrador
- ✓ Usuario de consulta
- ✓ Usuario supervisor

4.2. Elección de arquitectura

Esta aplicación por sencilla que parezca permite mostrar la fácil configuración, poco acoplamiento y adaptabilidad que cuenta la arquitectura, debido a que fácilmente, puede cambiarse de interfaz, cambio de base de datos, cambio en DBMS o algún nuevo requerimiento.

Se utilizará el modelo MVC como se presenta en la siguiente gráfica.

Figura 54. **Vista del modelo vista controlador**



Fuente: propia.

4.3. Creando la aplicación

4.3.1. Desarrollo de la aplicación

Se utilizó la librería propuesta como *jpa*-persistencia en el capítulo anterior, con la cual logra el acceso a datos unificado, se crea la capa de

negocio de la aplicación, la cual posee la lógica para la carga y descarga de productos del inventario. Para esto se definió los siguientes servicios:

- ✓ CargaSvc implementación CargaSvcImpl: estas clases poseen la lógica para verificar la existencia de los productos y agregar a la existencia necesaria.

- ✓ DescargaSvc implementación DescargaSvcImpl: esta tiene la lógica para realizar la disminución de la existencia en la base de datos, valida que existan los productos necesarios.

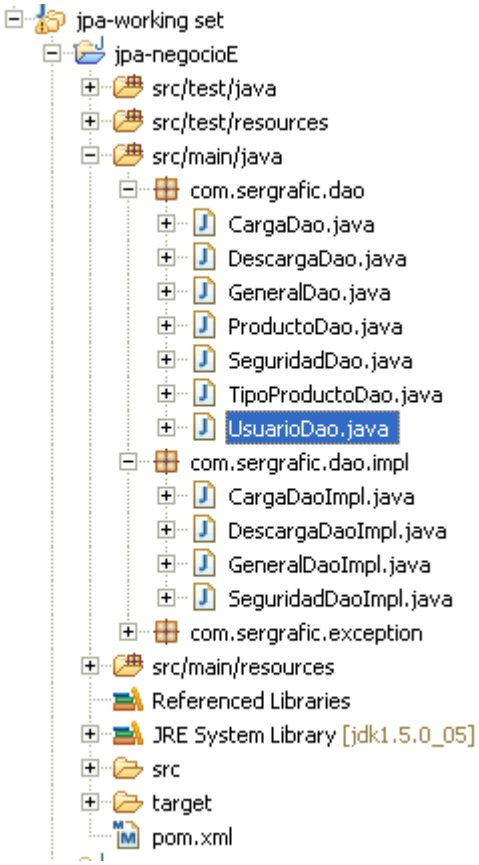
- ✓ SeguridadSvc implementación SeguridadSvcImpl: este servicio permite la administración de los usuarios, permisos y claves.

- ✓ ProductoDao: en esta clase se realiza el mapeo de la tabla producto, la cual se utiliza para las consultas y persistencia de los productos.

- ✓ TipoProductoDao: esta mapea la tabla tipo_producto.

- ✓ UsuarioDao: mapeo de la tabla usuario.

Figura 55. **Estructura de la aplicación de Sergrafic**



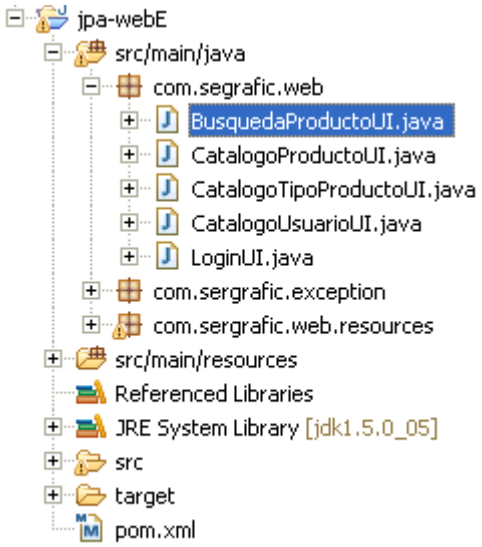
Fuente: propia.

Para la capa de presentación se generaron las siguientes páginas *JSF*, así como sus respectivos *beans* de respaldo.

- ✓ catalogoProducto
- ✓ catalogoTipoProducto
- ✓ catalogoUsario
- ✓ login
- ✓ busquedaProducto
- ✓ existenciaProducto
- ✓ catalogoProductoUI
- ✓ catalogoTipoProductoUI

- ✓ catalogoUsarioUI
- ✓ loginUI
- ✓ busquedaProductoUI
- ✓ existenciaProductoUI

Figura 56. **Estructura de proyecto *web* para Sergrafic**

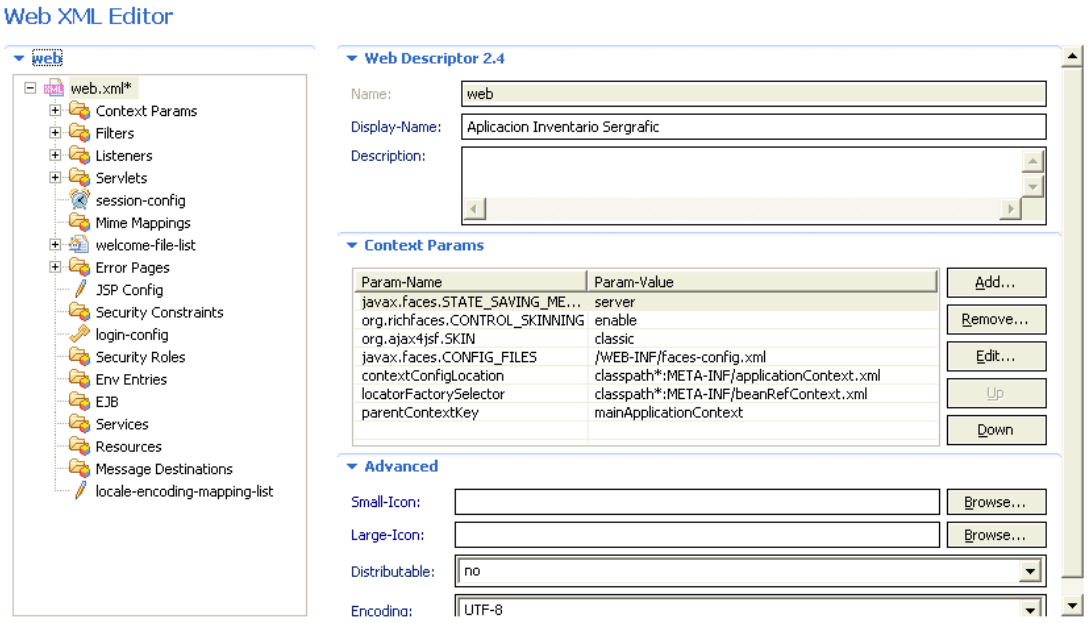


Fuente: propia.

4.3.2. Integración de capas

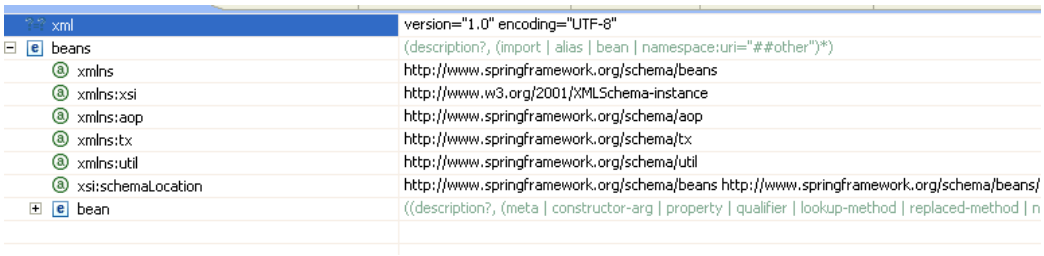
Se realizó la configuración de la aplicación *Web*, en el archivo *web*.xml se definió la utilización de *Application Context* para la inversión de control, los filtros de seguridad para no permitir las intrusiones al sistema. Además se realizó la configuración de los servicios en el applicationContext.xml, todos los servicios y *beans* se definieron de tipo *Singlenton.*

Figura 57. ***web.xml***



Fuente: propia.

Figura 58. **applicationContext.xml**



Fuente: propia.

Figura 59. **facesContext.xml**

```
                              "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>

    <application>
     <locale-config>
            <default-locale>es-gt</default-locale>
            <supported-locale>es-gt</supported-locale>
        </locale-config>
        <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variabl
    </application>

    <managed-bean>
        <managed-bean-name>bkn_generales</managed-bean-name>
        <managed-bean-class>gt.gob.sat.web.resources.DatosGenerales</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
```

Element : managed-bean-scope
The "managed-bean-scope" element represents the scope into which a newly created instance of the

Fuente: propia.

4.4. Revisión de cliente

4.4.1. Pantallas de aplicación

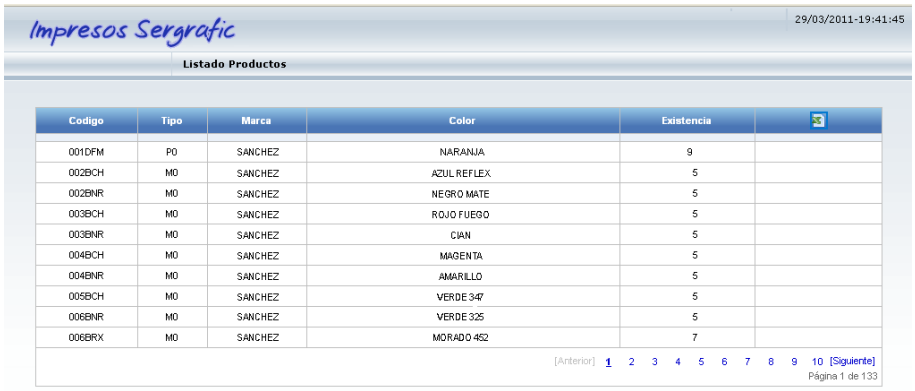
Pantalla de ingreso de productos al sistema.

Figura 60. **Pantalla de ingreso de productos**

Fuente: propia.

Pantalla de búsqueda de productos.

Figura 61. **Pantalla de consulta de inventario**



Fuente: propia.

4.4.2. Carta de aceptación de aplicación

Guatemala, 24 de Marzo 2,011

A QUIEN INTERESE:

Por este medio reciba un cordial saludo de parte de Sergrafic y esperamos que todas sus actividades se realicen con éxito.

Por este medio hago constar que **NELSON MORIS LARIN REYES**, quién se identifica con No. de Cédula E-5 43931 realizó un sistema para nuestro control de inventario interno, el cual consta de un sistema que permite tener Catálogo de Productos, Catálogo de Tipos de Producto, Búsqueda de Inventario y Control de Existencias.

Reintero mi más sincera satisfacción con el sistema realizado para Sergrafic y autorizó a que pueda ser utilizado únicamente para afines estudiantiles.

Sin otro particular me despido.

Atentamente

Marisela Friely
Contadora General

4ta. Avenida 17-14, Zona 12 Reformita Tels.: 2473 2245 - 2472 7953 - 2485 0538
Telefax: 2485 0382
e-mail: impsergrafic@gmail.com

CONCLUSIONES

1. Se concluyó que la arquitectura que se expone en este trabajo se enfoca en atributos de calidad, lo que permite decir que posee poco acoplamiento entre capas, genera una separación de éstas, que permite mayor mantenibilidad y fácil configuración para su interacción, generar cambios dentro de la aplicación, nuevos requerimientos, se puede realizar sin mayor trabajo en los componentes ya existentes, la integración con nuevos componentes se logra de forma ágil.

2. Una arquitectura es un conjunto de reglas y normas que ayudan a construir un *software* con un desempeño esperado, con el presente trabajo se llega a la conclusión que la guía permite la generación de aplicaciones *web* de gran calidad, buscando siempre una alta disponibilidad, poco acoplamiento, usabilidad y cumplimiento de requerimientos. Esta guía brinda una luz para el desarrollo y la implementación de inversión de control, inyección de dependencias y *AOP* en proyectos *web*.

3. Las necesidades y restricciones que posee un sistema, indican los parámetros que facilitan la búsqueda de arquitecturas candidatas. La elección de una arquitectura debe satisfacer todos los aspectos que la aplicación requiera para permitirle lograr los resultados esperados.

RECOMENDACIONES

1. Utilizar como parte del desarrollo de aplicaciones *web* el *Spring framework* para fomentar los atributos de calidad que promueve, así como las buenas prácticas de programación.

2. Seguir una metodología y generar un plan de trabajo para el desarrollo, ya que brinda un mejor control de la aplicación, se puede tener una aplicación con mejores prestaciones, se recomienda la utilización de *Scrum* para el desarrollo.

3. Evaluar las alternativas que encajan dentro de las restricciones y requerimientos de un sistema, así se puede optar por soluciones adecuadas, con resultados acordes a las necesidades.

4. Utilizar la arquitectura propuesta en este estudio, ya que provee de poco acoplamiento y gran mantenibilidad, creando sistemas adaptables, de fácil crecimiento.

BIBLIOGRAFÍA

1. FERNÁNDEZ, Daniel. *Definición de una arquitectura, software para el diseño de aplicaciones web basadas en tecnología java J2EE* [en línea]. Universidad de Oviedo, España. [ref. Diciembre 2010]. Disponible en web: < http://www.di.uniovi.es/~dflanvin/doctorado/ArquitecturaJ2EE.*PDF*>

2. FOWLER, Martin. *Contenedores de Inversión de control y el patrón de Inyección de Dependencias* [en línea]. Chicago, IL. [ref. Enero 2011]. Disponible en Web: <http://www.programacion.com/articulo/contenedores_de_inversion_de_control_y_el_patron_de_inyeccion_de_dependencias_304#1_componentes>

3. MEDIN, Juan. *Hacia una arquitectura con JSF, Spring, Hibernate y otros frameworks* [en línea]. Tenimap, Sevilla, España. 2008. [ref. Enero 2011] Disponible en web: <http://administracionelectronica.gob.es/archivos/pae_000002295.pdf>

4. O'REGAN, Graham. *Introduction to Aspect-Oriented Programming* [en línea]. Londres, Inglaterra. [ref. Enero 2011]. Disponible en Web: < http://tim.oreilly.com/pub/a/onjava/2004/01/14/aop.html?page=2>

5. RAMNIVAS, Laddad. *I Want AOP* [en línea]. USA, JavaWorld.com, [ref. Febrero 2011]. Disponible en Web: <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html?page=2>