DISEÑO E IMPLEMENTACIÓN DE UN FRAMEWORK DE PRESENTACIÓN PARA APLICACIONES J2EE.

Juan Arturo Ortega Rufas.

Ingeniería Informática.

Tutor: Josep Maria Camps i Riba.

Enero 2014.

Juan Arturo Ortega Rufas. PFC. Enero 2014.
Tutor: Josep Maria Camps i Riba.

1. Introducción.

1.1. Punto de partida y objetivos.

El objetivo principal es analizar y desarrollar un framework que haga que el desarrollo de la capa de presentación sea más sencillo y fácil de implementar en el caso de aplicaciones J2EE con un cliente ligero.

Uno de los principios de diseño de la capa de presentación de una aplicación J2EE es separar el control del flujo y las llamadas a la capa de negocio de la presentación. Para conseguirlo las aplicaciones J2EE usan lo que se denomina una arquitectura de tipo Model-2 que usa el patrón Modelo-Vista-Controlador:

- Donde el controlador se implementa con un Servlet que principalmente realiza las funciones de invocar a la lógica de negocio adecuada a partir de la petición del cliente y de seleccionar la siguiente vista según el resultado de la invocación y el estado de la aplicación.
- Los JavaBeans realizan el papel de modelo.
- Las vistas se pueden implementar con páginas JSP, que será el caso de este framework.

Las ventajas del uso de este patrón es que permite una clara separación de roles, permitiendo un tratamiento centralizado de las peticiones, de esta forma las aplicaciones son más flexibles y fáciles de modificar. No obstante existen una serie de tareas repetitivas que todas las aplicaciones deben implementar y que el uso de un framework simplifica. Las ventajas señaladas en la teoría al usar de frameworks:

- Simplifica y estandariza la validación de los parámetros de entrada.
- Desacopla la capa de presentación de la capa de negocio en componentes separados.
- Simplifica la gestión del flujo de navegación de la aplicación y proporciona un punto central de control.
- Permite un nivel muy alto de reutilización.
- Simplifica muchas tareas repetitivas.

Por lo tanto el punto de partida para desarrollar e implementar un framework propio, será estudiar las diferentes alternativas que existen actualmente. Creo que los frameworks escogidos son los más comunes:

- Struts 2.
- Spring MVC versión 3.
- Java Server Faces 2.0

Para realizar el proyecto se han destacado los siguientes objetivos, que también los podemos considerar como tareas o hitos del mismo:

1) Evaluar distintos frameworks de la capa de presentación en la tecnología J2EE, determinando sus fundamentos, su funcionamiento, aplicabilidad, ventajas e inconvenientes.

2) El estudio anterior permitirá profundizar en el conocimiento en la capa de presentación para poder recopilar los requisitos que debe tener un framework propio, teniendo en mente la facilidad de uso.

3) Realizar el diseño del framework, usando UML, describiendo los conceptos implementados con cada componente. La herramienta será Magic Draw 17 de la que disponemos la licencia.

4) Implementar el framework usando componentes basados en Servlets, JSPs, JavaBeans y XML.

5) Realizar un test del framework usando una pequeña aplicación diseñada a tal efecto.

6) Realizar una memoria y la documentación de un proyecto PFC

Antes de empezar a realizar una breve descripción de los distintos frameworks, comentar en esta introducción dos patrones por su importancia para el proyecto, el patrón ServiceToWorker que se usa en el proyecto y el patrón interceptor que aunque no implementado ha influido en el diseño cuando se han realizado las clases que he denominado Filtros.

- Interceptor.
- ServiceToWorker.

1.2. Patrón Interceptor.

Usado por Struts 2. Información obtenida de **Alur, Deepak et Al - Core J2EE Patterns, Best Practices and Design Strategies [PRENTICE HALL](2Ed-2003)**

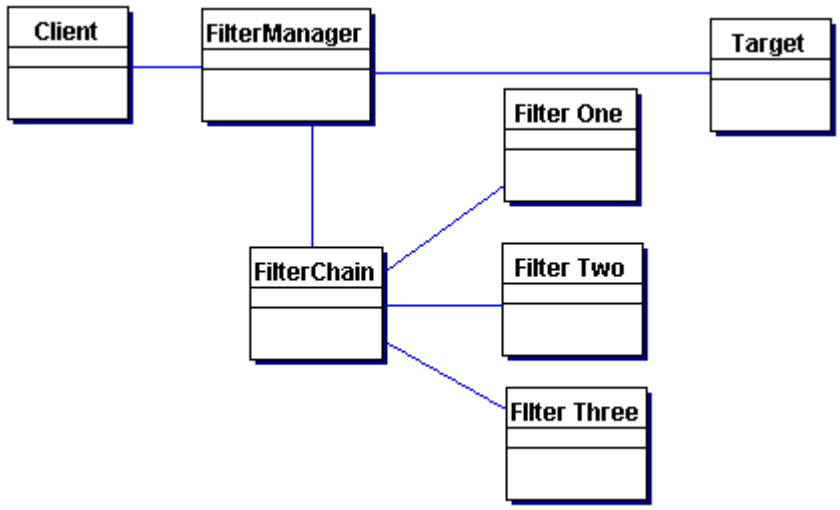
Contexto

La capa de presentación recibe muchos tipos diferentes de peticiones, cada una de las cuales requiere varios tipos de procesamiento, es decir, se requiere un pre-procesamiento y un post-procesamiento de unas peticiones o respuestas de un cliente Web. Se quiere evitar un estilo de programación consistente en una serie de chequeos condicionales con sentencias if/else anidadas. En el caso de que no se cumpla una condición la petición se aborta.

Solución

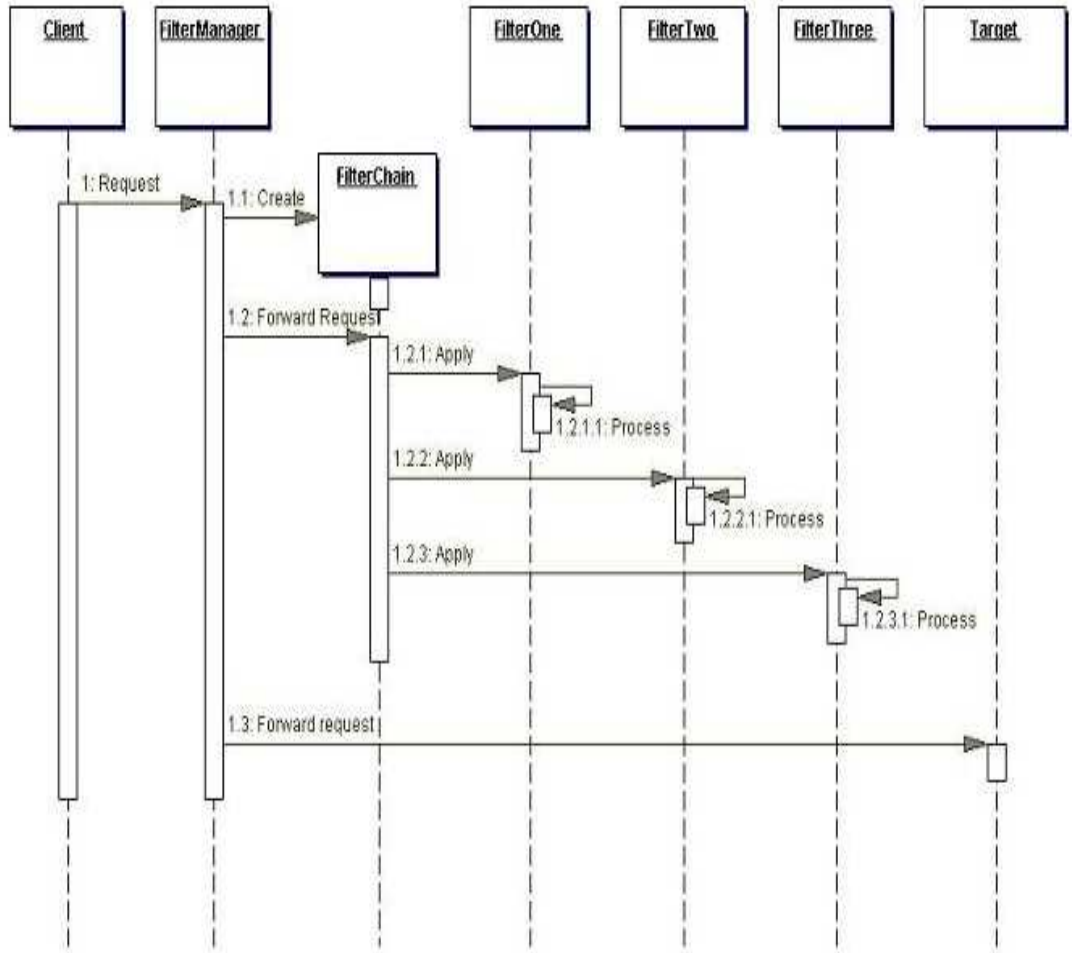
Crear filtros conectables para procesar servicios comunes de una forma estándar sin requerir cambios en el código principal del procesamiento de la petición. Los filtros interceptan las peticiones entrantes y las respuestas salientes, permitiendo un pre y post-procesamiento. Podemos añadir y eliminar estos filtros a discreción, sin necesitar cambios en nuestro código principal existente.

Diagrama de clases.



Responsabilidades.

Diagrama de secuencia del patrón.



Diagramas obtenidos de :http://www.programacion.com/articulo/catalogo_de_patrones_de_diseno_j2ee__i__capa_de_presentacion_240.

Donde:

El FilterManager, intercepta la petición y maneja el procesamiento de filtros. Crea el FilterChain con los filtros adecuados en el orden correcto e inicia el procesamiento. En Struts 2 el papel es adoptado por el punto de entrada el Distpatcher Filter.

El FilterChain es una colección ordenada de filtros independientes. Struts 2 lo implementa con la pila de interceptores. Al igual que los filtros individuales algunas pilas vienen preconfiguradas.

Los FilterOne, FilterTwo,…son los filtros individuales que son mapeados a un objetivo. En Struts 2 la mayoría vienen configurados pero también se pueden definir.

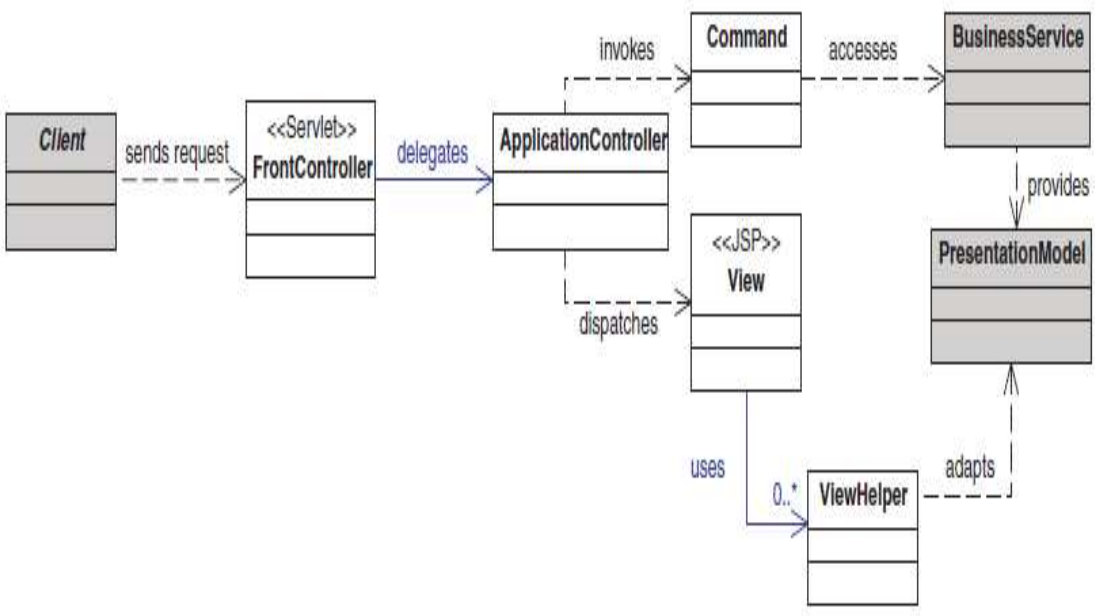
El Target es el recurso que el cliente ha solicitado, Action en Struts 2

1.3. Patrón Service To Worker.

Usado en el diseño de mi framework. Información obtenida *de* ***Alur, Deepak et Al - Core J2EE Patterns, Best Practices and Design Strategies [PRENTICE HALL](2Ed-2003).***

La idea es aplicar un MVC tipo 2 en el diseño del framework. Este patrón centraliza el control y la gestión de las peticiones obteniendo un modelo de presentación antes de devolver el control a la vista. La vista genera una respuesta dinámica basada en el modelo obtenido. Realmente el Service To Worker es una combinación de otros patrones como Front Controller, Application Controller y View Helper.

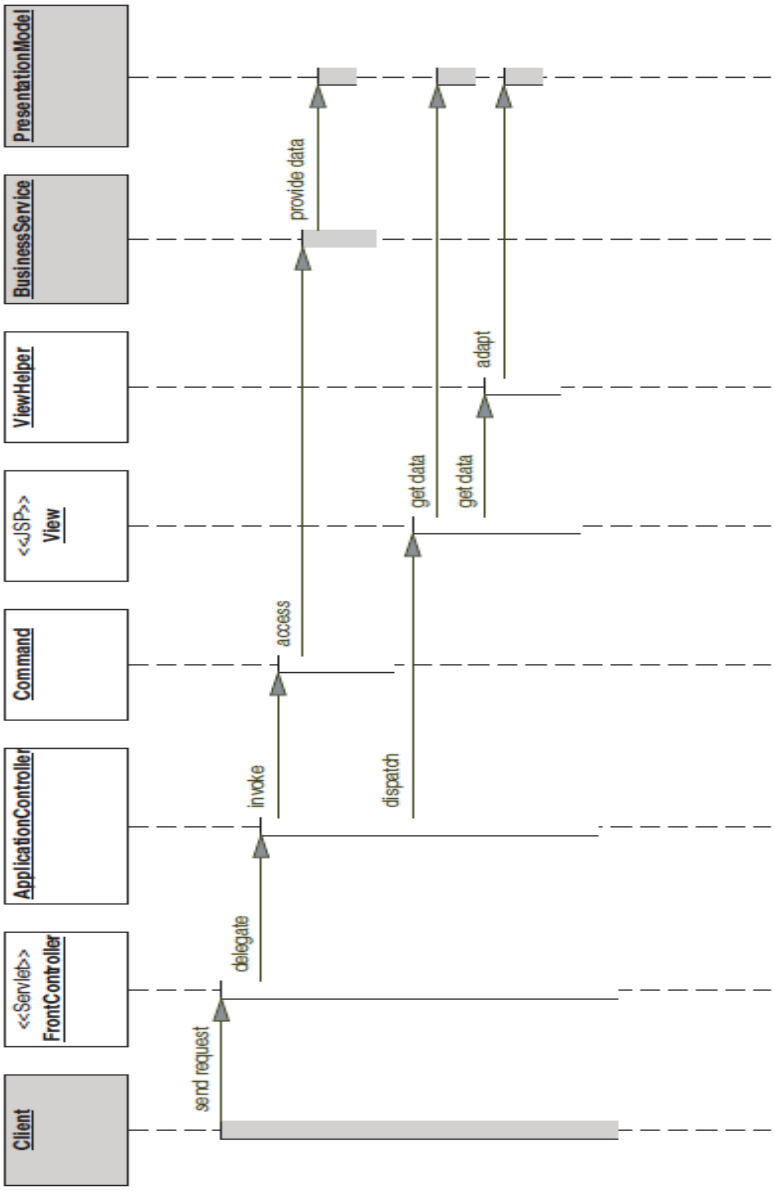
Comentar además que en el framework diseñado en este pfc , el Front Controller usa el patrón Context Object para encapsular el estado de la petición antes de pasarlo al Application Controller. **Diagrama de clases del patrón.**



El Applicattion Controller actúa como un Command Handler, mapeando nombres lógicos al comando (o acción) apropiado.

Participantes y Responsabilidades.

El Front Controller (servlet) recibe la petición que delega en el Application Controller que resuelve la petición entrante en la invocación del comando apropiado. El comando invoca un servicio de la lógica del negocio que devuelve los datos solicitados, denominado antes el modelo de presentación. En este punto el Application Controller realiza una gestión de la vista identificando la vista apropiada y dirigiendo el control a la misma. El View Helper adapta y transforma el modelo de presentación a la vista.



2. STRUTS 2

2.1. Introduccion:

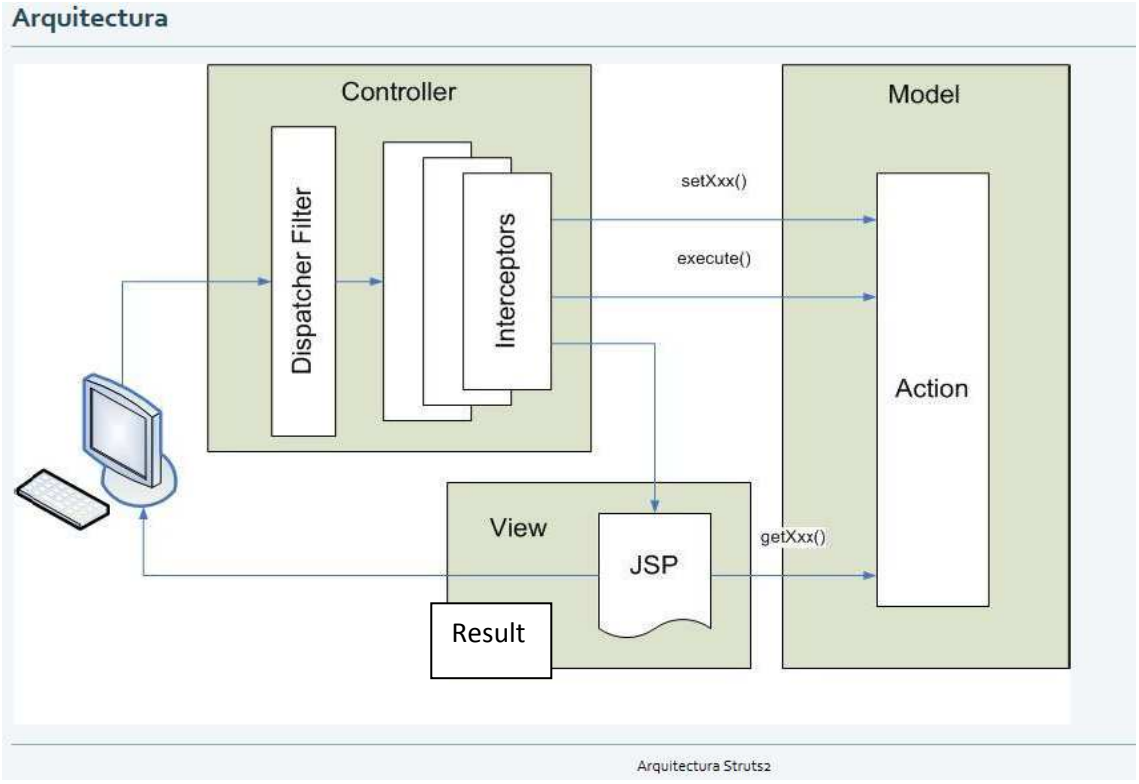
Struts 2 es un framework para el desarrollo de aplicaciones web, el cual hace que la implementación de las mismas sea más sencilla, más rápida y con menos complicaciones, para ello el framework automatiza las tareas comunes y tediosas del dominio de la aplicación.

Además hace que estas sean más robustas y flexibles, para conseguirlo busca una solución arquitectónica dentro del flujo de trabajo común del dominio en cuestión. **Struts 2** es un framework de presentación, dentro de las capas en las que se divide una aplicación en la arquitectura **JEE**. Este framework implementa el patrón de diseño **MVC (Modelo Vista Controlador)** tipo dos y que podemos configurar de varias maneras; además proporciona algunos componentes para la capa de vista. También, proporciona una integración perfecta con otros frameworks para implementar la capa del modelo (como **Hibernate** y **Spring**).

Para hacer más fácil presentar datos dinámicos, el framework incluye una biblioteca de etiquetas web. Las etiquetas interactúan con las validaciones y las características de internacionalización del framework, para asegurar que las entradas son válidas, y las salidas están localizadas. La biblioteca de etiquetas puede ser usada con **JSP**, **FreeMarker**, o **Velocity**; también pueden ser usadas otras bibliotecas de etiquetas como **JSTL** y soporta el uso de componentes **JSF**.

Como dije antes: el objetivo de **Struts 2** es hacer que el desarrollo de aplicaciones web sea fácil para los desarrolladores. Para lograr esto, **Struts 2** cuenta con características que permiten reducir la configuración gracias a que proporciona un conjunto inteligente de valores por defecto. Además hace uso de anotaciones y proporciona una forma de hacer la configuración de manera automática si usamos una serie de convenciones (y si hacemos uso de un plugin especial).

2.2. Arquitectura:



Dibujo cogido del *tutorial* ***http://joeljil.wordpress.com/2010/05/31/struts2/*** que a su vez está cogido de la documentación oficial de Apache.

Siguiendo el patrón de diseño MVC, el cual separa tres secciones diferenciadas llamadas Modelo, Vista y Controlador. Esto busca separar el modelo de datos, las interfaces de usuario y la lógica de negocios en tres componentes diferentes.

El **Modelo** consiste básicamente en los datos de la aplicación y las reglas de negocio, papel desempeñado en Struts 2 por los actions. En los actions se encapsulan las llamadas a la lógica del negocio (o la delegación de las mismas a otros componentes) y además sirve como lugar para la transferencia de los datos del negocio.

La **Vista** es el componente de presentación del patrón. Traduce el estado de la aplicación en una presentación visual con la que el usuario puede interactuar. En Struts 2, son los *Results*. Puede usarse cualquier tecnología de vista como los JSP, XSLT, Velocity o FreeMarker entre otros.

La función de **controlador** es asumida en Struts por el Filterdispatcher que inspecciona todas las peticiones entrantes del cliente (a menudo recibe el nombre de controlador frontal) para decidir la acción que debe encargarse de gestionar la petición. Únicamente se debe informar a Struts de qué URL se vincula con cada acción ya sea desde un fichero XML o desde anotaciones.

2.3. Funcionamiento. Principales Componentes:

Los principales componentes del framework son:

- Filterdispatcher
- Interceptors
- Actions
- Results.

Además se utiliza el lenguaje Object-Graph Navigation Language, ONGL y unos componentes denominados ValueStack y ActionContext.

2.3.1. Filterdispatcher.

El corazón de **Struts 2** es un filtro, conocido como el "**FilterDispatcher**". Este es el punto de entrada del framework. A partir de él se lanza la ejecución de todas las peticiones que involucran al framework.

Así las principales responsabilidades del "**FilterDispatcher**" son:

A. Ejecutar los **Action**s, que gestionan las peticiones.

B. Comenzar la ejecución de la cadena de interceptores. Aunque el Filterdisptacher ha seleccionado la acción a ejecutar, antes y después de la misma, se ejecutan estos componentes que liberan a la acción de realizar tareas comunes, por ejemplo de conversión y validación. Observar que esto no modifica el patrón MVC.

C. Limpiar el "**ActionContext**", para evitar fugas de memoria. Después se aclarará el concepto de ActionContext.

Así Struts 2 procesa las peticiones usando tres elementos principales: Interceptores , Acciones y Resultados .

2.3.2. Interceptores.

Los interceptores son clases que siguen el patrón interceptor. Estos permiten que se implementen funcionalidades cruzadas o comunes para todos los **Action**s, pero que se ejecuten fuera del **Action** (por ejemplo validaciones de datos, conversiones de tipos, población de datos, carga de ficheros etc.). El uso de interceptores permite mejorar la implementación del MVC, ya que en caso de no existir, sus tareas las debería realizar otro componente, por ejemplo el modelo, dando lugar a una baja cohesion.

Los interceptores realizan sus tareas **antes** y **después** de la ejecución de un **Action** y también pueden evitar que un **Action** se ejecute (por ejemplo si estamos haciendo alguna validación que no se ha cumplido).

Los interceptores más comunes ya vienen integrados y pre-configurados en Struts 2, pero si alguna funcionalidad que necesitamos no se encuentra en los interceptores de Struts podemos crear nuestro propio interceptor y agregarlo a la cadena que se ejecuta por default.

De hecho muchas de las características con que cuenta **Struts 2** son proporcionadas por los interceptores de ahí la explicación en la introducción.

2.3.2.a. Pilas de interceptores.

Sobre un **Action** se pueden aplicar más de un interceptor. Para lograr esto **Struts 2** permite crear pilas o **stacks de interceptores** y aplicarlas a los **Action**s. Cada interceptor es aplicado en el orden en el que aparece en el stack. También podemos formar pilas de interceptores en base a otras pilas.

Análogamente existen pilas de interceptores que vienen pre-configurados con **`Struts 2`**.

Ejemplo de los interceptores incorporados existentes y de las pilas por defecto extraído del fichero struts-default.xml

```
 <interceptors>
<interceptor name="alias"
class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
<interceptor name="autowiring"
class="com.opensymphony.xwork2.spring.interceptor.ActionAutowiringInterceptor
"/>
<interceptor name="chain"
class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
<interceptor name="conversionError"
class="org.apache.struts2.interceptor.StrutsConversionErrorInterceptor"/>
<interceptor name="cookie"
class="org.apache.struts2.interceptor.CookieInterceptor"/>
<interceptor name="clearSession"
class="org.apache.struts2.interceptor.ClearSessionInterceptor" />
<interceptor name="createSession"
class="org.apache.struts2.interceptor.CreateSessionInterceptor" />
```

```
<interceptor name="debugging"
class="org.apache.struts2.interceptor.debugging.DebuggingInterceptor" />
<interceptor name="execAndWait"
class="org.apache.struts2.interceptor.ExecuteAndWaitInterceptor"/>
<interceptor name="annotationWorkflow"
class="com.opensymphony.xwork2.interceptor.annotations.AnnotationWorkflowInte
rceptor" />
<interceptor name="multiselect"
class="org.apache.struts2.interceptor.MultiselectInterceptor" />
<!-- Basic stack -->
   <interceptor-stack name="basicStack">
        <interceptor-ref name="exception"/>
        <interceptor-ref name="servletConfig"/>
        <interceptor-ref name="prepare"/>
        <interceptor-ref name="checkbox"/>
        <interceptor-ref name="multiselect"/>
        <interceptor-ref name="actionMappingParams"/>
        <interceptor-ref name="params">
<param name="excludeParams">dojo\..*,^struts\..*,^session\..*,
^request\..*,^application\..*,^servlet(Request|Response)\..*,parameters\...*
</param>
        </interceptor-ref>
        <interceptor-ref name="conversionError"/>
   </interceptor-stack>

  <!-- Sample validation and workflow stack -->
     <interceptor-stack name="validationWorkflowStack">
           <interceptor-ref name="basicStack"/>
           <interceptor-ref name="validation"/>
           <interceptor-ref name="workflow"/>
   </interceptor-stack>
```

2.3.3. Acciones

Las acciones o **Action**s son clases encargadas de realizar la lógica para servir una petición. Cada **URL es mapeada a una acción específica**, la cual proporciona la lógica necesaria para servir a cada petición hecha por el usuario.

Así podemos destacar las tres tareas principales de una acción:

1. Cumple la función de modelo en el patrón MVC, agrupando el trabajo para una petición dada.
2. Ofrecer una ubicación para la transferencia de datos. Se explica en el punto que se habla del lenguaje OGNL y el ValueStack.
3. La acción indica al framework que resultado debe mostrar la vista que se devolverá en respuesta a la petición. La acción devuelve una cadena de control que selecciona el resultado a mostrar.

2.3.3.a. Requisitos de una acción

Las acciones de Struts 2 no necesitan implementar una interface o extender de alguna clase base. Ni siquiera tiene que implementar la interfaz Action. El único requisito para que una clase sea considerada un

Action es que debe tener **un método** que **no reciba argumentos** que **retorne ya sea un String o un objeto de tipo Result**. Por defecto el nombre de este método debe ser "**execute**" aunque podemos ponerle el nombre que queramos y posteriormente indicarlo en el archivo de configuración de **Struts**.

Aunque lo normal es que implementen la interfaz, com.opensymphony.xwork2.Action con el únic o método execute().

String execute() throws Exception.

Una de las ventajas de usar la interfaz Action es que ofrece unas determinadas constantes de String útiles para los valores de retorno:

Dibujo obtenido de la documentación de Struts.

Field Summary

static String	**ERROR** The action execution was a failure.
static String	**INPUT** The action execution require more input in order to succeed.
static String	**LOGIN** The action could not execute, since the user most was not logged in.
static String	**NONE** The action execution was successful but do not show a view.
static String	**SUCCESS** The action execution was successful.

Estas constantes se pueden usar como valores de la cadena de control devueltos por execute().

Cuando el resultado es un **String** (lo cual es lo más común), el **Result** correspondiente se obtiene de la configuración del **Action**. Esto se usa para generar una respuesta para el usuario, lo cual veremos un poco más adelante.

Así, los **Action**s pueden ser objetos java simples (**POJO**s) que cumplan con el requisito anterior, pero también pueden implementar la interface "**com.opensymphony.xwork2.Action**" o extender una clase base que proporciona **Struts 2**: "**com.opensymphony.xwork2.ActionSupport**" , que nos hace más sencilla su creación y manejo

La clase "**ActionSupport**" implementa la interface "**Action**" y contiene una implementación del método "**execute()**" que retorna un valor de "**SUCCESS**". Pero además implementa dos interfaces que se coordinan con el interceptor DefaultWorkflowInterceptor para ofrecer una ***validación lógica*** básica. Este interceptor está en la pila por defecto, si se observa el fichero struts-default.xml:

```
<interceptors>

..... .

<interceptor name="workflow"
class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"/>

.....
<interceptor-stack name="defaultStack">
```

```
.....
 <interceptor-ref name="params">

<paramname="excludeParams">dojo\..*,^struts\..*,^session\..*,^request\..*,^ap
plication\..*,^servlet(Request|Response)\..*,parameters\...*</param>
 </interceptor-ref>
<interceptor-ref name="workflow">
  <param name="excludeMethods">input,back,cancel,browse</param>
</interceptor-ref>
</interceptor-stack>
```

Además en este fichero se observa la declaración de la pila de interceptores por defecto la **defaultStack**. En la misma se observan dos interceptores importantes : el **params**, que mueve los datos de la petición al objeto acción y el interceptor **workflow** que valida los datos antes de aceptarlos en nuestro modelo. De ahí que **params** vaya siempre antes que workflow en la pila.

Tal como se comentaba anteriormente, ActionSupport se coordina con workflow para ofrecer validación de los datos. ActionSupport implementa un método valídate() donde está la lógica de validación de los datos; cuando se ejecuta el interceptor workflow, invoca el método valídate de la acción. Si algún dato no es correcto se almacenan los errores con métodos de mensajes de error de ActionSupport. Al acabar el método valídate, el control vuelve al interceptor que comprueba la existencia de mensajes de error. Si existen, se modifica el flujo de trabajo de la petición y devuelve al usuario el formulario de entrada.

Nota: Aparte del método valídate() de la interfaz Validateable, Struts 2 permite un marco de trabajo de validación que permite una solución más versátil y fácil de mantener. Este marco de trabajo tiene su propia estructura dentro de Struts 2.

Ejemplo de una clase Register que valida tres campos: username, password y portfolioName, usando el método valídate(). Lo importante en este ejemplo es ver que en valídate se ha implementado la lógica de negocio para ver si los datos son correctos y en caso de que no lo sean se llama al método `addFieldError` que dejará un mensaje de error que puede ser leído por el interceptor workflow para variar el flujo de trabajo.

```
public class Register extends ActionSupport {


      public String execute(){


            User user = new User();
            user.setPassword( getPassword() );
            user.setPortfolioName( getPortfolioName() );
            user.setUsername( getUsername() );

            getPortfolioService().createAccount( user );
            return SUCCESS;
      }


      /* JavaBeans Properties to Receive Request Parameters */

      private String username;
      private String password;
      private String portfolioName;

      public String getPortfolioName() {
```

```
            return portfolioName;
      }
```

……

```
public void validate(){

PortfolioService ps = getPortfolioService();

            /* Check that fields are not empty */
if ( getPassword().length() == 0 ){
      addFieldError( "password", getText("password.required") );
}if ( getUsername().length() == 0 ){
addFieldError( "username", getText("username.required") );
}if ( getPortfolioName().length() == 0  ){
addFieldError( "portfolioName", getText( "portfolioName.required" ));
}
/* Make sure user doesn't already have an account */
if ( ps.userExists(getUsername() ) ){
                  addFieldError("username", getText( "user.exists"));
   }

}
```

2.3.4. Results.

Antes de tratar los results se realiza una breve reseña a la construcción de las vistas.Para construir una vista usaremos etiquetas. Struts 2 posee una biblioteca de etiquetas que permiten crear de forma dinámica páginas web. Las interfaces de de la API de etiquetas se han implementado en tres tecnologías, JSP, Velocity y FreeMaker. Pero por no extenderme sólo mostraré ejemplos con JSP que son más comunes.

2.3.4.a. Etiquetas.

Struts posee 4 diferentes tipos de etiquetas:

1. Etiquetas de control de flujo. Ofrecen herramientas para alterar de forma condicional el flujo de renderización de la página. Estan las etiquetas iterator para tratar colecciones y las etiquetas if/else.

2. Etiquetas de datos. Se centran en los modos de extraer los datos del ValueStack y/o configurar valores en el mismo.

3. Etiquetas UI. Cada etiqueta de componente UI es una unidad funcional con la que el usuario puede interactuar e introducir datos. En cada componente UI existe un mecanismo de control de formularios HTML. Estos componentes integran todas las áreas del framework, desde transferencia de datos y validación hasta internalización y aspecto exterior. Estos componentes UI se construyen a su vez en una miniarquitectura por capas.

4. Etiquetas mixtas. Realmente las denominan así porque no coinciden con ningún tipo de las anteriores, sería como una miscelánea.

2.3.4.b. Sintaxis de las etiquetas la API de etiquetas de Struts 2.

Tal como se ha comentado las interfaces de de la API de etiquetas se han implementado en tres tecnologías, JSP, Velocity y FreeMaker.

Como realmente las versiones JSP de Struts 2 son exactamente iguales a cualquier otra etiqueta JSP, para usar etiquetas JSP lo único necesario es declarar la propiedad **taglib** al comienzo de la página:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
```

No comento las otras porque excede el objeto de la práctica.

En resumen, una acción que gestiona una petición recibe lo datos de la misma, ejecuta la lógica de negocio, deja los datos del dominio resultantes expuestos en el ValueStack y por último devuelve una

cadena de control para indicar al framework cual de los resultados disponibles debe renderizar la vista. Realmente el resultado es una encapsulación de las tareas de vista del patrón MVC de Struts 2.

Generalmente, estas tareas de vista equivalen a la creación de una página HTML que se devuelve al cliente. Por defecto Struts 2 usa un tipo de resultados que funciona con páginas JSP pero se pueden usar Velocity y Freemarker para renderizar las páginas HTML:

Así después de que un **Action** ha sido procesado se debe enviar la respuesta de regreso al usuario, esto se realiza usando **result**s. Este proceso tiene dos componentes, el tipo del **result** y el **result** mismo.

El tipo del **result** indica cómo debe ser tratado el resultado que se devolverá al cliente. Por ejemplo un tipo de **Result** puede enviar al usuario de vuelta una **JSP** (lo que haremos más a menudo), otro puede redirigirlo hacia otro sitio, mientras otro puede enviarle un flujo de bytes (para descargar un archivo por ejemplo).

Naturalmente un **Action** puede tener más de un **result** asociado. Esto nos permitirá enviar al usuario a una vista distinta dependiendo del resultado de la ejecución del **Action**. Por ejemplo en caso de que todo salga bien, enviaremos al usuario al **result** "**sucess**", si algo sale mal lo enviaremos al **result** "**error**", o si no tiene permisos lo enviaremos al **result** "**denied**".

2.3.4.c. Tipos de resultados más comunes.

Como es habitual los tipos se definen en struts-default.xml. Por ejemplo se muestra la declaración de FreeMarkerResult. Se explica puesto que en el framework se usan tres tipos de resultados con alguna similitud.

```
<package name="struts-default" abstract="true">
<result-types>
<result-type name="freemarker"
class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
```

1. RequestDispatcher (dispatcher). Este resultado se usa para renderizar una página JSP. Al heredar de struts-default, se deja el DispatcherResult como tipo resultado por defecto:

   ```
   <result-type name="dispatcher"
   class="org.apache.struts2.dispatcher.ServletDispatcherResult"
   default="true"/>
   ```

   Observar el campo default=true, al ser por defecto, se puede escribir los elementos de resultados de manera sencilla:

```
<action name="PortfolioHomePage" class=
"manning.chapterEight.PortfolioHomePage">

<result>/chapterEight/PortfolioHomePage.jsp</result>
</action>
```

En el núcleo del tipo de resultados DispatcherResult se encuentra un la clase javax.servlet.RequestDispatcher que procede de la API de Servlet y que permite a un servlet entregar el procesamiento a otro recurso de la aplicación web. Este objeto usa dos métodos, include() y forward().

Nota: Es diferente de una redirección HTTP, en este caso la entrega tiene lugar en el mismo hilo, los recursos están disponibles en el ActionContext. La redirección envía al explorador a realizar una petición a una URL diferente.

El tipo de result tiene dos parámetros:

a. Location: que ofrece la ubicación del recurso del servlet hacia al cual debemos despachar la petición.

b. Parse: que determina si la cadena anterior se parseara para expresiones OGNL.

2. Redirect. Este resultado indica al navegador que se redirija a otro URL. Con diferencia al caso anterior, con la redirección la primera petición ha terminado y la respuesta de redirección HTTP ordena al navegador apuntar a otra localización. Es decir desde el punto de vista del ActionContext el estado de la petición ha desaparecido.

   Generalmente se usa para modificar el url mostrado en el explorador. Tiene dos parámetros también, location y parse.

```
<result-type name="redirect"
class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
```

```
<action name="SendUserToSearchEngineAction" class="myActionClass">
   <result type="redirect" >http://www.google.com</result>
</action>
```

Tiene la utilidad de que se pueden embeber expresiones OGNL para crear localizaciones dinámicas.

```
<action name="SendUserToSearchEngineAction" class="myActionClass">
       <result type="redirect" >
        http://www.google.com/?myParam=${defaultUsername}
       </result>
</action>
```

Se usa **$** para la expresión OGNL.

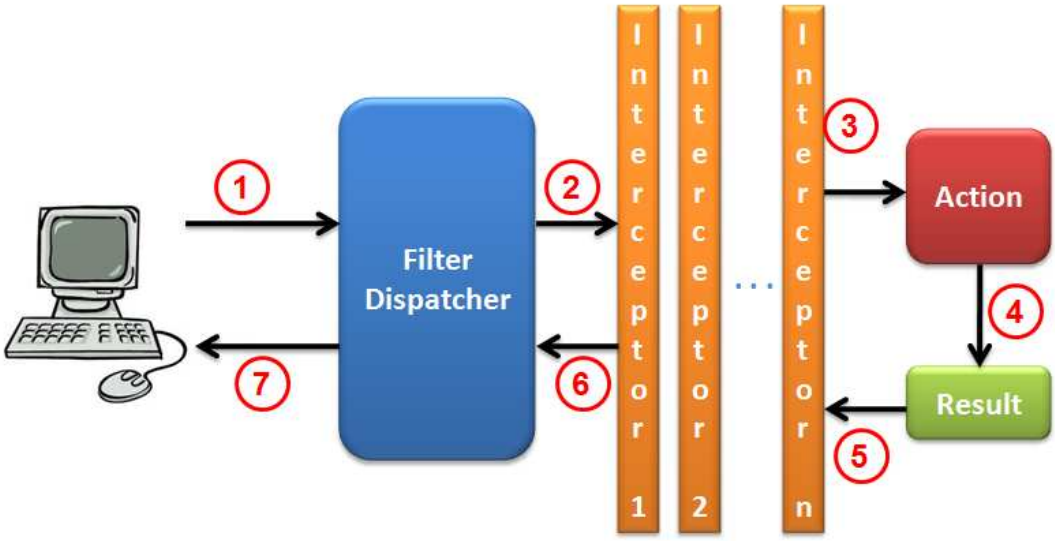
3. RedirectAction. Indica al navegador que se redirija a otra acción de Struts. Es parecido a redirect pero con RedirectAction se pueden incluir nombres lógicos de las acciones de Struts 2 tal como están definidos en la arquitectura declarativa. Es decir no es necesario embeber ningún url real en las declaraciones de resultados.

```
<result-type name="redirectAction"
class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
```

2.4. Funcionamiento general.

Dibujo original extraído *de http://www.javatutoriales.com/2011/06/struts-2-parte-1-configuracion.html* y modicado por mí.

Diagrama simplificado del funcionamiento.



Los pasos que sigue una petición son:

1 – El navegador web hace una petición para un recurso de la aplicación. El **FilterDispatcher** revisa la petición y determina el **Action** apropiado para servirla.

2 – Se aplican los interceptores, los cuales realizan algunas funciones como validaciones, flujos de trabajo, manejo de la subida de archivos, etc.

3 – Se ejecuta el método adecuado del **Action** (por default el método "**execute**"), este método usualmente almacena y/o devuelve alguna información referente al proceso.

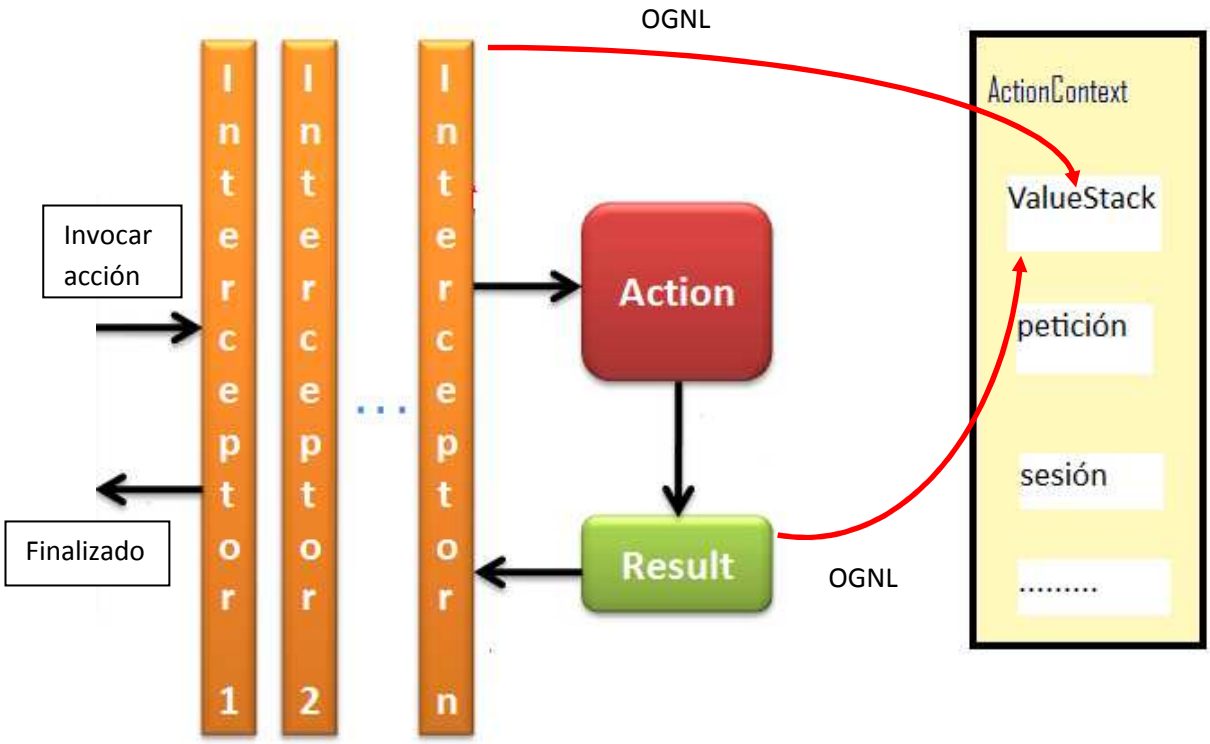
4 – El **Action** indica qué **result** debe ser aplicado. El **result** genera la salida apropiada dependiendo del resultado del proceso.

5 – Se aplican al resultado los mismos interceptores que se aplicaron a la petición, pero en orden inverso.

6 – El resultado vuelve a pasar por el **FilterDispatcher** aunque este ya no hace ningún proceso sobre el resultado (por definición de la especificación de **Servlets**, si una petición pasa por un filtro, su respuesta asociada pasa también por el mismo filtro).

7 – El resultado es enviado al usuario y este lo visualiza.

2.5. ValueStack y el lenguaje Object-Graph Navigation Language ONGL.



El ValueStack es una zona de almacenamiento que contiene todos los datos asociados con el procesamiento de una petición. Struts 2 usa esta zona de almacenamiento para los datos del dominio de aplicación que necesitará durante el procesamiento de la aplicación. Los datos se mueven a ValueStack en preparación de del procesamiento de la aplicación, son manipulados allí durante la ejecución de la acción y se leen desde allí cuando los resultados se muestran en páginas de respuesta.

ONGL es un lenguaje de expresiones que permite acceder a los datos colocados en ValueStack. ValueStack a su vez está almacenado en un contexto ThreadLocal llamado ActionContext. Nota, el uso de un ThreadLocal es una buena idea que también se usa en el framework implementado en el pfc. Se almacena la configuración que será accesible desde cualquier parte del thread pero cada thread tendrá su propia variable thread local, es decir no podrá acceder a la variable local de otro thread.

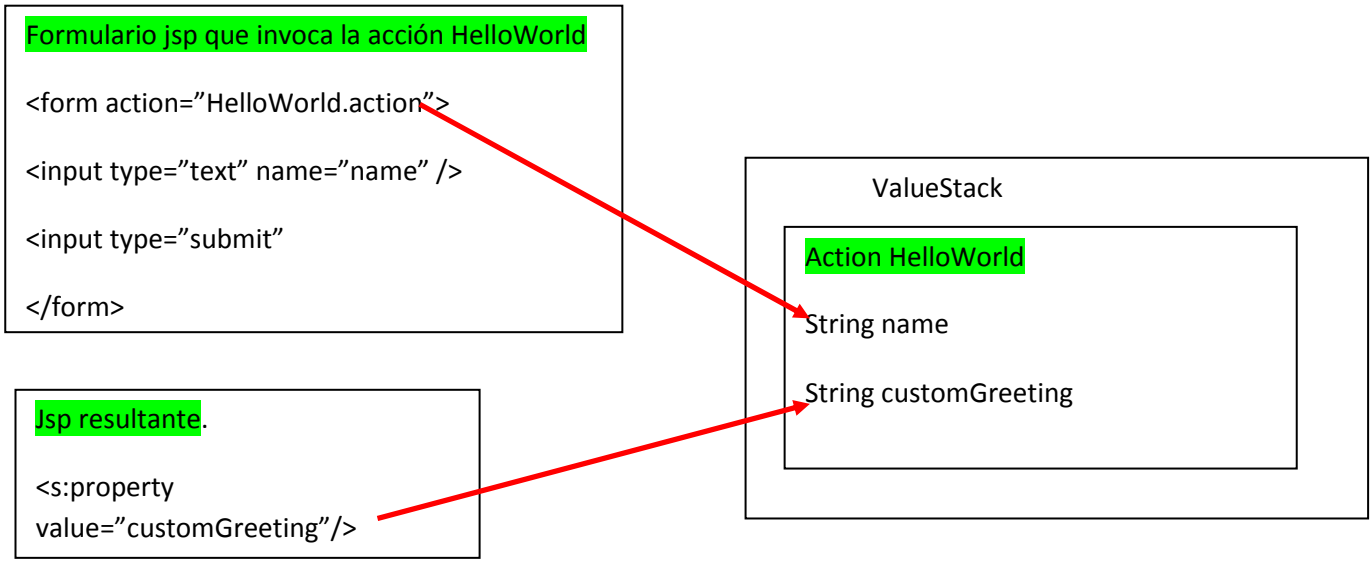
ActionContext contiene todos los datos que conforman el contexto en el que ocurre una acción determinada, incluyendo el ValueStack.

Como ejemplo la sencilla clase HelloWorld con dos atributos, name y customGreeting con sus correspondientes métodos get y set.

```
public class HelloWorld {

private static final String GREETING = "Hello ";

    public String execute()  {
       setCustomGreeting( GREETING + getName() );
       return "SUCCESS";
    }

    private String name;
```

```
    public String getName() {
        return name;
    }

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

    private String customGreeting;

    public String getCustomGreeting()
    {
      return customGreeting;
    }

    public void setCustomGreeting( String customGreeting ){
      this.customGreeting = customGreeting;
    }
}
```

Tal como se ha señalado antes, las acciones ofrecen una ubicación para la transferencia de datos, el objeto acción se sitúa en ValueStack permitiendo que el resto del framework pueda acceder a los mismos. Así por ejemplo en la siguiente figura se ve un formulario que tiene el atributo "name". Struts 2 mueve el valor de name hacia la propiedad de la acción. En el otro caso, la JSP resultante de Helloworld, extrae el valor de la propiedad "customGreeting".



2.6. Configuración del Framework.

Extraído del libro Struts 2, D. Brown, C.M. Davis y S Stanlick. Editorial Anaya.

Struts 2 usa una arquitectura declarativa, describe sus componentes arquitectónicos a través de ficheros XML o anotaciones java y deja la creación de las instancias en tiempo de ejecución al framework.

Así Struts 2 define los componentes que utilizará y como vincularlos para formar rutas de trabajo.

Tal como se ha comentado en párrafo anterior Struts 2 permite dos métodos de declaración de su arquitectura:

2.6.A. Mediante ficheros XML. El fichero es el struts.xml. La tarea de crear el archivo corresponde al desarrollador. Un ejemplo donde se

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

   <constant name="struts.devMode" value="true" />

   <package name="default" namespace="/" extends="struts-default">
                <action name="Menu">
                      <result>/menu/Menu.jsp</result>
                </action>

    </package>

    <include file="manning/chapterTwo/chapterTwo.xml"/>
</struts>
```

Struts-default.

Las acciones se organizan en paquetes, similares a los de java, donde un atributo importante es **extends.** Este atributo indica el paquete padre del que hereda. Para usar los componentes incorporados en struts se debe indicar `extends="struts-default".` Como es de suponer, este paquete está definido en el fichero struts-default.xml. Este paquete declara desde pilas de interceptores a los tipos comunes de resultados.

2.6.B. Mediante anotaciones java. Las anotaciones permiten añadir metadatos directamente a los archivos java, es decir las anotaciones se incluyen en las clases java que implementan las acciones. Al igual que con los elementos de XML, con las anotaciones Struts2 crea los componentes en tiempo de ejecución. Se ve un ejemplo de anotaciones, declara en rojo de la clase Helloword.

```
package manning.chapterTwo;
import org.apache.struts2.config.Result;
import org.apache.struts2.dispatcher.ServletDispatcherResult;
@Result(name="SUCCESS", value="/chapterTwo/HelloWorld.jsp" )
public class AnnotatedHelloWorldAction {
  private static final String GREETING = "Hello ";
   public String execute()  {
          setCustomGreeting( GREETING + getName() );
          return "SUCCESS";
   }
  private String name;
 public String getName() {
     return name; }
public void setName(String name) {
     this.name = name; }
  private String customGreeting;
  public String getCustomGreeting()
   {return customGreeting;}
    public void setCustomGreeting( String customGreeting ){ this.customGreeting = customGreeting;
```

```
  }
}
```

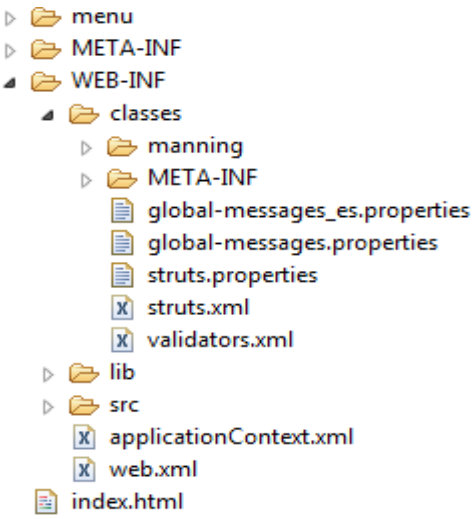
2.6.C. ¿Qué mecanismo elegir ?

Se puede usar cualquiera de los dos mecanismos sin consecuencias funcionales. En esta práctica se sigue el esquema de ficheros XML por estar más familiarizado y ser un buen método para empezar a trabajar con los struts2.

2.7. Diseño de una aplicación web Struts2

La mayor parte de los requisitos exigidos a la estructura de una aplicación con Struts 2 provienen de los requisitos exigidos a las aplicaciones web por la API de Servlet.

Ejemplo de una pequeña aplicación. Se muestra la estructura expandida de un .war :



Los directorios como "*menú*" se encuentran en el documento raíz de la aplicación web. Generalmente aquí se colocaran los archivos JSP, FreeMarker, Velocity.

En *" WEB-INF"* están los directorios *"lib"* y *"classes"* y el archivo web.xml , denominado descriptor de despliegue, archivo de configuración de todas las aplicaciones web . Nota, en *"src"* está el código fuente, no tiene porqué estar colocado allí aunque en este ejemplo si lo está. En "*lib"* están todas las dependencias de archivo JAR que necesita la aplicación.

En web.xml, están las definiciones de todos los servlets, filtros de servlets y otros componentes de la API de Servlet contenidos en la aplicación. A modo de ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>S2 Example Application - Chapter 1 - Hello World</display-
name>
<filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
        </filter-class>

         <init-param>
```

```
                <param-name>actionPackages</param-name>
                <param-value>manning</param-value>
        </init-param>
</filter>

<filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>


<servlet>
        <servlet-name>anotherServlet</servlet-name>
        <servlet-class>manning.servlet.AnotherServlet</servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>anotherServlet</servlet-name>
      <url-pattern>/anotherServlet</url-pattern>
 </servlet-mapping>

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

En el fichero se puede observar:

- En rojo los elementos <filter> y <filter-mapping> que configuran el FilterDispatcher. El filtro examinará todas las peticiones entrantes que busquen peticiones que tengan por objeto acciones de Struts 2 ya que el patrón al que se mapea el filtro es "/*".
- En verde se indica el parámetro actionPackages, necesario si se usan anotaciones en la aplicación. Este parámetro indica los paquetes con anotaciones.
- Además, en gris, se ha indicado un servlet externo a Struts 2.

Fichero struts.xml.

Es el punto de entrada de la arquitectura declarativa en versión xml. Un sencillo ejemplo.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

 <constant name="struts.devMode" value="true" />


<package name="chapterTwo" namespace="/chapterTwo" extends=
"strutsdefault">

<action name="Name">
       <result>/chapterTwo/NameCollector.jsp</result>
</action>


<action name="HelloWorld" class="manning.chapterTwo.HelloWorld">
      <result name="SUCCESS">/chapterTwo/HelloWorld.jsp</result>
</action>
</package>

<include file="manning/chapterTwo/chapterTwo.xml"/>
```

```
</struts>
```

En el ejemplo se declaran unas acciones y results. Con package se organizan las acciones, results y otros componentes.

3. SPRING MVC 3

3.1. Introducción.

Spring es un framework que se creó para hacer más simple el desarrollo de las aplicaciones empresariales, haciendo posible el uso de JavaBeans sencillos para conseguir objetivos que antes sólo eran posibles con EJB. Un componente de spring puede ser cualquier POJO.

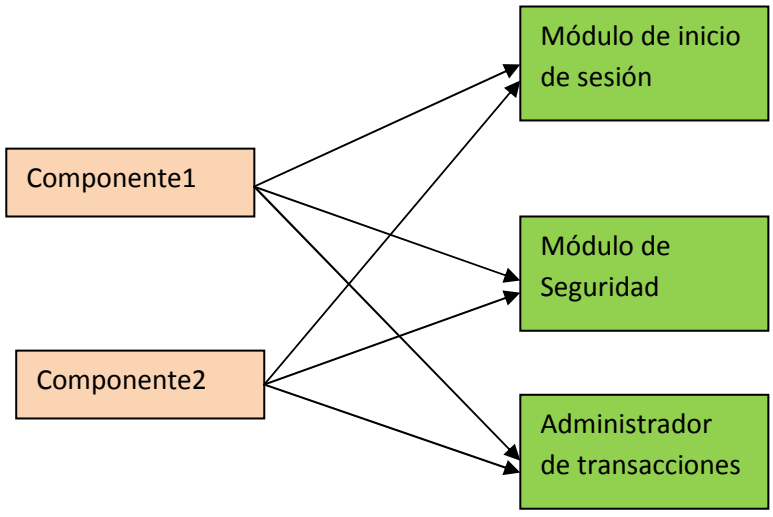
Para reducir la complejidad Spring usa cuatro estrategias:

1. Desarrollo ligero con objetos java simples y POJOs. Frente a EJB que tiene requisitos elevados como los métodos ejbActivate(), ejbPassivate()…al menos hasta la versión EJB 3.
2. Acoplamiento débil mediante inyección de dependencias y la orientación de interfaz.
3. Programación declarativa mediante aspectos y convenciones comunes.
4. Reducción del código reutilizable mediante aspectos y plantillas. Es decir, no escribir varias veces el mismo código. Por ejemplo una consulta JDBC supone establecer la conexión, preparar el statement, realizar la consulta, cerrar la conexión…. La idea es usar plantillas para evitar reescribir el código.

3.2. Programación declarativa mediante aspectos.

La programación orientada a aspectos se define como una técnica que promueve la separación de problemas dentro de un sistema software. Los sistemas están formados por componentes, cada uno de ellos con una responsabilidad específica pero que a menudo cuentan con responsabilidades adicionales más allá de su función básica. En estos componentes se pueden encuentran servicios del sistema como inicio de sesión, transacciones, seguridad. Estos servicios se denominan preocupaciones transversales ya que se deben incluir en diferentes componentes del sistema

Si se reparten estas preocupaciones entre diferentes componentes, se está introduciendo dos niveles de complejidad en el código.



- El código que implementa estas preocupaciones a nivel de sistema se duplica entre los distintos componentes, es decir si se quiere modificar la forma en que funcionan las preocupaciones, se tendrá que acceder a todos los componentes que las usan.

- Los componentes incluyen código no relacionado con la funcionalidad básica.

La programación orientada a aspectos AOP, incluye los servicios en módulos y los aplica de forma declarativa a los componentes que afecten. Esto permite una mayor cohesión de los componentes.

3.3. Contenedores para los bean.

Los objetos de una aplicación de Spring residen en el contenedor de Spring. El contenedor constituye el núcleo del framework, usando la inyección de dependencias para administrar los componentes que forman una aplicación, incluyendo las asociaciones entre componentes que colaboran entre sí, de esta forma se consigue que los objetos sean reutilizables, más fácil de comprender y más fáciles de probar.

Hay varios tipos de contenedores, que se pueden clasificar en dos tipos:

1. Las fábricas de beans, definidas por la interfaz org.springframework.beans.factory.BeanFactory que son los más simples.

2. Los contextos de aplicación definidos por la org.springframework.context.ApplicationContext que son los más usados.

Los tres contextos más habituales son:

1. **ClassPathXmlApplicationContext**: carga una definición de contexto a partir de un fichero xml situado en el classpath y trata los archivos de definición de contexto como recursos de classpath.

2. **FileSystemXmlApplicationContext**: carga la definición desde un fichero xml del sistema de archivos.

3. **XmlWebApplicationContext**: carga las definiciones a partir de un xml almacenado en una aplicación web.

Ejemplo de código de carga del fichero foo.xml.

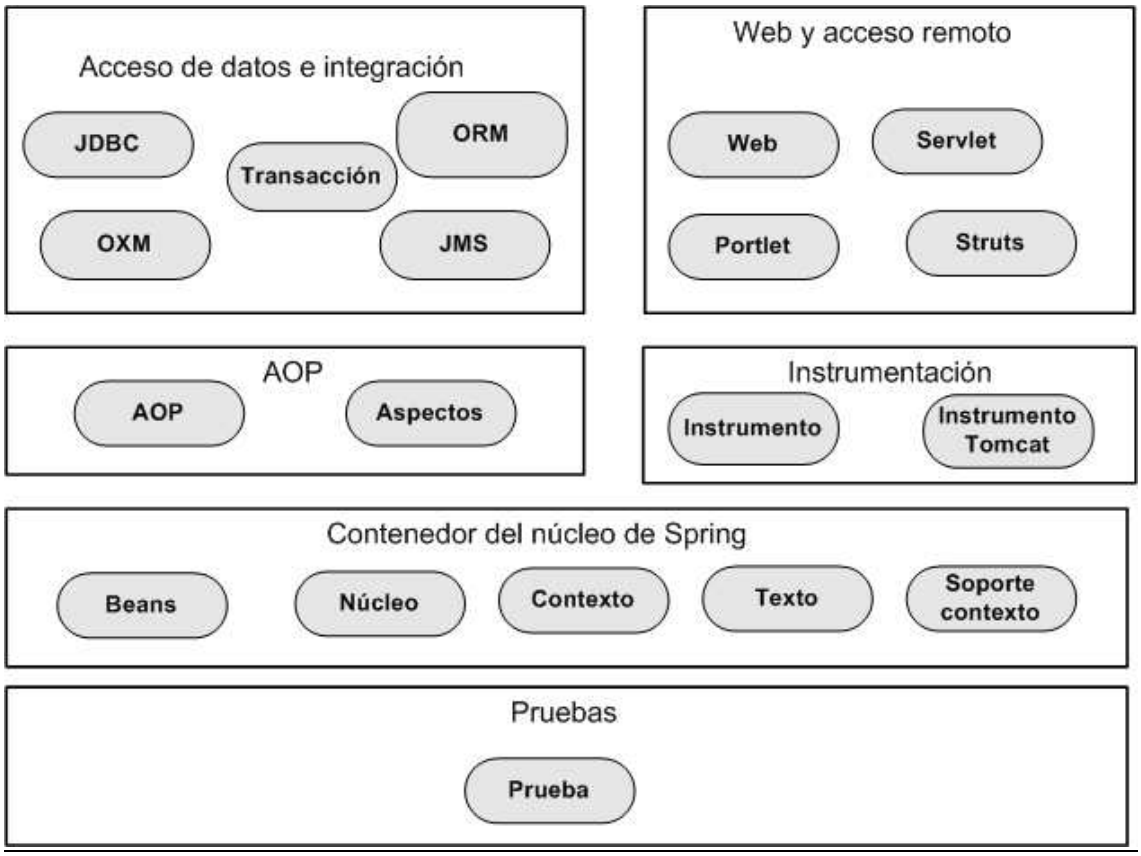
ApplicationContext contex = new FileSystemXmlApplicationContext(“c:\foo.xml”);

3.3.a. Ciclo de un bean.

El ciclo de un bean en el contenedor de Spring es complejo. Primero lo instancia, después le inyecta valores y lo referencia, va ejecutando distintos métodos según la configuración establecida hasta que el bean esta listo para su ejecución. Finalmente si el bean implementa la interfaz DisposableBean, Spring ejecutará sus métodos destroy()

3.4. Componentes de Spring.

Spring está formado por varios módulos:



3.5. Configuración de Spring.

Para usar Spring basado en un contenedor, este se debe configurar. En caso contrario se tendrá un contenedor vacio. Se debe configurar para indicar lo que deben contener los beans y como conectar los beans entre sí.

En Spring 3 hay dos formas de configurar bean en el contenedor Spring. La tradicional mediante ficheros xml y en esta versión 3, también mediante anotaciones java.

Un fichero xml típico tendría el siguiente aspecto:

```
<?xml version="l .0"encocling="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        Xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans~3.0.xsd">

<!—DECLARACIONES DE LOS BEANS-->

</beans>
```

Dentro de< beans > estará toda la configuración de Spring incluyendo declaraciones <bean>.
Beans no es el único espacio de trabajo que dispone Spring. Cuenta con diez espacios de nombre de configuración. Se pueden observar en la siguiente tabla:

Espacio de nombres	Propósito
aop	Proporciona elementos para declarar aspectos y para conectar de forma indirecta y de forma automática clases anotadas @Aspect J-como aspectos de Spring.
beans	El espacio de nombre originario del núcleo de Spring, que permite la declaración de bean e indica cómo deben conectarse.
context	Se incluye con elementos para configurar el contexto de aplicación de Spring, incluyendo la capacidad de detectar y conectar de forma automática bean e inyectar objetos que Spring no administre de forma directa.
jee	Ofrece integración con las API de Java EE, como JNDI y EJB.
jms	Proporciona elementos de configuración para declarar POJO basados en mensajes.
lang	Permite la declaración de bean implementados como secuencias de comandos de Groovy, JRuby o BeanShell.
mvc	Activa las características MVC de Spring, como controladores orientados a anotaciones, controladores de vista e interceptores.
oxm	Permite la configuración de las ubicaciones de asignación de objetos con XML de Spring.
tx	Permite la configuración de transacciones declarativas.
util	Una selección miscelánea de elementos de utilidad. Incluye la capacidad de declarar colecciones de bean y es compatible con elementos de marcador de propiedad.

Además muchos de los componentes del catálogo de Spring, como Spring Segurity y Spring Web Flow cuentan con su propio espacio de nombres de configuración.

3.5.a. Declarar beans:

- Ejemplo de declaración de un sencillo bean.

```
package com.springinaction.springidol;

public class Juggler implements Performer {
  private int beanBags = 3;

  public Juggler() {
  }

  public Juggler(int beanBags) {
    this.beanBags = beanBags;
  }

  public void perform() throws PerformanceException {
    System.out.println("JUGGLING " + beanBags + " BEANBAGS");
  }
}
```

Un bean denominado Duke del tipo Juggler está definido en un fichero spring-idol.xml

```
<bean id="duke"
      class="com.springinaction.springidol.Juggler" />
```

```
  <!--<end id="duke_bean" />-->
```

Este elemento bean indica al framework que cree el objeto. Con el atributo id se le da un nombre para referenciarlo. Es como si esta instancia se cargara con el código java (en realidad Spring los crea mediante reflexión)

```
new com.springinaction.springidol.Juggler();
```

Se carga el context de la aplicación y se ejecuta:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "com/springinaction/springidol/spring-idol.xml");

Performer performer = (Performer) ctx.getBean("duke");
performer.perform();
```

- Inyección referencias de objeto mediante constructores: Ejemplo de una clase PoeticJuggler que recibe un objeto de tipo Poem y un entero.

Declaración de PoeticJuggler que hereda de Juggler.

```
package com.springinaction.springidol;

public class PoeticJuggler extends Juggler {
  private Poem poem;

  public PoeticJuggler(Poem poem) { //<co id="co_injectPoem"/>
    super();
    this.poem = poem;
  }

  public PoeticJuggler(int beanBags, Poem poem) { // <co
id="co_injectPoemBeanBags"/>
    super(beanBags);
    this.poem = poem;
  }

  public void perform() throws PerformanceException {
    super.perform();
    System.out.println("While reciting...");
    poem.recite();
  }
}
```

Declaración de una clase que implementa la interface Poem.

```
package com.springinaction.springidol;
public class Sonnet29 implements Poem {
  private static String[] LINES = {
      "When, in disgrace with fortune and men's eyes,",
      "I all alone beweep my outcast state",
      ……
      "That then I scorn to change my state with kings." };

  public Sonnet29() {
  }
   public void recite() {
    for (int i = 0; i < LINES.length; i++) {
      System.out.println(LINES[i]);
    }
  }
}
```

Declaración de los beans que implementa la clase anterior y PoeticJuggler. Se observa como se pasa unos constructor-arg con value=15 para el entero y un constructor-arg con ref para el objeto "sonnet29" para crear el objeto poeticDuke.

```
<bean id="poeticDuke"
      class="com.springinaction.springidol.PoeticJuggler">
    <constructor-arg value="15" />
    <constructor-arg ref="sonnet29" />
  </bean>

  <bean id="sonnet29"
      class="com.springinaction.springidol.Sonnet29" />
```

Para ejecutarlo :

```
Poem sonnet29 = new Sonnet29();
Performer duke = new PoeticJuggler(15, sonnet29);
```

- Creación de beans mediante métodos de fábrica:

Caso de que haya que instanciar un objeto usando un método factory estático. En este caso se debe usar el atributo *factory-method.*

Ejemplo con clase Singleton:

```
package com.springinaction.springidol;

public class Stage {
  private Stage() {
  }

  private static class StageSingletonHolder {
    static Stage instance = new Stage(); //<co id="co_lazyLoad"/>
  }

  public static Stage getInstance() {
    return StageSingletonHolder.instance; //<co id="co_returnInstance"/>
  }
}
```

En este caso no se cuenta con un constructor público, se debe usar el método getInstance y en el bean usar el atributo *factory-method.*

```
<bean id="theStage" class="com.springinaction.springidol.Stage" factory-
method="getInstance"/>
```

3.5.b. Ámbito de los bean.

De forma predeterminada todos los bean de Spring son singleton. Cuando el contenedor proporciona un bean, siempre va a proporcionar exactamente la misma instancia de bean. En el caso de que se necesite una instancia de bean cada vez que se solicite se debe usar el atributo *scope.*

Por ejemplo en caso de un bean "ticket" que se necesita uno distinto cada vez, se le da el valor prototype al atributo scope:

```
<bean id="ticket" class="com.springinaction.springidol.Ticket"
scope="prototype"/>
```

Los valores de scope pueden ser:

Ámbito	Función
singleton	Valor por defecto
prototype	Permite que un bean se instancie cuantas veces se requiera
request	La definición del bean se incluye en un ámbito en el que se asigna a una solicitud HTTP. Sólo válido para Spring con funcionalidad web, como MVC Spring
session	La definición del bean se incluye en un ámbito en el que se asigna a una sesión HTTP. Sólo válido para Spring con funcionalidad web, como MVC Spring
global-session	La definición del bean se incluye en un ámbito en el que se asigna a una sesión HTTP global. Sólo válido para el contexto de un porlet.

3.5.c. Inyecciónes en las propiedades de los bean.

Por lo general las propiedades de los JavaBean son privadas y tienen un par de métodos de acceso setXXX() y getXXX(). Spring saca partido del método establecedor de una propiedad para configurar su valor.

a. Un atributo sencillo.

Por ejemplo una clase Instrumentalist que tuviera un atributo String song se declararía:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
</bean>
```

En resumen las propiedades de los beans se pueden configurar con el elemento property, que inyecta un método establecedor de la propiedad.

b. Referencia a otro bean.

Si además tuviera declarado otra propiedad instrument que fuera un objeto de la clase Instrument:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone" />
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
  </bean>
```

En este caso cuando se crea el objeto kenny2 de la clase Instrumentalist, la propiedad instrument referencia al objeto saxophone de la clase Instrument.

c. También se permite la inyección de beans internos, es decir definidos en el ámbito de otro bean.

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument">
       <bean class="com.springinaction.springidol.Saxophone" />
    </property>
</bean>
```

d. Se puede abreviar property indicando el espacio de nombres p. Ejemplo

```
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist"
      p:song = "Jingle Bells"
      p:"instrument-ref= "saxophone" />
```

e. Conexión de colecciones.

Spring permite cuatro elementos de configuración de colecciones <list>, <set>, <map>, <props>. Los dos primeros se corresponde ncon colecciones de java.util.Collection, map con java.util.Map y props con java.util.Properties.

Ejemplos de conexión de una lista de una lista de objetos instrumentos y un map con el nombre del instrumento y el objeto instrumento.

```
<bean id="hank"
class="com.springinaction.springidol.OneManBand">
<property name="instruments">
 <list>
    <ref bean="guitar" />
    <ref bean="cymbal" />
    <ref bean="harmonica" />
 </list>
 </property>
</bean>

<bean id="hank"
class="com.springinaction.springidol.OneManBand">
<property name="instruments">
<map>
  <entry key=”GUITAR” value-ref="guitar" />
  <entry key=”CYMBAL” value-ref="cymbal" />
  <entry key=”HARMONICA” value-ref="harmonica" />
</map>
</property>
</bean>
```

f. Conexión con expresiones. Una de las novedades de Spring 3 es que ha introducido el Lenguaje de Expresiones de Spring (Spring Expression Language- SpEL), que permite conectar valores con las propiedades de un bean o con los argumentos de un constructor utilizando expresiones que se evalúan en el tiempo de ejecución.

Ejemplos:

Referencia a un bean, observar que se usan los marcadores #{} con el atributo value del elemento property.

```
<property name="instrument" value="#{saxophone)}" />
```

Que es lo mismo que <property name="instrument" ref="saxophone" />

Ejecución del método song de un objeto Kenny.

```
<bean id="carl"
        class="com.springinaction.springidol.Instrumentalist">
       <property name="song" value="#{kenny.song()}" />
      </bean>
```

O de forma análoga con código java.

```
Instrumentalist carl = new Instrumentalist();
        carl.setSong(kenny.song());
```

Además SpEL permite operaciones, aritméticas, relacionales, lógicas, condicionales y expresiones regulares.

3.6. Minimizar la configuración XML en Spring.

Una aplicación grande supone escribir mucho código xml de configuración, para evitarlo Spring 3 cuenta con dos herramientas:

1. La conexión automática reduce o incluso elimina la necesidad de utilizar los elementos <property> y <constructor-arg> al permitir que Spring decida, de forma automática, como conectar las dependencias del bean.
2. La detección automática permite aumentar las capacidades de la conexión automática, al permitir que Spring decida que clases deben configurarse como bean, reduciendo la necesidad del elemento bean.

Usando estas herramientas conjuntamente se reduce el código xml.

3.6.a. Conexión mediante anotaciones.

Comentar que Spring permite declarar la conexión automática mediante anotaciones. Para usar las anotaciones se debe usar el elemento <context:annotation-config/>.
Spring admite diferentes tipos de anotaciones, sólo citándolas:

- @Autowired.
- @Inject.
- @Resource.

Además está la anotación @Qualifier para calificar dependencias ambiguas.

3.7. Spring orientado a aspectos.

La programación orientada a aspectos AOP permite modularizar las preocupaciones transversales de una aplicación, en otras palabras los servicios que afectan a los componentes de la aplicación.
Habitualmente para reutilizar una funcionalidad en java se usa la herencia o la delegación. AOP permite definir una funcionalidad común en una ubicación y definir de forma declarativa, cómo y dónde se va a aplicar, sin modificar la clase para aplicar este servicio.

3.7.a. Conceptos de AOP.

No se puede tratar Spring sin uno de sus puntos más importantes, la programación orientada a aspectos. Esta tecnología tiene sus propios términos. AOP se base en el concepto de aspecto. Los aspectos tienen un propósito es decir un trabajo para el que se han creado. Al propósito de un aspecto se le denomina consejo.

1. Consejo. Se refiere al propósito de una tarea. Involucra tanto a lo qué debe hacerse como al cuándo debe hacerse. De esta forma distinguimos cinco tipos de consejos:
   - Antes, la funcionalidad del consejo tiene lugar antes de que el método se ejecute.
   - Después.
   - Después de la devolución. La funcionalidad del consejo tiene lugar después de que el método se ejecute con éxito.
   - Después de producir error. La funcionalidad del consejo tiene lugar después de que el método produzca un error.
   - Alrededor. El consejo encapsula al método.

2. Punto de cruce. Son los puntos de ejecución de una aplicación en la que se puede conectar un aspecto.
3. Punto de corte. Permiten reducir los puntos de cruce aconsejados por un aspecto. El punto de corte define dónde se debe aplicar una tarea. Una definición de punto de corte coincide con uno o más puntos de cruce.
4. Aspecto. Combina los consejos y los puntos de corte. Es decir qué hacer, dónde hacerlo y cuándo debe hacerse.
5. Entrelazado. El entrelazado es el proceso de aplicar aspectos a un objeto de destino para crear un nuevo objeto proxy. Los aspectos se entrelazan en el objeto de destino en los puntos de cruce especificados. El entrelazado puede tener lugar en diferentes puntos a lo largo de la vida útil del objeto: en tiempo de compilación, en tiempo de carga o en tiempo de ejecución.

Respecto a la compatibilidad de Spring con AOP cuenta con cuatro variantes:

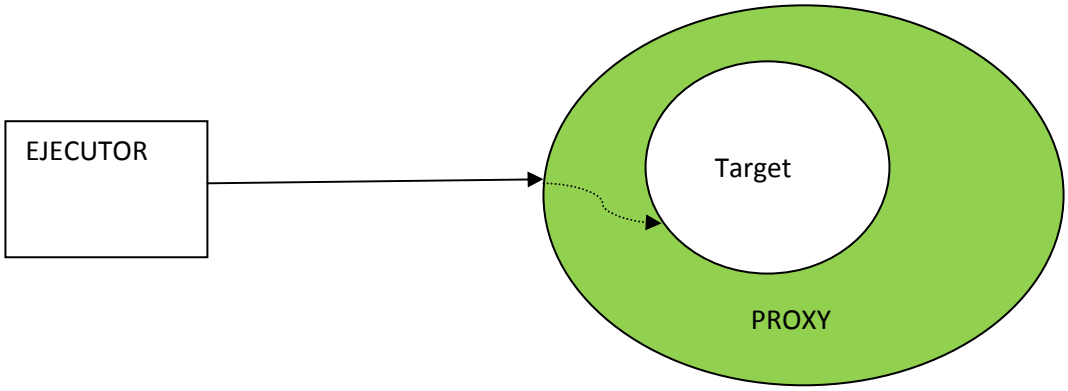
- AOP basado en proxy de Spring clásico.
- Aspectos basados en anotaciones de @AspectJ.
- Aspectos POJO puros.
- Aspectos AspectJ inyectados.

La compatibilidad de Spring con AOP se limita a la intercepción de métodos. Si se necesita intercepción de constructores o de propiedades se necesitará AspectJ.

3.7.b. Spring aconseja los objetos en tiempo de ejecución.

En Spring los aspectos se entrelazan con bean administrados por Spring durante el tiempo de ejecución, empaquetándolos en una clase proxy.

En la figura la clase proxy actúa como el bean objetivo, interceptando las ejecuciones de los métodos aconsejados y reenviando esas ejecuciones al bean de destino. Los aspectos de Spring se implementan como un proxy que empaqueta el objeto de destino



El proxy lleva a cabo la lógica de los aspectos entre el momento en el que proxy intercepta la ejecución del método y el momento que se invoca el método del bean de destino.
Spring sólo admite puntos de cruce de método.

3.7.c. Selección de puntos de cruce con puntos de corte.

En AOP de Spring los puntos de corte se definen usando un lenguaje de expresión de puntos de corte de AspectJ.
Tabla con los designadores de puntos de corte disponibles en Spring.

Designador	Descripción
args()	Limita las coincidencias del punto de cruce a la ejecución de aquellos métodos cuyos argumentos son instancias de los tipos dados.
@args()	Limita las coincidencias del punto de cruce a la ejecución de aquellos métodos cuyos argumentos se anotan con los tipos de anotación proporcionados.
execution()	Hace coincidir los puntos del cruce que son ejecuciones de un método.
this()	Limita las coincidencias del punto de cruce a aquéllas en las que la referencia del bean del proxy AOP es de un tipo determinado.
target()	Limita las coincidencias del punto de cruce a aquéllas en las que el objeto objetivo es de un tipo determinado.
@target()	Limita la coincidencia de los puntos de cruce cuando la clase de un objeto en ejecución cuenta con una anotación del tipo dado.
within()	Limita la coincidencia de los puntos de cruce dentro de ciertos tipos.
@ within()	Limita la coincidencia de los puntos de cruce dentro de ciertos tipos que cuenten con la anotación dada.
@annotation	Limita la coincidencia de los puntos de cruce a aquéllos en los que el asunto del punto de cruce cuenta con la anotación dada.

Escribir puntos de corte.

Ejemplo de punto de cruce que aplica un consejo siempre que se ejecute un método Instrument.play()
execution(* com.springinaction.springidol.Instrument.play(..))

- execution-> indica que se active con la ejecución del método.
- * -> indica que devuelve cualquier tipo.

- com.springinaction.springidol.Instrument-> tipo al que pertenece el método.
- play-> método.
- (..) -> indica que acepta cualquier argumento.

Ejemplo se quiere limitar el ámbito de ese punto de cruce sólo al paquete com.springinaction.springidol:

execution(* com.springinaction.springidol.Instrument.play(..)) && within(com.springinaction.springidol.*)

- && -> operador and.
- within(com.springinaction.springidol.*) -> limita a cuando el método se ejecuta dentro de cualquier clase del paquete com.springinaction.springidol

Ejemplo de limitación de un punto de corte a un bean específico. Para ello se usa el designador bean()

execution(* com.springinaction.springidol.Instrument.play(..)) and bean(eddie)

Se aplica el consejo del aspecto a la ejecucion de Instrument.play() limitándolo al bean cuyo ID sea Eddie.

3.7.d. Declarar aspectos en XML.

Se usa el espacio de nombres de configuración aop cuyos elementos se resumen en la siguiente tabla

Elemento AOP	Función
<aop:advisor>	Define un asesor AOP
<aop:after>	Define un AOP después de un consejo.
<aop:after-returning>	Define un consejo AOP devuelto
<aop:after-throwing>	Define un consejo AOP lanzado
<aop:around>	Define un consejo AOP alrededor
<aop:before>	Define un consejo AOP antes
<aop:config>	El elemento aop de nivel superior
<aop:declare-parents>	Introduce interfaces adicionales para objetos aconsejados
<aop:pointcut>	Define un punto de cruce.
<aop:aspect>	Define un aspecto

Ejemplo del uso de aspectos. Con una clase Audience y una interface Performer

```
public interface Performer {
  void perform() throws PerformanceException;
}
```

```
public class Audience {
  public void takeSeats() { //<co id="co_takeSeats"/>
    System.out.println("The audience is taking their seats.");
  }

  public void turnOffCellPhones() { //<co id="co_turnOffCellPhones"/>
    System.out.println("The audience is turning off their cellphones");
  }

  public void applaud() { //<co id="co_applaud"/>
    System.out.println("CLAP CLAP CLAP CLAP CLAP");
  }
```

```
  public void demandRefund() { //<co id="co_demandRefund"/>
    System.out.println("Boo! We want our money back!");
  }
}
```

Se declara de manera habitual:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Configuración de los aspectos con Spring.

```
<aop:config>
  <aop:aspect ref="audience"><!--<co id="co_refAudienceBean"/>-->

    <aop:before pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="takeSeats" /> <!--<co id="co_beforePointcut"/>-->

    <aop:before pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="turnOffCellPhones" /> <!--<co id="co_beforePointcut2"/>-->

    <aop:after-returning pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="applaud" /> <!--<co id="co_afterPointcut"/>-->

    <aop:after-throwing pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="demandRefund" /> <!--<co id="co_afterThrowingPointcut"/>-->

 </aop:aspect>
</aop:config>
```

Los elementos se han declarado dentro de <aop:config>.
Con <aop:aspect> se ha declarado el aspecto aunque se podían haber declarado varios y con el atributo ref se indica el bean POJO que se va a utilizar para proporcionar la funcionalidad del aspecto, en este caso audience. El bean al que se hace referencia con el atributo ref va a proporcionar los métodos ejecutados por cualquier consejo en el aspecto.

El aspecto cuenta con cuatro consejos:

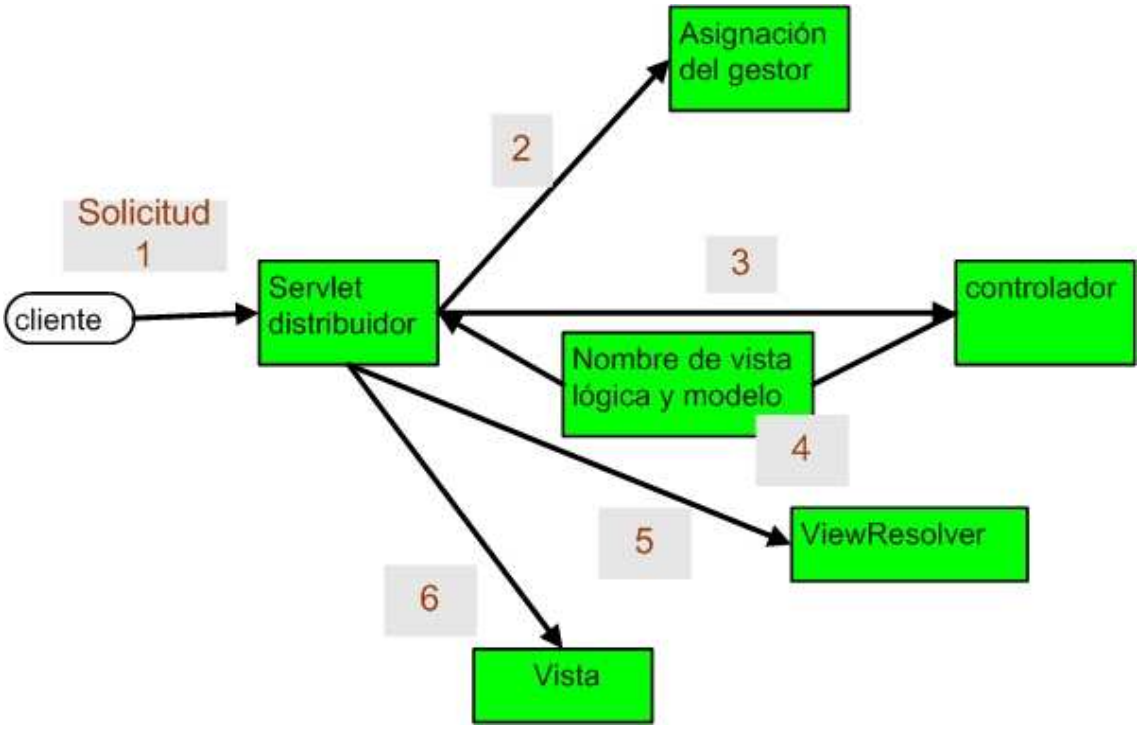
Los dos elementos <aop:before> que definen el método antes del consejo que va a ejecutar los métodos take-Seats y turnOffCellPhones() del bean audience, que se declaran con el atributo method, antes de que se ejecute cualquier método que coincida con el punto de cruce.
El elemento <aop:after-returning > que define un consejo tras la ejecución para ejecutar el método applaud().
El elemento <aop:after-thowing>> define un consejo después de generarse una excepción para ejecutar refund().
Con el atributo poincut se define el punto de cruce, en este caso es el mismo para los cuatro consejos, cuando se ejecute el método perform()de la clase que implemente la interfaz Performer.

En resumen, la lógica del negocio es performer que no sabe nada de los aspectos que se le aplican declarativamente.
Por último indicar que Spring 3 también permite utilizar anotaciones para crear aspectos.

3.8. Funcionamiento de una solicitud en Spring. Modelo MVC.

Nota respecto a otros patrones MVC he encontrado un poco confusa los términos usados en la descripción, los aclaro en la explicación.
Cuando un cliente realiza una petición, primero la trata el DispatcherServlet de Spring que actúa de controlador frontal, es decir todas las peticiones se canalizan por este servlet.

Esquema de una petición en Spring según los términos usados en el mismo.



La función de DisptacherServlet es enviar es enviar la solicitud a un controlador de Spring MVC. El controlador es el componente de Spring que procesa la solicitud y que generalmente delega la lógica de negocio a otros componentes y determina el nombre de la vista que debe generar el resultado por eso lo incluyo dentro del modelo.
Una aplicación puede tener varios controladores, siendo DispatcherServlet el que determina el mismo. Para realizarlo consulta el componente "gestor de asignaciones". El gestor de asignaciones formará parte del controlador en el modelo MVC.

Una vez seleccionado el controlador adecuado, DispatcherServlet envía la solicitud al mismo aunque después el controlador generalmente delegará la lógica de negocio a otros componentes. Una vez obtenida la información solicitada (esta información de negocio es la que denomina modelo en la figura) , la empaqueta, determina la vista que debe generar el resultado y envía la solicitud, el modelo (la información) y el nombre de la vista al DispatcherServlet.

El modelo estará constituido por los objetos a los que delega el controlador y llaman a la lógica de negocio y la información devuelta.

Las vistas generalmente estarán formadas por páginas JSP pero no tiene porque ser así, DispatcherServlet cuenta con un nombre lógico que se va a utilizar para examinar la vista que va a generar el resultado. DispatcherServlet consulta un ViewResolver para a hacer corresponder el nombre lógico con una implementación de vista específica.
Una vez determinada la vista, está usa los datos del modelo para generar el resultado que se mostrará al cliente.

3.9. Configurar Spring.

9. a. Configuración de Spring basada en XML.

En la explicación anterior se puede observar que el núcleo de Spring MVC es DispatcherServlet. Este se configura en web.xml como en otras aplicaciones web. Por ejemplo en el caso de un servlet denominado spitter.

```
<servlet>
        <servlet-name>spitter</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
 </servlet>
<servlet-mapping>
        <servlet-name>spitter</servlet-name>
        <url-pattern>/</url-pattern>
</servlet-mapping>
```

Nota: servlet-name es importante porque Spring de forma predeterminada cuando carga DispatcherServlet , va a cargar el contexto de aplicación de Spring desde un archivo xml, cuyo nombre se basa en el nombre del servlet. En este caso spitter-servlet.xml.

Análogamente a otras aplicaciones web con servlet-mapping se indica que URL van a ser gestionadas por DispatcherServlet. Lo habitual es encontrar este asignado con patrones URL como:

- *.htm, con este patrón se indica que la respuesta es en formato html.
- /*, con este patrón se indica que DispatcherServlet gestionará todas las solicitudes.
- /app, para ayudar a distinguir los tipos de contenido.

En lugar de estos patrones, DispatcherServlet se asigna con /. Con este patrón se indica que es el servlet predeterminado y gestiona todas las solicitudes, incluyendo contenido estático.

Para evitar que DispatcherServlet gestione el contenido estático se usa uno de los elementos de los espacios de nombres anteriormente indicados, el <mvc:resources> que gestiona las solicitudes de contenido estático. Se puede observar la configuración en spitter-servlet.xml que define el contexto de la aplicación.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:util="http://www.springframework.org/schema/util"
 xmlns:mvc="http://www.springframework.org/schema/mvc"
 xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
```

```
            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.0.xsd
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <!--<start id="mvc_resources"/>-->
   <mvc:resources mapping="/resources/**" location="/resources/" />
   <!--<end id="mvc_resources"/>-->

</beans>
```

Con <mvc:resources> se configura un gestor para proporcionar contenido estático. El atributo mapping se configura con /resources/** que indica que la ruta debe comenzar por /resources e incluirá cualquier subdirectorio.

9. b Configuración de Spring basada en anotaciones.

Para activar las características de Spring basadas en anotaciones se debe incluir en spitter-servlet.xml el siguiente elemento:

<mvc:annotation-driven/>

3.10. Uso de controladores.

Un buen diseño es escribir controladores orientados a recursos, tal como se indicaba en el punto de funcionamiento de una petición Spring. Así que generalmente tendremos los recursos propios del dominio de la aplicación y controladores generales como el de la página de inicio y el redirección, que son necesarios y no asignados a conceptos del dominio.

Tal como se ha indicado en el proceso de la petición, Spring consulta con un gestor de asignación para decidir a qué controlador enviar la solicitud. Spring cuenta con una serie de implementaciones de gestores de asignación, se muestra una tabla con los mismos.

Si se usa una de estas implementaciones, sólo se tendrá que configurarla como un bean de Spring. Pero si no encuentra un bean de este tipo, DispatcherServlet usará BeanNameUrlHandlerMapping y DefaultAnnotationHandlerMapping.

Gestor de asignación	Descripción
BeanNameUrlHandlerMapping	Asigna controladores a URL en función de los nombres de bean de los controladores
ControlerBeanNameHandlerMapping	Similar a la anterior pero los nombres de los bean no tienen que seguir las convenciones de las URL.
ControlerClassNameHandlerMapping	Asigna controladores a URL utilizando los nombres de clase de los controles como base para sus URL.
DefaultAnnotationHandlerMapping	Asigna una solicitud a un controlador y métodos de controlador anotados con @RequestMapping.
SimpleUrlHandlerMapping	Asigna controladores a URL utilizando una colección de propiedades definida en el contexto de aplicación de Spring.

Ejemplo del controlador de página de inicio.

Se observa que el método showHomePage devuelve un string que es el nombre lógico de la vista que debe procesar los resultados.

- @Controller-> indica que es una clase de controlador. Con esta anotación la clase queda registrada como bean por component-scan. Nota se debe realizar en spitter-servlet.xml <context:component-scan base-package="com.habuma.spitter.mvc" />
- @RequesMapping-> Identifica a showHomePage como gestor de solicitudes cuya ruta sea / o /home.
- @Inject-> Se usa para conectar de forma automática propiedades, métodos y constructores. Se espera que las dependencias anotadas con inject se cumplan, en caso contrario da error.

```
@Controller
public class HomeController {

  private SpitterService spitterService;

  @Inject
  public HomeController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }

  @RequestMapping(value={"/","/home”})

  public String showHomePage(Map<String, Object> model) {
    model.put("spittles",
              spitterService.getRecentSpittles(spittlesPerPage));
    return "home";
  }

  //<start id="spittlesPerPage"/>
  public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

  private int spittlesPerPage = DEFAULT_SPITTLES_PER_PAGE;

  public void setSpittlesPerPage(int spittlesPerPage) {
    this.spittlesPerPage = spittlesPerPage;
  }
  public int getSpittlesPerPage() {
    return spittlesPerPage;
  }
  //<end id="spittlesPerPage"/>
}
```

3.11. Resolución de vistas.

Por último al gestionar una solicitud se debe procesar el resultado para el usuario. Esta función está encomendada a la implementación de la vista. Para determinar qué vista va a gestionar una solicitud dada DispatcherServlet consulta con un solucionador de vista para intercambiar el nombre de vista lógico, proporcionado por un controlador, por una vista real que procese los resultados.
Lo que hace el solucionador es asignar el nombre de vista lógico con una implementación de org.springframework.Web.servlet.View.

En la tabla siguiente se muestran varias implementaciones de solucionador de vista. Aunque Spring también tiene solucionadores para Freemarker, JasperReports, Tiles y Velocity.

Solucionador de la vista	Descripción
BeanNameViewResolver	Busca una implementacion de view que este registrada como un <bean> y cuyo ID sea el mismo que el del nombre de la vista logica.
ContentNegotiatingViewResolver	Delega en uno o más solucionadores de vista la elección de cual se basa en el tipo de contenido solicitado.
InternalResourceViewResolver	Busca una plantilla de vista almacenada en el archivo WAR de la aplicación Web. La ruta para la plantilla de vista viene determinada por los prefijos y sufijos del nombre de vista lógico.
ResourceBundleViewResolver	Busca implementaciones view para un archivo de propiedades
UrlBasedViewResolver	La clase básica para algunos solucionadores de vista, como por ejemplo InternalResourceViewResolver. Puede utilizarse de forma individual, aunque no es tan potente como sus subclases. Por ejemplo, UrlBasedViewResolver no puede resolver vistas basadas en la ubicación actual.
XmlViewResolver	Busca una implementación de view que se de-clara como un <bean> en un archivo XML (/WEB-INF/views. xml). Este solucionador de vista es muy similar a BeanNameViewResolver, con la diferencia de que los <bean> de la vista se de-claran por separado de los del contexto de aplicación de Spring.
XsltViewResolver	Resuelve una vista basada en XSLT, donde la ruta de la hoja de estilo XSLT se deriva de los prefijos y sufijos del nombre de vista lógico

Ejemplo de resolución de vistas internas con InternalResourceViewResolver, en este ejemplo se han colocado los jsp en la carpeta **/WEB-INF/views**.

Como se ha comentado se encarga de resolver un nombre de vista lógico en un objeto View que está situado en el contexto de la aplicación web. Para realizarlo rodea el nombre de la vista lógica con un sufijo y un prefijo para llegar a la ruta de la plantilla.

La configuración de InternalResourceViewResolver en spitter-servlet.xml quedaría:

```
<bean class=
         "org.springframework.web.servlet.view.InternalResourceViewResolver">
 <property name="prefix" value="/WEB-INF/views/"/>
 <property name="suffix" value=".jsp"/>
</bean>
```

De esta forma cuando DispatcherServlet solicita a InternalResourceViewResolver que resuelva una vista, este añade al sufijo y prefijo al nombre, resultando la ruta de un .jsp que se encarga de procesar el resultado.

Si se usaran etiquetas JSTL se podría usar otro objeto JstlView configurando el atributo viewClass, por ejemplo:

```
<bean class=
         "org.springframework.web.servlet.view.InternalResourceViewResolver">
 <property name="viewClass"
     value="org.springframework.web.servlet.view.JstlView" />
 <property name="prefix" value="/WEB-INF/views/"/>
 <property name="suffix" value=".jsp"/>
</bean>
```

3.12. Estructuración de los ficheros xml de configuración.

Aunque no es obligatorio, un buen diseño de aplicación divide el fichero de configuración xml en capas, se generan ficheros xml por cada capa, uno para la capa de persistencia, otro para la capa de configuración del origen de datos otro para la capa de presentación web….
En los ejemplos para la configuración se cargaba el fichero spitter-servlet.xml a partir del nombre del servlet.
Para poder cargar el resto de ficheros de configuración se usa ContextLoaderListener, un agente de escucha de servlet que carga configuraciones adicionales en un contexto de aplicación de Spring, junto con un contexto de aplicación creado por DispatcherServlet.

Para usarlo:

1. debe definir en web.xml:

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

2. Especificar los archivos de configuración para que los cargue, para ello se configura el parámetro contextConfigLocation en web.xml

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spitter-security.xml
            classpath:service-context.xml
            classpath:persistence-context.xml
            classpath:dataSource-context.xml
        </param-value>
 </context-param>
```

El prefijo classpath se usa porque se cargan estos xml desde la ruta de la clase de la aplicación.

3.13. Spring Web Flow.

Por último mencionar Spring Web Flow que es un framework Web que permite el desarrollo de elementos que siguen un flujo configurado. Es una extensión de Spring MVC, separa la definición de flujo de una aplicación de las clases y vistas que implementan su comportamiento.

4. JAVA SERVER FACES 2.

4.1. Introducción.

Información extraída de la universidad de Málaga.

El objetivo de la tecnología JSF es desarrollar aplicaciones web de forma parecida a como se construyen aplicaciones locales con Java Swing, AWT, SWT o APIs similares.

JavaServer Faces pretende facilitar la construcción de estas aplicaciones proporcionando un entorno de trabajo (*framework*) vía web que gestiona las acciones producidas por el usuario en su página HTML y las traduce a eventos que son enviados al servidor con el objetivo de regenerar la página original y reflejar los

cambios pertinentes provocados por dichas acciones. En definitiva, se trata de hacer aplicaciones Java en las que el cliente no es una ventana de la clase **JFrame** o similar, sino una página HTML.

4.1.1. JSF 2.0 breve historia.

JSF es una especificación publicada por la Java Community Process (JCP) y fue creado en el 2001 como JSR 127. En 2006 JSF 1.2 se introdujo en java Enterprise como JSR 252 (con Java EE 5) . En esta versión se usaba JSP pero no se integraba bien con JSF. Las páginas JSP y JSF tienen diferentes ciclos de vida, por lo que se introdujeron los Facelets como alternativa a JSP. No quiere decir que no se pueda usar JSP pero no tendrán todas las características de las que se dispone con Facelets (páginas XHTML).
JSF 2 se desarrolló en la JSR 314 y es la opción preferida para el desarrollo web en Java EE 6. JSF 2 se ha inspirado en muchos web open source frameworks y dispone de muchas de sus características.

4.2. Arquitectura y componentes de una aplicación JavaServer Faces .

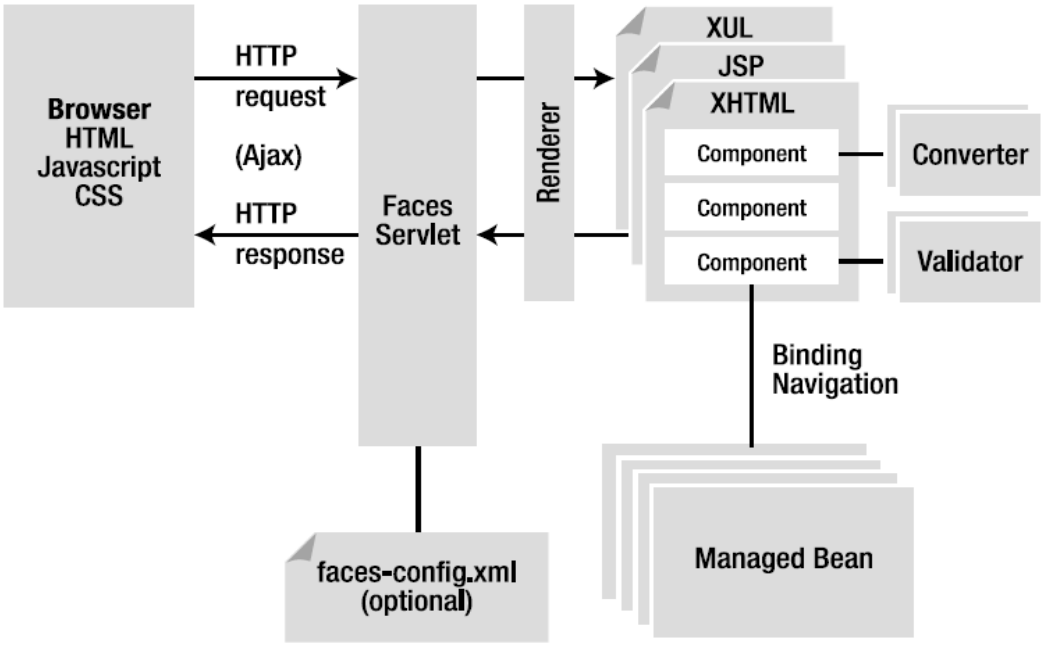


Figure 10-1. JSF architecture

Imágen de la arquitectura JSF a un alto nivel de abstración.
Información y figura extraída de Beginning Java™ EE 6 Platform with GlassFish™ 3, Second Edition de Antonio Goncalves.

Como se observa en la figura, las aplicaciones JSF son aplicaciones web estándar que interceptan peticiones HTTP vía el servlet Faces y producen HTML. La arquitectura JSF permite usar diferentes lenguajes de declaración de páginas, ser traducidos y visualizados por distintos dispositivos y crear páginas usando, componentes, eventos y listerners como Swing. Análogamente a Swing, JSF dispone de un conjunto de UI herramientas (botones, check boxes, campos de texto….) y permite una fácil conexión de componentes creados por terceros.

4.2.a. Componentes.

Como se observa en la figura, las piezas importantes de la arquitectura JSF son:

- **FacesServlet y faces-config.xml**
- **Páginas y componentes**.
- **Renderers**.
- **Converters.**
- **Validators.**
- **Managed bean y navegación.**
- **Soporte Ajax.**

4.2.b. FacesServlet y faces-config.xml.

JSF implementa el patrón MVC para desacoplar la vista (páginas) y el modelo (datos) que visualizaran en la vista. El controlador gestiona las acciones de usuario que podrían cambiar el modelo y actualiza las vistas. En JSF el controlador es un servlet denominado FacesServlet. Todas las peticiones de usuario van a través de FacesServlet, las examina y realiza acciones sobre el modelo a través de managed beans . FacesServlet como parte interna de JSF se debe configurar a través de metadatos; se puede configurar mediante el archivo descriptor faces-config.xml o desde JSF 2 también se puede configurar mediante anotaciones. Se dedica todo un apartado a la explicación del MVC en JSF.

4.2.c. Páginas y componentes.

JSF permite múltiples PDL (lenguajes de descripción de páginas) tales como Java Serves Pages y Facelets que es el recomendado desde JSF 2.

JSF tiene que enviar una página al dispositivo de salida del cliente y requiere algún tipo de tecnología de visualización conocida como PDL (o también VDL lenguaje de declaración de vistas).

Tanto las páginas JSP como Facelets se construyen como un árbol de componentes, también denominados widgets o controls, que proveen la funcionalidad para interactuar con el usuario final, son los botones, campos de texto….JSF también permite crear componentes propios.

La página pasa a través de un complejo ciclo de vida que maneja el árbol de componentes, inicialización, eventos, visualización….Se explica el ciclo más adelante.

4.2.d. Renderers.

Son los responsables de visualizar los componentes y transformar una entrada de usuario en valores de una propiedad de un componente. Es como un traductor entre cliente y servidor: decodifica la petición del usuario en un conjunto de valores del componente y codifica la respuesta para crear una representación del componente que el cliente pueda entender y visualizar.

Para asegurar la portabilidad de la aplicación, JSF incluye soporte para renderers de HTML 4.01, pero las implementaciones de JSF pueden crear su propio render kit para generar Wireless Markup Language WML, Scalable Vector Graphics SVGs y más.

4.2.e. Converters y Validators.

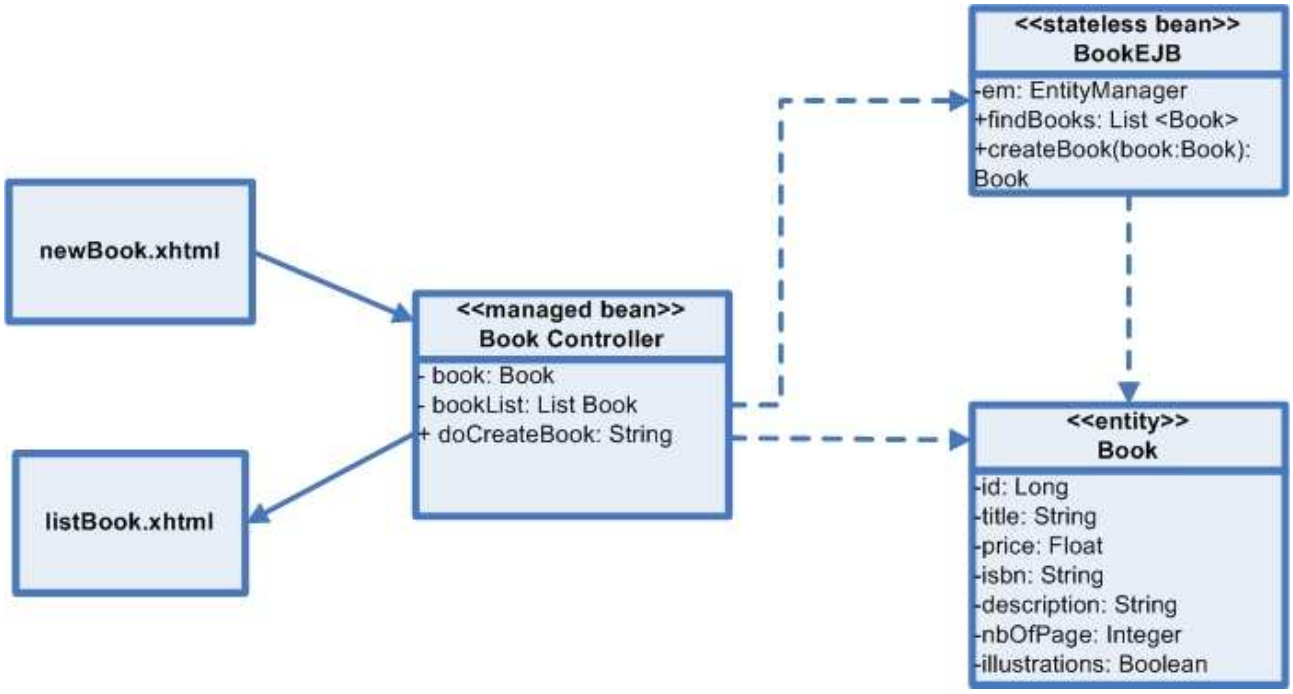
Los converters transforman las cadenas de HTML a un objeto (Date, Boolean etc) y al revés. Los validators son los responsables de asegurar que los valores introducidos por el usuario sean correctos. Se puede asociar más de un validator a un componente. JSF viene con un conjunto incorporado de validadores y permite crear los propios.

Naturalmente cuando se produce un error de conversión o de validación se envía un mensaje de respuesta.

4.2.f. Managed bean y navegación. Lenguaje de expresiones EL.

Un managed bean es una clase de java especializada que sincroniza valores de los componentes, procesa la lógica de negocio (generalmente se traducirá a llamadas a EJB de sesión) y controla la navegación entre páginas.

Para las explicaciones posteriores y de este apartado se ha tomado un sencillo ejemplo pero que reúne muchas de las características de JSF. El ejemplo se ha adaptado del libro de Antonio Goncalves. En el libro se ejecuta con Maven pero yo lo he ejecutado desde eclipse Indigo sobre un servidor glassfish 3.



En este ejemplo se usaba una entidad Book en la capa de persistencia para almacenar los libros en una base de datos Derby y un Enterprise Java Bean en la capa de sesión, un bean de sesión sin estado BookEJB . BookEJB utiliza un entitymanager que gestiona las entidades en una base de datos. A efectos de este PFC lo que interesa es que el managed bean Book controler delega la lógica de negocio en llamadas a BookEJB de la capa de negocio y en los atributos de la entidad Book que usarán en las páginas xhtml y el managed bean Book Controller.

El código del managed bean Book Controller:

```
package controller;

@ManagedBean
@RequestScoped
public class BookController {
    // =             Attributes            =
    @EJB
    private BookEJBLocal bookEJB;
```

```
    private Book book = new Book();
    private List<Book> bookList = new ArrayList<Book>();
    // =           Public Methods           =
    public String doNew() {
        return "newBook.xhtml";
    }

    public String doCreateBook() {
        book = bookEJB.createBook(book);
        bookList = bookEJB.findBooks();
        return "listBooks.xhtml";
    }

    // =          Getters & Setters         =
    public Book getBook() {
        return book;
    }
    public void setBook(Book book) {
        this.book = book;
    }
    public List<Book> getBookList() {
        return bookList;
    }
    public void setBookList(List<Book> bookList) {
        this.bookList = bookList;
    }
}
```

El código de listBook.xhtml:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>List of the books</title>
</h:head>
<h:body>
    <h1>List of the books</h1>
    <hr/>
    <h:dataTable value="#{bookController.bookList}" var="bk" border="1">
        <h:column>
            <f:facet name="header">
                <h:outputText value="ISBN"/>
            </f:facet>
            <h:outputText value="#{bk.isbn}"/>
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Title"/>
            </f:facet>
            <h:outputText value="#{bk.title}"/>
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Price"/>
            </f:facet>
            <h:outputText value="#{bk.price}"/>
        </h:column>
        <h:column>
            <f:facet name="header">
```

```
                <h:outputText value="Description"/>
            </f:facet>
            <h:outputText value="#{bk.description}"/>
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Number Of Pages"/>
            </f:facet>
            <h:outputText value="#{bk.nbOfPage}"/>
        </h:column>
    </h:dataTable>
    <h:form>
  <h:commandLink action="#{bookController.doNew}">Create a new book
  </h:commandLink>
    </h:form>
    <hr/>
    </h:body>
</html>
```

El código de createBook.xhtml:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head> <title>Creates a new book</title> </h:head>
<h:body>
    <h1>Create a new book</h1>
    <hr/>
    <h:form>
        <table border="0">
            <tr>
                <td> <h:outputLabel value="ISBN : "/> </td>
                <td>
                    <h:inputText value="#{bookController.book.isbn}"/>
                </td>
            </tr>
            <tr>
                <td> <h:outputLabel value="Title :"/> </td>
                <td>
                    <h:inputText value="#{bookController.book.title}"/>
                </td>
            </tr>
            <tr>
                <td> <h:outputLabel value="Price : "/> </td>
                <td>
                    <h:inputText value="#{bookController.book.price}"/>
                </td>
            </tr>
            <tr>
                <td> <h:outputLabel value="Description  : "/>
               </td>
                <td>
<h:inputTextarea value="#{bookController.book.description}"cols="20" rows="5"/>
                </td>
            </tr>
            <tr>
                <td>
                    <h:outputLabel value="Number of pages : "/>
                </td>
                <td>
                    <h:inputText value="#{bookController.book.nbOfPage}"/>
```

```
                </td>
            </tr>
        </table>
<h:commandButton value="Create a book" action="#{bookController.doCreateBook}"/>
</h:form>
</h:body>
</html>
```

4.2.f.1. Managed bean y navegación conexión mediante el Lenguaje de expresiones EL.

Para sincronizar los valores de los componentes se debe asociar un componente con una propiedad o con una acción del managed bean usando el Expression Language EL. Un ejemplo de un trozo de código de una página xhtml y del managed bean bookController:

```
<h:inputText value="#{bookController.book.isbn}"/>
<h:commandButton value="Create" action="#{bookController.doCreateBook}"/>

@ManagedBean
public class BookController {
@EJB
private BookEJB bookEJB;

private Book book = new Book();

public String doCreateBook() {
      book = bookEJB.createBook(book);
     return "listBooks.xhtml";
}
// Getters, setters
}
```

La primera línea de código conecta una entrada de texto a la propiedad book.isbn del managed bean denominado bookController (book es una entidad definida en la capa de persistencia que tiene el atributo isbn). El valor de la entrada de texto está sincronizado con la propiedad book.isbn.
Los managed bean también controlan eventos, la segunda línea asocia un botón con una acción, al pulsar este botón se dispara un evento al bean donde un listener event se ejecuta, en este caso doCreateBook.
Convertir una clase POJO en un managed bean es bastante sencillo sólo basta utilizar la anotación **@ManagedBean.** Comentar que la anotación **@EJB** es un ejemplo del patrón de inyección de dependencias, aquí se inyecta un EJB de la capa de negocio denominado bookEJB.
Respecto a la navegación también es bastante sencillo, si el método doCreateBook finaliza bien, el framework va a la página indicada en el return de doCreateBook, **listBooks.xhtml**.

4.2.f.2. Expression Language.

Inicialmente JSF utilizaba JSP. JSP usa los siguientes elementos de scripting para embeber código java dentro de JSP:

- **Elementos de directiva, <**%@ directive **attributes %>,** directive puede tomar los valores page, include y taglib.
- **<**%! **this is a declaration %>** para declarar variables.
- **<**% **this is a scriptlet %>** Dentro se incluye código java**.**
- **<**%= **this is an expression %>** para incluir expresiones java.

Mediante los estamentos EL se puede realizar las mismas acciones y es más fácil para los no programadores java. Además dispone de un conjunto de operadores matemáticos, lógicos y relacionales.
La sintaxis básica para un estamento EL es:**#{expresión}**

4.2.g. Soporte Ajax.

JSF 2 viene con soporte incorporado para Ajax . Usar Ajax permite actualizar solo pequeñas porciones de una página de forma asíncrona. El soporte Ajax ha sido añadido en forma de una librería JavaScrip (jsf.js) definida en la especificación y con etiquetas <f:ajax>.

4.3. Comentario sobre Facelets.

Cuando se creó JSF la idea era reusar JSP como el PDL principal. Ya se ha comentado que los ciclos de vida de las páginas de ambos no se ajustaban bien. Las etiquetas en JSP se procesan una vez de arriba abajo para generar la respuesta. JSF tiene un ciclo más complejo que se analiza en el siguiente punto. Aquí es donde entran en juego los facelets que se ajustan al ciclo de vida JSF.

Los Facelets que empezaron como una alternativa a JSP, no tiene un JSR y no forma parte de Java EE. Así Facelets fue un reemplazo para JSP y proveen una alternativa XHTML para páginas en una aplicación JSF.

Facelets ofrecen un modelo de programación más simple que JSP. Facelets viene con una librería de etiquetas para escribir interfaces de usuario y provee un soporte parcial de etiquetas JSTL, pero la característica clave de Facelets es su plantilla de página que es más flexible que la de JSP y permite crear componentes de usuario que se pueden usar en el modelo de árbol de componentes de JSF. La librería de etiquetas está definida por la URI **http://java.sun.com/jsf/facelets** y generalmente usa el prefijo **ui:.**

Tabla de etiquetas facelets.

ETIQUETA	DESCRIPCION
<ui:composition>	Define una composición que opcionalmente usa una plantilla.
<ui:component>	Crea un componente.
<ui:debug>	Captura información de debugging.
<ui:define>	Define contenido que se inserta en una página por una plantilla.
<ui:decorate>	Permite decorar algún contenido en una página.
<ui:fragment>	Añade un fragmento de una página.
<ui:include>	Encapsula y reusa contenido entre multiples páginas XHTLM, es similar a la directiva <jsp:include> de JSP.
<ui:insert>	Inserta contenido en una plantilla.
<ui:param>	Permite pasar parámetros a un fichero included (usando <ui:include> o a una plantilla.
<ui:repeat>	Define un bucle es como <c:forEach>
<ui:remove>	Borra contenido de una página.

Tabla de conjunto de etiquetas permitidas con facelets.

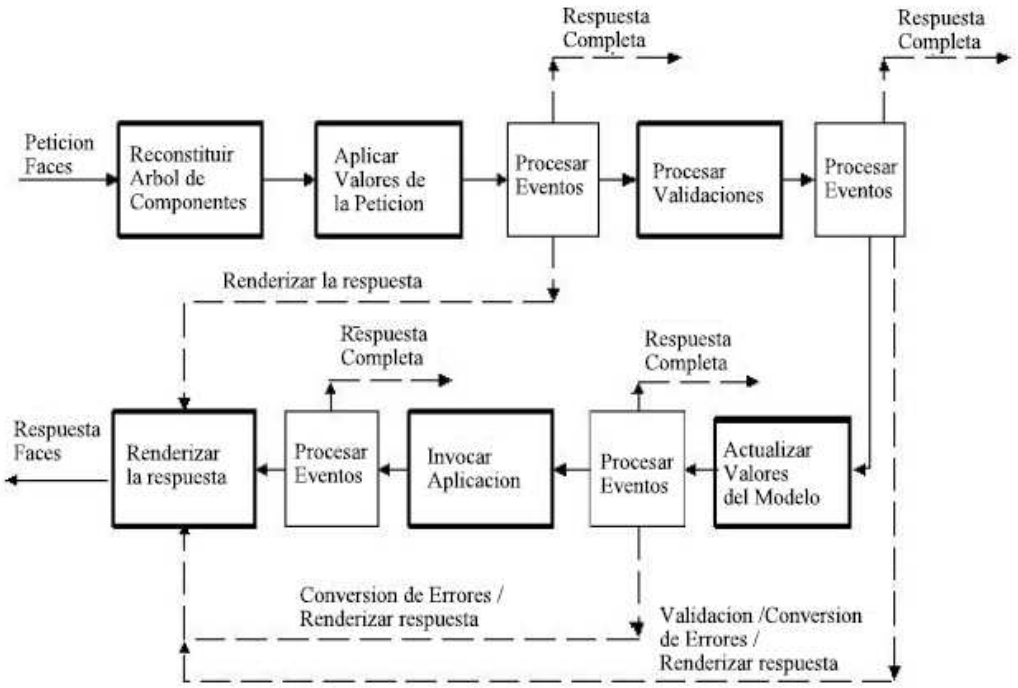
URI	Prefijo común usado	DESCRIPCION
http://java.sun.com/jsf/html	**h**	Esta librería de etiquetas contiene componentes y sus HTML renderers (**h:commandButton, h:commandLink**, **h:inputText**, etc)
http://java.sun.com/jsf/core	**f**	Esta librería contiene acciones del usuario que son independientes de cualquier rendering particular, validadores, convertidores ..(**f:selectedItem**, **f:validateLength, f:convertNumber**, etc)
http://java.sun.com/jsf/facelets	**ui**	Vista en la tabla anterior, estas etiquetas añaden soporte a las plantillas.
http://java.sun.com/jsf/composite	**composite**	Librería usada para definir componentes compuestos.
http://java.sun.com/jsp/jstl/core	**c**	Las páginas de facelets pueden usar algunas etiquetas de core JSP como (**<c:if/>**, **<c:forEach/>**, y **<c:catch/>**).
http://java.sun.com/jsp/jstl/functions	**fn**	Las páginas facelets pueden usar todas las funciones de las librerías de etiquetas JSP.

4.4. CICLO DE VIDA DE UNA PÁGINA.

Una página JSF es un arbol de componentes con un ciclo de vida específico, que consta de seis fases. Por ejemplo, al pulsar un botón hace que el browser envíe una petición al servidor. Esta petición se transforma en un evento que puede ser procesado por la lógica de la aplicación en el servidor. Todos los datos introducidos por el usuario entran en una fase de validación antes de que el modelo es actualizado y cualquier lógica de negocio invocada. JSF es responsable de asegurar que cualquier componente gráfico (hijos y componentes padres) es correctamente representado en el browser.

Con esta tecnología un desarrollador de aplicaciones JSF no necesitará preocuparse de los problemas de renderizado asociados con otras tecnologías UI. Así por ejemplo, si un cambio en un checkbox afecta a la apariencia de otro componente de la página, la tecnología JSF manejará este evento en la forma apropiada y no permitirá que se dibuje la página sin reflejar este cambio.

La siguiente figura extraída de un tutorial de la Universidad de Málaga muestra las diferentes fases del ciclo de vida de una página JSF, pero la explicación está extraída del libro de Antonio Goncalves.



1) **Reconstituir el árbol de componentes.**

JSF encuentra la vista correspondiente y aplica los valores introducidos al árbol. Si se trata de la primera visita a la página, JSF crea la vista como un componente **UIViewRoot** (componente raíz del árbol que constituye cualquier página). En caso de que no sea la primera visita, la previamente salvada UIViewRoot es obtenida para procesar la petición HTTP actual.

Así en esta fase JSF, construye el árbol de componentes de la página, conecta los manejadores de eventos y los validadores y graba el estado en el FacesContext (objeto usado por JSF comentado más adelante).

2) Aplicar los valores de la petición.

Una vez construido el árbol de componentes, se aplican a los mismos los valores que han llegado con la petición (sea desde campos de entrada en un formulario, desde cookies o desde la cabecera de la petición). Observar que sólo los componentes UI actualizan su estado y no los objetos de negocio que forman el modelo. Si la conversión falla se genera un mensaje de error asociado con el componente y se pone en la cola de FacesContext. Este mensaje se mostrará en la fase de renderizar la respuesta junto con mensajes que se pudieran producir en la fase de proceso de validación.

3) Proceso de validación.

Después de la fase anterior los componentes UI tienen un valor actualizado. En el proceso de validación, JSF recorre el árbol de componentes y para cada uno se asegura que el valor es aceptable, aplicando las reglas de validación a los atributos. Si ambas fases de conversión y validación son correctas, el ciclo va a la fase 4. En caso contrario, el ciclo va la fase de renderizar la respuesta con los mensajes de error apropiados, donde se dibujará de nuevo la página y los mensajes de error.

4) **Actualizar los valores del modelo.**

   Una vez se han actualizado y validados todos valores de los componentes, JSF permite que se actualicen los valores de los atributos de los managed beans asociados con los componentes.

5) **Invocación de la aplicación.**

   Ahora se realiza la lógica de negocio. Cualquier acción que se ha disparado se ejecutará sobre el managed bean adecuado y es donde la navegación se hace efectiva, ya que el return de la acción determinará la respuesta a renderizar. JSF configura el árbol de componentes de la respuesta y transfiere el control a la última fase.

6) **Renderizar la respuesta.**

   Se envía la respuesta al usuario y se salva el estado de la vista de manera que se pueda restaurar en la fase 1, si el usuario envía una nueva petición.

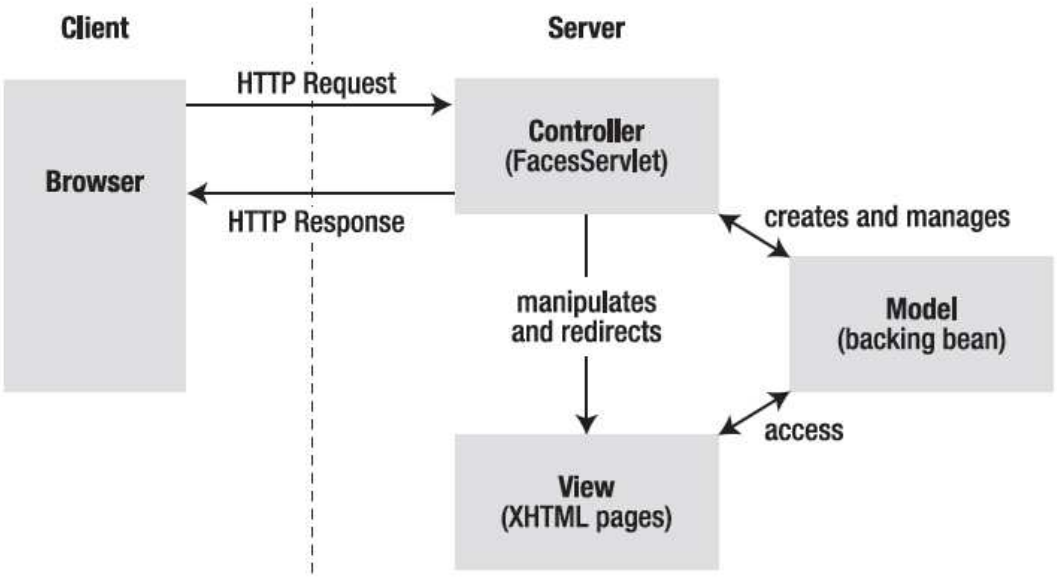
El thread de ejecución para un ciclo petición/respuesta puede ir a través de cada fase dependiendo de la petición y lo que ocurre durante el proceso de la misma. Tal como se ha indicado, si ocurre un error el flujo de ejecución se transfiere a la fase de renderizar la respuesta.

4.5 Patrón Modelo Vista Controlador en JSF 2.

JSF facilita la creación de interfaces de usuario, el control de la navegación de la aplicación y las llamadas a la lógica de negocio síncrona o asíncronamente y esto es posible porque JSF se construye aplicando el patrón Modelo Vista Controlador. Cada parte está separada de la otra, permitiendo que la interface de usuario cambie sin impactar en la lógica de negocio y al revés.

En MVC, el modelo representa los datos de la aplicación, la vista corresponde a la interface de usuario y el controlador gestiona la comunicación entre ambos.

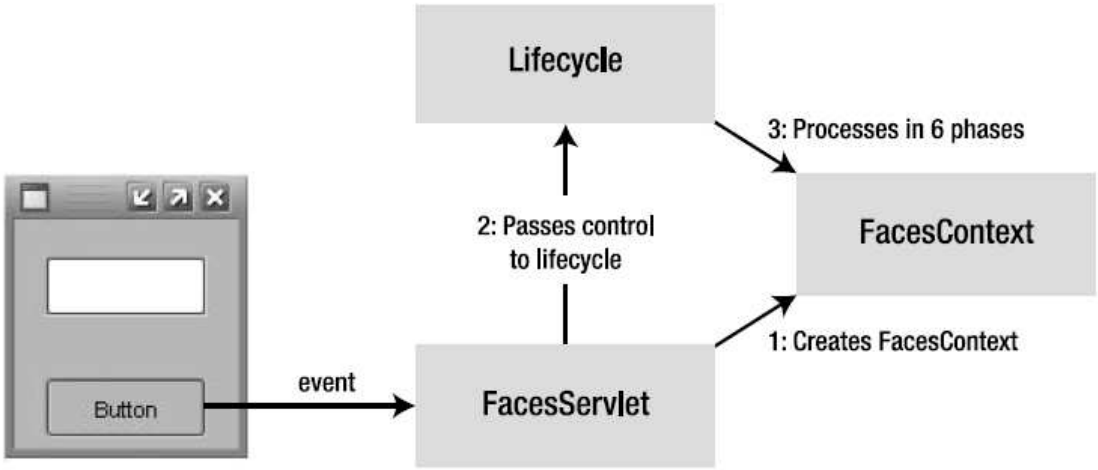
MVC en JSF. Figura extraída del libro de Antonio Goncalves, Beginning Java EE6 with GlassFish 3.



El modelo se construye sin preocuparse del diseño presentado al usuario. En JSF el modelo puede consistir en managed beans, llamadas a EJB de la capa de sesión, entidades de la capa de persistencia…..
La vista en JSF son las páginas XHTML (para interfaces web, pero podrían ser WML para dispositivos inalámbricos). La vista ofrece una representación gráfica del modelo. Un modelo podría tener varias vistas. La vista ya se ha comentado en los puntos anteriores.
Cuando un usuario manipula la vista, la vista informa al controlador de los cambios deseados. El controlador reúne, convierte y valida los datos, invoca la lógica de negocio y genera contenido en formato XHTML. Tal como se observa en la figura y ya se ha comentado, en JSF el controlador es el FacesServlet.

4.6.a. FacesServlet.

El FacesServlet es una implementación de javax.servlet.Servlet y actúa como el controlador central, a través del cual pasan todas las peticiones de usuario. Es decir **el framework JSF implementa un MVC modelo 2.**
En la siguiente figura extraída del mismo libro se observa que cuando por ejemplo un usuario pulsa un botón, la notificación del evento es enviada vía HTTP al servidor e interceptada por el javax.faces.webapp.FacesServlet. Este controlador examina la petición y realiza varias acciones sobre el modelo utilizando los managed beans.



Interacciones del FacesServlet. **Figura extraída del libro de Antonio Goncalves, Beginning Java EE6 with GlassFish 3.**

Tal como se observa en la figura, FacesServlet acepta las peticiones y maneja el control sobre el objeto javax.faces.lifecycle.Lifecycle. Además, usando una clase factoría crea el objeto FacesContex, que contiene toda la información relativa al estado de la petición.

Tal como se observa, FacesServlet crea un FacesContext para el thread actual y pasa el control al objeto Lifecycle, que utiliza al FacesContext en las 6 fases descritas en el apartado anterior sobre el ciclo de vida, antes de generar y devolver la respuesta.

Para que las peticiones sean procesadas por el FacesServlet, debe haber un elemento servlet mapping que lo indique en descriptor despliegue web.xml. Esto es análogo a lo realizado con la API de Servlets. Las páginas web, los managed beans … tienen que ser empaquetados con el fichero web.xml.
Se muestra un ejemplo simple del web.xml.

```
<?xml version='1.0' encoding='UTF-8'?>
```

```
<web-app version="2.5"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
</web-app>
```

En este caso el fichero web.xml define el javax.faces.webapp.FacesServlet asignándole el nombre Faces Servlet. Todas las peticiones que tengan la extensión .faces serán gestionadas por el Faces Servlet.

Dentro del fichero web.xml se puede indicar el elemento **<context-param>** y como hijo de este los elementos **<param-name>** que permiten configurar algunas características del framework JSF. Por ejemplo, con **javax.faces.DEFAULT_SUFFIX** se pueden definir una lista de sufijos alternativos para las páginas con contenido JSF, por ejemplo .xhtml.

Puesto que la implementación de FacesServlet es interna a JSF y no se tiene acceso a su código, se deben usar metadatos para configurarlo.

Desde JSF 2 existen dos posibles opciones para los metadatos mediante anotaciones o mediante ficheros descriptores de despliegue en XML, en este caso sería el:
/WEB-INF/faces-config.xml, del que se muestra un ejemplo del mismo.

Ejemplo de faces-config.xml.

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
        version="2.0">

  <application>
    <locale-config>
      <default-locale>fr</default-locale>
    </locale-config>
    <resource-bundle>
      <base-name>messages</base-name>
      <var>msg</var>
    </resource-bundle>
  </application>

<navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
```

```
        <from-outcome>doCreateBook-success</from-outcome>
        <to-view-id>/listBooks.htm</to-view-id>
      </navigation-case>
    </navigation-rule>
</faces-config>
```

Actualmente tanto los managed beans como los converters, event listeners, renderers y validators pueden usar y configurarse mediante anotaciones.

4.6.b. FacesContext.

Como ya se ha mencionado en los puntos anteriores, JSF define la clase abstracta javax.faces.context.FacesContext para representar la información asociada al procesado de una petición entrante y la creación de la correspondiente respuesta. Esta es la clase que permite la interacción con la interface de usuario y con el resto del entorno de JSF.

En las páginas se puede acceder a la misma desde la variable implícita facesContext y en los managed bean se puede obtener una referencia a la misma usando el método estático getCurrentInstance(). Una vez obtenida la instancia del FacesContext para el thread actual se pueden usar todos sus métodos, por ejemplo el método addMessage que agrega un mensaje (sea de información, aviso, error o fatal).

4.6.c. Managed Beans.

En la implementación del MVC en JSF los managed bean constituían el modelo, o la puerta al modelo si realmente realizan llamadas a los EJB de la capa de sesión.

Los managed bean son clases java con anotaciones y son elementos centrales en las aplicaciones web con JSF, son clases administradas por el Faces Servlet. Sus funciones son realizar la lógica de negocio (o delegar mediante llamadas a EJB de la capa de negocio) , controlar la navegación entre páginas y contener datos.

- **Backing Bean**. Una aplicación típica puede contener managed beans compartidos por varias páginas. Los datos se guardan en los atributos del managed bean, en este caso se les denomina "backing bean" (beans de respaldo). Los backing bean definen datos a los que un componente de la interface de usuario está enlazado. Para asociar los componentes de la interface de usario a un backing bean se deben usar expresiones del Expression Language. Asimismo también permite que se puedan invocar métodos definidos en el backing bean.

- **Modelo Bean**. Tal como ya se ha mostrado en un punto anterior para escribir un managed bean sólo se necesita la anotación @ManagedBean o su notación XML equivalente en el descriptor de despliegue faces-config.xml.

a. Muchas características vienen configuradas por defecto, como el ámbito que por defecto es @RequestScoped.
b. La clase debe ser pública, no debe ser ni final ni abstracta.
c. Debe tener un constructor sin argumentos que el framework usará para crear instancias.
d. La clase no debe definir el método finalize().
e. Los atributos deben tener getters y setters públicos.

Observar que b, c y e son requisitos típicos de cualquier bean. Así usando la configuración por defecto, un managed bean puede ser un POJO con la anotación @ManagedBean

4.6.c.1. Ámbito de los managed bean.

Los objetos que son creados como parte de un managed bean tienen un cierto tiempo de vida y podrían no estar accesibles para componentes de la interface de usuario u otros objetos en la aplicación web. El tiempo de vida y la accesibilidad de un objeto son conocidos como ámbito (scope). En la aplicación web se puede especificar el ámbito de un objeto usando seis tipos distintos de tiempo de vida.

- **Aplicación**. Es la duración menos restrictiva , con el mayor tiempo de vida. Los objetos creados están disponibles en todos los ciclos petición/respuesta para todos los clientes que usan la aplicación web. Atención estos objetos pueden ser llamados concurrentemente y necesitan ser thread seguros usando la instrucción synchronized. Los objetos con este ámbito pueden usar otros objetos sin ámbito o con ámbito de aplicación. Anotación: @ApplicationScoped

- **Sesión**. Los objetos están disponibles para cualquier ciclo petición / respuesta que pertenezca a la sesión del cliente. El estado de estos objetos persiste entre peticiones hasta que por último la sesión se invalida. Los objetos con este ámbito pueden usar objetos sin ámbito, con ámbito de sesión o con ámbito de aplicación. Anotación: (@SessionScoped).

- **Vista.** Los objetos están disponibles dentro de una vista dada hasta que la vista se cambia, el estado de los mismos persiste hasta que el usuario navega a una nueva vista, punto en el que se borran. Los objetos con este ámbito pueden usar otros objetos sin ámbito, ámbito de vista, de sesión o de aplicación. Anotación: @ViewScoped.

- **Petición.** Si no se indica lo contrario es el ámbito por defecto. Los objetos están disponibles desde el comienzo de una petición hasta que la respuesta se envía al cliente. Un cliente puede ejecutar varias peticiones pero permanecer sobre la misma vista. Esto es por lo que el ámbito de vista tiene mayor duración que el ámbito de petición. Los objetos con este ámbito pueden usar objetos sin ámbito, ámbito de petición, ámbito de vista, de sesión o de aplicación. Anotación: @RequestScoped

- **Flash.** Introducido en JSF 2, el nuevo ámbito flash provee una conversación de tiempo corto. Es una manera de pasar objetos temporales que se propagan a través de una transición en una sola vista y son borrados antes de pasar a otra vista. Este ámbito sólo puede ser usado mediante programa y no hay anotación.

- **Sin ámbito**. Los objetos con este ámbito no son visibles en ninguna página JSF y definen objetos que son usados por otros managed beans en la aplicación (con la anotación @NoneScoped) . Los objetos con este ámbito pueden usar objetos con el mismo ámbito.

Respecto al ámbito, a los managed bean sólo se les debería dar el ámbito estricto que necesiten. Un ámbito excesivo, por ejemplo de aplicación, incrementará el uso de la memoria y la necesidad de persistir los mismos causará un incremento del uso del disco. No tiene sentido dar ámbito de aplicación a un objeto que sólo se usa en un único componente. Por otra parte, un objeto con muchas restricciones, no estará disponible para diferentes partes de la aplicación.

4.6.d. @ManagedProperty

En un managed bean se puede indicar al sistema para que inyecte un valor en una propiedad(un atributo con getter y setter) ya sea usando el faces-config.xml o con la anotación **@javax.faces.bean.ManagedProperty** que tiene un atributo que puede tomar valor de literal o una expresión EL.

Se muestra un ejemplo en el que se inyectan valores. El valor de book se inicializa a través de una expresión EL. El valor de aTitle y aPrice se inicializa con unos literales. A pesar de que "999" es un string será convertido en Integer.

```
@ManagedBean
public class BookController {
  @ManagedProperty(value = "#{initController.defaultBook}")
   private Book book;
  @ManagedProperty(value = "this is a title")
   private String aTitle;
  @ManagedProperty(value = "999")
   private Integer aPrice;
}
```

4.7. Ciclo de vida y anotaciones de callback.

Tal como se ha visto el ciclo de vida de una página pasa por seis fases desde que se recibe la petición hasta que se renderiza la respuesta. Los managed beans tienen un ciclo más sencillo que pasa por dos estados. Si existen es durante el tiempo de vida indicado en el ámbito definido.

Imagen que muestra los estados de un managed bean.

Los managed beans se ejecutan en un contenedor servlet y pueden usar las anotaciones **@PostConstruct** and **@PreDestroy .** Después de que el contenedor cree una instancia de un managed bean , llama al método de callback **@PostConstruct** en el caso de que exista. Después de esta etapa, el managed bean está asociado a un ámbito y responde a cualquier petición de usuario. Antes de borrar el managed bean, el contenedor llama al método **@PreDestroy** si existe. Estos métodos pueden ser usados para inicializar atributos o crear y liberar cualquier recurso externo.

4.8. Navegación.

Las aplicaciones web están hechas de múltiples páginas a través de las que se necesita navegar. Una aplicación con muchas páginas necesitará un nivel de navegación más sofisticado; volver a la página inicial, volver a la página anterior o casos de la lógica de negocio donde se quiera ir a una determinada página

dependiendo de cierta regla. JSF dispone de varias opciones para la navegación y permite controlar la navegación, basada en unas páginas o para la aplicación entera.

Cuando se quiere ir a de una página a otra pulsando un botón o un enlace sin realizar ningún proceso, se pueden usar los UI componentes como **<h:button>**, **<h:link>** and **<h:outputLink>**.
En el ejemplo visto antes se tenía:

<h:link outcome="newBook.xhtml**" value="Create a new book"/>** Al pulsar el enlace el navegador va a la página newBook.xhtlm.

Pero en la mayoría de ocasiones esto no es suficiente porque se necesita acceder a la capa de negocio o a una base de datos para obtener y procesar datos. En este caso se deberían usar **<h:commandButton>** o **<h:commandLink>** que disponen del atributo **action** que permite ejecutar métodos de los managed bean. Ejemplo:

<h:commandButton value="Create" action="listBooks.xhtml**"/>**

La navegación se basa en un conjunto de reglas que definen todas las posibles rutas de navegación. En el sencillo ejemplo visto, se usa la regla más simple, cada método indica la página donde necesita ir. Nota aunque se indica listBooks.html se podría haber omitido .xhtml.

```
@ManagedBean
public class BookController {

@EJB
private BookEJB bookEJB;
private Book book = new Book();

public String doCreateBook() {
      book = bookEJB.createBook(book);
     return "listBooks.xhtml";
}
// Getters, setters }
```

JSF 2 también permite definir la navegación como se había hecho en versiones anteriores, a través del descriptor **faces-config.xml**. En este fichero la navegación se especifica con los elementos **<navigation-rule>** . Este elemento identifica la página de inicio, una condición y la página destino si la condición se cumple. Generalmente la condición se basa en un nombre lógico antes que en el identificador de la página, por ejemplo en el caso anterior, usando un descriptor de despliegue y el nombre lógico **success** :

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
          version="2.0">

  <navigation-rule>
    <from-view-id>newBook.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>listBooks.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>

</faces-config>
```

El managed bean BookController.

```
@ManagedBean
public class BookController {
   // ...
   public String doCreateBook() {
      book = bookEJB.createBook(book);
      bookList = bookEJB.findBooks();
      return "success";
   }
   // Constructors, getters, setters
}
```

La etiqueta <**from-view-id**> define la página donde la acción de la petición es inicialmente realizada, en este caso se empieza en newBook.xhtlm antes de realizar la llamada al managed bean. Si el nombre lógico devuelto es **success** (etiqueta **<from-outcome>**), el controlador Faces Servlet realizará un forward a la página listBooks.xhtml (etiqueta **<to-view-id>**).

Así la navegación se puede indicar directamente en los managed bean o en el descriptor faces-config.xml.
Si la navegación es muy compleja podría interesar tener las reglas en un único sitio centralizado y usar el faces-config.xml de manera que los cambios se puedan realizar más fácilmente.
Aunque el caso mostrado es muy simple, una página que sólo tiene una regla de navegación y una sola página donde ir, se podrían tener reglas más complejas, por ejemplo en el método doCreateBook(). En el caso de que se devuelva null, el usuario vuelve a la página que ya está.

```
public String doCreateBook() {
   book = bookEJB.createBook(book);
   bookList = bookEJB.findBooks();
   switch (value) {
      case 1: return "page1.xhtml"; break;
      case 2: return "page2.xhtml"; break;
      case 3: return "page3.xhtml"; break;
      default: return null; break;
   }
}
```

4.9. Gestión de mensajes.

Los managed beans procesan lógica de negocio, realizan llamadas a EJB y puesto que a veces las cosas pueden ir mal; en este caso el usuario tiene que ser informado a través de un mensaje que realice una acción. Los mensajes se pueden dividir en dos categorías:

1. Errores de aplicación que implican lógica de negocio, errores de bases de datos o errores de conexión de red. Los errores de aplicación pueden generar una página completamente diferente por ejemplo indicando al usuario que lo vuelva a intentar en unos momentos.
2. Errores de entrada de usuario. Campos no válidos, campos vacios…Generalmente los errores de entrada se muestran en la misma página, con mensajes explicando la causa.

Anteriormente se han visto dos etiquetas usadas para mostrar los mensajes en una página (**<h:message>** y **<h:messages>**). Para producir estos mensajes, JSF permite encolar estos mensajes llamando al método **FacesContext.addMessage()** en los managed beans que tiene la siguiente firma :

void addMessage(String clientId**,** FacesMessage **message)**

Este método añade un mensaje de la clase FacesMessage a la cola de mensajes que deben ser visualizados. El parámetro clientId se refiere a la ubicación Document Object Model del componente UI donde se registra el mensaje, por ejemplo **bookForm:isbn** se refiere al componente UI que tiene el identificador **isbn** dentro del formulario **bookForm**. Si el valor del parámetro clientId es null, el mensaje no se refiere a ningún componente en especial y es un mensaje global.

Nota: Puesto que una página XHTLM es un documento XML tiene una Document Object Model representación. Recordar que DOM es una especificación W3C para el acceso y modificación del contenido y estructura de los documentos XML. DOM es una representación en árbol de la estructura de un documento XML con raíz la etiqueta <html>.
DOM provee un standard para tratar los documentos XML. Se puede recorrer el árbol y editar el contenido de las hojas del mismo. Además es importante porque AJAX se basa en javascript interactuando con la representación DOM de una página web.

Un mensaje consiste en un texto de resumen, un texto detallado y un nivel de gravedad del mismo (fatal, error, warning e info). Los mensajes también pueden ser internacionalizados.
La firma de un mensaje Faces es:
FacesMessage**(Severity severity, String summary, String detail)**

Ejemplo del uso de los mensajes:

a. En el caso de que se capture una expeción se añade un mensaje global a la cola de mensajes con gravedad ERROR para ser visualizado.
b. Si el libro se crea con éxito un mensaje global de información se encola.
c. Si el título está vacío o repetido se genera un mensaje de aviso relativo al componente UI bookForm:title (que es su identificador DOM).
d. Análogamente con el componente bookForm:isbn.

```
public String doCreateBook() {

 FacesContext ctx = FacesContext.getCurrentInstance();

 if (book.getIsbn() == null || "".equals(book.getIsbn())) {
      ctx.addMessage(bookForm:isbn, new FacesMessage(FacesMessage.SEVERITY_WARN,
                              "Wrong isbn", "You should enter an ISBN number"));
   }
   if (book.getTitle() == null || "".equals(book.getTitle())) {
      ctx.addMessage(bookForm:title, new FacesMessage(FacesMessage.SEVERITY_WARN,
                          "Wrong title", "You should enter a title for the book"));
   }

   if (ctx.getMessageList().size() != 0)
      return null;

try {
      book = bookEJB.createBook(book);
      ctx.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, "Book created",
          "The book" + book.getTitle() + " has been created with id=" + book.getId()));
   } catch (Exception e) {
      ctx.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR,
                                  "Book hasn't been created", e.getMessage()));
   }
   return null;
}
```

Ejemplo del código de la página que asocia el mensaje al componente **isbn.**

```
<h:inputText id="isbn" value="#{bookController.book.isbn}"/>
<h:message for="isbn"/>
```

Resumiendo, una página puede mostrar mensajes globales con clientId null y usando una etiqueta **<h:messages> .** Si se quiere mostrar un mensaje en un lugar específico de la página para un determinado componente (lo usual es para los casos de error de conversión y validación) se usa **<h:message>.** Recordar que puesto que FacesContext está disponible para cualquier página se puede usar FacesMessage sin problemas.

4.10. Soporte para AJAX.

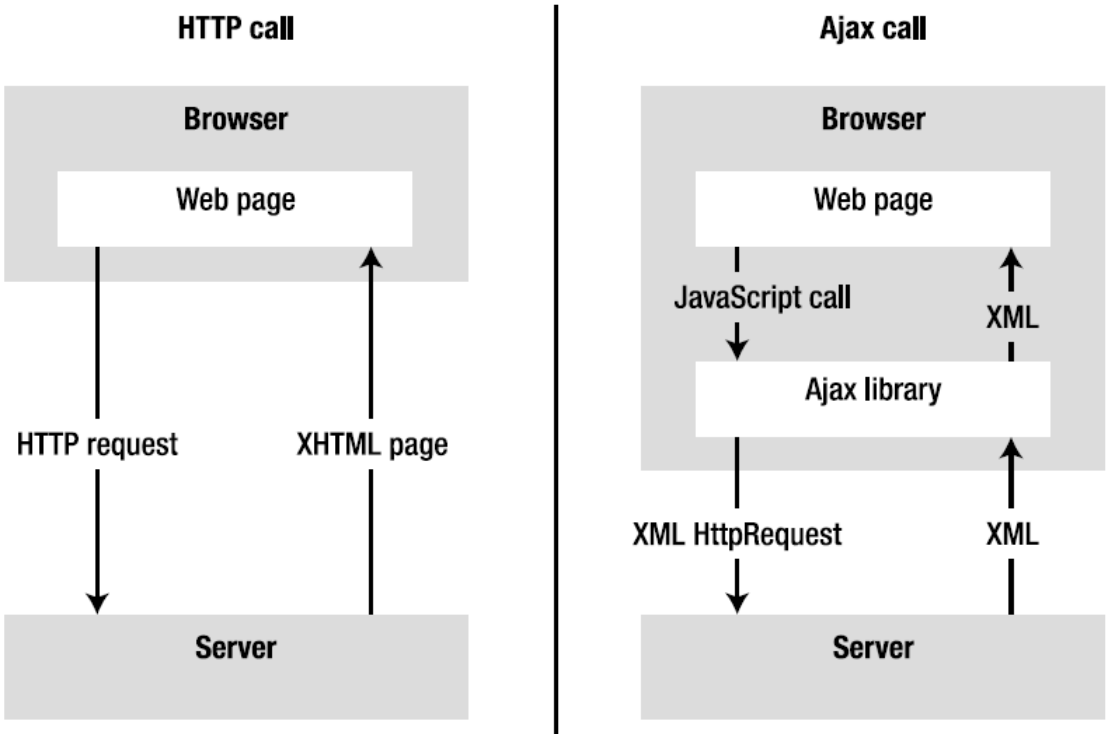
4.10.a. Nota sobre AJAX.

Breve explicación de AJAX.
En 2006 el W3C (world wide consortium) distribuyó la primera especificación para el objeto **XMLHttpRequest** que actualmente es soportado por la mayoría de navegadores. Al mismo tiempo se creó la Open Ajax Alliance para el desarrollo y uso de las tecnologías Ajax.

En principio, AJAX está basado en lo siguiente:

- XHTML y CSS para la presentación.
- DOM para la visualización dinámica e interacciones con los datos.
- XML y XSLT para el intercambio, manipulación y visualización de los datos.
- El objeto **XMLHttpRequest** para la comunicación asíncrona.
- JavaScript para juntar las tecnologías anteriores.

Figura siguiente. Diferencia entre una petición HTTP simple y una llamada HTTP Ajax. Figura extraída del libro de Antonio Goncalves, Beginning Java EE6 with GlassFish 3.



En la figura se observa como al pulsar el botón, en las aplicaciones tradicionales el navegador solicita la página web entera del servidor que cargará después. Por otra parte, AJAX usa transferencia de datos

asíncrona (con peticiones HTTP) entre el navegador y el servidor, permitiendo a las páginas web solicitar pequeñas porciones de información (en formato JSON o XML) al servidor en vez de la página entera. El usuario permanece en la misma página mientras una pequeña parte de Javascript solicita o envía datos a un servidor asíncronamente y sólo pequeñas porciones de la página son realmente modificadas haciendo a las aplicaciones web más rápidas en la respuesta y más amigables para el usuario.

Puesto que JSF 2 soporta Ajax nativamente, no hay que desarrollar JavaScript para usar XMLHttpRequest, pero si usar la librería JavaScript que está disponible con las implementaciones de JSF.

Nota: JSON:acrónimo de JavaScript Object Notation, es un formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML. Es una alternativa a XML en AJAX, una ventaja es que es mucho más sencillo escribir un analizador sintáctico (parser) de JSON.

4.10.b. Soporte en JSF.

Hay una librería JavaScript (**jsf.js**) para realizar la interacción con Ajax, lo que significa que no se deben desarrollar scripts ni manipular el objeto XMLHttpRequest directamente. Se pueden usar unas funciones estándar para enviar peticiones asíncronas y recibir datos. Para usar esta librería se debe añadir esta línea de código.

```
<h:outputScript name="jsf.js" library="javax.faces" target="head"/>
```

La etiqueta <h:outputScript> se traduce en un elemento de marcado que hace referencia al fichero de JavaScript **jsf.js** en la librería **javax.faces** (que sigue la nueva administración de recursos de JSF 2, es decir el fichero jsf.js está en el directorio **META-INF/resources/javax/faces.** Esta API de JavaScript se usa para inicializar las interacciones con JSF en el lado cliente incluyendo recorridos parciales del árbol y actualizaciones parciales de la página. La función que usaremos en nuestras páginas es la **request** responsable de enviar una petición Ajax al servidor. Su firma es:

jsf.ajax.request(ELEMENT, |event|, { |OPTIONS| });

Donde:

- **ELEMENT** es cualquier componente JSF o elemento XHTML desde donde se dispara el evento. Generalmente para enviar un formulario el element debería ser un botón.
- **EVENT** es cualquier evento JavaScript soportado por dicho elemento, por ejemplo **onmousedown**, **onclick**, **onblur…..**
- **OPTIONS** este argumento es un array que podría contener los siguientes pares nombre / valor:
    - **Execute**: <lista de componentes UI separados por espacios>: Envía una lista de componentes ID al servidor para que sean procesados durante la fase de petición.
    - **Render**: <lista de componentes UI separados por espacios>: Provee una lista de componentes ID que han sido procesados durante la fase de renderizado de la petición.

Como ejemplo se muestra el siguiente fragmento de código que muestra un botón que llama asíncronamente al método **doCreateBook** pasando todos los argumentos de un formulario de libro y que redibujará la lista de libros cuando el usuario añada un nuevo libro. El botón llama a la función

jsf.ajax.request cuando el usuario pulsa sobre él (evento **onclick**). El argumento **this** se refiere al elemento mismo (botón) y las opciones a los componentes (isbn, title, price…….)

```
< h:commandButton value="Create a book" onclick="jsf.ajax.request(this, event,
      {execute:'isbn title price description nbOfPage illustrations',
       render:'booklist'}); return false;"
      actionListener="#{bookController.doCreateBook}" />
```

Cuando el cliente realize una petición Ajax el ciclo de vida de la página en el servidor no varía, es decir, si todo es correcto pasa por las seis fases vistas. El beneficio es que la respuesta simplemente devuelve al navegador un subconjunto de la página antes que la página entera. Es en la segunda fase, cuando se aplican los valores de la petición, cuando se determina si es una petición parcial o no y el objeto **PartialViewContex** es usado a través del ciclo de vida de la página. Este contiene métodos y propiedades que forman parte del proceso de la petición parcial y del renderizado de la respuesta parcial. Al final del ciclo de vida, la respuesta Ajax (la respuesta parcial) se envía al cliente durante la fase "renderizado de la respuesta". Generalmente consiste en XHTLM, XML o JSON que el JavaScript del lado cliente analizará.

JSF 2 también incluye una aproximación declarativa que intenta ser más adecuada. Esta nueva aproximación saca partido de la nueva etiqueta **<f:ajax>**. En vez de codificar manualmente JavaScript para la llamada de la petición Ajax, se puede indicar el mismo comportamiento sin necesitar código JavaScript. Por ejemplo el fragmento de código anterior se puede escribir:

```
<h:commandButton value="Create a book" action="#{bookController.doCreateBook}">
   <f:ajax execute="@form" render=":booklist"/>
</h:commandButton>
```

En este ejemplo el atributo render apunta al componente que queremos renderizar (el componente booklist) . El atributo execute se refiere a todos los componentes que pertenecen al formulario. En la siguiente tabla se observan los valores que pueden tomar los atributos render y execute.

Tabla con los posibles valores de los atributos render y execute en la etiqueta ajax.

VALOR	DESCRIPCIÓN
@all	Ejecuta o renderiza todos los componentes en la vista.
@none	No ejecuta o renderiza ningún componente en la vista, es el valor por defecto si no se especifica render o execute.
@this	Ejecuta o renderiza sólo este componente (el componente que ha disparado la petición Ajax).
@form	Ejecuta o renderiza todos los componentes dentro del formulario desde donde la petición Ajax se disparó.
Lista de componentes Id separadas por espacios.	Uno o más componentes ID que deben ser procesados o renderizados.
Expresión de lenguaje EL	Expresión que resuelve una colección de Strings.

5. Tabla de comparación de los frameworks.

	Struts 2	Spring 3	Java Server Faces 2
Basado en	Framework J2EE	Framework J2EE	Framework J2EE
Entorno ejecución	En un contenedor de Servlets.	Usa su propio contenedor que gestiona el ciclo de vida los objetos que residen en el mismo. Permite dos tipos de contenedores pero el más usado es el contexto de aplicación ApplicationContext.	Especificación Java, cualquier servidor con J2EE lo soporta.
Enfoque orientado a…	El programador desarrolla pensando en Acciones.	Trabaja con controladores Spring que permiten procesar la solicitud. Usando anotaciones los controladores pueden ser simples POJO.	El programador desarrolla pensando en componentes UI, eventos y sus interacciones.
Arquitectura	Utiliza un MVC tipo dos.	Utiliza un MVC tipo dos en la capa de presentación.	Utiliza un MVC tipo dos.
Elemento a destacar	Patrón interceptor. Presenta una gran variedad de interceptores que facilitan el desarrollo.	Uso de la programación orientada a aspectos que permite un desarrollo más fácil y menos acoplado. Además permite presentar más funcionalidades que un framework exclusivo de la capa de presentación, como la gestión de transacciones.	Especificación Java, actualmente la JSR 314.
Configuración	Permite ficheros XML y anotaciones.	Permite ficheros XML y anotaciones.	Permite ficheros XML y anotaciones.
¿Permite Inyección de dependencias?	Si por ejemplo con Spring desde el fichero application-Context.xml	Si, desde el contenedor de Spring.	Si, desde el contenedor sobre los managed beans.
Tipo controlador.	Controlador frontal servlet FilterDispatcher.	Controlador frontal servlet DispatcherServlet.	Controlador Frontal servletFacesServlet.
Vistas usadas.	Amplia gama. Permite JSP, FreeMarker, Velocity Bibliotecas de etiquetas como JSLT.	Amplia gama. Permite JSP, Velocity, FreeMarker, Apache Tiles, el uso de XSLT…	JSP, las recomendadas son Facelets (páginas xhtml) y también páginas WML en dispositivos inalámbricos.

Arquitectura de Plugins .	Struts 2 permite añadir plugins añadiendo archivos JAR a la ruta de clases de la aplicación. Por ejemplo Tiles o JFreeChart. También permite integración con JSF a través de un plugin de Apache.	Permite añadir plugins añadiendo JAR. Ejemplos Spring Web Flow o Tiles. Presenta una buena capacidad de integración, observar que es más que un simple framework de la capa presentación así permite integración con JSF y Struts.	Permite integrarse con otros frameworks como Spring 3 o con Hibernate.
Integración con Frameworks de persistencia.	Permite una buena integración con Hibernate y Spring.	Spring ya presenta su propio framework de persistencia aunque se pueden usar otros.	Está en la especificación java, puede trabajar con la JPA del servidor por ejemplo en GlassFish. Aunque también he visto ejemplos de integración con Hibernate.
Soporte de validación.	Soporta un framework de validación configurable mediante XML y anotaciones.	Spring es compatible con JSR-303, especificación de validación de los Java Bean por ejemplo a través de la anotación @Valid.	Muy potente. Permite validar componentes individuales del formulario. Se puede validar usando los validadores estándares, creando métodos validadores en los backing beans o creando clases de validación especiales para casos genéricos.
Compartición entre proyectos de componentes UI.	No permite una reutilización muy alta.	Permite reutilización alta.	Alta. Permite fácil conexión de componentes creados por terceros.
Facilidad de Test	Presenta facilidades para pruebas unitarias, ejemplo la clase BaseStrutsTestCase.	Presenta librerías incorporadas para facilitar el test de las aplicaciones Spring y en concreto para el MVC.	Existen utilidades para facilitar el test como MyFaces Test framework para JSF 2 de Apache.
Autenticación	Permite autenticar construyendo un interceptor al efecto.	Proporciona el framework Spring Security que gestiona la autenticación y la autorización tanto a nivel de petición web como a nivel de método.	Se deben usar facilidades de los contenedores como objetos Realm o de la API servlet el servlet filter.
Soporte a la Internacionalización y Localización.	Facilita la internacionalización con un envoltorio de alto nivel al soporte java nativo para la internacionalización.	Facilita la internacionalización con utilidades propias como el bean “messageSource” que se usa conjuntamente con JSLT.	Provee utilidades para facilitar el manejo de la internalización de java.

6. Recopilación de ventajas y desventajas.

6.1 Struts.

Ventajas:

1) Arquitectura sencilla.
2) Más fácil de aprender, curva de aprendizaje menor.
3) Buena conversión y validación de datos muy completa. Dispone de validación en el cliente con JavaScript.
4) Según expertos, **J.S.F. 2 de Marty Hall,** el lenguaje de expresiones de Struts es más conciso y con más capacidades
5) No es una ventaja sino un dato, estadísticamente es de los frameworks más utilizados y según autores al llevar más tiempo en desarrollo, más "maduro".

Desventajas

1) Comparado con JSF 2 posee una gestión de eventos más pobre.
2) La reutilización de componentes de interface de usuario es inferior comparada con otros frameworks.

6.2 Spring 3.

Ventajas

1) Permite una reducción considerable del código de configuración de Spring a través la conexión y la detección automática.
2) Permite usar simples POJO usando anotaciones.
3) Permite la Programación Orientada a Aspectos AOP para gestionar transacciones, seguridad…..
4) El desarrollo del MVC Spring 3 hace que este sea compatible con los servicios web REST (transferencia de estados representacionales).
5) Al ser bastante más que un framework MCV, también permite uso de mensajería, tiene utilidades de test incorporadas….
6) Facilidad de integración con otros frameworks.

La desventaja de Spring es la curva de aprendizaje más lenta, es un framework muy completo pero con ello más difícil de usar.

6.3 Java Server Faces 2.

Ventajas

1) JSF es una especificación publicada por la Java Community Process, actualmente JSF está especificado en la JSR 314. Como parte de la especificación de Java EE todos los servidores que soportan JAVA EE incluyen JSF; JSF 2 es parte de Java EE 6.
2) JSF se ha inspirado en muchos open frameworks e intenta incorporar muchas de sus características.

3) Permite usar simples POJO como managed beans, con solo respetar los requisitos de un bean y usar anotaciones.

4) JSF 2 posee un extenso conjunto de APIS y etiquetas de cliente asociados para crear formularios HTML. JSF con sus manejadores de eventos facilita la indicación del código java que es invocado cuando los formularios son ejecutados. El código puede responder a botones específicos, cambios en ciertos valores, selecciones específicas del usuario. Esto responde a uno de los objetivos del diseño, desarrollar aplicaciones web de forma a la que se construyen aplicaciones locales con Java Swing.

5) JSF se puede utilizar para generar gráficos en formatos distintos de HTML, utilizando protocolos distintos de HTTP.

6) Permite combinar complejas GUI interfaces en un único manejable componente, cosa que por ejemplo Struts no permite. Además permite usar más librerías de componentes de terceras partes que Struts.

7) JSF permite usar jQuery y también AJAX a través de unas etiquetas simples, con Struts el soporte Ajax es más complejo.

8) Tiene mejor soporte de herramientas, tanto NetBeans como Eclipse tienen un moderado buen soporte integrado para JSF.

Desventajas

1) Comparado con Struts 2, su curva de aprendizaje es más lenta.
2) Actualmente La falta de validación en la capa cliente cosa que Struts usando Javascripts soporta.

7. Análisis del Framework del pfc.

En los puntos anteriores se han visto tres de los frameworks más utilizados en el desarrollo de aplicaciones web. Podemos observar las siguientes características que posee cualquier framework de la capa de presentación y que deberá tener el framework a desarrollar.

1) Para separarar la vista y el modelo, todos establecen un MVC tipo dos, donde un servlet controla las peticiones. Este controlador centraliza la lógica de la selección de la siguiente vista en base a los parámetros de entrada y el resultado de la petición, realmente, lo delega usando el patrón application controller.

2) Se produce una transferencia y conversión de manera automática de los datos de la petición del cliente a las Acciones (por ejemplo en Struts 2).

3) Se realiza una validación de los datos.

4) El flujo de navegación se establece de forma declarativa, es decir dada una petición la secuencia de comandos (o acciones) y vistas se declara de forma independiente del código.

5) Existen librerías de etiquetas que evitan incluir código en las páginas jsp.

6) Se da un soporte a la internacionalización, la aplicación se debe mostrar en un idioma especificado por el usuario sin modificar el código de la aplicación.

7) Debe ser fácilmente extensible. Se debenpoder añadir nuevas funcionalidades, es decir nuevos comandos que procesen la petición y vistas para mostrar los resultados.

7.1 Ideas generales sobre la arquitectura.

De lo mencionada en el punto anterior ya se puede deducir, que el pfc que desarrollaré se asemejará mas a un tipo Struts 2, orientado a Acciones.

Se usará un patrón arquitectónico Modelo Vista Controlador tipo dos. Para realizar un buen diseño se aplicará el patrón ServiceT Worker explicado en la introducción, además se usarár el patrón Context Object.

8. DISEÑO.

8.1 Introducción. MVC -2.

Se usará un patrón MVC de tipo 2, es decir con un controlador frontal. Para su diseño se usa la combinación de patrones Service to Worker y dentro de este se usa un Application Controller que usará el patrón Command para el diseño de las acciones. Además se tendrán los formularios con los campos que servirán de soporte a los campos.

Para definir la estructura del framework se aprovecha la tecnología Java Architecture for XML Binding. La capa de presentación estará definida por formularios y acciones. Es decir en la configuración tendremos formularios y acciones cada uno de ellos estará constituido por una clase. Además pará cada acción se tendrá definidos los resultados.

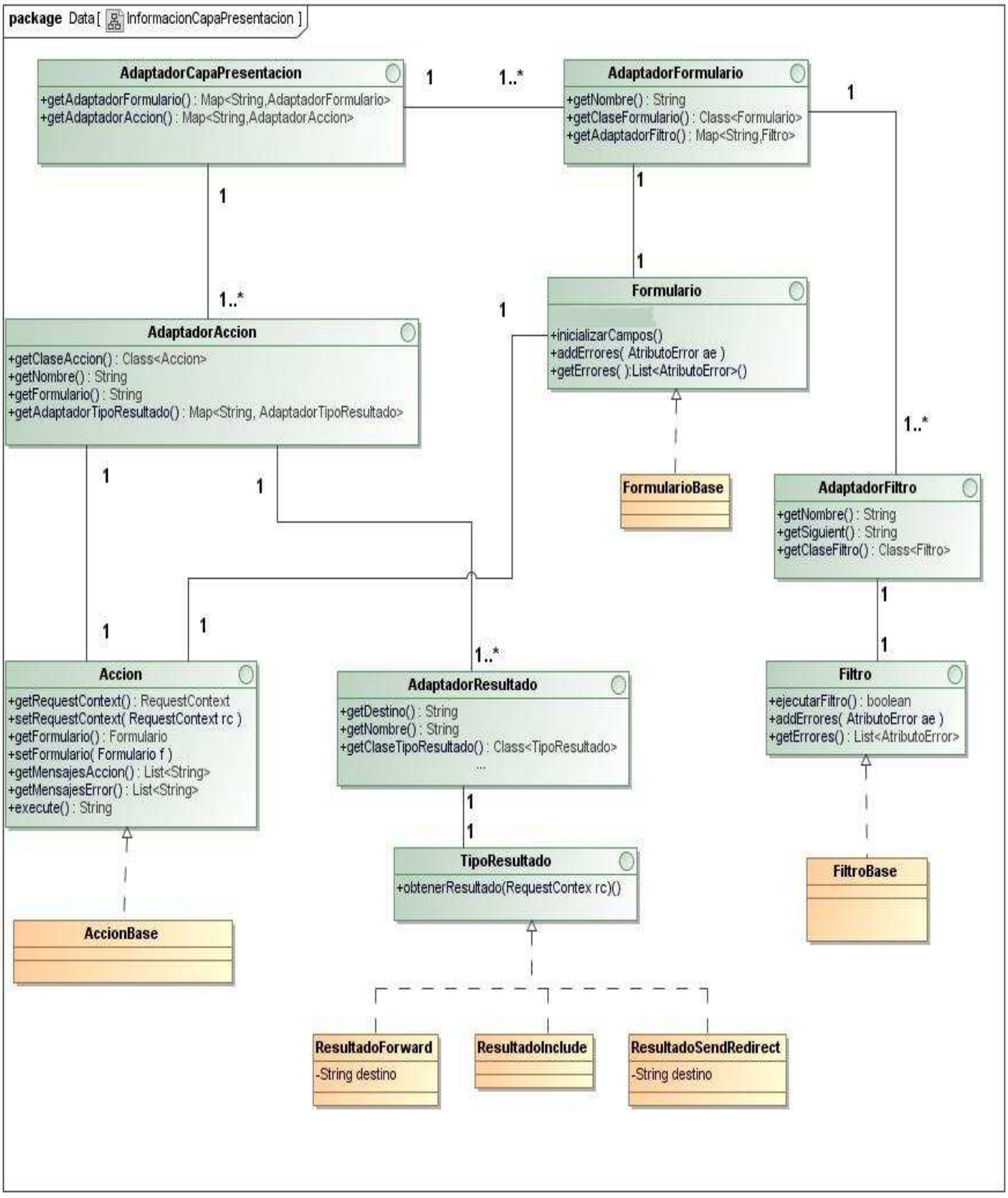
El resultado es un elemento con atributos que permite definir la navegación a través de ficheros xml, en caso de acierto o error u otras que nos inventemos. De esta manera se consigue uno de los **objetivos del framework**, parametrizar la navegación.

La carga de las acciones y formularios se realiza con la API Reflection. De esta manera conseguiremos **otro de los objetivos de los frameworks**, carga y validación automática de los campos de los formularios. La carga de los atributos y su validación se realiza a través de clases que implementan la interface filtro. A cada formulario se le asociará una cadena de filtros. Lo he denominado filtro pero no tienen relación con los filter de la API servlet ni son los interceptores de Struts. La idea si que era análoga a Struts, tener elementos reutilizables. Los filtros del framework son clases que colaboran con los formularios, realizando la carga de atributos, validaciones… pero no actúan antes y después de la ejecución de la acción.

Así el cliente definirá su capa de presentación a través de un fichero xml, en que indicarán sus formularios, acciones, resultados y filtros. Este fichero se valida contra el esquema

8.2 Información manejada por la aplicación.

En el siguiente diagrama de clases se observa la información que cualquier aplicación que utilice el framework debe manejar.



Las interfaces que he denominado adaptador son para enlazar con las clases de la tecnología JAXB pero lo importante es que cada capa de presentación manejará N acciones y M formularios.

Obsevar que existe una relacion 1:1 y que finalmente para cada AdaptadorXXX existe una acción, un formulario, un resultado y un filtro.

Cada acción implementa el patrón command y tiene asociado N resultados y puede tener asociado un formulario. Tal como se ha comentado el resultado permite la navegación según los parámetros y de la ejecución del método execute

A su vez cada formulario tiene asociado N filtros, por lo menos debe tener uno , se comprueba en la aplicación para realizar las tareas de carga y validación.

Finalmente cuando se usa el framework se deberán usar las clases indicadas en el diagrama, AccionBase, FiltroBase, ResultadoForward….que implementan las interfaces requeridas como punto de partida.

8.3 Patrón command.

Tal como se ha indicado en la introducción cada capa de presentación de una aplicación estará formada por un conjunto de acciones y formularios. Cada acción implementará el patrón command y unas características que se adecuan al framework.
El patrón command se ha aplicado exhaustivamente, también se ha aplicado a los filtros y los resultados. De esta manera nos permite trabajar con objetos a los que se solicitará una operación cuyo contenido no se conoce de antemano.

8.4 Patrón Application Controller.

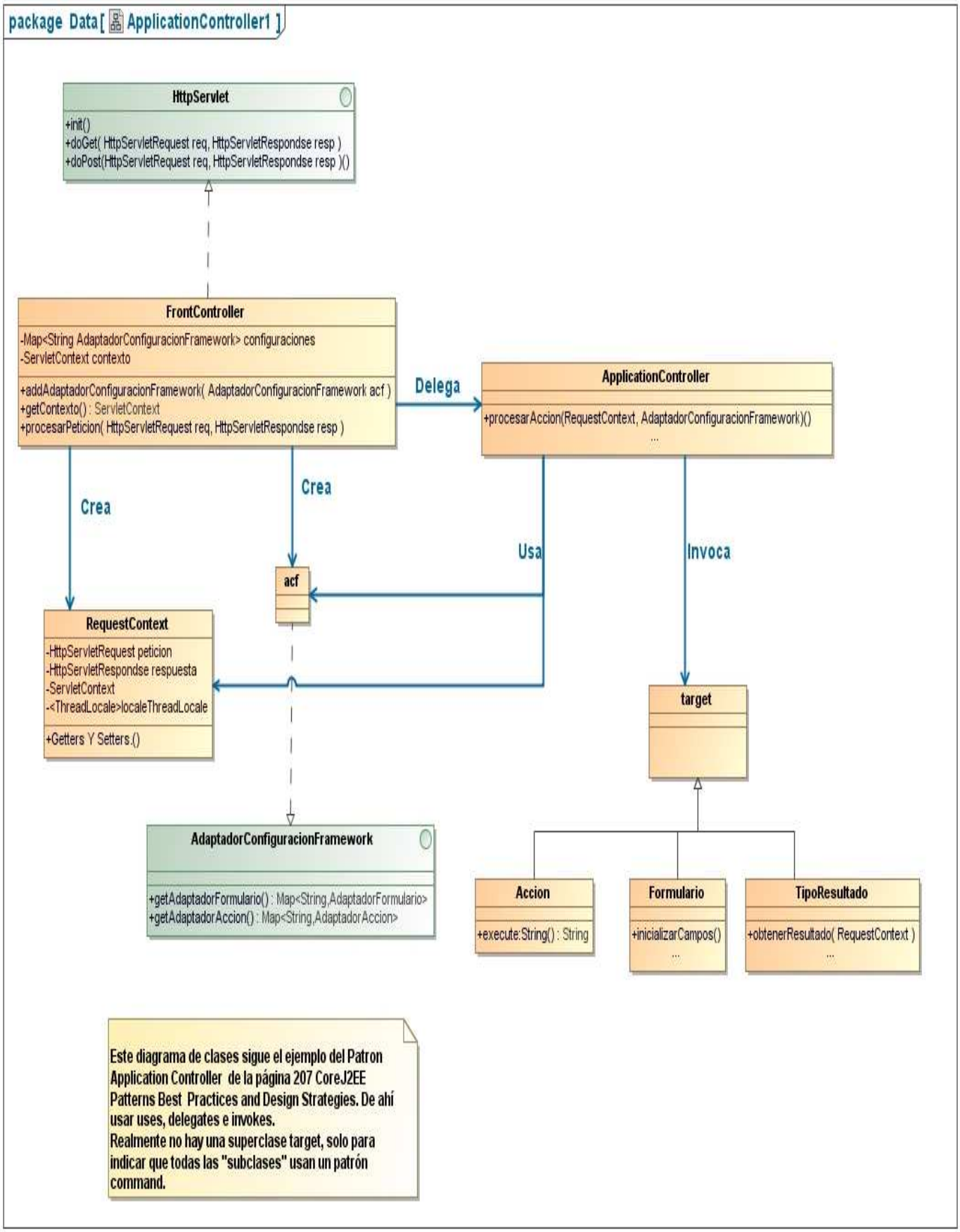
Tal como se ha indicado, se usa un MVC tipo 2 con controlador frontal. La clase se denomina FrontController y naturalmente hereda de HttpServlet.

Usar el Application Controller permite que extraer fuera del controlador frontal toda la lógica de proceso de la aplicación.

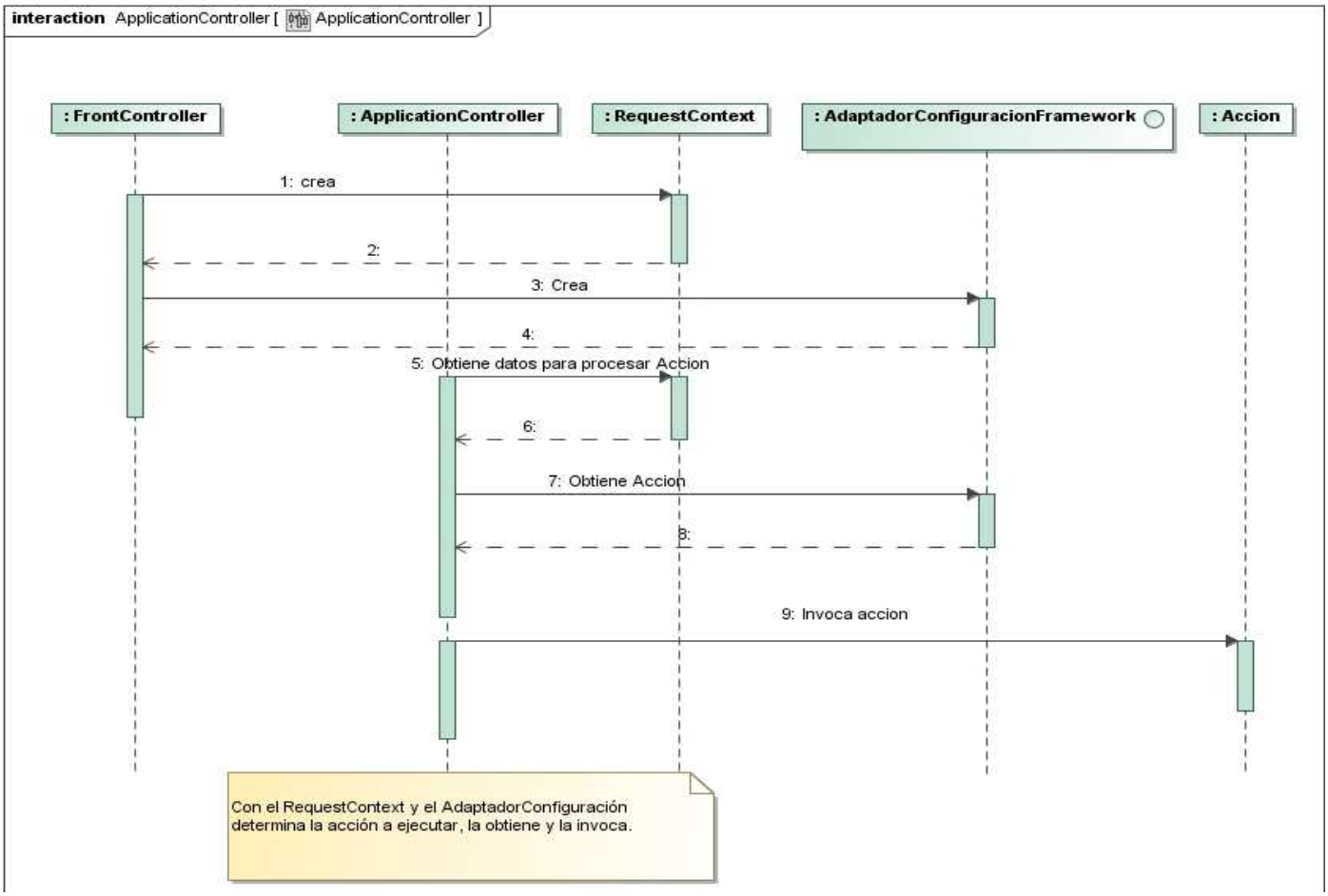
Se ha usado el patrón Contex Object para encapsular toda la información relativa a la petición. La clase es la RequestContext. El controlador frontal se encarga de crear este objeto puesto que conoce toda la información de la petición. Este objeto se pasa al ApplicationController.

Además su método init carga toda la información relativa a la configuración de la aplicación. El controlador frontal hace el papel de map para las distintas configuraciones que se puedan cargar. Un objeto que implementa la interface AdaptadorConfiguraciónFramework se pasa al application controller para gestionar las peticiones. En la teoría se suele indicar un objeto aparte.

Tal como se especifica en el diagrama se ha tomado el ejemplo de la página 270 de CoreJ2EE Patterns, bests practices and designs que usa asociaciones con “uses”, “invokes”, “delegates”…


package Data[ApplicationController1]
HttpServlet
+init()
+doGet(HttpServletRequest req, HttpServletRespondse resp)
+doPost(HttpServletRequest req, HttpServletRespondse resp)()
FrontController
-Map<String AdaptadorConfiguracionFramework> configuraciones
-ServletContext contexto
+addAdaptadorConfiguracionFramework(AdaptadorConfiguracionFramework acf)
+getContexto() : ServletContext
+procesarPeticion(HttpServletRequest req, HttpServletRespondse resp)
Delega
ApplicationController
+procesarAccion(RequestContext, AdaptadorConfiguracionFramework)()
...
Crea
Crea
Usa
Invoca
acf
RequestContext
-HttpServletRequest peticion
-HttpServletRespondse respuesta
-ServletContext
-<ThreadLocale>localeThreadLocale
+Getters Y Setters.()
target
AdaptadorConfiguracionFramework
+getAdaptadorFormulario() : Map<String,AdaptadorFormulario>
+getAdaptadorAccion() : Map<String,AdaptadorAccion>
Accion
+execute:String() : String
Formulario
+inicializarCampos()
...
TipoResultado
+obtenerResultado(RequestContext)
...
Este diagrama de clases sigue el ejemplo del Patron Application Controller de la página 207 CoreJ2EE Patterns Best Practices and Design Strategies. De ahí usar uses, delegates e invokes.
Realmente no hay una superclase target, solo para indicar que todas las "subclases" usan un patrón command.

Un diagrama de secuencia.



La parte fundamental de esta clase es el método procesarAccion

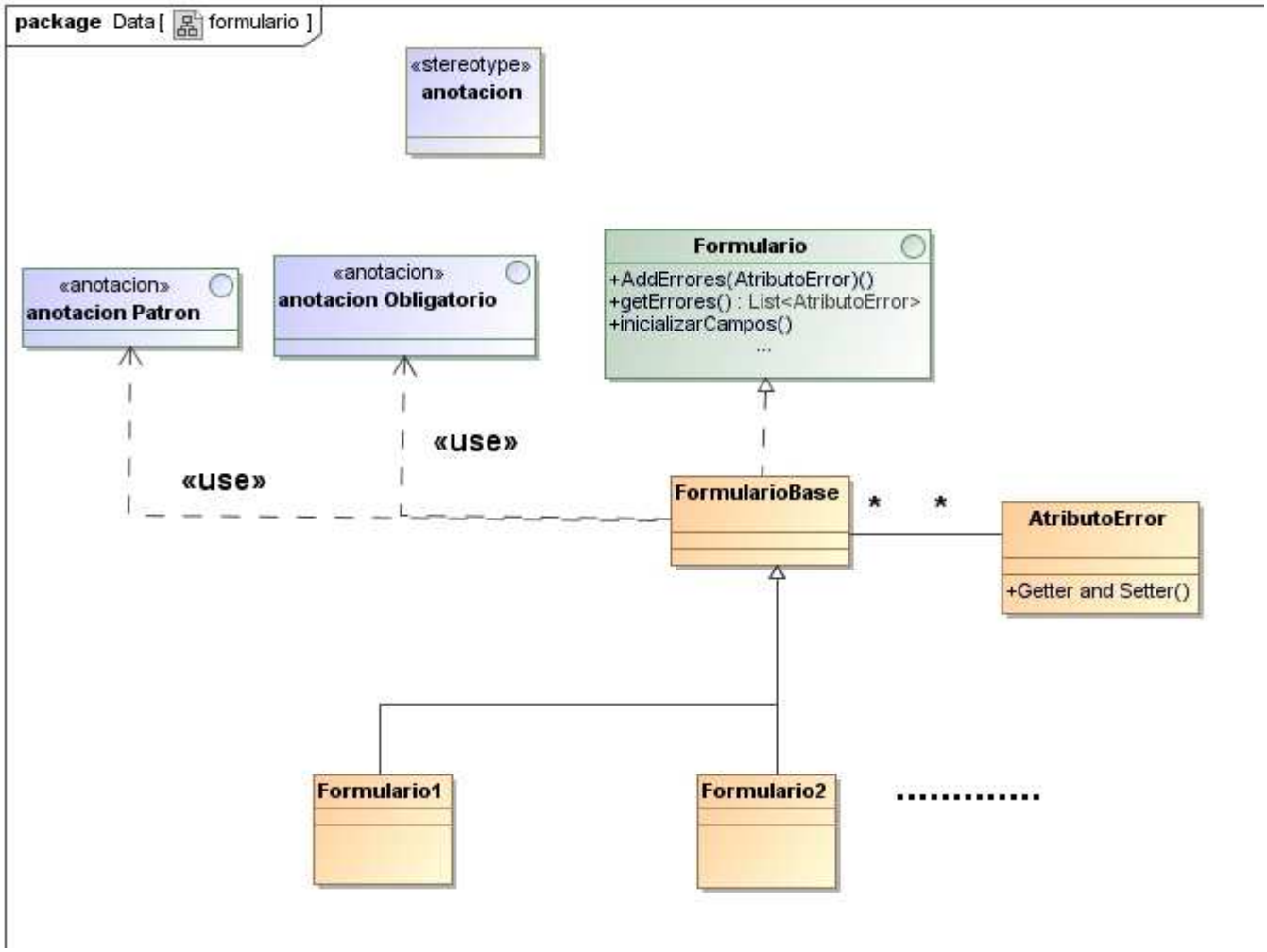
```
public static void procesarAccion (RequestContext rq,
AdaptadorConfiguracionFramework acf){
	try {
		loggerClase.log(Level.INFO, "inicio proceso acción.");
		// obtener adaptador
		adaptadorAccion =  getAdaptadorAccion(rq.getRequest(), acf);

		// Crear una instancia de la acción
		accion = crearInstanciaAccion(adaptadorAccion);

	// Procesamos el formulario, obteniendo el formulario y procesando filtros.
		formulario = obtenerFormulario(adaptadorAccion, rq, acf);

		// Establecemos el formulario
		accion.setFormulario(formulario);

		// guardamos la Accion en la request, para más adelante, acceder
			por ejemplo desde los TAGS
		rq.getRequest().setAttribute(ParametrosFramework.guardar_accion,
			accion);
// Almacenamos en la accion el contexto, por si se necesita acceder a la request..
```

```
 accion.setRequestContext(rq);
// Ejecutamos la accion
 String resultado = accion.execute();
// Procesamos el resultado
 generaResultado(adaptadorAccion,rq,resultado);
```

Este método va obteniendo la información extraída por la tecnología JAXB y usando la reflexión genera instancias de cada elemento acción, formulario, filtro y resultado y ejecuta los métodos del patrón command execute, generaResultado y ejecutarFiltro. La acción a ejecutar se determina con el método getServletPath() de la clase HttpServletRequest.

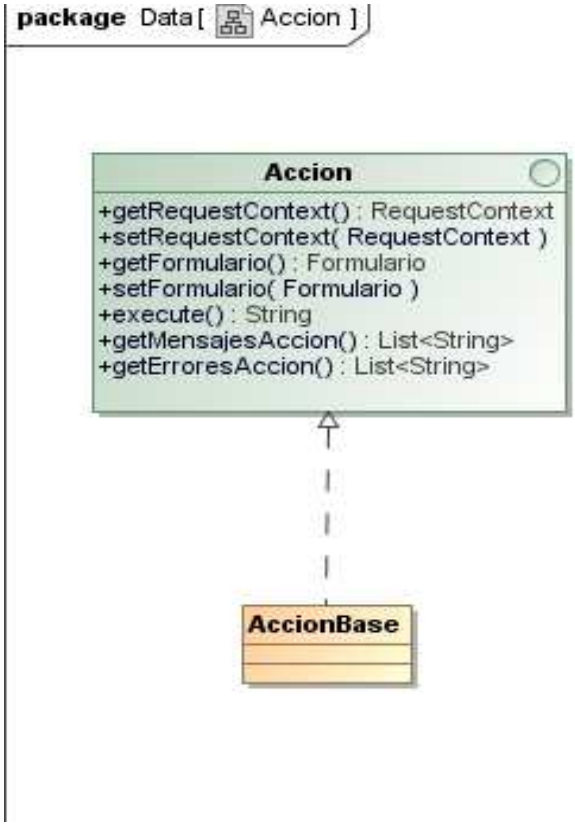
8.5 Formularios.

Respecto a los formularios, se utiliza la interface Formulario, la anotación Obligatorio, la anotación Patrón y la clase AtributoError para gestionar los errores de los formularios. Las anotaciones permiten tratar mejor los atributos, la anotación Obligatorio permite indicar atributos que no pueden quedar vacios y la anotación Patron indica un patrón que los atributos deben cumplir. Los formularios que usan el framework deben heredar de la clase FormularioBase que es una clase que sencillamente implementa la interface Formulario. Respecto a la implementación no ha sido necesario usar atributos públicos, usando las propiedades de la reflexión se puede acceder a atributos prívate.



8.6 Acciones.

Análogamente ocurre con las acciones. Las acciones de la capa de presentación que usen el framework deberán extender la clase AccionBase que implementa la interface Acción.

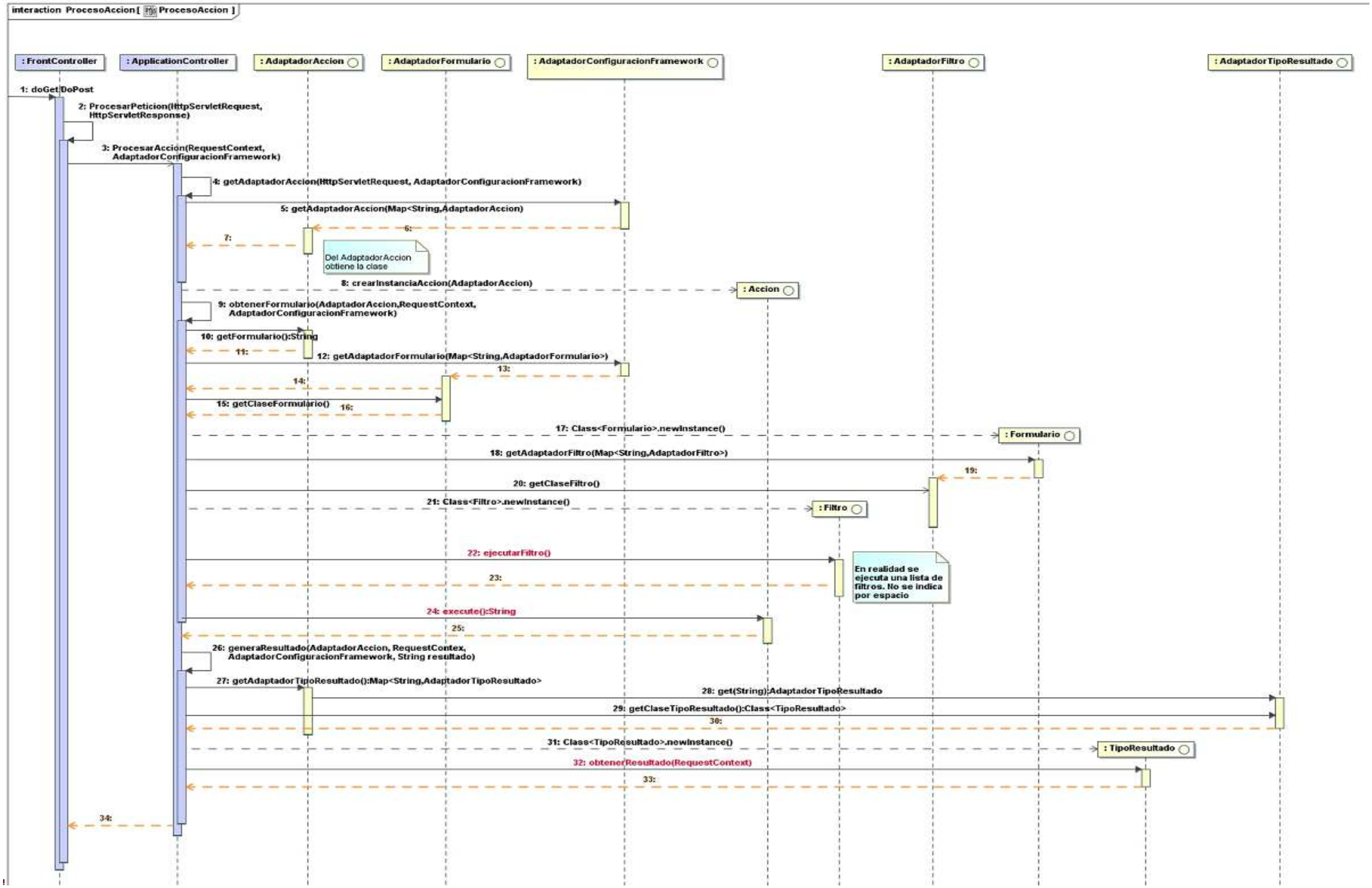


Respecto a la implementación, la clase RequestContext es la implementación del patrón Context Object y contiene información sobre la petición, respuesta y el contexto del Servlet.

8.6.1 Ejecución de una acción.

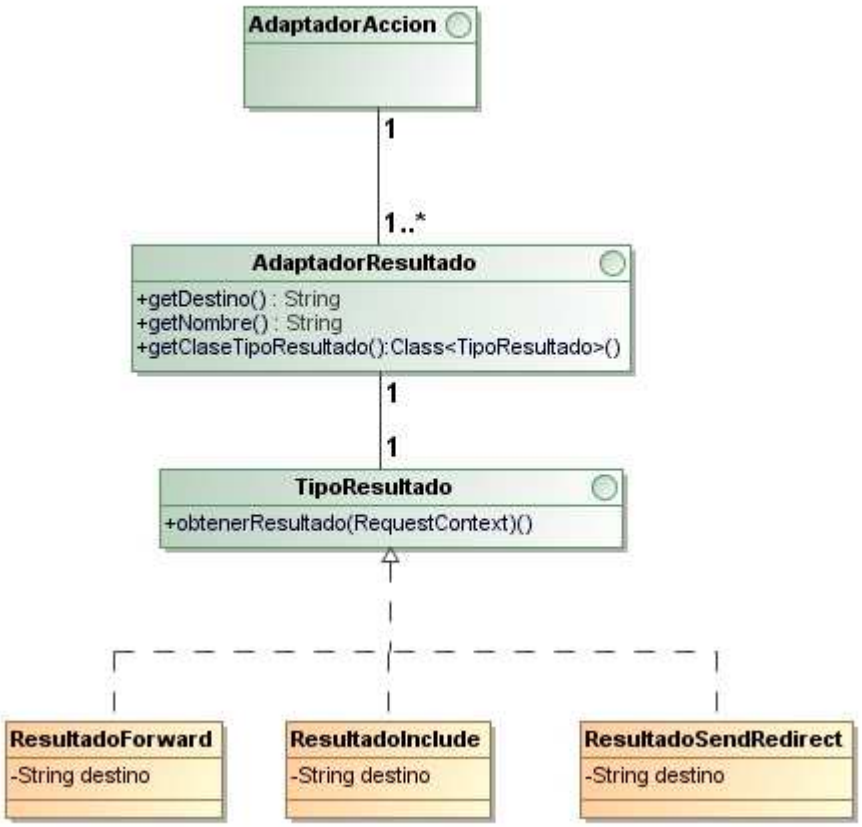
Como se ha señalado se ha usado bastante patrón command, en las acciones, en los filtros y en los resultados.
Señalar que en realidad se aplica un conjunto de filtros que ya no se indican pues el diagrama de secuencia sería más grande. También he omitido la obtención de la clase de la Acción. En rojo están las acciones del patrón command.

En la siguiente página se observa un diagrama de secuencia de la ejecución de una acción en general.


interaction ProcesoAccion [ProcesoAccion]
: FrontController
: ApplicationController
: AdaptadorAccion
: AdaptadorFormulario
: AdaptadorConfiguracionFramework
: AdaptadorFiltro
: AdaptadorTipoResultado
1: doGet/DoPost
2: ProcesarPeticion(HttpServletRequest, HttpServletResponse)
3: ProcesarAccion(RequestContext, AdaptadorConfiguracionFramework)
4: getAdaptadorAccion(HttpServletRequest, AdaptadorConfiguracionFramework)
5: getAdaptadorAccion(Map<String,AdaptadorAccion)
6:
7:
Del AdaptadorAccion obtiene la clase
8: crearInstanciaAccion(AdaptadorAccion)
: Accion
9: obtenerFormulario(AdaptadorAccion,RequestContext, AdaptadorConfiguracionFramework)
10: getFormulario():String
11:
12: getAdaptadorFormulario(Map<String,AdaptadorFormulario>)
13:
14:
15: getClaseFormulario()
16:
17: Class<Formulario>.newInstance()
: Formulario
18: getAdaptadorFiltro(Map<String,AdaptadorFiltro>)
19:
20: getClaseFiltro()
21: Class<Filtro>.newInstance()
: Filtro
22: ejecutarFiltro()
23:
En realidad se ejecuta una lista de filtros. No se indica por espacio
24: execute():String
25:
26: generaResultado(AdaptadorAccion, RequestContex, AdaptadorConfiguracionFramework, String resultado)
27: getAdaptadorTipoResultado():Map<String,AdaptadorTipoResultado>
28: get(String):AdaptadorTipoResultado
29: getClaseTipoResultado():Class<TipoResultado>
30:
31: Class<TipoResultado>.newInstance()
: TipoResultado
32: obtenerResultado(RequestContext)
33:
34:

8.7 Resultados.

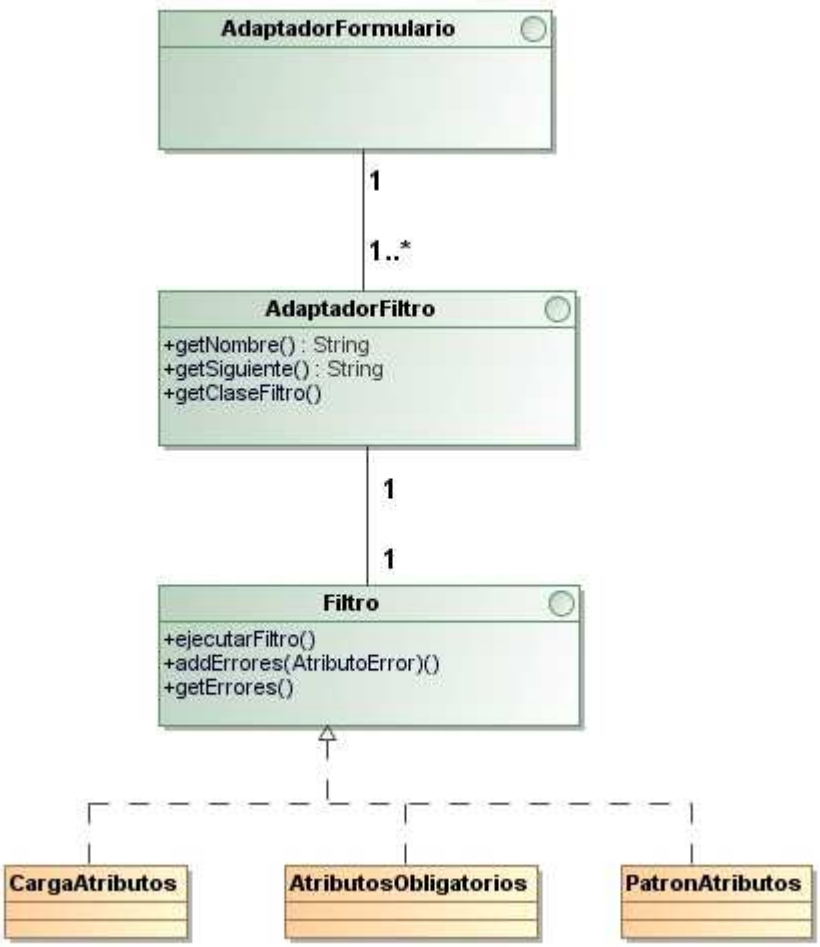
Respecto a los resultados también se ha basado en Struts 2 con unos tipos de resultados que usan la API Servlet. Implementando tipos de resultados como ResultadoForward que usa el método forward de la clase RequestDispatcher y también el ResultadoInclude que usa el método include de RequestDispatcher. Asimismo se ha incluido redirecciones usando la respuesta, implementando la clase ResultadoSendRedirect.



Aunque las tres clases usan el atributo destino, tienen significados distintos, con ResultadoForward indica el recurso, lá página jsp, donde se transfiere la petición. Análogamente para ResultadoInclude, indica el recurso de destino donde se realizará el include. Para ResultadoSendRedirect, deberá indicar la url hacia la que se tiene que generar la nueva petición HTTP.

8.8 Filtros.

Respecto a los filtros, la idea es conseguir clases reutilizables que realizan tareas comunes sobre los formularios, un poco como la pila de interceptores de Struts 2. A cada formulario le corresponden de 1 a N filtros, por lo menos uno ya que siempre debe existir el filtro que cargue los atributos. Así el primero es **CargaAtributos**. En la configuración XML el usuario debe definir qué filtro se aplicará después del primero, teniendo así una lista enlazada hasta indicar vacio.
Las clases deben implementar la interface Filtro.



8.9 Librería de etiquetas. Internacionalización.

Para implementar nuestra librería de etiquetas se usará la clase TagSupport, sólo usaremos etiquetas sin cuerpo. Las etiquetas que heredan de esta clase supone que sólo hemos de sobreescribir los métodos doStartTag que se ejecuta cuando se encuentra la etiqueta de apertura y el método doEndTag que se invoca cuando se encuentra la etiqueta de cierre. Se ha definido una clase Etiqueta Base que hereda de TagSupport. El resto de etiquetas hereda de EtiquetaBase.

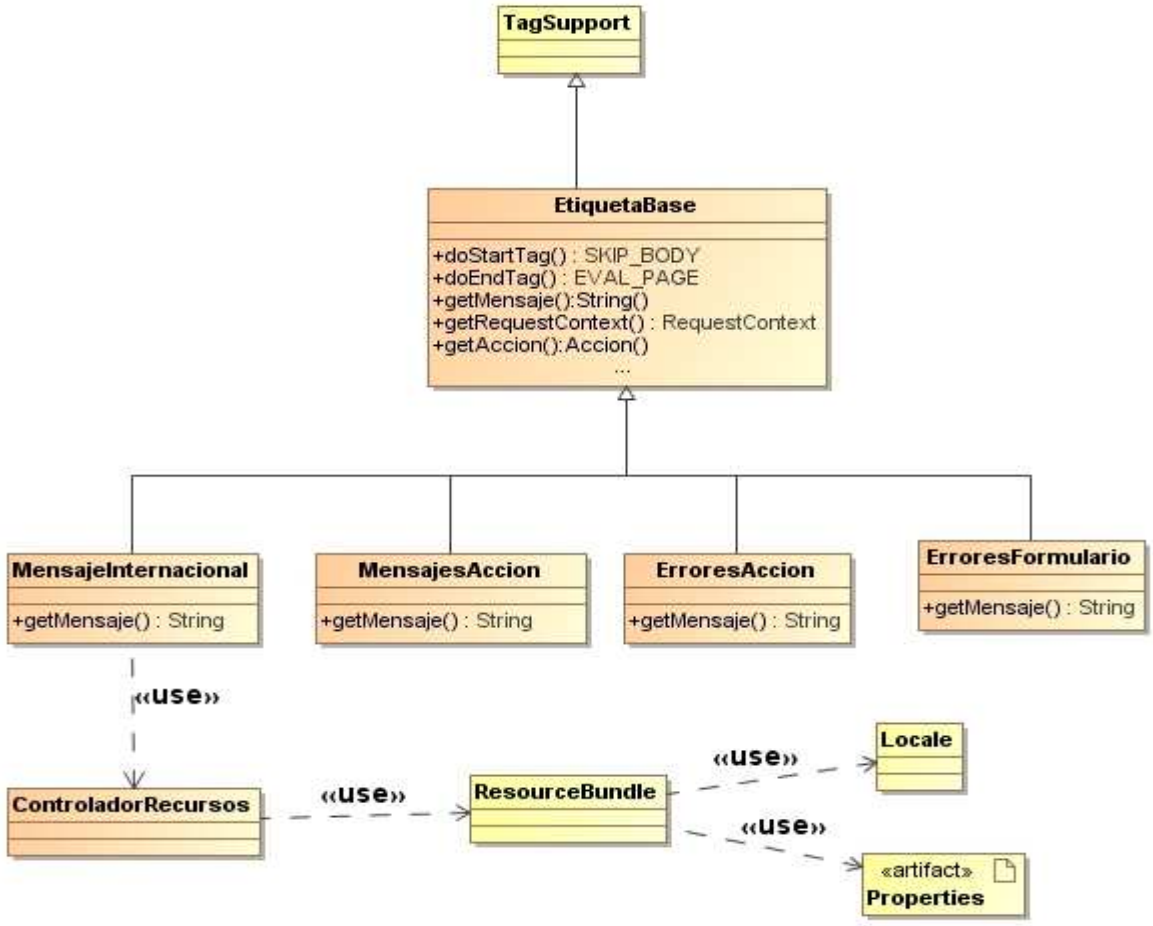
La clase EtiquetaBase posee además un método para obtener el contexto de la petición, otro para obtener la acción implicada y un método getMensaje() que es usado por doStartTag para escribir el mensaje. Este método es sobreescrito por las distintas etiquetas.

Métodos doStartTag y doEndTag de EtiquetaBase:

```
public int doStartTag() throws JspException {
    try{
        pageContext.getOut().write(getMensaje());
    }catch(IOException e){
       throw new JspException("Error IOException. Mensaje: "+ e.getMessage());
    }
    return SKIP_BODY;
}

public int doEndTag() throws JspException{
    return EVAL_PAGE;
}
```

El entero SKIP_BODY indica al motor de JSP que la etiqueta no tiene cuerpo y el entero EVAL_PAGE le indica que siga evaluando el resto de la página.

Diagrama de clases de las etiquetas donde se muestra el uso del controlador de recursos por MensajeInternacional, NO se muestra el uso de las otras clases por claridad del diagrama.



El uso de la librería de etiquetas permite la internacionalización. Por ejemplo en las páginas jsp se usa la clase MensajeInternacional para mostrar los mensajes. Esta clase utiliza la clase ControladorRecursos que busca los mensajes en distintos idiomas en un fichero properties según una clave. Para ello la clase ControladorRecursos utiliza el soporte java ResourceBundle. ResourceBundle a su vez usa la clase Locale y ficheros properties. El fichero properties es un fichero con estructura clave = valor.

De esta manera se pueden mostrar los mensajes de la aplicación en distintos idiomas según el Locale y usando el fichero de properties adecuado.

Se han usado dos ficheros properties por idioma, uno denominado librería para los mensajes de botones, etiquetas… y otro mensajesAplicacion para los mensajes de las distintas acciones.

9. Esquema del Framework.

La API JAXB asocia cada tipo a una clase. En rojo están las explicaciones.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://framework.uoc.es/configuracion" elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:fam="http://framework.uoc.es/configuracion"
xmlns:jxb= "http://java.sun.com/xml/ns/jaxb" jxb:version= "2.0" >

<annotation>
  <appinfo>
    <jxb:schemaBindings>
     <jxb:nameXmlTransform>
      <jxb:typeName prefix="Jaxb"/>
     </jxb:nameXmlTransform>
    </jxb:schemaBindings>
  </appinfo>
 </annotation>
```

Esta anotación es para permitir que las clases generadas tengan el prefijo Jaxb para facilitar su designación.

```
 <element name="Configuracion" type="fam:TipoConfiguracion"></element>
```

Elemento global y su tipo configuración debajo, constituido por formularios y acciones.

```
<complexType name="TipoConfiguracion">
        <sequence>
                <element name="BeansRespaldo" type="fam:TipoBeansRespaldo"
                minOccurs="0" maxOccurs="1"/>
                <element name="Acciones" type="fam:TipoAcciones" minOccurs="1"
                 maxOccurs="1"/>
        </sequence>
  </complexType>
```

```
<complexType name="TipoAcciones">
        <sequence>
                <element name="Accion" type="fam:TipoAccion" minOccurs="1"
                 maxOccurs="unbounded"/>
        </sequence>
  </complexType>
```

Genera una lista de acciones.

```
  <complexType name="TipoAccion">
        <sequence>
                <element name="Resultados" type="fam:TipoResultados" minOccurs="1"
                maxOccurs="1"/>
        </sequence>
        <attribute name="nombre" type="string" use="required" />
        <attribute name="clase" type="string" use="required" />
        <attribute name="formulario" type="string" use="optional"/>
  </complexType>
```

Definición del tipo acción, su nombre, su clase, el formulario asociado si lo tiene y los posibles resultados de la ejecución de la misma.

```
  <complexType name="TipoResultados">
        <sequence minOccurs="1" maxOccurs="unbounded">
                <element name="Resultado" type="fam:TipoResultado" minOccurs="1"
                maxOccurs="unbounded"/>
        </sequence>
  </complexType>
```

Lista de resultados.

```
  <complexType name="TipoResultado">
        <attribute name="nombre" type="string" use="required" />
        <attribute name="clase" type="string" use="required" />
        <attribute name="destino" type="string" use="required" />
  </complexType>
```

Definición del tipo resultado, su nombre, su clase y el atributo destino que nos indicará el recurso destino del mismo, lo normal una página jsp.

```
  <complexType name="TipoBeansRespaldo">
        <sequence>
                <element name="Formulario" type="fam:TipoFormulario" minOccurs="1"
                 maxOccurs="unbounded"/>
        </sequence>
  </complexType>
```

```
<complexType name="TipoFormulario">
        <sequence>
                <element name="Filtros" type="fam:TipoFiltros" minOccurs="1"
                 maxOccurs="1" />
        </sequence>
        <attribute name="nombre" type="string" use="required"/>
        <attribute name="clase" type="string" use="required"/>
   </complexType>
```

Definición del tipo formulario, su nombre, su clase y los filtros asociados. Los filtros suponen clases auxiliares que forman una cadena para realizar las acciones de carga de atributos, validación……

```
<complexType name="TipoFiltros" >
                <sequence minOccurs="1" maxOccurs="unbounded">
                <element name="Filtro" type="fam:TipoFiltro" minOccurs="1"
                 maxOccurs="unbounded"/>
                </sequence>
</complexType>
```

Definición de la cadena de filtros.

```
<complexType name="TipoFiltro">
        <attribute name="nombre" type="string" use="required"/>
        <attribute name="clase" type="string" use="required"/>
        <attribute name="siguiente" type="string" use="required"/>
   </complexType>
</schema>
```

Definición del filtros, su nombre su clase y el siguiente filtro a aplicar.

Con este esquema se generan unas clases (JAXB asocia cada tipo a una clase) que a través de la API JAXB y usando el método unmarshal de la clase Unmarshaller permite tratar el fichero xml que representa la aplicación cliente.

9.1. Comentario del paquete uoc.jaor.configuracion.jaxb, uso de XmlAdapter.

Las clases generadas por el compilador del esquema xjc son:

```
Analizando un esquema...
Compilando un esquema...
..\jaxb\JaxbTipoAccion.java
..\jaxb\JaxbTipoAcciones.java
..\jaxb\JaxbTipoBeansRespaldo.java
..\jaxb\JaxbTipoConfiguracion.java
..\jaxb\JaxbTipoFormulario.java
..\jaxb\JaxbTipoFiltro.java
```

..\jaxb\JaxbTiposFiltros.java
..\jaxb\JaxbTipoResultado.java
..\jaxb\JaxbTipoResultados.java
..\jaxb\ObjectFactory.java
..\jaxb\package-info.java

Estas clases se modifican para facilitar la reflexión posterior, convirtiendo strings en clases de acción Class<Accion>, clases de formulario Class<Formulario>, clases de resultados Class<TipoResultado> y clases de Filtro Class<Filtro>. Para ello se usa la clase XmlAdapter y la anotación @XmlJavaTypeAdapter.

Además puesto que se usa el patrón Application Controller en su implementación más sencilla, que usa un Map para guardar las acciones (también se hace con las formularios por extensión), se transforman las listas que por defecto ofrece JAXB en Map's, usando otra vez la clase XmlAdapter y la anotación @XmlJavaTypeAdapter. Lo mismo con los filtros utilizados. Realmente lo que XMLAdapter realiza es interceptar la asociación (binding) al realizar la operación de marshaling o unmarshaling. A modo de ejemplo se presenta la clase que transforma una lista de filtros en un map, se observa que se ha de programar la transformación de una estructura en otra.

```
public class AdaptadorJaxbTipoFiltrosMap extends
            XmlAdapter<JaxbTipoFiltros, Map<String,JaxbTipoFiltro>> {

@Override
public JaxbTipoFiltros marshal (Map<String,JaxbTipoFiltro> mjtf) throws Exception {
            JaxbTipoFiltros jtas = new JaxbTipoFiltros();
            jtas.setFiltro(new ArrayList<JaxbTipoFiltro>(mjtf.values()));
            return jtas;
}
@Override
public Map<String, JaxbTipoFiltro> unmarshal (JaxbTipoFiltros jtfs) throws Exception
{
      int numero = 0;
      if (jtfs != null && jtfs.getFiltro() != null)
      numero = jtfs.getFiltro().size();
      Map<String,JaxbTipoFiltro> mjtf = new HashMap<String,JaxbTipoFiltro>(numero);
      if (jtfs != null && jtfs.getFiltro() != null){
            for(JaxbTipoFiltro tf: jtfs.getFiltro()){
                  mjtf.put(tf.getNombre(), tf);
            }
      }
      return mjtf;
}
}
```

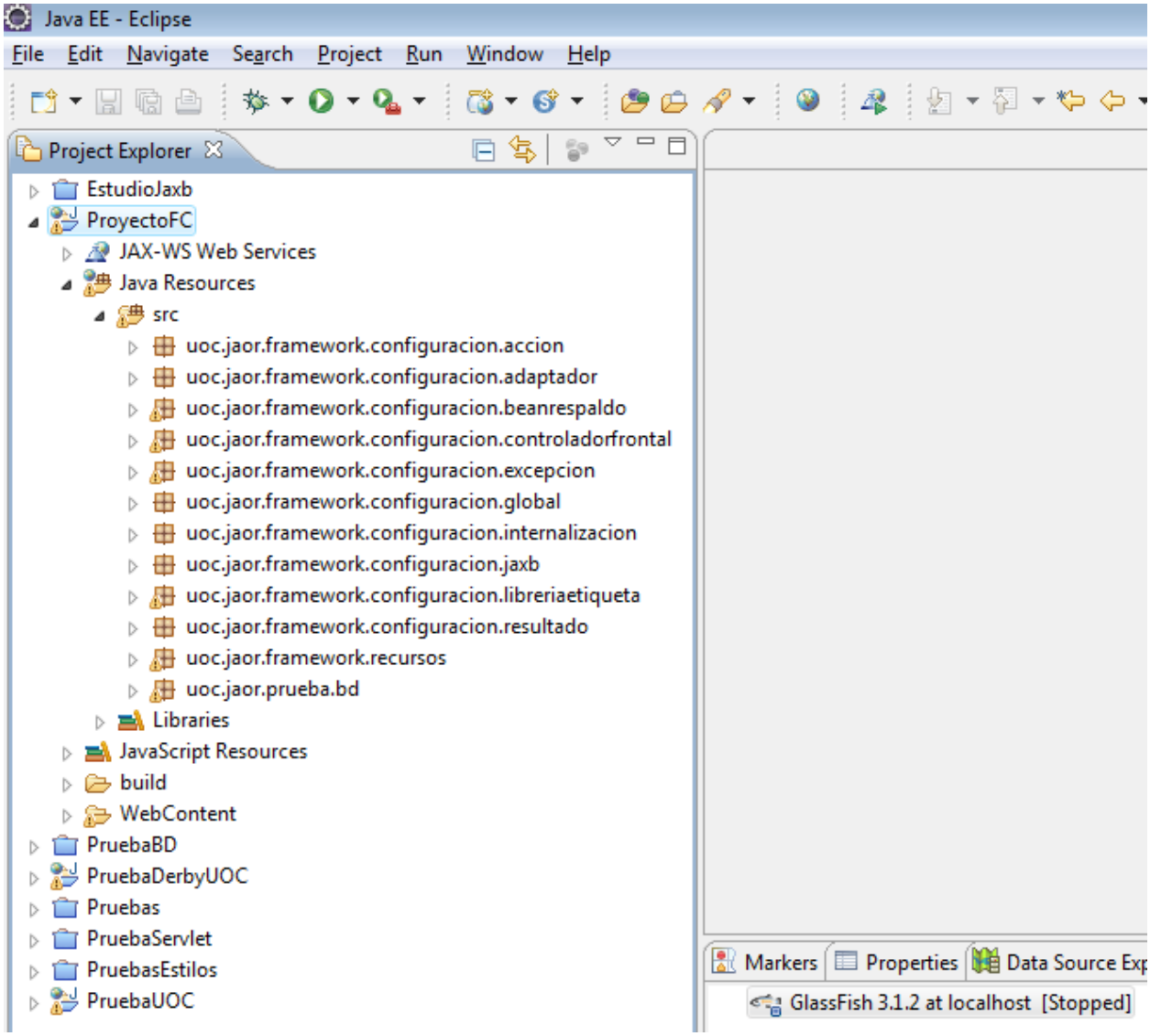
Finalmente con estas transformaciones no se usa el ObjectFactory por lo que se necesita un fichero jaxb.index que lista las clases.
Para que la configuración obtenida con tecnología JAXB la pueda utilizar el framework, estas clases implementan unas interfaces denominadas adaptadoras (no es que implementen el patrón adaptador).

Estas interfaces no hacen más que reflejar la información de cualquier aplicación que gestiona el framework.

10. Estructura del framework.

10.1. Paquetes del framework. Componentes.

Imagen del entorno de trabajo y de la estructura de paquetes del framework:



- En **uoc.jaor.framework.configuracion.accion**, está la interface Acción, la clase RequestContext que implementa el patrón Context Object y la clase Applicaction Controller que implementa el patrón del mismo nombre. Es decir los requisitos de las acciones que utilicen el framework y como se tratarán las acciones cuando el front Controller delega la ejecución.

- En **uoc.jaor.framework.configuracion.adaptador**, las interfaces que implementan en el paquete JAXB comentado en el punto anterior, para tratar la información obtenida de los xml del cliente.

- En **uoc.jaor.framework.configuracion.beanrespaldo**, están las clases para tratar los formularios. La interface formulario que deben cumplir los formularios que usen el framework, la interface filtro que deben cumplir las utilidades sobre los formularios, la clase AtributoError para tratar los errores detectados en los formularios. Además de los filtros ya implementados para la carga y validación de tipos de los atributos, validación de los atributos obligatorios y validación de patrones sobre los atributos. Estos filtros usan las anotaciones Obligatorio y Patrón.

  Definición de la anotación Obligatorio.

```
package uoc.jaor.framework.configuracion.beanrespaldo;
// Anotación para discriminar si los atributos son obligatorios en los beans
// de respaldo. Facilita la tarea cuando se utiliza la api de Reflection.
import java.lang.annotation.*;
// se aplica a los campos
@Target(ElementType.FIELD)
// Valida en tiempo de ejecución
@Retention(RetentionPolicy.RUNTIME)
public @interface Obligatorio {
    boolean loes() default true;
}
```

- En **uoc.jaor.framework.configuracion.controladorfrontal**, el servlet que implementa el patrón del FrontController.

- En **uoc.jaor.framework.configuracion.excepción**, están las excepciones encontradas en la carga de la configuración, la ejecución de las peticiones y carga de recursos. Son errores que una correcta definición del xml del cliente, una correcta programación y definición de recursos deberían evitar.

- En **uoc.jaor.framework.configuracion.global**, está la clase factoría que se encarga de usar la tecnología JAXB usando las clases del paquete uoc.jaor.framework.configuracion.jaxb para generar el árbol de objetos a partir del xml cliente que define la capa de presentación del cliente.

- En **uoc.jaor.framework.configuracion.internalizacion**, está la clase ControladorRecursos que permite la internacionalización. Esta clase busca los mensajes en un fichero properties según el locale definido.

- **uoc.jaor.framework.configuracion.jaxb**, ya está comentado en el punto 9.1.

- En **uoc.jaor.framework.configuracion.libreriaetiqueta**, están las clases que definen las etiquetas de usuario. Se usa la tecnología de JSP custom tags, así las clases heredan de EtiquetaBase que a su vez hereda de TagSupport. Estas etiquetas permiten la internacionalización.

- En **uoc.jaor.framework.configuracion.resultado**, están las clases que implementan los resultados usando la tecnología servlet. Se define la interface TipoResultado que deben cumplir los resultados que usen el framework.

- En **uoc.jaor.framework.configuracion.recursos**, se incluye el esquema de partida usado para definir las clases jaxb y que deben cumplir los ficheros xml de la aplicación cliente. Además de un fichero de mensajes usado por los logger en las pruebas y un fichero de parámetros que usa el framework.

- **NOTA: Se ha incluido uoc.jaor.prueba.bd**, con una clase que está vacía porque la aplicación PruebaUOC que genera los datos de la base de datos en memoria, se carga desde el init del controlador frontal y entonces por requisitos de compilación se ha usado (está sin contenido). Si se utilizará únicamente la aplicación PruebaDerbyUOC que usa una base de datos Derby, no sería necesario incluir esta referencia.

10.2. Web.xml.

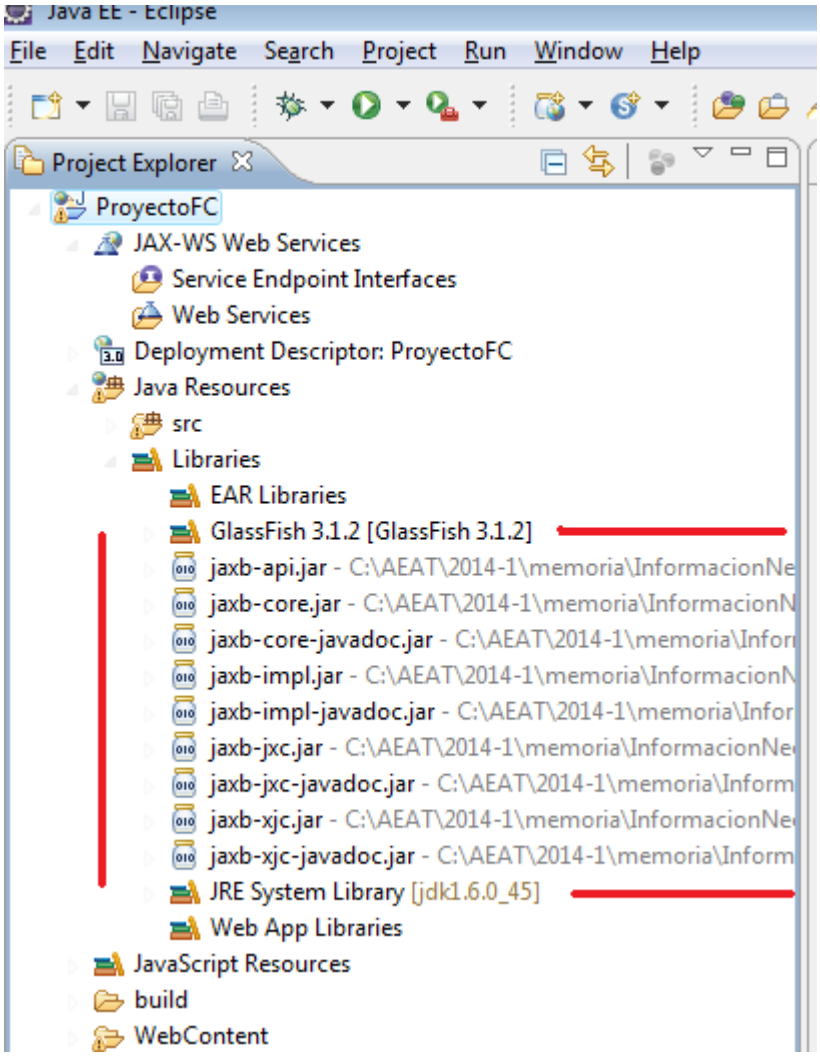
El fichero web.xml que se usa es muy sencillo, los puntos a destacar son la definición del servlet y su clase y el mapping del servlet controlador frontal a las URL que acaben en do (*.do). Como fichero de inicio se ha dejado el que se usa en la aplicación de prueba.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  <display-name>ProyectoFC</display-name>
  <servlet>
    <servlet-name>FrontController</servlet-name>
    <servlet-class>
       uoc.jaor.framework.configuracion.controladorfrontal.FrontController
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <session-config>
            <session-timeout>30</session-timeout>
  </session-config>
      <welcome-file-list>
            <welcome-file>/jsp/login.jsp</welcome-file>
      </welcome-file-list>
</web-app>
```

10.3. Librerías usadas.

- **Librerías de GlassFish 3.1.2** que incorpora Eclipse al seleccionar un Dynamic Web Project. Para hacer funcionar el servidor desde el mismo Eclipse Indigo, se ha usado el plug-in **oepe-repository-indigo-12.1.1.1.1.201209271754.zip,** descargable desde http://www.oracle.com/technetwork/developer-tools/eclipse/downloads/oepe-downloads-1861310.html. Al elegir un proyecto, de tipo Dynamic Web Project teniendo como servidor GlassFish, se agregan estás librerías. No se adjunta en el proyecto porque ocupa 194 M.

- **Librerías para JAXB**. Se han incorporado las librerías de JAXB, tal como se observa en la imagen adjunta. También se observan las librerías GlasshFish 3.1.2. Las librerías de JAXB versión jaxb-ri-2.2.7, bajada de https://jaxb.java.net/2.2.7/. Para agregarlas se agregan desde Java Build Path-> libraries->Add external Jar. Estos jar están en la carpeta lib descomprimida de jaxb-ri-2.2.7.



11. Aplicación de Prueba.

11.1. Introducción. Servidor GlassFish 3, plug-in usado.

Todo el desarrollo se ha realizado usando el servidor GlassFish 3. La versión para Windows, GlassFish 3.1.2. Como herramienta de desarrollo se ha usado Eclipse Java EE IDE for Web Developers, versión Indigo. Esta herramienta facilita el desarrollo mediante un plugin para glassfish .

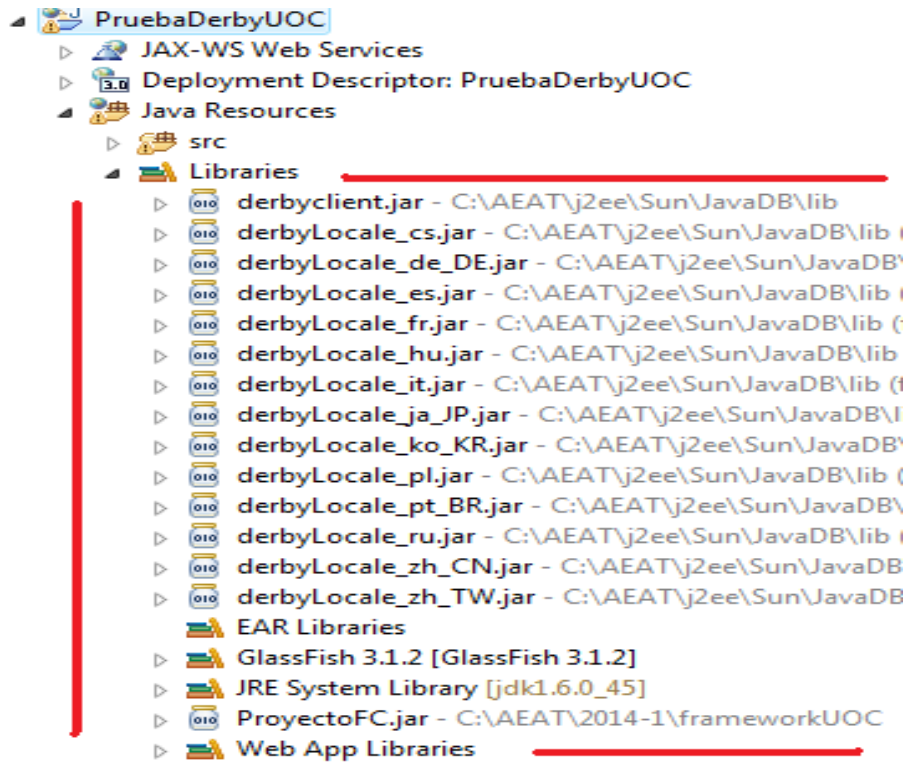
El plug-in usado es **oepe-repository-indigo-12.1.1.1.1.201209271754.zip** que se puede bajar de http://www.oracle.com/technetwork/developer-tools/eclipse/downloads/oepe-downloads-1861310.html. Para agregarlo se puede realizar desde Help->Install new Software.

NO lo adjunto atendiendo al último correo del tutor puesto que acupa 194 M.

11.2 Librerías usadas por la aplicación.

- Naturalmente se ha agregado el jar obtenido denominado ProyectoFC. También se ha agregado en la carpeta WEB-INF\lib.
- En el caso de la aplicación que corre en Derby se ha agregado derbyclient.jar que contiene el driver JDBC para acceder a la base de datos.

Se adjunta una imagen de las librerías de la aplicación PruebaDerbyUOC



11.3 Web.xml

Ya se ha comentado el fichero comentado en el punto 9.

11.4. Aplicaciones PruebaUOC, PruebaDerbyUOC.

Se presentan dos aplicaciones de prueba, pero a efectos del PFC son la misma puesto que lo único que varía es la capa de integración. PruebaUOC se ha diseñado con los datos en memoria y PruebaDerbyUOC con una base de datos en Derby y acceso JDBC pero a efectos de la capa de presentación son iguales y en la capa de negocio, la interface FachadaEJBRemota es igual, sólo varía la implementación FachadaEJB. Se adjunta un fichero de texto para crear la base en Derby.

La aplicación es sobre una librería. Operaciones.

Hay las operaciones de logón, listar todos los libros, listar autores, buscar los libros de un determinado autor, buscar autor por nombre y agregar libro. Atención, agregar libro no realiza un enlace con los autores, no se ha hecho por cuestión de tiempo.

Nota, la licencia Magic Draw 17.0 expiró y no hubo manera de renovarla. Estos dos diagramas se han realizado con Microsoft Visio 2007 que ofrece gratuitamente la UOC:

Diagrama de casos de uso.

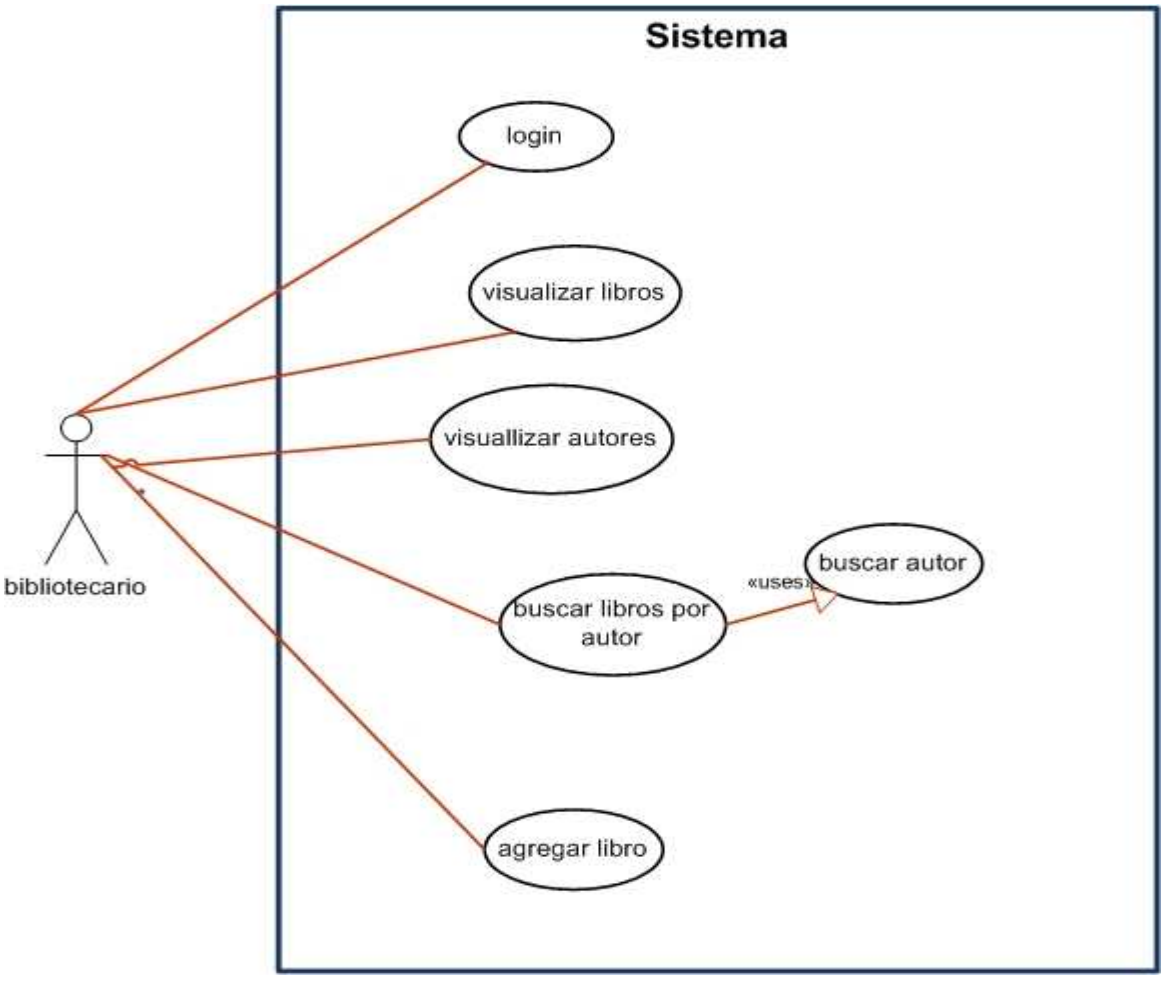
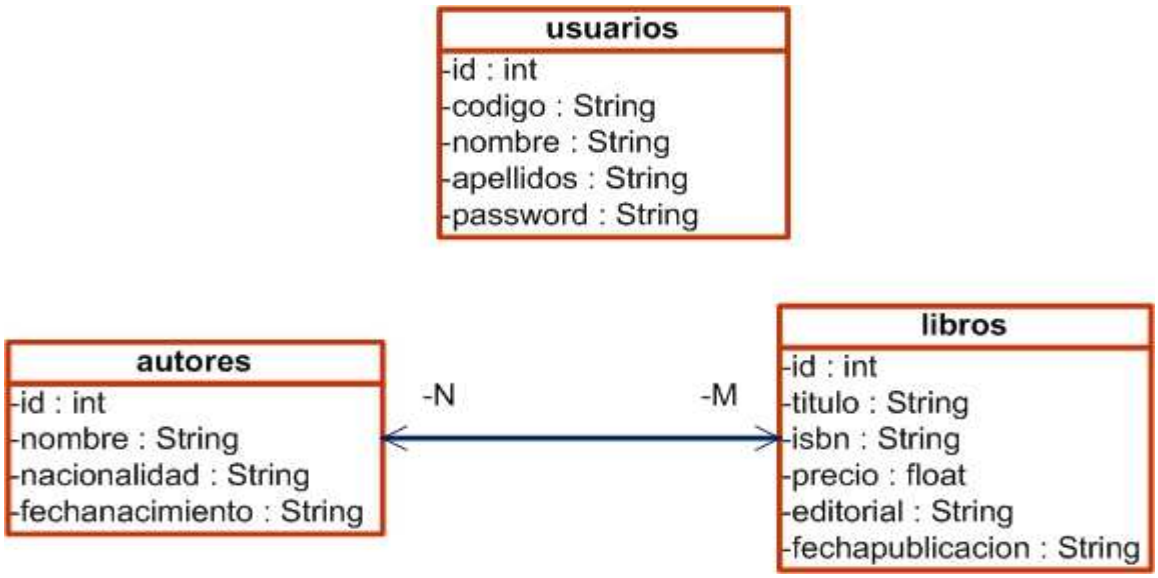


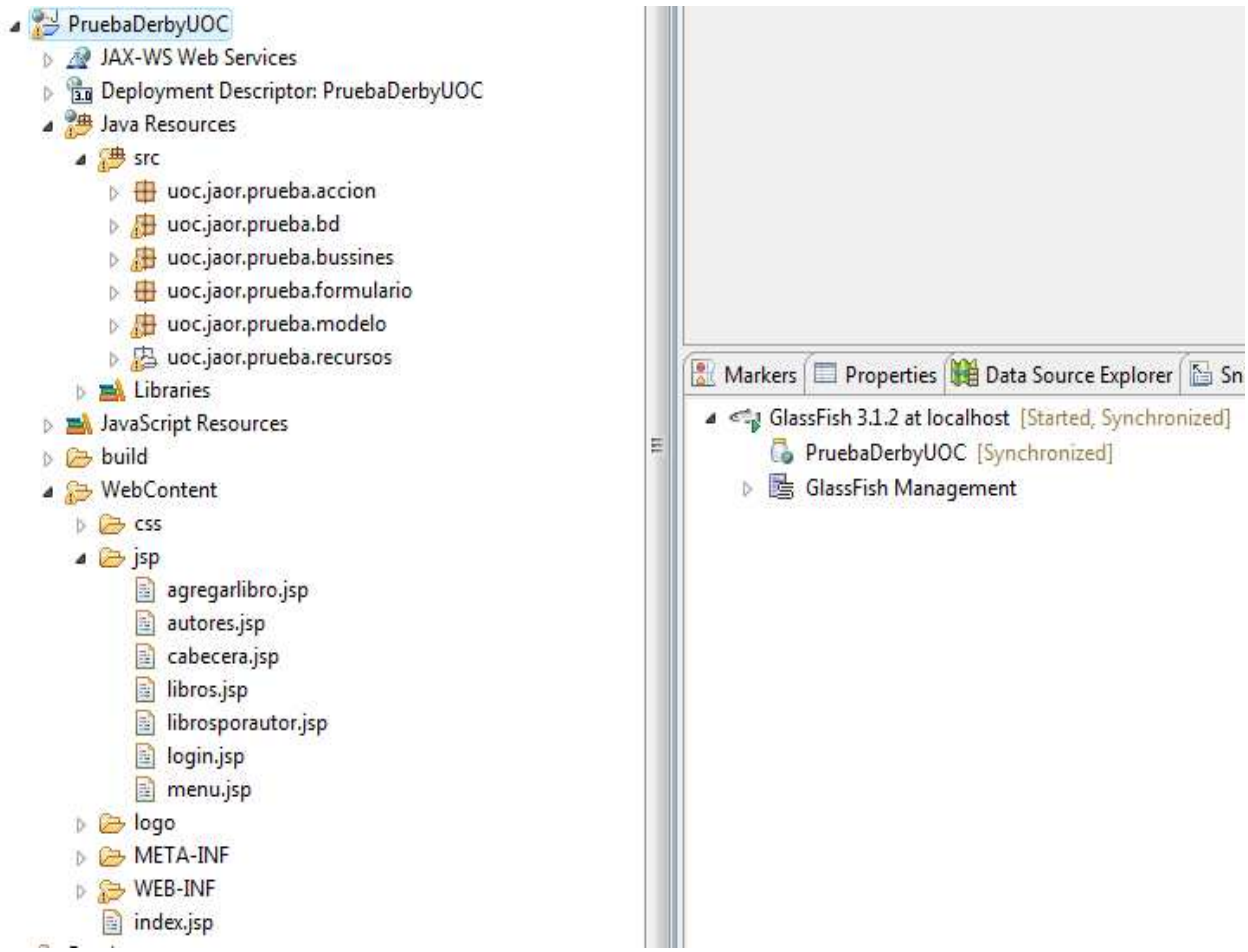
Diagrama de clases.



Hay una asociación M:N entre autores y libros.

11.4.1 Estructura de paquetes de PruebaDerbyUOC.

Creo que los nombres son bastante aclaratorios pero hago una breve explicación:



- En **uoc.jaor.prueba.acción** se definen las acciones de la aplicación. Los nombres son bastante ilustrativos. Está la clase AcciónBase que implementa la interface Accion y de la que heredan el resto de las acciones.

- En **uoc.jaor.prueba.bd**, está la clase BD con los métodos necesarios para realizar una conexión JDBC con Derby.

- **uoc.jaor.prueba.bussines,** es la capa de negocio, está la clase FachadaEJB que implementa la interface FachadaEJBRemota. Se ha aplicado el patrón Fachada dando un único punto de acceso a la capa de negocio.

  Patrón Fachada.

```
public interface FachadaEJBRemota {

   public Usuario buscarUsuarioPorCodigo(String  codigoBuscado);
   public List<Usuario> buscarUsuarios();
   public String logon(String codigoBuscado, String password);
   public void listar();

   public ArrayList<Libro> buscarLibros();

 public ArrayList<Autor> buscarAutores();
  public ArrayList<Libro> buscarLibrosPorAutor(Autor a);
  public Autor buscarAutorPorNombre(String nombreAutor);
  public void agregarLibro(Libro l);
}
```

  Nota, listar() solo se ha usado para pruebas.

- En **uoc.jaor.prueba.formulario,** se definen los formularios de la aplicación. Todos los formularios heredan de FormularioBase que implementa Formulario. FormularioEntrada se usa para el login, FormularioLibrosDeAutor se usa para la búsqueda de libros por nombre de autor y FormularioLibro para la acción de añadir un libro, AccionAgregarLibro. Se observa el uso de las anotaciones definidas.

- **uoc.jaor.prueba.modelo,** correspondería a la capa de integración se definen las "Entidades": Usuario, Autor Y Libro.

- En **uoc.jaor.prueba.recursos,** se incluyen los ficheros properties, **librería** para los mensajes y títulos de las páginas jsp como texto de las etiquetas, botones…. y en **mensajesAplicacion** se incluyen los mensajes de las acciones de acierto y error. Además se define el fichero **EsquemaCliente.xml** que define la estructura de la capa aplicación.

11.4.2 Comandos usados en Derby.

En la entrega se adjunta un fichero bd.txt con los comandos usados para crear las tablas e insertar valores.

11.5. Fichero EsquemaCliente de la aplicación librería.

Con este fichero se define la estructura de la capa de presentación de la aplicación cliente, incluyendo la navegación con los resultados.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuracion xmlns="http://framework.uoc.es/configuracion" >

<BeansRespaldo>
       <Formulario nombre="FormularioEntrada"
              clase="uoc.jaor.prueba.formulario.FormularioEntrada" >
              <Filtros>
                  <Filtro nombre="CargaAtributos"
                  clase="uoc.jaor.framework.configuracion.beanrespaldo.CargaAtributos"
                  siguiente="AtributosObligatorios"  />


           <Filtro nombre="AtributosObligatorios"
           clase="uoc.jaor.framework.configuracion.beanrespaldo.AtributosObligatorios"
            siguiente="PatronAtributos"  />
          <Filtro nombre="PatronAtributos"
           clase="uoc.jaor.framework.configuracion.beanrespaldo.PatronAtributos"
            siguiente=""  />
       </Filtros>
     </Formulario>

     <Formulario nombre="FormularioLibrosDeAutor"
         clase="uoc.jaor.prueba.formulario.FormularioLibrosDeAutor">
        <Filtros>
           <Filtro nombre="CargaAtributos"
            clase="uoc.jaor.framework.configuracion.beanrespaldo.CargaAtributos"
           siguiente=" AtributosObligatorios "  />
           <Filtro nombre="AtributosObligatorios"
          clase="uoc.jaor.framework.configuracion.beanrespaldo.AtributosObligatorios"
           siguiente=""  />
     </Filtros>
     </Formulario>

      <Formulario nombre="FormularioLibro"
           clase="uoc.jaor.prueba.formulario.FormularioLibro">
           <Filtros>
                <Filtro nombre="CargaAtributos"
                clase="uoc.jaor.framework.configuracion.beanrespaldo.CargaAtributos"
                siguiente="AtributosObligatorios"  />
               <Filtro nombre="AtributosObligatorios"
         clase="uoc.jaor.framework.configuracion.beanrespaldo.AtributosObligatorios"
               siguiente="PatronAtributos"  />
               <Filtro nombre="PatronAtributos"
               clase="uoc.jaor.framework.configuracion.beanrespaldo.PatronAtributos"
               siguiente=""  />
          </Filtros>
       </Formulario>
</BeansRespaldo>
```

```
<Acciones>
   <Accion nombre="/AccionLogin.do" clase="uoc.jaor.prueba.accion.AccionLogin"
                   formulario="FormularioEntrada">
        <Resultados>
             <Resultado nombre="ok"
             clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
             destino= "/jsp/menu.jsp">
             </Resultado>
             <Resultado nombre="error"
             clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
             destino="/jsp/login.jsp">
             </Resultado>
       </Resultados>
   </Accion>

<Accion nombre="/AccionListarAutores.do"
        clase="uoc.jaor.prueba.accion.AccionListarAutores">
        <Resultados>
             <Resultado nombre="ok"
              clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
              destino="/jsp/autores.jsp" >
             </Resultado>
             <Resultado nombre="error"
              clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
              destino="/jsp/menu.jsp" >
             </Resultado>
                   </Resultados>
</Accion>
<Accion nombre="/AccionListarLibros.do"
           clase="uoc.jaor.prueba.accion.AccionListarLibros">
            <Resultados>
                 <Resultado nombre="ok"
                  clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
                  destino="/jsp/libros.jsp" >
                  </Resultado>
                  <Resultado nombre="error"
                  clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
                  destino="/jsp/menu.jsp" >
                  </Resultado>
             </Resultados>
</Accion>
<Accion nombre="/AccionAgregarLibro.do"
           clase="uoc.jaor.prueba.accion.AccionAgregarLibro"
           formulario="FormularioLibro">
           <Resultados>
                <Resultado nombre="ok"
                 clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
                 destino="/jsp/menu.jsp" >
                </Resultado>
                <Resultado nombre="error"
                 clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
                 destino="/jsp/agregarlibro.jsp" >
                </Resultado>
         </Resultados>
</Accion>

<Accion nombre="/AccionListarLibrosPorAutor.do"
          clase="uoc.jaor.prueba.accion.AccionListarLibrosPorAutor"
          formulario="FormularioLibrosDeAutor">
```

```
          <Resultados>
               <Resultado nombre="ok"
               clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
               destino="/jsp/librosporautor.jsp" >
               </Resultado>
               <Resultado nombre="error"
               clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
               destino="/jsp/menu.jsp" >
               </Resultado>
           </Resultados>
</Accion>
<Accion nombre="/AccionIr.do" clase="uoc.jaor.prueba.accion.AccionIr">
        <Resultados>
            <Resultado nombre="ok"
             clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
             destino="/jsp/menu.jsp" >
            </Resultado>
        </Resultados>
</Accion>

<Accion nombre="/AccionIrAgregarLibro.do"
           clase="uoc.jaor.prueba.accion.AccionIrAgregarLibro">
          <Resultados>
               <Resultado nombre="ok"
               clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
               destino="/jsp/agregarlibro.jsp" >
              </Resultado>
         </Resultados>
</Accion>

<Accion nombre="/AccionLogout.do" clase="uoc.jaor.prueba.accion.AccionLogout">
          <Resultados>
              <Resultado nombre="ok"
               clase="uoc.jaor.framework.configuracion.resultado.ResultadoForward"
               destino="/jsp/login.jsp" >
              </Resultado>
          </Resultados>
</Accion>
</Acciones>
</Configuracion>
```

Se puede observar que se han definido los formularios y las acciones.

En cualquier formulario, por ejemplo **FormularioEntrada** , aparte de su clase, se ha definido la cadena de filtros que se le aplican. Primero obligatoriamente **CargaAtributos**, después **AtributosObligatorios** y finalmente **PatronAtributos**. Para cada filtro se define también su clase y el siguiente filtro a aplicar.

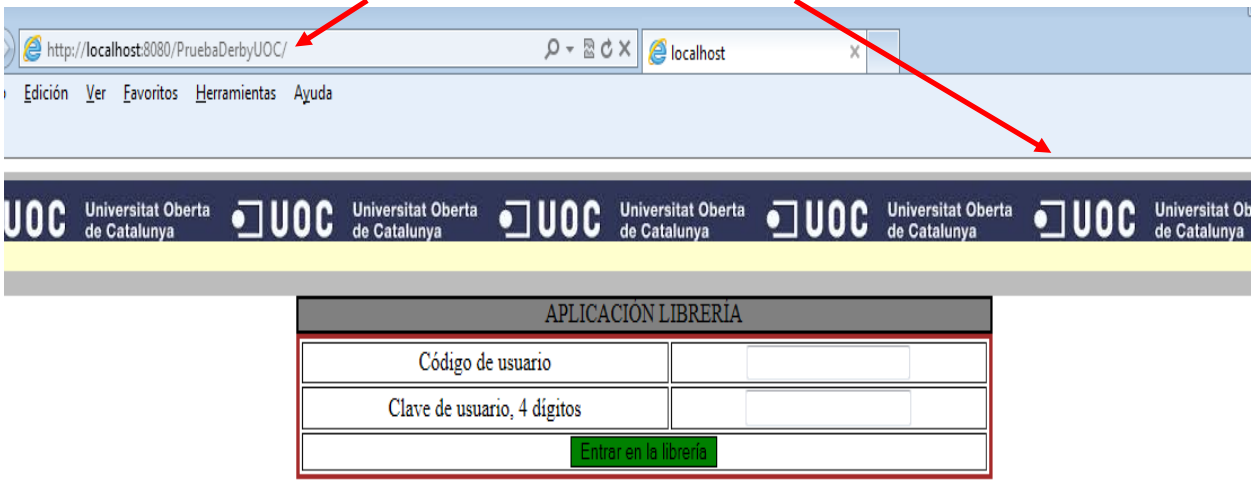
Para las acciones, por ejemplo **AccionLogin**, se define su clase, el formulario que lleva asociado, en este caso el **FormularioEntrada** y la lista de resultados asociados. En este caso dos, acierto "**ok**" y "**error**". El resultado indica el nombre de la clase y el recurso objetivo. Por ejemplo en caso de acierto, se aplicará un forward a la página **menu.jsp**. Hay acciones muy sencillas que no tienen formulario y solo tienen asociado un resultado como la acciónAccionIr.

11.6. Pantallas de ejecución de la aplicación librería.

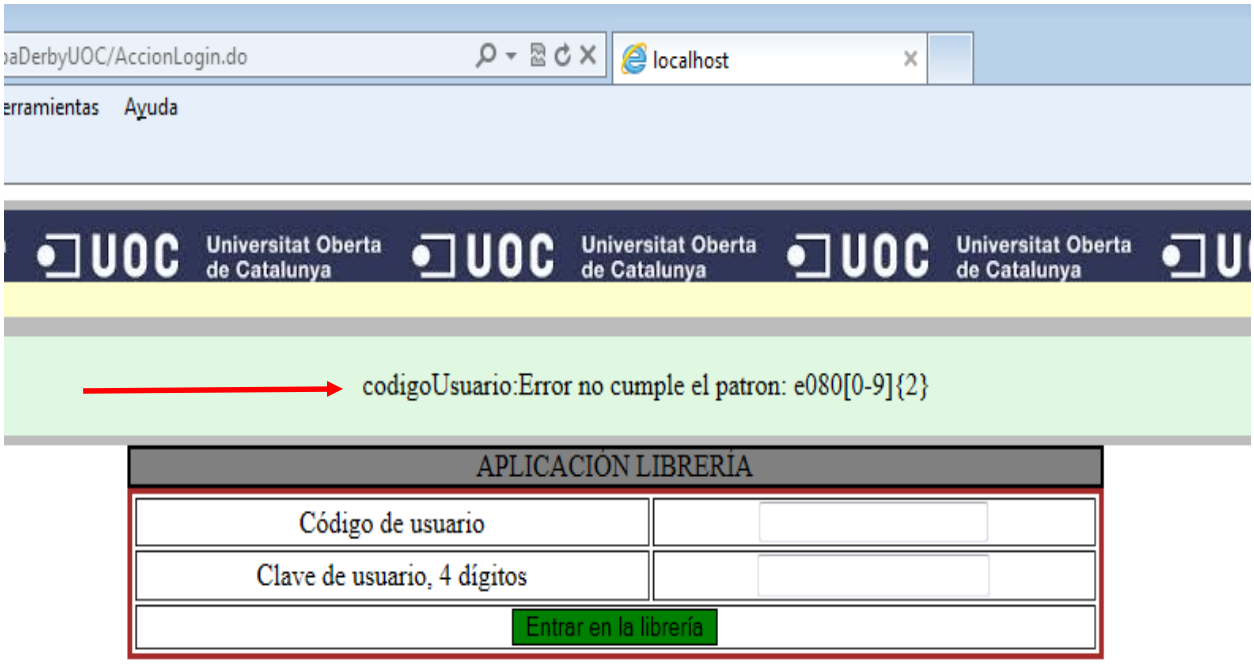
Se ha ejecutado en el servidor GlassFish 3 generando los war que también se adjunta, aparte de las pruebas desde el mismo eclipse.

Imagen de inicio. No cabe en el Word pero el logo de la UOC se repite a lo largo de la pantalla del explorador.

Aplicación PruebaDerbyUOC **Logo se repite**

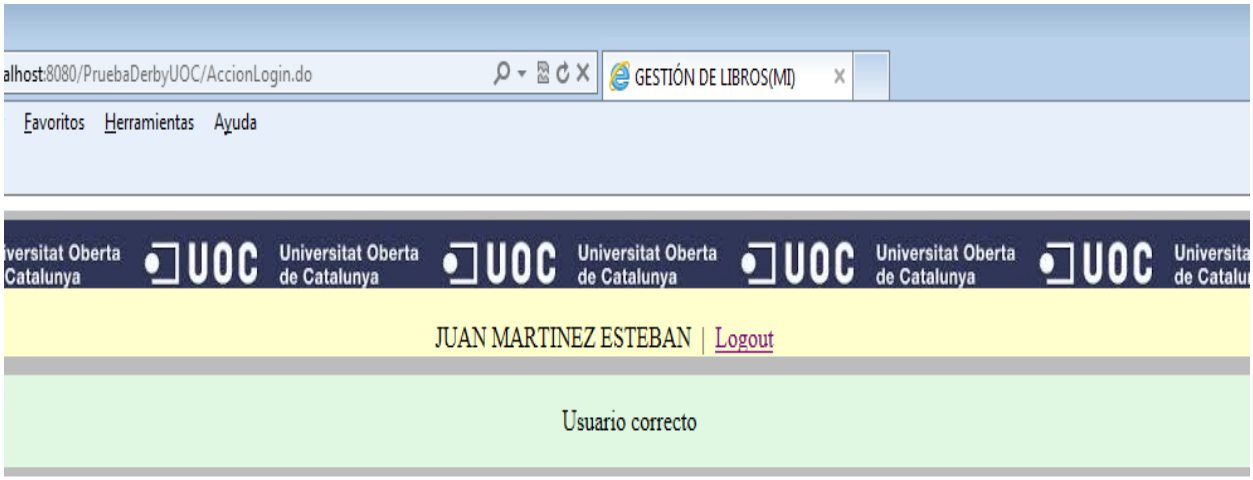


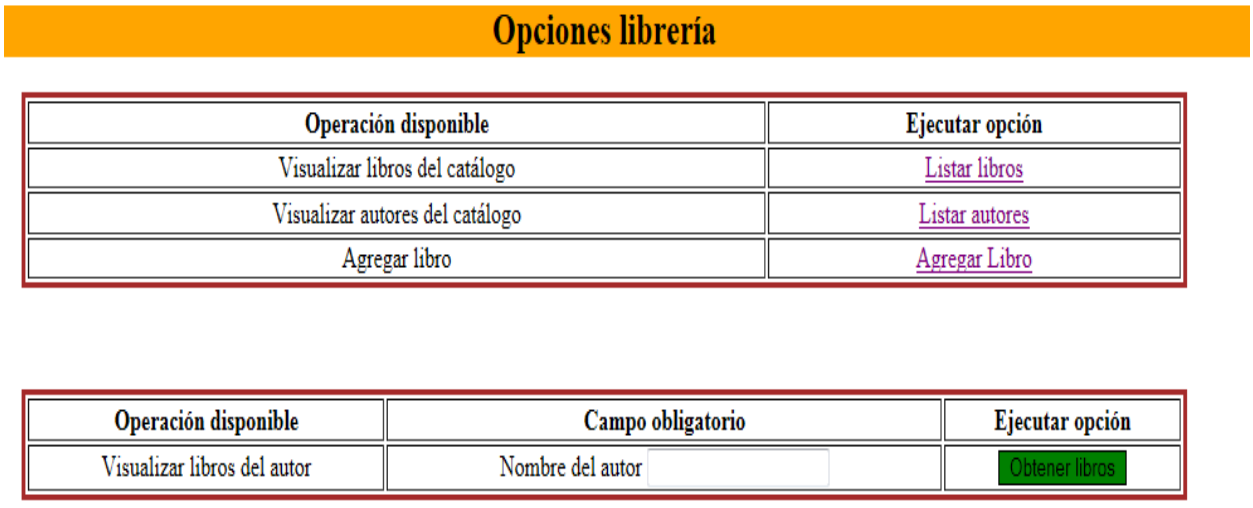
Los usuarios tienen un código del tipo e080XX, siendo X un dígito. En caso de que introduzca un código que no cumpla el patrón da error, ejemplo de uso del Filtro PatronObligatorio.



Error de password erróneo:

Usuario e08001, password 1001, imagen del menú principal:





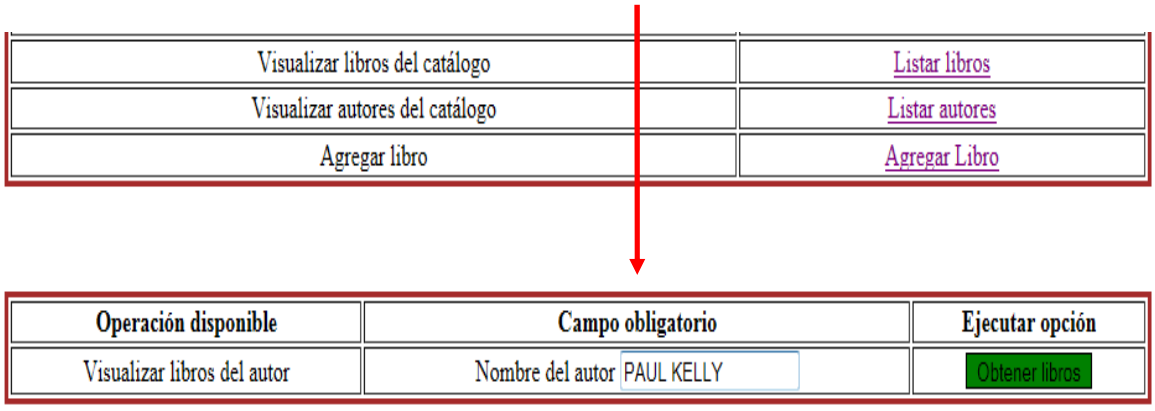
Opción listar autores. Con la opción volver al menú.

Catálogo de autores

Nombre	Fecha de nacimiento	Nacionalidad
PAUL KELLY	10/12/1962	INGLESA
JANE HUDSON	05/06/1982	INGLESA
MIKE STONE	24/03/1980	INGLESA
CRAIG WALLS	25/12/1985	USA
DONALD BROWN	14/02/1989	USA
CHAD MICHAEL DAVIS	28/02/1981	USA
SCOTT STALINCK	06/09/1987	USA
WILLIAM STANECK	12/12/1988	USA
CRAIG LARMAN	04/07/1978	USA
MARTIN FOWLER	10/10/1972	USA

Volver al menú

Opción Listarlibros por autor. Se introduce el autor PAUL KELLY.



Se pulsa obtener libros.

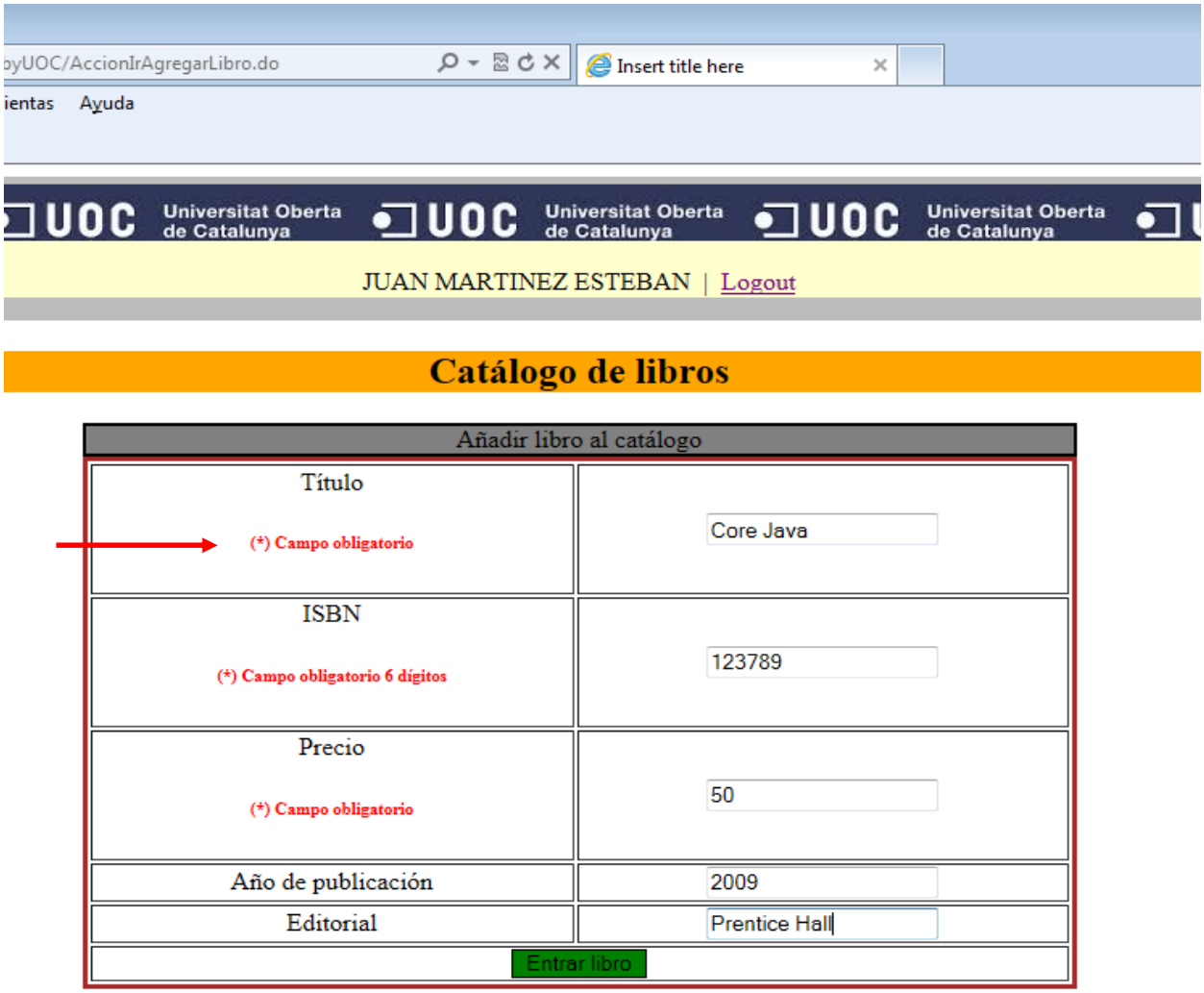
Autor

Nombre	Fecha de nacimiento	Nacionalidad
PAUL KELLY	10/12/1962	INGLESA

Obras escritas

Título	ISBN	Editorial	Precio	Año de publicación
Oxford English	121212	University Press	15.0	2009
English Grammar	214156	UNIVERSITY PRESS	20.0	2007
English Dictionary	711123	University Press	62.0	1988

Opción agregar libro. Nota, se agrega un libro pero no se enlaza con los autores. Se insertan los datos título "Core Java", isbn "123789", precio "50", editorial "Prentice Hall" y año "2009". Observar que hay campos obligatorios.



Se agrega el libro, se vuelve al menú y se visualiza el mensaje "Agregado libro, ver listar libros".

Se visualizan los libros y se observa que se ha agregado (faltan los autores tal como he comentado):

UML Y Patrones	787896	Prentice Hall	25.0	2004	CRAIG LARMAN
UML 2	457896	Prentice Hall	30.0	2006	DONALD BROWN WILLIAM STANECK CRAIG LARMAN
English Dictionary	711123	University Press	62.0	1988	PAUL KELLY MIKE STONE
Analysis Patterns	556688	Prentice Hall	55.0	2000	MARTIN FOWLER
Windows 8	365412	Microsoft Press	68.0	2012	CRAIG WALLS WILLIAM STANECK
Métrica	831794	Anaya	68.0	2006	
Core Java	123789	Prentice Hall	50.0	2009	

En **el caso de que falte un campo obligatorio da error, por ejemplo falta el campo isbn:**

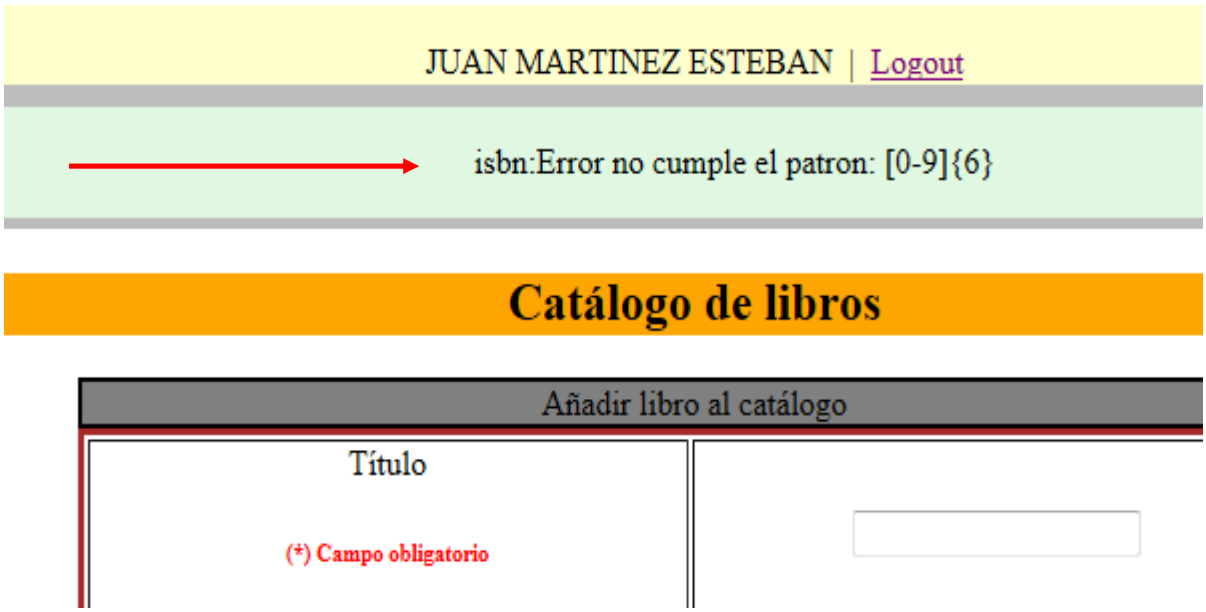
Catálogo de libros

Añadir libro al catálogo	
Título (*) Campo obligatorio	EL LIBRO DEL AGUA
ISBN (*) Campo obligatorio 6 dígitos	
Precio (*) Campo obligatorio	15
Año de publicación	
Editorial	
Entrar libro	

Se pulsa entrar y aparece el mensaje de "error campo vacio":

También se aplica la anotación Patrón sobre isbn que deben ser seis números, en caso de que no se cumpla el patrón, también se indica.

Se observa el mensaje de que no cumple el patrón:



12. Conclusiones.

Respecto a los frameworks estudiados, el que más me ha gustado es Java Server Faces, aunque no trabajo desarrollando aplicaciones web, el entorno Eclipse con GlassFish 3 me ha gustado mucho y supongo que ha influido en mi elección, creo que posee muy buenas características para desarrollar aplicaciones J2EE complejas. Sin embargo a efectos de desarrollar mi propio framework el modelo escogido ha sido Struts. Ha sido una elección por la claridad del modelo que presenta. Orientado a Acciones, Struts 2 , presenta una implementación de los patrones estudiados en la bibliografía mucho más clara.

El framework posee características, como separación de la capa de presentación de la de negocio, citar que al realizar las dos aplicaciones PruebaUOC y PruebaDerbyUOC , en el caso de no aplicar el patrón MCV y mezclar capas hubiera obligado a rediseñar la prueba PruebaDerbyUOC cuando la diferencia real es en la capa de integración. Ni siquiera se ha variado la interface del patrón Fachada de la capa de negocio.
Aplica el patrón Front Controller proporcionando un punto centralizado de control. Permite navegación parametrizada a través del elemento resultado, realiza una carga y validación de los atributos más sencillos, en este punto se ha querido usar cadenas de clases, los Filtros, que sean reutilizables, permite internacionalización y posee una librería de etiquetas.

El desarrollo del framework me ha permitido estudiar tecnología java que desconocía como JAXB, estudiar manuales respecto a los patrones de la capa de presentación como Core J2EE Patterns, Best Practices and Design Strategies que considero fundamentales, trabajar sobre un entorno de desarrollo

como GlassFish 3, estudiando libros como Beginning Java EE 6 Platform with GlasshFish, escrito por un investigador que ha colaborado en las JSR de JEE 6, así como estudiar algunas de las herramientas de la capa de presentación más utilizadas... En definitiva creo que me ha enriquecido mucho, por eso quiero agradecer a mi tutor Josep Maria Camps, la posibilidad de trabajar en este campo.

13. Fuentes.

13.1 Libros.

- Core J2EE Patterns, Best Practices and Design Strategies, 2ª Edicion, Deepack Alur, John Crupi, Dan Malks, Editorial Sun Microsystems Press Prentice Hall.
- UML y Patrones, Segunda Edición, Craig Larman, Editorial Prentice Hall.
- Beginning Java EE 6 Platform with GlasshFish 3. Second Edition. Antonio Goncalves. Editorial Apress.
- STRUTS 2, Donald brown, Chad Michael Davis, Scott Stanlick. Editorial Anaya.
- Core Java Server Faces, Third Edition, David Geary, Cay Horstmann, Editorial Prentice Hall.
- Spring, 3ª edición, Craig Walls, Editorial Anaya.
- Desarrollo de Aplicaciones web con JEE 6. Thierry Groussard. Ediciones ENI. Sobre API servlets y JSP.

13.2 Documentos y Tutoriales.

- Tutoriales de SUN (disponibles también en The Java Tutorial, Fifth Edition) sobre JAXB, la API de Reflection , Generics , anotaciones e Internacionalización. Bajado de la página de Oracle, http://docs.oracle.com/javase/tutorial/.
- Documentación y guías sobre GlassFish bajados de la página de Oracle.
- Using JAXB 2.0's XmlJavaTypeAdapter, https://weblogs.java.net/blog/kohsuke/archive/2005/09/using_jaxb_20s.html
- Java Server Faces 2 , Introduction and Overview, Mary Hall. Establece comparaciones con otros frameworks.

- Tutorial http://websphere.sys-con.com/node/46516/print 25/ establece comparación entre J.S.F. y Struts.
- Tutorial sobre Struts, http://joeljil.wordpress.com/2010/05/31/struts2/ que su vez está tomado de la documentación oficial de Apache.
- Documentación de Apache sobre Struts 2.
- Tutorial sobre Struts 2. http://www.javatutoriales.com/2011/06/struts-2-parte-1-configuracion.html.
- Documento de la Universidad de Málaga sobre Java Server Faces 2.
- Tutorial sobre hojas de estilo, http://es.html.net/tutorials/css/lesson2.php.
- Tutorial sobre JSP Custom Tags, http://www.programacion.com/articulo/jsp_custom_tags_etiquetas_a_medida_133.
- Tutorial sobre JSP, http://www.tutorialspoint.com/jsp/jstl_core_out_tag.htm.
- Tutorial sobre internacionalización de programas java: http://www.programacion.com/articulo/internacionalizacion_de_programas_java_140.