J2EE TIENDAVIRTUAL APPLICATION FRAMEWORK

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per

Eduardo Varga Laguna

i dirigit per

Joan Serra Sagristà

Bellaterra,.....de.....….....de 200...

Universitat Autònoma de Barcelona

etse
Escola Tècnica Superior d'Enginyeria

El sotasignat, ...

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

CERTIFICA:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en

I per tal que consti firma la present.

Signat: .....

Bellaterra, ...de.....de 200.....

INDICE

1. Introducción

1.1. Motivaciones

Las motivaciones al presentar una propuesta basada en la tecnología *J2EE* no era otra que la 'comodidad' y 'agilidad' que me aportaba el hecho de trabajar diariamente con esta tecnología en mi ámbito laboral. Al plantearme que tipo de proyecto realizar, primó el poder compaginar estas dos actividades y no perder excesivo tiempo en entender la tecnología y todos sus *'secretos'*, pudiendo así, centrarme en el desarrollo en si del proyecto de final de carrera.

Evidentemente, también hay un componente de ilusión, ya que si decidí en su día encaminar mi futuro en esta línea es porque me interesa mucho el desarrollo de aplicativos para clientes utilizando tecnologías 'modernas', y de plena actualidad como pueden ser este tipo de entornos.

1.2. Objetivos

El objetivo principal de este proyecto ha sido enseñar mediante un ejemplo de aplicativo sencillo, como desarrollar una aplicación con la plataforma *J2EE* mediante el Framework *Struts.*

1.3. Organización de la memoria

La organización de esta memoria sigue un orden lógico dentro de lo que es la explicación normal de un proyecto de Ingeniería Informática.

Primero se han expuesto los ***Fundamentos teóricos*** a partir de los cuales el lector de la memoria puede tener una referencia clara posteriormente de el porque de las cosas que se irá encontrando en el desarrollo del mismo.

Después se han descrito las ***Fases*** en las que se ha encontrado el proyecto desde su inicio hasta su fin, pasando por las primeras fases de captación de requerimientos, análisis posterior del problema, exposición de las posibles soluciones, diseño de los componentes que formarían parte, implementación de los mismos con sus pertinentes revisiones y retoques y finalmente, las pruebas correspondientes a cada parte de la aplicación.

En el siguiente capitulo se podrá ver que decisiones se tomaron una vez tenia todos los requerimientos claros en cuanto al ***Diseño*** del proyecto. Se explicará todos los patrones de diseño que se suelen utilizar como solución en este tipo de entornos.

Posteriormente ya se entra en la explicación de como el Diseño explicado en el capitulo anterior se llevo a cabo, es decir, a la ***Implementación*** propiamente dicha.

Finalmente, se incluye una serie de pruebas que se realizaron para comprobar el correcto funcionamiento del aplicativo y que cumpliese con los requerimientos marcados en el inicio, estas pruebas incluyen tanto las pruebas realizadas al fin, como las pruebas realizadas durante la implementación, es en este tipo de pruebas cuando se comprueba que realmente se puede pasar a la siguiente iteración de tu proyecto.

2. Fundamentos teóricos

Struts es un *framework* de desarrollo de aplicaciones, basada en tres tecnologías principales:

- La arquitectura MVC (Modelo-View-Controller)
- El estándar J2EE
- El framework de código abierto Yakarta Struts.

A continuación se hace una introducción a estas tres tecnologías:

2.1. MVC

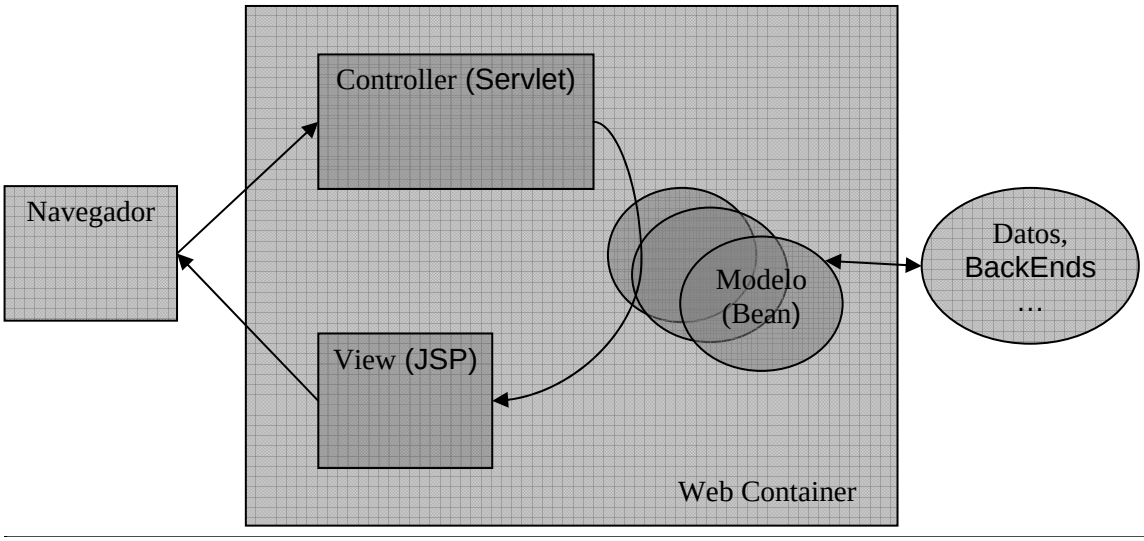
La arquitectura de Struts está basada en una arquitectura conocida como Model - View - Controller (MVC).

Cuando se quiere desarrollar una aplicación con Struts es preciso tener presente este hecho, y más claramente puede decirse que no tiene mucho sentido plantearse utilizar Struts sin estar dispuesto a hacer las separaciones entre las 3 capas.

MVC fue introducido el año 1988 para el desarrollo de interfaces de usuario en sistemas Smalltalk-80.

Las funciones de cada capa son las siguientes:

- ***Model***: contiene únicamente la lógica de negocio, y es completamente independiente de las otras 2 capas. Es el único lugar desde donde se accede a datos, back ends, etc.
- ***View***: contiene la lógica de presentación del modelo. Interactúa con el modelo para acceder a su estado. Recibe órdenes del *controller* para seleccionar la pantalla a mostrar.
- ***Controller***: contiene el flujo de la aplicación. Traslada las peticiones del usuario a llamadas al modelo para invocar las operaciones de negocio. Selecciona la siguiente pantalla a mostrar y lo transmite a la *view*.



Las acciones que el usuario realiza sobre el navegador se transmiten a un servlet que actúa de *controller*, intercepta todas las peticiones y las trata adecuadamente.

El tratamiento de una petición implica llamar a los objetos del model. Una vez se ha acabado la petición, el *controller* instancia la *view* correspondiente al resultado de la petición, y finalmente ésta *view* se muestra en el navegador del usuario.

2.2. J2EE

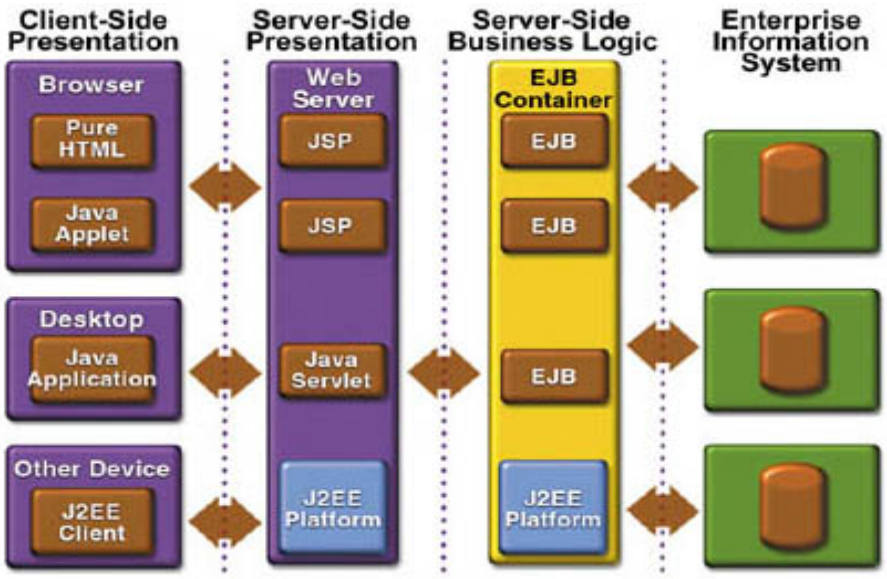
Java 2 Enterprise Edition (J2EE) es un estándar creado para facilitar el desarrollo de aplicaciones multi-nivel. Ante la complejidad y heterogeneidad de este tipo de sistemas, J2EE ofrece una arquitectura unificada y modular que facilita el desarrollo de aplicaciones distribuidas, proporcionándoles una serie de servicios que permiten acelerar el proceso de desarrollo dentro de las necesidades específicas de una empresa.

Las características más importantes de la arquitectura J2EE son la ***portabilidad*** (posibilitada por la definición de un estándar y de un proceso de certificación contra el estándar), la ***escalabilidad*** (gracias sobre todo a los servicios de distribución de componentes), la simplicidad (gracias a los servicios incorporados) y la capacidad de ***integración*** con sistemas propietarios.

El desarrollo en J2EE se basa principalmente en las especificaciones de Enterprise JavaBeans, Servlets y JSP, así como en la tecnología XML.

La plataforma J2EE proporciona implementaciones de los aspectos más complejos del desarrollo de aplicaciones de gran escala de modo que los desarrolladores puedan centrarse en los problemas específicos, generalmente la lógica de negocio y las interfaces de usuario.

El modelo de aplicaciones de J2EE encapsula las distintas capas de funcionalidad en distintos tipos de componentes. La lógica de negocio se encapsula en Enterprise JavaBeans (EJB), la interacción de usuario puede realizarse a través de páginas HTML, Applets u otras tecnologías, y controlarse desde el servidor mediante Servlets o JSP.



Modelo de aplicaciones J2EE

El estándar J2EE promueve la reutilización y la compatibilidad de productos dentro de la plataforma, por lo que normalmente una aplicación combinará el desarrollo de componentes específicos y presentaciones personalizadas con la integración de componentes o servicios de prefabricados. El modelo de aplicaciones J2EE divide las aplicaciones en tres partes fundamentales: ***componentes***, ***contenedores*** y ***conectores***.

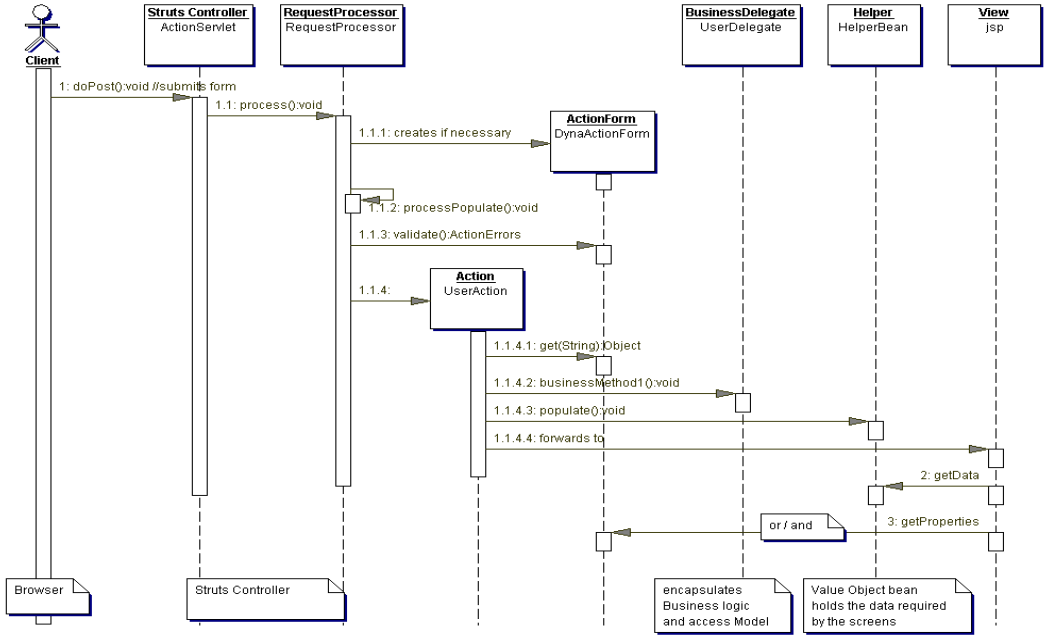
Los ***componentes*** contienen el desarrollo de la aplicación. Producidos con las tecnologías citadas más arriba, pueden ser reutilizados entre aplicaciones y fabricantes.

Los ***contenedores*** proporcionan el entorno de ejecución de los componentes, y los comunican con sus clientes, aportando varios servicios a ambos, permitiendo configurar muchos comportamientos en tiempo de despliegue, en vez de en tiempo de desarrollo. Los contenedores los desarrollan empresas de sistemas (típicamente servidores de aplicaciones: BEA, Oracle, IBM...).

Los ***conectores*** permiten la comunicación de aplicaciones J2EE con otros tipos de sistemas. Una API portable permite la integración con sistemas existentes dentro de la empresa o con tecnologías existentes más allá del estándar J2EE.

2.3. Struts

A continuación se explica el funcionamiento de Struts. En el siguiente diagrama de interacción, extraído de http://rollerjm.free.fr/pro/Struts11.html, puede verse el flujo de una interacción del usuario con una aplicación Struts.



Struts se centra principalmente en la capa de ***controller***. Todas las peticiones que hace el usuario desde su navegador hacia la aplicación pasan por un mismo servlet, el ***ActionServlet***, que delega la ejecución de la petición en un objeto de tipo ***RequestProcessor***, que es el que controla el flujo de la petición.

En primer lugar instancia un objeto ***ActionForm***. Los datos que conforman el input de la operación y que el usuario ha rellenado en un formulario HTML son recogidos por Struts y puestos dentro del ActionForm. Estos datos pueden tener diferentes niveles de visibilidad, y dependiendo de este

nivel Struts los almacena en un puesto u otro: ***request*** (HttpServletRequest), *session* (HttpSession), o ***application*** (ServletContext).

Según la URL invocada (***ActionMapping***), se instancia una subclase de ***Action*** entre los configurados en el fichero ***struts-config.xml***. El código del ***Action*** es el que hace los llamamientos al modelo, bien directamente, bien a través del patrón de diseño ***BusinessDelegate***, que viene a ser como una interfaz restringida del modelo en las necesidades particulares de la aplicación.

Los datos de vuelta de la operación llenan un bean que puede ser el mismo ***ActionForm*** si tiene scope de sesión, o bien, un bean específico por los datos de respuesta de la petición.

El resultado final de una petición es un objeto ***ActionForward*** que el ***Action*** ha escogido según ha ido la ejecución de la lógica de negocio ("ok", "error",...). En el fichero de configuración, se establece la relación (mapping) entre cada final de operación y el recurso que debe recibir control en cada caso. Este recurso puede ser otro ***Action***, una JSP, etc. El ***controller*** le pasa la petición al objeto correspondiente, haciendo un forward o un redirect según se haya definido en el mapping.

Ésta es la descripción más general de como es el flujo de una operación con Struts.

2.3.1. Estructura de una aplicación basada en Struts

2.3.1.1. Introducción

La ejecución de una petición a una aplicación Struts no difiere demasiado de la presentada anteriormente.

Para introducir de una manera rápida los elementos principales de una aplicación Struts, mostraremos la secuencia de la ejecución de una operación típica:

- Una JSP que contiene un formulario con unos campos de entrada se envía a la aplicación. Los tags de las taglibs de Struts con que se han creado la JSP hacen que los datos de los campos de entrada llenen los campos de un objeto del tipo ***ActionForm*** que se ha definido específicamente para esta operación.
- La aplicación tiene especificado en su deployment descriptor que todas las peticiones las trate siempre el mismo servlet: el ***ActionServlet*** proporcionado por Struts. El ***ActionServlet***, vía su ***RequestProcessor***, obtiene un objeto ***ActionMapping*** que contiene la información (almacenada en el fichero de configuración) de qué clase de ***Action*** se tiene que instanciar para tratar la petición. El ***RequestProcessor*** instancia al ***Action*** concreto y le invoca su método execute ().

- El ***Action*** es propiamente la implementación de la operación. Dentro de su método execute() tiene a su alcance toda la información que necesita para procesar la operación: el objeto ***ActionForm*** que se ha creado en el paso 1, el servlet context de donde puede obtener los datos de sesión, y los objetos ***HttpServletRequest*** y ***HttpServletResponse*** por si los necesita. Para procesar la operación necesitará llamar a algún método de las clases del modelo de negocio.
- Las clases del modelo hacen su trabajo apoyándose en los servicios que la infraestructura les proporciona y retornan su resultado, o lanzan una excepción. Es importante notar que estas clases no son en absoluto conscientes de si las están llamando desde una aplicación Struts o desde cualquier otro tipo de aplicación.
- Según cuál sea el objeto devuelto por la lógica de negocio, o la excepción que se haya lanzado, el ***Action*** creará beans con datos de respuesta y los pondrá como atributos de la ***HttpRequest***, o bien creará unos ***ActionErrors***. Finalmente el Action escoge uno de los ***ActionForwards*** definidos en el fichero de configuración ***struts-config.xml*** y lo pone como valor de retorno del método execute().
- El ***ActionServlet*** busca a qué corresponde el forward que le ha pasado el ***Action*** y le pasa el control. En este ejemplo el forward corresponde a una JSP, pero también podría ser otro ***Action***. La JSP de respuesta ya tendrá los tags correspondientes para extraer la información de los ***ActionErrors*** o de los request attributes que el ***Action*** ha rellenado.

2.3.1.2. Componentes de la view

2.3.1.2.1 JSPs

Struts no requiere en concreto ninguna tecnología concreta de implementación de la ***view***, JavaServer Pages es la más extendida y es la que se usa para ilustrar esta parte de la estructura de una aplicación.

Es importante notar que dentro del estándar JSP hay algunas prácticas que pueden impactar negativamente en algunos aspectos importantes como pueden ser la facilidad de mantenimiento o la reusabilidad.

Si se tienen en cuenta las recomendaciones, las JSP serán muy compactos, fáciles de entender y de mantener, y estarán formadas únicamente por tags HTML y de las diferentes taglibs escogidas.

A continuación se relacionan las taglibs más importantes proporcionadas por Struts:

- ***html***: Generan código HTML, sobretodo formularios. Su funcionalidad consiste en enlazar de manera muy dinámica los objetos que manejan los ***Actions*** (***ActionForms***, request attributes, etc) con los controles HTML del formulario.
- ***bean***: Acceso a beans que los ***Action*** han dejado en algún scope (request o session), y creación de beans a partir de parámetros, cabeceras o cookies, que después se pueden referenciar en otros lugares de la página.
- ***logic***: Generación condicional de HTML, o generación iterativa de HTML a partir de beans que sean colecciones java.util o arrays. Si un ***Action*** produce un bean que es una colección de beans, es posible generar una tabla con unos pocos tags.

Para obtener información de todos los taglibs existentes para el desarrollo de las JSP, se puede acceder al siguiente enlace:

http://struts.apache.org/struts-taglib/

2.3.1.2.2 Recursos

El concepto de la ***internacionalización*** (i18n) se basa en conseguir generar un aplicativo que soporte cualquier idioma mediante el uso de un fichero de *properties* en el que definiremos las *keys* y su respectivo literal en el idioma en concreto.

De este modo tendremos un fichero de recursos por cada idioma soportado por la aplicación.

No hay límite en el número de idiomas que una aplicación puede soportar. Struts adapta el soporte de multiidioma de Struts basado en el locale, mediante el cual detecta el idioma que hay en el navegador y utiliza el recurso (fichero de *properties*) adecuado.

2.3.1.2.3 ActionForms

Muchas de las JSPs de una aplicación contienen datos de entrada en un formulario HTML. En la arquitectura Struts, estos datos quedan a disposición de las ***Actions*** en forma de objetos del tipo ***ActionForm***.

Hay dos maneras de implementar un ***ActionForm***:

- Haciendo una subclase trivial de ***ActionForm*** con una variable por cada campo del formulario, con sus métodos accessors, y que implemente los métodos validate() y reset().
- Declarándola del tipo ***DynaActionForm*** y configurándola en el ***struts-config.xml***. Las ***DynaActionForms*** crean dinámicamente los campos que se le han configurado, con el tipo correspondiente, y evitan tener que hacer tantas subclases triviales de ***ActionForm***. Para hacer validaciones específicas sí que es preciso hacer subclases de ***DynaActionForm***, o bien utilizar el Struts Validator

Es dentro de las ***ActionForms*** donde se produce la validación de los datos. En las del primer tipo ya se ve que el método validate() es lo que hace este trabajo. El método devuelve un objeto ***ActionErrors*** con la lista de los errores que se han encontrado. Si esta lista no es vacía, Struts vuelve a redireccionar a la página de entrada del ***Action*** donde el usuario podrá ver los errores de validación mediante los tags <html:errores> correspondientes.

Uno de los problemas que puede tener esta manera de validar es que se acaba escribiendo mucho código de validación similar en muchos ***ActionForms***. Es por ello que existe un componente de Struts llamado ***Validator*** que permite definir las validaciones en un lugar centralizado (el mismo Validator ya proporciona los más habituales) y referenciarlas desde el fichero de configuración.

2.3.1.2.4 ActionErrors

Cualquier excepción que se produzca en el model durante la ejecución de la aplicación, o cualquier error del usuario, por ejemplo en forma de datos de entrada incorrectos, tienen que acabar convirtiéndose en un ***ActionError***.

Los ***ActionErrors*** se pueden crear a partir de una string con el mensaje de error, pero lo recomendable es hacerlo a partir de una clave en el fichero de recursos, lo cual permitirá que el error se presente en el idioma del usuario.

El tag ***<html:errors>*** es el utilizado para mostrar los ***ActionErrors*** a los usuarios. Éste tag puede estar en cualquier JSP de entrada de datos. Si el objeto ***ActionErrors*** está vacío no genera nada, pero si contiene algún ***ActionError*** entonces se mostrará. Además con el parámetro property se pueden seleccionar qué mensajes de la lista se quieren incluir en aquella posición concreta de la página. Así, si se quiere, se pueden mostrar mensajes de error de validación cerca de los campos de entrada donde se han producido.

2.3.1.3. Componentes del controller

2.3.1.3.1 ActionServlet

Es el componente que hace propiamente de ***controller*** servlet del JSP, y que por tanto centraliza todas las peticiones que se hacen en la aplicación. De trabajo hace bien poco ya que rápidamente delega en un objeto ***RequestProcessor***.

Normalmente una aplicación Struts no necesitará más que especificar el servlet en su deployment descriptor y asignarle las URLs que le interesen (típicamente, *.do).

2.3.1.3.2 RequestProcessor

RequestProcessor está diseñado para poder modificar el comportamiento. Consta de una serie de métodos processXxxxx (por ejemplo, processPreProcess, processActionForm, processValidate, processForwardConfig) que se invocan de manera ordenada y documentada, de manera que se puede modificar el comportamiento.

2.3.1.3.3 Ficheros de configuración

Todos los parámetros configurables de una aplicación Struts se especifican en un único fichero ***struts-config.xml***. Hay muchos aspectos de la aplicación que requieren sus entradas en el fichero. La lista que va a continuación expone los más importantes. Para una referencia completa referirse a la documentación de Struts o directamente a

http://yakarta.apache.org/struts/dtds/struts-config_1_1.dtd.

Action Mappings	Una entrada por cada ***Action*** definida a la aplicación, con su path, la referencia a una ***ActionForm***, si es que usa, y la clase que la implementa
Action Forwards	Por cada ***Action***, una entrada por cada uno de los posibles finales de la operación. Típicamente “success” o “failure”, pero pueden haber

	muchos más. Cada definición de forward tiene la URL a la que se trasladará la petición, y si se hará mediante un “forward” (lo más normal) o un “redirect”
Global Forwards	Si hay muchas operaciones que pueden acabar invocando el mismo forward se puede definir a nivel global. Buenos ejemplos son la página de inicio, o una página genérica de error.
Form Beans	Todas las ***ActionForms*** (o form beans) definidas para la aplicación. Se especifica de qué subclase de ***ActionForm*** son, o, en el caso de ser ***DynaForms***, cuáles properties tienen y de qué tipos son
Message Resources	Los ficheros de recursos definidos por la aplicación.
Plug-ins	Los Struts plug-ins instalados.
Controller	Se especifica de qué clase es el request processor.

Como se puede apreciar, es mucha información. Si la aplicación es muy grande eso podría representar un problema de mantenimiento. Por ello Struts tiene el concepto de módulos que permiten dividir la aplicación en una serie de partes, cada una de ellas con su fichero de configuración.

2.3.1.3.4 ActionMapping

Un objeto ***ActionMapping*** no es más que la definición de un ***Action*** que se ha encontrado en el fichero de configuración transformada en un objeto. Lo utiliza el propio *framework* para saber que ***Action*** concreto se ha de instanciar, y después la pasa también como parámetro al propio ***Action*** por sí se necesita (normalmente sólo hace falta para buscar el forward a retornar).

2.3.1.3.5 Action

El objeto ***Action*** tiene su método principal, execute(), que es donde se escribe la lógica de la operación. Cada operación se implementa habitualmente en una subclase de ***Action*** diferente. Struts pasa 4 parámetros a execute():

- La petición, en forma de objeto ***HttpServletRequest***
- La respuesta, en forma de objeto ***HttpServletResponse***
- El ***ActionMapping*** que ha originado la petición
- El ***ActionForm*** que contiene los datos de entrada

El proceso de la ejecución de un ***Action*** es, de manera simplificada:

- Obtener los datos de entrada (en este punto ya han sido validadas)
- Instanciar objetos del modelo y de servicios, invocar los métodos que sean necesarios para llevar a cabo la operación.
- Poner los datos de salida en el puesto que haga falta (como request attribute, o en el mismo ***ActionForm*** si ésta tiene un scope de sesión).
- Retornar un objeto ***ActionForward*** según haya sido el resultado de la operación

Si los objetos del modelo lanzan alguna excepción, ésta se ha de capturar, y retornar un forward adecuado, que probablemente será uno que lleve a una JSP de error, o en la misma JSP de entrada de datos, donde se mostrará un mensaje entendedor para el usuario.

Es muy normal que la mayoría de operaciones tengan algún proceso común en todas las operaciones. Por ejemplo, el tratamiento de excepciones genéricas, o escribir en el log la entrada y la salida de la acción, etc. Por ello se puede crear una action básica que derive de ***Action***, y hacer que todas las acciones de la aplicación deriven de nuestra acción base.

2.3.1.3.6 ActionForward

Un ***Action*** siempre tiene que acabar devolviendo un objeto ***ActionForward***. No lo ha de crear explícitamente, sino que estará configurado en el ***struts-config.xml***, y por tanto disponible en el objeto ***ActionMapping*** que se le pasa con execute().

2.3.1.4. Componentes del modelo

Struts no proporciona clases que se puedan considerar pertenecientes al modelo. Cada aplicación debe empezar desde cero, a partir de los requerimientos, a construir un modelo conceptual primero, y después un modelo de diseño. A menudo, los dos modelos no son demasiado diferentes, sobre todo si la aplicación no es demasiado grande. De manera simplista puede decirse que el modelo de diseño es más detallado que el conceptual y ya contiene algún elemento más tecnológico, como los servicios de invocar o patrones de diseño. El modelo conceptual es más un modelo de análisis.

2.3.1.4.1 Modelo conceptual

La interfaz del modelo debe ser la más sencilla posible que cumpla todos los requisitos. La simplicidad es básica para poder ser llamado fácilmente desde los ***Actions***, y para poder codificar un juego de pruebas de aceptación que sirvan para determinar si el modelo funciona tal y como se especifica en los requisitos.

La interfaz del modelo también debe ser bastante intuitiva. Los objetos presentes en el modelo tienen que representar objetos en el dominio de negocio, o sea, aquellos nombres de los que se habla en los requerimientos (objetos de negocio o business objects).

2.3.1.4.2 Modelo de diseño

Para hacer un buen modelo de diseño es muy útil conocer las experiencias de gente que ha intentado hacer cosas parecidas anteriormente. Un lugar donde están recogidas estas experiencias de manera sistematizada es en los llamados ***patrones de diseño*** o ***design patterns***.

Los primeros patrones de diseño fueron publicados el año 1994 en el libro “Design Patterns” de Gamma, Helm, Johnson y Vlissides. Allí se define un patrón de diseño como “una descripción de comunicaciones entre objetos que resuelven un problema genérico de diseño”.

Todos los patrones de diseño son aplicables a cualquier tipo de aplicación. En el mundo de las aplicaciones web también han aparecido muchos problemas que son más específicos del entorno J2EE. Tal vez el mejor lugar donde se recogen patrones de diseño específicos de J2EE es a los Java Blueprints de Sun. A continuación se presentan algunos patrones de diseño muy útiles de Java Blueprints.

Front Controller	Centralización de todas las peticiones de una aplicación en un único objeto que las distribuye y procesa de manera consistente. Justamente: Struts ***ActionServlet***.
Data Access Object (DAO)	Desacopla el acceso a los datos de la interfaz externa de un objeto persistente. Todo el acceso a datos está incluido en la implementación del DAO, pero su interfaz externa es la de un objeto normal, independiente de base de datos.
Transfer Object (o Value Object)	Los valores de retorno de los métodos de los business objects a menudo pueden constar de datos que se han extraído en diferentes llamadas internas, por ejemplo haciendo joins de diferentes tablas y después añadiendo datos obtenidos invocando una transacción. Un transfer object reúne todos estos datos y los devuelve de una sola vez al cliente. Es uno de los mejores artilugios para hacer más sencilla una interfaz de modelo.
Value List Handler	Estrategia para implementar listas virtuales. Es el caso en que el cliente hace una petición que tiene que retornar una lista de n elementos donde n puede llegar a ser muy grande. Junto con Transfer Object y Data Access Object es posible iterar sobre una lista larga sólo unos pocos elementos a la vez sin haberlos de pedir todos de entrada.

2.3.1.4.3 Modelo de datos

``*Un modelo de datos es un sistema formal y abstracto que permite describir los datos de acuerdo con reglas y convenios predefinidos. Es formal pues los objetos del sistema se manipulan*

siguiendo reglas perfectamente definidas y utilizando exclusivamente los operadores definidos en el sistema, independientemente de lo que estos objetos y operadores puedan significar."

Según Codd :

``*Un modelo de datos es una combinación de tres componentes:*

- Una colección de estructuras de datos (los bloques constructores de cualquier base de datos que conforman el modelo);
- Una colección de operadores o reglas de inferencia, los cuales pueden ser aplicados a cualquier instancia de los tipos de datos listados en *(1)*, para consultar o derivar datos de cualquier parte de estas estructuras en cualquier combinación deseada;
- Una colección de reglas generales de integridad, las cuales explícita o implícitamente definen un conjunto de estados consistentes --estas reglas algunas veces son expresadas como reglas de insertar-actualizar-borrar.''

Un modelo de datos puede ser usado de las siguientes maneras:

- Como una herramienta para especificar los tipos de datos y la organización de los mismos que son permisibles en una base de datos específica;
- Como una base para el desarrollo de una metodología general de diseño para las bases de datos;
- Como una base para el desarrollo de familias de lenguajes de alto nivel para manipulación de consultas (*querys*) y datos;
- Como el elemento clave en el diseño de la arquitectura de un manejador de bases de datos.

El primer modelo de datos desarrollado con toda la formalidad que esto implica fue el ***modelo relacional***, en 1969, mucho antes incluso que los modelos jerárquicos y de red. A pesar de que los sistemas jerárquicos y de red como software para manejar bases de datos son previos al modelo relacional, no fue sino hasta 1973 que los modelos de tales sistemas fueron definidos, apenas unos cuantos años antes de que estos sistemas empezaran a caer en desuso.

3. Fases del proyecto

3.1. Introducción

En el transcurso de un proyecto de Ingenieria del Software, sea cual sea la plataforma o entorno de desarrollo, hay un conjunto de fases por las que pasa el mismo que son perfectamente definibles y secuenciales, ya que la consecución de una conlleva el inicio de la siguiente.

En el caso real, cuando te ves inmerso, esto no es así 100%, ya que normalmente por falta de tiempo por los periodos marcados por las fechas de entrega y por razones obvias de la imperfección del ser humano para hacer las cosas correctamente, estas fases se ven mezcladas entre si y eso conlleva a que en fechas posteriores a la entrega del proyecto haya un, a veces excesivamente largo y otras veces inacabable proceso de detección de errores y corrección de incidencias.

Las fases por las que pasa el proyecto serian las siguientes:

1) Captura de requerimientos.
2) Diseño de la solución para satisfacer los requerimientos previos.
3) Implementación del diseño pactado.
4) Pruebas del producto.
5) Rectificaciones

3.2. Captura de Requerimientos

3.2.1. Introducción

En el inicio de un ‘proyecto real’ es imprescindible una serie de reuniones previas con el cliente para establecer cuales van a ser los requerimientos del sistema.

Estos requerimientos no son estáticos, esto quiere decir que no son tomados y establecidos al principio del proyecto y se mantienen inamovibles durante la implementación del mismo, sino que van sufriendo alteraciones / modificaciones a partir de las nuevas ideas del cliente o bien en las presentaciones de la evolución del proyecto en las cuales el cliente tiene contacto directo y visual con la futura aplicación.

Observaciones : Por cuestiones obvias implícitas en este proyecto en las que el cliente y el consultor son la misma persona, obviaremos el cambio de requerimientos durante la implementación y plantearemos una captación de requerimientos inicial. En los siguientes puntos se intentará simular de la mejor manera posible el proceso.

3.2.2. Clasificación de los requerimientos.

El clasificar requerimientos es una forma de organizarlos, hay requerimientos que por sus características no pueden ser tratados iguales. Por ejemplo, los requerimientos de entrenamiento de personal no son tratados de la misma manera que los requerimientos de una conexión a Internet.

La siguiente es una recomendación de como pueden ser clasificados los requerimientos aunque cada proyecto de software pueda usar sus propias clasificaciones.

3.2.2.1. Requerimientos del "entorno"

3.2.2.1.1 Descripción

El entorno es todo lo que rodea al sistema. Aunque no podemos cambiar el entorno, existen cierto tipo de requerimientos que se clasifican en esta categoría por que:

El sistema usa el entorno y lo necesita como una fuente de los servicios necesarios para que funcione. Ejemplos del entorno podemos mencionar: sistemas operativos, sistema de archivos, bases de datos.

El sistema debe de ser robusto y tolerar los errores que puedan ocurrir en el entorno, tales como congestión en los dispositivos y errores de entrada de datos, por lo tanto el entorno se debe de considerar dentro de los requerimientos.

3.2.2.1.2 Recopilación de requerimientos del cliente

- El aplicativo ha de poder ser visitado desde cualquier Terminal portátil o PC de tipo estándar.

- S.O. -> Windows, a partir de la versión Windows 2000
- S.O. -> Linux, cualquiera de las distribuciones más comunes (Red Hat, Suse, etc...)

- Ha de soportar el acceso desde cualquiera de los dos navegadores de mayor uso entre los usuarios.
  - Internet Explorer.
  - Mozilla Firefox.
- El sistema de almacenamiento ha de ser robusto y garantizar la persistencia de los datos, asi como un volumen considerable de peticiones de clientes del servicio.

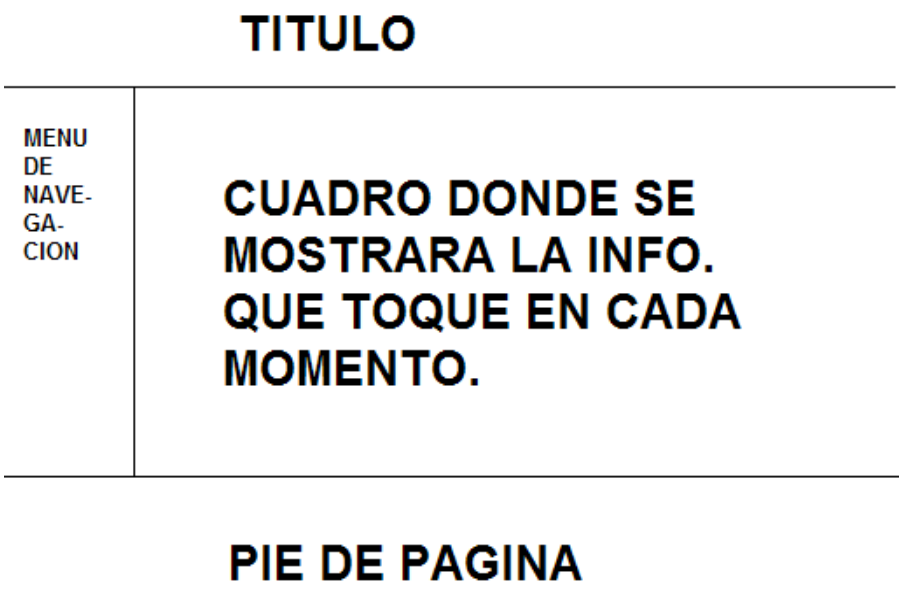
3.2.2.2. Requerimientos "ergonómicos"

3.2.2.2.1 Descripción

El mas conocido de los requerimientos ergonómicos es la interface con el usuario o GUI (Graphic User Interface). En otras palabras, los requerimientos ergonómicos son la forma en que el ser humano interactua con el sistema.

3.2.2.2.2 Recopilación de requerimientos del cliente

- El 'look & feel' del aplicativo ha de comportar un aprendizaje rápido e intuitivo. Seguir el concepto de formularios, tablas para la muestra de información por pantalla, botonera de fácil acceso a todos los puntos del aplicativo.
- El tema del diseño del mismo se dejará total libertad al diseñador, pudiendo el cliente rectificar colores y posicionamiento de objetos en posteriores presentaciones del estado del proyecto, cambios que no comporten una reestructuración del comportamiento del aplicativo, sólo cambios 'estéticos'.
- La GUI ha de seguir el siguiente esquema:



3.2.2.3. Requerimientos de interface

3.2.2.3.1 Descripción

La interface es como interactua el sistema con el ser humano o con otros sistemas (el enfoque es prácticamente el opuesto a los requerimientos ergonómicos), La interface es la especificación formal de los datos que el sistema recibe o manda al exterior. Usualmente se especifica el protocolo, el tipo de información, el medio para comunicarse y el formato de los datos que se van a comunicar.

3.2.2.3.2 Recopilación de requerimientos del cliente

- La navegación se hará mediante el clic de las opciones que saldrán continuamente en el lado izquiero de la interfaceo bien los botones que toque en cada momento.
- Toda la introducción de datos se realizará mediante formularios,cada parámetro dependerá de a que se refiere el input en cuestión.
    - Long para objetos como precios.

- Strings para nombres, apellidos, etc...

3.2.2.4. Requerimientos funcionales

3.2.2.4.1 Descripción

Estos son los que describen lo que el sistema debe de hacer. Es importante que se describa el ¿Que? Y no el ¿Como?. Estos requerimientos al tiempo que avanza el proyecto de software se convierten en los algoritmos, la lógica y gran parte del código del sistema.

3.2.2.4.2 Recopilación de requerimientos del cliente

- Se permitirá hacer Login y Logout a usuarios ya registrados.
- Permitirá el registro de nuevos usuarios.
- Permitirá la modificación de los datos de un usuario registrado.
- Permitirá mostrar el catálogo de todos los productos en todo momento.
- Permitirá mostrar el catálogo por categoría de todos los productos en todo momento.
- Permitirá mostrar la información de cada producto por separado.
- Permitirá añadir un producto a la cesta.
- Permitirá eliminar/modificar linea de un pedido.
- Permitirá ver la cesta de un usuario que este logueado.
- Permitirá ver los pedidos de un usuario que este logueado.
- Permitirá eliminar la cesta de pedidos de un usuario logueado.
- Permitirá tramitar un pedido de un usuario logueado.
- Permitirá exportar a pdf el catálogo de productos.

3.2.2.5. Requerimientos de desempeño

3.2.2.5.1 Descripción

Estos requerimientos nos informan las características de desempeño que deben de tener el sistema. ¿Que tan rápido?, ¿Que tan seguido?, ¿Cuantos recursos?, ¿Cuantas transacciones? .

Este tipo de requerimientos es de especial importancia en los sistemas de tiempo real en donde el desempeño de un sistema es tan crítico como su funcionamiento.

3.2.2.5.2 Recopilación de requerimientos del cliente

- Apartado sin requeriemientos ya que aquí entramos en un tema muy específico y muy técnico.
- El aplicativo tendrá que ser rápido.
- Aceptar un número de transacciones concurrentes aceptable.
- Etc...

3.2.2.6. Disponibilidad

3.2.2.6.1 Descripción

Este tipo de requerimientos se refiere a la durabilidad, degradación, portabilidad, flexibilidad, contabilidad y capacidad de actualización. Este tipo de requerimientos es también muy importante en sistemas de tiempo real puesto que estos sistemas manejan aplicaciones críticas que no deben de estar fuera de servicio por periodos prolongados de tiempo.

3.2.2.6.2 Recopilación de requerimientos del cliente

- Apartado sin requeriemientos ya que aquí entramos en un tema muy específico y muy técnico.

3.2.2.7. Entrenamiento

3.2.2.7.1 Descripción

Este tipo de requerimientos se enfoca a las personas que van usar el sistema. ¿Que tipo de usuarios son?, ¿Que tipo de operadores?, ¿Que manuales se entregarán y en que idioma?

Este tipo de requerimientos, aunque muchas veces no termina en un pedazo de código dentro de el sistema, son muy importantes en el proceso de diseño ya que facilitan la introducción y aceptación de el sistema en donde será implementado.

3.2.2.7.2 Recopilación de requerimientos del cliente

- Se cuenta con que los usuarios serán de todo tipo : Tanto usuarios avanzados como usuarios con poca experiencia en el manejo de aplicativos online, por lo tanto como buscamos siempre solucionar el problema en el peor de los casos, el aplicativo tendrá que ser 'intuitivo' y de fácil aprendizaje.
- El aplicativo no contará con ningun tipo de manual de utilización, ya que será un aplicativo de carácter ***online***, por lo tanto no es factible el hecho de anexar ningun tipo de ayuda.

3.2.2.8. Restricciones de diseño

3.2.2.8.1 Descripción

Muchas veces las soluciones de un sistema de software son normadas por leyes o estándares, este tipo de normas caen como "restricciones de diseño".

3.2.2.8.2 Recopilación de requerimientos del cliente

- No hay ningun tipo de restricción de diseño.

3.2.2.9. Materiales

3.2.2.9.1 Descripción

Aquí se especifica en que medio se entregara el sistema y como esta empaquetado. Es importante para definir los costos de industrialización del sistema.

3.2.2.9.2 Recopilación de requerimientos del cliente

- Al tratarse de un aplicativo online en este apartado sólo se tendrá en cuenta que el sistema será implantado en un servidor y debería tener un responsable que diese soporte al sistema, para estar pendiente a posibles caidas del servidos, fallos en el sistema, interrupciones temporales del servicio, etc..
- Los *resources* del aplicativo serán entregados en un ***war (web application resources)***, para su fácil despliegue en cualquier ***Servidor Web*** o ***Servidor de Aplicaciones***. Este ***war*** contendrá la estructura lógica de carpetas de cualquier aplicación Web para su correcto despliegue.

3.2.3. Manejo de requerimientos.

De acuerdo con el "Capability Maturity Model" (CMM) [A4], el manejo de requerimientos involucra:

"Establecer y mantener un acuerdo con el cliente sobre los requerimientos de el proyecto de software. Este acuerdo son los requerimientos de el sistema alojados al software." … "Este acuerdo cubre requerimientos técnicos y no técnicos (como fechas de entrega). El acuerdo forma las bases para estimar, planear, ejecutar y monitorear el proyecto de desarrollo de software a través de todo su ciclo de vida." … "Bajo las restricciones del proyecto, el grupo de manejo de requerimientos toma las medidas necesarias para que los requerimientos que están bajo su responsabilidad estén documentados y controlados"

¿De que manera podemos controlar los requerimientos de software si estos siempre evolucionan con el tiempo?. El CMM nos proporciona las guías para lograrlo.

"Para lograr el control de los requerimientos, el grupo de requerimientos revisa los requerimientos antes de que estos sean incorporados al proyecto de software y cada vez que los requerimientos cambian los planes, productos, y actividades son ajustadas para quedar en línea con los nuevos requerimientos de software".

En otras palabras, para obtener el nivel que requiere el CMM en manejo de requerimientos débenos de tomar en cuenta dos cosas.

Que los requerimientos deben de ser **revisados (y aprobados)** por el grupo de requerimientos, y **no son impuestos** por en su totalidad por presiones externas ajenas al proyecto.

El requerimiento técnico podrá ser impuesto por el mercado o presiones de la competencia, pero entonces los requerimientos no técnicos (Calidad, Costo y Tiempo de entrega) deberán estar especificados de común acuerdo con el grupo de requerimientos del proyecto de software.

Los requerimientos técnicos y no técnicos forman un conjunto entre si, si cambia uno forzosamente deberán cambiar los demás. Esto es: **más contenido técnico implica o más costo, o menos calidad o mas tiempo estimado de entrega.** De modo que los cambios técnicos deberán ser aprobados por el grupo de requerimientos y este grupo estimará los impactos en tiempo, costo, calidad. El resultado de la estimación es la entrada a los líderes del proyecto para decidir si el cambio se acepta o no.

Estos dos puntos son los esenciales del manejo de requerimientos en CMM.

Una version completa del CMM (en ingles) puede ser bajada gratuitamente de el "Software Engineering Institute" de la Universidad de Carnegie Mellon.

3.3. Diseño

3.3.1. El concepto de servicio

La capa lógica es la encargada de proporcionar implementaciones de la lógica de negocio para la capa cliente. Está basada en el concepto de servicio, el cual podríamos definir como un grupo de funcionalidades relacionadas entre sí. Así pues, un servicio de la capa de lógica proporcionará diferentes métodos que contienen la implementación necesaria para solucionar una funcionalidad requerida para el sistema, y cada uno de ellos podrá ser invocado de manera independiente del resto.

3.3.2. Construcción de servicios

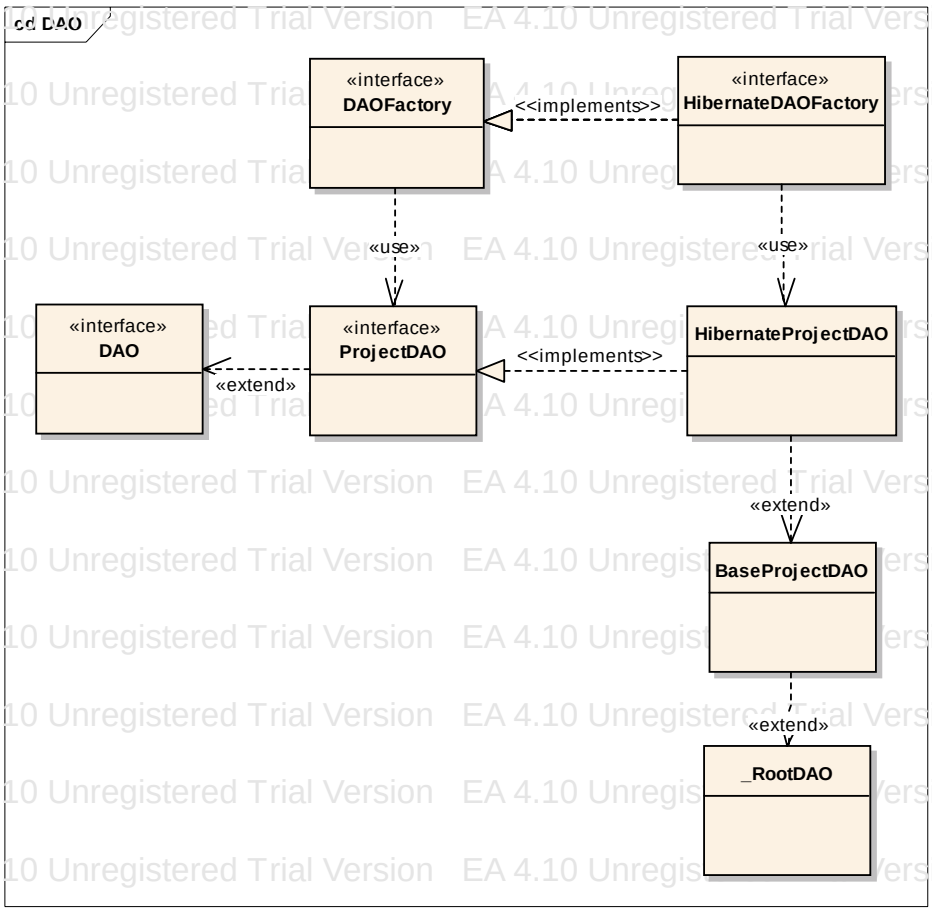
A continuación podemos ver los pasos necesarios para crear un componente de negocio bajo la concepción de servicio:

- Construir un interface de tipo ***BusinessInterface***, que publique los métodos de negocio que el servicio ofrece. Dicho interface ha de extender el interface de la arquitectura ***common***.***saf***.***j2ee***.***common***.***business***.***BusinessInterface***, y se ha de denominar siguiendo el patrón de nomenclatura ***XXXService***, donde ***XXX*** es la etiqueta identificativa del servicio, y el sufijo ***Service*** denota el hecho de ofrecer un servicio.
- Construir una clase de tipo ***BusinessObject***, que implemente el ***BusinessInterface***, y que contenga la implementación de la lógica de negocio requerida para el servicio. Se ha de denominar siguiendo el patrón de nomenclatura ***XXXBO***, donde ***XXX*** es la etiqueta identificativa del servicio, y el sufijo ***BO*** denota el hecho de ser un ***BusinessObject***.
- Construir una clase de tipo ***BusinessDelegate***, que implemente el ***BusinessInterface***, que realice labores de fachada, interceptando las peticiones y delegándolas al ***BusinessObject***. Se ha de denominar siguiendo el patrón de nomenclatura ***XXXDelegate***, donde ***XXX*** es la etiqueta identificativa del servicio.

3.3.3. Modelo de datos

Para resolver la capa de acceso a los datos existen diferentes estrategias como el uso del patrón ***DAO***, el uso de frameworks de control de la persistencia o también, alguna combinación de los anteriores. Más adelante vamos a ver la combinación del patrón arquitectural ***DAO*** junto con el uso de un framework de persistencia como ***Hibernate***.

El patrón ***DAO*** (Data Access Object) es el elegido para manejar la persistencia de los objetos del modelo del negocio, formando una capa separada que aísla al resto de capas de los detalles de implementación de acceso a datos. Un objeto ***DAO*** puede controlar la persistencia de un único objeto ***TransferObject*** simple, o de varios objetos ***TransferObject*** con una relación de agregación entre ellos. Así pues, no es necesario que haya un objeto ***DAO*** para cada objeto ***TransferObject*** existente en la aplicación. En el siguiente diagrama se muestra la relación entre los objetos de nuestra implementación del patrón ***DAO***:



Patrón arquitectural DAO

Para la localización de objetos ***DAO***, se seguirá el patrón factoría, para ello la arquitectura proporciona una clase abstracta: ***common.saf.j2ee.database.dao.DAOFactory*** y los objetos factorías concretas, que implementan la factoría abstracta y que son los encargados de instanciar los diferentes ***DAO***. La clase ***DAOFactory*** abstracta declara métodos para localizar los objetos ***DAO***, del tipo ***getXXXDAO***(), donde ***XXX*** hace referencia al objeto de negocio del que el ***DAO*** controla la persistencia. Se han de implementar tantas factorías concretas como sistemas gestores de persistencia haya (por ejemplo, puede haber una única factoría relacionada con el motor de persistencia Hibernate), y así como tantos métodos de obtención de ***DAO's*** como objetos de este tipo haya. La clase factoría concreta se especifica en el siguiente parámetro del fichero de configuración de la aplicación:

El tipo de factoría usada, así como el tipo de ***DAO*** concreto son totalmente transparentes para el cliente.

Cada clase ***DAO*** es la encargada de dar persistencia en el sistema elegido al ***TransferObject*** concreto, accediendo a dicho sistema mediante conexiones o sesiones proporcionadas por Servicios de Acceso específicos para cada sistema gestor de persistencia. Un ***DAO*** ha de implementar un interface específico para cada ***TransferObject***, que a su vez ha de extender del interface genérico ***DAO***. Para la construcción de objetos ***DAO*** se han de seguir las siguientes normas:

- Construir un interface que declare los métodos a implementar por la clase ***DAO*** encargada de proporcionar persistencia al ***TO*** correspondiente. Dicho interface ha de extender del interface genérico: ***common.saf.j2ee.database.dao.DAO***.
- Implementar los interfaces ***DAO*** necesarios según los sistemas gestores de persistencia que necesitemos, implementado los métodos definidos anteriormente.
- Implementar el método adecuado ***getXXXDAO ()*** en la factoría concreta, que permita localizar el ***DAO*** específico.

Además, los interfaces de tipo ***DAO*** han de seguir el patrón de nomenclatura ***XXXDAO***, donde ***XXX*** es el nombre del objeto del modelo de negocio al que dotan de persistencia, y ***DAO*** denota el hecho de tratarse de un objeto siguiendo dicho patrón.

3.3.4. Comunicación de datos entre las capas de aplicaciones

La transferencia de datos entre capas se hará siguiendo el patrón ***TransferObject***, un patrón ampliamente usado en la transferencia de datos entre capas, y en el cual los objetos ***TransferObject*** son representaciones de objetos del modelo de negocio. Básicamente, son objetos muy sencillos, simples contenedores de datos, correspondientes a clases serializables, con atributos privados y métodos getter/setter públicos, y los constructores necesarios. Todo objeto de negocio, habrá de ser diseñado siguiendo el patrón ***TO***, ya sea mediante un único ***TransferObject*** simple, o mediante un ***TransferObject*** compuesto de agregaciones de otros objetos ***TransferObject***. Para la construcción de objetos ***TO*** se ha de construir una clase pública siguiendo las siguientes normas:

- Se ha de implementar el interface proporcionado por la arquitectura ***java.io.Serializable***.
- Han de disponer de al menos un constructor sin argumentos. Se pueden desarrollar tantos constructores como se considere necesario.

- Todos sus atributos, ya sean tipos primitivos, tipos Java, u otros ***TransferObject***, o colecciones de alguno de éstos, son definidos como privados, y además son serializables.
- Se han de proporcionar métodos getter y setter públicos para cada uno de los atributos definidos.
- Opcionalmente, se ha de rescribir el método toString () heredado de la clase Object, para que devuelva una representación del objeto ***TransferObject***, mostrando todos sus atributos. El formato para dicha representación es el siguiente: [nombreAttr1=valor, nombreAttr2=valor,....]. Se ha de tener en cuenta que si el ***TransferObject*** contiene algún ***TransferObject*** agregado, la representación de este atributo se ha de realizar invocando el respectivo método toString () de éste.

Además, los objetos de tipo ***TransferObject*** han de seguir el patrón de nomenclatura ***XXXTO***, donde ***XXX*** es el nombre del objeto del modelo de negocio que representa el ***TransferObject***, y ***TO*** denota el hecho de tratarse de un objeto siguiendo dicho patrón.

3.3.5. Patrones de diseño utilizados en el aplicativo

3.3.5.1. BUSINESS DELEGATE

3.3.5.1.1 Contexto

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

3.3.5.1.2 Problema

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

servicio de negocio, la implementación del código expuesto en la capa de presentación también debe cambiar.

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

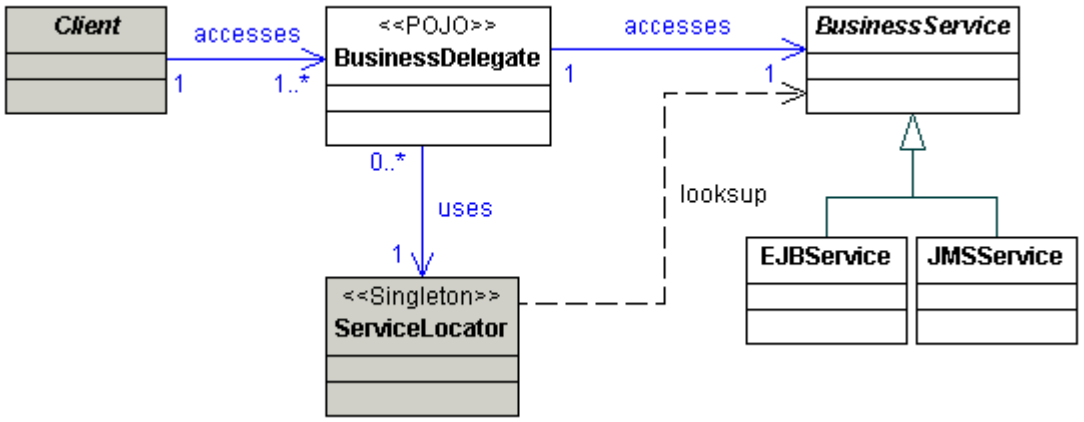
3.3.5.1.3 Causas

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

3.3.5.1.4 Solución

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

El ***Business Delegate*** actúa como una abstracción de negocio del lado del cliente; proporciona una abstracción para, y por lo tanto oculta, la implementación de los servicios del negocio. Utilizando ***Business Delegate*** se reduce el acoplamiento entre los clientes de la capa de presentación y los servicios de negocio del sistema. Dependiendo de la estrategia de implementación, ***Business Delegate*** podría aislar a los clientes de la posible volatilidad en la implementación del API de los servicios de negocio. Potencialmente, esto reduce el número de cambios que se deben hacer en el código de cliente de la capa de presentación cuando cambie el API del servicio de negocio o su implementación subyacente.



Patrón Business Delegate

Sin embargo, los métodos de interface en el ***Business Delegate*** aún podrían requerir modificaciones si cambia el API del servicio de negocio. Si bien es cierto, que los cambios se harán con más probabilidad en el servicio de negocio que en el ***Business Delegate***.

Con frecuencia, los desarrolladores son escépticos cuando un objetivo de diseño como la abstracción de la capa de negocio provoca un trabajo adicional como pago por futuras ganancias. Sin embargo, utilizando este patrón o esta estrategia resulta sólo en una pequeña cantidad de trabajo extra y proporciona unos beneficios considerables. El principal beneficio es ocultar los detalles del servicio. Por ejemplo, el cliente puede ser transparente para los servicios de búsqueda y nombrado. El ***Business Delegate*** también maneja las excepciones de los servicios de negocio.

El ***Business Delegate*** podría interceptar dichas excepciones a nivel de servicio y generar en su lugar excepciones a nivel de aplicación. Las excepciones de nivel de aplicación son fáciles de manejar por los clientes, y pueden ser amigables para el usuario. El ***Business Delegate*** también podría realizar de forma transparente cualquier operación de reintento o de recuperación necesaria en el caso de un fallo en el servicio no exponer el cliente al problema hasta que se haya determinado que el problema no es solucionable. Estas ganancias representan una razón competitiva para utilizar el patrón.

Otro beneficio es que el delegado podría almacenar resultados y referencias a servicios de negocio remotos. El Caché puede mejorar el rendimiento de forma significativa, porque limita los innecesarios y potencialmente costosos viajes por la red.

Finalmente, se debería tener en cuenta que este patrón se podría utilizar para reducir el acoplamiento entre otra capas, no simplemente entre las capas de presentación y de negocio.

3.3.5.1.5 Consecuencias

- Reduce el Acoplamiento y la manejabilidad: El ***Business Delegate*** reduce el acoplamiento entre las capas de presentación y de negocio ocultando todos los detalles de implementación de la capa de negocio. Es fácil manejar los cambios porque están centralizados en un sólo lugar, el ***Business Delegate***.
- Puede traduce las excepciones del servicio de negocio: El ***Business Delegate*** es el responsable de traducir cualquier excepción de red o relacionada con la infraestructura en excepciones de negocio, aislando a los clientes del conocimiento de las particularidades de la implementación.
- Implementa recuperación de fallos y sincronización de Threads: Cuando el ***Business Delegate*** encuentra un fallo en el servicio de negocio, puede implementar características de recuperación automática sin exponer el problema al cliente. Si la recuperación tiene éxito, el cliente no necesita saber nada sobre el fallo. Si el intento de recuperación no tiene éxito, entonces el ***Business Delegate*** necesita informar al cliente del fallo. Además, los métodos del ***Business Delegate*** podrían estar sincronizados, si fuera necesario.
- Expone un Interface Simple y Uniforme a la Capa de Negocio: El ***Business Delegate***, para servir mejor a sus clientes, podría proporcionar una variante del interface proporcionado por los EJB subyacentes.
- Impacto en el Rendimiento: El ***Business Delegate*** podría proporcionar servicio de caché (y un mejor rendimiento) a la capa de presentación para las peticiones de servicios comunes.
- Presenta una Capa Adicional: El ***Business Delegate*** podría verse como la adicción de una capa innecesaria entre el cliente y el servicio, y con eso incrementar la complejidad y disminuir la flexibilidad.
- Oculta los elementos Remotos: Aunque la localización transparente es uno de los beneficios de este patrón, podría surgir un problema diferente debido a que el desarrollador está tratando con un servicio remoto como si fuera un servicio local. Esto podría suceder si el desarrollador del cliente no entiende que el ***Business Delegate*** es cliente-proxy a un servicio remoto. Normalmente, unas llamadas a métodos en el Business Delegate resultan en unas llamadas a métodos remotos.

3.3.5.2. TRANSFER OBJECT

3.3.5.2.1 Contexto

Las aplicaciones cliente necesitan intercambiar datos con Beans Enterprise.

3.3.5.2.2 Problema

Las aplicaciones de la plataforma J2EE implementan componentes de negocio del lado del servidor como beans de sesión y de entidad. Algunos métodos expuestos por los componentes de negocio devuelven datos al cliente. Algunas veces, el cliente invoca a los métodos get de un objeto de negocio varias veces para obtener todos los valores de los atributos.

Los beans de sesión representan los servicios de negocio y no se comparten entre usuarios. Un bean de sesión proporciona métodos de servicios genéricos cuando se implementan mediante el patrón ***Session Facade***.

Por otro lado, los beans de entidad, son objetos transaccionales, multiusuario que representan datos persistentes. Un bean de entidad expone los valores de los atributos proporcionando un método accesor (también referidos como métodos get) por cada atributo que desea exponer.

Toda llamada a método hecha al objeto de servicio de negocio, ya sea a un bean de entidad o a un bean de sesión, potencialmente es una llamada remota. Así, en una aplicación de JavaBeans Enterprise (EJB) dichas llamadas remotas usan la capa de red sin importar la proximidad del cliente al bean, creando una sobrecarga en la red. Las llamadas a métodos de beans enterprise podría penetrar las capas de red incluso si tanto el cliente como el contenedor EJB que mantiene el bean de entidad se están ejecutando en la misma JVM (Máquina Virtual Java), el mismo Sistema Operativo o máquina física. Algunos vendedores podrían implementar mecanismos para reducir esta sobrecarga utilizando una aproximación de acceso más directa pasando por encima de la red.

Según se incrementa la utilización de estos métodos remotos, el rendimiento de la aplicación se puede degradar significativamente. Por lo tanto, utilizar varias llamadas a métodos get que devuelven simples valores de atributos es ineficiente para obtener valores de datos desde un bean enterprise.

3.3.5.2.3 Causas

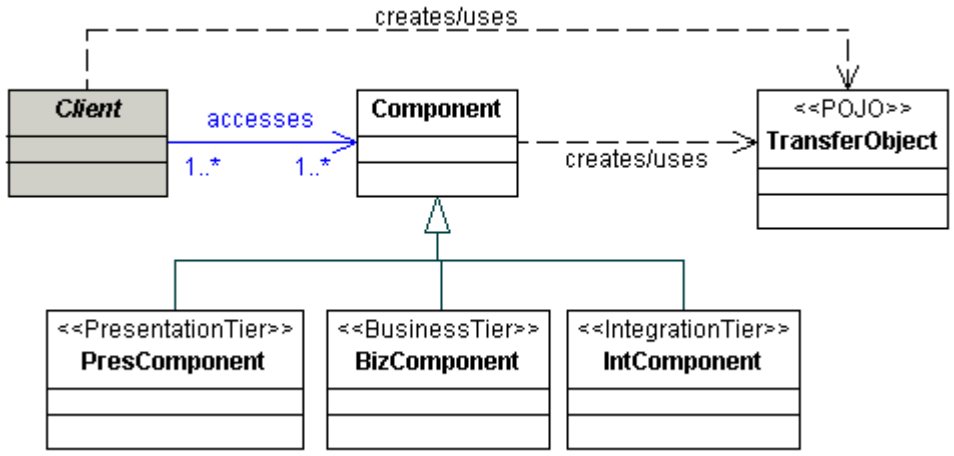
- Todos los accesos a un bean enterprise se realizan mediante interfaces remotos. Cada llamada a un bean enterprise potencialmente es una llamada a un método remoto con sobrecarga de red.
- Normalmente, las aplicaciones tienen transacciones de lectura con mayor frecuencia que las de actualización. El cliente solicita los datos desde la capa de negocio para su presentación, y otros tipos de procesamientos de sólo-lectura. El cliente actualiza los datos de la capa de negocio con mucha menos frecuencia con la que los lee.
- El cliente normalmente solicita valores que son algo más que un atributo o que son dependientes del objeto de un bean enterprise. Así, el cliente podría invocar varias llamadas remotas para obtener los datos necesarios.
- El número de llamadas que un cliente hace al bean enterprise impactan en el rendimiento de la red.

3.3.5.2.4 Solución

Utilizar un ***Transfer Object*** para encapsular los datos de negocio. Se utiliza una única llamada a un método para enviar y recuperar el ***Transfer Object***. Cuando el cliente solicita los datos de negocio al bean enterprise, éste puede construir el ***Transfer Object***, rellenarlo con sus valores de atributos y pasarlo por valor al cliente.

Los clientes normalmente solicitan más de un valor a un bean enterprise. Para reducir el número de llamadas remotas y evitar la sobrecarga asociada, es mejor el ***Transfer Objects*** para transportar los datos desde el bean enterprise al cliente.

Cuando un bean enterprise utiliza un ***Transfer Object***, el cliente hace una sola llamada a un método remoto del bean enterprise para solicitar el ***Transfer Object*** en vez de numerosas llamadas remotas para obtener valores de atributos individuales. Entonces el bean enterprise construye un nuevo ejemplar ***Transfer Object***, copia dentro los valores del objeto y lo devuelve al cliente. El cliente recibe el ***Transfer Object*** y puede entonces invocar los métodos de acceso (o get) del ***Transfer Object*** para obtener los valores de atributos individuales del objeto ***Transfer Object***. O, la implementación del ***Transfer Object*** podría hacer que todos los atributos fueran públicos.



Patrón Transfer Object

Como el ***Transfer Object*** se pasa por valor al cliente, todas las llamadas al ejemplar ***Transfer Object*** son llamadas locales en vez de invocaciones de métodos remotos.

3.3.5.2.5 Consecuencias

- Simplifica el Bean de Entidad y el Interface Remoto: El bean de entidad proporciona un método getData() para obtener el ***Transfer Object*** que contiene los valores de los atributos. Esto podría eliminarse implementando múltiples métodos get en el bean y definiéndolos en el interface remoto del bean. De forma similar, si el bean de entidad proporciona un método setData() para actualizar los valores de atributos del bean de entidad con una sola llamada a método, se podría eliminar implementando varios métodos set en el bean.
- Transfiere más Datos en menos llamadas remotas: En lugar de realizar múltiples llamadas sobre la red al ***BusinessObject*** para obtener los valores de los atributos, esta solución proporciona una sola llamada a un método. Al mismo tiempo, esta única llamada obtiene una mayor cantidad de datos. Cuando consideremos la utilización de este patrón, debemos tener en cuenta el inconveniente de disminuir el número de llamadas contra la mayor transmisión de datos por cada llamada.

- Reduce el tráfico de red: Un ***Transfer Object*** transfiere los valores desde el bean de

entidad al cliente en una llamada a un método remoto. El ***Transfer Object*** actúa como un transportista de datos y reduce el número de llamadas remotas requeridas para obtener los valores de los atributos del bean; y esto significa un mejor rendimiento de la red.

- Accesos y transacciones concurrentes: Cuando dos o más clientes acceden de forma concurrente al ***BusinessObject*** pueden aparecer inconsistencias en el Transfer Objects por accesos concurrentes. Además, tendremos que tratar con problemas relacionados con la sincronización, el control de versión y los ***Transfer Objects*** obsoletos.

3.3.5.3. DATA ACCESS OBJECT

3.3.5.3.1 Contexto

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

3.3.5.3.2 Problema

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

Normalmente, las aplicaciones utilizan componentes distribuidos y compartidos como los beans de entidad para representar los datos persistentes. Se considera que una aplicación emplea consistencia manejada por el bean (BMP) cuando sus beans de entidad acceden explícitamente al almacenamiento persistente -- el bean de entidad incluye código para hacer esto. Una aplicación con requerimientos sencillos podría por lo tanto utilizar beans de entidad en lugar de beans de sesión o servlets para acceder al almacenamiento persistente y recuperar o modificar los datos. O, la aplicación podría usar

beans de entidad con persistencia manejada por el contenedor, y así dejar que el contenedor maneje los detalles de las transacciones y de la persistencia.

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

Incluso hay una mayor variación con diferentes tipos de almacenamientos persistentes. Los mecanismos de acceso, los APIs soportados, y sus características varían entre los diferentes tipos de almacenamientos persistentes, como bases de datos relacionales, bases de datos orientadas a objetos, ficheros planos, etc. Las aplicaciones que necesitan acceder a datos de un sistema legal o un sistema dispar (como un mainframe o un servicio B2B) se ven obligados a utilizar APIs que podrían ser propietarios. Dichas fuentes de datos dispares ofrecen retos a la aplicación y potencialmente pueden crear una dependencia directa entre el código de la aplicación y el código de acceso a los datos. Cuando los componentes de negocio -- beans de entidad, beans de sesión e incluso componentes de presentación como servlets y beans de apoyo para páginas JSP -- necesitan acceder a una fuente de datos, pueden utilizar el API apropiado para conseguir la conectividad y manipular la fuente de datos. Pero introducir el código de conectividad y de acceso a datos dentro de estos componentes genera un fuerte acoplamiento entre los componentes y la implementación de la fuente de datos. Dichas dependencias de código en los componentes hace difícil y tedioso migrar la aplicación de un tipo de fuente de datos a otro. Cuando cambia la fuente de datos, también deben cambiar los componentes para manejar el nuevo tipo de fuente de datos.

3.3.5.3.3 Causas

- Los componentes como los beans de entidad controlados por el bean, los beans de sesión, los servlets, y otros objetos como beans de apoyo para páginas JSP necesitan recuperar y almacenar información desde almacenamientos persistentes y otras fuentes de datos como sistemas legales, B2B, LDAP, etc.
- Los APIs para almacenamiento persistente varían dependiendo del vendedor del producto. Otras fuentes de datos podrían tener APIS que no son estándar y/o propietarios. Estos APIs y sus capacidades también varían dependiendo del tipo de almacenamiento -- bases de datos relacionales, bases de datos orientadas a objetos,

documentos XML, ficheros planos, etc. Hay una falta de APIs uniformes para corregir los requerimientos de acceso a sistemas tan dispares.

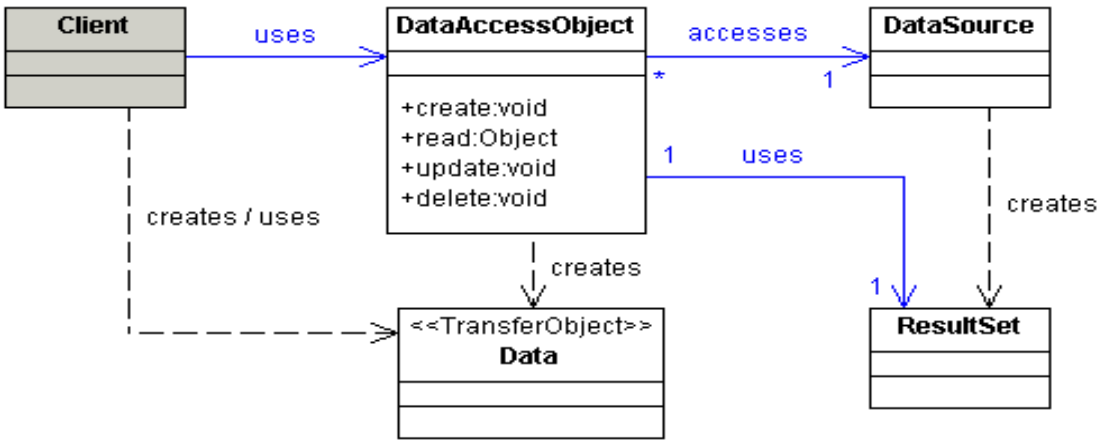
- Los componentes normalmente utilizan APIs propietarios para acceder a sistemas externos y/o legales para recuperar y almacenar datos.
- La portabilidad de los componentes se ve afectada directamente cuando se incluyen APIs y mecanismos de acceso específicos.
- Los componentes necesitan ser transparentes al almacenamiento persistente real o la implementación de la fuente de datos para proporcionar una migración sencilla a diferentes productos, diferentes tipos de almacenamiento y diferentes tipos de fuentes de datos.

3.3.5.3.4 Solución

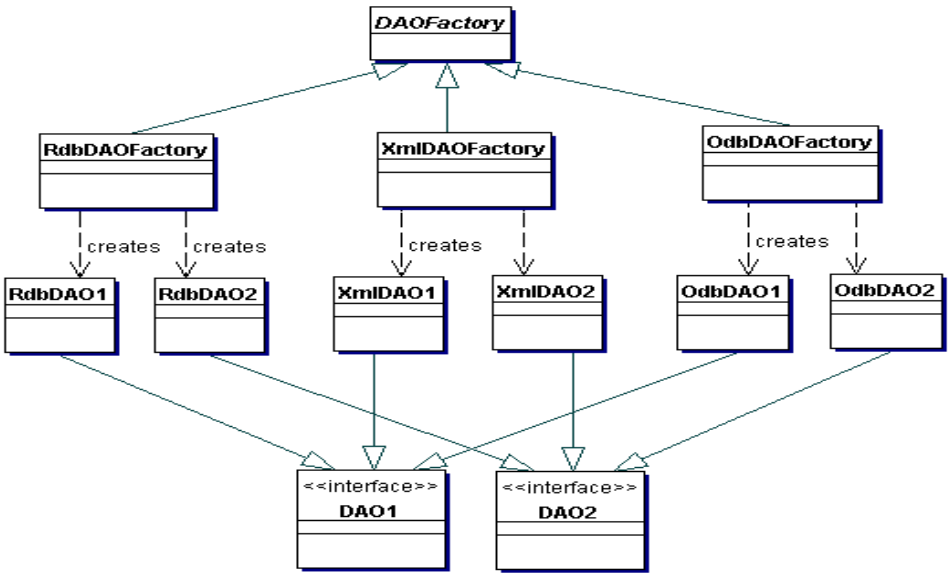
Utilizar un ***Data Access Object*** (***DAO***) para abstraer y encapsular todos los accesos a la fuente de datos. El ***DAO*** maneja la conexión con la fuente de datos para obtener y almacenar datos.

El ***DAO*** implementa el mecanismo de acceso requerido para trabajar con la fuente de datos. Esta fuente de datos puede ser un almacenamiento persistente como una RDMBS, un servicio externo como un intercambio B2B, un repositorio LDAP, o un servicio de negocios al que se accede mediante CORBA Internet Inter-ORB Protocol (IIOP) o sockets de bajo nivel.

Los componentes de negocio que tratan con el ***DAO*** utilizan un interface simple expuesto por el ***DAO*** para sus clientes. El ***DAO*** oculta completamente los detalles de implementación de la fuente de datos a sus clientes. Como el interface expuesto por el ***DAO*** no cambia cuando cambia la implementación de la fuente de datos subyacente, este patrón permite al ***DAO*** adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio. Esencialmente, el ***DAO*** actúa como un adaptador entre el componente y la fuente de datos.



Patrón Data Access Object



Factoría abstracta para la gestión de DAO

3.3.5.3.5 Consecuencias

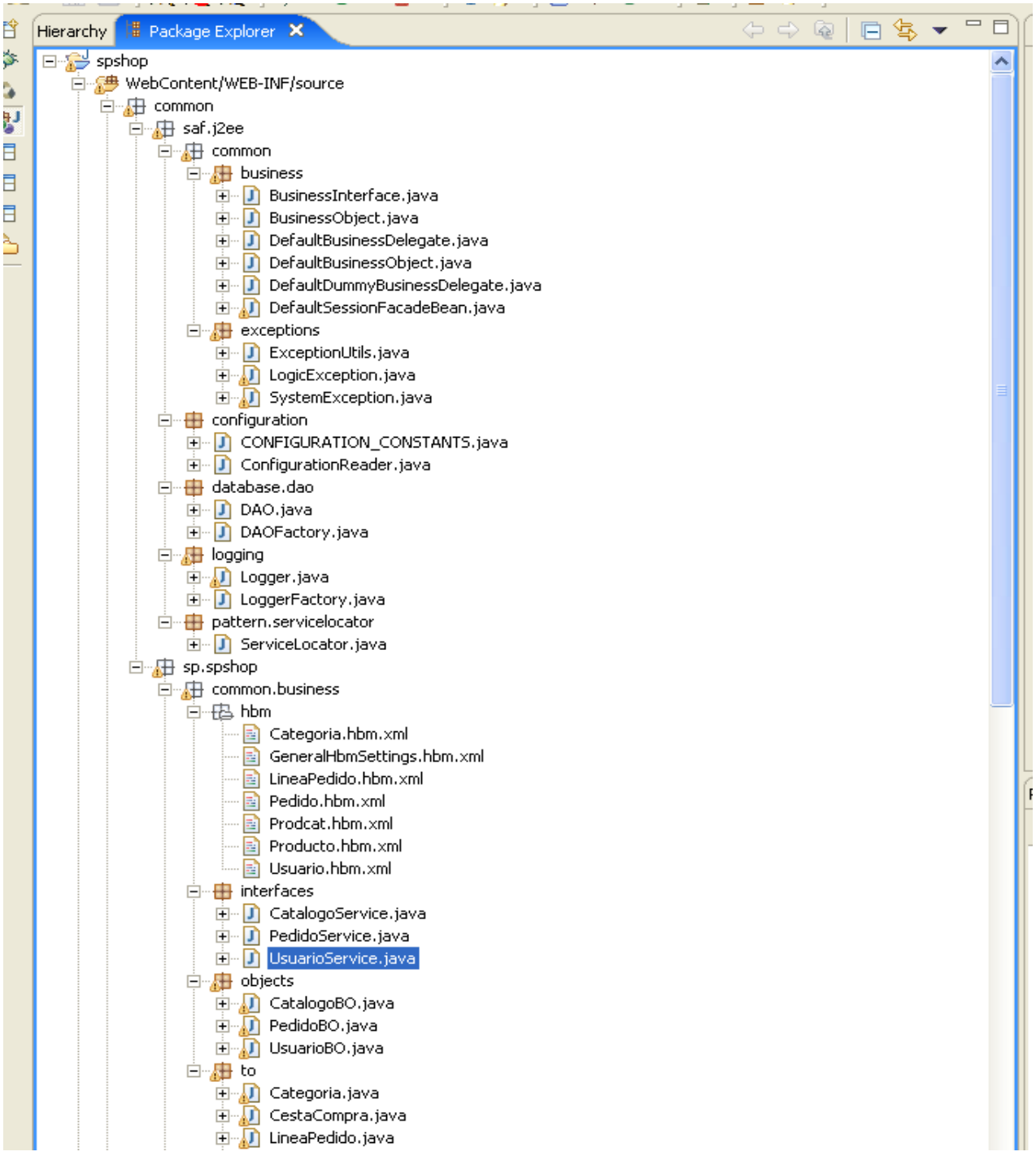
- Permite la transparencia: Los objetos de negocio puede utilizar la fuente de datos sin

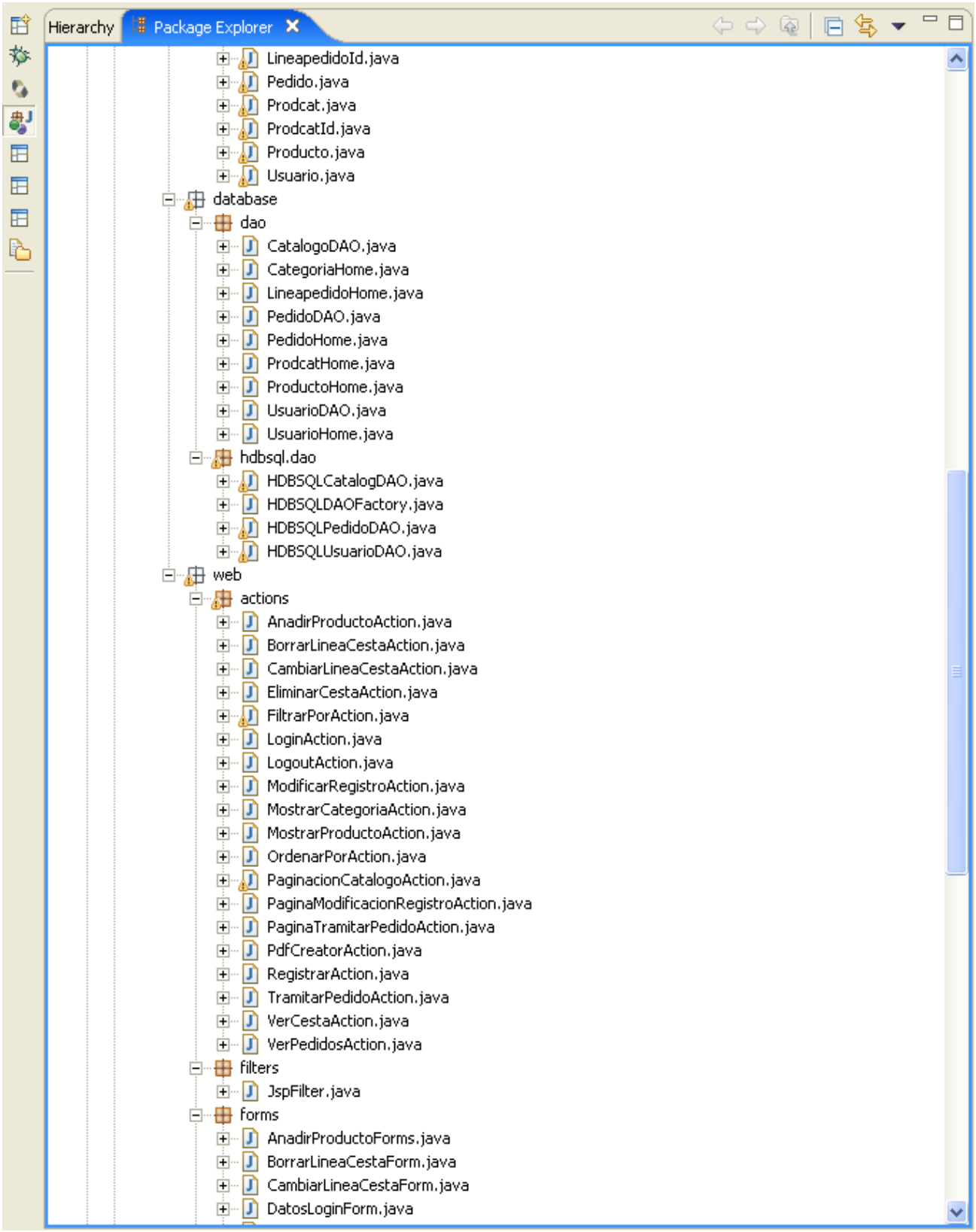
conocer los detalles específicos de su implementación. El acceso es transparente porque los detalles de la implementación se ocultan dentro del ***DAO***.

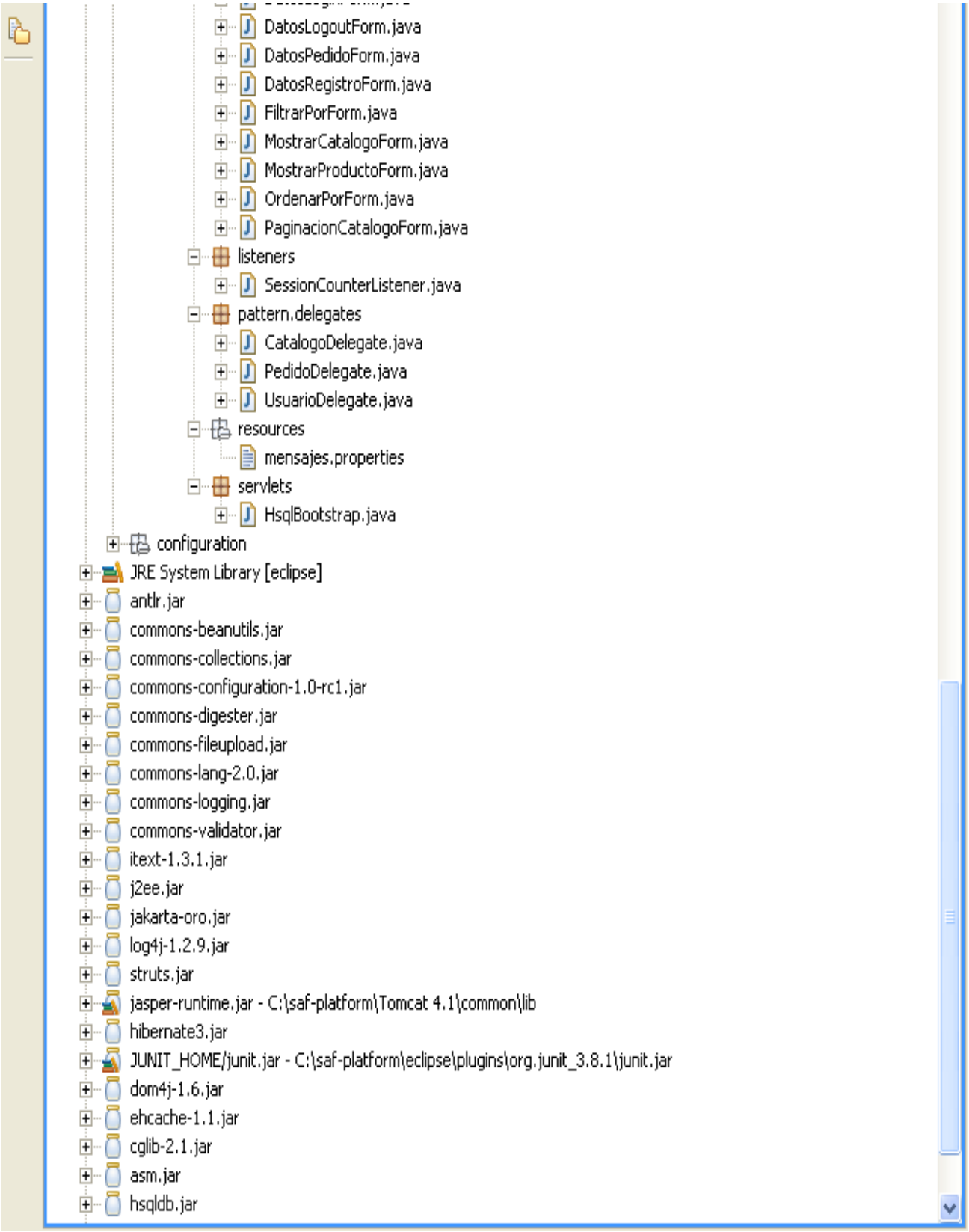
- Permite una migración más fácil: Una capa de ***DAOs*** hace más fácil que una aplicación pueda migrar a una implementación de base de datos diferente. Los objetos de negocio no conocen la implementación de datos subyacente, la migración implica cambios sólo en la capa ***DAO***. Además, si se emplea la estrategia de factorías, es posible proporcionar una implementación de factorías concretas por cada implementación del almacenamiento subyacente. En este caso, la migración a un almacenamiento diferente significa proporcionar a la aplicación una nueva implementación de la factoría.
- Reduce la complejidad del código de los objetos de negocio: Como los ***DAOs*** manejan todas las complejidades del acceso a los datos, se simplifica el código de los objetos de negocio y de otros clientes que utilizan los ***DAOs***. Todo el código relacionado con la implementación (como las sentencias SQL) están dentro del ***DAO*** y no en el objeto de negocio. Esto mejora la lectura del código y la productividad del desarrollo.
- Centraliza todos los accesos a datos en un capa independiente: Como todas las operaciones de acceso a los datos se ha delegado en los ***DAOs***, esto se puede ver como una capa que aísla el resto de la aplicación de la implementación de acceso a los datos. Esta centralización hace que la aplicación sea más sencilla de mantener y de manejar.

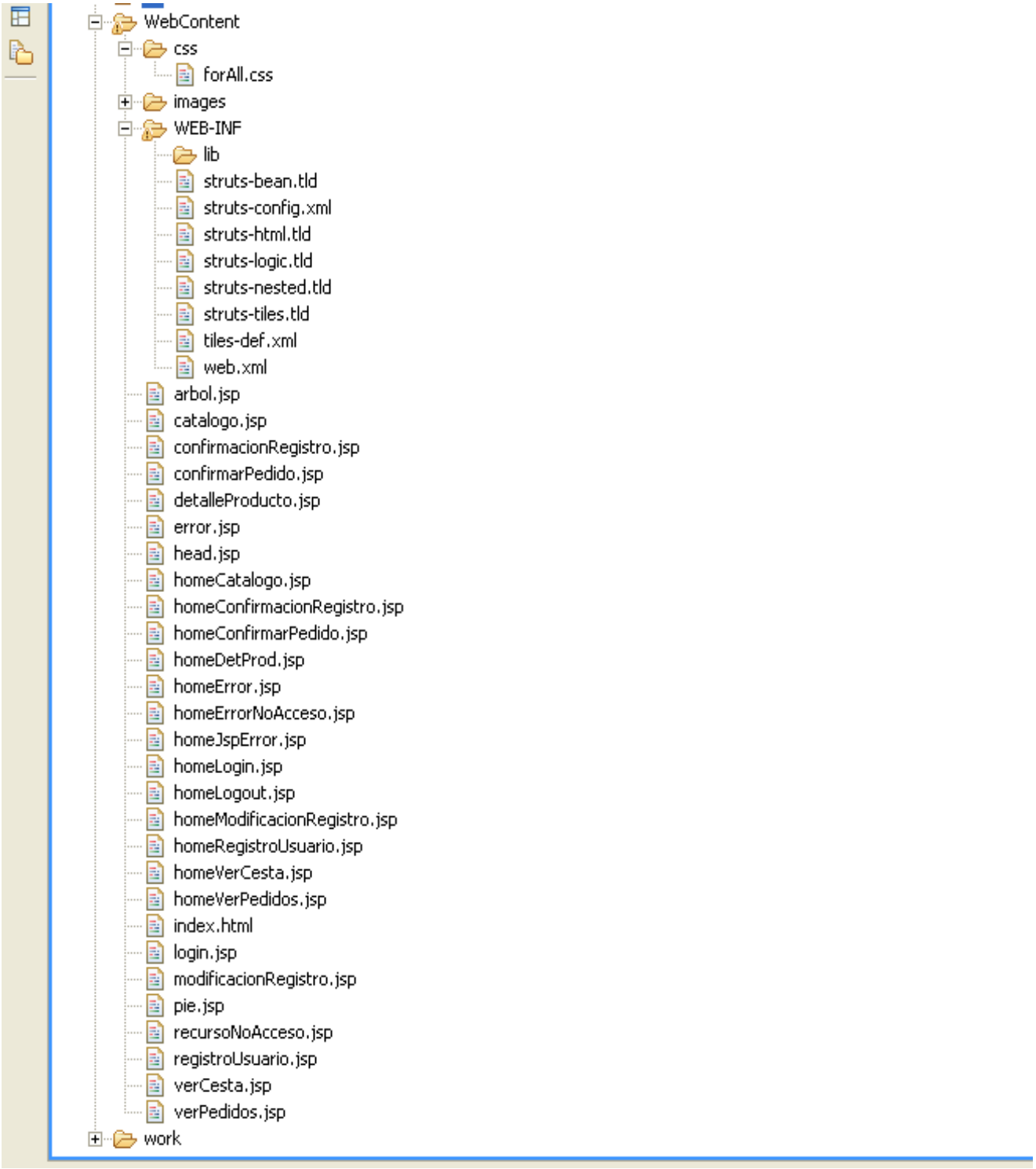
3.4. Implementación

3.4.1. Organización del repositorio




Hierarchy
Package Explorer
LineapedidoId.java
Pedido.java
Prodcat.java
ProdcatId.java
Producto.java
Usuario.java
database
dao
CatalogoDAO.java
CategoriaHome.java
LineapedidoHome.java
PedidoDAO.java
PedidoHome.java
ProdcatHome.java
ProductoHome.java
UsuarioDAO.java
UsuarioHome.java
hdbsql.dao
HDBSQLCatalogDAO.java
HDBSQLDAOFactory.java
HDBSQLPedidoDAO.java
HDBSQLUsuarioDAO.java
web
actions
AnadirProductoAction.java
BorrarLineaCestaAction.java
CambiarLineaCestaAction.java
EliminarCestaAction.java
FiltrarPorAction.java
LoginAction.java
LogoutAction.java
ModificarRegistroAction.java
MostrarCategoriaAction.java
MostrarProductoAction.java
OrdenarPorAction.java
PaginacionCatalogoAction.java
PaginaModificacionRegistroAction.java
PaginaTramitarPedidoAction.java
PdfCreatorAction.java
RegistrarAction.java
TramitarPedidoAction.java
VerCestaAction.java
VerPedidosAction.java
filters
JspFilter.java
forms
AnadirProductoForms.java
BorrarLineaCestaForm.java
CambiarLineaCestaForm.java
DatosLoginForm.java


DatosLogoutForm.java
DatosPedidoForm.java
DatosRegistroForm.java
FiltrarPorForm.java
MostrarCatalogoForm.java
MostrarProductoForm.java
OrdenarPorForm.java
PaginacionCatalogoForm.java
listeners
SessionCounterListener.java
pattern.delegates
CatalogoDelegate.java
PedidoDelegate.java
UsuarioDelegate.java
resources
mensajes.properties
servlets
HsqlBootstrap.java
configuration
JRE System Library [eclipse]
antlr.jar
commons-beanutils.jar
commons-collections.jar
commons-configuration-1.0-rc1.jar
commons-digester.jar
commons-fileupload.jar
commons-lang-2.0.jar
commons-logging.jar
commons-validator.jar
itext-1.3.1.jar
j2ee.jar
jakarta-oro.jar
log4j-1.2.9.jar
struts.jar
jasper-runtime.jar - C:\saf-platform\Tomcat 4.1\common\lib
hibernate3.jar
JUNIT_HOME/junit.jar - C:\saf-platform\eclipse\plugins\org.junit_3.8.1\junit.jar
dom4j-1.6.jar
ehcache-1.1.jar
cglib-2.1.jar
asm.jar
hsqldb.jar


WebContent
css
forAll.css
images
WEB-INF
lib
struts-bean.tld
struts-config.xml
struts-html.tld
struts-logic.tld
struts-nested.tld
struts-tiles.tld
tiles-def.xml
web.xml
arbol.jsp
catalogo.jsp
confirmacionRegistro.jsp
confirmarPedido.jsp
detalleProducto.jsp
error.jsp
head.jsp
homeCatalogo.jsp
homeConfirmacionRegistro.jsp
homeConfirmarPedido.jsp
homeDetProd.jsp
homeError.jsp
homeErrorNoAcceso.jsp
homeJspError.jsp
homeLogin.jsp
homeLogout.jsp
homeModificacionRegistro.jsp
homeRegistroUsuario.jsp
homeVerCesta.jsp
homeVerPedidos.jsp
index.html
login.jsp
modificacionRegistro.jsp
pie.jsp
recursoNoAcceso.jsp
registroUsuario.jsp
verCesta.jsp
verPedidos.jsp
work

3.4.2. Descriptor de despligue del Web Server (web.xml)

Para empezar describiremos el descriptor de despliegue del ***Web Server***, este fichero es un xml con el nombre **web.xml**.

El contenido de este fichero es básico en el arranque del ***Web Server*** , ya que en el mismo, se describen los elementos que contendrá y el modo en que se accede a los mismos. También se definen aspectos de seguridad, ficheros de bienvenida, parámetros iniciales, parámetros de contexto.

Al arrancar el ***Web Server*** lo primero que hace es ir en busca de este fichero y leer su contenido, cualquier fallo en el mismo arrojará una serie de excepciones en las cuales se indica que el arranque no ha sido satisfactorio.

Este fichero es privado, esto quiere decir que es inaccesible su contenido para los usuarios de futuras aplicaciones contenidas en el contenedor del ***Web Server***.

A continuación mostramos la definición del mismo y la explicación de cada una de las entradas del fichero.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp">
        <display-name>mercadoLibreOnline</display-name>

        <!-- filter configuration -->
        <filter>
                <filter-name>JspFilter</filter-name>
                <filter-class>common.sp.spshop.web.filters.JspFilter</filter-class>
        </filter>

        <filter-mapping>
                <filter-name>JspFilter</filter-name>
                <url-pattern>*.jsp</url-pattern>
        </filter-mapping>

        <!-- Register the session counting event listener. -->
        <listener>
                <listener-class>
                        common.sp.spshop.web.listeners.SessionCounterListener
                </listener-class>
        </listener>

        <!-- servlet configuration -->
        <servlet>
                <servlet-name>HsqlBootstrap</servlet-name>
                <display-name>HsqlBootstrap</display-name>
                <servlet-class>
                       common.sp.spshop.web.servlets.HsqlBootstrap
                </servlet-class>
                <load-on-startup>2</load-on-startup>
        </servlet>
        <servlet>
                <servlet-name>action</servlet-name>
                <display-name>ActionServlet</display-name>
                <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
                <init-param>
                        <param-name>config</param-name>
                        <param-value>/WEB-INF/struts-config.xml</param-value>
                </init-param>
```

```
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- end error page definition -->
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>2</session-timeout>
    </session-config>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <!-- error configuration -->
    <error-page>
        <error-code>404</error-code>
        <location>/homeJspError.jsp</location>
    </error-page>

    <!-- taglib definition -->
    <!-- struts tag library descriptors -->
    <taglib>
        <taglib-uri>/tags/struts-bean</taglib-uri>
        <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
    </taglib>
    <taglib>
        <taglib-uri>/tags/struts-html</taglib-uri>
        <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
    </taglib>
    <taglib>
        <taglib-uri>/tags/struts-logic</taglib-uri>
        <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
    </taglib>
    <taglib>
        <taglib-uri>/tags/struts-nested</taglib-uri>
        <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
    </taglib>
    <taglib>
        <taglib-uri>/tags/tiles</taglib-uri>
        <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
    </taglib>
    <!-- end struts tag library descriptors -->
    <!-- end taglib definition -->
</web-app>
```

1)

```
<display-name>mercadoLibreOnline</display-name>
```

Este elemento es opcional, es un nombre corto que posteriormente podrá ser utilizado por las herramientas de la GUI para ser mostrado.

2)

```
<filter>
        <filter-name>JspFilter</filter-name>
        <filter-class>common.sp.spshop.web.filters.JspFilter</filter-class>
    </filter>
```

En el descriptor de despliegue se pueden definir una serie de filtros, no es obligatorio tener definido ninguno, pero para nuestro aplicativo puede ser interesante.

En la definición del filtro, le indicamos el nombre ***JspFilter***, este nombre se utiliza para posteriores referéncias a este filtro dentro del descriptor de despliegue, y la clase que contendrá la implementación que se ejecutará cuando se aplique el filtro.

La implementación de la clase (***common.sp.spshop.web.filters.JspFilter***) la mostramos a continuación:

```
package common.sp.spshop.web.filters;

import (…)

/**
 * @author Eduardo Varga
 *
 */
public class JspFilter implements Filter{

	private FilterConfig config;

	public void init(FilterConfig config) throws ServletException {

		this.config=config;
	}

	public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws
IOException, ServletException {

		HttpServletResponse response = (HttpServletResponse) res;
		response.sendError(404);
	}

	public void destroy() {}
}
```

Como vemos nuestra clase implementará la interfície **javax.servlet.Filter,** esto supone implementar los métodos de dicha interfície, los métodos són los que se reseñan en **rojo**.

- `public void init(FilterConfig config) throws ServletException` será invocado por el container del ***Web Server*** cuando el filtro tenga que entrar en servicio. La inicialización puede ser interrumpida por dos razones.
  - O bien, se lanza una ***ServletException***.
  - O bien, no devuelve una respuesta en un tiempo definido en el proprio **web.xml**.
- `public void init(FilterConfig config) throws ServletException` será invocado por el container del ***Web Server*** cada vez que una request/response pase por el filtro, una vez haya sido inicializado.
- `public void destroy()`será invocado por el container del ***Web Server*** una vez el filtro tenga que finalizar su ejecución.

3)

```
<filter-mapping>
        <filter-name>JspFilter</filter-name>
        <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

El filtro anteriormente definido ha de ser mapeado para definir sobre que URL's ha de ser aplicado.

Como vemos el ***filter-name*** ha de coincidir con el nombre definido para esta propiedad en la definición del filtro ***JspFilter***, además, ha de incluir el parámetro ***url-pattern*** que definirá los elementos que recibirán el trato por parte del filtro.

La forma en la que el ***Web Server*** resuelve este ***url-pattern*** es :

http://host:port + ContextPath

Por lo tanto este filtro será aplicado en la resolución de todas las URL's que redirijan a una jsp.

4)

```
<listener>
        <listener-class>
                common.sp.spshop.web.listeners.SessionCounterListener
        </listener-class>
</listener>
```

Otra de las opciones que se nos permite en el ***web.xml***, es definir una serie de ***listeners***, esto nos permite tener un elemento que está pendiente de una serie de eventos en la aplicación.

Como podemos ver en el ejemplo de nuestra aplicación, la clase que implementará nuestro ***listener*** es **common.sp.spshop.web.listeners.SessionCounterListener**, esta clase será invocada en el momento que el evento para el que ha sido creada suceda.

A continuación mostramos el código de esta clase y explicaremos que contiene:

```
package common.sp.spshop.web.listeners;

import (…)

/**
* Nuestro listener guardará el número de sesiones que está utilizando la          aplicación Web
concurrentemente,
* ademas de las que ha utilizado durante su ciclo de vida.
*/

        public class SessionCounterListener implements HttpSessionListener {

        (…)

        public synchronized void sessionCreated(HttpSessionEvent event) {
        (…)
        }

        public synchronized void sessionDestroyed(HttpSessionEvent event) {
        (…)
        }
```

```
        /** Número total de sesiones creadas. */

        public int getTotalSessionCount() {
        (…)
        }

        /** Número de sesiones cocurrentes en memoria. */

        public int getCurrentSessionCount() {
        (…)
        }

        /**
        * El mayor número de sesiones que haya habido en algun momento en memoria.
        */

        public int getMaxSessionCount() {
        (…)
        }

        /**
        * Guardamos en el ServletContext los datos obtenidos
        * para que tanto desde cualquier servlet o JSP tenga acceso
        * a la cuenta de sesiones
        */

        private synchronized void storeInServletContext(HttpSessionEvent event) {
        (…)
        }

        private void logSessionCounter(){
        (…)
        }
}
```

Como vemos nuestra clase implementará la interfície **javax.servlet.HttpSessionListener,** esto supone implementar los métodos de dicha interfície, los métodos són los que se reseñan en **rojo**.

- `public synchronized void sessionCreated(HttpSessionEvent event)` cada vez que el container del ***Web Server*** detecte que hay un evento relacionado con una ***HttpSession***, en el caso de ser la creación de una sesión invocará a este método.
- `public synchronized void sessionDestroyed(HttpSessionEvent event)` en el caso contrario, que se detecte un evento de este tipo, pero sea por la destrucción de una sesión se llamará a este método.

Luego dentro de nuestra clase podemos añadir una serie de métodos que nos sirvan a nosotros, por si queremos otorgarle alguna funcionalidad extra , los métodos añadidos són los que se reseñan en **verde**.

- `public int getTotalSessionCount()` con este método obtendremos el número total de sesiones creadas.
- `public int getCurrentSessionCount()` con este método obtendremos el número total de sesiones concurrentes en memória.

- `public int getMaxSessionCount()` con este método obtendremos el número máximo de sesiones que haya habido en memória concurrentemente.
- `private synchronized void storeInServletContext(HttpSessionEvent event)` con este método guardaremos en el ***ServletContext*** el contador de sesiones, por si en algu momento lo necesitamos desde cualquier servlet o JSP.
- `private void logSessionCounter()` este método nos servirá para guardar en un fichero de log, para su psterior consulta, los datos que se van registrando en cada momento.

5)

```
<servlet>
        <servlet-name>HsqlBootstrap</servlet-name>
        <display-name>HsqlBootstrap</display-name>
        <servlet-class>
                common.sp.spshop.web.servlets.HsqlBootstrap
        </servlet-class>
        <load-on-startup>2</load-on-startup>
</servlet>
```

A continuación incluimos la definición de un servlet que se cargará en el arranque del ***Web Server***, esto quiere decir que lo que implementemos en el código de la clase del servlet será posteriormente utilizado por nuestra aplicación.

Como vemos en la definición encontramos una serie de entradas como el nombre del servlet ***HsqlBootstrap***, el mismo nombre utilizaremos por si quiere ser posteriormente consultado desde cualquier JSP o servlet (***display-name***), luego viene la definición de la clase que contendrá la implementación del servlet y finalmente, tenemos el parámetro ***load-on-startup*** que es un parámetro muy útil en el caso de que en el arranque tengamos varios servlets y sea importante el orden en el que sean cargados, ya que los resultados que carga uno pueden ser posteriormente necesitados para la carga del siguiente.

La clase en la que tenemos la implementación del servlet es **common.sp.spshop.web.servlets.HsqlBootstrap,** a continuación veremos el código y comentaremos los aspectos más destacados :

```
package common.sp.spshop.web.servlets;

import (…)

/**
 * Servlet que crea la base de datos
 */
public class HsqlBootstrap extends HttpServlet implements Servlet{
(…)

        public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
                // inicializacion hsqldb
                try {
                        initHsql();
                } catch (Exception e) {
                        throw new ServletException("Imposible inicializar HSQLDB");
                }
```

```
	}

	/**
	* @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
	*/
	public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
		// inicializacion hsqldb
		initHsql();
	}

	public final void init(ServletConfig config) throws ServletException {
		super.init(config);

		// inicializacion hsqldb
		initHsql();
	}

	/**
	 *
	 */
	private void initHsql() {
		Connection conn = null;
		Statement st = null;

		try {
			Context ctx = new InitialContext();

			String dsname=ConfigurationReader.getString(DATASOURCE_NAME_CFG_PROPERTY);
			DataSource ds=ServiceLocator.getInstance().getDataSource(dsname);

			//si existe un DataSource previo eliminamos las tablas existentes
			if (ds != null) {
				conn = ds.getConnection();

				int rc;
				st = conn.createStatement();

				try {
					rc = st.executeUpdate("DROP TABLE USUARIO");
				} catch (SQLException e1) {
				(…)
				}

				//creación de las tablas y relaciones existentes entre ellas
				rc = st.executeUpdate("CREATE TABLE USUARIO (USRID INTEGER NOT NULL
IDENTITY PRIMARY KEY,  USRLOGIN CHAR(10) NOT NULL,  USRPASSWORD CHAR(10) NOT NULL,  USRNOMBRE
CHAR(20) NOT NULL,  USRAPELLIDO1 CHAR(30) NOT NULL,  USRAPELLIDO2 CHAR(30) NULL,  USRDIRECCION
VARCHAR(100) NOT NULL,  USRPOBLACION CHAR(50) NOT NULL,  USRCODPOSTAL CHAR(5) NOT NULL,
USRPROVINCIA CHAR(50) NULL,  USRPAIS CHAR(40) NOT NULL,  USRTELEFONO CHAR(20) NOT NULL,  USRMAIL
CHAR(20) NOT NULL )");
				(…)

				st.close();
				conn.close();

				logger.logInfo("Base de datos creada con éxito");
			}

		} catch (NamingException e) {
			logger.logError("Error al crear base de datos",e);
		} catch (SQLException e) {
			logger.logError("Error al crear base de datos",e);
		} finally {

			try {
				if (st != null)
					st.close();
				if (conn != null)
					conn.close();
```

```
                    } catch (SQLException e) {
                    }

            }
        }
}
```

Como podemos ver esta clase es algo mas pesada, en esta clase implementé la creación de la Base de Datos, por lo tanto, y como bien se puede intuir, la Base de Datos durará mientras este arrancado el ***Web Server***.

Se podría haber creado la base de datos permanente, con la definición del datasource a nivel de descriptor de despliegue y posteriormente crear una base de datos con un cliente, para que nuestra Base de Datos fuese permanente y se pudiese gestionar, pero como no era uno de los objetivos del proyecto el tema de la Base de Datos (gestión y administración de la misma), decidí crear algo más ligero como lo implementado.

Suficiente para que cumpliese su misión y tuviesemos una Base de Datos, que recuerdo, mientras el ***Web Server*** está arrancado cumple la misma misión que una más compleja, y la interactuación que tenemos con ella desde el código de nuestra aplicación sería la misma que si hubiesemos creado una Base de Datos 'de verdad'.

Una vez realizado el comentario anterior pasaremos a comentar el código en si.

Podemos ver como nuestra clase:

- Extiende de la clase ***javax.servlet.http.HttpServlet***.
- Implementa la interfície ***javax.servlet.Servlet*** .

Al ser un servlet ha de implementar siempre la clase comentada anteriormente, para implementar los métodos de la misma hay que hacer que nuestra clase ***servlet*** extienda o bien de un **javax.servlet.GenericServlet** o bien de un ***javax.servlet.http.HttpServlet***, de esta forma nuestro servlet sólo podrá recibir y responder a peticiones de ***Web clients*** que sigan el protocolo ***HTTP***, los métodos que se han de implementar són los remarcados en **rojo**.

- `public void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException` este método se invoca cuando se realiza una petición del tipo **HTTP GET.**

.

- `public void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException` este método se invoca cuando se realiza una petición del tipo **HTTP POST**.

- `public final void init(ServletConfig config) throws ServletException` este método es invocado por el container del ***Web Server*** cuando detecta que el ***servlet*** definido ha de entrar en servicio.

Como vemos, la finalidad de este servlet es la creación de la Base de Datos en el arranque, por lo tanto los 3 métodos que estamos obligados ha implementar realizan la misma acción, que no es otra que invocar al método definido por mi, que contiene todo el negocio de la creación de la Base de Datos.

A continuación pasaré a comentar los datos mas relevantes (marcados en **verde**) dentro de las acciones que se realizan dentro de este método `private void initHsql()`.

Primero nos encontramos con la definición del datasource, obtenemos el nombre del mismo (un simple objeto String) y obtenemos una referencia al mismo (… 1 …).

Despues mediante la referéncia que hemos obtenido del datasource obtenemos una conexión hacia la Base de Datos (… 2 …), de esta conexión obtenida crearemos un *Statement* mediante el cual ya tendremos conexión directa para realiza nuestras peticiones (… 3 …).

Como podemos observar (… 4 …) la forma de invocar y de lanzar nuestras *sentencias SQL* es de la forma que se muestra.

Una vez acabadas todas las acciones es muy importante que cerremos (… 5 …) tanto el *Statement* como la *Conexión*.

Finalmente, hay que asegurarse que el cierre se ha realizado, para ello completamos la acción contenida en el *try…catch* (… 6 …)

```
… 1 …
String dsname=ConfigurationReader.getString(DATASOURCE_NAME_CFG_PROPERTY);
                        DataSource ds=ServiceLocator.getInstance().getDataSource(dsname);
… 2 …
conn = ds.getConnection();
… 3 …
int rc;
st = conn.createStatement();
… 4 …
rc = st.executeUpdate(SENTENCIA SQL)
… 5 …
st.close();
conn.close();
… 6 …
try {
        if (st != null)
                st.close();
        if (conn != null)
                conn.close();
} catch (SQLException e) {}
```

6)

```
        <servlet>
                <servlet-name>action</servlet-name>
                <display-name>ActionServlet</display-name>
                <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
                <init-param>
```

```
                <param-name>config</param-name>
                <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>
```

A continuación tenemos la definición del ***servlet*** que posteriormente servirá nuestras peticiones durante el transcurso de nuestro aplicativo (***Front Controller***).

Al haber utilizado el *framework* de desarrollo *Struts*, la configuración de este ***ActionServlet*** en el arranque de nuestro Web Server es totalmente obligatorio.

Las entradas que nos encontramos són las que hemos venido viendo hasta ahora, un nombre **action**, el mismo nombre por si posteriormente queremos obtener una referencia (***display-name***). Luego la clase que implementará nuestro ***servlet***, que no es otra que **org.apache.struts.action.ActionServlet**.

Finalmente, nos encontramos que en este caso, tenemos un parámetro de entrada para este ***servlet***, este parámetro no es otro que el descriptor de *Struts* -> ***struts-config.xml***, nuestro servlet sacará, durante la ejecución del aplicativo, todo lo que necesita precisamente de este fichero, posteriormente a la explicación del descriptor de despliegue del ***Web Server*** que estamos realizando entraremos a profundizar en el contenido de dicho fichero.

7)

```
<servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

El ***servlet*** anterior necesita que le indiquemos a que peticiones tiene que 'hacer caso', tal y como explicamos anteriormente para el uso de ***Filtros***, la forma en la que el Web Server resuelve este ***url-pattern*** es :

http://host:port + ContextPath

8)

```
<session-config>
        <session-timeout>2</session-timeout>
</session-config>
```

La siguiente entrada que nos encontramos nos define un atributo para la sesión de nuestra aplicación web.

Este atributo, como su nombre indica (***session-timeout***), nos define los minutos a partir de los cuales, debido a falta de actividad en el ***Web Server***, la sesión finalizará.

9)

```
<welcome-file-list>
        <welcome-file>index.html</welcome-file>
```

```
</welcome-file-list>
```

En esta entrada nos encontramos un ‘fichero de bienvenida’, en mi caso sólo tengo una entrada, pero aquí podría haber una lista de ‘ficheros de bienvenida’, esta ‘lista’ de ficheros no tiene otra misión que servir una entrada en caso que la URL responda a un directorio, de esta forma el ***Web Server*** recurre a nuestra lista y la recorre secuencialmente hasta que encuentra un fichero que responda a la ruta especificada por la URL.

10)

```
<error-page>
        <error-code>404</error-code>
        <location>/homeJspError.jsp</location>
</error-page>
```

Esta entrada es muy útil ya que nos establece un mapeo entre un código de error de los conocidos y un fichero que responda ha dicho código de error.

Esto es muy práctico ya que evitará, de una forma muy sencilla, las tediosas pantallas de error servidas por defecto por los navegadores y substituirlas por presentaciones creadas por nostros.

En mi caso sólo he realizado esto para el conocidisimo código de error ***404***.

11)

```
<taglib>
        <taglib-uri>/tags/struts-bean</taglib-uri>
        <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<taglib>
        <taglib-uri>/tags/struts-html</taglib-uri>
        <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
<taglib>
        <taglib-uri>/tags/struts-logic</taglib-uri>
        <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
<taglib>
        <taglib-uri>/tags/struts-nested</taglib-uri>
        <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
</taglib>
<taglib>
        <taglib-uri>/tags/tiles</taglib-uri>
        <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>
```

Finalmente, nos encontramos con la carga de todos los ***tld***, posteriormente necesarios en la creación de las *JSP* en la capa de presentación de nuestro aplicativo.

La carga de estos ***taglibs*** es totalmente necesaria ya que en la creación de nuestras JSP el uso de los ***tags*** definidos en las mismas es imprescindible.

3.4.3. struts-config.xml :

Tal y como comentamos en el apartado anterior, a continuación pasaremos a desglosar y explicar en detalle otro de los ficheros importantes dentro de nuestro aplicativo, ya que en este fichero se encuentra toda la información que necesita el framework ***Struts*** para el manejo de peticiones y respuesta a las mismas.

El código que se encuentra en este fichero lo mostraremos a continuación para su posterior descripción, entrada por entrada, así como hicimos con el descriptor de despliegue del ***Web Server***.

El código es el siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

        <!-- ========== Data Source Configuration =============================== -->

        <!-- ========== Form Bean Definitions =================================== -->

        <form-beans>
                <form-bean name="mostrarCatalogo"
type="common.sp.spshop.web.forms.MostrarCatalogoForm"></form-bean>
                <form-bean name="datosRegistro"
type="common.sp.spshop.web.forms.DatosRegistroForm"></form-bean>
                <form-bean name="datosLogin"
type="common.sp.spshop.web.forms.DatosLoginForm"></form-bean>
                <form-bean name="datosLogout"
type="common.sp.spshop.web.forms.DatosLogoutForm"></form-bean>
                <form-bean name="mostrarProducto"
type="common.sp.spshop.web.forms.MostrarProductoForm"></form-bean>
                <form-bean name="anadirProducto"
type="common.sp.spshop.web.forms.AnadirProductoForms"></form-bean>
                <form-bean name="datosPedido"
type="common.sp.spshop.web.forms.DatosPedidoForm"></form-bean>
                <form-bean name="borrarLineaCesta"
type="common.sp.spshop.web.forms.BorrarLineaCestaForm"></form-bean>
                <form-bean name="cambiarLineaCesta"
type="common.sp.spshop.web.forms.CambiarLineaCestaForm"></form-bean>
                <form-bean name="filtrarPor"
type="common.sp.spshop.web.forms.FiltrarPorForm"></form-bean>
                <form-bean name="ordenarPor"
type="common.sp.spshop.web.forms.OrdenarPorForm"></form-bean>
                <form-bean name="paginacionCatalogo"
type="common.sp.spshop.web.forms.PaginacionCatalogoForm"></form-bean>
        </form-beans>


        <!-- ========== Global Forward Definitions ============================== -->
        <global-forwards>
                <forward name="error" path="/homeError.jsp"></forward>
        </global-forwards>

        <!-- ========== Action Mapping Definitions ============================== -->
        <action-mappings>

                <action path="/mostrarCategoria"
type="common.sp.spshop.web.actions.MostrarCategoriaAction" name="mostrarCatalogo" scope="request"
validate="false">
```

```
                <forward name="ok" path="/homeCatalogo.jsp"/>
            </action>

            <action path="/verCesta" type="common.sp.spshop.web.actions.VerCestaAction">
                <forward name="ok" path="/homeVerCesta.jsp"/>
            </action>

            <action path="/verPedidos" type="common.sp.spshop.web.actions.VerPedidosAction">
                <forward name="ok" path="/homeVerPedidos.jsp" />
                <forward name="login" path="/homeLogin.jsp"/>
            </action>

            <action path="/paginaModificacionRegistro"
type="common.sp.spshop.web.actions.PaginaModificacionRegistroAction">
                <forward name="login" path="/homeLogin.jsp"/>
                <forward name="ok" path="/homeModificacionRegistro.jsp"/>
            </action>

            <action path="/export-products-pdf"
type="common.sp.spshop.web.actions.PdfCreatorAction">
                <forward name="ok" path="/homeCatalogo.jsp"/>
            </action>

            <action path="/paginaRegistro" forward="/homeRegistroUsuario.jsp"
name="datosRegistro" validate="false"/>

            <action path="/hacerLogout" type="common.sp.spshop.web.actions.LogoutAction">
                <forward name="volver" path="/mostrarCategoria.do" />
            </action>

            <action path="/registrar" type="common.sp.spshop.web.actions.RegistrarAction"
name="datosRegistro" validate="true" input="volver">
                <forward name="volver" path="/homeRegistroUsuario.jsp" />
                <forward name="ok" path="/confirmacionRegistro.do" redirect="true"/>
            </action>

            <action path="/confirmacionRegistro" forward="/homeConfirmacionRegistro.jsp"/>

            <action path="/modificarRegistro"
type="common.sp.spshop.web.actions.ModificarRegistroAction" name="datosRegistro" validate="true"
input="volver" scope="request">
                <forward name="volver" path="/homeRegistroUsuario.jsp" />
                <forward name="ok" path="/homeConfirmacionRegistro.jsp" />
            </action>

            <action path="/hacerLogin" type="common.sp.spshop.web.actions.LoginAction"
name="datosLogin" validate="true" input="volver" scope="request">
                <forward name="volver" path="/homeLogin.jsp" />
                <forward name="verPedidos" redirect="true" path="/verPedidos.do" />
                <forward name="modificarRegistro" path="/paginaModificacionRegistro.do"/>
                <forward name="tramitarPedido" path="/paginaTramitarPedido.do"/>
            </action>

            <action path="/mostrarProducto"
type="common.sp.spshop.web.actions.MostrarProductoAction" name="mostrarProducto" validate="true"
scope="request">
                <forward name="ok" path="/homeDetProd.jsp"/>
            </action>

            <action path="/anadirProducto"
type="common.sp.spshop.web.actions.AnadirProductoAction" name="anadirProducto" scope="request"
validate="true">
                <forward name="ok" path="/verCesta.do" redirect="true"/>
            </action>

            <action path="/paginaTramitarPedido"
type="common.sp.spshop.web.actions.PaginaTramitarPedidoAction">
                <forward name="ok" path="/homeConfirmarPedido.jsp"/>
                <forward name="login" path="/homeLogin.jsp"/>
            </action>
```

```
		<action path="/tramitarPedido"
type="common.sp.spshop.web.actions.TramitarPedidoAction" name="datosPedido" scope="request"
validate="true" input="volver">
			<forward name="volver" path="/homeConfirmarPedido.jsp" />
			<forward name="ok" redirect="true" path="/verPedidos.do"/>
		</action>

		<action path="/cambiarLineaCesta"
type="common.sp.spshop.web.actions.CambiarLineaCestaAction" name="cambiarLineaCesta"
input="volver" scope="request" validate="true">
			<forward name="ok" path="/homeVerCesta.jsp"/>
			<forward name="volver" path="/homeVerCesta.jsp"/>
		</action>

		<action path="/borrarLineaCesta"
type="common.sp.spshop.web.actions.BorrarLineaCestaAction" name="borrarLineaCesta"
scope="request" validate="true" input="volver">
			<forward name="ok" path="/homeVerCesta.jsp"/>
			<forward name="volver" path="/homeVerCesta.jsp"/>
		</action>

		<action path="/eliminarCesta"
type="common.sp.spshop.web.actions.EliminarCestaAction" scope="request">
			<forward name="ok" path="/homeVerCesta.jsp"/>
		</action>

		<action path="/filtrarPor" type="common.sp.spshop.web.actions.FiltrarPorAction"
name="filtrarPor" scope="request">
			<forward name="ok" path="/homeCatalogo.jsp"/>
		</action>

		<action path="/ordenarPor" type="common.sp.spshop.web.actions.OrdenarPorAction"
name="ordenarPor" scope="request">
			<forward name="ok" path="/homeCatalogo.jsp"/>
		</action>

		<action path="/paginacionCatalogo"
type="common.sp.spshop.web.actions.PaginacionCatalogoAction" name="paginacionCatalogo"
scope="request">
			<forward name="ok" path="/homeCatalogo.jsp"/>
		</action>

	</action-mappings>
	<!-- controller definition -->
	<controller>
		<!-- The "input" parameter on "action" elements is the name of a
			local or global "forward" rather than a module-relative path -->
		<set-property property="inputForward" value="true" />

	</controller>
	<!--  end controller definition -->
	<!-- message properties definition -->
	<message-resources parameter="common.sp.spshop.web.resources.mensajes" />
	<!-- message properties definition -->
	<!-- plug-in definition -->
	<plug-in className="org.apache.struts.plugins.ModuleConfigVerifier" />

	<!-- end plug-in definition -->
</struts-config>
```

1) Primeramente nos encontramos con los ***<form-beans>*** , aquí están definidos todos aquellos ***Java Beans*** que posteriormente serán utilizados en nuestra ***Aplicación Web***.

La entrada consta de un nombre para cada ***<form-bean>***, así como un atributo ***type*** que no es mas que la clase que implementa el contenido de dicho elemento.

A continuación mostraremos el contenido de una de estas clases, para ver como es su composición.

Como ejemplo utilizaremos uno de los ***<form-bean>*** definido, este es ***common.sp.spshop.web.forms.AnadirProductoForms***:

```
package common.sp.spshop.web.forms;

import (...)

/**
 * @author Eduardo Varga Laguna
 */
public class AnadirProductoForms extends ActionForm {
(...)

	private Long prodId = null;

	/**
	 * Get prodId
	 * @return Long
	 */
	public Long getProdId() {
		return prodId;
	}

	/**
	 * Set prodId
	 * @param <code>Long</code>
	 */
	public void setProdId(Long p) {
		this.prodId = p;
	}

	public void reset(ActionMapping mapping, HttpServletRequest request) {
		prodId = null;

	}

	public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
		ActionErrors errors = new ActionErrors();
		if (prodId == null) {
			logger.logDebug("Error al validar datos de producto a añadir");
			errors.add("prodId", new
ActionMessage("error.catalogo.errorProdId"));
		}

		return errors;
	}
}
```

A comentar sobre la implementación vista arriba, reseñamos en **rojo** lo más significativo:

- Primero observamos que nuestra clase extiende de ***org.apache.struts.action.ActionForm***, este objeto ***ActionForm*** és un JavaBean y nos servirá como contenedor de datos.
- Como podemos observar en nuestra clase tendremos definidos todos aquellos atributos que necesitarán ser inicializados, así como sus ***getters-setter***, para la posterior manipulación de los mismos.

- Tenemos también implementado el método ***reset(…)***, para en caso de necesitarlo, tener una forma rápida de vaciar el objeto.
- Finalmente, hemos implementado el método ***validate(…)***, el uso de este método y la invocación del mismo sigue el siguiente orden:

    - Cuando en nuestro código se ejecuta un ***Action***, lo primero que se hace es obtener el ***ActionForm*** y rellenar los parámetros con los valores introducidos por el usuario.
    - Una vez tenemos el objeto con los datos pertinentes, se invoca a nuestro método y se hace la comprobación pertinente, devolviendose un objeto del tipo ***ActionErrors***, que podrá ser capturado en nuestra JSP, para que muestre los datos que toca pro pantalla.

2)

```
<global-forwards>
        <forward name="error" path="/homeError.jsp"></forward>
</global-forwards>
```

La siguiente entrada que nos encontramos és un ***<global-forwards>*** que como su nombre indica, nos permitirá un redireccionamiento global para cualquiera de nuestras acciones, simplemente pasandole la cádena ***"error"*** que es con la que nuestro contenedor del ***Web Server*** asocia este ***global-forward***.

3) Lo siguiente que nos encontramos es el ***<action-mappings>*** con sus pertinentes ***<actions>*** definidos.

Como podemos observar las entradas de los ***<action>*** pueden seguir uno de los siguientes patrones:

3.1)

```
<action path="/verPedidos" type="common.sp.spshop.web.actions.VerPedidosAction">
        <forward name="ok" path="/homeVerPedidos.jsp" />
        <forward name="login" path="/homeLogin.jsp"/>
</action>
```

En este primer tipo, vemos como tenemos un ***path*****,** que es el nombre que utilizaremos en las JSP's para asociar el ***Submit*** al ***action*** pertinente.

Luego tenemos el ***type***, que es la clase que implementa toda la lógica de negocio de nuestro action.

A continuación vemos dos entradas del tipo ***forward***, esto tiene la misma misión que el ***global-forward*** pero asociado exclusivamente a este ***action***, así, una vez ejecutado nuestro código dentro de nuestra clase podremos, mediante el nombre asociado (***“ok” / ”login”***), hacer la redirección pertinente.

Más adelante mostraremos el contenido de una de estas clases, lo mostraremos en el apartado donde haremos el seguimiento de un ***caso de uso***, desde la capa de presentación hasta la capa de datos, pasando por la capa de negocio, así como el retorno de la respuesta.

3.2)

```
<action path="/confirmacionRegistro" forward="/homeConfirmacionRegistro.jsp"/>
```

En este segundo tipo, vemos que no hay ninguna clase que contenga como se comportará nuestro ***action***, aquí simplemente tenemos, como en el caso anterior, un ***path*** que será el nombre con el que conseguiremos invocar desde nuestra JSP a este action y aquí, a diferencia del caso anterior, lo único que realizamos es un ***forward*** directo a la JSP definida.

Este tipo de entrada es útil para aquellas entradas que no tienen ningun tipo de lógica de negocio, aquellas que no tratan datos, simplemente hacen de puente entre una JSP y otra. Al pasar todas las acciones por nuestro servlet definido (***ActionServlet***) es obligatorio pasar por aquí.

4)

```
<controller>
        <set-property property="inputForward" value="true" />
</controller>
```

La siguiente entrada que nos encontramos es la de ***<controller>,*** esta entrada es utilizada para definir los parámetros del ***controller*** (***org.apache.struts.action.ActionServlet***), en mi caso tomará los parámetros por defecto.

5)

```
<message-resources parameter="common.sp.spshop.web.resources.mensajes" />
```

Luego tenemos definidos los ***<message-resources>***, que es todo aquel contenido estático que posteriormente será utilizado.

En estos ficheros es donde incorporamos todos aquellos strings que se muestran en nuestras JSP, o que utilizamos como mensaje de error, etc…

6)

```
<plug-in className="org.apache.struts.plugins.ModuleConfigVerifier" />
```

Finalmente, vemos que se pueden definir ***plug-ins*** para nuestra aplicación,

3.4.4. Explicación de un Caso de Uso:

3.4.4.1. Introducción

Para poder ver todo lo explicado teoricamente en los apartados ***Teorico, Diseño e Implementación*** de una forma más clara y entendedora, para ver como interactua todo lo anteriormente expuesto de una forma más sencilla, haremos un completo seguimiento de un caso de uso pasando por todos los estados, capas, etc… que sean necesarios para así, finalmente, poder tener una visión global del problema y la solución utilizada mediante la tecnología ***J2EE*** y el framework de desarrollo ***Struts***.

Cogeremos la acción de ***Añadir un artículo al carro de la compra*** para hacer el seguimiento.

3.4.4.2. Capas dentro del Caso de Uso

3.4.4.2.1. JSP de inicio

Para empezar situaremos la explicación en la JSP de inicio, esta es ***catalogo.jsp***, el código de esta JSP es el siguiente:

(1)

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<td>
<table width="100%" border="0" cellspacing="0" cellpadding="15">
        <tr>
        <% if(request.getAttribute("spshop.cat.ruta") != null){ %>
                <td class="forTexts">
                        <logic:iterate id="cat" name="spshop.cat.ruta">
                                <html:link styleClass="forCategoriaLista"
                                        action="mostrarCategoria" paramId="catId"
                                        paramName="cat" paramProperty="id"  >
                                <bean:write name="cat" property="nombre" /> </html:link>
                        </logic:iterate><br>
                        <br>
                        <logic:notEmpty name="spshop.cat.subcategorias">
```

```
                                <html:img page="/images/header_categorias.gif" width="200"
height="27"/><br>
                                <logic:iterate id="cat" name="spshop.cat.subcategorias">
                                        <html:link
                                                action="mostrarCategoria" paramId="catId"
paramName="cat" paramProperty="id"
                                                styleClass="forCategoria"> <bean:write name="cat"
property="nombre" /> </html:link>
                                        <br>
                                </logic:iterate> <br>
                        </logic:notEmpty>
                <% }%>
                <html:img page="/images/header_productos.gif" width="200" height="27"/><br>
                <table cellpadding=10 cellspacing=0 border=0>
                        <logic:iterate id="prod" name="spshop.cat.productos" offset="0"
length="3">
                                <tr class=forProductoLista>
                                 <bean:define name="prod" property="imagen" id="imagenURL"/>
                                        <td width=50>
                                                <html:img src="<%=imagenURL.toString() %>"
border="0"/>
                                        </td>
                                        <td width=200><html:link styleClass="forProductoLista"
action="mostrarProducto" paramId="prodId" paramName="prod" paramProperty="id"><bean:write
name="prod" property="titulo" /></html:link><br>
                                        <br>
                                        <bean:write name="prod" property="artista" /></td>
                                        <td width=80><bean:write name="prod" property="precio" />
Euros<br>
                                        <br>
```

(2)

```
<html:link styleClass="forProductoLista" action="anadirProducto" paramId="prodId"
paramName="prod" paramProperty="id">
                                                <html:img page="/images/cesta.gif" border="0"/>
                                        </html:link></td>
                                </tr>
                        </logic:iterate>
                </table>
                </td>
        </tr>
</table>
<table width="100%" cellspacing="0" cellpadding="15" border="0">
        <tr>
                <td width="35%"></td>
                <td width="30%" align="left">
                        <html:img page="/images/inicio.gif" border="0" style="{cursor:hand;}"
onclick="inicio();"/>
                        <html:img page="/images/anterior.gif" border="0" style="{cursor:hand;}"
onclick="anteriores();"/>
                        <html:img page="/images/siguiente.gif" border="0" style="{cursor:hand;}"
onclick="siguientes();"/>
                        <html:img page="/images/fin.gif" border="0" style="{cursor:hand;}"
onclick="fin();"/>
                </td>
                <td width="35%"></td>
        </tr>
</table>
</td>
```

(1) Como comentamos en el apartado en que explicamos el fichero de despliegue ***web.xml***, en las JSP es necesario el uso de las ***taglib***, que son la fuente de la que obtendremos todas las etiquetas típicas de las JSP, estos tags los iremos encontrando continuamente en el código. Algunos de estos tags són: `<logic:iterate`, `<html:link`, `<logic:notEmpty`, `<bean:write`

(2) Para el caso de uso que estamos planteando, la línea más significativa es la siguiente:

```
<html:link  styleClass="forProductoLista"  action="anadirProducto"  paramId="prodId"
paramName="prod" paramProperty="id">
```

Aquí podemos ver como tenemos el atributo ***action***, que tendrá correspondencia con alguna de las entradas del ***ActionMapping*** definido en nuestro fichero de configuración ***struts-config.xml***, explicado anteriormente.

3.4.4.2.2. Invocación del Action

Una vez hemos dado al enlace que tenemos en la JSP, nuestra petición será capturada por el ***ActionServlet*** que hemos definido en el fichero ***web.xml.*** Esta petición será seguidamente procesada por el ***RequestProcessor***.

La siguiente acción que se realizará será la creación e introducción de valores en el objeto ***ActionForm*** correspondiente definido en el fichero ***struts-config.xml***

```
<form-bean name="anadirProducto"type="common.sp.spshop.web.forms.AnadirProductoForms">
</form-bean>
```

El código asociado a la clase (***common.sp.spshop.web.forms.AnadirProductoForms***) del ***form-bean*** es el siguiente:

```
package common.sp.spshop.web.forms;

import (...)

/**
 * @author Eduardo Varga Laguna
 */
public class AnadirProductoForms extends ActionForm {
(...)

        private Long prodId = null;


        public Long getProdId() {
                return prodId;
        }


        public void setProdId(Long p) {
                this.prodId = p;
        }

        public void reset(ActionMapping mapping, HttpServletRequest request) {
                prodId = null;

        }

        public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
        (...)
        }
}
```

Una vez este completado el objeto ***ActionForm*** y el método ***validate(...)*** no haya fallado , el ***RequestProcessor*** irá al ***ActionMapping*** definido en el ***struts-config.xml*** y nos linkará con el ***action*** que tenemos definido con el nombre que teniamos en el link de la JSP , la entrada correspondiente a la acción que estamos utilizando de ejemplo es la siguiente:

```
<action path="/anadirProducto" type="common.sp.spshop.web.actions.AnadirProductoAction"
name="anadirProducto" scope="request" validate="true">
	<forward name="ok" path="/verCesta.do" redirect="true"/>
</action>
```

Vemos que la clase que implementa nuestra acción es ***common.sp.spshop.web.actions.AnadirProductoAction*** y que contiene un ***forward*** asociado.

Primero veremos el código que tenemos asociado a esa clase, el código es el siguiente:

```
package common.sp.spshop.web.actions;

import (…)

/**
 * @author Eduardo Varga
 */
(1)
public class AnadirProductoAction extends Action {

(…)

	(2)
	public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
request, HttpServletResponse response) throws Exception {

		(3)
		ActionMessages errors = new ActionMessages();
		ActionForward forward = new ActionForward();

		(4)
		try {

			PedidoDelegate pedidoDelegate = new PedidoDelegate();
			// localizamos la cesta de la compra
			CestaCompra cesta = (CestaCompra)
request.getSession().getAttribute("cesta");
			if (null == cesta) {
				cesta = pedidoDelegate.crearCestaCompra();
				request.getSession().setAttribute("cesta", cesta);
			}
			// recuperamos el producto que tenemos que añadir

			AnadirProductoForms apForm = (AnadirProductoForms) form;
			CatalogoDelegate catalogoDelegate = new CatalogoDelegate();
			Producto prod =
catalogoDelegate.getProducto(apForm.getProdId().longValue());

			pedidoDelegate.anadirLinea(cesta, prod, 1);

		} catch (Exception e) {
			logger.logError("Error al añadir producto.",e);
			(5)
			errors.add("error", new ActionMessage("error.comun.error"));
		}
```

```
                if (!errors.isEmpty()) {
                        saveErrors(request, errors);
                        forward = mapping.findForward("error");
                } else {
                        forward = mapping.findForward("ok");
                }

                return (forward);
        }
}
```

Sobre esta clase hay un monton de cosas interesantes ha comentar, todos ellos resaltados en **rojo** y que pasaremos a comentar seguidamente:

(1) El primer punto destacable es ver que nuestra clase extiende de la clase ***org.apache.struts.action.Action***, Un ***Action*** no deja de ser una conexión entre lo que quiere el usuario cuando realiza la request y la lógica interna de negocio que tiene el aplicativo para llevar a cabo lo que el usuario quiere.

(2) El ***RequestProcessor*** una vez ha localizado la acció y la clase que la implementa se encargá de invocar el método `public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response) throws Exception,`

En este método es donde encontraremos todo lo necesario para llevar a cabo la acción requerida.

(3) Una vez ya dentro del método ***execute(…)*** nos topamos con la declaración de dos objetos ***ActionMessages*** y ***ActionForward*** que serán al finalizar el recorrido del ***action*** muy importantes, posteriormente explicaremos el papel de estos dos objetos en toda nuestra lógica de negocio.

(4) Llegados a este punto del método ***execute(…)*** es donde nos encontramos toda la auténtica lógica de negocio, no entraremos a que hace exactamente este apartado del código, simplemente entraremos a analizar una de sus líneas que es la que nos permitirá continuar con la explicación y el flujo de nuestra acción.

(5) Aquí vemos los objetos anteriormente citados en el punto ***(3)***, el objeto ***ActionMessages*** nos servirá como contenedor de errores, en caso de haberlos, para su posterior tratamiento.

El objeto ***ActionForward*** es precisamente el objeto de retorno que devuelve el método ***execute(…)*** . Es este objeto el que contendra el objeto que luego mapee con los ***forward*** pertinentes, definidos dentro del fichero ***struts-config.xml***.

3.4.4.2.3. Acceso a la capa de negocio

Una vez visto el ***Action*** por dentro entraremos más hacia la capa de negocio de nuestro aplicativo, como hemos visto en el punto ***(4)*** hay una serie de objetos por ahí definidos que son a los que nuestro ***Action*** delega lo que se ha de hacer,

```
PedidoDelegate pedidoDelegate = new PedidoDelegate();
…
pedidoDelegate.anadirLinea(cesta, prod, 1);
```

Como podemos ver el nombre de estos objetos sigue el siguiente patrón ***xxxDelegate***, para continuar entonces, pasaremos a ver que hay declarado y definido en esa clase, el código de ***common.sp.spshop.web.pattern.delegates.PedidoDelegate*** es el siguiente:

```
package common.sp.spshop.web.pattern.delegates;

import (...);


/**
 * @author Eduardo Varga
 *
 */
public class PedidoDelegate extends DefaultBusinessDelegate implements PedidoService{
(...)

	public PedidoDelegate(){
		this._service=new PedidoBO();
	}

	public Pedido confirmarPedido(Usuario usuario, CestaCompra cesta, String dirEnvio, String
locEnvio, String provEnvio, String cpEnvio, String paisEnvio){
		this._service.confirmarPedido(pedido);
	}

	(...)
}
```

Es aquí donde veremos toda la lógica explicada en el apartado ***3.3.2 Construcción de servicios***.

Como vemos nuestra clase :

- Extiende de la clase :
  common.saf.j2ee.common.business.DefaultBusinessDelegate (1)
- Implementa la interfície :
  common.sp.spshop.common.business.interfaces.PedidoService (2)
- Tambien podemos observar como en el constructor de nuestra clase se realiza la creación del servicio a través de la creación de un objeto del tipo

common.sp.spshop.common.business.objects.PedidoBO (3)

A continuación pasaremos a mostrar el código de las clases nombradas en estos 3 puntos y así entenderemos el porque de todo este montaje.

(1)

```
package common.saf.j2ee.common.business;

import java.io.Serializable;

public abstract class DefaultBusinessDelegate implements Serializable{

}
```

El interface **Serializable** proporciona serialización automática mediante la utilización de las herramientas de Java Object Serialization. **Serializable** no declara métodos; actúa como un marcador, diciéndole a las herramientas de Serialización de Objetos que nuestra clase Bean es serializable. Marcar las clases con **Serializable** significa que le estamos diciendo a la Máquina Virtual Java (JVM) que estamos seguros de que nuestra clase funcionará con la serialización por defecto. Aquí tenemos algunos puntos importantes para el trabajo con el interface **Serializable**.

- Las clases que implementan **Serializable** deben tener un constructor **sin argumentos**. Este constructor será llamado cuando un objeto sea "reconstituido" desde un fichero **.ser**.
- No es necesario implementar **Serializable** en nuestra subclase si ya está implementado en una superclase.
- Todos los campos **excepto static y transient** son serializados. Utilizaremos el modificador **transient** para especificar los campos que no queremos serializar, y para especificar las clases que no son serializables.

(2)

```
package common.sp.spshop.common.business.interfaces;

import (...)

/**
 * @author Eduardo Varga
 *
 */
public interface PedidoService extends BusinessInterface{

	public Pedido confirmarPedido(Usuario usuario, CestaCompra cesta, String dirEnvio, String
locEnvio, String provEnvio, String cpEnvio, String paisEnvio);
```

```
(...)
}
```

Esta clase es simplemente la interfície donde definimos los métodos que tendrá nuestro servicio. Como ya sabemos en la interfície sólo está la declaración, nunca la implementación.

Como vemos nuestra interfície extiende de ***common.saf.j2ee.common.business.BusinessInterface,*** esta clase es simplemente:

```
package common.saf.j2ee.common.business;

import java.io.Serializable;

/**
 * @author Eduardo Varga Laguna
 *
 * Todas las interfaces que creemos en la capa de negocio tendrán que extender esta interface
base.
 */
public interface BusinessInterface extends Serializable {
}
```

(3)

```
package common.sp.spshop.common.business.objects;

import (...)

/**
 * @author Eduardo Varga
 *
 */
public class PedidoBO extends DefaultBusinessObject implements PedidoService{

	(...)

	public Pedido confirmarPedido(Usuario usuario, CestaCompra cesta, String dirEnvio, String
locEnvio, String provEnvio, String cpEnvio, String paisEnvio) {
		Pedido pedido=null;
		PedidoDAO pedidoDAO=null;

		pedidoDAO=DAOFactory.getInstance().getPedidoDAO();

	pedido=pedidoDAO.confirmarPedido(usuario,cesta,dirEnvio,locEnvio,provEnvio,cpEnvio,paisEn
vio);
		if(logger.isDebugEnabled())
			logger.logDebug("Confirmado pedido "+pedido.getId());
		return pedido;
	}

	(...)
}
```

Es en esta clase donde se encuentra la implementación de los métodos, desde la implementación de los métodos aquí definidos es donde pasamos a la capa final, la capa de datos.

3.4.4.2.4. Acceso a la capa de datos

Como hemos dicho en el punto anterior, a través de la implementación de uno de los métodos de la clase anterior explicaremos los objetos que nos quedan, de esta forma accederemos a la última capa, la que tiene el contacto directo con la Base de Datos.

El código del método que utilizaremos como ejemplo es el siguiente:

```
public Pedido confirmarPedido(Usuario usuario, CestaCompra cesta, String dirEnvio, String
locEnvio, String provEnvio, String cpEnvio, String paisEnvio) {

        PedidoDAO pedidoDAO=DAOFactory.getInstance().getPedidoDAO();

        Pedido pedido=
        pedidoDAO.confirmarPedido(usuario,cesta,dirEnvio,locEnvio,provEnvio,cpEnvio,paisEnvio);

        (...)
```

Como vemos, aquí aparecen dos objetos nuevos:

- ***common.sp.spshop.database.dao.PedidoDAO (1)***
- ***common.saf.j2ee.database.dao.DAOFactory (2)***

A continuación mostramos el código correspondiente a estas dos clases para proceder a la explicación de las mismas:

(1)

```
package common.sp.spshop.database.dao;

import (...)

/**
 * @author Eduardo Varga
 *
 */
public interface PedidoDAO extends DAO{

        public Pedido confirmarPedido(Usuario usuario, CestaCompra cesta, String dirEnvio, String
locEnvio, String provEnvio, String cpEnvio, String paisEnvio);

        public List listaPedidos(Usuario usuario);

        public void cambiarEstado(Pedido pedido, String estado);
}
```

Tal y como explicamos en el apartado de ***Patrones de Diseño***, para el acceso a la Base de Datos utilizamos el patrón conocido como ***Data Acces Object (DAO)***.

Como podemos observar, nuestra clase és una interfície en la cual declararemos la firma de todos y cadauno de los métodos (marcados en ***verde***) que posteriormente tendrán que ser implementados para el acceso y control de la persistencia en Base de Datos.

Otro dato relevante es ver como nuestra interfície extiende de una clase genérica ***common.saf.j2ee.database.dao.DAO,*** el contenido de dicha clase lo mostramos a continuación:

```
package common.saf.j2ee.database.dao;

import java.io.Serializable;

/**
 * @author Eduardo Varga Laguna
 */
public interface DAO extends Serializable{
}
```

Una vez mas dicha clase, debido a su uso genérico, simplemente extiende de ***java.io.Serializable***. (Explicado anteriormente)

(2)

En este segundo punto nos encontramos con la declaración de la clase que no hará de factoría para los ***DAO's***, el nombre de nuestra clase será ***common.saf.j2ee.database.dao.DAOFactory***, nombre que viene establecido por los ***Patrones de Diseño***.

A continuación mostraremos el código de dicha clase y posteriormente pasaremos a explicar los puntos claves.

El código es el siguiente:

```
package common.saf.j2ee.database.dao;

import (...)

/**
 * Clase DAOFactory abstracta que ha de extender la factoria de DAOs concreta
 */
public abstract class DAOFactory {

	(...)

	//*****************************************************************************
	//PARA CADA DAO NECESARIO, AÑADIR UN METHOD ABSTRACTO QUE RETORNE DICHO DAO
	//*****************************************************************************
	public abstract CatalogoDAO getCatalogoDAO();
	public abstract PedidoDAO getPedidoDAO();
	public abstract UsuarioDAO getUsuarioDAO();
}
```

Tal y como expusimos en el tema de ***Patrones de diseños -> DAO***, en esta clase es donde obtendremos referéncia al ***DAO*** que queramos invocar, por lo tanto, lo más importante es la creación de los métodos ***getXXXDAO()*** que nos permitirán dicha invocación. (Reseñado en ***rojo***).

Finalmente nos encontramos con la implementación en si de la interfície del ***DAO***, es aquí donde se encuentra implementada toda la lógica de negocio de nuestro objeto de acceso a datos. En la implementación de estos métodos es donde llevaremos el control de todos los accesos, consultas, inserciones, borrado, etc... en Base de Datos.

El código de dicha clase ***common.sp.spshop.database.hdbsql.dao.HDBSQLPedidoDAO*** lo mostramos a continuación, pasando posteriormente a comentar lo más destacado que nos podemos encontrar en dicha clase:

```
/**
 *
 */
package common.sp.spshop.database.hdbsql.dao;

import (...)

/**
 * @author Eduardo Varga
 *
 */
public class HDBSQLPedidoDAO implements PedidoDAO {

	/** logger para trazas */
	private static Logger logger = LoggerFactory.getLogger(HDBSQLPedidoDAO.class);

	private Connection getConnection(){
		return ((HDBSQLDAOFactory)DAOFactory.getInstance()).getConnection();
	}


	public Pedido confirmarPedido(Usuario usuario, CestaCompra cesta,   String dirEnvio,
String locEnvio, String provEnvio, String cpEnvio,String paisEnvio) {

			(...)
	}


	public List listaPedidos(Usuario usuario) {

		// insertamos pedido
		StringBuffer query = new StringBuffer();

		query = new StringBuffer();
		query.append("SELECT * FROM PEDIDO WHERE PEDUSRID=");
		query.append(usuario.getId());

		ArrayList lista = new ArrayList();

		Connection conn = null;
		Statement stmt = null;
		ResultSet rs = null;
		try {
			conn=getConnection();
			stmt = conn.createStatement();
			rs = stmt.executeQuery(query.toString());
			while (rs.next()) {
				Pedido pedido = new Pedido();
				pedido.setId(new Integer((int)rs.getLong("PEDID")));
				pedido.setEstado(rs.getString("PEDESTADO"));
				java.sql.Date fecha = rs.getDate("PEDFECHA");
				GregorianCalendar gc = new GregorianCalendar();
				gc.setTime(fecha);
				pedido.setFecha(gc.getGregorianChange());
				lista.add(pedido);
			}
			logger.logDebug("Recuperada lista de pedidos del usuario:
"+usuario.getId()+" ("+lista+" pedidos)");
			return lista;
		} catch (SQLException e) {
			try {
				conn.rollback();
			} catch (SQLException e1) {
				logger.logError("Error al hacer rollback.", e);
```

```
                }
                throw new SystemException("Error en comando registraUsuario().", e);
            } finally {
                try {
                    if (stmt != null)
                        stmt.close();
                    if (conn != null)
                        conn.close();
                } catch (SQLException e) {
                    logger.logError("Error al cerrar recursos JDBC", e);
                    throw new SystemException("Error al cerrar recursos JDBC", e);
                }
            }
        }

        public void cambiarEstado(Pedido pedido, String estado) {
            (...)
        }
}
```

Como vemos, y debido a la importancia de la implementación de esta clase, nos encontramos con bastantes puntos de interes.

Los pasaremos a comentar a continuación, estos puntos són los marcados en ***rojo*** en el código anterior.

- `implements PedidoDAO` como vemos nuestra clase implementará la interfície que creamos anteriormente, esto quiere decir que es aquí donde los métodos serán desarrollados.

- `StringBuffer query = new StringBuffer()` declaración del objeto que nos servirá para la creación de la query.

- `query.append(SENTENCIA SQL)`, la forma de montar la query es con el método ***.append(SENTENCIA SQL)*** de los objetos ***StringBuffer***.

- ```
  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;
  ```
  declaración de los 3 objetos con los que realizaremos la interacción

- ```
  conn=getConnection();
  stmt = conn.createStatement();
  rs = stmt.executeQuery(query.toString());
  ```
  los 3 objetos anteriormente inicializados nos sirven ahora para obtener la conexión, obtener un objeto del tipo ***Statement*** y finalmente mediante el método ***.executeQuery(...)*** de los objetos ***Statement*** obtener el resultado de la consulta en

Base de Datos en un objeto ***ResultSet*** que será con el que iremos obteniendo los resultados.

- ```
  while (rs.next()) {
      Pedido pedido = new Pedido();
      pedido.setId(new Integer((int)rs.getLong("PEDID")));
      pedido.setEstado(rs.getString("PEDESTADO"));
      java.sql.Date fecha = rs.getDate("PEDFECHA");
              ...
  }
  ```

una vez tenemos el objeto ***ResultSet*** nos disponemos a su manipulación, como vemos mediante un ***while*** vamos recorriendo uno por uno los resultados de la ***query*** en Base de Datos. Aquí aparece el concepto explicado en los ***patrones de diseño*** que no es otro que el ***Transfer Object,*** el objeto ***Pedido*** sigue este patrón, ya que como vemos interactua como contenedor en la obtención de los datos del ***ResultSet*** mediante los métodos ***getXXX(NOMBRE ATRIBUTO EN BD).***

- ```
  if (stmt != null)
      stmt.close();
  if (conn != null)
      conn.close();
  ```

Finalmente, nos disponemos ha cerrar los objetos previamente creados, para no dejar nada consumiendo recursos que nos pueda generar problemas en la ***memória virtual*** , quedandose objetos sin cerrar.

3.4.4.2.5. Retorno del resultado

Una vez obtenido el resultado deseado, simplemente queda el retorno de ese resultado hacia la capa de presentación.

El punto en el que tenemos que centrarnos se encuentra dentro de nuestra clase inicial de ***Action***, en dicha clase encontramos las siguientes líneas que són las que nos desvelan el 'fin del trayecto'.

```
if (!errors.isEmpty()) {
        saveErrors(request, errors);
        forward = mapping.findForward("error");
} else {
        forward = mapping.findForward("ok");
}
```

Aquí es donde le daremos el valor deseado al objeto ***ActionForward*** que creamos al inicio de la clase, como vemos le asignamos dos valores posibles ***"ok" / "error"*** dependiendo del éxito o no de toda la operación.

Depende del valor que le introduzcamos el ***RequestProcessor*** luego realizará el link a una u otra posibilidad, dependiendo de la definición que hicimos en el fichero ***struts-config.xml***.

3.4.5. Implementación del modelo de datos

3.4.5.1. Tablas

Las tablas propietarias utilizadas són las siguientes:

3.4.5.1.1 USUARIO

USRID (PK) INTEGER

USRLOGIN CHAR(10)

USRPASSWORD CHAR(10)

USRNOMBRE CHAR(20)

USRAPELLIDO1 CHAR(30)

USRAPELLIDO2 CHAR(30)

USRDIRECCION VARCHAR(100)

USRPOBLACION CHAR(50)

USRCODPOSTAL CHAR(5)

USRPROVINCIA CHAR(50)

USRPAIS CHAR(40)

USRTELEFONO CHAR(20)

USRMAIL CHAR(20)

3.4.5.1.2 PEDIDO

PEDID (PK) INTEGER

PEDUSRID INTEGER

PEDFECHA DATE

PEDESTADO CHAR(10)

PEDFECHACANCEL DATE

PEDDIRECCIONENVIO VARCHAR(100)

PEDPOBLACIONENVIO CHAR(50)

PEDPROVINCIAENVIO CHAR(50)

PEDPAISENVIO CHAR(40)

PEDCODPOSTAL CHAR(5)

3.4.5.1.3 LINEAPEDIDO

LINID (PK) INTEGER

LINPEDIDOID (PK) INTEGER

LINPRODUCTO INTEGER

LINCANTIDAD INTEGER

LINIMPORTE FLOAT

LINIVA FLOAT

LINTOTAL FLOAT

3.4.5.1.4 PRODUCTO

PRODID (PK) INTEGER

PRODTITULO CHAR(60)

PRODARTISTA CHAR(60)

PRODDESCRIPCION VARCHAR(150)

PRODPRECIO FLOAT

PRODDESCUENTO FLOAT

PRODIMAGEN VARCHAR(60)

3.4.5.1.5 CATEGORIA

CATID (PK) INTEGER

CATPADRE INTEGER

CATNOMBRE CHAR(20)

CATDESCRIP VARCHAR(150)

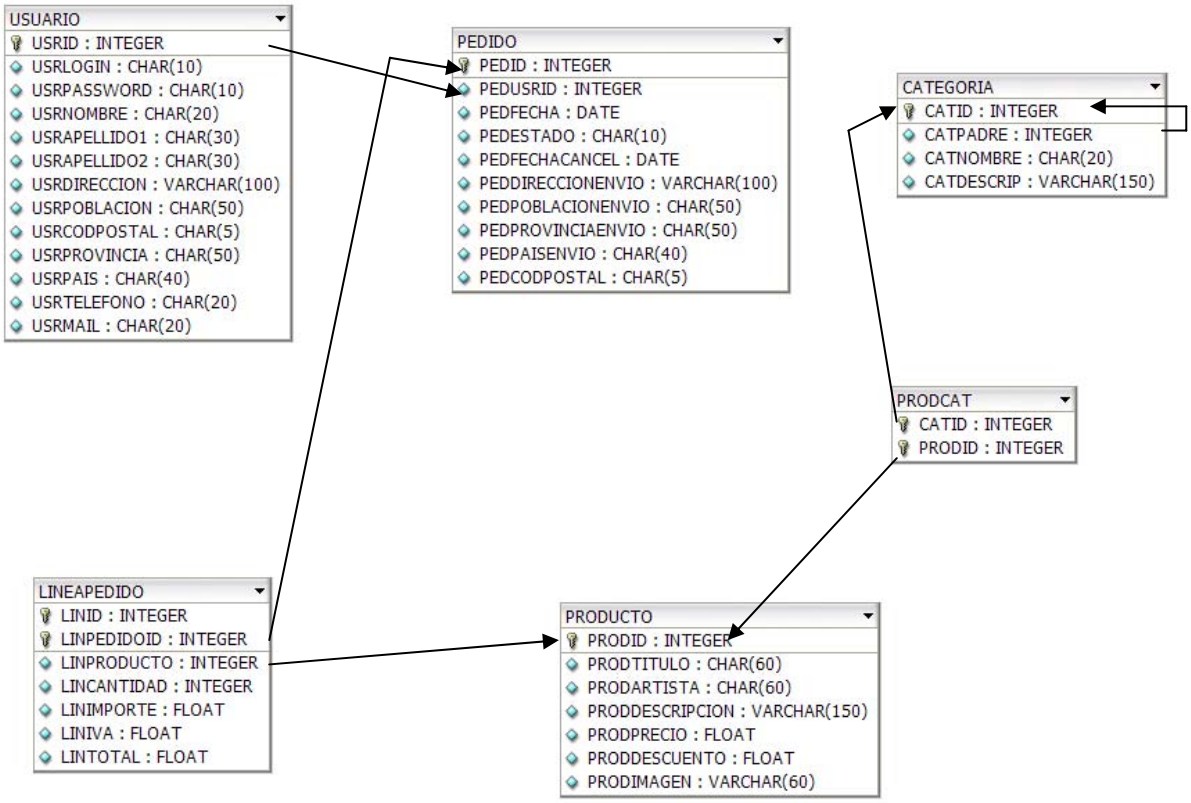
3.4.5.1.6 PRODCAT

CATID (PK) INTEGER

PRODID (PK) INTEGER

3.4.5.2. Representación gráfica de las Tablas de la BD

A continuación mostraremos la representación gráfica de las tablas que componen la BD:



donde (definición de ***Foreign Keys***):

- USUARIO_USRID = PEDIDO_PEDUSRID.
- LINEAPEDIDO_LINPEDIDOID = PEDIDO_PEDID.
- LINEAPEDIDO_LINPRODUCTO = PRODUCTO_PROID.
- PRODCAT_PRODID = PRODUCTO_PRODID.
- PRODCAT_CATID = CATEGORIA_CATID.
- CATEGORIA_CATPADRE = CATEGORIA_CATID.

3.5. Grupo de pruebas

3.5.1. Introducción

Debido a que el aplicativo es a pequeña escala y el rol de desarrollador y probador está encarnado en mí, el conjunto de pruebas al que he sometido el aplicativo no van más allá que a las pruebas básicas de navegación comprobando que todo funcionase.

Debido a que el desarrollo de un aplicativo de este tipo invita a realizarlo de forma modular desarrollando cada caso de uso desde el principio hasta el final, las pruebas se realizaron despues de tener completo cada uno de los casos de uso, desde la capa de presentación hasta la capa de Base de Datos, de esta forma fui localizando los problemas, a través de los logs del propio servidor.

3.5.2. Posibles pruebas

En un proyecto de Ingenieria Informática a gran escala suele haber un equipo que se encarga única y exclusivamente ha realizar estas pruebas.

Estas pruebas se pueden separar y clasificar, ha continuación expondré varias pruebas que se podrian realizar, si el aplicativo hubiese sido de mayor envergadura:

- ***Pruebas de stress:*** Una de las pruebas a la que mayor atención y recursos se le suele dedicar es a las conocidas como ***pruebas de stress***, en estas pruebas no se intenta otra cosa que someter al aplicativo a una *sobrecarga* de trabajo para poder encontrar el rendimiento de todos los componentes llevados al extremo y encontrar posibles ***cuellos de botella.***

- ***Pruebas de navegación:*** Estas, al ser las mas obvias, suelen ser las pruebas que siempre se realizan, són las más 'simples' y a su vez las que se realizan con menos planificación. El problema de estas pruebas es que se le suelen asociar al desarrollador ya que es el que tiene contacto dia a dia con el aplicativo, y a veces, al propio usuario una vez el aplicativo ya ha sido puesto en 'producción' con la consiguiente aparición de errores, ***bugs*** y demás. Lo aconsejable para este tipo de pruebas sería realizar un desglose y un diagrama de flujo en el que se tratasen, sino todos, la mayoria de caminos disponibles dentro del aplicativo.

- ***Pruebas de satisfacción de requerimientos:*** Este tipo de pruebas suelen realizarse durante el desarrollo del aplicativo, en las llamadas ***reuniones de seguimiento***. En dichas pruebas, se intenta ver que el aplicativo va cumpliendo con lo acordado en la ***captación de requermientos*** inicial. Dichas pruebas engloban un gran número de perspectivas debido al gran número y tipología de ***requermientos*** existentes, como ya se vio en el capítulo de ***captación de requerimientos***.

Con estos tres tipos de pruebas se podría establecer un nivel óptimo de satisfacción sobre la calidad del producto final. Existen muchas y múltiples metodologias y herramientas de prueba existentes en el mercado.

4. Recomendaciones

A continuación se relacionan algunas recomendaciones de desarrollo que se han obtenido a partir de la lectura de documentación sobre Struts, forums y la propia experiencia.

4.1. Alternativas al uso de JSP en la capa view

Uno de los sistemas más extendidos en las aplicaciones web para construir los componentes de la view se basa en la tecnología JavaServer Pages.

Struts se basa también en esta tecnología para implementar este tipos de componentes.

A pesar de eso el framework permite sustituir este sistema de presentación o añadir otros mediante extensiones del Struts. A continuación mencionaremos los más relevantes:

4.1.1. stxx

Se trata de una extensión para Struts para soportar tenologias XML y XSL. Permite que las clases que implementan acciones de Struts regresen XML para ser transformado mediante XSL y Anakia.

4.1.2. StrutsCX

Igualmente como el anterior, se trata de una extensión para Struts que permite generar salida en formatos como HTML, XML usando tecnologías XML y XSL estándard.

4.1.3. VelocityStruts

Esta extensión habilita el uso de plantillas de Velocity en la capa de view para Struts.

Las plantillas Velocity permiten diseñar la vista de forma sencilla mediante el uso de un lenguaje específico de definición de plantillas.

El uso de esta extensión no inhabilita el uso de páginas JSP, de manera que es posible mantener conjuntamente estos dos tipos de presentación.

4.1.4. Struts Cocoon

Esta extensión permite integrar Cocoon a Struts. Eso se consigue enlazando la salida de las acciones de Struts con una pipeline XML del Cocoon. Con ello se aporta la potencia y flexibilidad del Cocoon al framework.

4.2. El modelo debe ser simple

Lo más probable es que cuando uno se enfrenta con el diseño de un modelo para una aplicación ya tenga una serie de consideraciones tecnológicas más o menos complejas detrás. La aplicación tendrá que encadenar unas transacciones de host de maneras bien extrañas, o hacer joins de diferentes tablas porque las tablas ya existen y no se pueden tocar, etc.

Es muy importante que estas complejidades no afloren en la interfaz del modelo. Seguro que hay una manera lo más simple posible de expresarlo, probablemente aquélla que haríamos servir para explicarlo a un usuario final o a un sponsor del proyecto, y que tal vez incluso está documentada en un documento de requerimientos.

Si se consigue dar este primer paso, el modelo será más fácilmente reusable, y se alcanzará hacer un controller y una view formada por una serie de clases con muy poco código (por tanto, con pocas oportunidades de fallar)

En resumen, la complejidad tiene que estar escondida dentro del modelo, y éste sólo expone una interfaz lo más sencilla posible sin ningún rastro de detalle de implementación.

4.3. El modelo no tiene que exponer la tecnología utilizada

Los nombres de los objetos tienen que tener significado de negocio, no tecnológico. La implementación concreta del modelo también tiene que quedar escondida. Un Action, cuando llama a un modelo, no tiene porque tener que saber que detrás hay una base de datos. Eso se concreta en que no debe importar ningún package de JDBC. El patrón de diseño de Transfer Object o Value Object es muy útil para tal efecto.

Los métodos del modelo tienen como parámetros, valores de retorno y excepciones lanzadas objetos básicos o del propio modelo, pero no de la tecnología utilizada para implementarlo.

En el caso de las excepciones, el modelo de las nested exceptions o excepciones encadenadas es imprescindible: si un método del modelo se encuentra un problema con la base de datos, crea una excepción de un tipo significativo a nivel del modelo e incluye la excepción original adentro. Así no se pierde la información detallada de la causa del problema, pero tampoco pasa responsabilidad al Action de tener que tratar con objetos tecnológicos.

4.4. El modelo tiene que estar muy probado

El testing siempre es el punto más fácil de recortar cuando las fechas aprietan. En un modelo de desarrollo test-driven, eso no nos lo podemos permitir. Si hay algún aspecto donde focalizar el testing, éste es el modelo. Como es dentro del modelo donde se encuentra escondida la mayor complejidad, es la capa donde más se pueden producir los errores, y donde los errores son más importantes.

Además, el hecho de tener el modelo bien separado del resto hace ideal la utilización de JUnit como herramienta de pruebas. Cada nueva funcionalidad hay que traducirla en unos pocos test cases, que prueben tanto los casos favorables como las condiciones de error.

4.5. El Action no es parte del modelo

Éste es uno de los puntos donde es preciso tener clara la división entre controller y modelo. El modelo tiene que contener toda la lógica de negocio, con unas interfaces claras y significativas bajo el punto de vista de los requerimientos y del usuario.

Es habitual, en un desarrollo iterativo, que para implementar una nueva funcionalidad uno se vea tentado de codificar lógica de negocio dentro del Action. Como el Action se debe hacer de todas

maneras, puede dar pereza añadir un método en una clase del modelo, o añadir una nueva clase al modelo, y se pone dentro del Action.

Eso no es bueno por muchas razones, entre las que lo más importante es que si se debe volver en hacer lo mismo en otro Action se tendrá que repetir el mismo código. Además, uno de los requerimientos que nos lleva a escoger Struts es el de hacer aplicaciones más reusables: el mismo modelo puede servir para otras aplicaciones, o por la misma con otra interfaz.

En resumen, el código del Action idealmente debe ser muy corto, y limitarse a hacer llamadas a un modelo que debe tener una interfaz simple.

Es más fácil seguir esta norma si se tiene una batería de pruebas para el modelo, y cada nueva funcionalidad genera una o diversas nuevas pruebas, aparte del Action y los componentes de view correspondientes.

4.6. La *view* no es parte del modelo

Esta es aún más clara que el anterior, porque son dos de las capas del MVC, pero no está de más también remarcarla. No hay que codificar lógica de negocio dentro de la view, por las mismas razones que antes.

4.7. La *view* no tiene que llamar al modelo

La view se ha de limitar a las tareas de presentación. Su función hay que limitarla a recoger los datos que le ha pasado el Action anterior, presentarlas al usuario, recoger las que entre el usuario por la siguiente petición, posiblemente validarlas sintácticamente (no semánticamente, de eso se encarga el modelo), y finalmente enviar la petición al controller. El controller, y en particular el Action, es quién llama al modelo y establece el resultado final de la petición (ActionForward), a partir del que se invocará el siguiente recurso.

Todo eso se puede hacer, en el caso de las JSPs, con unos pocos tags HTML y unos pocos custom tags.

La manera más fácil de adherirse a esta norma es hacerlo siguiendo lo siguiente:

4.7.1. No usar scriptlets en las JSPs

Los scriptlets son la manera más fácil de mezclar lógica con presentación, y son una potencial fuente de problemas porque promueven fácilmente la técnica del cut-and-paste de código en múltiples JSPs, dando lugar a una aplicación más difícil de mantener.

Si se ve que una misma lógica, que realmente pertenece en la presentación, hay que repetir en múltiples JSPs, la manera correcta es crear un custom tag que la implemente, y en la JSP simplemente escribir el tag.

Lo más probable, es que éste tag ya exista entre los muchos que proporciona Struts o la JSTL (JSP Standard Tag Library).

4.7.2. Utilizar taglibs

Las taglib de Struts son una fuente constante de sorpresas agradables. Si se utilizan con toda su potencia se pueden hacer JSPs realmente compactos, fáciles de entender y de mantener.

La recomendación es intentar entenderlas en toda su extensión, y con cada nuevo problema o requerimiento que uno se encuentre buscar primero en su documentación, o por Internet. Probablemente alguien ya lo ha hecho antes y se ha incorporado al framework.

4.7.3. No enlazar directamente JSPs entre ellas

La función del controller en el MVC es la de tratar todas las peticiones que llegan a la aplicación y, una vez efectuada la operación, redireccionar a la view que corresponda. Si desde una JSP (por ejemplo, desde un menú) ponemos un enlace directo a otra JSP, esta petición no pasará nunca por el controller (l'ActionServlet).

Por ello, cuando lo que se quiere hacer es enlazar de una JSP a otra lo que se hace es pasarlo por una action especial que ya vé con Struts, el ActionForward, que simplemente hace un forward a la JSP que se le pasa como parámetro.

Puede que haya aplicaciones Struts genéricas en que el hecho de no pasar por el controller entre dos JSPs no suponga problema alguno. En el caso de una aplicación Intranet-Struts sí, porque se

pierde el proceso que hace el nuestro RequestProcessor (por ejemplo, convertir la cookie de idioma en locale, o comprobar los roles del usuario)

4.7.4. Pensar y leer mucho, codificar poco

La idea que hay detrás de esta norma es muy sencilla: seguro que alguien ya lo ha intentado antes. Struts consta de unas 300 clases muy arquitecturadas, y para conseguir la mayoría de tareas es necesaria una cantidad de código sorprendentemente pequeña. Al ser código abierto, la evolución de Struts ha venido de la práctica de muchísimos usuarios que han visto resueltas de manera elegante sus requerimientos.

Por ello, ante la duda sobre como afrontar un problema de desarrollo de una aplicación Struts, lo mejor es leerse bien el API (tanto la de Struts como la J2EE: Servlet y JSP), probablemente con un par de líneas de código se resuelve. La web de Struts y los forums que allí se relacionan también son muy útiles si no se encuentra directamente la respuesta en la documentación. A veces también está bien dar una ojeada a los fuentes de Struts, allí también veremos que las clases de Struts son sorprendentemente pequeñas...

Si el modelo es simple, también será más fácil seguir esta norma.

4.7.5. Preferir ActionForms con request scope que con session scope

En un entorno de alta disponibilidad, cuando algún dato se guarda a nivel de sesión (HttpSession), este dato se almacena en la base de datos de persistencia de las sesiones. Dependiendo de la longitud del dato y de su cantidad, eso puede acabar provocando una ralentización de la aplicación sólo por el hecho de tener que persistir la sesión.

Por tanto es preciso evitar, en la medida del posible, ir acumulando datos en la sesión.

Si finalmente se guardan datos en la sesión, es preciso recordar borrarlas cuando ya no se necesitan.

En el caso de Struts, las ActionForms con scope de sesión tienen sentido cuando hay una serie de interacciones con el usuario previas a la ejecución de una operación de negocio (diálogos tipos wizard). En este caso son indicadas sin duda.

Intanet Struts añade en la sesión un pequeño objeto AisSessionBean para mantenimiento de datos de sessio de infraestructura.

4.7.6. Refactorizar a menudo

Es impresionante la cantidad de problemas que uno se puede evitar dedicando un tiempo a rehacer una jerarquía de clases en cuentas de introducir con calzador un método a la clase que se tiene más a mano. Merece la pena aprovechar la potencia del WSAD para cambiar el nombre de una clase o de un método, o para subir un método a la clase padre, o para cambiar de package una clase, etc. Eso combinado con una buena batería de pruebas permite aventurarse mucho más en la tarea de refactoritzar: si los tests siguen pasando, quiere decir que no hemos roto nada.

4.7.7. El buen código se comenta solo

No hay nada más desagradable que heredar un código lleno de comentarios que no ténen nada que ver con lo que hace realmente. Desgraciadamente eso es lo más normal. La mejor manera de evitarlo es poniendo el mínimo de comentarios posibles, que se sustituyen por un diseño entendedor y un código claro: métodos con pocas líneas y con nombres bien escogidos que indiquen lo que hacen. Vale más dedicar tiempo en pensar en una interfaz lo más sencilla y aclaratoria posible que escribiendo comentarios que un día u otro seguro de que quedarán obsoletos.

Tenemos la suerte que el Java es un lenguaje fácil de leer, siempre y cuando no se tenga que leer s_mft.getUpC1(). Si para que el código sea legible el nombre de una variable o método debe ser más largo, pues se hace más largo. Las herramientas de asistencia del WSAD (Ctrl + espacio) ya ahorran de escribir nombres largos, merece la pena aprovecharlo.

Sólo en el caso de que el comentario aporte alguna información útil es adecuado posarlo.

En esta recomendación no está incluidos los comentarios Javadoc que documentan la interfaz pública de una librería o framework. En este caso, sí que son necesarios y entonces es preciso ser muy estricto en modificar el Javadoc, si es preciso, cada vez que se toca algún método. Por su parte, tampoco merece la pena escribir un Javadoc lleno de descripciones triviales que sólo repiten las palabras de que consta el nombre del método.

5. Conclusiones

5.1. Posibles mejoras en el producto final

Debido a que el proyecto pretendía mostrar de una forma ‘sencilla’ y clara lo que sería en la realidad el desarrollo de un aplicativo en ***J2EE*** con el framework de desarrollo ***Struts***, hay muchas cosas que se han implementado de una forma sencilla y ligera para que no aportasen un trabajo añadido al cómputo global del proyecto.

A continuación reflejaré una série de mejoras que podrían introducirse para que el proyecto cogiese mas forma y cuerpo:

- Como primera mejora, y ya comentada en unos de los puntos anteriores de esta memoria se encuentra el tema de la Base de Datos. Como ya expliqué, la Base de Datos en la actualidad es creada mediante un servlet en el arranque del ***Web Server*** que no permité la perdurabilidad de los datos mas allá de la ***vida*** que tenga la sesión del ***Web Server*** en cuestion.

  De esta forma se puede ver claramente una de las posibles mejoras, que no sería otra que desarrollar esta parte del proyecto de una forma mas amplía, generando en una máquina aparte la Base de Datos, con sus herramientas de gestión y administración y el posterior tratamiento a través del código, configurando en los descriptores de despliegue y configuración todo el tema de la ***API JBDC*** utilizada.

- Otra posible mejora hubiese sido desarrollar de una forma eficaz y eficiente el conjunto de pruebas, haciendose eco de todas aquellas herramientas que existen en el mercado para este fin y desarrollando esta parte del proyecto como un módulo aparte del mismo

- En el tema de la presentación se podría también haber desarrollado un complejo sistema de navegación que siguiese un patrón mas actual y no el simple hecho de pantalla -> acción -> pantalla. Este trabajo podría correr aparte por cuenta de un buen ***Diseñador Web*** que se encargase de un profundo estudio para potenciar la parte estética y de *marketing*. No hay que olvidar que este proyecto no dejaría de ser un aplicativo para vender libros, con el consiguiente peso comercial que tendría.

Estas són algunas de las mejoras que se me ocurren, pero probablemente debido a la

complejidad del proyecto se podría hacer hincapié en cada uno de los módulos y apartados del mismo, especializando cada uno de estos módulos hasta el extremo.

5.2. Valoración personal

Como autor de esta memoria, diseñador, desarrollador y ideólogo de este proyecto, creo que puedo decir que el objetivo con el que se empezó todo esto queda cumplido, por lo menos para las expectativas que yo deposité en él.

La idea no fue en ningún momento crear una aplicación cerrada y especializada en cada uno de los puntos que toca, sino todo lo contrario, de una forma 'ligera' y sencilla conseguir crear en el lector una idea general de cómo arrancar un proyecto de este tipo. Por esta razón se y asumo que si se observa el proyecto con lupa genere en el lector y usuario del aplicativo muchas lagunas, o si mas no, generé cierto interés en profundizar en esos temas 'abiertos' .

Saludos cordiales del autor.

Eduardo Varga Laguna.

6. Referencias

6.1. Libros

- Cavaness, Chuck. Programming Yakarta Struts. O ' Reilly. 2002
- Java Enterprise – in a Nutshell. O'Reilly. 2002
- Professional Jakarta Struts. Goodwill & Hightower. Wrox. 2004

6.2. Enlaces internet

- http://www.oreilly.com/catalog/yakarta/
- http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html
- http://www.geocities.com/txmetsb/req-mgm-2.htm
- http://www.cs.ualberta.ca/~pfiguero/soo/metod/requerimientos.html
- http://www.clikear.com/manuales/uml/modelos.asp
- http://www.javahispano.org/tutorials.item.action?id=28
- http://civil.fe.up.pt/acruz/access/modeloER.htm
- http://yakarta.apache.org/struts/index.html
- http://yakarta.apache.org/struts/userGuide/index.html
- http://struts.sourceforge.net/community/extensions.html
- http://it.cappuccinonet.com/strutscx/index.php
- http://stxx.sourceforge.net/
- http://yakarta.apache.org/velocity/anakia.html
- http://yakarta.apache.org/velocity/tools/struts/
- http://struts.sourceforge.net/struts-cocoon/
- http://logging.apache.org/log4j
- http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html
- http://java.sun.com/blueprints/index.html
- http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf
- http://www.adrformacion.com/cursos/javaser/leccion3/tutorial2.html
- http://e-docs.bea.com/wls/docs92/webapp/web_xml.html#wp1015950
- http://hsqldb.org/

Anexo I. Instalación del entorno de desarrollo

Anexo I.1. Software a utilizar

En la medida que podamos vamos a intentar utilizar herramientas que sean gratuitas, aquellas herramientas que tienen licencia GPL. Las herramientas software que vamos a utilizar son las siguientes:

Nombre	**Versión**	**URL Proveedor**
Java Development Kit	*1.4.2_08*	http://java.sun.com/j2se
Lomboz Eclipse IDE	*1.0.0 (R1 Lomboz for Eclipse 3.1, build 20050526)*	http://forge.objectweb.org/projects/lomboz
Apache Tomcat	*4.1.31*	http://jakarta.apache.org/tomcat

Es necesaria la creación de una carpeta de nombre entorno-desarrollo en la unidad de la máquina donde se vaya a realizar la instalación, que se usará como carpeta base para la instalación de todo el software necesario.

Anexo I.2. Instalación de JDK 1.4

Anexo I.2.1 Proceso de Instalación

Para la instalación de éste software se han de seguir los siguientes pasos:

1) Ejecutar el archivo j2sdk-1_4_2_08-windows-i586-p.exe y aceptar la licencia.

2) Después de la aceptación de la licencia, aparecerá una pantalla en la cual hemos de establecer las referencias de instalación.

- Para cambiar el directorio de instalación por defecto, basta pulsar el botón Change y establecer el valor C:\entorno-desarrollo\j2sdk1.4.2_08\:
- Seleccionar únicamente Internet Explorer como navegador a instalar el plugin y comenzar el proceso de instalación pulsando el botón de Install.

3) Una vez finalizado el proceso de instalación, es necesario añadir una variable de entorno de nombre JAVA_HOME y valor C:\entorno-desarrollo\j2sdk1.4.2_08.

4) Por último, es necesario modificar la variable de entorno PATH para añadir el path C:\entorno-desarrollo\j2sdk1.4.2_08\jre\bin.

Anexo I.2.2 Verificación de la Instalación

Para la verificar la correcta instalación del JDK, abrir una ventana de comandos, y tras ejecutar la instrucción ***java –version***.

Anexo I.3. Instalación de Lomboz Eclipse IDE

Anexo I.3.1 Introducción

Lomboz Eclipse IDE es simplemente una distribución de Eclipse IDE que lleva integrada el plugin Lomboz de ObjectWeb para el desarrollo de aplicaciones J2EE.

Anexo I.3.2 Proceso de Instalación

Para la instalación de éste software se han de seguir los siguientes pasos:

1) Extraer el archivo lomboz-eclipse-emf-gef-jem-I20050526.zip, en la carpeta siguiente: C:/entorno-desarrollo.

2) Crear un acceso directo en el escritorio al fichero C:/entorno-desarrollo /eclipse/eclipse.exe para poder ejecutar la herramienta:

Anexo I.3.3 Verificación de la Instalación

Para verificar la correcta instalación de Eclipse, pulsar sobre el acceso directo creado y aceptar la localización del workspace ofrecido por defecto -> Aparecerá la pantalla principal de la herramienta Eclipse.

Anexo I.3.4 Configuración de Eclipse

Antes de comenzar a usar Eclipse, es necesario realizar las siguientes operaciones de configuración:

1) Abrir Eclipse, y seleccionar la opción Window→Preferences. Expandir la opción Java→Build Path, y marcar el checkbox de Folder (asegurarse de que las carpetas de source y output se llaman src y bin, respectivamente). Seleccionar además la opción de JRE_LIB variable como librería JRE a usar.

Anexo I.4. Instalación de Apache Tomcat

Anexo I.4.1 Proceso de Instalación

Para la instalación de éste software se han de seguir los siguientes pasos:

1) Descargar de la página oficial el ejecutable Jakarta-tomcat-5.0.30.exe. Ejecutarlo.

2) Tras pulsar el botón de Aceptar, aparecerá la pantalla de licencia. Tras aceptar la licencia pulsando el botón de ***I Agree***, aparecerá la pantalla de opciones de instalación.

3) Seleccionamos la instalación Normal, y pulsamos el botón de Next. En la siguiente pantalla, introducimos como directorio de instalación el directorio C:\entorno-desarrollo\Tomcat5.0.30 , y pulsamos el botón de Install.

4) Una vez el proceso de instalación haya finalizado, pulsamos el botón de Next.

5) Aceptamos la información de configuración que viene por defecto, y pulsamos el botón Finish para finalizar el proceso de instalación.

Anexo I.4.2. Verificación de la Instalación

Para verificar la correcta instalación de Apache Tomcat, arrancamos Tomcat desde el menú Inicio→Programas→Apache Tomcat 5.0→Start Tomcat

Aparecerá una ventana de DOS que corresponderá a la salida estándar y de errores de Tomcat.

Cuando haya finalizado el proceso de inicialización, abrimos un navegador y tras solicitar la URL http://localhost:8080/index.jsp , veremos la página de inicio de Tomcat.

En el siguiente documento podrá encontrar de una forma clara y entendedora a través de la creación de un sencillo aplicativo el mecanismo para la creación de una aplicación J2EE basada en el Framework de desarrollo Yakarta Struts.
En el mismo partirá desde cero, desde el inicio en la captación de requerimientos, pasando por la etapa de análisis y diseño y la posterior implementación.